On the Efficient Implementation of Fair Queueing

Srinivasan Keshav

Computer Science Division, Department of EECS,
University of California, Berkeley,
Berkeley, CA 94720, USA.

Abstract

The performance of packet switched data networks is greatly influenced by the queue service discipline in routers and switches. In particular, the Fair Queueing discipline [1] has several advantages over the traditional first-come-first-served discipline. This paper studies data structures and algorithms for the efficient implementation of Fair Queueing. We present a novel performance evaluation methodology and use it to evaluate the relative merits of several alternate implementations.

Keywords: Computer networks, Fair Queueing, Implementation

Introduction

Network performance is quite sensitive to the queue service discipline implemented at the output trunks of routers and switches. While most current implementations are of the first-come-first-served discipline, recent work has shown that the Fair Queueing (FQ) discipline [1] provides better performance. Thus, there has been considerable interest in studying the theoretical and practical aspects of the algorithm [2-7].

Earlier work on Fair Queueing discussed its properties and its behavior in simulated networks. However, no particular implementation strategy was suggested. If future networks are to implement the discipline, it is desirable to study efficient implementation strategies. Thus, this paper examines data structures and algorithms for the efficient implementation of Fair Queueing.

We begin by reviewing the Fair Queueing Algorithm. After pointing out its three components, we study the efficient implementation of each component. The element that critically affects implementation performance is a bounded size priority queue. In the rest of the paper, we develop a novel technique to study average case performance of data structures, and use it to compare several priority queue implementations. Our results indicate that cheap and efficient implementations of Fair Queueing are possible. Specifically, if packet loss can be avoided, an ordered link list implements a bounded priority queue simply and efficiently. If losses can occur, then explicit per-conversation queues provide excellent performance.

The Fair Queueing Algorithm

The algorithm is implemented at the server that schedules packets on the output trunk of a router or switch in a store-and-forward packet network. Assume for the moment that data from each source-destination pair (a *conversation*) can be distinguished, and is stored in a logically distinct per-conversation queue. Consider a hypothetical service discipline where one bit from each logical queue is served in round robin order. This discipline allocates bandwidth fairly, since at every instant in time each conversation gets exactly its fair share of the trunk bandwidth. Let $R(t)$ denote the number of rounds of service made in this hypothetical service discipline up to time t. Let $N_{ac}(t)$ denote the number of active conversations, i.e. those that have bits in their queue at time t. Then, $\partial R / \partial t = L / N_{ac}(t)$, where L is the linespeed of the gateway's outgoing line. Clearly, a packet of size P whose first bit gets serviced at time t_0 will have its last bit serviced *P* rounds later, at time *t* such that $R(t) = R(t_0) + P$.

Let t_i^α be the time that packet *i* belonging to conversation α arrives at the gateway. Define the variables S_i^α and F_i^α to be the values of $R(t)$ when the packet started and finished service, and let P_i^α denote the size of the packet. Then:

$$F_i^\alpha = S_i^\alpha + P_i^\alpha$$
$$S_i^\alpha = MAX(F_{i-1}^\alpha,\ R(t_i^\alpha))$$

Sending packets in a bit-by-bit round robin fashion is impractical, and the Fair Queueing algorithm can be thought of as a way to emulate the hypothetical service discipline by a practical packet-by-packet transmission scheme. We define a conversation to be *active* whenever $R(t) \leq F_i^\alpha$ for $i = MAX(j | t_j^\alpha \leq t)$ (i.e. whenever the round number is less than the largest finish number of all packets queued for a conversation). The packet-by-packet transmission algorithm is simply that whenever a packet finishes transmission, the next packet sent is the one with the smallest value of F_i^α. It can be shown that over sufficiently long conversations, this packetized algorithm asymptotically approaches the fair bandwidth allocation of the BR scheme [2].

The delay and bandwidth allocations are separated by introducing a nonnegative parameter δ, and defining a new quantity, the *bid* B_i^α, by:

$$B_i^\alpha = P_i^\alpha + MAX(F_{i-1}^\alpha,\ R(t_i^\alpha) - \delta)$$

The quantities $R(t)$, $N_{ac}(t)$, F_i^α, and S_i^α are as before, but the sending order is determined by the bid numbers, not the finish numbers. Note that the asymptotic bandwidth allocation is independent of δ, since the finish numbers control the bandwidth allocation. However, the algorithm can give lower delays to packets that arrive at an inactive conversation. The parameter δ controls the extent of this additional promptness.

Summary

Let the *i*th packet from the conversation α, of size P_i^α arrive at the server at time *t*. Let F^α denote the largest finish number for any packet that has ever been queued for conversation α. Then, we compute the packet finish number F_i^α and the packet bid number B_i^α as follows:

```
if( $\alpha$ is active )
        $F_i^\alpha = F^\alpha + P_i^\alpha$ ;
else
        $F_i^\alpha = R(t_i^\alpha) + P_i^\alpha$ ;
endif

$B_i^\alpha = P_i^\alpha + MAX( F^\alpha, R(t_i^\alpha) - \delta^\alpha)$ ;
$F^\alpha = F_i^\alpha$ ;
```

If the packet arrives when there is no more free buffer space, packets are dropped in order of decreasing bid number until there is space for the incoming packet. The next packet sent on the output line is the one with the smallest bid number.

Components of a Fair Queueing Server

It is illustrative to trace a FQ server's actions on packet arrival and departure. When a packet arrives at the server, it first determines the packet's conversation ID α. The server then updates the current round number $R(t)$. The conversation ID is used to index into the server state to retrieve the conversation's finish number F^α and offset δ^α. These are used to compute the packet's finish and bid numbers, F_i^α and B_i^α.

If the output line is idle, the packet is sent out immediately, else it is buffered. If the buffers are full, some buffered packets may need to be discarded. On an interrupt from the output line indicating that the next packet can be sent, the packet in the buffer with the smallest bid number is retrieved and transmitted.

From this description, we identify three major components of a FQ implementation: bid number computation, round number computation and packet buffering. We discuss each component in turn.

Bid number computation

A FQ server maintains, as internal state, the finish number F^α and the offset δ^α of each conversation α passing through that gateway. An implementor has to make two design choices: determining the ID of a conversation, and deciding how to access the state for that conversation.

The choice of the conversation ID depends on the entity to whom fair service is granted (see the discussion in Reference [1]), and the naming space of the network. For example, if the unit is a transport connection in the IP Internet, one such unique identifier is the tuple (source address, destination address, source port number, destination port number, protocol type). The elements of the tuple can be concatenated to produce a unique conversation ID. For virtual circuit based networks, the Virtual Circuit ID itself can be used as the conversation ID.

Note that for the IP Internet, one cannot always use the source and destination port numbers, since some protocols do not define them. For example, if an IP packet is generated by a transport protocol such as NetBlt [8], this information may not be available. An engineering decision could be to recognize port numbers for some common protocols and use the IP (source address, destination address) pair for all other protocols. This may result in some unfairness since transport connections sharing the same address pair are treated first-come-first-served.

The conversation ID is used to access a data structure for storing state. Since IDs could span large address spaces, the standard solution is to hash the ID onto a index, and the technology for this is well known [9]. Recently, a simple and efficient hashing scheme that ignores hash collisions has been proposed [5]. In this approach, some conversations could share the same state, leading to unfair service, since these conversations are served first-come-first-served. However, this is attenuated by occasionally perturbing the hash function.

Round number computation

The round number at a time t is defined to be the number of rounds that a bit-by-bit round robin server would have completed at that time. To compute the round number, the FQ server keeps track of the number of active conversations $N_{ac}(t)$, since the round number grows at a rate that is inversely proportional to N_{ac}. However, this computation is complicated by the fact that determining whether or not a conversation is active is itself a function of the round number.

Consider the following example. Suppose that a packet of size 100 bits arrives at time 0 on conversation A, and let $L = 1$. During [0,50), since $N_{ac} = 1$, and $\partial R(t) / \partial t = 1/N_{ac}$, $R(50) = 50$. Suppose that a packet of size 100 bits arrives at conversation B at time 50. It will be assigned a finish number of 150 (= 50 + 100). At time 100, $P_0{}^A$ has finished service. However, in the time interval [50, 100), $N_{ac} = 2$, and so $R(100) = 75$. Since $F^A = 100$, A is still active, and N_{ac} stays at 2. At $t = 200$, $P_0{}^B$ completes service. What should $R(200)$ be? The number of conversations went down to 1 when $R(t) = 100$. This must have happened at $t = 150$, since $R(100) = 75$, and $\partial R(t)/\partial t = 1/2$. Thus, $R(200) = 100+50 = 150$.

Note that each conversation departure speeds up the $R(t)$, and this makes it more likely that some other conversation has become inactive. Thus, it is necessary to do an *iterated deletion* of conversations to compute $R(t)$, as shown in Figure 1.

```
/* $F$, $\Delta$ and $N$ are temporary variables */
$N = N_{ac}(t_{chk})$;
do:
        $F = MIN(F^{\alpha} | \alpha \ is \ active)$;
        $\Delta = t - t_{chk}$;
        if ($F \le R_{chk} + \Delta * L/N$) {
                declare the conversation with $F^{\alpha} = F$ inactive;
                $t_{chk} = t_{chk} + (F - R_{chk}) * N/L$;
                $R_{chk} = F$;
                $N = N - 1$;
        }
        else {
                $R(t) = R_{chk} + \Delta * L / N$;
                $R_{chk} = R(t)$;
                $t_{chk} = t$;
                $N_{ac}(t) = N$
                exit;
        }
od
```

Figure 1: Round number computation

The server maintains two state variables, t_{chk} and $R_{chk} = R(t_{chk})$. A lower bound on $R(t)$ is $R_{chk} + L/N_{ac}(t_{chk}) * (t - t_{chk})$, since N_{ac} is strictly non-increasing in $[t_{chk}, t]$. If some F^{α} is less than this expression, then conversation α has become inactive some time before time t. We determine the time when this happened, checkpoint the state at that time by updating the t_{chk}, R_{chk} pair, and repeat this computation till no more conversations are found to be inactive at time t.

Round number computation involves a MIN operation over the F^{α}s, which suggests a simple scheme for efficient implementation. The F^{α}s are maintained in a heap, and as packets arrive the heap is adjusted (since F^{α} is monotonically increasing for a given α, this is necessary for each incoming packet). This takes time O(log $N_{ac}(t)$) per operation. However, it only takes constant time to find the minimum, and so each step of the iterated deletion takes time O(log $N_{ac}(t)$) (for readjusting the heap after the deletion of the conversation with the smallest finish number).

In related work, a heuristic for computing the round number has been proposed [6]. In this scheme, the round number is set to the finish number of the packet currently being transmitted, and all packets with the same finish number are served first-come-first-served. If this heuristic (or a small variant) is acceptable, the round number can be easily computed.

Packet buffering

FQ defines the packet selected for transmission to be the one with the smallest bid number. If all the buffers are full, the server drops the packet with the largest bid number (unlike the algorithm in Reference [1], this buffer allocation policy accounts for differences in packet lengths). The abstract data structure required for packet buffering is a *bounded heap*. A bounded heap is named by its root, and contains a set of packets that are tagged by their bid number. It is associated with two operations, `insert(root, item, conversation_ID)` and `get_min(root)`, and a parameter, MAX, which is the maximum size of the heap.

`insert()` first places an item in the bounded heap. While the heap size exceeds MAX, it repeatedly discards the item with the largest tag value. We insert an item before removing the largest item since the inserted packet itself may be deleted, and it is easier to handle this case if the item is already in the heap. To allow this, we always keep enough free apace in the buffer to accommodate a maximum sized packet. `get_min()` returns a pointer to the item with the smallest tag value and deletes it.

Determining a good implementation for a bounded heap is an interesting problem. There are two broad choices.

I Since we are interested only in the minimum and maximum bid values, we can ignore the conversation ID, and place packets in a single homogeneous data structure.

II We know that within each conversation, the bid numbers are strictly monotonically increasing. This fact can be used to do some optimization.

It is not immediately apparent what the best course of action should be, particularly since per-conversation queueing is computationally more expensive. Thus, we did a performance analysis to help determine the best data structure and algorithms for packet buffering. The next sections describe some implementation alternatives, evaluation methodology and the results of the evaluation.

Buffering Alternatives

We considered four buffering schemes: an ordered linked list (LINK), a binary tree (TREE), a double heap (HEAP), and a combination of per-conversation queueing and heaps (PERC). We expect that the reader is familiar with details of the list, tree and heap data structures. They are also described in standard texts such as References [10, 11].

Ordered List

Tag values usually increase with time, since bid numbers are strictly monotonic within each conversation. This suggests that packets should be buffered in a ordered linked list, inserting incoming packets by linearly scanning from the largest tag value. Monotonicity implies that most insertions are near the end, and so this reduces the number of link traversals required.

Binary Tree

We studied a binary tree, since this is simple to implement and has good average performance. Unfortunately, monotonic tag values can skew the tree heavily to one side, and the insertion time can becomes almost linear. This skew can be removed by using self-balancing trees such as AVL trees, 2-3 trees or Fibbonacci trees. However, the performance of the self-balancing trees is comparable to that of a heap, since operations on balanced trees as well as heaps require a logarithmic number of steps. Since we do study heaps, we have not evaluated self-balancing trees explicitly. Our performance evaluation of heaps will also be representative of the results for self-balancing trees.

Double Heap

A heap is a data structure that maintains a partial order. The tag value at any node is the supremum (or infremum) of all the tags that lie in the subtree rooted at that node. Since we require both the minimum and the maximum elements in the heap, we maintain two heaps (implemented as arrays) and cross pointers between them. The code for implementing double heaps is presented in Appendix 1.

We know that within a conversation, bid numbers are strictly monotonic. So, we queue packets per conversation, and keep two heaps keyed on the bid numbers of the head and tail of the queue for each conversation. `insert()` adds a packet to the end of the per channel queue and updates the max heap. `get_min()` finds the packet with smallest bid number from the min heap and dequeues it.

Performance Evaluation

The performance of a data structure is measured by the cost of performing an elementary operation, such as an insertion or a deletion of an element, on it. Traditionally, performance has been measured by the asymptotic *worst* case cost of the operation as the size of the data structure grows without bound. For example, the insertion cost into a ordered list of length N is O(N), since in the worst case we may need to traverse N links to insert an item into the list.

How should we measure the performance of the four buffering data structures for the `insert()` and `get_min()` operations? Since constant work is needed to add or delete a single item at a known position to any data structure, the unit of work for linked lists and trees is a link traversal and for heaps is a swap of two elements. For linked lists and trees, the time for `get_min()` is a constant, and for the other two data structures, it is comparable to the insertion time. Thus, an appropriate way to measure the performance of the data structures is to measure the number of links of the data structure that are traversed, or the expected number of swaps, during an `insert()` operation. Let *B* denote the number of buffers in a gateway, and let *N* denote the and the number of conversations present at any time. Table 1 presents well known results for the performance of the data structures described above for the `insert()` operation.

	Best	Worst	Average (Uniformly random workload)
LINK	O(1)	O(B)	O(B)
TREE	O(1)	O(B)	O(log(B))
HEAP	O(log(B))	O(log(B))	O(log(B))
PERC	O(log(N))	O(log(N))	O(log(N))

Table 1: Theoretical insertion costs

While the asymptotic worst case cost is an interesting metric, we feel that it is also desirable to know the average cost. However, average case behavior is influenced by the workload (the exact sequence of `insert` and `get_min` operations) that is presented to the data structure. Thus the best that we can do analytically is to assume that the workload is drawn from some standard distribution (uniform, gaussian, etc.), and compute the expected cost. We believe that this is not adequate. Instead, we use a general analysis methodology that we think is practical, and has considerable predictive power.

We first parameterize the workload by some (small) number of parameters. The parameters are then fed to a realistic network simulator to create a trace of the workload for that parameter point. Then, we implement the data structure and associated algorithms, and measure the average performance over the trace length. This enables us to associate an average performance metric at that point in the workload parameter space. By judicious exploration of the parameter space, it is possible to map out the average performance as a function of the workload, and thus extrapolate performance to regions of the space that are not directly explored.

In our opinion, this methodology avoids a significant difficulty in average case analysis, that is, reliance on unwarranted assumptions about the workload distribution. Also, by mapping algorithm performance onto the workload space, it enables a network designer to choose an appropriate algorithm given the operating workload.

The drawback with this method is that it requires a realistic network simulator, and considerable amounts of computing time. Further, parameterizing the workload and exploration of the state space are more of an art than a science. However, we feel that these drawbacks are more than compensated for by the quality of the results that can be obtained.

Evaluation Results

We chose the scenario of Figure 2 for detailed exploration.

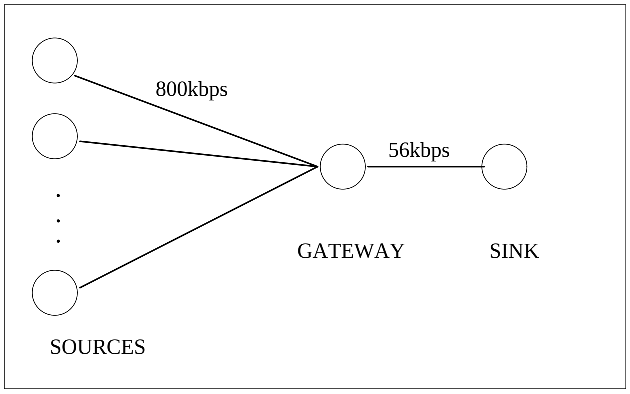
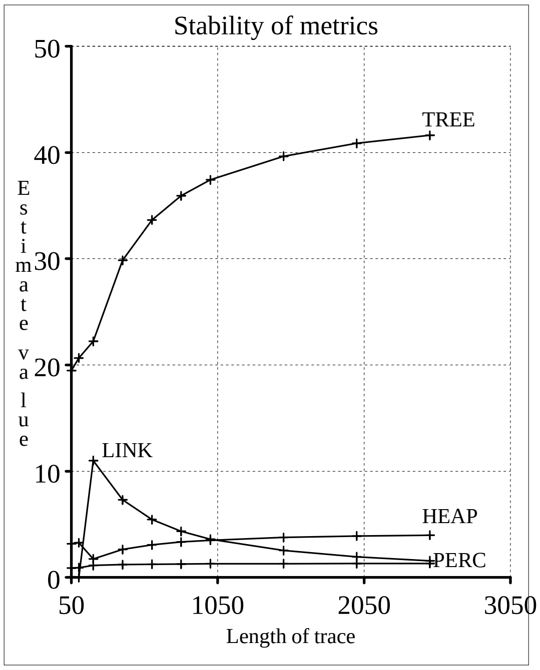
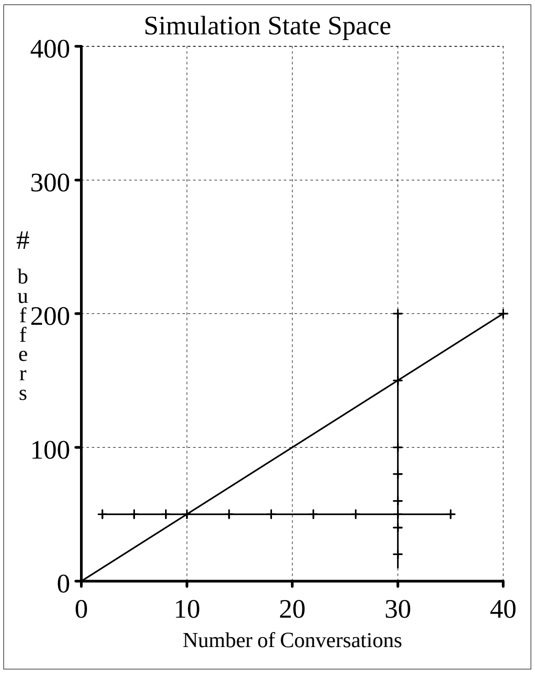


Figure 2: Simulation scenario


Stability of metrics
50
40
30
20
10
0
Estimate value
TREE
LINK
HEAP
PERC
50
1050
2050
3050
Length of trace


Simulation State Space
400
300
200
100
0
buffers
0
10
20
30
40
Number of Conversations