Report Number: WUCS-96-10

1996-01-01

Leap Forward Virtual Clock: An O(loglogN) Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness

Subhash Suri, George Varghese, and Girish P. Chandranmenon

We describe an efficient fair queuing scheme, Leap Forward Virtual Clock, that provides end-to-end delay bounds almost identical to that of PGPS fair queuing, along with throughput fairness. Our scheme can be implemented with a worst-case time O(loglogN) per packet guaranteed delay and throughput fairness. As its name suggests, our scheme is based on Zhang's virtual clock. While the original virtual clock scheme does not achieve throughput fairness, we can modify it with a simple leap forward mechanism that keeps the server clock from lagging too far behind the packet tags. We prove that our scheme guarantees a fair... **Read complete abstract on page 2.**

Leap Forward Virtual Clock: An O(loglogN) Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness

Subhash Suri, George Varghese, and Girish P. Chandranmenon

Complete Abstract:

We describe an efficient fair queuing scheme, Leap Forward Virtual Clock, that provides end-to-end delay bounds almost identical to that of PGPS fair queuing, along with throughput fairness. Our scheme can be implemented with a worst-case time O(loglogN) per packet guaranteed delay and throughput fairness. As its name suggests, our scheme is based on Zhang's virtual clock. While the original virtual clock scheme does not achieve throughput fairness, we can modify it with a simple leap forward mechanism that keeps the server clock from lagging too far behind the packet tags. We prove that our scheme guarantees a fair share of the available bandwidth to each of the backlogged users, while precisely matching the delay bounds of PGPS schemes. In order to improve computational efficiency, we introduce a "coarsened" version of our scheme in which all tags assume values from a set of O(N) integers. We then use "approximate sorting" and a finite-universe priority queue to achieve O(loglogN) processing time per packet. We can show that the coarsening of tags increases the delay bound by a very small additive constant. Finally, our proofs are based on a dual version of the algorithm called Leap Backward, whose behavior is identical to the Leap Forward but that admits a simpler analysis.

Leap Forward Virtual Clock:

An $O(\log \log N)$ Fair Queuing Scheme with

Guaranteed Delays and Throughput Fairness

Subhash Suri
George Varghese
Girish P. Chandranmenon

WUCS-96-10

April 15, 1996

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

We describe an efficient fair queuing scheme, *Leap Forward Virtual Clock*, that provides end-to-end delay bounds almost identical to that of PGPS fair queuing, along with throughput fairness. Our scheme can be implemented with a worst-case time $O(\log \log N)$ per packet (inclusive of sorting costs), which improves upon all previously known schemes that achieve guaranteed delay and throughput fairness. As its name suggests, our scheme is based on Zhang's virtual clock. While the original virtual clock scheme *does not* achieve throughput fairness, we can modify it with a simple *leap forward* mechanism that keeps the server clock from lagging too far behind the packet tags. We prove that our scheme guarantees a fair share of the available bandwidth to each of the backlogged users, while precisely matching the delay bounds of PGPS schemes. In order to improve computational efficiency, we introduce a "coarsened" version of our scheme in which all tags assume values from a set of $O(N)$ integers. We then use "approximate sorting" and a finite-universe priority queue to achieve $O(\log \log N)$ processing time per packet. We can show that the coarsening of tags increases the delay bound by a very small additive constant. Finally, our proofs are based on a dual version of the algorithm called Leap Backward, whose behavior is identical to the Leap Forward but that admits a simpler analysis.

Leap Forward Virtual Clock:

An $O(\log \log N)$ Fair Queuing Scheme with

Guaranteed Delays and Throughput Fairness

Subhash Suri
suri@cs.wustl.edu
+1 314 935 7546

George Varghese
varghese@askew.wustl.edu
+1 314 935 4963

Girish P. Chandranmenon
girish@cs.wustl.edu
+1 314 935 4163

1. Introduction

Due to the popularity of the World Wide Web and teleconferencing tools such as *wb*, *vat* and *vic*, traffic on the Internet is growing quite rapidly. Future applications like electronic commerce, video-on-demand, remote medical diagnosis, and multiparty games are likely to strain the capabilities of the Internet even further. While the traffic volume is clearly growing in quantity, it is also undergoing a fundamental *qualitative* change, in large part due to the increase in audio and video data. A fundamental distinction between an audio/video application and more traditional applications like file transfer is that the former usually demands a *quality-of-service* (QoS) guarantee, which the latter does not. Specifically, unless a large fraction of the audio or video packets are received within a bounded delay, the received signal fails to meet perceptual standards acceptable to most users. This latency requirement introduces a new dimension for traditional data networks like the Internet.

As the Internet expands to include a variety of commercial and private users, another form of quality of service, called *traffic isolation*, becomes important; this is desirable even for traditional data applications. In the new milieu, we need firewalls between contending users, so that well-behaved users can be insulated from ill-behaved ones. Today's Internet does not contain such firewalls. A rogue source that sends at an uncontrolled rate can seize a large fraction of the buffers at an intermediate router and harm well-behaved users. A solution to this problem is needed to *isolate* the effects of bad behavior to users that are behaving badly.

If cost were no concern, then simple-minded solutions could be used. For instance, the Internet could be *over-engineered* using the fastest links to guarantee QoS for latency-critical applications, and routers could be equipped with special-purpose hardware to run traffic isolation algorithms at Gigabit speeds. Such an Internet will be underutilized most of the time, and expensive to build and maintain.

However, it is clear that cost *is* a major issue. As commercial applications begin to use the Internet, funding for the Internet will shift from government subsidies to revenue obtained from customers. In this new setting, market forces will require cost-effective solutions. Even today, with limited and shrinking government subsidies, it is important to consider cheap solutions that can be widely deployed using existing links and existing hardware platforms.

There is a remarkable scheme called Weighted Fair Queuing (WFQ) [1] that has been regarded as a central mechanism for the new Integrated Services Internet. WFQ provides guaranteed delay bounds, fair throughput sharing, as well as traffic isolation. Unfortunately, the computational overhead for implementing this scheme is rather large—it requires $O(N)$ computation per packet, where N is the number of conversations (flows) using a link. Other more efficient variants [9], fail to match the delay bounds provided by WFQ. Very recently, an algorithm has been proposed [10] whose service guarantee is almost as good as that of WFQ, but the new algorithm still requires $O(\log N)$ time per packet.

While algorithms requiring $O(\log N)$ time per packet are a great improvement over the $O(N)$ time schemes, logarithmic computational overhead remains a considerable cost for routers that run at Gigabit speeds. Some papers on fair queuing, such as [5, 10], suggest using special-purpose hardware at routers to overcome the $\Theta(\log N)$ sorting bottleneck.[1] However, the use of special-purpose hardware drives up the cost of routers. Also, it seems better to implement router scheduling algorithms in software for the following reasons: (1) given the rate at which the computing power has been increasing for the last decade, by the time special-purpose hardware is designed and deployed, a general purpose CPU may offer similar or even better performance, and (2) the algorithms and protocols for fair queuing are likely to evolve; software upgrades are cheaper and easier than retrofitting hardware.

In this paper, we describe a fair queuing scheme called *Leap Forward Virtual Clock* that almost matches the delay, throughput fairness, and traffic isolation performance of Weighted Fair Queuing, but requires only $O(\log\log N)$ time per packet to implement. This makes our scheme a viable candidate for efficient software implementation in Internet routers. The $O(\log\log N)$ bound is a small constant for all practical purposes—for instance, $\log\log N \leq 5$ for all $N \leq 4 * 10^9$; the constants buried under the order of complexity notation are also reasonable. Note that our computational cost *includes all sorting overheads*. This contrasts with several existing algorithms [5, 10], whose $O(1)$ time complexity bound accounts only for "tag computation;" these methods still incur an additional cost of $O(\log N)$ per packet to insert each tag in a priority queue. Thus, our algorithm appears to be the first to break the $\Omega(\log N)$ sorting bottleneck, while retaining all the desirable properties of WFQ, namely, throughput fairness and delay bounds.

Our paper also shows that the *Virtual Clock* scheme of Zhang [12] can be modified to achieve delay and throughput properties comparable to WFQ. Previous papers have shown that Virtual Clock provided identical delay bounds to PGPS [11, 3], but did not provide fair throughput sharing [5, 10]. We present an intuitive "Leap Forward" modification to Virtual Clock that preserves throughput fairness. By contrast, many previous approaches to providing efficient fair queuing have been based on modifications to WFQ (e.g., [10]).[2]

The rest of the paper is organized as follows. In Section 2 we review previous fair queuing schemes. In Section 3, we introduce our main algorithm, Leap Forward Virtual Clock. In Section 4, we introduce a dual version of this algorithm, called Leap Backward Virtual Clock, whose extenal behavior is identical to Leap Forward but lends itself to an easier analysis. We prove our end-to-end delay bound theorem in Section 5, and throughput fairness theorem in Section 6. Finally, in Section 7, we describe our tag coarsening technique and the $O(\log\log N)$ time implementation.

[1] Indeed, by using special-purpose hardware, it appears possible to implement WFQ in $O(1)$ time.

[2] We have not been able to determine the details of the Frame Based Frame Queuing scheme because the available Technical Report [10] only refers to the scheme without providing details.

2. Previous Work

FIFO is perhaps the simplest possible scheduling algorithm, but it falls well short of providing throughput fairness. On the other end of the spectrum lies idealized round-robin (or GPS), which provides ideal latency and fairness, but is computationally impractical. There are a number of known scheduling algorithms that have properties similar to GPS at a more reasonable cost. We only review *work-conserving* scheduling algorithms, since they lead to better delay bounds than non-work-conserving algorithms. (A scheduling algorithm is work-conserving if the server is never idle as long as there is a packet in an input queue.) Non-work-conserving disciplines, like the one proposed by Figuera and Pasquale [3], can be useful for bounding jitter, which is important in some applications. The ideas of our paper can be combined with the results of [3] to bound jitter in the context of a non-work-conserving server (see concluding remarks).

Weighted Fair Queuing (WFQ) was proposed by Demers, Keshav and Shenker [1], who suggested its efficacy for throughput fairness and delay bounds. The main idea behind WFQ is to compute a tag for each packet based on the packet's scheduled departure time in an idealized bit-by-bit round-robin discipline, and then to serve packets in non-decreasing order of tag values. If N is the number of active flows contending for a link, then tag computation/update can take $\Theta(N)$ time.

The work of Demers et al. [1] was extended by Parekh and Gallager [7, 8], who showed that end-to-end delay bound for a flow can be computed under the following assumptions: (1) the burstiness of the flow is controlled by a suitable token bucket filter, and (2) all servers in the path of a flow use the WFQ discipline. Parekh and Gallager use the name PGPS (packetized general processor sharing) for WFQ, and they also refer to the idealized form of PGPS as GPS. We will use the terms PGPS and WFQ interchangably in this paper. The end-to-end delay bounds proved by Parekh and Gallager serve as benchmarks for comparing delay guarantees of other fair queuing algorithms.

Zhang [12] introduced Virtual Clock Fair Queuing, in which each packet is stamped with a tag based on a hypothetical clock that "ticks" at the rate assigned to that packet's flow. As in WFQ, the Virtual Clock scheme services packets in the non-decreasing order of packet tags. Zhang's scheme achieves the PGPS delay bound [3, 11] but fails to provide throughput fairness; Figure 1 shows an example where throughput fairness can be arbitrarily bad. The throughput unfairness of Virtual Clock is removed by Golestani's Self-Clocked Fair Queuing algorithm [5]. Unfortunately, this scheme (SCFQ) suffers from large delay bounds, as shown by [4]—the worst-case delay of a packet may be almost N times the delay guaranteed by PGPS, where N is the number of active flows.

Deficit Round Robin (DRR) [9], a variation of the classical *round robin* is also able to guarantee throughput fairness. Instead of servicing one packet per flow in each round, DRR services upto a specified amount Q per flow. DRR is almost as simple as round robin and can be implemented in $O(1)$ time per packet. Unfortunately, like SCFQ, the delay bounds for DRR can be unacceptably large. Recently, a new scheme proposed by Stiliadis and Varma [10], called Frame Based FQ, claims to guarantee PGPS delay and fairness bounds and be implementable in $O(\log N)$ time per packet. We have not been able to obtain details of this scheme, except for a brief description of its properties in [10], because of some patent-related issues. Table 1 summarizes the salient features of perviously known scheduling algorithms, and compares them to our new Leap Forward Virtual Clock scheme.

It can be seen from the table that our *Leap Forward Virtual Clock* scheme provides end-to-end delay bounds almost identical to that of PGPS fair queuing, along with throughput fairness, and yet can be implemented with a worst-case time $O(\log \log N)$ per packet. No other scheme we know of matches these bounds.

We remark that a vanilla implementation of our algorithm has *exactly* the same delay bounds as PGPS and needs $O(\log N)$ time per packet, thus matching the Stiliadis and Varma [10] scheme. However, the use of "tag coarsening" idea leads to an "exponential speedup" resulting in an $O(\log \log N)$

Scheme	Delay Bound	Fairness	Efficiency
GPS	0	Fair	Impractical
PGPS	Small	Fair	$O(N)$
SCFQ	Large	Fair	$O(\log N)$
Virtual Clock	Small	Unfair	$O(\log N)$
Deficit Round Robin	Large	Fair	$O(1)$
Frame Based FQ	Small	Fair	$O(\log N)$
Leap Forward VC	Small	Fair	$O(\log \log N)$

Table 1: A comparison of several well-known scheduling algorithms. By a "small" delay, we mean a delay that is only a small additive constant larger than GPS delay, while "fair" throughput is fairness comparable to that of SCFQ or GPS.

time implementation. Tag coarsening introduces only a minor increase in the delay bound—indeed, the increase is *comparable to the difference between the delay bounds of PGPS and GPS*. Since this difference has been considered negligible, it appears that our exponential speedup comes at a negligible cost in the delay bound. Our coarsening technique should have broader applicability, since it introduces the useful notion of trading small delays for significant gains in computational efficiency.

3. Leap Forward Virtual Clock

We use the term *flow* to denote a logical connection between a source and a destination. Each packet in a flow carries with it the ID of the flow to which it belongs. The packets in a flow pass through a sequence of servers (or, routers) along their path to the destination. We concentrate on analyzing the performance of our queuing algorithm at a single server. We will show that our algorithm falls within the general framework of *guaranteed rate* (GR) fair-queuing algorithms as defined by Goyal et al. [4], which allows us to establish a PGPS-like bound on the end-to-end delay in a multi-hop system. In the following, therefore, we describe the algorithm at a single server S, with an output rate B bits per second.

Let $f_1, f_2, \ldots, f_N$ denote the set of flows that are serviced at S, where flow f_i has a guaranteed rate of r_{f_i} bits/sec, and $\sum_{i=1}^{N} r_{f_i} = B$. The sequence of packets in a particular flow f is denoted $p_f^1, p_f^2, \ldots, p_f^k$, and their sizes (in bits) are denoted $l_f^1, l_f^2, \ldots, l_f^k$. As remarked earlier, we will assume a work-conserving server: as long there is a packet queued for some flow, the server will not be idle. The virtual clock underlying our fair queuing algorithm is associated with the server, and we can think of it as a counter that keeps track of the number of bits sent out by S. The output rate of S is B bits/sec, and thus servicing a packet p of length l increments the clock by l/B. Every packet to be serviced by S receives a *tag*, indicating the server clock value by which it must be serviced. The factors determining the tag of a packet p of flow f are

1. the length of p,
2. the rate assigned to f, and
3. the server clock value when p reaches the head of its queue.

The tag of the jth packet in flow f is denoted $T(p_f^j)$.

The tags associated with the packets of a flow can be interpreted as a local clock for that flow. Under ideal traffic conditions, when all flows are sending packets at their guaranteed rates, the local clocks of all flows (almost) concur with the server clock. In reality, flows send packets in bursts and some flows misbehave, causing local clocks to deviate significantly from the server clock. The following simple example, due to Stiliadis and Varma [10], shows how the original virtual clock scheme of Zhang fails to provide throughput fairness. Figure 1 illustrates this example.

> Consider two flows, each with a guaranteed rate of 0.5 byte/sec, and let the server rate be 1 byte/sec. For the first 1000 secs, only flow 1 is active, and it sends at twice its allocated rate. In the absence of any other flows, it is able to send 1000 bytes during this interval. At server time 1000, flow 2 becomes active. The first packet of flow 2 receives the virtual clock value 1000, while the virtual clock of flow 1 equals 2000. Thus, for the next 500 seconds, flow 2 can send at twice its rate, and flow 1 will not be able to send any packet. So, even though both flows are backlogged during the time interval [1000, 1500], one of the flows receives no service at all, violating throughput fairness.

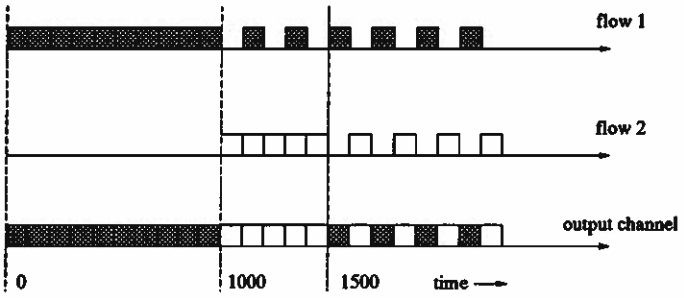

Figure 1: Why the Virtual Clock scheme does not guarantee throughput fairness.

Our scheme will rectify this shortcoming by ensuring that the tags assigned to different flows at any time lie within a bounded interval of the current server clock value. First, we need to define a few more terms.

Let t_s^c denote the *current* server time (clock value) at any instant. The arrival time of a packet p_f^j, denoted $A(p_f^j)$, is defined as the server time when p_f^j reaches the head of its flow queue. We define the *tag* (scheduled service time) of p_f^j as follows:

$$T(p_f^j) = \max\{A(p_f^i),\ T(p_f^{j-1})\} + \frac{l_f^j}{r_f}. \tag{1}$$

The definition of service tags is nearly identical to the one used in the classical virtual clock algorithm;[3] later we will describe an important modification of the original virtual clock scheme, where we occasionally advance the server clock.

Define the *current packet* of a flow f, denoted p_f^c, to be the one at the head of its queue. Define the *current tag* of a flow f, denoted t_f^c, to be the larger of the current server time and the tag of the current packet of f. That is,

$$t_f^c = \max\{T(p_f^c),\ t_s^c\}. \tag{2}$$

[3] One minor difference: instead of a real-time clock, we use a discrete server clock to stamp arrival instants. This only helps in the implementation because we can dispense with the real-time clock.

Let t_f^{prev} denote the tag of the last packet sent by f; it is zero if no packet of f has been sent. For technical convenience, if a flow f does not have a packet in its queue at the current time, then we assume $p_f^c = NIL$, $l_f^c = 0$, and $t_f^c = \max\left\{t_f^{prev}, t_s^c\right\}$. Therefore, the following formula for computing tags is equivalent to the one in Eq. (1):

$$T(p_f^j) = \max\{t_f^{prev}, t_s^c\} + \frac{l_f^j}{r_f}. \tag{3}$$

Finally, we define a quantity Δ that controls the *leap forward* step in our algorithm:

$$\Delta = \max_f \left(\frac{l_f^{max}}{r_f}\right), \tag{4}$$

where l_f^{max} is the size of the largest packet in flow f. Thus, Δ is the time needed to send the largest packet by a flow at its guaranteed rate. With these preliminaries, we are now ready to describe our *leap forward virtual clock* fair queuing algorithm.

Leap Forward Virtual Clock

1. [**Initialize.**] Each flow f has a variable t_f^{prev}, which maintains the tag of the last packet of f serviced. The global variable t_s^c maintains the server clock. All these variables are initialized to 0.
2. [**Enqueue.**] When a packet p_f^j arrives at the head of f's queue, its tag is computed using the expression in Eq. (3), and $T(p_f^j)$ is inserted into the priority queue H.
3. [**Dequeue**] When the server is ready to service the next packet, let p be the packet in the priority queue H having the smallest tag. Suppose that p belongs to the flow f.
   - [**Leap Forward.**] If $T(p) > t_s^c + 2\Delta$, then set $t_s^c = t_s^c + \Delta$.
   - [**Service.**] Transmit the packet p. Increment the server clock to $t_s^c + \frac{l}{B}$, where l is the size of p and B is the server rate. Delete p's entry from H, and update $t_f^{prev} = T(p)$.

The algorithm maintains a priority queue H over the current packets of each flow in the increasing order of their tags. Within each flow the packets are serviced in first-in-first-out order, and the first unserviced packet has its tag in H. (Thus, each flow with unsent packets waiting in its queue has exactly one packet in H.) The server always picks the packet p in H with the smallest tag for servicing next; when p's transmission is complete, p is deleted from H and the server clock is incremented by l/B, where l is the length of p.

The algorithm initializes the server clock to zero at the start, and whenever the server becomes idle. After the initialization, the algorithm repeatedly performs the following two operations asynchronously: enqueue, which inserts into H a packet that reaches the head of its queue, and dequeue, which picks the packet in H with the smallest tag for servicing next. Our crucial modification of the Virtual Clock scheme occurs in the dequeue operation where *we advance the server clock by* Δ

whenever the smallest tag in H differs from t_s^c by more than 2Δ. This step is the key to achieving throughput fairness. Figure 2 gives an example illustrating the leap forward (and leap backward) operation.

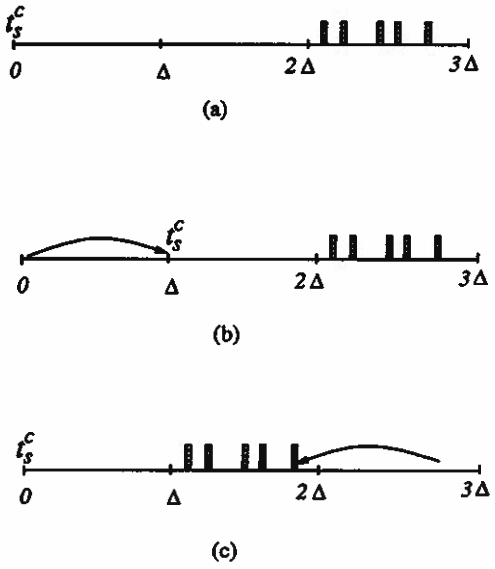

Figure 2: Illustration of the Leap Forward and Leap Backward steps. In Fig. (a), the server clock lags more than 2Δ behind the smallest tag in the priority queue. Fig. (b) shows the Leap Forward view, where t_s^c advances by Δ, while the tags remains the same. Fig. (c) shows the Leap Backward view, where all tags retreat by Δ, while the server clock remains the same.

4. Leap Backward Virtual Clock

While Leap Forward is an elegant and simple modification of the original virtual clock, its analysis is complicated by the leap forward step, which can advance the server clock in unpredictable ways. However, a dual version of this algorithm, which we call *Leap Backward*, has identical behavior but allows an easier analysis. It differs from Leap Forward only in the clock shifting step.

> *Whenever the Leap Forward scheme advances the server clock by* Δ, *the Leap Backward scheme decreases the tags of all active flows by* Δ.

Again, see Figure 2. The two schemes have identical external behaviors because the only difference between them is a relativistic time shift, which does not change the values of tags relative to the server clock. Unlike Leap Forward, which advances a single value (server clock), Leap Backward might have to update tags of all the flows at every step, making it computationally unattractive. However, our main motivation for introducing Leap Backward is the simplicity of its analysis; Leap Forward is clearly the better choice for implementation. A key property of Leap Backward scheme that proves critical in analyzing delay bounds is that its *server clock runs almost in lockstep with real time*. Due to their behavioral equivalence, of course, the delay and throughput bounds for the two systems are the same, and so we will only analyze the backward system.

Leap Backward Virtual Clock

1. **[Enqueue.]** Identical to Leap Forward scheme.
2. **[Dequeue.]** Let p be the packet in the priority queue having the smallest tag. Suppose that p belongs to the flow f.
   - **[Leap Backward.]** If $T(p) > t_s^c + 2\Delta$, then
     (a) For all flows f, update $t_f^{prev} = t_f^{prev} - \Delta$.
     (b) For all tags T in H, update $T = T - \Delta$.
   - **[Service.]** Transmit the packet p. Increment the server clock to $t_s^c + \frac{l}{B}$, where l is the size of p and B is the server rate. Delete p's entry from H, and update $t_f^{prev} = T(p)$.

Leap Backward has the same behavior as the Leap Forward scheme, as proved in the following lemma. Specifically, given an identical packet sequence, the two queuing schemes will output packets in the same sequence at the same time.

Lemma 4.1 (Relativistic Equivalence) *Let LF and LB be two systems using the Leap Forward and Leap Backward server disciplines, respectively. If the two systems are input the same arrival distributions of packets and are run concurrently, then the following holds:*

- *Each flow f has the same relative tag $(t_f^{prev} - t_s^c)$ in both the systems.*
- *The priority queues of the two systems have tags of the same packets, and the relative values of these tags $(T(p) - t_s^c)$ are the same in both systems.*

PROOF. The only difference between LF and LB is that whenever the former adds Δ to the server clock, the latter subtracts Δ from each of the tags. It follows then that the *relative* difference between a tag and the server clock in *both* the system is the same. Finally, in both systems, the next packet to be serviced is the one whose tag is *closest* to the server clock, the sequence of packets serviced and the packets at the head of each queue are the same. This completes the proof. □

In the next section, we show that Leap Backward guarantees that a packet is serviced before the server clock reaches its tag value. Observe that a packet's tag can be modified by the algorithm, but since the modification only *decreases* a tag (Leap Backward), servicing a packet by its *current* tag surely guarantees that it is serviced by its *original* tag.

5. Proof of the Delay Bound

A PGPS type delay bound is not obvious for the virtual clock scheme. Simply servicing packets in the order of increasing tags does not guarantee that the server has enough time to meet all deadlines. Consider the hypothetical example of N flows, each with a guaranteed rate of 1 byte/sec. Suppose

a (small) 10 byte packet arrives at the head of flow f when the server clock has value 1000, and the virtual clock scheme assigns it a tag at least as large as 1010. It might be possible that each of the remaining $N-1$ flows already has a large packet, say of size 1000 bytes, in the priority queue with tags strictly less than 1010—these packets may have arrived much earlier, say around time zero. Then, the small packet in the first flow will suffer a delay of about 1000 secs, in violation of the PGPS delay guarantee. In the PGPS system, the delay of the 10-byte packet should be about 10 secs.

We will show that situations like the one described above *cannot occur* in our Leap Backward scheme or in ordinary Virtual Clock[12]. We first establish the delay bound at a single server, then extend it to the end-to-end case.

5.1. Delay at a single server

The key to our proof is an important inequality relating a time window $(t_s^c,\ t)$ and the total size of all packets with service tags belonging to that window (Lemma 5.1). The lemma is intuitively depicted in Fig. 3. Consider an arbitrary future time $t > t_s^c$. Packets with tags less than t represent the backlog that must be serviced before the server advances to t, and the server has bandwidth to transmit $(t-t_s^c)\ B$ bits during the interval (t_s^c, t). Thus, a necessary condition is that the backlog be less than $(t_s^c - t)\ B$. In fact, a stronger inequality between the backlog packet sizes and the server bandwidth is needed because: i) currently inactive flows can send new packets during the interval (t_s^c, t), and ii) a flow with its current tag less than t can also schedule another packet in the interval (t_s^c, t). The corrective term needed in the inequality turns out to be each packet's "slack," which represents the gap between t and the packet's tag (Figure 3).

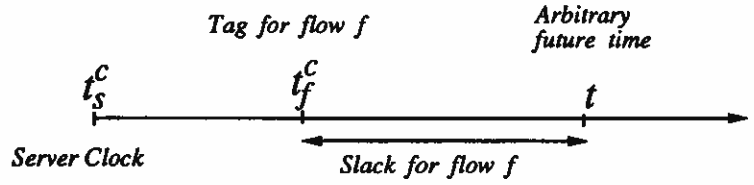

Figure 3: Lemma 5.1 states that the backlog of packets with tags less than an arbitrary time t is never more than the number of bits that the server can transmit in the interval (t_s^c, t) *minus* the slack contributed by each flow.

To state the lemma formally we need the following notation. Consider the current server time t_s^c, and any future time $t > t_s^c$. Let Φ_t^c denote the set of all flows whose current tags lie in the open interval $(t_s^c,\ t)$. That is,

$$\Phi_t^c = \{f \mid t_s^c < t_f^c < t\}.$$

Recall that the current tag of any flow is never less than the current server time. Not all flows of Φ_t^c necessarily have a packet at the head of their queue at the current time. A flow might have had a packet tagged with clock value between t_s^c and t, which has already been serviced, and so its current tag equals the tag of that packet. Thus, Φ_t^c is a superset of flows whose packets might need to be serviced before server clock reaches t.

Lemma 5.1 (Backlog Inequality) *Let S be a server, with output rate B, using the Leap Backward algorithm. If t_s^c is the current server time, and $t > t_s^c$ is an arbitrary time in the future, then the following bound holds:*

$$\sum_{f \in \Phi_t^c} l_f^c \ \leq\ (t-t_s^c)B \ -\ \sum_{f \in \Phi_t^c} (t-t_f^c)\ r_f. \tag{5}$$

PROOF. Our proof is by induction on the number of *events*, where an event is one of the threee possible steps of the algorithm: packet servicing, insertion of a new tag (enqueue), and the Leap Backward step. The base case of the induction holds trivially at initialization, where $t_s^c = 0$ and the priority queue is empty. In the following, we show that the inequality is preserved by each of the three events.

1. [*Service.*] The server chooses the packet p having the smallest tag in H for service, deletes p from H, and the server clock is advanced by the amount l/B, where l is the length of p. This event decreases the left hand side of the inequality by l. We show that the right hand side decreases by no more than l.

   Let $t' = t_s^c + l/B$ be the new server clock value. Then, the first term in the right hand side decreases by $(t - t_s^c)B - (t - t')B = (t' - t_s^c)B$, which is precisely l. The second term remains unchanged, since there can be no f for which $t_f^c \in (t_s^c,\ t')$—that would contradict Ineq. (5) for $t = t'$ because p had the smallest tag in H and both p and the current packet of f would need to be serviced before t'. Thus, advancing the server clock from t_s^c to t' does not change the second term, and this proves the inductive step for the first type of event.

2. [*Enqueue.*] Suppose a packet p_f arrives at the head of queue for the flow f. Let t_f^c be the current tag of f before p_f arrives. Then, p_f receives the following tag:

$$T(p_f) = t_f^c + \frac{l_f}{r_f}.$$

   There are two cases to consider, depending upon whether or not $T(p_f) < t$. If $T(p_f) \geq t$, then $f \notin \Phi_t^c$, and the lemma clearly holds. On the other hand, if $T(p_f) < t$, then the left hand side of our inequality increases by l_f. The first term of the right hand side remains unchanged, but the second term *decreases* by

$$(t - t_f^c)\ r_f\ -\ (t - T(p_f))\ r_f\ =\ (T(p_f) - t_f^c)\ r_f\ =\ l_f,$$

   which shows that the inequality is preserved.

3. [*Leap Backward.*] This event occurs when the minimum tag in H exceeds $t_s^c + 2\Delta$. Clearly, decreasing the current tags does not change the left hand side, but it significantly reduces the right hand side. We will show that right hand side has sufficient slack to absorb the decrease caused by this relative advancement of the server clock. Indeed, we prove that a stronger inequality holds:

$$\sum_{f \in \Phi_t^c} l_f^c\ \leq\ \sum_{f \in \Phi_t^c} (t - t_s^c)\ r_f\ -\ \sum_{f \in \Phi_t^c} (t - t_f^c)\ r_f, \tag{6}$$

   which clearly implies the lemma as $\sum_{f \in \Phi_t^c} r_f \leq B$. Let us consider any flow $f \in \Phi_t^c$, whose current packet is p_f of length l_f. Let t_s^c denote the current time; let t_f^c and t_f^n, respectively, denote the current tag of f before and after the Leap Backward step. Then, by assumption, $t_f^c > t_s^c + 2\Delta$. Together with the Leap Backward equality, namely, $t_f^n = t_f^c - \Delta$, we get the following inequalities:

$$\begin{aligned} \Delta &< t_f^n - t_s^c \\ \frac{l_f^{max}}{r_f} &< t_f^n - t_s^c \\ l_f^{max} &< ((t - t_s^c) - (t - t_f^n))\ r_f. \end{aligned}$$

The right hand side of the above inequality is the contribution of f to the right hand side of Ineq. (5) *after* the Leap Backward step. Since l_f^{max} denotes the maximum size of a packet of f, it follows that Leap Backward preserves the inequality for each flow, and therefore also for the set of flows Φ_t^c.

Thus, we have shown that each of the three types of events that modify the priority queue or the packet tags preserves the inequality. This completes the proof. □

The preceding lemma implies that at any time t the server has sufficient bandwidth to service all the packets scheduled before t. This allows us to bound the service time of a packet.

Lemma 5.2 (Service Time) *A packet p is serviced by the time the server clock reaches the value of its tag, namely, $T(p)$. Thus no packet in H has a tag less than t_s^c.*

PROOF. Our proof is by induction on the number of Dequeue operations. The base case ($n = 0$) holds trivially. Now, assume that the lemma holds for n operations, where the nth dequeue services packet q, with tag $T(q)$. Let t_s^c be the server clock when the service of q is completed. Suppose that the $(n+1)$st operation services a packet p with tag $T(p)$. Since packets are serviced in the order of increasing tags, we must have $T(p) \geq T(q)$. If l_p is the length of p, then invoking Lemma 5.1 with $t = T(p)$ gives

$$l_p \leq (T(p) - t_s^c)B.$$

Thus, the service of p is completed at time $t' = t_s^c + \frac{l_p}{B} \leq T(p)$. Thus, the $(n+1)$st dequeue operation is finished by $T(p)$, completing the induction. Our proof above shows that p is serviced no later than its current tag $T(p)$. This value may be *smaller* than the original tag due to Leap Backward adjustments, but that only works in favor of our delay bound proof. □

The preceding lemma bounds a packet's service time in terms of the virtual server clock, whereas one might prefer it in terms of real time. This, however, is easy because these two time values differ only by a small constant under Leap Backward scheme.

Lemma 5.3 (Server Clock Deviation) *The server clock value in the Leap Backward algorithm always satisfies the following bounds:*

$$t_s^c \leq t \leq t_s^c + \beta,$$

where t is real time and $\beta = \frac{l^{max}}{B}$.

PROOF. During the busy period, the server clock is ticking at the same rate as the real time, with the only difference that while the real time advances smoothly, the server clock advances in discrete steps of size l_i/B, where l_i are the sizes of packets serviced by the server. The server clock is incremented by l/B when it completes the service of p. Thus, real time never lags the server clock, and it can be ahead of the server clock by at most $\frac{l^{max}}{B}$, where l^{max} is the size of the largest packet serviced by B. This establishes the lemma. □

The preceding two lemmas imply that a packet p leaves the Leap Backward system no later than the real time value $T(p) + \beta$, where $T(p)$ is the *initial* tag of packet p and β is the time spent by the server in transmitting the largest packet in the system. We will use this in the next subsection to prove end-to-end delay bounds.

5.2. End-to-End Delay Bound

Seminal work on End-to-end delay bounds has been done by Parekh and Gallager [7, 8]. Recently, Goyal et al. [4] have synthesized an elegant and general strategy for proving delay bounds in the style of Parekh-Gallager. Their formulation applies to a large class of scheduling/queuing algorithms, which they call Guaranteed Rate scheduling.[4] We show that Leap Backward and Leap Forward schemes also belong to the Guaranteed Rate class, and thus we can use the framework of [4] to derive end-to-end delay bounds. We begin by reviewing the definition of the Guaranteed Rate class from [4].

Consider a flow f with a guaranteed rate r_f. Let p_f^j and l_f^j denote the jth packet of flow f and its length. Let $GRC(p_f^j)$ denote the GR clock value assigned to p_f^j, and let $A_G(p_f^j)$ denote the arrival time of packet p_f^j at the server; the packet arrival in the GR system is measured in real time, and so we use primed notation to distinguish it from our arrival time, which is measured in server clock time. The guaranteed rate clock values are defined as follows:

$$\begin{aligned} GRC(p_f^0) &= 0 \\ GRC(p_f^j) &= \max\{A_G(p_f^j),\ GRC(p_f^{j-1})\} + \frac{l_f^j}{r_f} \end{aligned} \tag{7}$$

A scheduling algorithm belongs to the class GR if it can guarantee that a packet p_f^j is transmitted by $GRC(p_f^j) + \beta$, where β is a constant dependent only on the scheduling algorithm and the server. We show that the Leap Forward (and Leap Backward) scheme belongs to this class. We first show a preliminary lemma:

Lemma 5.4 (Initial Tags are Bounded by Guaranteed Rate Clock) *Let $T(p_f^j)$ be the tag assigned to a packet p_f^j by a server using the Leap Backward Virtual Clock algorithm; this is the initial tag computed at the time of packet's arrival, and before any Leap Backward adjustments made by the algorithm. Then, $T(p_f^j) \leq GRC(p_f^j)$.*

PROOF. We prove the lemma by induction on j, the number of packets in a flow f. The base case ($j = 0$) clearly holds as $GRC(p_f^0) = T(p_f^0) = 0$. The tag $T(p_f^j)$ is computed Leap Backward computes the tag using Eq. (1):

$$T(p_f^j) = \max\{A(p_f^j),\ T_L(p_f^{j-1})\} + \frac{l_f^j}{r_f},$$

where $T_L(p_f^{j-1})$ is the *final* value of the tag as adjusted by Leap Backward algorithm. Clearly, $T_L(p_f^{j-1}) \leq T(p_f^{j-1})$, since Leap Backward only decreases a tag. Next, while A is measured in server clock time and A_G is measured in *real time*, Lemma 5.3 shows that these times satisfy $A(p_f^j) \leq A_G(p_f^j)$. Finally, the induction hypothesis guarantees that $T(p_f^{j-1}) \leq GRC(p_f^{j-1})$, which together with the preceding inequalities establishes the claim that $T(p_f^j) \leq GRC(p_f^j)$. □

Lemma 5.5 (Leap Backward is GR) *Leap Backward and Leap Forward Virtual Clock algorithms belong to the class GR, for the constant $\beta = \frac{l^{max}}{B}$, where l^{max} is the largest size packet transmitted by the server and B is server's rate.*

[4] Many well-known fair queueing algorithms, including virtual clock, SCFQ, PGPS, belong to the class GR.

PROOF. Lemmas 5.2 and 5.3 show that a packet p_f^j is transmitted by the server no later than real time $T(p_f^j)+\beta$, which is no later than $GRC(p_f^j) + \beta$, as proved by Lemma 5.4. This completes the proof. □

The GR framework allows us to bound end-to-end delay assuming that the burstiness of flow f is controlled. The use of a *leaky bucket* traffic shaper at the source is a standard assumption; we will use this assumption to establish a PGPS-like bound for our algorithm. We say that a flow f obeys a *leaky bucket process* with parameters (σ_f, r_f) if the total number of bits arriving during a interval (t_1, t_2) in flow f satisfy

$$AP_f(t_1, t_2) \leq \sigma_f + r_f(t_2 - t_1).$$

Now suppose there are K servers along the path of a flow f, where the ith server is denoted i. Let 0 and $K+1$, respectively, denote the source and the destination. Let β^i represent the value of β for the i-th server. Let $\alpha^i = \beta^i + \tau^{i,i+1}$, where $\tau^{i,i+1}$ is the *propagation* delay between the servers i and $i+1$. Then, the following result about end-to-end delay is established in Goyal et al. [4].

Lemma 5.6 (Goyal et al.) *Suppose that a flow f conforms to a leaky bucket process with parameters (σ_f, r_f), and the scheduling algorithm at each of the K servers on its path belongs to GR. Then, the end-to-end delay of a packet p_f^n, denoted by d_f^n, is given by the following:*

$$d_f^n \leq \frac{\sigma_f}{r_f} + (K-1)\max_{j=1}^{n}\frac{l_f^j}{r_f} + \sum_{i=1}^{K}\alpha^i.$$

We can now state our main theorem on end-to-end delay bound.

Theorem 5.1 (End-to-end Delay Bound for Leap Forward Fair Queuing) *Suppose that a flow f conforms to a leaky bucket process with parameters (σ_f, r_f), and the scheduling algorithm at each of the K servers on its path is Leap Forward or Leap Backward Virtual Clock. Then, the end-to-end delay of a packet p_f^n, denoted by d_f^n, is given by the following:*

$$d_f^n \leq \frac{\sigma_f}{r_f} + (K-1)\max_{j=1}^{n}\frac{l_f^j}{r_f} + \sum_{i=1}^{K}\alpha^i.$$

PROOF. The theorem follows immediately from Lemmas 5.5 and 5.6. □

Goyal et al. [4] also show how to obtain probabilistic bounds on the end-to-end delay when the burstiness of a flow is bounded with a stochastic process (e.g., exponentially bounded burstiness). The same results also apply to our scheduling algorithms. In next section, we address the throughput fairness of our algorithm.

6. Proof of Throughput Fairness

We will show that our scheme guarantees near-ideal throughput fairness. Informally, a scheme is fair if each backlogged flow receives its fair share of the available server bandwidth. More formally, we adopt a measure of throughput fairness defined by Golestani [5].

A flow is said to be *backlogged* during an interval (t_1, t_2) if the queue for flow f is never empty during (t_1, t_2). Let $sent_f(t_1, t_2)$ denote the total number of bits of f transmitted during (t_1, t_2) by the server. The throughput fairness of an algorithm is defined to be the maximum (absolute) difference between rate-normalized values of $sent_f(t_1, t_2)$ and $sent_f(t_1, t_2)$ over all pairs of backlogged flows and over all intervals (t_1, t_2). In other words, consider an execution of our scheduling algorithm, and define $F(t_1, t_2)$ as follows:

$$F(t_1, t_2) = \max_{f,g} \left| \frac{sent_f(t_1, t_2)}{r_f} - \frac{sent_g(t_1, t_2)}{r_g} \right|,$$

where max is over all pairs of flows that are backlogged during (t_1, t_2). Then, the throughput fairness is measured by the worst-case maximum value of $F(t_1, t_2)$ over all intervals and all executions of the queuing algorithm:

$$F = \max_{(t_1,t_2)} F(t_1, t_2).$$

We call a service discipline *fair* if the quantity F is a small. In particular, F should be a constant, independent of the length of the time interval [5]. PGPS schemes achieve $F \leq l^{max}$, and the self-clocked fair queuing (SCFQ) has $F \leq 2l^{max}$. However, no finite bound can be shown for the original virtual clock scheme—in a worst case, $F \to \infty$.

We prove that our Leap Forward or Leap Backward Virtual Clock algorithms achieve $F \leq 8l^{max}$. The constant factor in this bound can be improved to 6 or even smaller with minor modification of the algorithm, at the expense of slightly complicating the implementation. However, we believe that these constants are small enough that throughput fairness should not be an issue for our algorithm, even without the additional fine tuning.

Our proof of the throughput fairness depends crucially on the following fact: *the current tags of any two backlogged flows can differ by at most* 3Δ *at any time.* Before we prove that lemma, we establish the following auxiliary lemma.

Lemma 6.1 (Flow Tag Bound) *Let f be a flow that currently does not have a packet in its queue. Then, $t_f^c < t_s^c + 2\Delta$.*

PROOF. If f does not have a current packet, then either $t_f^c = t_s^c$ or $t_f^c = t_f^{prev}$. In the former case, the lemma holds trivially. In the latter case, let t be the server time at which p_f^{prev} was serviced. At that moment, indeed $T(p_f^{prev})$ was the minimum value in H. If $T(p_f^{prev}) > t + 2\Delta$, then the Leap Backward step must have decremented the tag of f by Δ, thus ensuring that when the service of p_f^{prev} was completed, the gap between t_f^{prev} and the current server time was less than 2Δ. This completes the proof. □

Lemma 6.2 (Tag Difference Bound) *Let p_1 and p_2 be two packets present in the priority queue at any time. Then, their tags satisfy the following:*

$$|T(p_1) - T(p_2)| \leq 3\Delta.$$

PROOF. Following our proof for the delay bound, we will use induction on the number of events. The base case of the induction holds trivially at the start, when the priority queue is empty. There are three events: enqueue, service, and leap backward; for the purpose of this lemma, the only nontrivial event is the enqueue operation.

1. [*Enqueue.*] A new packet p arrives at the head of f's queue. It receives the tag

$$T(p_f) = t_f^c + \frac{l_f}{r_f}.$$

Since a flow can have at most one packet in H, at the time of p's arrival, f had no packet in H. By the preceding lemma, therefore, $t_f^c < t_s^c + 2\Delta$. Since $l_f/r_f \leq \Delta$, we see that $T(p_f) < t_s^c + 3\Delta$. By the delay bound property of our scheme, every packet currently in H has tag value at least as large as t_s^c, and therefore the lemma holds for this case.

2. [*Service.*] The server picks the packet p with the minimum tag in H, services it, deletes it from H, and the server clock advances by l/B. The lemma clearly holds in this case, since no new packet is added in this step.

3. [*Leap Backward.*] The minimum tag in H has value greater than $t_s^c + 2\Delta$, and so the tags of all flows are decremented by Δ. This step clearly does not change the *relative* magnitudes of the tags.

This completes the proof of the lemma. □

We can now prove the throughput fairness theorem for our algorithm.

Theorem 6.1 (Throughput Fairness for Leap Forward Fair Queuing) *Leap Forward (or Leap Backward) Virtual Clock algorithm guarantees that* $F \leq 8l^{max}$.

PROOF. Consider a flow f backlogged during an interval (t_1, t_2). Let p_f^j and p_f^{j+m}, respectively, be the packets at the head of f's queue at time t_1 and t_2. Observe that during an interval when f is backlogged, the tags satisfy:

$$T(p_f^{i+1}) = T(p_f^i) + \frac{l_f^{i+1}}{r_f}, \qquad \text{for } j \leq i \leq j+m-1.$$

Therefore, we have the following equation relating the tags of jth and $(j+m)$th packet:

$$T(p_f^{j+m}) = T(p_f^j) + \sum_{i=1}^{m} \frac{l_f^{j+i}}{r_f}.$$

On the other hand, the total number of bits sent by f is upper bounded by

$$sent_f(t_1, t_2) \leq \sum_{i=0}^{m-1} \frac{l_f^{j+i}}{r_f};$$

the packets p_f^j through p_f^{j+m-1} have all been transmitted, and p_f^{j+m} may be in the process of transmission at time t_2.

Letting $p_f^{t_1}$ and $p_f^{t_2}$, respectively, denote packets at the head of f's queue at the two ends of the interval (t_1, t_2), we get the following inequality:

$$\frac{sent_f(t_1,t_2)}{r_f} \leq T(p_f^{t_2}) - T(p_f^{t_1}) - \frac{l_f^{t_1}}{r_f} + \frac{l_f^{t_2}}{r_f}. \tag{8}$$

Similarly, another flow g backlogged during the same interval yields:

$$\frac{sent_g(t_1,t_2)}{r_g} \leq T(p_g^{t_2}) - T(p_g^{t_1}) - \frac{l_g^{t_1}}{r_g} + \frac{l_g^{t_2}}{r_g}. \tag{9}$$

Lemma 6.2 guarantees that tags of two packets present in H at any time cannot differ by more than 3Δ. Also by the definition of Δ, we have that $|\frac{l_f^{t_1}}{r_f} - \frac{l_g^{t_1}}{r_g}|, |\frac{l_f^{t_2}}{r_f} - \frac{l_g^{t_2}}{r_g}| \leq \Delta$. Plugging these bounds in Eqs. 8 and 9, we get

$$\left|\frac{sent_f(t_1,t_2)}{r_f} - \frac{sent_g(t_1,t_2)}{r_g}\right| \leq 8\Delta.$$

This completes the proof of the theorem. □

7. Implementation and Data Structures

In this section we will only consider the Leap Forward scheme because it is easier to implement although the throughput and delay bounds apply to both schemes. The only nontrivial data structure needed for implementing the Leap Forward Virtual Clock is a priority queue. There are several well-known priority queue data structures that achieve $O(\log N)$ time per operation for insert, delete, and findmin. Each flow maintains its incoming packets in a queue, where a new packet can be added at the tail or the packet at the head of the queue can be extracted in $O(1)$ time. Thus, using standard data structures for maintaining priority queues, we can implement the Leap Forward Virtual Clock algorithm in $O(\log N)$ time per packet.

In the next subsection, we show how to achieve a significant saving in computation overhead, with only a minor penalty in delay and throughput fairness. In particular, we show that by maintaining only a coarse virtual clock and a finite-universe priority queue, we can achieve $O(\log \log N)$ processing time per packet.

7.1. Tag Coarsening and Finite-Universe Priority Queue

The basic idea of tag coarsening is that maintaining *exact* order among virtual clock tags is overkill if one is willing to tolerate a minor increase in latency. For instance, suppose the server rate is B and the largest packet size in any flow is M. Then, rounding up all the tags to multiples of M/B dramatically reduces the underlying key space of the priority queue, while possibly increasing the delay by at most M/B. We further reduce the key space to a set of $O(N)$ integers, in the range $[1, cN]$ for a fixed constant c, by using a tag-separation property of our algorithm and *modular arithmetic* to recycle tags. With these ideas in place, we can then use "approximate sorting" and a finite-universe priority queue of van Emde Boas [2] to achieve $O(\log \log N)$ processing time per packet.

Our technique applies to most clock-based fair queuing schemes, and so a similar performance tradeoff can be realized for these schemes as well. Assume, by normalization, that the server bandwidth B is 1 bit/sec; M is the size of the largest packet in any flow; and the *guaranteed* rate of every flow is at least $\frac{1}{cN}$ bits/sec, where $c > 0$ is an absolute constant.[5]

Let us begin by considering the impact of coarsening the virtual clock on our algorithm's performance. Since the smallest packet size is 1 and the minimum rate is $1/cN$, the *smallest* amount by which the current tag of a flow increases is cN. On the other hand, the *largest* advance in a flow's tag is at most cMN, since M is the largest packet size in any flow. As a concrete example, let us set the level of granularity to be M. Then, all our tag values are *rounded up* to match the next smallest multiple of M. In other words, $T'(p)$, the *coarse tag* for a packet p, is defined as follows:

$$T'(p) \;=\; M \cdot \left\lceil \frac{T(p)}{M} \right\rceil . \tag{10}$$

The server clock is not affected by this coarsening–it still increments at the rate of $1/B$ per bit sent.

Let us first show that the use of coarsened tags increase the worst-case delay of a packet by only a minor amount, namely, $\beta = M/B$. The coarse clock has the granularity of M, meaning that the order between two tags that are less than M apart is indistinguishable. As a matter of fact, all packets with tags in the semi-open range $((i-1)M,\ iM]$ are scheduled for service at time iM. The following lemma shows a packet with coarse tag T is serviced no later than server clock time T.

Lemma 7.1 (Delay Penalty for Coarse Tags) *A packet is serviced before the server clock reaches its coarse tag value. The additional delay introduced by the coarsening of tags is at most M/B.*

PROOF. Intuitively, the lemma holds because coarsening can only relax the constraints on the server by delaying the scheduled time of service for a packet. More formally, consider the basic inequality on the sizes of pending packets, namely, Eq. 5. Clearly, the inequality holds true if we *increase* the tag of every pending packet in H (possibly by different amount for each packet)—the increase keeps the first term of the right hand side unchanged but may *decreases* the second term, which is *negative*. Since coarsening the tags (rounding up) only increases their values, we conclude that server will service each packet no later than its coarse tag. Finally, since rounding up can increase the tag of a packet by at most M, the additional delay suffered by a packet is no more than M/B. This completes the proof. □

7.2. Mapping Tags to a Finite Range

The preceding discussion shows that we can use rounding to restrict our tag values to multiples of M, causing only a minor increase in the delay bounds. Taking this idea one step further, we show that tag values can be mapped to a finite universe of integers, which allows us to use a specialized priority queue yielding a worst-case bound of $O(\log\log N)$ per packet. While the coarse tags are always multiples of M, their absolute values can grow without bounds, as the server clock itself grows with the total amount of bits sent. Lemma 6.2 implies that the largest tag in the priority queue is always within 3Δ of the server clock.

[5] If there are flows with vastly different rates, we can partition them into groups of similar rates. The technique we are about to present works as long as the minimum and the maximum rates are within a constant factor of each other in each group. We maintain flows of each group in a separate priority queue, and organize them into a higher level priority queue. We omit the details of this standard method from this version.

Thus, to keep the tags within a bounded range of integers, we use *modular arithmetic* over the coarse tags, as follows. Recalling that $1/cN$ is the minimum possible rate for a flow, for some fixed constant $c > 0$, we get $\Delta = cMN$. We compute the coarse tags *modulo* $L = 3cMN + 1$, so that all values are multiple of M in the range $[0, 3cMN]$. That is,

$$T'(p) \;=\; M \cdot \left\lceil \frac{T(p) \pmod L}{M} \right\rceil. \tag{11}$$

With the modular arithmetic, we need to maintain the priority queue of tags as a *circular list*, where the tags are ordered circularly starting from t_s^c, since the tag wrap around whenever they exceed $L - 1$. When maintaining a circular list as a priority queue, we also need to modify the meaning of the *minimum* element—the packet to be serviced next is the one with the smallest tag greater than t_s^c *mod* L. This is, however, an implementation issue that we will discuss shortly when describing our priority queue. More pertinent to the present discussion is the observation that performing tag calculations modulo L does not affect the delay or throughput behavior of our algorithm in any way. Thus, we have succeeded in establishing the following lemma.

Lemma 7.2 (Rounding) *We can implement Leap-Forward Virtual Clock using coarse tags so that all values are multiples of M and they lie in the range $[0, 3cNM]$. The coarsening increases the worst-case delay of a packet by at most M/B seconds, where M is the largest packet size and B the output rate of the server.*

In the following section, we show that the coarse tags can indeed lead to significant efficiency gains while compromising the delay and throughput performance only in a very minor way. Without any restriction on the size and range of service tags, it can be shown that $\Omega(\log N)$ overhead per packet is necessary to maintain a priority queue on service tags—the problem is equivalent to sorting N numbers. We beat this bound using a data structure called *van Emde Boas priority queue* by exploiting the fact that our tags lie in a finite universe of size $O(N)$.

7.3. The van Emde Boas Data Structure

We will use a priority queue that facilitates the operations *insert, delete,* and *successor* in $O(\log\log L)$ time, *provided* that the elements are integers in the range $[0, L]$. Assume that $U = \{0, 1, 2, \ldots, L\}$ is the underlying universe, and $X \subset U$ is a subset currently maintained by the priority queue. The operation *insert*(y, X) adds a *new* element y to X, *delete*(y, X) removes an element y currently in X, and *successor*(y, X) finds the *smallest* element in X that is larger than y; if y is the largest element, a special symbol ∞ is returned. We quote the following result from van Emde Boas [2]; Mehlhorn's book [6] contains an eminently readable account of this data structure.

Lemma 7.3 (van Emde Boas) *Let $U = \{0, 1, 2, \ldots, L\}$, and let $X \subset U$ be a subset. The operation insert(y, X), delete(y, X), and successor(y, X) can be implemented in time $O(\log\log L)$ each. The priority queue can be initialized in time $O(L \log\log L)$ using $O(L \log\log L)$ space.*

In our algorithm, the service tags are integer multiples of M in the range $[0, 3cMN]$. Thus, the finite universe in our case is the set $\{0, 1, \ldots, 3cN\}$, which has size $O(N)$. The operations insert and delete are the standard priority queue operations, used to put in or remove a packet from the queue. The operation of finding the packet with the next service tag is implemented using *successor*. We call the function *successor*(t_s^c, X). Unless the special symbol ∞ is returned, we have found the next packet to be serviced. However, if ∞ is returned, we know that, because of the modular arithmetic, the next packet lies before t_s^c. In this case, we call *successor*$(-1, X)$ to find the smallest tag in X. We conclude with the following theorem, which summarizes the main result of our paper.

Theorem 7.1 (Leap Forward Virtual Clock Theorem) *Leap Forward Virtual Clock fair queuing algorithm achieves end-to-end delay bound almost as good as PGPS, has throughput fairness, and can be implemented with the worst-case computational overhead $O(\log\log N)$ per packet, where N is the number of active flows in the system.*

7.4. Preliminary Implementations

We have implemented the $O(\lg\lg N)$ priority queue but have not finished optimizing it. The constants are reasonable but we have found that a second scheme based on tries, outperforms the (unoptimized) Van Emde Boas structure for small values of N. The trie scheme views a K bit tag b bits at a time, and goes through a tree of arrays b bits at a time. If b is small we can avoid searching through empty array elements by keeping a bitmap and doing a table lookup to determine the lowest non-null array position. A trie-based implementation for values in the range $[0..2^{21}-1]$ requires worst-case 85 Sparc instructions for Insert and 183 instructions for ExtractMin. Note that for both the trie scheme and the Van Emde Boas scheme, reducing the size of the tag by coarsening is crucial for decreasing processing costs. We hope that further optimizations will reduce the cost to around 40 instructions for values of $N < 10000$.

8. Concluding Remarks

We believe there are several important contributions of this paper. The first contribution is the fair queuing algorithm itself: Leap Forward Virtual Clock. We have shown that a simple modification of Zhang's virtual clock scheme is able to provide throughput fairness, without compromising its delay bounds. In view of the elegance and simplicity of the original virtual clock scheme, our scheme should have practical appeal.[6] While our suggested Leap Forward modification is simple to state, there are a number of seemingly simple variants that do not work. One particularly intuitive scheme we explored was to simply *clamp* a packet's tag at time $\max\{T(p), t_s^c + c\Delta\}$, for any constant c. This has the desired effect of controlling the tag's deviation from the server clock, but we discovered that it fails to guarantee throughout fairness because clamping loses information about a packet's size.

A more theoretical contribution of our paper is to show that $\Omega(\log N)$ processing per packet is *not necessary* for achieving latency and throughout fairness. The implicit use of *sorting* in all virtual-clocked (or self-clocked) scheduling algorithms might lead one to believe the logarithmic lower bound. Indeed, prior to our work, no fair queuing algorithm with delay and throughput bounds matching those of PGPS was known to use sub-logarithmic computational overhead per packet. We have shown that by using "coarsened" packet tags, one can avoid the general complexity of sorting arbitrary numbers, and achieve $O(\log\log N)$ cost per packet. It remains a tantalizing theoretical open problem whether fair queuing with near-ideal (approaching PGPS) delay and throughput bounds can be realized with $O(1)$ cost per packet.

The $O(\log\log N)$ computational overhead is a small constant for all practical values of N. As an illustration, consider a router of the future handling $N = 32,000$ flows; then, $\log N \approx 15$, while $\log\log N \leq 4$. We believe that the *scalability* offered by the $O(\log\log N)$ scheme holds tremendous promise as the networks become larger, more complex, and operate at increasingly higher speeds. Our preliminary implementation shows that the underlying constant is reasonable; we believe that with further optimization we can rival more conventional (e.g., trie-based) schemes, for even small

[6] Some readers may have noticed a small difference between our and Zhang's scheme, in that we use an artificial server clock to mark packet arrival times, whereas Zhang uses real time. This is a rather superficial difference but we prefer our scheme because it does not require the use of a real-time clock.

values of N. Tag coarsening is crucial to the speed of both our trie-based and $O(\log\log N)$ implementations because their costs decrease as the tag size decreases.

Trading a small delay bound for a significant gain in computational efficiency appears to be a useful general idea, which should have other applications beyond the specific $O(\log\log N)$ fair queuing algorithm of our paper. By rounding tags to different multiples, we can navigate to various points in this tradeoff space. We used the van Emde Boas data structure in our scheme, but any priority queue that beats logarithmic bound for keys in a bounded range (such as the trie data structure) can be used instead. When the value of N is small to moderate, these data structures may even yield better constants. We are currently starting to implement several different data structures to test and compare their relative performance in practice.

In this paper, we have focused on work-conserving service disciplines. While non-work-conserving disciplines in general do not lead to better latency bounds, they can be useful for bounding jitter, which is useful in some applications. We believe that our ideas can be generalized to these settings as well using ideas similar to that in Figuera and Pasquale [3], where the virtual clock algorithm is generalized by adding delay regulators.

A final feature of our scheme that might be of possible interest is that it can give improved delay bounds for flows that exceed their assigned rates. The Leap Forward step of our algorithm ensures that a flow's tag never exceeds $t_s^c + 3\Delta$. Thus the delay penalty for such a flow is always bounded, without adversely affecting the delay bounds or the throughput fairness of other flows. Except for this effect, the average (not the worst-case) delay bounds of our scheme should be similar to those of the standard virtual clock scheme [12]. We are currently in the process of carrying out a simulation study to verify these and other behaviors.

References

[1] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Proc. Sigcomm '89*, 19(4):1–12, September 1989.

[2] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[3] N. Figuera and J. Pasquale. Leave-in-time: A new service discipline for real-time communication in a packet-switching data network. *Proc. Sigcomm '95*, September 1995.

[4] P. Goyal, S. S. Lam, and H. M. Vin. Determining End-to-End Delay Bounds in Heterogeneous Networks. In *Proceedings of Workshop on Network and OS Support forAudio-Video*, pages 287–298, April 1995.

[5] S. J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, April 1994.

[6] K. Mehlhorn. *Sorting and Searching*, Volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Heidelberg, West Germany, 1984.

[7] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control: The Single Node Case. In *Proceedings of IEEE INFOCOM '92*, volume 2, pages 915–924, May 1992.

[8] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Controlin Integrated Services Networks: The Multiple Node Case. In *Proceedings of IEEE INFOCOM '93*, volume 2, pages 521–530, March 1993.

[9] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM '95*, pages 231–242, September 1995.

[10] D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *Technical Report UCSC-CRL-95-38, Dept. of Computer Enginieering and Information Sciences, University of California, Santa Cruz*, July 1995.

[11] G. G. Xie and S. S. Lam. Delay Guarantee of a Virtual Clock Server. *Techincal Report TR-94-24, Dept. of Computer Sciences UT-Austin*, October 1994.

[12] Lixia Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet-Switched Networks. *ACM Transactions on Computer Systems*, 9(2), May 1991.