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 $R(t) \le F_i^\alpha$ for $i = MAX(j \,|\, t_j^\alpha \le t)$ (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(α *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, Δ and N are temporary variables */
N = N_ac(t_chk);
do:
        F = MIN(F^α | α is active);
        Δ = t − t_chk;
        if (F ≤ R_chk + Δ * L/N) {
                declare the conversation with F^α = F inactive;
                t_chk = t_chk + (F − R_chk) * N/L;
                R_chk = F;
                N = N − 1;
        }
        else {
                R(t) = R_chk + Δ * 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.

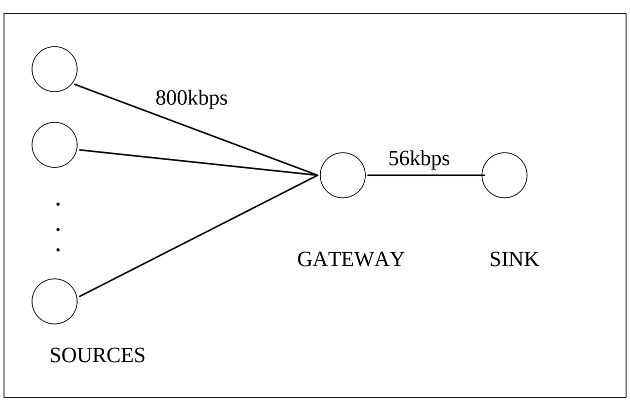
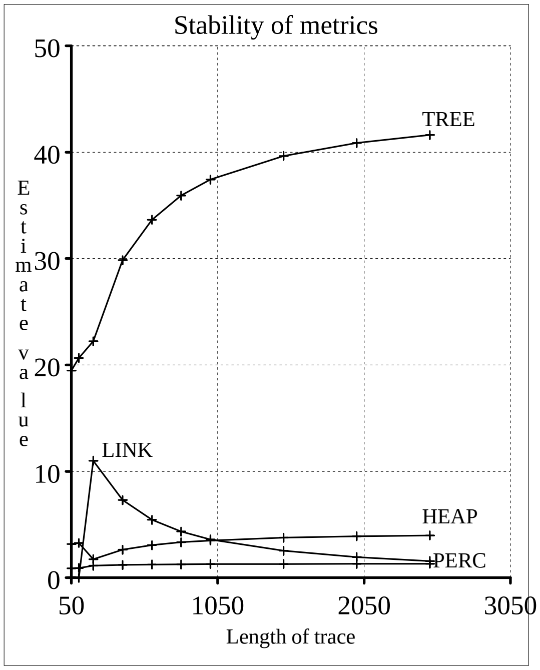


Figure 2: Simulation scenario



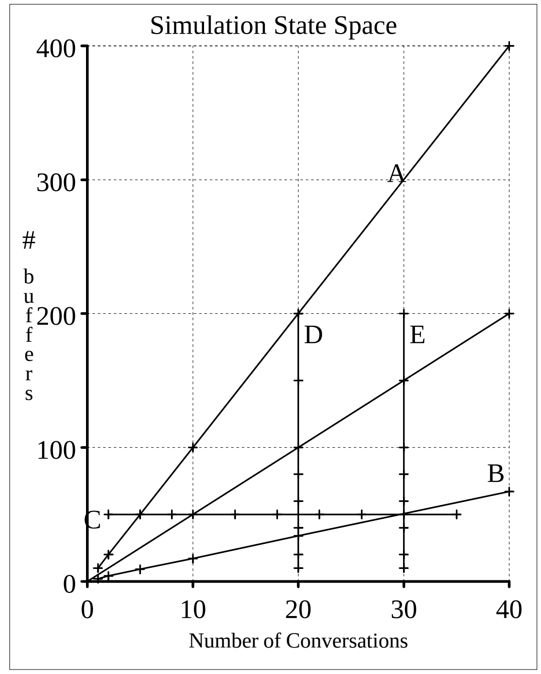


Figure 3 *Figure 4*

The gateway serves multiple sources (each of which originates one conversation) that share two common resources: the bandwidth of the output (trunk) line, and buffers in the gateway. Since there are no inter-trunk service dependencies, it suffices to model a single output trunk. Further, by changing the number of sources, and the number of buffers, it is possible to drive the system into congestion, something that we want to study. Finally, it is simple enough that it can be easily parameterized. Thus, our choice.

Note that we do not introduce any ‘non-conformant’ traffic in the sense of [6], since we wish to explore design decisions for well behaved sources only. If the network is expected to carry non-conformant sources as well, then an performance evaluation similar to the one described here probably needs to be carried out.

The simulated sources obey the DARPA TCP protocol [12] with the modifications made by Jacobson [13]. They have a maximum window size of W each. By virtue of the flow control scheme, each source dynamically increases its window size, till either a packet is dropped by the gateway (leading to a timeout and a retransmission) or the window size reaches W. It is clear that the gateway cannot be congested if

$$W * N \leq B$$

If the network is not congested, then each source behaves nearly independently, and the workload is deterministic. When there is congestion, retransmissions and packet losses dramatically change the workload. Thus, one parameter for the workload is the ratio N/B. We also expect the workload to change as the number of conversations N increases. Thus, keeping W fixed, the two parameters that determine the workload are N and B.

Following the experimental methodology outlined above, we used the REAL network simulator [14] to generate workload traces for a number of (N, B) tuples. One practical problem was to determine the appropriate trace length. Since generating a trace takes a considerable amount of computation, we decided to generate the shortest trace for which the cost metrics for all the four implementations stabilized. For simplicity, we determined this length for a single workload, with $N = 10$, $B = 200$, and generated a trace for 2500 seconds of simulated time. We then plotted the four cost metrics as a function of the trace length (Figure 3). We find that at 2500 seconds, all the metrics are no more than 10% away from their asymptotes. Since we only wanted to make qualitative cost comparisons, we generated each trace for 2500 seconds.

The (N, B) state space was explored along the five axes (labeled A through E) shown in Figure 4. Each '+' marks a simulation, there were a total of 35 simulations. Cost metrics for each implementation were determined along each axis. Axis A is the underloaded axis - along every point in the axis the gateway is lightly loaded, that is $W * N < B$. Symmetrically, axis B is the overloaded axis. Axes C, D and E are partly in the underloaded regime, and partly in the overloaded regime. Thus, congestion-dependent transitions in relative cost of the implementations occur along these axes. The axis between axes A and B marks $W*N = B$.

Figures 5 and 6 show the average insertion cost along each of the five axes for each implementation. This is computed as

$$\#\ elementary\ operations / \#\ insertions\ in\ the\ trace$$

where an elementary operation is the traversal of a single link or a single heap exchange. All Y axes are drawn to logarithmic scale. Conceptually, one can imagine that for each implementation, there is a performance surface overlaying the workload space. Figures 5 and 6 represent cross sections of these surfaces as we slice along axes A-E. We can extrapolate the surfaces from these cross sections.

Results

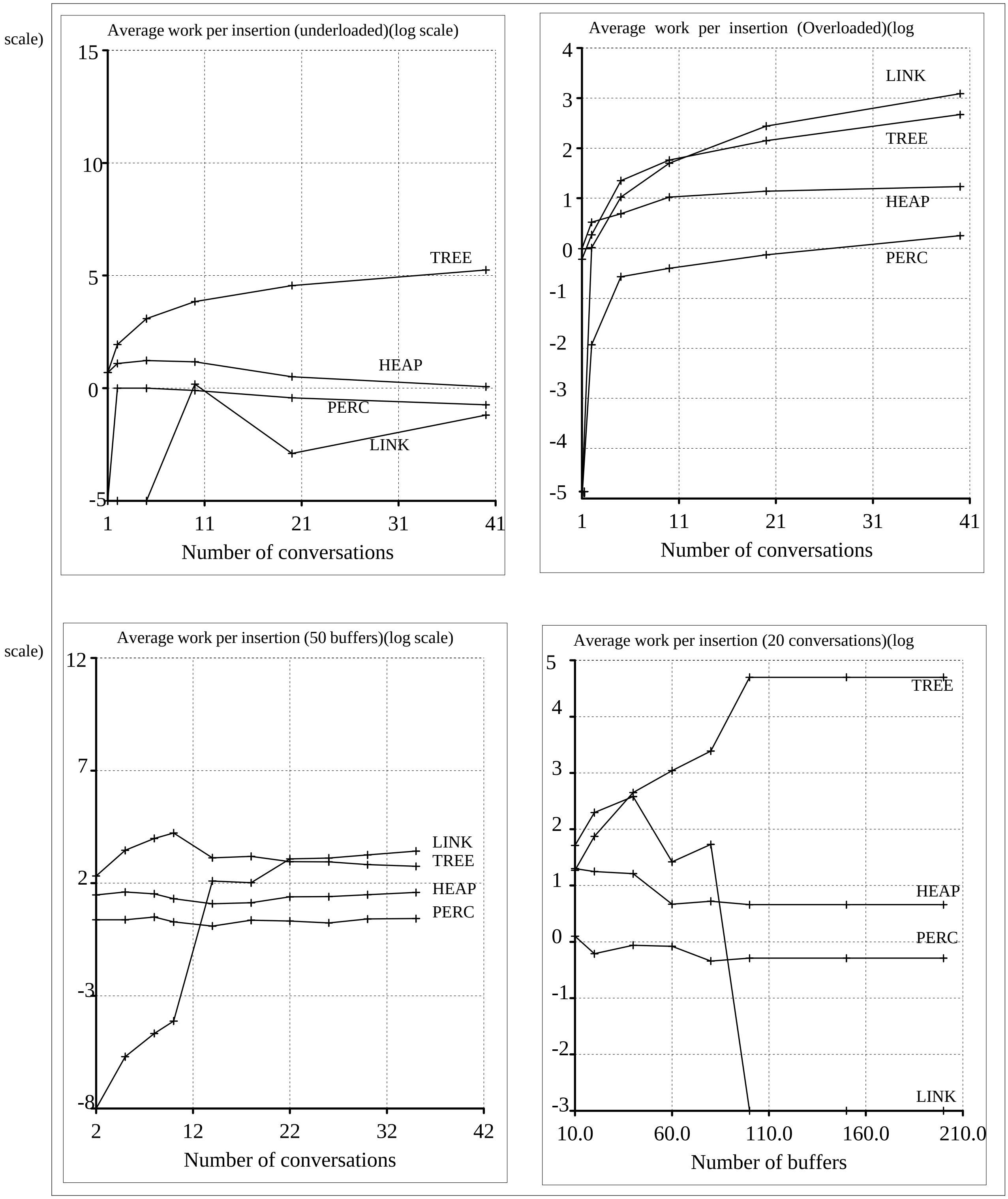


Figure 5: Average cost results

Examination of the surfaces points out several facts:

- The performance surface for all the implementations (except LINK) are generally smooth, with few discontinuities. Thus, extrapolating the curves is meaningful.
- LINK behavior is somewhat erratic, since the insertion cost is is highly dependent on the workload. However, it still has a well defined behavior: in some cases, it is the by far the cheapest implementation, in others, it is clearly the most expensive. Figure 7 divides the workload space into three zones, numbered I-III. In zone I, it is best to use LINK, in zone III, LINK has the worst metric.
- As the number of conversations increases, the average HEAP and PERC insertion cost increases in the overloaded regime and is roughly constant in the underloaded regime.
- The cost metric for PERC is always less than that for HEAP or TREE.
- The cost metric for HEAP is within an order of magnitude of that for PERC in most cases.
- In the underloaded regime, binary trees become skewed, and hence are costly. They perform better in the overloaded regime.
- Average insertion cost for PERC is less than its theoretical average case cost.
- The maximum work done, which is shown for a typical case in Figure 6, is as expected from theory.
- In the underloaded case, HEAP and PERC show a declining trend, but this is offset by a larger increasing trend in the deletion time (not shown here).

Interpretation of Results

The results give several guidelines for FQ implementation. TREE performs the worst in the underloaded regime, and in the overloaded regime, HEAP and PERC are better. Hence, TREE is a bad implementation choice. We will not discuss it further.

Among the other strategies, PERC is always better than HEAP, and both of them have small worst case insertion costs. The worst case work per insertion is bounded by O(log(B)) for HEAP, and by O(log(N)) for PERC. Assuming that a gateway has 32 Mbytes of buffering per trunk line, and that packets are, on the average, 1Kbyte long, there will be at most be on the order of 32K packets in the buffer. The number of conversations will be on the order of the square root of this number, i.e. around 200. With these figures, HEAP requires log(32K) ˜ 15, and PERC requires log(200) ˜ 8 elementary operations. Our simulations (Figure 5) show that in the trace driven simulation, the average work for HEAP and PERC is less than half of the worst case work. Thus, the average cost per insertion for PERC will be more like 4 elementary operations. This is a small price to pay to implement Fair Queueing.

The behavior of LINK (Figure 7) points to another implementation tactic. Note that in region I, LINK has the least cost. If the network designer can guarantee that the system will never enter the overloaded region (for example, by preallocating enough buffers for conversations, as in the Datakit network), then implementing LINK is the best strategy.

One consideration that is orthogonal to the insertion cost is implementation cost. For example, it is clear that an implementing PERC involves much more work than LINK. There are two implementation costs, corresponding to the work that is done independent of the number of elementary operations (static cost), and the work done per elementary operation (dynamic cost).

One simple metric to measure static cost is to measure the code size for `insert()`. We extracted the code for `insert()` and all the functions that it calls, for each implementation and placed it in a file. This file was compiled to produce optimized assembly code (in Unix, cc -S -O -c). We then stripped the file of all assembler directives, leaving pure assembly code. Since this was done on a RISC machine, all instructions have the same cost, and the file length is a good metric of the complexity of implementing a given strategy. Table 2 presents this metric for the four implementations, normalized to the cost of implementing LINK.

Implementation	Static Cost	Dynamic Cost
LINK	1.0	5
TREE	1.1	18
HEAP	2.5	88
PERC	5.5	96

Table 2: Implementation cost

The dynamic cost was determined by examining the optimized assembly code, and counting the number of instructions executed per elementary operation. Table 2 presents the results. We did not specifically concentrate on reducing the number of instructions while writing the source code. We believe that the dynamic cost can be considerably reduced by hand coding in assembly language.

To summarize, we draw four conclusions:

1 Implementing TREE is a bad idea.

2 HEAP provides good performance with low implementation cost.

3 PERC consistently provides the best performance, but has the highest implementation cost.

4 If the network designer can guarantee that the network never goes into overload, LINK is cheap to implement and has the minimum running cost.

Conclusions

In this paper, we have considered the components of a FQ server, and have presented and compared several implementation strategies. Our work indicates that cheap and efficient implementations of FQ are possible. Along with the work done by McKenney [5] and Heybey et al [6], this work provides the practitioner with well defined guidelines for FQ implementation. We hope that these studies will encourage more implementations of Fair Queueing in real networks.

The performance evaluation methodology described here enables realistic evaluation of the average case performance of network algorithms. As LINK shows, this can lead to interesting results. We believe that a similar methodology can be used to evaluate a number of other workload sensitive network algorithms.

Finally, we believe that these results can be extended to other scheduling disciplines that are similar to Fair Queueing, such as the Virtual Clock algorithm [15]. Thus, our work has some generality of application.

Future Work

This paper does not examine hardware implementation of Fair Queueing. Given the need for faster packet processing in high speed networks, this is an obvious direction to pursue.

While our paper presented the means for the cost metric, we ignore the variance. This is because our simulations are completely deterministic. It would be useful to enhance the performance methodology described earlier to determine the variance and confidence intervals.

Acknowledgments

This work was prompted by conversations with D. Presotto and E. Hahne at Bell Labs, Murray Hill. The idea of backward insertion into a linked list is due to D. Presotto. My thanks to Prof. D. Ferrari at UC Berkeley for his advice on performance analysis.

References

1. Demers, A., Keshav, S. and Shenker, S., Analysis and Simulation of a Fair Queueing Algorithm, *Journal of Internetworking Research and Experience,* September 1990, 3-26;. also Proc. ACM SigComm, Sept. 1989, pp 1-12..
2. Greenberg, A. and Madras, N., How Fair is Fair Queueing?, *Proc. Performance 90*, 1990.
3. Shenker, S., Making Greed Work in Networks: A Game-Theoretic Analysis of Gateway Service Disciplines, Preprint, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304., September 1989.
4. Shenker, S., A Theoretical Analysis of Feedback Flow Control, *Proc. ACM SigComm 1990,* September 1990, 156-165.
5. McKenney, P. E., Stochastic Fairness Queueing, *Proc. INFOCOM '90*, June 1990.
6. Heybey, A. T. and Davin, J. R., A Simulation Study of Fair Queueing, *Computer Communications Review 20*, 5 (October 1990), 23-29.
7. Mankin, A. and Ramakrishnan, K. K., Performance and Congestion Control Working Group Report, *Internet Engineering Task Force Meeting*, July 1989.
8. Clark, D. D., Lambert, M. L. and Zhang, L., NETBLT: A Bulk Data Transfer Protocol, RFC-998, Network Working Group, March 1987.
9. Jain, R., A Comparison of Hashing Schemes for Address Lookup in Computer Networks , Tech. Rpt.-593, Digital Equipment Corporation , February 1989.
10. Horowitz, E. and Sahni, S., *Fundamentals of Data Structures* , Prentice Hall, 1981.
11. Knuth, D., Fundamental Algorithms, *Addison-Wesley*, 1973.

12. Postel, J., Transmission Control Protocol, RFC 793, USC Information Sciences Institute, 1981.

13. Jacobson, V., Congestion Avoidance and Control, *Proc. ACM SigComm 1988*, August 1988, 314-329.

14. Keshav, S., REAL : A Network Simulator, Comp. Sci. Dept. Tech. Rpt. 88/472 , University of California, Berkeley, December 1988.

15. Zhang, L., A New Architecture for Packet Switching Network Protocols, *PhD thesis*, Massachusetts Institute of Technology, July 1989.

Appendix I

A double heap consists of a pair of heaps. Since operations on one heap must be reflected in the other, we need pointers between the two instances of an element in the double heap. Since we represent heaps as arrays, pointers are indices, and we implement cross pointers using two integer arrays of indices.

The physical data structures used are four arrays, `min`, `max`, `i_min` and `i_max`. `min` and `max` are the arrays that store the two heaps, one has the minimum element at the root, the other has the maximum. `i_min[k]` is the position in `max` of the `k`th element of `min`. `i_max` is defined symmetrically.

Every move in either heap must update `i_min` and `i_max`. We note that the only time an element is moved is when it is exchanged with some other element. We encapsulate this into an operation `exchg()` that swaps elements in the min or max heap, and simultaneously updates `i_min` and `i_max` so that the pointers are consistent. We actually need two symmetric operations, `min_exchg()` and `max_exchg()` that swap elements in the min and max heap respectively. `min_exchg()` looks like the following:

```
min_exchg(a, b)      /* calls to swap are call by name */
{
swap(min[a], min[b]);
swap(i_max[i_min[a]], i_max[i_min[b]]);
swap(i_min[a], i_min[b]);
}
```

We now prove that this operation preserves pointer consistency, i.e. `i_min[i_max[a]] = a` and `i_max[i_min[a]] = a`. Elements are inserted only in the last (say, `n`th) position in the heap, so the initial pointer positions are: `i_min[n] = i_max[n] = n` . It is easy to see that at the end of each `min_exchg()` operation, the pointers will remain consistent. Hence, by induction, pointers are always consistent.

Given the exchange operation, the rest of the heap operations are simple to implement. Heap insertion is done by placing data in the last element, and sifting up.

```
min_insert(data,num)
/* num is the current size of the heap */
{
ptr = num + 1;
min[ptr] = data;

for (; (ptr/2 >= 1) &&(min[ptr] < min (ptr/2]); ptr /=2)
    min_exchg(ptr, ptr/2);
}
```

Deletion is done by changing both the min and the max heaps, then adjusting them to recover the heap property.

```
min_delete()
{
int save;

min[1] = INFINITY;
save = i_min[1];
```

```
min_exchg(1,num);
min_adjust(1);

max[save] = -1;
max_exchg(save,num);
max_adjust(save);
}
```

Adjusting a heap consists of recursively sifting the marked element up or down as the case may be. Termination in a logarithmic number of steps is assured: because of the heap property, calls either go up the heap or down, and there can be no cycles.

```
min_adjust(a)
{
int smaller, smaller_son;

smaller = a;
if (min[a] < min[a/2]) smaller = a/2;
smaller_son = (min[lson(a)] < min[rson(a)]) ? lson(a) : rson(a);
if (min[smaller_son] < min[a]) smaller = smaller_son;
if (smaller != a)
    {
     min_exchg(a, smaller);
     min_adjust(smaller);        /* recursive call */
    }
}
```