Efficient Fair Queuing Using Deficit Round-Robin

M. Shreedhar and George Varghese, *Member, IEEE*

***Abstract*— Fair queuing is a technique that allows each flow passing through a network device to have a fair share of network resources. Previous schemes for fair queuing that achieved nearly perfect fairness were expensive to implement; specifically, the work required to process a packet in these schemes was $O(\log(n))$, where n is the number of active flows. This is expensive at high speeds. On the other hand, cheaper approximations of fair queuing reported in the literature exhibit unfair behavior. In this paper, we describe a new approximation of fair queuing, that we call *deficit round-robin*. Our scheme achieves nearly perfect fairness in terms of throughput, requires only $O(1)$ work to process a packet, and is simple enough to implement in hardware. Deficit round-robin is also applicable to other scheduling problems where servicing cannot be broken up into smaller units (such as load balancing) and to distributed queues.**

I. Introduction

WHEN THERE is contention for resources, it is important for resources to be allocated or scheduled fairly. We need firewalls between contending users, so that the fair allocation is followed strictly. For example, in an operating system, CPU scheduling of user processes controls the use of CPU resources by processes, and insulates well-behaved users from ill-behaved users. Unfortunately, in most computer networks, there are no such firewalls; most networks are susceptible to sources that behave badly. A rogue source that sends at an uncontrolled rate can seize a large fraction of the buffers at an intermediate router; this can result in dropped packets for other sources sending at more moderate rates. A solution to this problem is needed to *isolate* the effects of bad behavior to users that are behaving badly.

An isolation mechanism called fair queuing (FQ) [3] has been proposed, and has been proven [8] to have nearly perfect isolation and fairness. Unfortunately, FQ appears to be expensive to implement. Specifically, FQ requires $O(\log(n))$ work per packet, where n is the number of packet streams that are concurrently active at the gateway or router. With a large number of active packet streams, FQ is hard to implement at high speeds.[1] Some attempts have been made to improve the efficiency of FQ, however, such attempts either do not avoid the $O(\log(n))$ bottleneck or are unfair. We will use the capitalized "Fair Queuing" (FQ) to refer to the implementation in [3], and the uncapitalized "fair queuing" to refer to the generic idea.

In this paper, we shall define an isolation mechanism that achieves nearly perfect fairness (in terms of throughput), and takes $O(1)$ processing work per packet. Our scheme is simple (and, therefore, inexpensive) to implement at high speeds at a router or gateway. Furthermore, we provide analytical results that do not depend on assumptions about traffic distributions.

Flows: Our intent is to provide firewalls between different packet streams. We formalize the intuitive notion of a packet stream using the more precise notion of a *flow* [18]. A flow has two properties:

- A flow is a stream of packets that traverses the same route from the source to the destination and requires the same grade of service at each router or gateway in the path.
- In addition, every packet can be uniquely assigned to a flow using prespecified fields in the packet header.

The notion of a flow is quite general and applies to datagram networks (e.g., IP, OSI) and virtual circuit networks (e.g., X.25, ATM). For example, in a virtual circuit network, a flow could be identified by a virtual circuit identifier (VCI). On the other hand, in a datagram network, a flow could be identified by packets with the same source–destination addresses.[2] While source and destination addresses are used for routing, we could discriminate flows at a finer granularity by also using port numbers (which identify the transport layer session) to determine the flow of a packet. For example, this level of discrimination allows a file transfer connection between source A and destination B to receive a larger share of the bandwidth than a virtual terminal connection between A and B.

As in all fair queuing variants, our solution can be used to provide fair service to the various flows that thread a router, regardless of the way a flow is defined.

Organization: The rest of the paper is organized as follows. In the next section, we review the relevant previous work. A new technique for avoiding the unfairness of round-robin scheduling called *deficit round-robin* is described in Section III. Round-robin scheduling [13] can be unfair if different flows use different packet sizes; our scheme avoids this problem by keeping state, per flow, that measures the deficit or past unfairness. We analyze the behavior of our scheme using both analysis and simulation in Sections IV and V. Basic deficit round-robin provides throughput in terms of fairness but provides no latency bounds. In Section VI, we describe how to augment our scheme to provide latency bounds.

Manuscript received August 9, 1995; revised November 11, 1995; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Crowcroft. This work was supported by the National Science Foundation under Grant NCR-940997.

M. Shreedhar was with Washington University, St. Louis, MO 63130 USA. He is now with Microsoft Corporation, Redmond, WA, 98052 USA (e-mail: shreem@microsoft.com).

G. Varghese is with the Department of Computer Science, Washington University, St. Louis, MO 63130 USA (e-mail: varghese@askew.wustl.edu).

Publisher Item Identifier S 1063-6692(96)04215-X.

[1] Alternately, while hardware architectures could be devised to implement FQ, this will probably drive up the cost of the router.

[2] Note that a flow might not always traverse the same path in datagram networks, since the routing tables can change during the lifetime of a connection. Since the probability of such an event is low, we shall assume that it traverses the same path during a session.

1063–6692/96$05.00 © 1996 IEEE

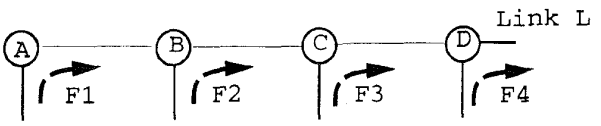


Fig. 1. The parking lot problem.

II. Previous Work

Existing Routers: Most routers use first-come first-serve (FCFS) service on output links. In FCFS, the order of arrival completely determines the allocation of packets to output buffers. The presumption is that congestion control is implemented by the source. In feedback schemes for congestion control, connections are supposed to reduce their sending rate when they sense congestion. However, a rogue flow can keep increasing its share of the bandwidth and cause other (well-behaved) flows to reduce their share. With FCFS queuing, if a rogue connection sends packets at a high rate, it can capture an arbitrary fraction of the outgoing bandwidth. This is what we want to prevent by building firewalls between flows.

Typically, routers try to enforce some amount of fairness by giving fair access to traffic coming on different input links. However, this crude form of resource allocation can produce exponentially bad fairness properties as shown below.

In Fig. 1 for example, assume that all four flows F1–F4 wish to flow through link L to the right of node D, and that all flows always have data to send. If node D does not discriminate flows, node D can only provide fair treatment by alternately serving traffic arriving on its input links. Thus, flow F4 gets half the bandwidth of link L and all other flows combined get the remaining half. A similar analysis at C shows that F3 gets half the bandwidth on the link from C to D. Thus, without discriminating flows, F4 gets 1/2 the bandwidth of link L, F3 gets 1/4 of the bandwidth, F2 gets 1/8 of the bandwidth, and F1 gets 1/8 of the bandwidth. In other words, the portion allocated to a flow can drop exponentially with the number of hops that the flow must traverse. This is sometimes called the *parking lot* problem because of its similarity to a crowded parking lot with one exit.

Nagle's solution: In Fig. 1, the problem arose because the router allocated bandwidth based on input links. Thus, at router D, F4 is offered the same bandwidth as flows F1, F2, and F3 combined. It is unfair to allocate bandwidth based on topology. A better idea is to distinguish flows at a router and treat them separately.

Nagle [13] proposed an approximate solution to this problem for datagram networks by having routers discriminate flows and then providing round-robin service to flows for every output link. Nagle proposed identifying flows using source-destination addresses and using separate output queues for each flow; the queues are serviced in round-robin fashion. This prevents a source from arbitrarily increasing its share of the bandwidth. When a source sends packets too quickly, it merely increases the length of its own queue. An ill-behaved source's packets will get dropped repeatedly.

Despite its merits, there is a flaw in this scheme. It ignores packet lengths. The hope is that the average packet size over the duration of a flow is the same for all flows; in this case, each flow gets an equal share of the output link bandwidth. However, in the worst case, a flow can get *max/min* times the bandwidth of another flow, where *max* is the maximum packet size and *min* is the minimum packet size.

Fair Queuing: Demers *et al.* devised an ideal algorithm called *bit-by-bit round-robin* (BR) that solves the flaw in Nagle's solution. In the BR scheme, each flow sends one bit at a time in round-robin fashion. Since it is impossible to implement such a system, they suggest approximately *simulating* BR. To do so, they calculate the time when a packet would have left the router using the BR algorithm. The packet is then inserted into a queue of packets sorted on departure times. Unfortunately, it is expensive to insert into a sorted queue. The best known algorithms for inserting into a sorted queue require $O(\log(n))$ time, where n is the number of flows. While the BR guarantees fairness [8], the packet processing cost makes it hard to implement cheaply at high speeds.

A naive FQ server would require $O(\log(m))$, where m is the number of packets in the router. However, Keshav [11] shows that only one entry per flow need be inserted into a sorted queue. This still results in $O(\log(n))$ overhead. Keshav's other implementation ideas [11] take at least $O(\log(n))$ time in the worst case.

Stochastic Fair Queuing (SFQ): SFQ was proposed by McKenney [12] to address the inefficiencies of Nagle's algorithm. McKenney uses hashing to map packets to corresponding queues. Normally, one would use hashing with chaining to map the flow ID in a packet to the corresponding queue. One would also require one queue for every possible flow through the router. McKenney, however, suggests that the number of queues be considerably less than the number of possible flows. All flows that happen to hash into the same bucket are treated equivalently. This simplifies the hash computation [hash computation is now guaranteed to take $O(1)$ time], and allows the use of a smaller number of queues. The disadvantage is that flows that collide with other flows will be treated unfairly. The fairness guarantees are probabilistic, hence, the name *stochastic fair queuing*. However, if the size of the hash index is sufficiently larger than the number of active flows through the router, the probability of unfairness will be small. Notice that the number of queues need only be a small multiple of the number of *active flows* (as opposed to the number of *possible* flows, as required by Nagle's scheme).

Queues are serviced in round-robin fashion, without considering packet lengths. When there are no free buffers to store a packet, the packet at the end of the longest queue is dropped. McKenney shows how to implement this buffer-stealing scheme in $O(1)$ time using bucket sorting techniques. Notice that buffer stealing allows better buffer utilization as buffers are essentially shared by all flows. The major contributions of McKenney's scheme are the buffer stealing algorithm, and the idea of using hashing and ignoring collisions. However, this scheme does nothing about the inherent unfairness of Nagle's round-robin scheme.

Other Relevant Work: Golestani introduced [9] a fair queuing scheme, called self-clocked fair queuing. This scheme uses a virtual time function that makes computation of the departure

times simpler than in ordinary Fair Queuing. However, this approach retains the $O(\log(n))$ sorting bottleneck.

Together with weighted fair queuing, a pioneering approach to queue management is the virtual clock approach of Zhang [18]. Delay bounds based on this queuing discipline have recently been discovered [18]. However, the approach still has the computational cost associated with sorting.

V. Jacobson and S. Floyd have proposed a resource allocation scheme called class-based queuing that has been implemented. In the context of that scheme, and independent of our work, S. Floyd has proposed a queuing algorithm [5]–[7] that is similar to our deficit round-robin scheme described below. Her work does not have our theorems about throughput properties of various flows, however, it does have interesting results on delay bounds and also considers the more general case of multiple priority classes.

A recent paper [15] has (independently) proposed a similar idea to our scheme: in the context of a specific local area network (LAN) protocol (DQDB) they propose keeping track of remainders across rounds. Their algorithm is, however, mixed in with a number of other features needed for DQDB. We believe that we have cleanly abstracted the problem, thus, our results are simpler and applicable to a variety of contexts.

A paper by Parekh and Gallagher [14] showed that Fair Queuing could be used together with a leaky bucket admission policy to provide delay guarantees. This showed that FQ provides more than isolation; it also provides end-to-end latency bounds. While it increased the attractiveness of FQ, it provided no solution for the high overhead of FQ.

III. Deficit Round-Robin

Ordinary round-robin servicing of queues can be done in constant time. The major problem, however, is the unfairness caused by possibly different packet sizes used by different flows. We now show how this flaw can be removed, while still requiring only constant time. Since our scheme is a simple modification of round-robin servicing, we call our scheme *deficit round-robin*.

We use stochastic fair queuing to assign flows to queues. To service the queues, we use round-robin servicing with a quantum of service assigned to each queue; the only difference from traditional round-robin is that if a queue was not able to send a packet in the previous round because its packet size was too large, the remainder from the previous quantum is added to the quantum for the next round. Thus, deficits are kept track off; queues that were shortchanged in a round are compensated in the next round.

In the next few sections, we will describe and precisely prove the properties of deficit round-robin schemes. We start by defining the figures of merit used to evaluate different schemes.

Figures of Merit: Currently, there is no uniform figure of merit defined for fair queuing algorithms. We define two measures: FM (that measures the fairness of the queuing discipline) and *work* (that measures the time complexity of the queuing algorithm). Similar fairness measures have been defined before, but no definition of work has been proposed. It is important to have measures that are not specific to deficit round-robin so that they can be applied to other forms of fair queuing.

To define the work measure, we assume the following model of a router. We assume that packets sent by flows arrive to an enqueue process that queues a packet to an output link for a router. We assume there is a dequeue process at each output link that is active whenever there are packets queued for the output link; whenever a packet is transmitted, this process picks the next packet (if any) and begins to transmit it. Thus, the work to process a packet involves two parts: enqueuing and dequeuing.

Definition 1: Work is defined as the maximum of the time complexities to enqueue or dequeue a packet.

For example, if a fair queuing algorithm takes $O(\log(n))$ time to enqueue a packet and $O(1)$ time to dequeue a packet, we say that the *work* of the algorithm is $O(\log(n))$.

We will use a throughput fairness measure FM due to Golestani [9], which measures the worst case difference between the normalized service received by different flows that are backlogged during any time interval. Clearly, it makes no sense to compare a flow that is not backlogged with one that is, because the former does not receive any service when it is not backlogged. If the fairness measure is very small, this amounts to saying that the the service discipline closely emulates a bit-by-bit round-robin server [3], which is considered an ideal fair queueing system. Note that if the service discipline is idealized in a fluid-flow model to offer arbitarily small increments of service, then FM becomes zero.

Definition 2: A flow is backlogged during an interval I of an execution if the queue for flow i is never empty during interval I.

We assume there is some quantity f_i, settable by a manager, that expresses the ideal share to be obtained by flow i. Let $\text{sent}_i(t_1, t_2)$ be the total number of bytes sent on the output line by flow i in the interval (t_1, t_2). Fix an execution of the DRR scheme. We can now express the fairness measure of an interval (t_1, t_2) as follows. We define it to be the worst case [across all pairs of flows i and j that are backlogged during (t_1, t_2)], of the difference in the normalized bytes sent for flows i and j during (t_1, t_2).

Definition 3: Let $FM(t_1, t_2)$ be the maximum, over all pairs of flows i, j that are backlogged in the interval (t_1, t_2), of $(\text{sent}_i(t_1, t_2)/f_i - \text{sent}_j(t_1, t_2)/f_j)$. Define FM to be the maximum value of $FM(t_1, t_2)$ over all possible executions of the fair queueing scheme and all possible intervals (t_1, t_2) in an execution.

Finally, we can define a service discipline to be fair if FM is a small constant. In particular, $FM(t_1, t_2)$ should not depend on the size of the interval [9].

Algorithm: We propose an algorithm for servicing queues in a router called *deficit round-robin* (Figs. 2–3). We will assume that the quantities f_i, which indicate the share given to flow i, are specified as follows.[3] We assume that each flow i is allocated Q_i worth of bits in each round. Define

[3] More precisely, this is the share given to queue i and to all flows that hash into this queue. However, we will ignore this distinction until we incorporate the effects of hashing.

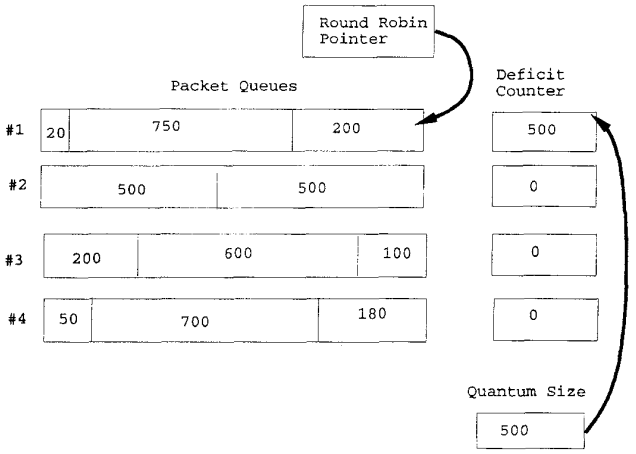


Fig. 2. Deficit round-robin: At the start, all the DC variables are initialized to zero. The round-robin pointer points to the top of the active list. When the first queue is serviced, the Q value of 500 is added to the DC value. The remainder after servicing the queue is left in the DC variable.

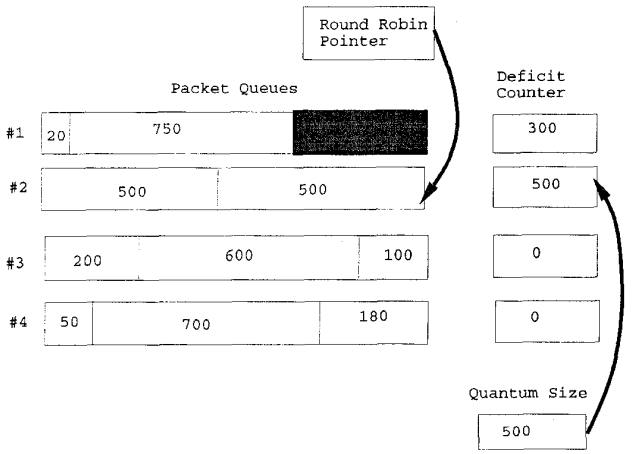


Fig. 3. Deficit round-robin (2): After sending out a packet of size 200, the queue had 300 bytes of its quantum left. It could not use it the current round, since the next packet in the queue is 750 bytes. Therefore, the amount 300 will carry over to the next round when it can send packets of size totaling 300 (deficit from previous round) + 500 (quantum).

$Q = \text{Min}_i(Q_i)$. The share f_i allocated to flow i is simply Q_i/Q. Finally, since the algorithm works in rounds, we can measure time in terms of rounds. A round is one round-robin iteration over the queues that are backlogged.

Packets coming in on different flows are stored in different queues. Let the number of bytes sent out for queue i in round k be $\text{bytes}_i(k)$. Each queue i is allowed to send out packets in the first round subject to the restriction that $\text{bytes}_i(1) \leq Q_i$. If there are no more packets in queue i after the queue has been serviced, a state variable called DC_i is reset to zero. Otherwise, the remaining amount $(Q_i - \text{bytes}_i(k))$ is stored in the state variable DC_i. In subsequent rounds, the amount of bandwidth usable by this flow is the sum of DC_i of the previous round added to Q_i. Pseudo-code for this algorithm is shown in Fig. 4.

To avoid examining empty queues, we keep an auxiliary list *ActiveList* that is a list of indices of queues that contain at least one packet. Whenever a packet arrives to a previously empty queue i, i is added to the end of *ActiveList*. Whenever index i is at the head of *ActiveList*, the algorithm services up to $Q_i + DC_i$ worth of bytes from queue i; if at the end of this service opportunity, queue i still has packets to send, the index i is moved to the end of *ActiveList*; otherwise, DC_i is set to zero and index i is removed from *ActiveList*.

In the simplest case $Q_i = Q_j$ for all flows i, j. Exactly as in weighted fair queuing [3], however, each flow i can ask for a larger relative bandwidth allocation and the system manager can convert it into an equivalent value of Q_i. Clearly, if $Q_i = 2Q_j$, the manager intends that flow i get twice the bandwidth of flow j when both i and j are active.

Comparing DRR with BR: The reader may be tempted to believe that DRR is just a crude approximation of BR. For instance, the reader may suspect that when the quantum size is one, the two schemes are identical. This plausible conjecture is incorrect.

Consider an example. Suppose $n - 1$ flows have large packets of size *Max* and the nth flow is empty. Even with a quantum size of one bit, the deficit counter will eventually count up to $\text{Max} - 1$ (after $\text{Max} - 1$ scans of the $n - 1$ queues). Now assume that a small one-bit packet arrives to the nth flow queue. On the next scan, DRR will allow all other flows to send a maximum sized packet before sending the one-bit packet of the nth flow. Thus, even with one bit quanta, the maximum delay suffered by a one-bit packet (once it comes to the head of the queue) can be as bad as $(n - 1) * \text{Max}$, while in bit-by-bit, it can never be worse than $n - 1$ bit delays. Thus, DRR is off by a multiplicative factor in delays and the two schemes are not identical.

IV. Analytical Results

We begin with an invariant that is true for all executions of the DRR algorithm (not just for the backlogged intervals that are used to evaluate fairness). Recall that an invariant is meant to be true after every program action; it is not required to be true in the middle of a program action.

Lemma 1: For all i, the following invariant holds for every execution of the DRR algorithm: $0 \leq DC_i < \text{Max}$.

Proof: Initially, $DC_i = 0 \Rightarrow DC_i < Q_i$. Notice that DC_i only changes value when queue i is serviced. During a round, when the servicing of queue i completes, there are two possibilities:

- If a packet is left in the queue for flow i, then it must be of size strictly greater than DC_i. Also, by definition, the size of any packet is no more than Max, thus, DC_i is strictly less than *Max*. Also, the code guarantees that $DC_i \geq 0$.
- If no packets are left in the queue, the algorithm resets DC_i to zero.

□

The router services the queues in a round-robin manner according to the DRR algorithm defined earlier. A round is one round-robin iteration over the queues that are backlogged.

We first show that during any period in which a flow i is backlogged, the number of bytes sent on behalf of flow i is

roughly equal to $m \cdot Q_i$, where m is the number of round-robin service opportunities received by flow i during this interval.

Lemma 2: Consider any execution of the DRR scheme and any interval (t_1, t_2) of any execution such that flow i is backlogged during (t_1, t_2). Let m be the number of round-robin service opportunities received by flow i during the interval (t_1, t_2). Then

$$m \cdot Q_i - \text{Max} \leq \text{sent}_i(t_1, t_2) \leq m \cdot Q_i + \text{Max}.$$

Proof: We start with some definitions. Let us use the term *round* to denote service opportunities received by flow i within the interval (t_1, t_2)). Number these rounds sequentially starting from one and ending with round m. For notational convenience, we regard t_1, the start of the interval, as the end of a hypothetical round zero.

Let $DC_i(k)$ be the value of DC_i for flow i at the end of round k. Let $\text{bytes}_i(k)$ be the bytes sent by flow i in round k. Let $\text{sent}_i(k)$ be the bytes sent by flow i in rounds one through k. Thus, $sent_i(m) = \Sigma_{k=1}^{m} \text{bytes}_i(k)$.

The main observation (which follows immediately from the protocol) is: $\text{bytes}_i(k) + DC_i(k) = Q_i + DC_i(k-1)$. We use the assumption that flow i always has a backlog in the above equation. Thus, in round k, the total allocation to flow i is $Q_i + DC_i(k-1)$. Thus, if flow i sends $\text{bytes}_i(k)$, then the remainder will be stored in $DC_i(k)$, because queue i never empties during the interval (t_1, t_2). This equation reduces to

$$\text{bytes}_i(k) = Q_i + DC_i(k-1) - DC_i(k).$$

Summing the last equation over m rounds of servicing of flow i, we get a telescoping series. Since $\text{sent}_i(m) = \Sigma_{k=1}^{m} \text{bytes}_i(k)$ we get

$$\text{sent}_i(m) = m \cdot Q_i + DC_i(0) - DC_i(m).$$

The lemma follows because the value of DC_i is always non-negative and $\leq \text{Max}$ (using Lemma 1). □

The following theorem establishes the fact that the fairness measure for any interval is bounded by a small constant.

Theorem 3: For an interval (t_1, t_2) in any execution of the DRR service discipline

$$FM(t_1, t_2) \leq 2\text{Max} + Q, \quad \text{where} \quad Q = \text{Min}_i(Q_i).$$

Proof: Consider any interval (t_1, t_2) in any execution of DRR and any two flows i and j that are backlogged in this interval.

A basic invariant of the DRR algorithm (Fig. 4) is that during any interval in which two flows i and j are backlogged, between any two round-robin opportunities given to flow i, flow j must have had a round-robin opportunity. This is easy to see because at the end of a flow i opportunity, the index i is put at the rear of the active queue. Since flow j is backlogged, index j is in the active queue and, thus, flow j will be served before flow i is served again. Thus, if we let m be the number of round-robin opportunities given to flow i in interval (t_1, t_2) and if we let m' be the number of round-robin opportunities given to flow j in the same interval, then $|m - m'| \leq 1$.

Thus, from Lemma 2, we get

$$\text{sent}_i(t_1, t_2) \leq m \cdot Q_i + \text{Max}.$$

```
Consider any output link for a given router.
Queue_i is the ith queue, which stores packets
with flow id i. Queues are numbered 0 to (n - 1),
n is the maximum number of output link queues.

Enqueue(), Dequeue() are standard Queue operators.
We use a list of active flows, ActiveList, with
standard operations like InsertActiveList, which adds
a flow index to the end of the active list.
FreeBuffer() frees a buffer from the flow with the
longest queue using using McKenney's buffer stealing.
Q_i is the quantum allocated to Queue_i.
DC_i contains the bytes that Queue_i did not
use in the previous round.

Initialization:
  For (i = 0; i < n; i = i + 1)
    DC_i = 0;

Enqueuing module: on arrival of packet p
  i = ExtractFlow(p)
  If (ExistsInActiveList(i) == FALSE) then
    InsertActiveList(i); (*add i to active list*)
    DC_i = 0;
  If no free buffers left then
    FreeBuffer(); (* using buffer stealing *)
  Enqueue(i, p); (* enqueue packet p to queue i*)

Dequeuing module:
  While(TRUE) do
    If ActiveList is not empty then
      Remove head of ActiveList, say flow i
      DC_i = Q_i + DC_i;
      while ((DC_i > 0) and
            (Queue_i not empty)) do
        PacketSize = Size(Head(Queue_i));
        If (PacketSize ≤ DC_i) then
          Send(Dequeue(Queue_i));
          DC_i = DC_i
            - PacketSize;
        Else break; (*skip while loop *)
      If (Empty(Queue_i)) then
        DC_i = 0;
      Else InsertActiveList(i);
```

Fig. 4. Code for deficit round-robin.

Thus

$$\text{sent}_i(t_1, t_2) \leq (m-1) \cdot Q_i + Q_i + \text{Max}.$$

From the definition, f_i, the share given to any flow i, is equal to Q_i/Q. Thus, we can calculate the normalized service received by i as

$$\text{sent}_i(t_1, t_2)/f_i \leq (m-1) \cdot Q + Q + \text{Max}/f_i$$

since $Q_i = f_i Q$. Recall that Q is the smallest value of Q_i over all flows i. Similarly, we can show for flow j (using

Lemma 2) that

$$\text{sent}_j(t_1, t_2) \geq m' \cdot Q_i - \text{Max}$$

and so

$$\text{sent}_j(t_1, t_2)/f_j \geq m' \cdot Q - \text{Max}/f_j.$$

Subtracting the equations for the normalized service for flows i and j, and using the fact that $m' \geq m - 1$, we get

$$\frac{\text{sent}_i(t_1, t_2)}{f_i} - \frac{\text{sent}_j(t_1, t_2)}{f_j} \leq Q + \frac{\text{Max}}{f_j} + \frac{\text{Max}}{f_i} \quad (1)$$

The theorem follows because both f_i and f_j are ≥ 1 for DRR. □

Having dealt with the fairness properties of DRR, we analyze the worst-case packet processing work. It is easy to see that the size of the various Q variables in the algorithm determines the number of packets that can be serviced from a queue in a round. This means that the latency for a packet (at low loads) and the throughput of the router (at high loads) is dependent on the value of the Q variables.

Theorem 4: The work for deficit round-robin is $O(1)$, if for all $i, Q_i \geq \text{Max}$.

Proof: Enqueuing a packet requires finding the queue used by the flow [$O(1)$ time complexity using hashing since we ignore collisions], appending the packet to the tail of the queue, and possibly stealing a buffer ($O(1)$ time using the technique in [12]). Dequeuing a packet requires determining the next queue to service by examining the head of *ActiveList*, and then doing a constant number of operations (per packet sent from the queue) in order to update the deficit counter and *ActiveList*. If $Q \geq \text{Max}$, we are guaranteed to send at least one packet every time we visit a queue and, thus, the worst-case time complexity is $O(1)$. □

Note that if we use hashing and we do not ignore collisions, then *Work* for DRR becomes $O(1)$ expected [as opposed to O (1) worst case] because of possible collisions.

A. Comparison to Other Fair Queuing Schemes

Golestani [9] states the following result for the fairness measure of self-clocked Fair Queueing (for any time interval (t_1, t_2))

$$\text{sent}_i(t_1, t_2)/f_i - \text{sent}_j(t_1, t_2)/f_j \leq \text{Max}/f_j + \text{Max}/f_i.$$

On the other hand, (1) shows that for DRR

$$\text{sent}_i(t_1, t_2)/f_i - \text{sent}_j(t_1, t_2)/f_j \leq Q + \text{Max}_j/f_j + Max_i/f_i.$$

Thus, the only difference in the fairness measure is the additive term of Q caused by DRR. Since we need $Q \geq Max$ to make the work complexity $O(1)$, this translates into an additive term of *Max*, assuming $Q = Max$.

Let us call the Demers–Keshav–Shenker scheme [3] DKS Fair Queuing. In summary: DKS Fair Queuing has a maximum value of FM of *Max*; self-clocked Fair Queuing has a maximum value of FM of $2Max$, and DRR fair queuing has a maximum value of FM of $3Max$. In all three cases, the fairness measure is a small constant that does not depend on the interval size, and becomes negligible for large intervals. Thus, the small extra discrepancy caused by DRR in throughput fairness seems insignificant.

TABLE I
PERFORMANCE OF FAIR QUEUING ALGORITHMS

Algorithm	Fairness Measure	Work Complexity
Round Robin ([Nag87])	∞	O(1) expected
Fair Queuing ([DKS89])	*Max*	O(log(n))
Self-clocked Fair Queuing	$2Max$	O(log(n))
Deficit Round Robin	$3Max$	O(1) expected

We compare the FM and *Work* of the major fair queuing algorithms that have been proposed, until now, in Table I. For this comparison only, assume that DRR does hashing but does not incorporate collisions. This is the only reasonable way to compare the algorithms because the trick of using hashing and ignoring collisions (as pioneered by [12]) can be applied to *all* fair queueing algorithms. At the same time, DRR can easily be modified to treat each flow separately (as opposed to treating all flows that hash into the same bucket equivalently). We will analyze the effect of ignoring collisions in the next subsection. We have also taken the fairness measure for round-robin schemes as infinity. This is because if we consider two flows, one that uses large packets only, and a second that uses small packets only, then over any infinitely large interval, the first flow will get infinitely more service that the second. We have also taken the *Work* of round-robin to be $O(1)$ expected, because even in ordinary round-robin schemes, we need to look up the state for a flow, using say hashing.

From the table, deficit round-robin is the only algorithm that provides a fairness measure equal to a small constant and a *Work* of expected $O(1)$.

B. Incorporating Hashing

In the previous analysis, we showed that if we did not lump together flows that hashed into the same bucket, then DRR achieves an FM equal to $3Max$ and a $work = O(1)$, expected. This is a reasonable way to compare DRR to other schemes (except Stochastic Fair Queueing) that do the same thing.

On the other hand, we have argued that implementations are likely to profit from the use of McKenney's idea of using hashing and ignoring collisions. The consequence of this implementation idea is that there is now some probability that two or more flows will collide; the colliding flows will then share the bandwidth allocated to that bucket.

The average number of other flows that collide with a flow can be shown [2] to be n/Q, where n is the number of flows and Q is the number of queues. For example, if we have 1000 concurrent flows and 10 000 queues (a factor of ten, which is achievable with modest amounts of memory) the average number of collisions is 0.1. If B is the bandwidth allocated to a flow, the effective bandwidth in such a situation becomes $B/1 + (n/Q)$. For instance, with 10 000 queues and 1000 concurrent flows, this means that two backlogged flows with identical quanta can differ in terms of throughput by 10% on the avarage, in addition to the additive difference of $3Max$ guaranteed by the fairness measure. Thus, assuming a

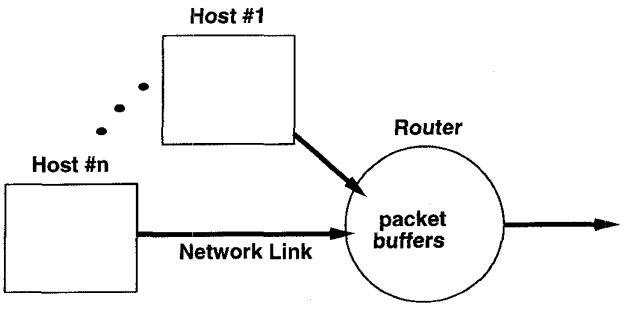


Fig. 5. Single router configuration.

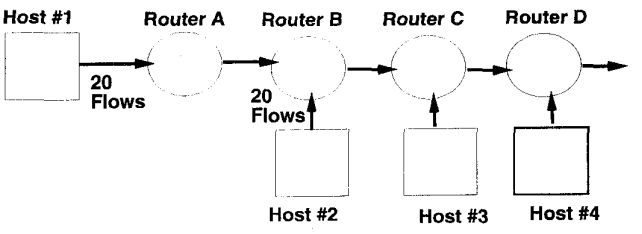


Fig. 6. Multiple router configuration.

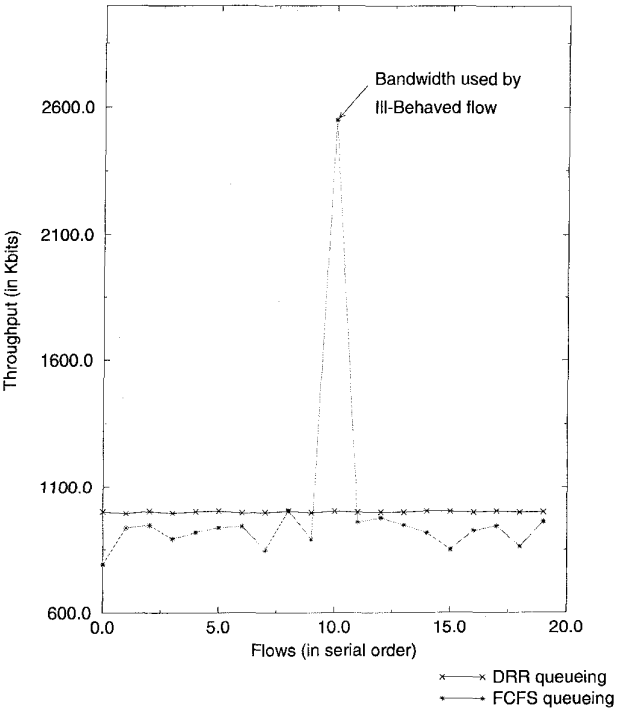


Fig. 7. This is a plot of the bandwidth offered to flows using FCFS queuing and DRR. In FCFS, the ill-behaved flow (flow 10) obtains an arbitrary share of the bandwidth. The isolation property of DRR is clearly illustrated.

reasonably large number of queues, the unfairness caused by ignoring collisions should be small.

V. Simulation Results

We wish to answer the following questions about the performance of DRR:

- We would like to confirm experimentally that DRR provides isolation superior to FCFS as the theory indicates, especially in the backlogged case.
- The theoretical analysis of DRR is for a single router (i.e., one hop). How are the results affected in a multiple hop network?
- We want to confirm that the fairness provided by DRR is still good when the flows arrive at different (not necessarily backlogged) rates and with different distributions. Is the fairness sensitive to packet size distributions and arrival distributions?

Since we have multiple effects, we have devised experiments to isolate each of these effects. However, there are other parameters (such as number of packet buffers and flow index size) that also impact performance. We first did parametric experiments to determine these values before investigating our main questions. For lack of space, we only present a few experiments and refer the reader to [16] for more details.

A. Default Simulation Settings

Unless otherwise specified, the default for all the later experiments is as specified here. We measure the throughput in terms of delivered bits in a simulation interval, typically 2000 s.[4]

In the single router case (see Fig. 5), there are one or more hosts. Each host has twenty flows, each of which generates packets at a Poisson average rate of 10 packets/s. The packet sizes are randomly selected between zero and *Max* (which is 4500 b). Ill-behaved flows send packets at three times the rate at which the other flows send packets (i.e., 30 packets/s). Each host in such an experiment is configured to have one ill-behaved flow.

In Fig. 6, we show the typical settings in a multiple hop topology. There are hosts connected at each hop (a router) and each host behaves as as described in the previous section. In multiple hop routes, where there are more than twenty flows through a router, we use large buffer sizes (around 500 packet buffers) to factor out any effects due to lack of buffers. In multiple hop topologies, all outgoing links are set at 1 Mb/s.

[4]Throughput is typically measured in b/s. However, it is easy to convert our results into b/s by dividing by the simulation time.

B. Comparison of DRR and FCFS

To show how DRR performs with respect to FCFS, we performed the following two experiments. In Fig. 7, we use a single router configuration and one host with twenty flows sending packets at the default rate through the router. The only exception is that flow 10 is a misbehaving flow. We use Poisson packet arrivals and random packet sizes (uniformly distributed between 0 and 4500 b). All parameters used are the default settings. The figures show the bandwidth offered to different flows using the FCFS and DRR queuing disciplines. We find that in FCFS the ill-behaved flow grabs an arbitrary share of bandwidth, while in DRR, there is nearly perfect fairness.

We further contrast FCFS scheduling with DRR scheduling by examining the throughput offered to each flow at different

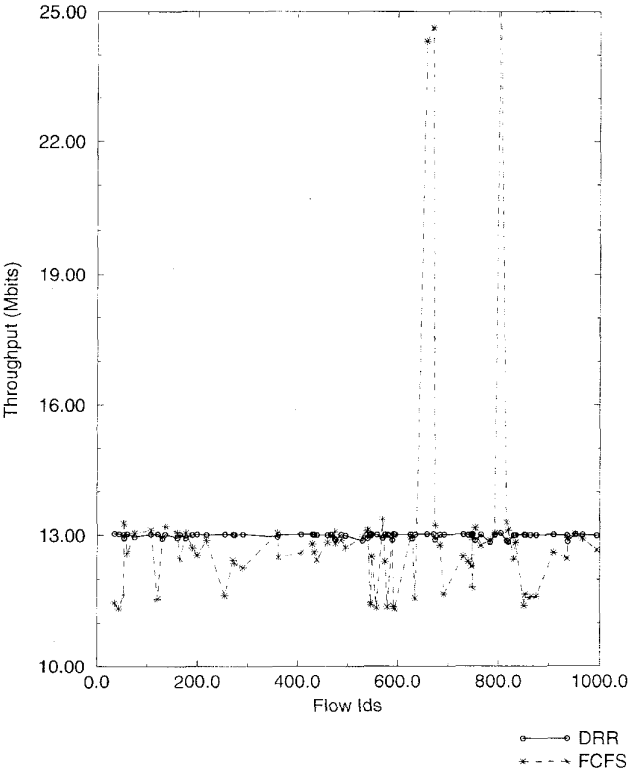


Fig. 8. The bandwidth allocated to different flows at router D using FCFS and DRR to schedule the departure of packets.

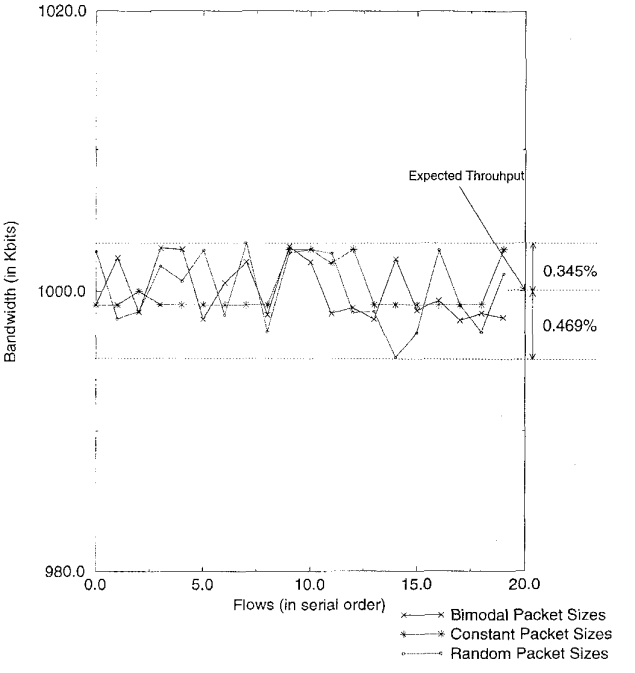


Fig. 9. The bandwidth offered to different flows with exponential interpacket times and constant, bimodal and random packet sizes.

stages in a multiple hop topology. The experimental setup is the default multiple hop topology described in Section V-A. The throughput offered is measured at router D (see Fig. 8). This time we have a number of misbehaving flows. The figure shows that DRR behaves well in multihop topologies.

C. Independence with Respect to Packet Sizes

We investigate the effect of different packet size on the fairness properties. The packet sizes in a train of packets can be modeled as random, constant or bimodal. We use a single router configuration with one host that has 20 flows. In the first experiment, we use random (uniformly distributed between 0 and 4500) packet sizes. In the next two experiments, instead of using the random packet sizes, we first use a constant packet size of 100 b, and then a bimodal size that is either 100 or 4500 b.

Figure 9 shows the lack of any particular pattern in response to the usage of different packet sizes in the packet traffic into the router. The difference in bandwidth offered to a flow while using the three different packet size distributions is negligible. The maximum deviation from this figure while using constant, random and bimodal cases turned out to be 0.3%, 0.4699%, and 0.32%, respectively. Thus, DRR seems fairly insensitive to packet size distributions.

This property becomes clearer when the DRR algorithm is contrasted with the performance of the SFQ algorithm. While using the SFQ algorithm, flows sending larger packets consistently get higher throughput than the flows sending random sized packets, while all flows get equal bandwidth while using DRR.

It is clear from Fig. 10 that DRR offers equal throughput to all kinds of sources. SFQ offers higher throughput to flows sending larger packets (for example, flow indices 44, 53, 863, 875 etc. which are flows originating from the clever host); this affects the throughput offered to normal flows (e.g. flow indices 60, 107, 190, etc. which are flows originating in the host sending random sized packets), which get substantially lower throughput.

D. Independence with Respect to Traffic Patterns

We show that DRR's performance is independent of the traffic distribution of the packets entering the router. We used two models of traffic generators: exponential and constant interarrival times, respectively, and collected data on the number of bytes sent by each of the flows. We then examined the bandwidth obtained. The other parameters (e.g. number of buffers in the router) are kept constant at sufficiently high values in this simulation.

The experiment used a single router configuration with default settings. The outgoing link bandwidth was set to 10 Kb/s. Therefore, if there are 20 input flows each sending at rates higher than 0.5 Kb/s, there is contention for the outgoing link bandwidth. We found almost equal bandwidth allocation for all flows. The maximum deviation from the average throughput offered to all flows in the constant traffic sources case is 0.3869%. In the Poisson case, it is bounded by 0.3391% from the average throughput. Thus, DRR appears to work well regardless of the input traffic distributions.

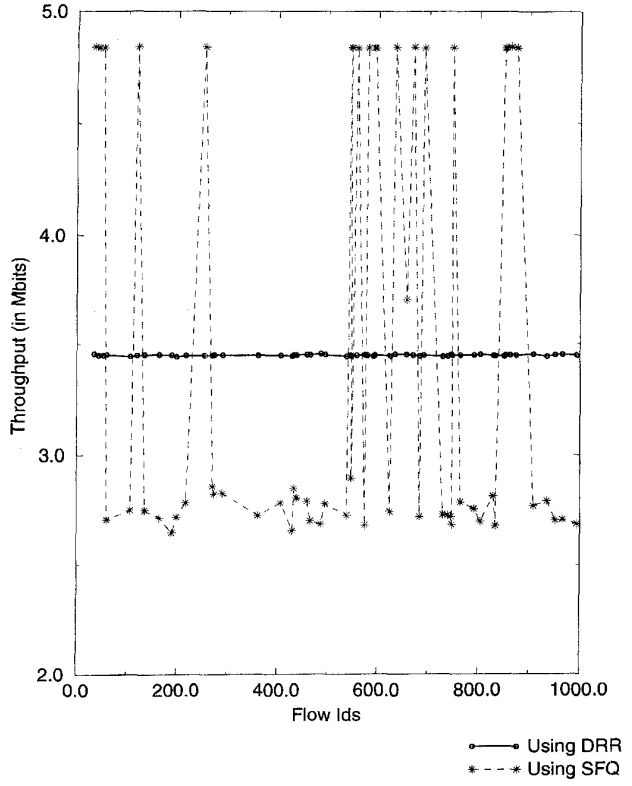


Fig. 10. Comparison of DRR and SFQ: SFQ offers higher bandwidth to flows sending large packets, while DRR offers equal bandwidth to all flows.

VI. Latency Requirements

Consider a packet p for flow i that arrives at a router. Assume that the packet is queued for an output link instantly and there are no other packets for flow i at the router. Let s be the size of packet p in bits. If we use bit-by-bit round-robin, then the packet will be delayed by s round-robin rounds. Assuming that there are no more than n active flows at any time, this leads to a latency bound of $n * s/B$, where B is the bandwidth of the output line in b/s. In other words, a small packet can only be delayed by an amount proportional to its own size by every other flow. The Demers–Keshav–Shenker (DKS) approximation only adds a small error factor to this latency bound.

The original motivation in both the work of Nagle and DKS was the notion of isolation. Isolation is essentially a throughput issue: we wish to give each flow a fair share of the overall throughput. In terms of isolation, the proofs given in the previous sections indicate that deficit round-robin is competitive with DKS Fair Queuing. However, the additional latency properties of BR and DKS have attracted considerable interest. In particular, Parekh and Gallager [14] have calculated bounds for end-to-end delay, assuming the use of DKS Fair Queuing at routers and token bucket traffic shaping at sources.

At first glance, DRR fails to provide strong latency bounds. In the example of the arrival of packet p given above, the latency bound provided by DRR is $\Sigma_i^n Q_i/B$. In other words, a small packet can be delayed by a quantum's worth by every other flow. Thus, in the case where all the quanta are equal to *Max* [which is needed to make the work $O(1)$], the ratio of the delay bounds for DRR and BR is *Max/Min*.

We note that the counter-example given in Section III indicates that reducing the quantum size does not help improve the worst-case latency bounds for DRR.

However, the real motivation behind providing latency bounds is to allow real-time traffic to have predictable and dependable performance. Since most traffic will consist of a mix of best-effort and real-time traffic, the simplest solution is to reserve a portion of the bandwidth for real-time traffic and use a separate Fair Queuing algorithm for the real-time traffic while continuing to use DRR for the best-effort traffic. This allows efficient packet processing for best-effort traffic; at the same time, it allows the use of other fair queuing schemes that provide delay bounds for real-time traffic at reasonable cost.

As a simple example of combining fair queuing schemes, consider the following modification of DRR called DRR+. In DRR+, there are two classes of flows: *latency critical* and *best-effort*. A latency critical flow must contract to send no more than x bytes in some period T. If a latency critical flow f meets its contract, whenever a packet for flow f arrives to an empty flow f queue, *the flow* f is placed at the head of the round-robin list.

Suppose, for instance, that each latency critical flow guarantees to send, at most, a single packet (of size, at most, s) every T s. Assume that T is large enough to service one packet of every latency critical flow as well as one quantum's worth for every other flow. Then if all latency critical flows meet their contract, it appears that each latency critical flow is, at most, delayed by $(n' * s) + \mathrm{Max}/B$, where n' is the number of latency critical flows. In other words, a latency critical flow is delayed by one small packet from every other latency critical flow, as well as an error term of one maximum size packet (the error term is inevitable in all schemes unless the router preempts large packets). In this simple case, the final bound appears better than the DKS bound because a latency critical flow is only delayed by other latency critical flows.

In the simple case, it is easy to police the contract for latency critical flows. A single bit that is part of the state of such a flow is cleared whenever a timer expires and is set whenever a packet arrives; the timer is reset for T time units when a packet arrives. Finally, if a packet arrives and the bit is set, the flow has violated its contract; an effective (but user-friendly) countermeasure is to place the flow ID of a deviant flow at the end of the round-robin list. This effectively moves the flow from the class of latency critical flows to the class of best-effort flows.

Figure 11 shows the results of a simulation experiment using DRR+ instead of DRR. The simulation parameters are as follows. We use a single router configuration as described earlier. The structure of a BONeS host for these experiments is as follows. Each host has 20 flows, out of which, one is marked as latency critical. Therefore, to have three *LatencyCritical* flows, we use three hosts. The default parameters are set in the DRR+ router and the hosts (The *LatencyCritical* and *BestEffort* flows send packets with exponential interarrival times at an average of 10 packets/s and they have random packet sizes). The delay experienced by the *LatencyCritical*

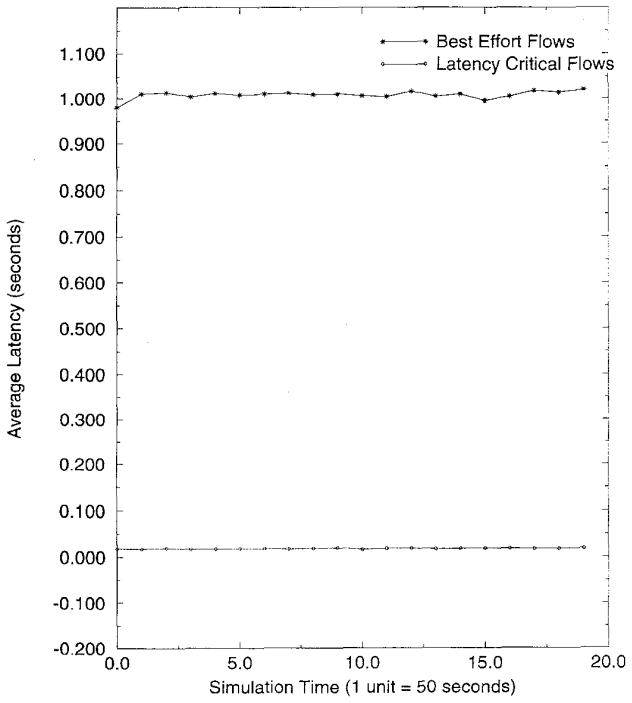


Fig. 11. The latency experienced by latency critical flows and best-effort flows when there are three latency critical flows through the router. The algorithm used for servicing the queues was DRR+.

flows is averaged into 20 batch intervals spread over the simulation time. The rest of the flows (i.e., the *BestEffort* flows) are also batched together and averaged.

It is evident from the figure that the *LatencyCritical* flows experience bounded latency in a DRR+ environment. However, note that this experiment only considered a single router. DRR+ is a simple example of combining DRR with other fair queuing algorithms to handle latency critical flows. By using other schemes for the latency critical flows, we can provide better bounds while allowing more general traffic shaping rules.

VII. Other Applications of DRR

DRR schemes can be applied to other scheduling contexts in which jobs must be serviced as whole units. In other words, jobs cannot be served in several time slices as in a typical operating system. This is true for packet scheduling because packets cannot be interleaved on the output lines, but it is true for other contexts as well. Note also that DRR is applicable to distributed queues because it needs only local information to implement. We describe two other specific applications: token ring protocols and load balancing schemes.

The current 802.5 token ring uses token holding timers that limit the number of bits a station can send at each token opportunity. If a station's packets do not fit exactly into the allowed number of bits, the remainder is not kept track off; this allows the possibility of unfairness. The unfairness can easily be removed by keeping a deficit counter at each ring node and by a small modification to the token ring protocol.

Another application is load balancing or, as it is sometimes termed, striping. Consider a router that has traffic arriving on a high speed line that needs to be sent out to a destination over multiple slower speed lines. If the router sends packets from the fast link in a round-robin fashion across the slower links, then the load may not balance perfectly if the packet sizes are highly variable. For example, if packets alternate between large and small packets, then round-robin across two lines can cause the second line to be underutilized. But load balancing is almost the inverse of fair queuing. It is is not hard to see that deficit round-robin solves the problem; we send up to to a quantum limit per output line but we keep track of deficits. This should produce nearly perfect load balancing; as usual, it can be extended to weighted load balancing. In [1], we show how to obtain perfect load balancing and yet guarantee first-in first-out (FIFO) delivery. Our load balancing scheme [1] appears to be a novel solution to a very old problem.

VIII. Conclusions

We have described a new scheme, DRR, that provides near-perfect isolation at very low implementation cost. As far as we know, this is the first fair queuing solution that provides near-perfect throughput fairness with $O(1)$ packet processing. DRR should be attractive to use while implementing fair queuing at gateways and routers.

A number of readers have conjectured that DRR should reduce to BR when the quantum size is one bit. This is not true. The two schemes have radically different behaviors. A counter-example is described in Section III. The counter-example also indicates that reducing the quantum size to be less than *Max* does not improve the worst-case latency at all. Thus DRR is not just a crude approximation of BR; it has more interesting behaviors of its own.

We have described theorems that describe the behavior of DRR in ensuring throughput fairness. In particular, we show that DRR satisfies Golestani's [9] definition of throughput fairness, i.e., the normalized bandwidth allocated to any two backlogged flows in any interval is roughly equal. Earlier versions of this paper [16], [17] had only shown throughput fairness for the case when all flows were backlogged. Our use of Golestani's definition makes the results in this paper more general, and indicates why DRR works well in nonbacklogged and backlogged scenarios. Our simulations indicate that DRR works well in all scenarios.

The Q size is required to be at least *Max* for the work complexity to be $O(1)$. We feel that while Fair Queuing using DRR is general enough for any kind of network, it is best suited for datagram networks. In ATM networks, packets are fixed size cells, therefore, Nagle's solution (simple round-robin) will work as well as DRR. However, if connections in an ATM network require weighted fair queuing with arbitrary weights, DRR will be useful.

DRR can be combined with other fair queuing algorithms such that DRR is used to service only the best-effort traffic. We described a trivial combination algorithm called DRR+ that offers good latency bounds to latency critical flows as long as they meet their contracts. However, even if the source

meets the contract, the contract may be violated due to bunching effects at intermediate routers. Thus, other combinations need to be investigated. Recall that DRR requires having the quantum size be at least a maximum packet size in order for the packet processing work to be low; this does affect delay bounds.

We believe that DRR should be easy to implement using existing technology. It only requires a few instructions beyond the simplest queuing algorithm (FCFS), and this addition should be a small percentage of the instructions needed for routing packets. The memory needs are also modest; 6K size memory should give a small number of collisions for about 100 concurrent flows. This is a small amount of extra memory compared to the buffer memory used in many routers. Note that the buffer size requirements should be identical to the buffering for FCFS because in DRR buffers are shared between queues using McKenney's buffer stealing algorithm.

DRR can be applied to other scheduling contexts in which jobs must be serviced as whole units. We have described two other applications: token rings with holding timers and load balancing. For these reasons, we believe that DRR scheduling is a general and useful tool. We hope our readers will use it in other ways.

ACKNOWLEDGMENT

The authors would like to thank several people for listening patiently to ideas and providing valuable feedback. They are: D. Clark, S. Floyd, A. Fingerhut, S. Keshav, P. McKenney, and L. Zhang. The authors also thank A. Costello, A. Fingerhut, and S. Keshav for their careful reading of this paper, and S. Keshav and the anonymous SIGCOMM referees who prodded the authors into using Golestani's fairness definition. They thank R. Gopalakrishnan and A. Dalianis for valuable discussions.

REFERENCES

[1] H. Adiseshu, G. Parulkar, and G. Varghese, "Reliable FIFO load balancing over multiple FIFO channels," Washington Univ., St. Louis, MO, Tech. Rep. 95-11, available by FTP.

[2] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA/New York: MIT Press/McGraw-Hill, 1990.

[3] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. SIGCOMM'89*, vol. 19, no. 4, Sept. 1989, pp. 1–12.

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

[5] S. Floyd, "Notes on guaranteed service in resource management," Unpublished note. 1993.

[6] ——, Personal communication. 1993.

[7] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, Aug. 95.

[8] A. Greenberg and N. Madras, "How fair is fair queueing?" in *Proc. Performance'90*, 1990.

[9] S. Golestani, "A self clocked fair queueing scheme for broadband applications," in *Proc. IEEE INFOCOMM'94*, 1994.

[10] R. Jain and S. Routhier, "Packet trains measurement and a new model for computer network traffic," *IEEE J. Select. Areas Commun.*, Sept. 1986.

[11] S. Keshav, "On the efficient implementation of fair queueing," in *Internetworking: Research and Experience*, vol. 2, Sept. 1991, pp. 157-173.

[12] P. McKenney, "Stochastic fairness queueing," in *Internetworking: Research and Experience*, vol. 2, Jan. 1991, pp. 113–131.

[13] John Nagle, "On packet switches with infinite storage," *IEEE Trans. Commun.*, vol. COM-35, no. 4, Apr. 1987.

[14] A. K. Parekh and R. G. Gallagher, "A generalized processor sharing approach to flow control in integrated services networks," in *Proc. IEEE INFOCOMM'93*, 1993.

[15] D. Saha and M. Saksena and S. Mukherjee, and S. Tripathi, "On guaranteed delivery of time-critical messages in DQDB," in *Proc. IEEE INFOCOMM'94*, 1994.

[16] M. Shreedhar, "Efficient fair queuing using deficit round robin," M.S. Thesis, Dept. of Computer Science, Washington Univ., St. Louis, MO, Dec. 1994.

[17] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round robin," Previous version of this paper, in *Proc. SIGCOMM'95*, Boston, MA, Aug. 1995.

[18] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switched networks," *ACM Trans. Comput. Syst.*, vol. 9, no. 2, pp. 101–125, May 1991.

M. Shreedhar received the B.E. degree from the Birla Institute of Technology and Science, Pilani, India, in June 1992 and the M.S. degree from Washington University, St. Louis, MO, in December 1994, both in computer science.

He currently works on Mobile Networks as part of Microsoft Corporation in Seattle, WA.

George Varghese (M'94) received the Ph.D. degree in computer science from The Massachusetts Institute of Technology, Cambridge, MA, in 1992.

He began his professional career in 1983 at Digital Equipment Corporation where he worked on designing network protocols and doing systems research as part of the DECNET architecture and advanced development group. He has worked on designing protocols and algorithms for the DECNET and GIGAswitch products. He has been an Associate Professor of Computer Science at Washington University since September 1993. His research interests are in two areas: first, applying the theory of distributed algorithms to the design of fault-tolerant protocols for real networks; second, the design of efficient algorithms to speed up protocol implementations. Together with colleagues at DEC, he has been awarded six patents, with six more patents pending.

Dr. Varghese's Ph.D. dissertation was jointly awarded the Sprowls Prize for best thesis in computer science at MIT. He was granted the ONR Young Investigator Award in 1996.