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-97-30

July 22, 1997

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 similar to PGPS, 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 guarantee delay and throughput fairness similar to PGPS. Interestingly, both the classical virtual clock and the Self-Clocked Fair Queuing schemes can be thought of as special cases of our scheme, by setting the *leap forward* parameter appropriately.

We use Zhang's virtual clock as a point of departure. We modify the original virtual clock scheme, which does not give throughput fairness, using two simple ideas: a *quarantine* mechanism that temporarily takes oversubscribed flows offline, and a *leap forward* mechanism that advances the server clock forward when all flows are oversubscribed. These two mechanisms keep 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 fact, by the worst-case fairness measure, our scheme is comparable to Worst-case Weighted Fair Queuing—the burstiness shown by PGPS does not occur in our scheme.

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.

This work was supported by National Science Foundation under grant NCR-9628145. A short version of this paper appeared in Proceedings of Infocom '97, Kobe, Japan.

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

The study of service disciplines in integrated services networks has gained increasing significance with the emergence of multimedia applications and electronic commerce. Unlike the traditional data applications, the new breed of real-time applications such as video-on-demand, internet telephony, teleconferencing, and remote medical imaging must meet stringent quality-of-service (QoS) standards to be successful. In an integrated services network, a wide variety of applications with widely different communication rates (ranging from few bits to several megabits per sec), traffic patterns, and holding times share communication links, and thereby pose challenging problems in providing good bounds on end-to-end delay, bandwidth, jitter, and packet loss.

With a large number of delay-sensitive traffic sources contending for the bandwidth, a service discipline needs to provide a fast and effective mechanism for servicing the packets so that real-time traffic is not adversely affected by non-time-critical traffic such as ftp. At the same time, both kinds of traffic must receive a (preset) fair share of bandwidth. A lack of sufficient "firewalls" can result in a rogue traffic source (either a misbehaving user or a malfunctioning device) unjustly penalizing well-behaved sources. In today's Internet, for instance, a rogue user can send data an an uncontrolled rate, and thus seize a large fraction of the available bandwidth at the expense of "good" users. What are the most important goals of a service discipline?

1.1. Measures of a Good Service Discipline

Informally, a service discipline is a mechanism employed by the intermediate nodes of a network to decide which packet (from possibly many waiting ones) to send next on a outbound link. The widely

Sorting Scheme	Disadvantages
Hardware Architectures	Expensive and Inflexible
Calendar Queues	Average $O(1)$ but worst-case $O(N)$
Fibonacci Heaps	$O(1)$ for enqueue, $O(\log N)$ for dequeue
Our Scheme	$O(\log \log N)$ for both enqueue and dequeue

Table 1: Prior techniques for sorting tags are either expensive to implement or need $\Omega(\log N)$ time in the worst-case.

varying rates and unpredictability of traffic patterns make it a nontrivial problem, and many simple-minded schemes, such as first-in-first-out, have been shown to not meet QoS guarantees. We believe that the three most important properties of a service discipline are: its delay guarantee, throughput fairness, and computational overhead. An idealized scheme called GPS (generalized processor sharing) has been proposed as a benchmark for delay and throughput bounds. Unfortunately, the scheme is computationally impractical, and therefore a tremendous amount of effort has been spent in the past few years trying to realize a practical variant of GPS.

One such proposal was made by Demers et al. [5], called Weighted Fair Queuing. The delay and throughput bounds of this scheme similar to GPS; however, the computational overhead is $\Theta(N)$ per packet, which can be excessive as N is the number of flows sharing the link. Other more efficient fair queuing schemes, such as Deficit Round Robin [13] or Self-Clocked Fair Queuing [9] have significantly larger delay bounds. Very recently, two new algorithms have been proposed that match the service guarantees of WFQ [2, 14], and whose computational overhead is $O(\log N)$ time per packet.

While algorithms requiring $O(\log N)$ time per packet are a great improvement over the $O(N)$ time schemes, even this computational overhead may be too severe for routers that run at Gigabit speeds. Some researchers suggest using special-purpose hardware at routers to overcome the $\Theta(\log N)$ sorting bottleneck [9, 14]. Special-purpose hardware, however, can be costly. More significantly, as the service discipline algorithms and protocols are likely to evolve with time, a software implementation will permit easier and cheaper upgrades than one done in hardware.

Table 1 shows some of the techniques that other authors have proposed to avoid the sorting bottleneck. Some researchers consider only the tag computation step, not the cost of maintaining the tags in a priority queue. These algorithms claim $O(1)$ overhead per packet, but that does not include the sorting time. Indeed, the tag computation in our algorithm also takes $O(1)$ time. We have already mentioned the disadvantages of using a hardware architecture for tag maintenance. Some papers also cite calendar queues [3] as a data structure for maintaining tag in $O(1)$ time. This is also a misperception: while calendar queues have good average case bounds for insertion and retrieval, the worst-case time bound is $\Theta(N)$, which is no better than WFQ. Similarly, other data structures such as Binomial queues or Fibonacci heaps [4] also have the $\Omega(\log N)$ overhead for either the insertion or deletion of tag. Since each packet is enqueued and dequeued at least once, the cost of processing a packet is dominated by the more expensive operation. Of course, since sorting N arbitrary numbers has a information-theoretic lower bound of $\Omega(N \log N)$, no general-purpose scheme can avoid the cost of $O(\log N)$ for either insertion or deletion. Our speedup comes from using coarsened tags, which allows us to restrict the tag values within a small set of numbers.

1.2. Our Contributions

Our main result is a service discipline called *Leap Forward Virtual Clock*. The delay bound and throughput fairness of our scheme is almost identical to that of Weighted Fair Queuing. In fact, by

Scheme	Delay Bound	Fairness	Worst-case Fairness	Efficiency
GPS	0	Fair	Excellent	Impractical
PGPS	Small	Fair	Poor	$O(N)$
SCFQ	Large	Fair	Poor	$O(\log N)$
Virtual Clock	Small	Unfair	Poor	$O(\log N)$
Deficit Round Robin	Large	Fair	Poor	$O(1)$
Frame Based FQ	Small	Fair	Poor	$O(\log N)$
WF2Q	Small	Fair	Good	$O(\log N)$
Leap Forward VC	Small	Fair	Good	$O(\log \log N)$

Table 2: 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. By poor worst-case fairness, we mean a worst-case index that grows with the number of flows.

the worst-case fairness measure of Bennett-Zhang, our scheme is comparable to WF2Q; in particular, our scheme does not exhibit the bursty behavior shown by WFQ due to packets being serviced way ahead of schedule. Unlike the existing schemes, however, the computational overhead of our algorithm is only $O(\log \log N)$ per packet. This time bound is worst-case and includes all sorting overheads, in contrast with several existing algorithms whose $O(1)$ time complexity bound accounts only for "tag computation" and not the additional $O(\log N)$ cost each packet incurs for priority queue operations. 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 quite small.

Leap Forward Virtual Clock appears to be the first service discipline that achieves near-ideal delay and throughput bounds but has a computational overhead smaller than $O(\log N)$. Our paper also shows that the Virtual Clock scheme of Zhang [17] can be modified to achieve delay and throughput properties comparable to WFQ. Previously it was known that while VC has good delay bound [7, 16], it fails to provide good throughput fairness [9, 14]. We present an intuitive "Leap Forward" modification to Virtual Clock that preserves throughput fairness. By contrast, all the previous schemes [14, 1] with good delay and throughput bounds have been based on modifications of WFQ.

1.3. A Comparison with Previous Work

There are several different service disciplines, with varying performance metrics. How does our scheme compare to them? Due to lack of space, we are unable to discuss the previous work in detail. Instead, we summarize the salient features of previous algorithms in Table 1, and compare them to the Leap Forward Virtual Clock scheme. (It appears possible to modify the Frame Based Fair Queuing scheme to achieve a good worst-case fairness index; however, the scheme described in [14] has a high WFI.)

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 worst-case fairness, and yet can be implemented with a worst-case time $O(\log \log N)$ per packet; the computational overhead of our scheme is the lowest among those schemes that deliver good bounds for both delay and throughput fairness. A vanilla implementation of our algorithm has *exactly* the same delay bounds as PGPS but it needs only $O(\log N)$ time per packet, thus matching the results of Stiliadis-Varma [14] and Bennett-Zhang [2].

The rest of the paper is organized as follows. In Section 2, we formulate two key conditions, Delay Condition and Throughput Condition, which are necessary and sufficient to guarantee PGPS-like delay and throughput performance for virtual clock based schemes. In Section 3, we introduce our main algorithm, Leap Forward Virtual Clock. We prove our end-to-end delay bound theorem in Section 4, and throughput fairness theorem in Section 5. In Section 6, we describe our tag coarsening technique and the $O(\log\log N)$ time implementation. In Section 7, we report on our simulation experiments, and finally conclude in Section 8.

2. Virtual Clock Service Disciplines

Service disciplines based on a virtual clock work by assigning a tag (deadline) to each packet, representing the clock value by which the packet must be transmitted. The packets are then serviced in non-decreasing order of their current tags. It is not obvious that such a scheme can meet all tag deadlines, though it has been shown to be the case [7, 16]. In our investigations, we have isolated two distinct fundamental conditions that underlie delay and throughput guarantees. A virtual clock-based scheme needs to balance these two conditions in order to meet delay and throughput bounds. Our Leap Forward Virtual Clock scheme is one instantiation of a general class of service disciplines that can be derived from these conditions. In particular, by setting the parameters of these conditions appropriately, both the classical virtual clock and the Self-Clocked Fair Queuing can be derived as special cases of our general scheme.

We begin with some notation that is needed to describe our algorithm. 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. Packets in a flow pass through a sequence of servers (or routers) along their path to the destination. We will later use the general framework of *guaranteed rate clock* (GRC) fair-queuing algorithms introduced by Goyal et al. [8], to establish PGPS-like bounds on the end-to-end delay in a multi-hop system. With this framework, it suffices to establish the delay bound at a single server, and therefore we concentrate on a server S, with 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_i bits/sec, and $\sum_{i=1}^{N} r_i \leq B$. 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 $\ell_f^1, \ell_f^2, \ldots, \ell_f^k$. We also use the notation $\ell(p)$ to denote the length of p. We assume that S is work-conserving, meaning that it does not sit idle while there is a packet in some queue. The virtual clock is associated with the server, and we can think of it as a counter that keeps track of the 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 tag of the jth packet in flow f is denoted $T(p_f^j)$.

Let t_s 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. 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. We define the *tag* of p_f^j as follows:[1]

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

Next, we define the *current tag* of a flow f, denoted t_f. We say a flow f is *active* if it currently has at least one packet in its queue. Otherwise, f is called *idle*. If f is active, let p_f^c denote the packet at the head of f's queue; this is the current packet for f. The current tag of f is defined as follows:

$$t_f = \begin{cases} T(p_f^c) & \text{if } f \text{ is active} \\ \max\{t_f^{prev},\ t_s\} & \text{otherwise} \end{cases}$$

Finally, we define an important parameter Δ_f for each flow f:

$$\Delta_f = \frac{\ell_f^{max}}{r_f}, \tag{2}$$

where ℓ_f^{max} is the size of the largest packet in flow f. Note that Δ_f is the time needed to send the largest packet by a flow at its guaranteed rate. Lastly, we use the notation τ to denote the time to transmit a largest packet at the server rate; thus, $\tau = \frac{M}{B}$. (This quantity is the smallest delay one can guarantee in any system where preemption is not allowed.)

2.1. Delay Bound for the Virtual Clock Scheme

The classical virtual clock scheme [17] assigns a tag to each arriving packet using Eq. (1), and then services the packets in the non-decreasing order of tags. If one can show that each packet is transmitted no later than its tag value, then the GRC framework of Goyal et al. [8] can be used to show a good end-to-end delay bound. We distill a fundamental invariant, called *Backlog Inequality*, that is essential to ensuring that a packet is serviced by its tag. Although others have shown the delay bound for the virtual clock scheme, we believe that our analysis is new and simpler; more importantly, it paves the way for understanding our new scheme, Leap Forward Virtual Clock.

In order to formulate the Backlog Inequality, let t be an arbitrary server time such that $t \geq t_s$ and consider the time window (t_s, t). The Backlog Inequality relates the total amount of backlog (both packets waiting and potential future arrivals) that must be serviced within the time window (t_s, t) and the amount of bandwidth available $B \times (t - t_s)$. Naturally, a service discipline can meet all of its tag commitments only if the available bandwidth is no less than the backlog.

An intuitive depiction of the inequality is given in Fig. 1. Packets with tags less than t represent the current backlog that must be serviced before the server advances to t. In addition, there is also the potential backlog that can arise from (1) currently inactive flows sending new packets during this interval, and (2) a flow with its current tag less than t scheduling another packet after the current one is serviced. The maximum amount that a flow f can contribute to the potential backlog at t, however, is bounded by $(t - t_f)r_f$, which we call the "slack" of f at time t. If f sends more packets than this slack, then the tags of these packets will fall beyond t.

---

[1] The definition of service tags is nearly identical to the one used in the classical virtual clock algorithm. One minor difference is that, 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. The second formula for tags using t_f^{prev} is clearly equivalent to the first, and so we also use it frequently in our computation.

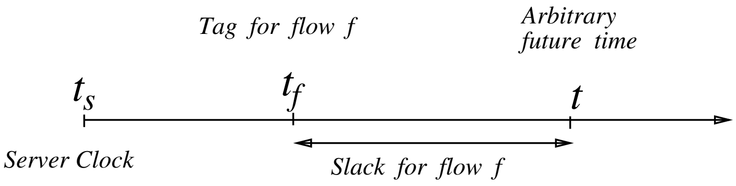

Figure 1: The Backlog Inequality states that the backlog of packets with tags less than an arbitrary time t *plus* the potential backlog contributed by future packets sent by each flow (the flow's slack) is never more than the number of bits that the server can transmit in the interval (t_s, t).

It is easy to see that the Backlog Inequality is a necessary condition for meeting the delay bound. We show later that this condition is also sufficient: *any service discipline that services packets in increasing order of tags and maintains the backlog inequality can meet all its deadlines.* Of course, the difficult part in the analysis is to show that a service discipline maintains the inequality throughput its operation.

We now formulate the Backlog Inequality more formally. Let Φ_t denote the set of all flows whose current tags have value at most t:

$$\Phi_t = \{f \mid t_f \leq t\}.$$

Thus Φ_t is the set of flows whose most recent packet to receive service has tag less than t. These are the flows whose packets might need to be serviced before server clock reaches t.

[**Backlog Inequality:**] Given a server S with output rate B, the current server time t_s and a future server clock value t, we say that the Backlog Inequality holds if the following inequality is satisfied:

$$\sum_{f \in \Phi_t} \ell_f + \sum_{f \in \Phi_t} r_f(t - t_f) \leq (t - t_s) \times B \tag{3}$$

On the left hand side, the first term is the current backlog, and the second term is the potential backlog or slack. The right hand side represents the available bandwidth in this time window. Later, we will demonstrate the power of this inequality: *any service discipline that maintains the Backlog Inequality throughput its operation guarantees a PGPS-like delay bound.*

2.2. Throughput Unfairness of Virtual Clock

Next, we revisit the problem that causes the throughput unfairness in the virtual clock scheme. In essence, the problem is caused by a large deviation between the server clock and a flow tag. This can happen when some flows send packets in bursts while others remain idle, as illustrated by the following example [14]. (See Section 7 for details of an actual simulation.)

Example 1: Consider a server with rate 1 byte/sec and two flows, each with a guaranteed rate of 1/2 byte/sec. Suppose that for the first 1000 secs only flow 1 is active, and it sends packets at twice its allocated rate. Since the server is work-conserving, it sends 1000 bytes of flow 1 during this interval. At server time 1000, flow 2 becomes active, and it also begins to send at

> the rate of 1 byte/sec. At this point, the current tag of flow 2 is 1000, while the current tag of flow 1 is 2000. Therefore, despite both flow 1 and 2 being backlogged, during the next 500 seconds, only flow 2 can send packets (at twice its rate), while flow 1 waits until time the server time reaches 1500. This violates the throughput fairness condition for flow 1.

Virtual clock can also give short-term throughput unfairness when the rates of flows can vary widely, as shown by the following example [1].

> **Example 2:** Consider ten flows $f_1, \ldots, f_{10}$ such that $r_1 = 1$ and $r_2 \ldots r_{10} = 0.1$. Assume that at time 0, all flows have a large number of 1 bit packets to send. The first packet of f_1 will receive a tag of 1 and the first packets of all other flows will receive a tag of 10. Thus, the first packet of f_1 will be serviced first; the second packet of f_1 will receive a tag of 2 and will again be serviced before the others. Continuing this way, the first 10 packets of flow 1 can be sent before any packets of the other flows are sent. At this point, the 11-th packet of f_1 has tag 11, while other flows have packets with tags equal to 10. Thus, for the next 9 packet service times, f_1 does not receive any service *even though it has the largest allocated bandwidth.* In summary, while f_1 does receive a fair share of the bandwidth on average, it can be locked out for a period equal to N packet service times, where N is the number of flows, causing "burstiness" in the service of a flow.

Somewhat surprisingly, Bennett and Zhang [1] show that even PGPS exhibits this form of bursty behavior. A better service discipline would allow packets of the low rate flows to be interleaved with those of the high rate flows during the initial period, thus avoiding the lockout periods for both the high and low rate flows. This is precisely what Leap Forward Virtual Clock does.

2.3. Delay and Throughput Conditions

The examples of the previous section help identify the cause of throughput unfairness. Example 1 illustrates how, in the virtual clock scheme, the current tag of a flow can deviate significantly from the server clock; it is easy to see that this deviation can grow arbitrarily large. Example 2 illustrates a more subtle (but still important) problem in that the tag of a flow can deviate from the server clock by N packet times, causing poor worst-case fairness in short term. We suggest that a service discipline should satisfy the following delay and throughput conditions if it is to deliver PGPS-like delay bound and a good fairness index.

[**Delay Condition:**] A service discipline satisfies the delay condition if it satisfies the Backlog Inequality at all times.

[**Throughput Condition:**] Let k be a constant independent of N. A service discipline satisfies the throughput condition if each flow f satisfies the following condition at all times:

$$t_f \ \leq \ t_s + k \cdot \Delta_f .$$

These conditions are necessary for a virtual clock scheme to guarantee delay and throughput bounds similar to PGPS. In the following sections, we establish the sufficiency of these conditions. These conditions quite naturally motivate our Leap Forward algorithm, which modifies the classical virtual clock by introducing two key ideas (quarantine and leap forward).

3. Leap Forward Virtual Clock

We begin with an informal description of the algorithm. At a conceptual level, the Leap Forward algorithm uses a common strategy employed by parents for disciplining children: take them temporarily offline. In our context, a flow in possible need of discipline is one that has recently received more than its allocated share of the bandwidth, and as such its flow tag might violate the Throughput Condition. We call such a flow *oversubscribed.* Once a flow becomes oversubscribed, any packets for the flow are placed in a holding area L (see Figure 2). The remaining, well-behaved flows are placed in a priority queue H. The server always picks the packet with the lowest current tag in H for servicing next. We note that the lowest tag in H may very well be bigger than the lowest tag in L, yet the packet from H is serviced first. *Thus, as our first departure from the classical virtual clock, our scheme does not necessarily service packets in the order of tags.* The data structures H and L should be thought of as *High* and *Low* priority queues.

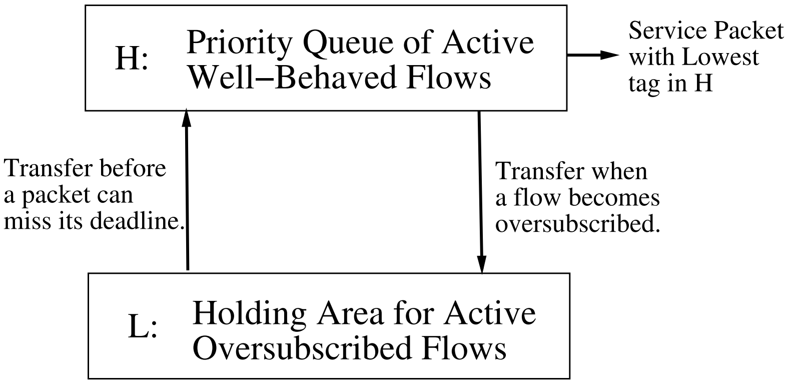

Figure 2: Conceptually Leap Forward Virtual Clock consists of a (high) priority queue H from which the next packet is chosen for service, and a holding area L for temporarily keeping oversubscribed flows. The precise rules for transferring flows between H and L are specified later. Observe that our algorithm does not pick the lowest active flow tag for service, but only the lowest among the active and *well-behaved* flows.

Just as children are forgiven after a "time off" period, we need to transfer an active flow from L to H for the flow to receive further service. How long can we wait before transferring a flow f in L back to H? Obviously, we must transfer f before there is any chance of violating the delay condition. If f is a flow in L and p is the packet at the head of its queue, then a safe test for keeping f in L is the following:

$$T(p) - t_s \ \geq \ \Delta_f .$$

We must transfer f to H before this condition is violated. We will show that this transfer policy allows us to maintain the delay condition, and every packet is serviced no later than its tag.

Thus, our scheme transfers a flow to L when it is likely to cause a throughput problem, and restores it to H before the delay guarantee of its packet can be violated. There remains, however, one problem: what happens if all flows are oversubscribed, and therefore H is empty? The work-conserving assumption of the server says that we must transmit a packet, yet there is no eligible packet. This brings us to the second important feature of our algorithm: Leap Forward. When H is empty, we *advance the server clock as far forward as possible without violating the delay invariant of any flows in L.*

It turns out that the Delay Condition for a flow f in L is satisfied if the following holds: $t_f - t_s \geq \Delta_f$. (We prove this fact later.) We call our scheme Leap Forward Virtual Clock because of the way the server clock can leap forward when all active flows are oversubscribed. After the leap forward step, we are guaranteed that at least one active flow in L becomes eligible for transfer to H. This ensures the work-conserving aspect of our server.

Observe that the leap forward step can be implemented by advancing the server clock to the smallest value of $(t_f - \Delta_f)$ among all active flows f that are in L. This observation suggests that we organize the flows in L as a priority queue with the key $t_f - \Delta_f$ (instead of t_f as in H). The use of these offset flow tags as a key allows us to check the Leap Forward and Transfer conditions only for the flow(s) at the head of L, instead of checking them for *all* flows in L.

3.1. Transfer Thresholds and the Structure of L

The general description above can be used to generate a whole family of Leap Forward schemes. A particular implementation involves fixing the values for certain parameters. For instance, maintaining the Delay Condition requires that there be "sufficient time" left for the deadlines of all flows in L. In our implementation, we will ensure that any flow f in L satisfies the following *No Rush* condition:

$$t_f - \Delta_f \;\geq\; t_s.$$

Any flow f in violation of the "No Rush" condition must be transferred to H. Since L is a priority queue with $t_f - \Delta_f$ as keys, it suffices to check this condition for the flow g at the head of L. If g violates the No Rush condition, we transfer it to H, and repeat the process on the new flow at the head of L, until we reach a flow that meets the condition. At this point, we can be sure that all flows remaining in L also satisfy the condition since they have larger values of $t_f - \Delta_f$. While at any time, we may have to perform multiple transfers from L to H, it is easy to see that each packet is extracted from L at most once, and so the amortized cost per packet is at most one Delete operation from L.

A subtle issue remains due to the discrete nature of our server clock. Since we test for the No Rush condition only after servicing each packet, a flow not in violation of this condition *before* the transmission of a packet might be in violation *after* the transmission; recall that the server clock advances in discrete steps at the transmission of a packet. This problem is easily fixed by being slightly more conservative in our check. We demand that a flow in L satisfy the following stronger condition:

$$(t_f - \Delta_f) \;\geq\; t_s + \tau.$$

Since servicing a packet can advance the server clock by at most $\tau = \frac{M}{B}$, where M is the largest packet in the system, we are guaranteed that if the No Rush condition holds prior to a packet's transmission, it holds after the transmission as well.

Finally, for our Throughput Condition, we fix $k = 2$. In other words, we will guarantee that any flow f satisfies the following inequality at all times:

$$t_f \;\leq\; t_s + \tau + 2\Delta_f.$$

(The quantity $\tau = M/B$ is added to ensure that the threshold to transfer a packet from H to L is higher than the one for transferring it back.) Maintaining this condition requires that we transfer a flow f from H to L when f has a packet to send and $t_f - \Delta_f \;>\; t_s + \tau$; otherwise, a single maximum sized packet of f transmitted from H will cause f to violate the Throughput Condition. Thus, the conditions for transfer between H and L have a rather pleasing symmetry.

3.2. Allowing Rounded Tags

The Internal Revenue Service allows tax filers to round all numbers to the nearest dollar. The intent is to reduce computation on the part of the tax filer, at the expense of a possible small loss to the treasury. We wish to investigate a similar tradeoff between accuracy of tags and the computational overhead needed for maintaining order among them. In particular, let ρ be some arbitrary number. Then, we we will "round" up tag values to a whole multiple of ρ. Implementing the Leap Forward scheme with exact tags corresponds to setting $\rho = 0$.

Rounded tags cause a minor increase in the delay and throughput bounds, but they allow for substantial reduction in the computational overhead needed for maintaining these tags in priority queues. Later we will see that reasonable values of ρ can lead to fairly large savings in computation. At the risk of complicating the presentation, we choose to develop our algorithm with rounded tags from the start. The reader who wishes to concentrate on the uncoarsened Leap Forward algorithm may mentally substitute $\rho = 0$ in the code and theorems throughput the following discussion.

Let us fix a value of ρ. The main idea of the coarsened version of Leap Forward algorithm is that while we continue to compute fully accurate tags t_f and t_s, we round them up to the nearest multiple of ρ *before inserting them into the priority queues H or L.* Since rounding may introduce an error of at most ρ in the relative values of two tags, all our tests, such as Throughput and Delay Conditions, can be off by ρ. We will later prove that this uncertainty increases the delay bound by at most ρ and also worsens the worst-case throughput fairness index by a small amount.

Both the Leap Forward and the Transfer conditions get modified by ρ to account for the loss of information by rounding. In particular, in the Leap Forward step, we advance the server clock to the ρ less than the minimum key in L. Similarly, a flow is transferred from L to H when the following condition is violated:

$$t_f - \Delta_f \;\geq\; (t_s + \tau + \rho).$$

Having increased the threshold for transferring a flow from L to H by ρ, we need to do the same for the reverse threshold. So, a packet p in flow f is added to L if $T(p) \;>\; (t_s + \tau + \rho + \Delta_f)$.

Finally, the tag-coarsening also affects the Backlog Inequality. Since our priority queues can no longer distinguish between tags that round to the same multiple of ρ, we cannot guarantee that a packet is serviced by the time server clock reaches its tag. Instead, we can only guarantee that a packet will be serviced by the time the server clock reaches the *rounded* value of its tag (hence, the additional delay term ρ). Consequently, we need only prove the Backlog Inequality for times that are whole multiples of ρ. For the non-rounded version of the algorithm ($\rho = 0$), however, the Backlog Inequality holds at all times.

3.3. Leap Forward Code

The Leap Forward algorithm asynchronously performs two operations repeatedly: Enqueue and Dequeue. The former handles the task of inserting a newly arrived packet in appropriate data structures, while the latter deals with prioritizing packets for service. The server always picks for transmission the packet p with the current smallest tag in H. The server clock is advanced by $\ell(p)/B$, and p is deleted from the flow queue. Only then does the server begin transmitting the packet p. Real time elapses only during the transmission of a packet, while all other events are assumed to be instantaneous. Notice that the server clock is incremented before the packet is actually transmitted!

The code for Leap Forward algorithm is described below. The algorithm maintains two key data structures, H and L, which are priority queues, allowing fast *Insertion* and *ExtractMin* operations. In addition, each flow f has associated with it a FIFO queue Q_f, which maintains the sequence of packets that have not been serviced yet. The variable t_f maintains the flow tag of f. The server clock is maintained by the global variable t_s. The variables t_s as well as t_f, for all f, are initialized to 0, and reset to zero whenever the server becomes idle.

The procedure $Insert(\mathcal{A},\ p,\ x)$ inserts the *rounded* value of x (assuming some fixed coarsening parameter ρ) into the priority queue $\mathcal{A}$. The value x is the tag of p if $\mathcal{A} = H$ and the offset tag if $\mathcal{A} = L$. In addition, the following procedure $ProcessHead$ is used to handle a packet p when it reaches the head of its queue Q_f.

```
$ProcessHead\ (Q_f)$
    if ($Q_f$ is empty) return;
    $p \leftarrow head(Q_f)$;
    $t_f \leftarrow T(p) \leftarrow \max(t_s,\ t_f) + \frac{\ell(p)}{r_f}$;
    if ($t_f \le t_s + \Delta_f + \tau + \rho$) then
        $Insert(H,\ p,\ t_f)$);
    else $Insert(L,\ p,\ t_f - \Delta_f)$;
$End.$
```

Leap Forward Virtual Clock

```
1. [Enqueue.] (* A new packet $p_f$ arrives. *)
        $AddToTail\ (Q_f)$;
        if ($Q_f$ was empty) then $ProcessHead\ (Q_f)$;

2. [Dequeue.] (* The server is idle and there is a packet in the system. *)

    $k_{min} \leftarrow MinKey(L)$;
    Let $f$ be the flow corresponding to the key $k_{min}$.
    if ($H$ is empty) then $t_s = \max(t_s,\ k_{min} - \rho)$        (* Leap forward *)
    while ($k_{min} < t_s + \tau + \rho$ )
        $p_f \leftarrow ExtractMin(L)$;
        $Insert(H,\ p_f,\ T(p_f))$;        (* Transfer *)
        $k_{min} \leftarrow MinKey(L)$;
    end
    $p_f \leftarrow ExtractMin(H)$;
    $t_s \leftarrow t_s + \frac{\ell(p_f)}{B}$.                (* Start Service *)
    $RemovefromTail\ (Q_f)$
    $ProcessHead\ (Q_f)$;
    Transmit $p_f$;                         (* Real time elapses *)
```

The following theorem shows that Leap Forward is work-conserving, and it never deadlocks.

THEOREM 3.1. *Leap Forward Virtual Clock is a work-conserving service discipline.*

PROOF. It suffices to show if there are packets queued in the system, a call to Dequeue results in a packet transmission. If any flow has a queued packet, it must have a tag either in H or L. If H

is nonempty, then obviously the packet with the lowest tag in H is transmitted, and the theorem holds. So, let us assume that only L has tags. The Leap Forward step ensures that the server clock is no less than $k_{min} - \rho$, where k_{min} is the smallest (rounded) key in L. If g is the flow associated with the key k_{min}, then we have $t_g < t_s + \Delta_g + \tau + \rho$, and therefore g is transferred to H, making the latter nonempty. This completes the proof. □

4. Delay Bounds for Leap Forward

We first establish the delay bound at a single server, and then extend it to multiple hops.

4.1. Delay at a Single Server

The key to our proof is showing that our algorithm maintains the Backlog Inequality. Due to our use of rounded tags, we only require that this Inequality hold for times t that are whole multiples of the parameter ρ. The following lemma collects some useful invariants on flow tags; the proof of this lemma can be found in the technical report [15].

Lemma 4.1 (Tag invariants) *Consider a flow f, and its current tag t_f. Leap Forward algorithm maintains the following invariants with respect to the current server time t_s:*

T1: *If t_f is in H, then $t_f \leq t_s + \Delta_f + \tau + \rho$.*

T2: *If t_f is in L then $t_f \geq t_s + \Delta_f$.*

T3: *Let ℓ_f be the length of packet at the head of f's queue (assume $\ell_f = 0$ if f is idle). Then, $t_f \leq t_s + \tau + \rho + \Delta_f + \ell_f / r_f$.*

PROOF. All three invariants are proved by induction on the number of events. The base case is trivial in all cases: in T1 and T2, there are no elements in H and L initially; in T3, $t_s = t_f = 0$ initially. Let us prove the general case for these invariants in order.

In proving the first invariant, observe that the only routines that add a flow to H are $ProcessHead$ and $Transfer$, and they do so only when the condition $t_f \leq t_s + \Delta_f + \rho$ is satisfied; the only event that removes a packet from H is *Service*, but it only increases the server clock, and hence maintains the invariant. The Leap Forward does not affect elements in H because it occurs only when H is empty.

In proving the invariant T2, observe that $ProcessHead$ is the only routine that adds a flow tag to L, but the flow must satisfy the condition $t_f > t_s + \Delta_f + \tau + \rho$, which maintains T2. Next, to consider Dequeue, notice that after the $Transfer$ phase, the inequality $t_f \geq t_s + \Delta_f + \tau + \rho$ must hold for flow at the head of L. Since a real tag can be at most ρ smaller than its rounded counterpart, we can be sure that the real tag of every flow in L now satisfies: $t_f \geq t_s + \Delta_f + \tau$. Servicing of the current packet can advance t_s by no more than τ, which still leaves enough slack to ensure that $t_f \geq t_s + \Delta_f$ for any flow in L after Dequeue.

Finally, to prove T3, note than the flow tag t_f is updated whenever a new packet p arrives at the head of Q_f. If p arrives at an empty queue, then by induction, the following condition holds prior to p's arrival: $t_f \leq t_s + \Delta_f + \tau + \rho$, since $\ell_f = 0$ prior to p's arrival. After p's arrival t_f increases by $\ell(p)/r_f$, which maintains the invariant. If, on the other hand, p arrives at a nonempty queue, then p reaches the head of its queue during the service component of Dequeue, when the predecessor of

p is transmitted. In this case, $t_f \leq t_s + \Delta_f + \tau + \rho$ holds prior to p reaching the head of the queue (cf. T1), and so after p reaches the head of its queue, t_f increases by $\ell(p)/r_f$, which maintains the invariant. If the service of a packet p of flow f leaves behind an empty queue, the invariant also follows easily, since in this case $\ell_f = 0$. The last two events affecting the invariant is the service component of Dequeue and Leap Forward, but they both advance t_s, and hence cannot violate the invariant. The proof is complete. □

Our next lemma proves a weaker version of the backlog inequality, which we use in establishing our final lemma. Let us recall the definition of the set Φ_t from earlier discussion: $\Phi_t = \{f \mid t_f \leq t\}$ is the set of flows whose current tags are less than t, for a server time $t \geq t_s$. We call a flow *backlogged* if it currently has at least one packet in its queue.

Lemma 4.2 (Sufficient Bandwidth) *The Backlog Inequality holds at t_s if, for all $t \geq t_s$, every backlogged flow $f \in \Phi_t$ satisfies $\ell_f \leq (t_f - t_s) r_f$.*

Proof. If a flow $f \in \Phi_t$ is *not* backlogged, then $\ell_f = 0$. Thus, our hypothesis implies that

$$\begin{aligned} \sum_{f \in \Phi_t} \ell_f &\leq \sum_{f \in \Phi_t} (t_f - t_s)\, r_f \\ &= \sum_{f \in \Phi_t} (t - t_s)\, r_f - \sum_{f \in \Phi_t} (t - t_f)\, r_f \\ &\leq (t - t_s)\, B - \sum_{f \in \Phi_t} (t - t_f)\, r_f \end{aligned}$$

where the last inequality uses the fact that $\sum_{f \in \Phi_t} r_f \leq B$. This establishes the lemma. □

Let us use the notation $R(n)$ to denote the value of n *rounded up* to the nearest multiple of ρ. Abusing the notation slightly, we also use $R(p)$ to denote $R(T(p))$. We are now ready to state and prove the following key lemma.

Lemma 4.3 (Backlog Lemma) *In the Leap Forward Virtual Clock scheme, the Backlog Inequality holds for all rounded values of t. That is, if t is a whole multiple of ρ, then the following holds:*

$$\sum_{f \in \Phi_t} \ell_f + \sum_{f \in \Phi_t} (t - t_f)\, r_f \leq (t - t_s) \times B.$$

Proof. Our proof is by induction on the number of *events*, where an event is either an Enqueue, or any of the following four steps of Dequeue: leap Forward, Transfer, Start Service, and End Service. Leap Forward and Transfer are well-defined; Start service includes all the code in the service routine except the actual transmission of the packet, which is considered a separate event called End Service. Note that real time elapses only during End Service.

The base case of the induction holds trivially at initialization, since H and L are empty, $t_s = 0$, $t_f = 0$ for all flows f, and no flow is backlogged. We now show inductively that each of the five events preserve the backlog inequality.

1. [*Enqueue.*] Suppose a packet p_f arrives at the head of queue for the flow f. Let t_f be the current tag of f before p_f arrives. Then, the tag of p_f is given by $T(p_f) = t_f + \frac{\ell_f}{r_f}$. We consider two cases, depending upon whether or not $T(p_f) < t$. If $T(p_f) \geq t$, then $f \notin \Phi_t$, and the lemma clearly holds. If $T(p_f) < t$, then the left hand side of the Backlog Inequality increases by ℓ_f. The first term of the right hand side remains unchanged, but the second term *decreases* by $(t - t_f)\ r_f \ - \ (t - T(p_f))\ r_f \ = \ (T(p_f) - t_f)\ r_f \ = \ \ell_f$, which shows that the inequality is preserved.

2. [*Transfer.*] This event transfers tags from L to H, but does not change any of the variables in the backlog lemma, namely t, t_f, t_s, or Φ_t.

3. [*Leap Forward.*] This event can advance the server clock, but it only occurs if H is empty. If the clock is advanced, its new value equals $t'_s \ = \ k_{min} - \rho$. Since the real tag of a flow can be at most ρ less than its rounded tag, and the smallest rounded tag in L equals k_{min}, it follows that after the Leap Forward step, the following condition holds for all flows (including the ones in Φ_t) in L: $t_f - \Delta_f \ \geq \ t'_s$, which implies the Backlog Inequality via Lemma 4.2.

4. [*Start Service.*] The server chooses a packet p with the smallest rounded tag in H for service, deletes it from H, and the server clock is advances by $\ell(p)/B$. Let the real tag of p be $T(p)$, and let t be a future time that is a whole multiple of ρ. We consider two cases, depending on whether or not $t \geq T(p)$. If $t \geq T(p)$, then Φ_t includes the flow f to which p belongs. Start service in this case decreases the left hand side of the Backlog Inequality by $\ell(p)$. We show that the right hand side decreases by no more than $\ell(p)$. Let $t' = t_s + \ell(p)/B$ be the new server clock value. Then, the first term in the right hand side decreases by $(t - t_s)B - (t - t')B = (t' - t_s)B$, which is precisely $\ell(p)$. The second term remains unchanged, since none of the other flow tags are changed by this event.

   If $t < T(p)$, then Φ_t clearly includes only flows with $t_f < T(p)$. If such a flow has a packet, it must be in L; since p has the lowest rounded tag in H, any packet with a lower rounded tag must be in L. By Lemma 4.1, however, any such flow f with a packet q satisfies $T(q) \geq t_s + \Delta_f$. Thus, each backlogged flow f in Φ_t satisfies $t_f \geq t_s + \Delta_f$, which implies the Backlog Inequality by Lemma 4.2.

5. [*End Service.*] This event does not affect any of the state variables of the lemma.

Thus, we have shown that each of the five types of events that modify the priority queues or the packet tags preserves the backlog 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, as shown in the following lemma.

Lemma 4.4 (Service Time) *A packet p is completely transmitted by the time the server clock reaches $R(p) \ \leq \ T(p) + \rho$. Thus, no packet in H or L has a tag smaller than $t_s - \rho$.*

Proof. Since t_s keeps increasing during a busy period, it will eventually reach $R(p)$. We show that p must have been serviced by this time. Only two events advance the server clock: Leap Forward and Start Service. It is easy to see that the former event cannot advance the server clock past $R(p)$ for any packet p: Leap Forward happens when there are no packets in H, and the new clock value does not exceed $t_f - \Delta_f - \rho$, where $t_f - \Delta_f$ is the smallest key in L and $R(p) \geq t_f - \rho$.

Now, consider the *first* time Start Service advances the server clock past a packet's rounded tag. Specifically, assume that this violation occurs when Dequeue process a packet p, and Start Service advances the clock past the rounded tag $R(q)$ of some packet q; we allow the possibility that $p = q$. Let t_s and t'_s, respectively, denote the server clock values before and after the service event.

We first note that q cannot be L at t_s: if q is a packet of flow f, then membership in L requires that $T(q) \geq t_s + \Delta_f + \tau$, and since $t'_s \leq t_s + \tau$, we immediately get the contradiction $t'_s \leq T(q) \leq R(q)$. So q must be H before the event, and we must also have $R(q) \geq R(p)$ since p was picked for service instead of q. Since this is the first violation, we also have $t_s \; \leq \; R(q)$. We now apply the Backlog Inequality with server time t_s and $t = R(q)$:

$$\ell(p) \; \leq \; \sum_{f \in \Phi_t} \ell_f \; \leq \; (R(q) - t_s) \times B.$$

The service of p is, therefore, completed at server clock time $t'_s \; = \; t_s \; + \; \frac{\ell(p)}{B} \; \leq \; R(q)$, which contradicts our assumption that $t'_s > R(q)$. This completes the proof. □

The preceding lemma bounds a packet's service time in terms of the virtual server clock, whereas one might prefer a statement in terms of the real time. We observe that real time can be considerably ahead of our server clock because server clock time is reinitialized at the start of every busy period. Within each busy period, the server clock ticks roughly in lockstep with real time except for leap forward adjustments. In order to relate real time to server time, we must keep track of the last idle instant as well as the amount of leap forward jumps in the server clock during the current busy period. We introduce two hypothetical variables, often called *history* variables, I and L for this purpose; these variables are not part of the algorithm, but merely a conceptual device for our proofs.

Let I be a variable that maintains the value of the real time when the server was last idle. If the server is currently busy, then I equals the real time when the last busy period began. If the server is idle, then I equals the current value of real time. Similarly, let L be a variable that maintains the amount of leap-forward done in the current busy period up to the current instant of time. At the start of a busy period, we set $L = 0$. Note that L is non-decreasing during a busy period. Observe that I only changes during idle periods, while L only changes during busy periods. The following lemma shows that real time equals server time plus $I - L$, except during a packet transmission, when it can be behind by at most τ.

Lemma 4.5 (Server Clock Deviation) *Let t be real time and let I and L be defined as above. Then, the following holds:*

1. *During a packet transmission, $t \; \leq \; (t_s + I - L) \; \leq \; t + \tau$.*
2. *At all other times, $t \; = \; (t_s + I - L)$.*

Proof. The invariant clearly holds during an idle period, since t_s and L are zero, while $t = I$. Once a busy period starts, the server clock changes only due to Leap Forward or service steps. If the server clock leaps forward by x, then both L and t_s increase by x, maintaining the invariant. When servicing a packet p, just before the service begins, we have $t = t_s + I - L$ (by inductive hypothesis. As soon as the service begins, the server clock t_s jumps by $\ell(p)/B$; the real time t hasn't changed yet, but the invariant holds. During transmission, t_s stays unchanged but t increases until p is completed transmitted at which point $t = t_s + I - L$ holds. □

The preceding two lemmas imply that a packet p is serviced in the Leap Forward system no later than the real time value $R(p) + I - L$. We will use this in the next subsection to prove end-to-end delay bounds.

4.2. End-to-End Delay Bounds for Leap Forward

Goyal et al. [8] have recently synthesized an elegant and general strategy for proving delay bounds in the style of Parekh and Gallager [11, 12]. Their formulation applies to a large class of algorithms, which they call Guaranteed Rate scheduling.[2] We show that Leap Forward schemes also belong to the GR class, and so we can use this framework to derive end-to-end delay bounds. We begin by reviewing the definition of the Guaranteed Rate class.

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 $A_R(p_f^j)$ 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 A_R to distinguish it from our arrival time, which is measured in server clock time. The guaranteed rate clock values are defined as follows:

$$GRC(p_f^j) \quad = \quad \max\{A_R(p_f^j),\ GRC(p_f^{j-1})\} \ + \ \frac{\ell_f^j}{r_f}, \tag{4}$$

where $GRC(p_f^0) = 0$. 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.

In the Leap Forward discipline, $T(p_f^j)$ denotes the tag assigned to a packet p_f^j. Recall our hypothetical variables I and L that keep track of the last instant when the server was idle and the total leap forward jumps during the current busy period. Let $I(p_f^j)$ and $L(p_f^j)$, respectively, denote the values of variable I and L at the instant the service of p_f^j is completed. Similarly, let $I'(p_f^j)$ and $L'(p_f^j)$, respectively, denote the values of I and L when p_f^j arrives at the head of its queue.

Lemma 4.6 (Leap Forward Tags vs. Guaranteed Rate Clock) *The following inequality always holds:* $T(p_f^j) + I(p_f^j) - L(p_f^j) \ \leq \ GRC(p_f^j) + \tau,$ *where* $\tau = M/B$.

PROOF. We prove the lemma by induction on j, the number of packets sent by a flow f. The base case ($j = 0$) clearly holds as $GRC(p_f^0) = I(p_f^0) = L(p_f^0) = T(p_f^0) = 0$. Now, assume that the lemma holds for $j - 1$, and consider the packet p_f^j. We analyze two cases separately depending on the magnitudes of $A(p_f^j)$ and $T(p_f^{j-1})$.

[**Case 1:**] Suppose $A(p_f^j) > \ T(p_f^{j-1})$. Then, we conclude that the tag of p_f^j satisfies:

$$T(p_f^j) = A(p_f^i) + \frac{\ell_f^j}{r_f}. \tag{5}$$

Using Lemma 4.5, we get the following inequality:

$$A(p_f^j) + I'(p_f^j) - L'(p_f^j) \ \leq \ A_R(p_f^j) + \tau.$$

Adding $\frac{\ell_f^j}{r_f}$ to both sides, and using Equation 5, we get the following:

---

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

$$T(p_f^j) + I'(p_f^j) - L'(p_f^j) \ \leq \ \max\{A_R(p_f^j),\ GRC(p_f^{j-1})\} \ + \ \frac{\ell_f^j}{r_f} + \tau \tag{6}$$

Since we are in a busy period, we have $I'(p_f^j) = I(p_f^j)$ and $L'(p_f^j) \leq L(p_f^j)$, we get the following inequality:

$$T(p_f^j) + I(p_f^j) - L(p_f^j) \ \leq \ GRC(p_f^j) \ + \ \tau,$$

which establishes the lemma for Case 1.

[**Case 2:**] Assume $A(p_f^j) \leq \ T(p_f^{j-1})$. In this case, we have $T(p_f^j) = \ T(p_f^{j-1}) + \frac{\ell_f^j}{r_f}$, which together with the inductive hypothesis leads to the following inequalities:

$$\begin{aligned} T(p_f^{j-1}) + I(p_f^{j-1}) - L'(p_f^{j-1}) \ &\leq \ GRC(p_f^{j-1}) + \tau \\ T(p_f^j) + I(p_f^{j-1}) - L(p_f^{j-1}) \ &\leq \ \max\{A_R(p_f^j),\ GRC(p_f^{j-1})\} \ + \ \frac{\ell_f^j}{r_f} + \tau \end{aligned} \tag{7}$$

Since we have $A(p_f^j) \ \leq \ T(p_f^{j-1})$, p_f^j and p_f^{j-1} both arrive and depart in the same busy period. Therefore, $I(p_f^{j-1}) = I(p_f^j)$ and $L(p_f^{j-1}) \leq L(p_f^j)$, and so Equation 7 and the definition of GRC gives

$$T(p_f^j) + I(p_f^j) - L(p_f^j) \ \leq \ GRC(p_f^j) \ + \ \tau,$$

which establishes the lemma in Case 2 and completes the proof. □

Lemma 4.7 (Leap Forward is GR) *Leap Forward Virtual Clock service discipline belongs to the class GR, with* $\beta = \rho + \tau$, *where* $\tau = M/B$, M *is the size of the largest packet in the system,* B *is server rate, and* ρ *is the coarsening parameter, which we can also set to* τ.

PROOF. The Leap Forward algorithm services a packet p_f^j no later than the server time $T(p_f^j) + \rho$ (Lemma 4.4). At the end of a packet transmission, real time equals $t_s + I - L$ (Lemma 4.5). Thus, the service of p_f^j is completed by real time $T(p_f^j) \ + \ \rho \ + \ I(p_f^j) \ - \ L(p_f^j)$, which is no later than $GRC(p_f^j) \ + \ \rho + \tau$ (Lemma 4.6). This completes the proof. □

The GR framework allows us to bound end-to-end delay for a flow f assuming that f's burstiness is appropriately controlled; no assumption is made regarding other flows. The use of a *leaky bucket* traffic shaper at the source is a standard assumption, which we also make to prove delay bounds similar to PGPS for our algorithm. We say that a flow f obeys a *leaky bucket process* with parameters $(\sigma_f,\ r_f)$ if the total number of bits arriving during a interval (t_1, t_2) in flow f satisfy the following inequality:

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

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 $\alpha^i = \beta^i + \pi^{i,i+1}$, where $\pi^{i,i+1}$ is the *propagation* delay between the servers i and $i+1$. Then, we have the following result:

Theorem 4.1 (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 Virtual Clock. Then, the end-to-end delay of a packet p_f^n, denoted by d_f^n, is given by the following:*

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

Proof. The theorem follows immediately from Lemma 4.7 and the result of Goyal et al. [8]. □

Note that Leap Forward has $\beta = \rho + \tau$, and if we eliminate tag coarsening (set $\rho = 0$), our result is identical to the standard delay bound of the classical virtual clock and PGPS. Tag coarsening adds an additional delay of at most ρ per server on the path. The parameter ρ is fairly flexible, and can be used to derive tradeoffs between this additional delay and savings in computational overhead of maintaining priority among tag values—the finer the tags, the greater the overhead.

Goyal et al. [8] 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, such as the exponentially bounded burstiness. The same results also apply to our scheduling algorithms. In next section, we address the throughput fairness of our algorithm.

5. Throughput Fairness of Leap Forward

Informally speaking, a scheme is fair if each backlogged flow receives its fair share of the available server bandwidth. There are two commonly used fairness measures: the first, proposed by Golestani [9], is a gross measure, while the second, introduced recently by Bennett and Zhang [1], is a more refined measure. Bennett-Zhang call their measure *worst-case fairness index* (WFI), and show that while PGPS is fair by Golestani's measure, it falls short of the ideal GPS by the worst-case fairness index. WFI more closely measures short-term bursty behavior. We will show that our Leap Forward Scheme satisfies both Golestani's as well as Bennett-Zhang's definition of fairness.

5.1. Fairness Measure of Golestani

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 the Leap Forward 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 algorithm:

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

We call a service discipline *fair* if F is a small. In particular, F should be a constant, independent of the length of the time interval [9]. It is known that in the case of the classical virtual clock, $F \rightarrow \infty$ in the worst-case.

We will prove that the fairness measure of Leap Forward Virtual Clock satisfies the following bound:

$$F \leq \max_{f,g} 3(\Delta_f + \Delta_g) + 2\tau + 4\rho.$$

To put this bound in perspective, we note that $\tau + \rho$ is negligible in practice, and the dominant term is $\max_{f,g} 3(\Delta_f + \Delta_g)$. By a slightly more refined analysis, the multiplicative factor can be improved to 2 (we omit the details here). By contrast, the Self-Clocked Fair Queuing algorithm of Golestani has fairness $\max_{f,g}(\Delta_f + \Delta_g)$, but it has poor delay guarantees.

Our proof of throughput fairness depends critically on two facts: one, the current tag of a backlogged flow differs from the server clock by no more than $2\Delta_f + \tau + \rho$ (Lemma 4.1), and, two, during a backlogged period (t_1, t_2), the initial and final tags of f differ by (roughly) the number of bits of flow f transmitted in (t_1, t_2) divided by its rate r_f. We are now ready to give a formal proof.

Theorem 5.1 (Throughput Fairness for Leap Forward Fair Queuing) *The fairness measure of Leap Forward Virtual Clock satisfies the following bound:*

$$F \leq \max_{f,g} 3(\Delta_f + \Delta_g) + 2\tau + 4\rho,$$

where $\tau = M/B$ and ρ is a tag coarsening parameter.

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:

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

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

$$T(p_f^{j+m}) = T(p_f^j) + \sum_{i=1}^{m} \frac{\ell_f^{j+i}}{r_f}. \tag{8}$$

The total number of bits sent by f is upper-bounded by

$$sent_f(t_1, t_2) \leq \sum_{i=0}^{m-1} \frac{\ell_f^{j+i}}{r_f}, \tag{9}$$

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

Let p_f^1 and p_f^2, respectively denote the packet at the head of f's queue at the two ends of the interval (t_1, t_2), and let ℓ_f^1 and ℓ_f^2, respectively, denote their lengths. Then, the preceding inequalities give the following bounds:

$$T(p_f^2) - T(p_f^1) - \frac{\ell_f^2}{r_f} \leq \frac{sent_f(t_1, t_2)}{r_f} \leq T(p_f^2) - T(p_f^1) + \frac{\ell_f^1}{r_f} \tag{10}$$

Let t_s^1 and t_s^2 denote the server time at the two ends of the interval (t_1, t_2). By Lemmas 4.1 and 4.4, we know that $t_s^2 - \rho \leq T(p_f^2) \leq t_s^2 + 2\Delta_f + \tau + \rho$ and $t_s^1 - \rho \leq T(p_f^1) \leq t_s^1 + 2\Delta_f + \tau + \rho$. Combining these with Equation 10, we get

$$t_s^2 - t_s^1 - 2\Delta_f - \tau - 2\rho \leq T(p_f^2) \; - \; T(p_f^1) \leq t_s^2 - t_s^1 + 2\Delta_f + \tau + 2\rho$$

.

Substituting the last equation in Equation 10 and using the fact that $\frac{\ell_f^1}{r_f}, \frac{\ell_f^2}{r_f} \leq \Delta_f$, we get:

$$t_s^2 - t_s^1 - 3\Delta_f - \tau - 2\rho \leq \frac{sent_f(t_1, t_2)}{r_f} \leq t_s^2 - t_s^1 + 3\Delta_f + \tau + 2\rho$$

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

$$t_s^2 - t_s^1 - 3\Delta_g - \tau - 2\rho \; \leq \; \frac{sent_g(t_1, t_2)}{r_g}$$

Subtracting the above two inequalities, we get

$$\left| \frac{sent_f(t_1, t_2)}{r_f} \; - \; \frac{sent_g(t_1, t_2)}{r_g} \right| \; \leq \; 3(\Delta_f + \Delta_g) + 2\tau + 4\rho$$

This completes the proof of the theorem. □

5.2. Worst Case Fairness Index of Leap Forward

The following measure of throughput fairness, due to Bennett and Zhang [2] provides a more refined measure of short term throughput unfairness than the Golestani measure described above. We start by introducing some notation.

The delay $D(p)$ of a packet p is the real time that elapses between the arrival time of p and the time p is completely transmitted. The arrival time of a packet p is denoted by $A(p)$. Assume p belongs to flow f. Let $Q(p)$ denote the size (in bits) of the queue in front of p (including p) at the time of p's arrival.

Definition 5.1. *A service discipline is said to be worst-case for flow f if, for any packet p in f, the following holds:*

$$D(p) \; < \; \frac{Q(p)}{r_f} \; + \; C_f,$$

where C_f is a constant independent of other flows sharing this server. The normalized worst-case fair index (WFI) is defined as

$$C = \max_f \frac{r_f C_f}{B}.$$

Bennett and Zhang [1] have shown that Worst-case Weighted Fair Queuing has an optimal WFI equal to $\tau = M/B$. We will show that the WFI of our scheme is nearly the same. Our proof depends critically on two facts: one, the current tag of a backlogged flow differs from the server clock by no more than $2\Delta_f + \tau + \rho$ (Lemma 4.1), and, two, during a backlogged period (t_1, t_2), the difference of the initial and final tags of f is (roughly) the number of bits of flow f transmitted during (t_1, t_2) divided by its rate r_f.

THEOREM 5.2. *The Worst-Case Fairness Index of Leap Forward Virtual Clock is bounded by*

$$\tau + \max_f \, 2(\tau + \rho)\, \frac{r_f}{B}.$$

PROOF. Consider a packet p for a flow f. Suppose that when p arrives at time $A(p)$, the packet at the head of f's queue is p_f^j. Suppose further that there are $m \geq 0$ packets in the queue in front of p, meaning $p = p_f^{j+m}$. Let t_1 denote the real arrival time $A(p)$ and t_2 the real time when p's service is complete. Let t_s^1, t_s^2 (resp. t_f^1, t_f^2) denote the server clock (resp. flow tags of f) corresponding to the real times t_1 and t_2. Lemma 4.1 guarantees the following inequality:

$$t_f^1 \;\leq\; t_s^1 + \Delta_f + k\frac{\ell_f^j}{r_f} + \tau + \rho,$$

where $k = 1$ if p_f^j is in L at $A(p)$, and $k = 0$ otherwise. Lemma 4.4 gives the bound $t_s^2 \leq t_f^2 + \rho$. Subtracting t_s^1 from t_s^2 gives the following:

$$t_s^2 - t_s^1 \;\leq\; t_f^2 - t_f^1 + \Delta_f + \tau + 2\rho + k\frac{\ell_f^j}{r_f} \tag{11}$$

Using the fact that f is backlogged during (t_f^1, t_f^2), we get $t_f^2 \;=\; t_f^1 \;+\; \sum_{i=1}^{m} \frac{\ell_f^{j+i}}{r_f}$, which when combined with the previous two equations gives

$$t_s^2 - t_s^1 \;\leq\; \sum_{i=1}^{m} \frac{\ell_f^{j+i}}{r_f} + \Delta_f + \tau + 2\rho + k\frac{\ell_f^j}{r_f} \tag{12}$$

Next, the queue size $Q(p)$ satisfies the inequality:

$$\sum_{i=1}^{m} \ell_f^{j+i} + k\ell_f^j \;\leq\; Q(p), \tag{13}$$

because $k \leq 1$. Plugging this bound in Equation 12 yields

$$t_s^2 - t_s^1 \;\leq\; \frac{Q(p)}{r_f} + \Delta_f + 2\rho + \tau. \tag{14}$$

Now, at time $A(p)$, we know that $t_s^1 + I - L_1 \;\leq\; A(p) + \tau$ (Lemma 4.5), where I is the idle counter and L_1 is the amount of leap forward in this busy period before the arrival of p. Therefore,

$$A(p) \;\geq\; t_s^1 + I - L_1 - \tau. \tag{15}$$

Similarly, we have $t_2 \;=\; t_s^2 + I - L_2$ (by Lemma 4.5), where L_2 is the amount of leap forward in this busy period until p departs. Clearly $L_2 \geq L_1$. Therefore,

$$t_2 \;\leq\; t_s^2 + I - L_1. \tag{16}$$

Now, using Eqs. 15 and 16, we can bound the delay of packet p as follows:

$$
\begin{aligned}
D(p) &= t_2 - A(p) \\
&\le (t_s^2 + I - L_1) - (t_s^1 + I - L_1 - \tau) \\
&= t_s^2 - t_s^1 + \tau \\
&\le \frac{Q(p)}{r_f} + \Delta_f + 2\tau + 2\rho \qquad \text{(By Eq. 14.)}
\end{aligned}
$$

Using the notation of Bennett and Zhang, we get $C_f \le \Delta_f + 2\tau$, which gives the following bound on the fairness index of Leap Forward:

$$
\text{WFI} \quad \le \quad \tau + \max_f \; 2(\tau + \rho)\frac{r_f}{B}.
$$

This completes the proof. □

In practice, r_f/B should be quite small, and if we set the coarsening parameter ρ to roughly τ, the WFI of Leap Forward approaches τ, which is the optimal value of WFI [1].

6. Implementation and Data Structures

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.

6.1. The 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 [6] to achieve $O(\log\log N)$ processing time per packet.

Our technique probably applies to other clock-based fair queuing schemes as well; however, working out the details requires careful proofs of the coarsened equivalents of such algorithms. We will start by describing our scheme for the case in which the maximum and minimum rates are within a constant factor of each other. We will later describe how to extend this technique to arbitrary rates by partitioning flows into groups that have similar rates; we will then use the same technique within each group, and an ordinary priority queue to find the lowest tags among groups.

Assume, by normalization, that the server bandwidth B is 1 bit/sec; M is the size of the largest packet in any flow. If the rates are within a constant factor of each other, the *guaranteed* rate of every flow is at least $\frac{1}{cN}$, where $c > 0$ is a constant. 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, $\rho = M$. Then, all our tag values are *rounded up* to match the next smallest multiple of M as described earlier. That is, all packets with tags in the range $((i-1)M,\ iM]$ are scheduled for service at time iM. The results of the previous subsection show that the use of coarsened tags increase the worst-case delay of a packet by only a minor amount, namely, $\tau = M/B$.

6.2. 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 4.1 implies that the largest tag in the priority queue is always within $2\Delta_f$ of the server clock.

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* $L = 2cMN + 1$, so that all values are multiple of M in the range $[0, 2cMN]$. That is,

$$T'(p) \;=\; M \cdot \left\lceil \frac{T(p) \pmod{L}}{M} \right\rceil . \tag{17}$$

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, 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 $t_s \pmod{L}$. 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 6.1 (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* $[0, 2cNM]$. *The coarsening increases the worst-case delay of a packet by at most* $\tau = M/B$, *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 $O(N)$.

6.3. 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 [6]; Mehlhorn's book [10] contains an eminently readable account of this data structure.

Lemma 6.2 (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 $[0, 2cMN]$. Thus, the finite universe in our case is the set $\{0, 1, \ldots, 2cN\}$, 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 $successor(t_s, X)$. 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. In this case, we call $successor(-1, X)$ to find the smallest tag in X.

Finally, it is easy to see from the code that once a packet p leaves L it can never return to L (the test for moving a packet to H is only done when the packet tag is assigned). Thus a packet must be serviced after at most one visit to L. Similarly, it is easy to see that once a packet is in H, it can never leave H till it is serviced. Thus a packet is serviced after at most one visit to H and L. Since each visit, can require an enqueue followed by a dequeue, this requires at most 2 enqueue and 2 dequeue operations per packet, which keeps the amortized packet processing cost at $O(\log\log N)$ per packet. We can make the cost of visits to L negligible by transferring packets to L after a higher threshold than $\Delta_f + \tau + \rho$. For example, if we use a threshold of $3\Delta_f + \tau + \rho$, we can ensure that at least 3 packets of a flow are sent completely from H before any packet is sent to L. This can reduce the average additional cost due to L to 1/3, but at the cost of a higher WFI.

6.4. Extending the Technique to Arbitrary Rates

We use a hierarchical scheme to extend the technique to arbitrary rates. Let r be the smallest rate of any flow in the system. We will group flows logarithmically in groups such that flows with rates in the range $[r, 2r)$ are assigned to group 1, flows with rates in the range $[2r, 4r)$ are assigned to group 2, and so on. Thus there can be at most $\log\frac{R}{r}$ such groups, where R is the maximum rate. We implement H and L separately for each group using van Emde Boas data structures. These structures allow us to find the packet with the smallest eligible tag within each group in $O(\log\log N)$ time.

The smallest tags of these H data structures are then organized in a separate top-level priority queue, and a similar priority queue is made for the minimum keys of all the L data structures. Since there are only $O(\log\frac{R}{r})$ elements to maintain at the top level, an ordinary heap data structure can be used to perform Insert and ExtractMin operations in $O(\log\log\frac{R}{r})$ time.

As long as there is some flow tag in the H data structure of some group, we process that tag, possibly updating the H data structure for that group. If all the H data structures are empty, we pick the lowest tag in the L structures, leap forward, transfer as many packets as are required to the H structures, and then dequeue again from the H structures. There are several details in the construction that we are leaving out here, and are deferred to a future paper. We note that the total cost per packet in the final scheme is $O(\log\log\frac{R}{r} + \log\log N)$, which should be comparable to $O(\log\log N)$ in most practical cases. We conclude with the following theorem, which summarizes the main result of our paper.

Theorem 6.1 (Leap Forward Virtual Clock Theorem) *Leap Forward Virtual Clock service discipline provides a guaranteed delay bound comparable to PGPS, throughput fairness comparable to worst-case WFQ, and can be implemented with the worst-case computational overhead of* $O(\log\log N)$ *per packet, where* N *is the number of active flows in the system.*

6.5. Preliminary Implementations

We have implemented the $O(\lg\lg N)$ priority queue but have not finished optimizing it. The constants are roughly 250 Sparc instructions per operation. 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 $N < 10000$.

7. Simulations

We used a stripped down version of the network simulator, freely available from Lawrence Berkeley Laboratories,[3] to carry out simulations. We implemented several fair-queuing algorithms and compared their performance to Leap Forward Virtual Clock scheme. The simulator allows us to examine packets that go through a link, and calculate delays of individual packets and overall bandwidth seen by each flow.

Due to space constraints, we show only one of the simulation results in Table 7 where we compare the delay experienced by different flows under four different fair queuing schemes. Average delay experienced by flows under LFVC are similar to those under VC and much smaller than those under DRR and SCFQ. Other results [15] illustrate the merits of LFVC over other fair queuing schemes.

Based on our experiments, we observe: **1)** LFVC rations the output link fairly among contending flows. The leap forward step prevents credit accumulation. **2)** Flows sending packets at or under their reserved rate experience very small delay in LFVC but the delay suffered by ill-behaved flows (on average) appears to be worse in Leap Forward. **3)** LFVC does not cause bursty behavior as can be exhibited by both SCFQ and WFQ.

---

[3] `http://www-nrg.ee.lbl.gov/ns/`

[4] The rate allocations in this experiment are borrowed from [14]; but the flows here send data at a constant bit rate as opposed to an on-off model used in their experiments.

Flow	Reserved rate[4]	arrival rate	Delay (seconds)			
			DRR	SCFQ	VC	LFVC
0	0.5	0.498	0.00143	0.00112	0.000213	0.000213
1	0.0625	0.1	0.368	0.365	0.382	0.382
2	0.0625	0.062	0.00242	0.00526	0.00186	0.00203
3	0.0625	0.061	0.00243	0.00535	0.002	0.00211
4	0.078125	0.076	0.00307	0.00404	0.000881	0.000861
5	0.078125	0.076	0.0035	0.00354	0.0025	0.00228
6	0.078125	0.076	0.00371	0.00335	0.00217	0.00229
7	0.078125	0.076	0.00392	0.00356	0.00114	0.00125

Table 3: Average delay experienced by flows under Deficit Round Robin (DRR), Virtual Clock, Self-Clocked, and Leap Forward. In this experiment we use constant bit rate flows with a packet size of 53 bytes. All flows except flow 1 are operating under their reserved rate. All these queuing algorithms provide fair sharing of the bandwidth. Since the columns representing throughput used by each flow under different schemes are identical to their allocated rate, they are not enumerated here. On the other hand, the delay experienced by flow 0 for example, varies widely across the different queuing disciplines. DRR and SCFQ do not provide good delay bounds, as can be seen from the table. Average delay experienced by flows under LFVC are similar to those under VC and much smaller than those under DRR and SCFQ.

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 goes well beyond Virtual Clock by adding two non-trivial modifications: a quarantine mechanism for temporarily holding oversubscribed flows, and a leap forward mechanism for avoiding deadlock when all flows are oversubscribed. The resulting scheme provides throughput fairness, even in a worst-case sense, without compromising delay bounds. The algorithm is elegant and simple to implement, and thus should have practical appeal.

Second, we have formulated two necessary and sufficient conditions for providing delay and throughput guarantees in a clock-based scheme. These conditions lead to new insight into the fair queuing problem, and allow algorithm designers to balance the delay and throughput requirements in appropriate ways to suit their taste and needs. Our technique of maintaining a throughput invariant is quite different from the one used by Bennett and Zhang [1], although both lead to low WFI. Our analysis of the delay at a single server is based on an extremely intuitive Backlog Lemma; our analysis seems much simpler than previous delay analyses of Virtual Clock schemes. We believe that, with appropriate modifications, similar analysis is possible for other schemes as well (PGPS, SCFQ etc.)

Third, we have introduced the idea of trading off delay bounds for reduced computation by introducing an allowable error term ρ. By rounding up tags to multiples of ρ, we allow the use of efficient priority queues. Even with small values of ρ, we can realize an exponential speedup in packet processing. Thus a theoretical contribution of our paper is to show that $\Omega(\log N)$ processing per packet is *not necessary* for achieving latency and throughput 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 "rounded" 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 reasonably small (insert, delete, and successor/findmin operations cost between 200 and 300 Sparc instructions). We believe that with further optimization we can rival more conventional scheme (such as tries), for even small values of N. Tag rounding 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 the 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 have already started to implement several different data structures to test and compare their relative performance.

Leap Forward Virtual Clock scheme can be considered a *generalization* of the classical virtual clock and self-clocked schemes. If we *never* leap forward, we get virtual clock; we always leap forward to the tag of the current packet in service, we get Self-Clocked. We also improve on Frame Based Queuing [14] and Worst-case Weighted Fair Queuing [1] by providing $O(\log\log N)$ computation instead of $O(\log N)$. While our coarsening ideas may be applicable to these schemes, a careful proof is required of the effect of coarsening on these schemes.

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 using the idea of *delay regulators* [7]. Our quarantine data structure L closely resembles a delay regulator.

Our work on fair-queuing is continuing. Our immediate goal is to use the Leap Forward discipline as a component in a Hierarchical Fair Queuing system. This would allow the link bandwidth to be partitioned into multiple classes (representing different organizations), which can be further subdivided into users. Bennett and Zhang [2] show that providing a low WFI is critical in a hierarchical fair queuing system. As such our leap forward mechanism should be useful. We also plan to work on our coarsening ideas to achieve further simplifications and improved time bounds for fair queuing. Plans for implementing Leap Forward Virtual Clock scheme in a real router testbed are under consideration.

References

[1] J. Bennett and H. Zhang Worst-case Fair Weighted Fair Queuing. *Proc. INFOCOM*, 1995.

[2] J. Bennett and H. Zhang Hierarchical Packet Fair Queuing Algorithms. To appear in *Proc. SIGCOMM '96.*

[3] R. Brown Calendar Queues: a fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM, Vol 31,10, October 1988.*

[4] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. *MIT Press, 1990.*

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

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

[7] 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.

[8] 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.

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

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

[11] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control: The Single Node Case. In *Proc. of IEEE INFOCOM*, pp. 915–924, 1992.

[12] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. In *Proc. of IEEE INFOCOM*, pp. 521–530, 1993.

[13] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proc. of SIGCOMM*, pp. 231–242, 1995.

[14] 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 Engineering and Information Sciences, UC Santa Cruz*, July 1995.

[15] S. Suri, G. Varghese and G. Chandranmenon. Leap Forward Virtual Clock: A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness. *Technical Report, Dept. of Computer Science, Washington University, Saint Louis*, 1996. Use the WWW link: `www.cs.wustl.edu/~suri/leapforward.ps` to access the report.

[16] 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.

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