Report Number: WUCS-94-17

1994-01-01

Efficient Fair Queueing using Deficit Round Robin

George Varghese and M. Shreedhar

Fair queuing is a technique that allows each flow passing through a network device to have 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 that have been 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... **Read complete abstract on page 2.**

Efficient Fair Queueing using Deficit Round Robin

George Varghese and M. Shreedhar

Complete Abstract:

Fair queuing is a technique that allows each flow passing through a network device to have 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 that have been 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.

Efficient Fair Queueing using Deficit Round Robin

George Varghese and M. Shreedhar

WUCS-94-17

July 1994

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

Efficient Fair Queuing using Deficit Round Robin

M. Shreedhar and George Varghese *

June 30, 1994

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 that have been 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.

*Washington University in St. Louis.

1 Introduction

Whenever 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 usage of CPU resources by processes, and insulates well-behaved users from ill–behaved users. Unfortunately, in most computer networks, there are no such firewalls, and most networks are susceptible to badly–behaving sources. A rogue source that sends at an uncontrolled rate can seize a large fraction of the buffers at an intermediate gateway; this can result in packets being dropped for other sources that are sending at more moderate rates! A solution to this problem is needed to *isolate* the effects of bad behavior to the users that are behaving badly.

An isolation mechanism called Fair Queuing [DKS89] has been proposed that achieves nearly perfect isolation and fairness. Unfortunately, Fair Queuing (FQ) appears to be somewhat expensive to implement. Specifically, FQ requires $O(log(n))$ work per packet to implement fair queuing, 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[1] at high speeds. Some attempts have been made to improve the efficiency of FQ; however such attempts either do not avoid the $O(log(n))$ bottleneck or [McK91] are not fair.

In this paper we shall define an isolation mechanism that achieves nearly perfect fairness (in terms of throughput) and which takes $O(1)$ packet processing work. Our scheme is simple and therefore inexpensive to implement at high–speeds at a router (or gateway).

1.1 Flows - a generalization of datagram and VC networks

Our intent is to provide firewalls between different packet streams. In this subsection, we define the notion of a packet stream more carefully using the notion of a *flow* [Zha91]. As in [Zha91], a flow has two properties:

- A flow is a stream of packets which traverse the same route from the source to the destination and that require 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.

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

The notion of a flow is quite general and applies to datagram networks (e.g., IP, OSI) and Virtual Circuit networks like X.25 and ATM. For example, a flow could be identified by a Virtual Circuit Identifier (VCI) in a virtual circuit network like X.25 or ATM. On the other hand, in an IP network flow could be identified by packets with the same source–destination IP addresses.[2] While the 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 an FTP connection between source A and destination B to receive a larger share of the bandwidth than a telnet connection between A and B.

As in all FQ 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.

1.2 Paper Organization

The rest of the paper is organized as follows. In the next section, we review the relevant previous work in more detail. In particular, we will use some of the techniques introduced by McKenney [McK91]; however, McKenney's scheme is potentially unfair, while ours is not. Readers who are familiar with the previous work should consider skipping Section 2. Any implementation of fair queuing has to solve two problems: the lookup problem and the queue servicing problem. Thus our final solution consists of two distinct and orthogonal ideas for these two problems:

- A new technique for reducing the cost of lookups in datagram networks called *source hashing* that is described in Section 3. We apply source hashing to the problem of looking up the state information associated with a flow. (Note that this lookup is required in all versions of fair queuing.)
- A new technique for avoiding the unfairness of round-robin scheduling called *deficit round-robin* that is described in Section 4. Round-robin scheduling [Nag87] 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.

[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.

2 Previous Work

As described in [DKS89], the key characteristics of a queuing algorithm are the bandwidth allocation (which packets get transmitted), promptness (when do these packets get transmitted), and buffering (which packets are discarded at the gateway). In first-come-first-serve (FCFS), the most common queuing algorithm used these days, the presumption is that congestion control is implemented by the sources. In FCFS, the order of arrival completely determines promptness, buffering and bandwidth.

In feedback schemes for congestion control, where connections are supposed to reduce the rate at which they send when they sense a congestion, a rogue connection can keep increasing its share of the bandwidth and cause the other (well–behaved) connections to reduce their share. It is easy in FCFS queuing, for a rogue connection to send packets at a high rate and capture an arbitrary part of the outgoing bandwidth. It is important to build firewalls between flows, so that the badly-behaved flows cannot penalize other 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 fairness can produce exponentially bad fairness properties as shown below.

In Figure 1 for examples 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 discriminates 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 node 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.

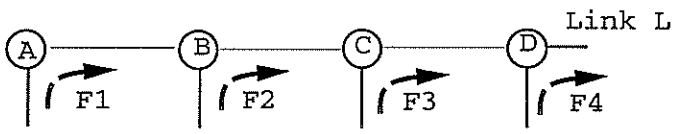

Figure 1: Solving the parking lot problem using source hashing to produce consistent random labels.

2.1 Nagle's solution

In Figure 1, the problem arose because the router gave fair access based on input links. Thus at router D, $F4$ is offered the same bandwidth as flows $F1$, $F2$ and $F3$ put together. It is unfair to allocate bandwidth based on topology. A better idea is to distinguish flows at a router and treat them separately.

Nagle [Nag87] 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 bottlenecked resource. Nagle proposed identifying flows using source-destination addresses and then using separate output queues for each flow; the queues are then serviced in round-robin fashion. This prevents a source from arbitrarily increasing its share of the bandwidth or the delay of other sources. In fact, when a source sends packets too quickly, it merely increases the length of its own queue. Thus a source that behaves itself does not lose anything; a source that does not behave itself, eats into its own share of buffering[3] at a router. An ill–behaved source's packets will get dropped repeatedly.

Despite its merits, there are flaws in this scheme. It ignores packet lengths: the hope is that the average packet size over the duration of a session, is the same for all flows, in which case each flow gets an equal share of the outbound bandwidth. However, in the worst case, a flow can get $\frac{Max}{Min}$ times the bandwidth of another flow, where Max is the maximum packet size and Min is the minimum packet size. Also, a packet which arrives into an empty queue that has just been examined has to wait until all the other active connections have been serviced, so there is no guarantee of promptness. The service time of a packet is dependent on the size of the packets which are scheduled to be serviced before it in the round robin order. This results in higher worst–case latency.

2.2 Fair Queuing

Demers, Keshav and Shenker devised an ideal algorithm called *bit–by–bit round robin – (BR)* which solves the flaws in Nagle's solution. In the BR scheme, each flow gets to send one bit at a time in round robin fashion. Since it is impractical to implement such a system, they suggest calculating the time (i.e. the departure time) when the 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 and deleting from queues require a time complexity of $O(log(n))$. Further, we need to lookup state for each flow (in order to calculate departure times).

[3] the problem of allocating buffers between flows was not specifically addressed by Nagle; McKenney proposed an elegant algorithm for sharing the total buffers among all flows that we describe in Section 2.3

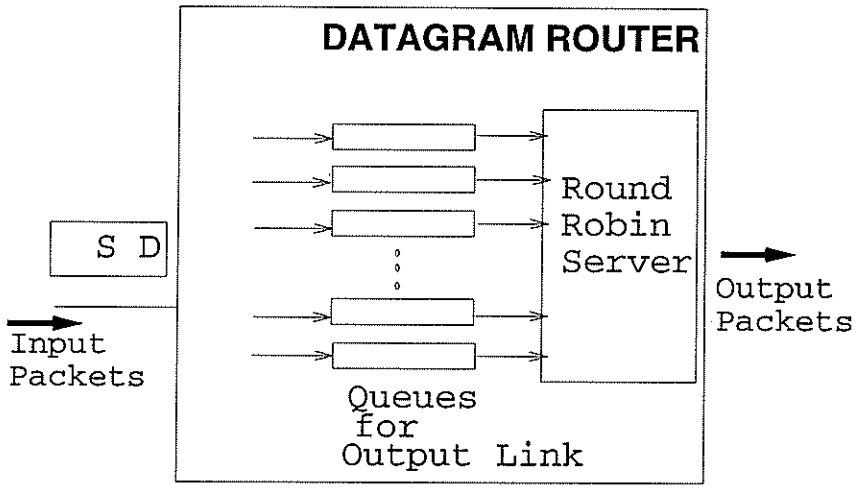

Figure 2: Nagle's scheme uniquely identifies a connection with a queue. He assumes finite buffering at the router and does not consider packet sizes while servicing packets in a round robin manner.

In this scheme, buffers are allocated to flows as long as there are free buffers. On the arrival of a packet if there are no free buffers, the buffer storing the last packet in the queue is taken and is used to store the incoming packet. While this scheme guarantees absolute fairness, the packet processing cost makes it hard to implement cheaply at high speeds.

2.3 Stochastic Fair Queuing

Stochastic Fair Queuing – SFQ was proposed by McKenney to address the inefficiencies of the DKS algorithm. As in Nagle's scheme, McKenney uses a simple and efficient hashing scheme (that ignores hash collisions) to map incoming 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 all flows that happen to hash into the same bucket be treated equivalently. This simplifies the hash computation (the hash computation is now guaranteed to be $O(1)$ in the worst case since we ignore collisions) and also allows the use of a finite number of queues. The down side is that flows that happen to collide with a number of 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 flows through the router, the probability of unfairness will be small.

The queues are then serviced in a round robin manner, without considering packet lengths. When there are no free buffers to store a packet, the packet at the end of the longest queue is

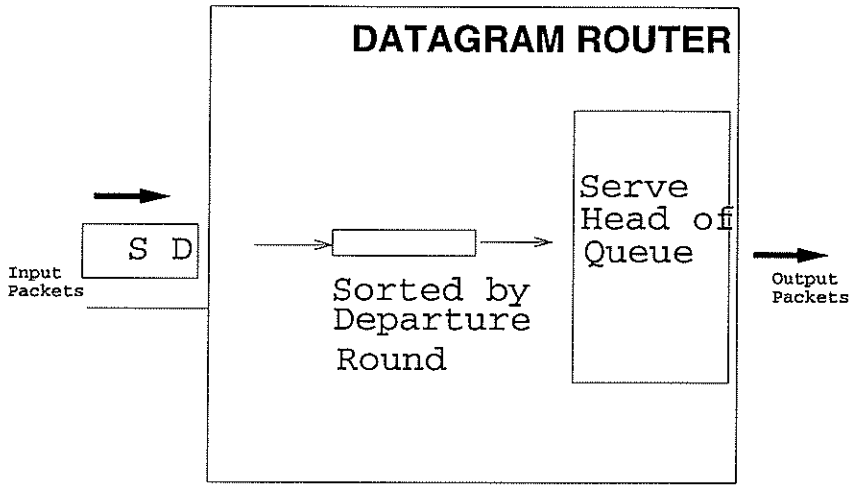

Figure 3: The Demers, Keshav and Shenker bit–by–bit round robin algorithm. The departure time is calculated when the packet enters the router.

dropped, since it is the one to be transmitted last. McKenney implements this scheme with $O(1)$ time complexity for all operations on the queues. To allow the buffer stealing algorithm to take $O(1)$ time, McKenney uses a bucket sorting technique: a table with M entries (M is the maximum number of buffers) is used to keep track of the number of buffers used by each flow, as well as the flow which currently uses the largest number of buffers.

The major contributions of McKenney's scheme are the buffer stealing algorithm, and the idea of using hashing and ignoring collisions. However, his scheme does nothing about the inherent unfairness of Nagle's round-robin scheme, since the scheme makes no attempt to redress the unfairness arising from disparities in the packet sizes of flows. It is probably more accurate to describe McKenney's scheme as a technique for improving the efficiency of Nagle's scheme without addressing the unfairness problem.

Besides the packet size problem (which is the major problem), there are also three smaller problems with the use of a hash function:

- The hash functions suggested by McKenney are uniform and require fewer than 10 instructions on an MC68020 processor. However, these are specific to IP addresses and are unlikely to be as efficient when used with OSI addresses (which have variable length fields). Also, with multiprotocol routers gaining in popularity, there will have to be multiple hash functions used in a router.
- McKenney identifies flows based on IP source–destination pairs only. This may not provide a fine enough level of discrimination. McKenney does consider using the TCP

ports along with the source destination pair in the hash function, but this would require more processing, and would also be a layer violation.

- Finally if two connections collide, (or hash on to the same queue) because the hash of the two sets of addresses is the same, they will continue to hash on to the same queue in all subsequent connections. McKenney perturbs the hash function periodically to remedy this but perturbing the hash function with active sessions is likely to cause reordering of packets unless more complex measures are taken. Note that reordering can reduce the performance of many popular transport protocols (e.g., TCP).

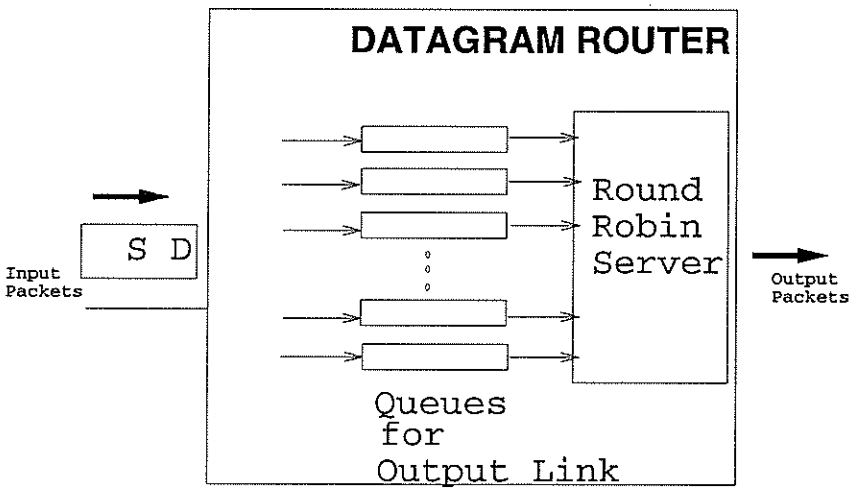

Figure 4: McKenney uses fixed number of queues and performs hashing with chaining to map flows to queues. This mapping might not be unique at all times and when there are collisions, they are ignored.

In Section 5, we will see how a new technique called *source hashing* can be used to avoid some of these problems with ordinary hashing.

Recall that the implementation of fair queuing can be divided into two problems: the lookup problem and the queue servicing problem. In the next two sections, we will describe our approach for solving these two problems.

3 Source Hashing and the Lookup Problem

The techniques used for one–to–one mapping from a flow ID (e.g., source–destination address pair) into a corresponding queue in previous work [McK91, DKS89] do not lend themselves well to high-speed implementations.

Hardware assists with content addressable memories (CAMs) can be used. They have worked well with gateways that implement a single protocol that is not subject to growth and revision. Since multiprotocol gateways are the norm, this can be a difficult option.

Various types of trees and tries are another alternative. Trees will require many memory references and so they are not good for high-speed implementations. In a trie, there would be at least an eight–level tree (apart from each level having a 256–way branching (TCP/IP)); the number of levels is much larger in OSI. Also, there will have to be periodic cleanup iterations which will have to delete pointers to sessions which have been inactive for a substantial amount of time.

A hash function with chaining can be used on the source–destination addresses to map the packet to a flow. The source and destination addresses of the packet will have to be read once to compute the hash and then they will have to be read again to compare with the head of the corresponding queue to ensure that there is no collision. In addition, periodic checks have to be performed to delete state being maintained for inactive flows. These requirements increase the overhead of the scheme.

3.1 Source Hashing

In normal hashing, the consumer of an identifier (e.g., a compiler looking up variables in a symbol table) uses a deterministic function to convert the identifier into an index into a hash table. The function must be deterministic so that all accesses to the identifier will index into the same position. However, it is hoped that the indexes produced by hashing various identifiers will be "random" so that hash collisions will be rare. In *source hashing*, by contrast, the producer of an identifier (e.g., the programmer!) picks a random index for each identifier and consistently tags every instance of the identifier with the same random index. The random index is used (in place of the output of a hash function) to index into the consumer's hash table.

Source hashing is certainly absurd in the simple compiler example described above. However, it is more interesting if the producer is the source of a packet and the consumer is the packet destination which must lookup a state table entry (corresponding to the source) in order to process the packet. Consider 6 byte Ethernet (more accurately, these are administered by the IEEE 802.1 committee) addresses. A number of network devices (most notably bridges), have hardware support to lookup state tables that are keyed by such 6 byte addresses. Some bridges use hashing for lookups. A bridge might use a hash function to convert the 6 byte address into a smaller index (say 12 bits); this hash index is used as an index into a hash table.

If hashing with chaining is used, then each index entry points to a list of entries for the various addresses that have hashed into this bucket.

In source hashing, however, we suggest that the 6 byte address is increased to an 8 byte address: 6 bytes are still used for uniqueness, but the remaining 2 bytes is used as a random index that is *supplied by the source.* If it were possible to change the manufacturing process (which assigns unique addresses in ROMs), it is conceivable that the manufacturing process could even produce these extra 2 byte random numbers. However, if that is not possible it is easy for the source to provide the random 2 bytes using a pseudorandom number generator whose seed depends on the source address. All we need is that the same 2 byte random index is associated with the same 6 byte UID; this association can be maintained by the source or through the manufacturing process. The source can change the association; the only cost is performance, and not correctness.

What we gain by this is that the *receiver does not need to compute a hash function*; the hash index is, in a sense, precomputed by the source. Computing a hash function was a trivial part of overall packet processing at low speeds; however, at higher (especially Gigabit) speeds the computation of the hash function is expensive in terms of either extra hardware or time.[4]. A second advantage is *the hash indices are guaranteed to be more truly random than those computed by hash functions.* In ordinary hashing, there is a tradeoff between the complexity of the hash function [McK91] and the randomness of the resulting indices: good hash functions based on CRC-like functions are expensive; cheaper hash functions typically are not as good or need to be tuned to the details of the particular addressing format. These problems are avoided with source hashing.

Source hashing is a very general idea and is applicable to lookups at all layers in the network. This idea, together with other ideas for reducing the cost of lookups, is explored in more detail in another paper [Var94].

For now, let us return to the problem of looking up the state associated with a flow ID (which is say identified by source-destination addresses). Thus in Figure 5 assume that all packets sent by the source to the destination contain a random number r (that we will call a *random index*) as well as the source identifier S.

The typical analyses for hashing *assume* hashed values are uniformly distributed. In source hashing (given reasonable pseudorandom numbers), this assumption should be quite valid. Thus assume that each flow uses a different random index with high probability. Then we can use the random index to discriminate flows. However, as in [McK91], we ignore collisions

[4]For example, DEC's GIGAswitch product, which is a multiport FDDI bridge, uses a semi-custom chip to do hashing at 100 Mbps

(Figure 5); if two flows happen to choose the same random index, their packets will be placed in a common queue and they will have to share in a common round-robin opportunity. However, if the random index is large enough, this should happen rarely.

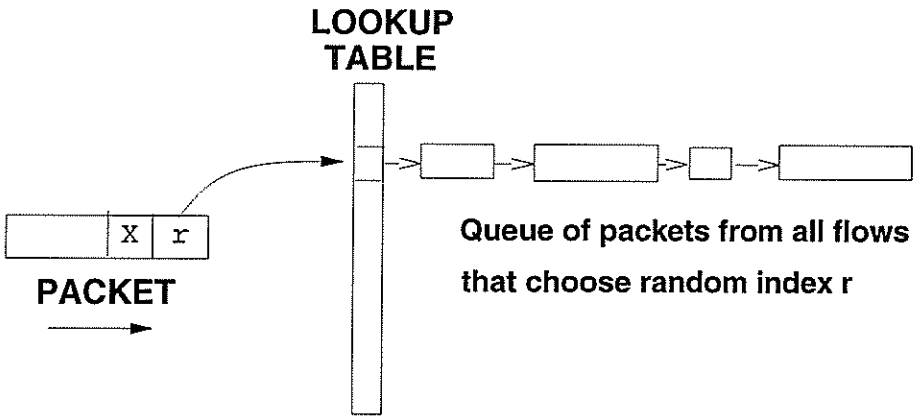

Figure 5: In source hashing, a random index r in the packet is used to index into a lookup table to yield a pointer into the state. Combining this with McKenney's idea of ignoring collisions, we allow all flows that choose the same random index to hash onto the same queue. Given a sufficient large r, two flows should choose the same random index with low probability.

Sources can use pseudorandom numbers, preferably with seeds that depend on a unique source address. Source hashing does not require truly random random numbers that meet all statistical tests. To save computing time, a small queue of random numbers can be computed when a source first comes up. When a flow starts, the source picks the random number at the head of the queue; random numbers can be reused by placing them at the end of the queue when a flow terminates or becomes idle. What size random numbers should be used? The simplest approach is to require sources to provide fairly large random numbers (say 16 or 24 bits). Destinations that use smaller lookup tables can (assuming that the lookup table size is a power of 2) efficiently extract the right number of random bits — e.g., a destination with a lookup table of size 256 could use the low order 8 bits of the source random index.

We list the advantages of source hashing over ordinary hashing:

- There is no need for the receiver to compute a hash function. Even the simplest hash function computation described in [McK91] takes about 10 instructions, and this savings becomes more pronounced at high speeds.
- The hash indices are more random than those produced by the best (and expensive) hash functions, assuming the use of good pseudorandom number generators at the source.

- In ordinary hashing, if multiple protocols are used (e.g., variable length OSI addresses of up to 20 bytes and 4 byte IP addresses), there will be a need to find optimal hash functions for each type of address. This is is unnecessary in source hashing as the random numbers are generated independently of the address format.

- In ordinary fair queuing, if a router wishes to discriminate more finely than at the level of source-destination addresses, the router needs to use information in the transport header (e.g., TCP port numbers), a layer violation. While there are aesthetic reasons to avoid layer violations, the more pragmatic reason is that it requires changing all gateways/routers every time any transport header changes! With our version of McKenney's scheme based on source hashing, the transport layer can pass down a random index to the routing layer.

- In McKenney's scheme using ordinary hashing, the problem of a flow suffering frequent collision is avoided by periodically changing the hash function at the receiver. But this allows the possibility of out-of-order packets; avoiding this adds complexity. In source hashing, the source can change its random index at any time; however, it can do so between transport sessions or when all outstanding packets have been acked for a connection, thereby avoiding the misordering problem.

The reader may wonder about the following "problems" with source hashing:

- *If two sources pick the same random indices using source hashing, won't they keep colliding?* Since a random index is generated when a connection is setup and is used for the duration of a connection, two sources which select the same random indices are likely to collide for the duration of a connection. Also note that repeated collisions can also occur with ordinary hashing: if the hash function chosen by a router results in two addresses A and B colliding, then the two addresses will keep colliding until the hash function changes. It is true, however, that in ordinary hashing, the router has sufficient information to change its hash function; however, this is rarely done in real implementations. The advantage in source hashing is that the probability of two sources generating the same random indices for successive sessions is very low; in ordinary hashing, unless the hash function is changed they will continue to collide.

- *In source hashing, does the probability of collision depend on the distribution of addresses on the network?* Actually, assuming good pseudorandom numbers, the probability that a source picks a random index used by another source is just $1/M$, where M is the number of distinct random indices. It does not depend on the details of a particular network.

4 Deficit Round Robin and the Queue servicing Problem

While ordinary round-robin servicing of queues can be done using $O(1)$ work (as in the proposals of Nagle and McKenney), the major problem is the unfairness caused by possibly different packet sizes used by different flows. In this section, we show how to remove this flaw, while still requiring only $O(1)$ work. Since our scheme is a simple modification of round-robin servicing, we call our scheme *deficit round-robin.*

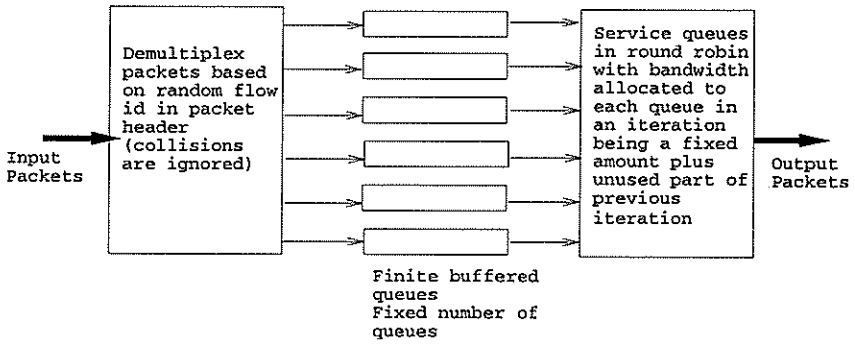

Figure 6: Deficit Round Robin: The packets generated by a host have the random number (flow id or connection id) generated by the host for that session. They are stored in queues based on this flow id. The servicing of the queues is done using a modified round robin method.

A sketch of the main idea is shown in Figure 6. We use source hashing 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 iteration because its packet size was too large, the remainder from the previous quantum is added to the quantum for the next iteration.* Thus deficits are kept track off; queues that were shortchanged in an iteration are compensated in the next iteration.

In the next few subsections, we will state and precisely prove the properties of deficit round-robin schemes. We start by defining precisely the figures of merit (i.e., fairness, work) used to evaluate different schemes; we then give a precise statement of the algorithm; finally, we prove that the fairness of deficit round-robin is nearly perfect, and that the work is $O(1)$.

In defining the figures of merit we will make two assumptions:

- When combining deficit round-robin with source hashing, there are two orthogonal issues which affect the performance of the combination. The first issue is how well the random ids map different flows into different queues (source hashing); this can be calculated

by the standard probabilistic analysis for hashing. The second is how fair is the queue servicing algorithm (deficit round robin). To cleanly separate these issues, we will assume during the analysis of deficit round-robin that flows are mapped uniquely into different queues. Later we will incorporate the effect of source hashing.

- In calculating the measures, we will assume that each flow always has a packet to send. Our fairness metric will then measure the fair share of bandwidth that each flow receives. Another important metric (that we will ignore for the next two sections) is latency. In fact, some fair queuing schemes also provide powerful latency guarantees. We will return to the problem of guaranteeing latency in Section 6.

4.1 Figure of Merit for Fair Queuing Algorithms

Currently, there is no uniform figure of merit defined for Fair Queuing algorithms. We shall define two measures: *FairnessIndex* (that measures how fair the queuing discipline is) and *WorkQuotient* (for the time complexity of the queuing algorithm). Similar fairness measures have been defined before, but no precise 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 as well.

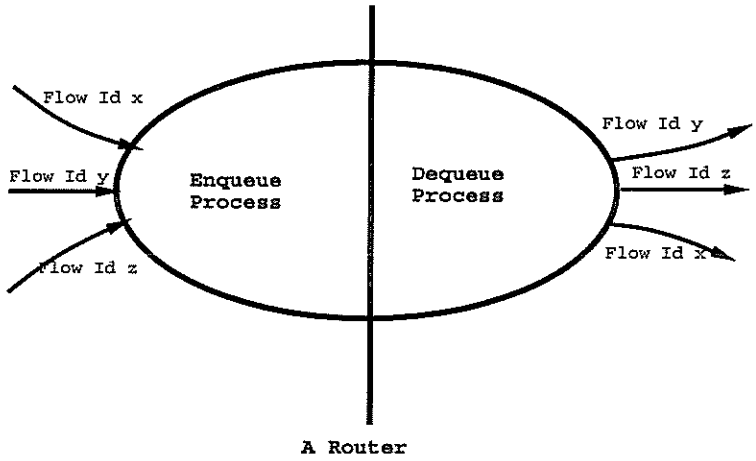

Figure 7: Router: The two processes going on in a router are enqueuing of a packet and dequeuing of a packet. There is another process which actually does the routing, but we shall not concern ourselves with that aspect of a router.

To define the work measure, we assume the following model of a router and the traffic pattern that is illustrated in Figure 7. 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 per output link (although the figure shows a single Dequeue Process) 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 4.1 *WorkQuotient is defined as the maximum of the number of instructions in enqueuing or dequeuing a packet from the router.*

$$WorkQuotient = Max(\#Instructions_{Enqueue}, \#Instructions_{Dequeue})$$

Definition 4.2 *WorkComplexity is defined as the maximum of the time complexities in enqueuing or dequeuing a packet from the router.*

$$WorkComplexity = Max(TimeComplexity_{Enqueue}, TimeComplexity_{Dequeue})$$

For example, if a fair queuing algorithm takes $O(log(n))$ to enqueue a packet and $O(1)$ to dequeue a packet, we say that the *WorkComplexity* of the algorithm is $O(log(n))$. Similarly, if it takes 10 instructions (on say, a SPARC chip) to enqueue a packet and 14 instructions to dequeue a packet, we would say that the implementation has a *WorkQuotient* of 14 instructions.

To define the fairness measure, we assume a heavy traffic model as shown in Figure 8. Thus all n flows have a continuous stream of arbitrary sized packets arriving to the router, and all these flows wish to leave the router on the same outgoing link. The essential assumption is that there is always a backlog for each flow, and the backlog consists of arbitrary sized packets.

Assume that we start sending packets out on the outgoing link at time 0. Let $sent_{i,t}$ be the total number of bytes sent by flow i by time t; let $sent_t$ be the total number of bytes sent by all n flows by time t. Intuitively, we will define a fairness quotient for flow i that is the worst-case ratio (across all possible input packet size distributions in Figure 8) of the bytes sent by flow i to bytes sent by all flows. This merely expresses the worst-case "share" obtained by flow i. While we can define such a quotient after any amount of time, it is more natural (since we are measuring throughput) to take the limit as t tends to infinity. Note that the reason we use time as an index (instead of using rounds, which is more natural for a round robin system) is that we want our definition to hold for implementations that do not involve the notion of a round.

Definition 4.3 $FQ_i = Max(\lim_{t\to\infty} \frac{sent_{i,t}}{sent_t})$, *where the maximum is taken across all possible input packet size distributions for all flows (Figure 8).*

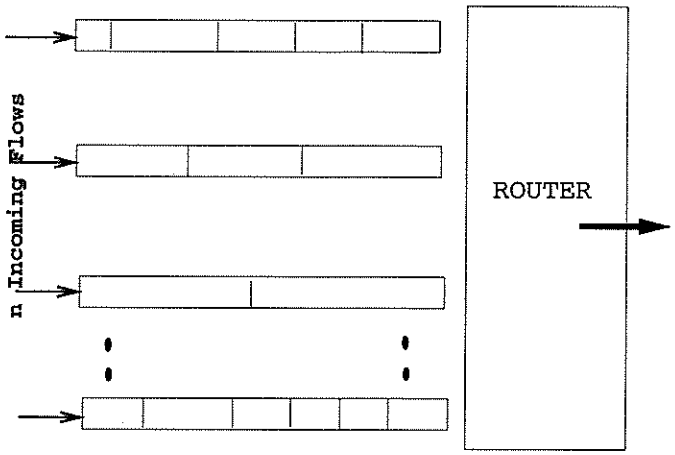

Figure 8: The router has n active incoming flows, with packets coming in at least at the service rate of the outgoing link. This means that the buffers will always be full at steady state. The packet sizes are arbitrary but less than Max for the network and greater than Min.

Next, we assume there is some quantity f_i, settable by a manager, per flow which expresses the ideal share to be obtained by flow i in an ideal fair queuing system. Thus the "ideal" fairness quotient for flow i is $\frac{f_i}{\sum_{j=1}^{n} f_j}$. In the simplest case, all the f_i are equal and the ideal fairness quotient is $1/n$. Finally, we can measure how far a fair queuing implementation departs from the ideal by measuring the ratio of actual fairness quotient achieved to the ideal fairness quotient. We call this the fairness index, *FairnessIndex*.

Definition 4.4 *The fairness index for a flow i in a fair queuing implementation is:*

$$FairnessIndex_i = \frac{FQ_i \cdot \sum_{j=1}^{n} f_j}{f_i},$$

These measures can be thought of in terms of bandwidths. For example, if a flow is supposed to get one–fifth of a link bandwidth of 5Mbps and it gets 800Kbps, then its *FairnessIndex* is 0.8.

4.2 Algorithm

We propose an algorithm called *Deficit Round Robin* (Figure 9, Figure 10) for servicing queues in a router (or a switch). We shall use router and switch interchangeably in the rest of this paper. We will assume that the quantities f_i, that indicate the share given to flow i, are specified by a quantity called $Quantum_i$ for each flow (for reasons that will be apparent below). Also, since the algorithm works in rounds, we will measure time in terms of rounds.

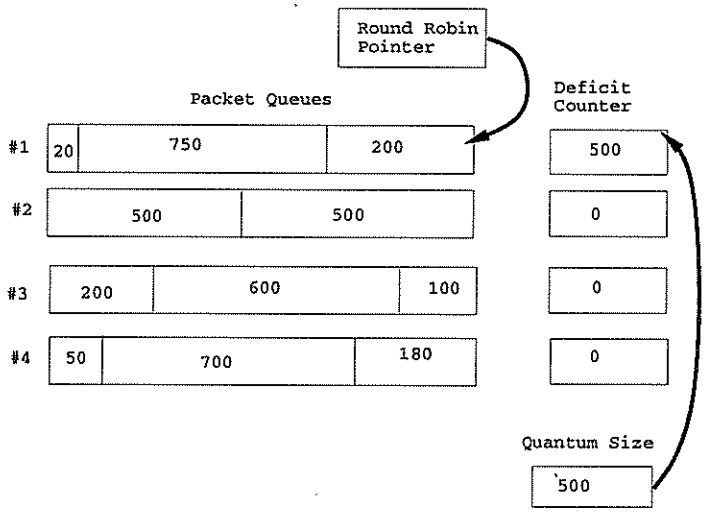

Figure 9: Deficit Round Robin: At the start, all the *DeficitCounter* are initialized to zero. The round robin pointer points to the top of the active list. Then, when the first queue is serviced the *Quantum* is added on to the *DeficitCounter*. The remainder after servicing the queue is left in the *DeficitCounter*

The packets coming in on different flows are stored in different queues. Let the number of bytes sent out by a flow i in round k be $bytes_{i,k}$. Each queue can send out packets in the first round such that $bytes_{i,1} \leq Quantum_i$. The remaining amount is stored in a state variable called $DeficitCounter_i$, which is reset to 0 if there are no more packets in the queue for flow i at the end of the round. In subsequent rounds, the amount of bandwidth usable by this flow is the sum of $DeficitCounter_i$ of the previous round added to $Quantum_i$. The pseudo code for this algorithm is shown in Figure 11

In the simplest case $Quantum_i = Quantum_j$ for all flows. Exactly as in weighted fair queuing, however, each flow i can ask for some relativw bandwidth allocation and the system manager can convert it into an equivalent value of $Quantum_i$. Clearly if $Quantum_i = 2Quantum_j$, the manager intends that flow i get twice the bandwidth of flow j when both i and j are active.

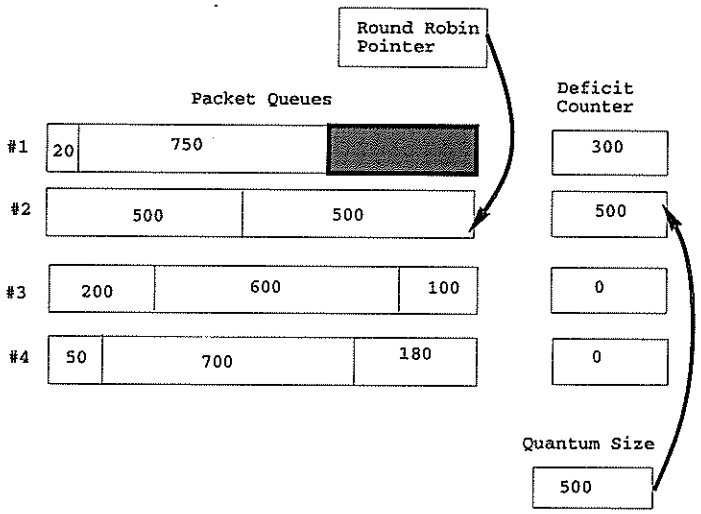

Figure 10: Deficit Round Robin (2): After sending out a packet of size 200, the queue had 300 bytes of its quantum left. It could not use it this round, since the next packet in the queue was 750 bytes. Therefore, the amount 300 will carry over to the next iteration when it can send packets of size totaling 300 (deficit from previous round) + 500 (quantum).

4.3 Performance and Bounds

We begin with a lemma that is true for all executions of the DRR algorithm (not just for the heavy traffic scenario which is used to evaluate fairness):

Lemma 4.1 *For all i and all executions of the DRR algorithm:*

$0 \leq DeficitCounter_i < Max.$

Proof: Initially, $DeficitCounter_i = 0 \Longrightarrow DeficitCounter_i < Quantum_i$.

At the end of a round, there are two possibilities:

- if a packet is left in the queue for flow i, then it must be of size strictly greater than $DeficitCounter_i$. Also, by definition, the size of any packet is no more than Max, so $DeficitCounter_i$ is strictly less than Max. It is also easy to see that the servicing of packets guarantees that $DeficitCounter_i \geq 0$.
- if no packets are left in the queue, then the algorithm resets $DeficitCounter_i$ to zero.

$Queue_i$ – refers to the ith queue in the *router* which
stores packets having a flow id i.
Enqueue(), Dequeue() are standard *Queue* operators.
n is the maximum number of flows for this output link.
In addition, we define a list of active flows *ActiveList*; there are
standard operations defined on it. e.g. *InsertActiveList*.
The FreeBuffer() function frees up a buffer from the flow having
the longest queue.

Initialization:
for($i = 0; i < n; i = i + 1$)
$DeficitCounter_i = 0$;

Enqueuing module: on arrival of packet p
$i = ExtractFlow(p)$
$if(ExistsInActiveList(i) == FALSE)$
$InsertActiveList(i)$; (*add i to end of active list*)
$DeficitCounter_i = 0$;
if no free buffers left then
FreeBuffer(); (* can be done using McKenney's buffer stealing scheme*)
$Enqueue(i, p)$;

Dequeuing module:
While(TRUE) do
If ActiveList is not empty
Remove head of active list, say flow ID i
$DeficitCounter_i = Quantum_i + DeficitCounter_i$;
while(($DeficitCounter_i > 0$) and ($Queue_i$ not empty))
$PacketSize = Size(Head(Queue_i))$;
if($PacketSize \leq DeficitCounter_i$)
$Send(Dequeue(Queue_i))$;
$DeficitCounter_i = DeficitCounter_i - PacketSize$;
Else break; (*skip while loop if packet is too large*)
if ($Empty(Queue_i)$)
$DeficitCounter_i = 0$;
Else $InsertActiveList(i)$; (*add i to end of active list*)

Figure 11: Code for Deficit Round Robin at a router

□

Next we consider the case where only flow i always has a backlog (i.e., the other flows may or may not be backlogged or even active), and show that the difference between the ideal and actual allocation to flow i is always bounded by the size of a maximum-size packet. While this result will imply that the fairness index of a flow is 1, it has even stronger implications. This is because it implies that even during arbitrarily short intervals, the discrepancy between what a flow gets and what it should get is bounded by *Max*.

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 n queues. We assume that rounds are numbered starting from 1, and the start of Round 1 can be considered the end of a hypothetical Round 0.

Theorem 4.2 *Consider any execution of the DRR scheme in which flow i always has a packet to send in every round. After any k rounds the difference in the bytes that flow i should have sent and the bytes that it actually sends is bounded by Max.*

Proof: We define the following terms:

$DeficitCounter_{i,k}$ = the value of $DeficitCounter_i$ for flow i at the end of round k,
$bytes_{i,k}$ = bytes sent by flow i in round k,

$sent_{i,k}$ = bytes sent by flow i in rounds 1 through k.

Therefore:

$$sent_{i,K} = \sum_{k=1}^{K} bytes_{i,k}. \tag{1}$$

Initially, we have:

$\forall i$, $DeficitCounter_{i,0} = bytes_{i,0} = 0$;

The main observation (which follows immediately from the protocol) is:

$bytes_{i,k} + DeficitCounter_{i,k} = Quantum_i + DeficitCounter_{i,k-1}$.

This reduces to:

$bytes_{i,k} = Quantum_i + DeficitCounter_{i,k-1} - DeficitCounter_{i,k}$.

Over K rounds of servicing of flow i we get the following values of $bytes_{i,k}$:

$$bytes_{i,1} = Quantum_i + DeficitCounter_{i,0} - DeficitCounter_{i,1}. \quad (2)$$
$$bytes_{i,2} = Quantum_i + DeficitCounter_{i,1} - DeficitCounter_{i,2}. \quad (3)$$
$$\vdots$$
$$bytes_{i,m} = Quantum_i + DeficitCounter_{i,m-1} - DeficitCounter_{i,m}. \quad (4)$$
$$\vdots$$
$$bytes_{i,K} = Quantum_i + DeficitCounter_{i,K-1} - DeficitCounter_{i,K}. \quad (5)$$
$$(6)$$

Summing up the K equations above results in:

$$sent_{i,K} = bytes_{i,1} + bytes_{i,2} + ... + bytes_{i,K}.$$
$$sent_{i,K} = K \cdot Quantum_i + DeficitCounter_{i,0} - DeficitCounter_{i,K}.$$

since $DeficitCounter_{i,0} = 0$, we get:

$$sent_{i,K} = K \cdot Quantum_i - DeficitCounter_{i,K}. \quad (7)$$

The ideal bytes allocated to a flow i after K rounds is $K \cdot Quantum_i$. Subtracting, it is easy to see that the difference is $\leq Max$ (using Lemma 4.1)

□

Corollary 4.3 *When all flows require equal bandwidth, (i.e. all flows have the same quantum) the maximum difference in the total bytes allocated to any two flows after any number of rounds is less than Max.*

Proof: From Equation 7 we get the bytes used by a flow after K rounds,

$$sent_{i,K} = K \cdot Quantum_i - DeficitCounter_{i,K}$$

any other flow j, has a similar equation:

$$sent_{j,K} = K \cdot Quantum_j - DeficitCounter_{j,K} \quad (8)$$

Since the flows i and j have the same *Quantum* values over all the K rounds, it follows that the maximum difference (using equations Equation 7, Equation 8) is:

$|DeficitCounter_{i,K} - DeficitCounter_{j,K}| \leq Max$ (using Lemma 4.1). $\square$

Finally, let us consider the heavy traffic scenario (Figure 8) in which there are n continuously active incoming flows. Recall that we assume that there are packets available to be sent out from all flow queues at all times. The packets are of arbitrary sizes in the range between *Min* and *Max*. Using this scenario, we show that the fairness index of all flows is 1.

Theorem 4.4 *For any flow i,* $FairnessIndex_i = 1$.

Proof: We know from Theorem 4.2 that the maximum difference between a flow's ideal and actual allocations is *Max*. Thus for any flow i and any round K:

$K \cdot Quantum_i - Max \leq sent_{i,K} \leq K \cdot Quantum_i$

Thus, summing over all n flows we get:

$K \sum_{i=1}^{n} Quantum_i - n \cdot Max \leq sent_K \leq K \sum_{i=1}^{n} Quantum_i$.

Thus,

$\frac{K \cdot Quantum_i - Max}{K \sum_{i=1}^{n} Quantum_i} \leq \frac{sent_{i,K}}{sent_K} \leq \frac{K \cdot Quantum_i}{K \sum_{i=1}^{n} Quantum_i - n \cdot Max}$.

Now in the limit as time t tends to infinity, so does the number of rounds K. If we take the limit of the previous equation as K goes to infinity we get (see definition of FQ given in Definition 4.3):

$\frac{Quantum_i}{\sum_{i=1}^{n} Quantum_i} \leq FQ_i \leq \frac{Quantum_i}{\sum_{i=1}^{n} Quantum_i}$,

which implies that:

$FQ_i = \frac{Quantum_i}{\sum_{i=1}^{n} Quantum_i}$.

Finally, since $f_i = Quantum_i$ (recall that in DRR the share allocated to each flow is expressed using the quantum allocated to each flow), we get:

$FairnessIndex_i = \frac{FQ_i \cdot \sum_{j=1}^{n} f_j}{f_i} = 1$. $\square$

Having dealt with the fairness properties of the DRR scheme, we turn to analyzing the worst-case packet processing work. It is easy to see that the size of the various *Quantum* 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 *Quantum* variables.

Theorem 4.5 *The WorkComplexity for Deficit Round Robin is O(1), if for all i, $Quantum_i \geq Max$.*

Proof:

While enqueuing a packet we lookup the queue the packet goes into, which takes $O(1)$ time complexity using source hashing. Once we figure out the queue to enqueue the packet, we append it to the end of the queue; this also has a complexity of $O(1)$. In case a buffer has to be freed, we have to perform a "buffer stealing" operation, which can be done in $O(1)$ time using data structures defined in [McK91]. Therefore, the worst case time complexity for an Enqueue operation is $O(1)$.

While dequeuing a packet, we have to figure out the next queue to service, which can be done in $O(1)$ time in a round robin queuing discipline since we keep a list of active flows. Next we have to perform an add operation, a compare operation, which can be done in $O(1)$ time. Finally, there are (a variable number of) packet send operations. This number of send operations is dependent upon the size of the $Quantum_i$ and the size of the packets that we have in that queue. If for any i, $Quantum_i$ is less than Max then, the worst case time complexity is $O(\frac{n \cdot Max}{Quantum})$, where $Quantum$ is $Min_i(Quantum_i)$. (This is because if the quantum is too small we have to keep skipping a flow queue until its deficit builds up to be large enough to send the packet at the head of the flow queue.) However, it is possible to define an amortized work complexity, and prove that the amortized work complexity is $O(\frac{Max}{Quantum})$ if $Quantum$ is less than Max. To define the amortized work complexity, we charge the work to skip between lists to the packet at the head of the queue we are skipping to, even if we don't send a packet for that queue.

However, if $Quantum \geq Max$, the worst-case time complexity is $O(1)$. □

It should noted that in the average case, when the average packet size is much smaller than the Max, there will be multiple packets sent out in an iteration. Therefore, the cost of the operations in dequeuing a packet (which is $O(1)$ already) will be further amortized over the number sent out in that iteration.

5 Evaluation of Deficit Round Robin

In the previous analysis, we showed that if we ignored the collisions introduced by source hashing, deficit round robin (DRR) can achieve a *FairnessIndex* that is close to 1 and a $WorkComplexity = O(1)$. If we combine DRR with source hashing, the only difficulty is that

the *FairnessIndex* must be adjusted by the probability of collision. However, if the size of the random index (and the lookup table), is much larger than the number of concurrent flows (say a factor of 10), then the probability of collision is very small (0.1) and the *FairnessIndex* remains close to 1. More accurate analyses can be obtained using the standard analysis for hashing (see for example [CLR90]). Thus assuming a suitable choice of quantum size and the size of the random index, our overall scheme (which we will continue to refer to as DRR) achieves a *FairnessIndex* that is close to 1 and a *WorkComplexity* = $O(1)$.

We compare the *FairnessIndex* and *WorkComplexity* of the fair queuing algorithms that have been proposed until now. Nagle and McKenney do not consider packet sizes in their algorithms; therefore, in the worst case their schemes will have a fairness index of $\frac{Max}{Min}$. Also, if Nagle's scheme uses hashing with chaining to implement separate queues for each connection, the work will still be O(1) in the expected case. In contrast, McKenney's scheme takes $O(1)$ work in all cases. Deficit FQ is the only algorithm that provides a *FairnessIndex* of 1 and a O(1) *WorkComplexity*.

Table 1: Performance of Fair Queuing Algorithms

Fair Queuing Algorithm	**Fairness Index**	**Work Complexity**
Fair Queuing ([Nag87])	$\frac{Max}{Min}$	O(1) Expected
Fair Queuing ([DKS89])	1	O(log(n))
Stochastic Fair Queuing	$\frac{Max}{Min}$	O(1)
Deficit Round Robin	1 (Expected)	O(1)

Deficit Round Robin and FCFS The processing speeds of high speed packet switches and routers should match the high speeds of fiber. Therefore, there is a need to minimize the number of instructions required for packet enqueuing and dequeuing. While we have proved that our algorithm has a *WorkComplexity* of O(1), it is important to also show that the instruction count is low. FCFS queuing is the simplest queuing discipline and so it (probably) has the minimal number of instructions. Thus it makes sense to compare the packet processing work that has to be done in FCFS and DRR.

In packet enqueuing, there is an additional lookup to determine which queue to use. This can be achieved in one or two instructions in many architectures. If the buffers are all used up, then we need to determine the flow using the maximum number of buffers, and reclaim a buffer from that flow. In this (hopefully unlikely) event, it requires a few more instructions.

In sending out a packet, we have to traverse an additional list of active connections, which might require a few instructions. Additionally, a compare and a subtraction has to be done for every queue serviced. This cost may however be amortized over several packets, depending on the size of *Quantum*. Overall, in the typical cases, DRR will only require a few more instructions than a FCFS implementation.

Finally, in terms of memory, recall that as in FCFS, the overall buffering is shared between all flows, so the packet buffer sizes can be identical. The only extra cost is for the hash table (for example, 1000 concurrent flows and a 10,000 size hash table would make collisions very unlikely) and the active list.

6 Latency Requirements and DRR+

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 iterations, 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 bits/sec. In other words, a small packet can only be delayed by an amount proportional to its own size by every other flow. The DKS (Demers–Keshav– Shenker) 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 section indicate that Deficit Round Robin is competitive with Fair Queuing.

However, the additional latency properties of BR and DKS have attracted considerable interest. In particular, Parekh (see [Par93] for a simplified presentation) has calculated bounds for end-to-end delay assuming the use of DKS fair queuing at routers and token bucket traffic shaping at sources. As communication and computing continue to converge, and real-time (especially video) traffic gets sent over datagram networks, there is an increasing need for latency bounds.

At first glance, Deficit Round Robin (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 $\frac{\sum_i^n Quantum_i}{B}$. 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*.

However, the real motivation behind providing latency bounds is to allow real-time traffic to have predictable and dependable performance. The following modification of Deficit Round Robin (which for want of a better name we shall call DRR+) makes better provision for such traffic.

In DRR+ there are two classes of flows: the *latency critical* flows and the *best-effort* flows. Essentially 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.*

As a simple example, suppose each latency critical flow guarantees to send at most a single packet (of size at most s) every T seconds. 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 (this needs a more careful proof which is left for further work) that each latency critical flow is at most delayed by $((n' * s) + Max)/B$, 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.

In more complicated cases, it may help to have two round-robin active lists, one for latency critical flows and one for best-effort flows. The two lists can be serviced alternately using similar notions of a quantum for each list (in addition to a quantum for each flow). We are exploring more careful proofs and further extensions of DRR+. We are also working on simulations of this scheme which we will report later.

7 Conclusions

We have described a new scheme, Deficit-Round-Robin (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 fairness with $O(1)$ packet processing. The maximum discrepancy in the bandwidth allocated to each flow never exceeds the quantum size at any instant. If the quantum size is equal to *Max*, the discrepancy is the minimum possible amount. Further, over large periods of time this discrepancy accounts for a vanishingly small percentage.

DRR does not, however, provide strong latency bounds compared with the more ideal fair queuing schemes. We have suggested the use of a small modification, called DRR+, that essentially works by having a subset of metered flows be given priority in the round-robin servicing order. While it is easy to see how DRR+ works in simple cases, more work needs to be done to generalize the scheme and prove precise latency bounds.

All the evaluation of fair queuing Schemes in this paper has been analytical. We are currently working on simulation results to confirm the analytical results in this report. We will update this report when we do so.

The lookup problem (for identifying the flow corresponding to an incoming packet) is completely orthogonal to the DRR scheme. We emphasize that DRR can be used with ordinary hashing techniques (e.g., [McK91]). However, we have suggested that source hashing has advantages over hashing in terms of reduced computation and the avoidance of layer violations. Source hashing has been proposed by one of us [Var94] as part of a general family of schemes that use additional index fields in packets to reduce the cost of lookups. Thus source hashing is a general technique, and applicable in other contexts; more examples can be found in [Var94].

Lastly, we note that deficit round robin 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. We hope that deficit round robin scheduling will be a tool that our readers will use in other contexts.

References

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings of the Sigcomm '89 Symposium on Communications Archtectures and Protocols*, 19(4):1–12, September 1989. part of ACM Sigcomm Computer Communication Review.

[McK91] Paul E. McKenney. Stochastic fairness queueing. In *Internetworking: Research and Experience Vol.2, 113-131*, January 1991.

[Nag87] John Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, COM-35(4), April 1987.

[Par93] Craig Partridge. *Gigabit Networking.* Addison-Wesley, Reading,MA, 1993.

[Var94] G. Varghese. Trading packet headers for packet processing. Computer Science Technical Report WUCS-94-16, Washington University, July 1994.

[Zha91] Lixia Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. *acmtocs*, 9(2):101–125, May 1991.