Report Number: WUCS-95-11

1995-01-01

Reliable FIFO Load Balancing over Multiple FIFO Channels

Hari Adieseshu, Gurudatta M. Parulkar, and George Varghese

Link striping algorithms are often used to overcome transmission bottlenecks in computer networks. However, traidtional striping algorithms suffer from two major disadvantages. They provide inadequate load sharing in the presence of variable length packets, and may result in non-FIFO delivery of data. We describe a new family of link striping algorithms that solve both problems. Our scheme applies to packets at any layer (physical, data, link, network, and transport) that work over multiple FIFO channels. We deal with variable sized packets by showing how a class of fair queueing algorithms can be converted into load sharing algorithms. Our transformation... **Read complete abstract on page 2.**

Reliable FIFO Load Balancing over Multiple FIFO Channels

Hari Adieseshu, Gurudatta M. Parulkar, and George Varghese

Complete Abstract:

Link striping algorithms are often used to overcome transmission bottlenecks in computer networks. However, traidtional striping algorithms suffer from two major disadvantages. They provide inadequate load sharing in the presence of variable length packets, and may result in non-FIFO delivery of data. We describe a new family of link striping algorithms that solve both problems. Our scheme applies to packets at any layer (physical, data, link, network, and transport) that work over multiple FIFO channels. We deal with variable sized packets by showing how a class of fair queueing algorithms can be converted into load sharing algorithms. Our transformation results in practical load sharing protocols, and also shows a theoretical connection between two seemingly different problem areas. We deal with the FIFO requirement for two separate cases. If a header (with a sequence number) can be added to each packet, we show how to speed up packet processing by letting the receiver simulate the sender algorithm. If no header can be added (e.g., ATM cells), we show how to provide quasi-FIFO delivery. Quasi-FIFO is FIFO except during occasional periods of loss of synchronization between the sender and the receiver. We argue that quasi-FIFO should be adequate for most applications. To deal with loss of synchronization between the sender and receiver, we present simple recovery protocols. We provide performance analysis, experimental results, and proofs of our assertions.

Reliable FIFO Load Balancing over Multiple FIFO Channels

Hari Adiseshu, Gurudatta M. Parulkar and George Varghese

WUCS-95-11

May 3, 1995

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

Abstract

Link striping algorithms are often used to overcome transmission bottlenecks in computer networks. However, traditional striping algorithms suffer from two major disadvantages. They provide *inadequate load sharing* in the presence of variable length packets, and may result in *non-FIFO delivery* of data. We describe a new family of link striping algorithms that solve both problems. Our scheme applies to packets at any layer (physical, data link, network, and transport) that work over multiple FIFO channels.

We deal with variable sized packets by showing how a class of fair queueing algorithms can be converted into load sharing algorithms. Our transformation results in practical load sharing protocols, and also shows a theoretical connection between two seemingly different problem areas. We deal with the FIFO requirement for two separate cases. If a header (with a sequence number) can be added to each packet, we show how to speed up packet processing by letting the receiver simulate the sender algorithm. If no header can be added (e.g., ATM cells), we show how to provide quasi-FIFO delivery. Quasi-FIFO is FIFO except during occasional periods of loss of synchronization between the sender and the receiver. We argue that quasi-FIFO should be adequate for most applications. To deal with loss of synchronization between the sender and receiver, we present simple recovery protocols. We provide performance analysis, experimental results, and proofs of our assertions.

1. Introduction

Parallel architectures are attractive for two reasons. First, they are essential when scalar architectures with the required performance are simply unavailable. Second, even if scalar solutions with the required performance are available, parallel solutions may offer better price-performance and better scalability. The first reason is the motivation for highly parallel supercomputers such as the Connection Machine; the second reason is the motivation for replacing expensive, high performance disks with arrays of inexpensive disks. Parallel solutions, however, often have a cost for synchronization (e.g., the need to keep multiprocessor caches coherent) and fault-tolerance (e.g., the need for redundant disks in disk arrays).

Similar considerations apply to computer networks because of transmission and processing bottlenecks. For example, consider a supercomputer (e.g., a CRAY) that can easily saturate existing Local Area Networks (LANs). If more performance is required between two CRAYs than can be provided by a single LAN, systems builders often resort to "striping" data across multiple adaptors and multiple LANs between the two computers. A second example is a wide-area network connecting campuses that are internally wired with 100 Mbps FDDI LANs. If the only available wide area links are 1.5 Mbps T1 lines, then many network designers resort to multiple T1 connections between campuses. In these examples, the parallel solution of using several slow speed channels is the *only* solution available to the designer. However, in the second example it may also be that a higher speed SONET link is available between campuses, but the cost is prohibitive. In that case, the parallel solution may be favored because it has a better price-performance ratio.

For these and other reasons, channel striping (also known as *load sharing*, or *inverse multiplexing*) has frequently been considered in computer networks. This trend is likely to continue because bandwidth continues to be less expensive in LAN environments than in WAN environments, and the processing power of workstations continues to grow faster than the speed of reasonably priced LANs.

However, just as in other parallel solutions, there are some basic synchronization and fault-tolerance issues that are inherent to channel striping algorithms. For example, if a FIFO (First-In-First-Out) stream of packets is striped across multiple channels, the packets may be physically received out of order at the receiver because of different delays (often called *skews*) on the different channels. In many applications (e.g., ATM) the receiver must reconstruct the sequence of packets sent by the sender from the parallel packet streams arriving on the channels. This adds a basic synchronization cost. Clearly, the channel striping algorithm must also be resilient to common faults such as occasional bit errors and link crashes.

As we will see in the next section, earlier solutions to the synchronization and fault-tolerance problems were either expensive or were dependent on assumptions[1] that made them infeasible in certain important ap-

[1] such as the ability to add sequencing information to packets sent over channels

plication domains. This paper, on the other hand, describes a new family of channel striping algorithms that are both general and efficient. Our algorithms are based on a combination of two (we believe) novel ideas: *fair load sharing* and *logical FIFO reception.*

To allow our algorithms to be employed in as wide a context as possible, we use a broad definition of a channel. For the rest of this paper, we define a *channel* to refer to a logical FIFO path which can exist at either the physical, data link, network, or transport layer. We use *packets* to refer to the atomic units of exchange between two entities communicating across a channel. The generic channel striping configuration is depicted in Figure 1.

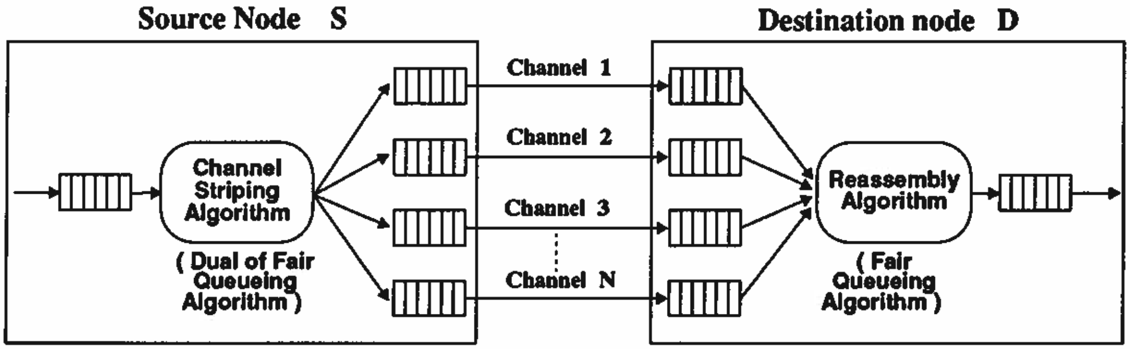
Source Node S
Destination node D
Channel Striping Algorithm
(Dual of Fair Queueing Algorithm)
Channel 1
Channel 2
Channel 3
Channel N
Reassembly Algorithm
(Fair Queueing Algorithm)

Figure 1: Channel striping configuration

The simplest example of a channel at the data link layer is a point-to-point fiber link that connects two high speed devices, where the two devices could be workstations, switches, routers, or bridges. A somewhat less obvious example of a Data Link channel is a LAN like an Ethernet or a Token Ring; although these LANs are multiaccess LANs and do not guarantee FIFO delivery across sources, they *do* guarantee FIFO delivery between a given source and a given destination. Thus a high speed workstation may stripe packets across several cheap Ethernet adaptors that are each connected to a separate Ethernet.

At the network layer, a channel is an end-to-end pipe that guarantees FIFO delivery across the network. An important example of such a "network" channel is a virtual circuit provided by an X.25 or an ATM network. For example, it may well be true that in the wide area context, lower bandwidth ATM virtual circuits have better price-performance. This would make end-to-end striping across wide area ATM networks attractive. Such striping could either be done at the packet or at the cell layer; we will argue later that packet striping across ATM virtual circuits has some attractive properties. Even in datagram networks, it may be possible to construct "network" channels. For example, we could use strict IP source routing to set up multiple paths between two IP endpoints, and use routers that process packets in FIFO order. While the latter example is somewhat contrived, it does illustrate the generality of our setting.

Finally, since most transport protocols like TCP provide a stream service, it is possible to think of a channel as a transport connection. For example, if a network adaptor card implements the protocol stack up to the transport layer, then a fast CPU may achieve a higher throughput by sending data across multiple adaptors.

We will return to a more detailed discussion of these applications in the last section. For now, we note that the most important application of our techniques appear to be at the data link layer and at the network layer across virtual circuits.

1.1. Model

Figure 1 illustrates the model used to describe our algorithms. There are N channels between the source node S and the destination node D. For simplicity, we consider the traffic in only one direction. Exactly the same analysis and algorithms apply for the reverse direction. Node S implements the striping algorithm to stripe its outgoing traffic across the N channels, and node D implements the reassembly algorithm to combine the traffic from the N channels into a single stream. Since the most interesting metric of a channel striping algorithm is throughput, we will often assume, for the purpose of analysis, that the source S is fully loaded — i.e., it always has packets to transmit. However, our algorithms are general enough to work for any traffic pattern offered to the source.

Let the bandwidth of the i^{th} channel be denoted by B_i. We will sometimes distinguish between the *rated bandwidth* of a channel and the *actual bandwidth* of a channel. For example, a network channel may exhibit considerable disparity between rated and actual bandwidths. Unless otherwise stated, B_i refers to the rated bandwidth of the i^{th} channel. All channels are assumed to be FIFO. Channels can be subject to packet loss and corruption. Note that some channels such as reliable data link (e.g., HDLC links) or transport (e.g., TCP connections) can be considered to be lossless. However, our algorithms are robust enough to deal with channels that lose packets. Channels that occasionally deviate from FIFO delivery can be handled by our model, as such lapses in FIFO delivery can be modeled by burst errors. Finally, *we allow for the fact that the end-to-end latency across each channel is potentially different and can vary on a packet to packet basis.* Note that it is important to take variable latency into account in order to model network channels. This also rules out simple solutions to the reordering problem based on skew compensation, if the skew cannot be bounded or characterized.

1.2. Properties of a Channel Striping Scheme

We list desirable properties of a channel striping scheme:

- **Fair Load sharing with variable sized packets:** Data should be striped across the channels (which may have differing capacities) such that the bits transmitted across each channel is proportional to the capacity of the channel. The load sharing property should hold even when the packets are of variable size.

- **FIFO delivery of data:** The channel striping scheme should ideally provide FIFO delivery of data at the remote end. For example, if we stripe packets between a pair of bridges, FIFO delivery is important because many LAN protocols will work poorly if data is delivered out of order. While reordering packets is not catastrophic in datagram networks, current protocol implementations and optimizations make FIFO delivery a desirable feature. Some transport protocols do not buffer out of order packets, while other protocols implement optimizations (e.g., TCP header prediction logic [Jac90]) which work best when packets are delivered in sequence. Also, while FIFO delivery is important for optimal protocol performance in datagram networks, it is an explicit requirement for ATM networks.

- **Ability to work over many types of channels:** The striping algorithms should ideally apply to a wide variety of channel implementations. For instance, the algorithms should work over channels that have dynamically varying skews. Such skews can be caused by varying path lengths, the presence of multiplexing equipment[2], and queuing and processing delays. The algorithms should also work over lossy channels with varying bandwidths.

- **Ability to work over existing channels:** In many striping problems, the channels are obtained from existing equipment and cannot be modified. In such cases, the systems designer does not have the freedom to modify the link hardware or packet formats. As described earlier, the striping algorithms should apply to channels at all layers.

- **Robustness:** The striping scheme must be robust enough to recover from bit and burst errors, and packet losses caused by lossy channels. It must also be able to handle channels failures by effectively ignoring channels that are performing poorly, and automatically adding in channels that recover after a failure.

- **Efficiency:** The striping scheme should be simple enough not to impose any significant computational or hardware overhead. Since current protocol processing is highly streamlined and takes only a few hundred instructions or less, a channel striping algorithm should take an order of magnitude less complexity to execute–preferably in the order of a few tens of instructions or less. Alternatively, it should be amenable to a simple hardware implementation, if implemented at the data link or physical layers.

1.3. Existing Channel Striping Algorithms

The simplest scheme for channel striping is *round-robin striping* — the source sends packets in round robin order on the channels. This scheme, though simple, provides for neither load sharing with variable sized packets,

[2] For example, SONET links

nor FIFO delivery without packet modification (to add sequencing information). Consider a simple scenario in which the sender communicates with the receiver over two channels and the senders queue contains big and small packets in alternation. It is clear that all the big packets will be sent over one link, and the small ones over the other, so the fair load sharing property does not hold. Also, since the channels may have varying delays for reasons explained previously, the physical arrival of packets at the receiver may differ from their logical ordering. Without sequencing information in the packets, FIFO order cannot be guaranteed in the presence of dynamically varying skews

While simple round robin may not guarantee FIFO delivery, we can augment it by adding a header containing a sequence number to every packet. We obtain FIFO delivery by using this header at the receiver to resequence the incoming packets. This violates two of our goals. The most important violation is the requirement to work over existing channels which may not allow the addition of such sequencing information since the packet sizes and headers are already fixed. For example, in ATM networks, where the cell size is fixed at 53 bytes, it appears difficult to add extra headers to cells (e.g., to stripe between two switches) and yet use existing equipment. Even channels that allow variable sized packets, (e.g., Ethernets) typically have a restriction on the maximum packet size (e.g., 1500 for Ethernet). Thus we cannot add an extra header if the packets that the source wishes to send are already maximum sized packets. Finally, even if it were possible to add a sequence number, the sender and receiver processes have to incur overhead for buffer manipulation (to add and delete sequence numbers) and for sequence number bookkeeping (especially at the receiver).

Both the variable packet size problem and the FIFO problem can be solved if the channel striping algorithm can modify the equipment (typically hardware) at the endpoints of a channel. For instance, bit or byte interleaving is often done at the hardware level using devices known as inverse multiplexers. Inverse multiplexers which operate on 56 kbps and 64kbps circuit switched channels are commercially available. Recently, an industry consortium, called BONDING [Dun94][Fre94][Gro92] was formed and it has issued standards for a frame structure and procedures for establishing a wideband communications channel by combining multiple switched 56 and 64-kbps channels through the use of an inverse multiplexer.

In the BONDING scheme, the packets coming in to the inverse multiplexer at the sending end are split into fixed size chunks, or frames, which are then transmitted in round robin order across the multiple channels. At the receiver, the longest delay path is chosen as a reference, and appropriate delays are added to the data from other paths to allow interleaving of frames at the receiver in the same order as transmitted. Each frame is also numbered using sequence numbers, and this is used for resynchronization in case of loss of synchronization at the receiver. While this scheme provides load balancing with variable sized packets and FIFO delivery, it violates two of our goals. It requires special equipment to be inserted at the channel endpoints, which can be expensive and even infeasible for existing channels. Secondly, the sequence number scheme potentially requires extra complexity at both the sending and the receiving endpoints.[3]

The BONDING scheme also relies on a frame structure which requires synchronized frame timings between sender and receiver. A synchronized frame structure is helpful because it allows the sender to put data in predetermined positions with the frames on each channel; the receiver can then pick data from received frames based on this predetermined order. Errors in synchronization only affect a single frame. This solution is implicitly used, for instance, in [HS94]. However, such schemes essentially rely on *robust frame synchronization between sender and receiver that is provided by the physical layer.* Once again such robust frame synchronization is simply unavailable on a number of interesting channels.

Finally, Druschel et. al. [DP94] describes an actual example of cell striping over ATM channels. A single packet is sent as a number of "minipackets" on each channel and a parallel reassembly of the packets is done at the receiver. However, this requires modifying the packet[4] and the designers report that this scheme was difficult to implement with the existing processor on the network adaptor. The paper also discusses two other possible striping schemes. One scheme involves adding sequence numbers to the AAL headers of each cell. This was ruled out for adding significant complexity to the reassembly code. In the other scheme, if the skew on all channels can be made identical (skew compensation), then the receiver can remove packets in the same order as the sender. The schemes described by Druschel et. al. rely on assumptions on channel properties (e.g., no dynamic skew variation) or on modifications to packet formats.

Table 1 summarizes the features of the various solutions to channel striping. The last two rows describe our solution, which is explained in the following sections.

2. Overview of Solution Components and Plan of the Paper

Our goals require that any viable solution for channel striping should achieve load sharing with variable sized packets, and ensure FIFO delivery in the face of dynamically varying channel skews.

To provide load sharing in the presence of varying length packets, we propose the use of *fair load sharing* algorithms. We show such fair load sharing algorithms can be automatically derived by transforming a class of *fair queuing* algorithms. Fair queuing refers to multiplexing packets from several queues onto a single line in a fair way; the problem has been well studied in an independent context. In Section 3, we develop a criterion for this transformation, and provide two instances of fair load sharing algorithms. Our algorithms are simple enough to provide efficient implementations in either hardware or software.

[3] However, we have been unable to determine the reassembly algorithm used by the BONDING inverse multiplexers; thus our first objection is the major reason for looking for schemes other than BONDING.

[4] more precisely, it requires adding an additional framing bit in each minipacket

Scheme	**FIFO delivery**	**Variable length packets**	**Generality**	**Efficiency**
Round-Robin, no header	May be non-FIFO (based on receiver implementation)	Bad load sharing for variable sized packets	At all levels	Simple
Round-Robin with header	Guaranteed FIFO	Bad load sharing for variable packets	Works only if we can add headers	Resequencing pack- ets from different links is complex
BONDING	Guaranteed FIFO	Good load sharing	Works only if we can modify the packet format over a channel. Requires synchronized frame structure over all channels	Needs resequencing logic
Fair Queuing algorithm with header	Guaranteed FIFO	Good load sharing	Works only over channels where we can add headers	Simple
Fair Queuing algorithm, no header	Quasi-FIFO	Good load sharing	Works over all channels	Simple

Table 1: Features of different channel striping solutions

In Section 4, we deal with the FIFO delivery problem. We provide two solutions, both of which are compatible with our fair load sharing solutions described in Section 3. Our main idea is the notion of *logical reception*, in which we separate physical reception from logical reception by a per-channel buffer, and then have the receiver simulate the sender algorithm in order to remove packets from channel buffers.

In the case where an extra header can be added to each packet, and sequence numbers introduced, we show how the receiver processing can be simplified by simulating the sender's fair queuing algorithm. We believe, however, *that the most important application of this technique is the case where an extra header cannot be added.* In this case, we show how to achieve what we call *quasi-FIFO* delivery at the receiver, without any modification of the transmitted packets, by employing the logical reception idea.

We define quasi-FIFO delivery as FIFO delivery except during periods of loss of synchronization between the sender and the receiver. The sender and the receiver are said to be *synchronized* when the receiver algorithm picks up packets in the same sequence as the sender transmits them, thus maintaining FIFO delivery. However, undetected loss of packets between the sender node and the receiver node may cause the receiver algorithms to deliver packets out of order. Thus we show in Section 4.3 how to detect and recover from such loss of synchronization in a timely manner. We also argue in Section 4.2 that quasi-FIFO delivery is adequate for many applications,[5] even for ATM applications, where loss of FIFO at any time is ostensibly forbidden by the standards.

[5] Some readers may wish to turn to this subsection immediately since quasi-FIFO seems distasteful at first glance

Finally, in Section 5, we present experimental verification of the load sharing and FIFO delivery properties of our channel striping schemes, and analyze their performance implications.

3. Solving the Variable Packet Size Problem by Transforming Fair Queuing Algorithms

In this section, we describe how to solve the variable packet size problem by transforming fair queuing algorithms into load sharing algorithms.

We use the term fair queuing to refer to a generic class of algorithms that are used to share a single channel among multiple queues (rather than to one of the most celebrated members of this class, the DKS fair queuing algorithm [DKS89]). Henceforth we will refer to such algorithms as FQ algorithms. In FQ, we try to partition the traffic on a *single output channel* equitably from a set of *input queues* which feed that channel. In load sharing, on the other hand, we seek to partition the traffic arriving on a *single input queue* equitably among a set of *output channels*. The phrasing reveals that the two problems are complementary in some intuitive sense. We wish to make this relationship precise.

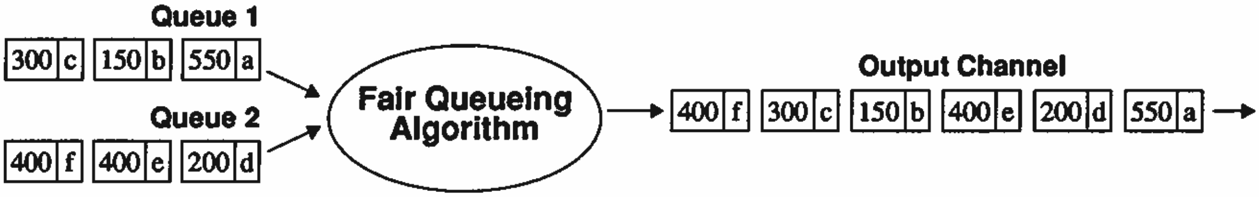

Figure 2: Example of fair queuing

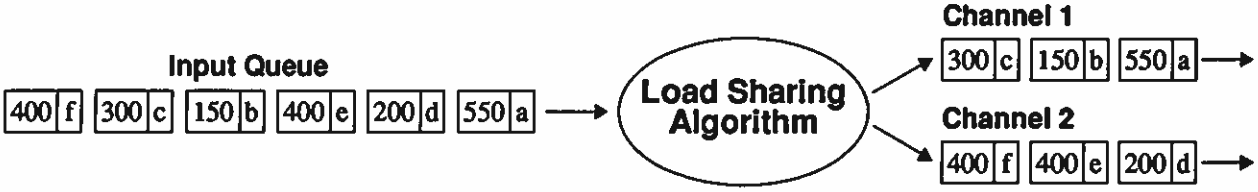

Figure 3: Example of load sharing

Figures 2 and 3 help in making this relationship clearer. In Figure 2, an arbitrary FQ algorithm is used to feed an outgoing channel from two queues. In the figure, each packet is marked with its size (in bytes) and a unique identifier, which ranges from a to f. The FQ algorithm transmits the packets in a particular sequence as shown in the output. Notice that the bandwidth of the channel is partitioned roughly equally among the channels (they have the same fair share of 500 bytes). Now, consider the operation of the FQ algorithm in a time reversed manner, with the direction of the arrows reversed. We would then obtain the situation shown in Figure 3.

In a rough sense, load sharing algorithms are 'time reversals' of fair queuing algorithms [SV94]. We simply run a FQ algorithm as the load sharing algorithm at the sender! The reversal lies in reversing the direction of flow of packets– where the FQ algorithm transmits packets from one of the many queues on to the single channel, the load sharing algorithm transmits packets from the single queue to one of the many channels. However, as we shall soon see, only a subset of fair queuing algorithms can be used as load sharing algorithms with the FIFO delivery property.

3.1. Oblivious and Non Oblivious Fair Queuing Algorithms

Consider a node running a FQ algorithm to feed a channel from multiple queues. Within each queue, packets are transmitted in FIFO order. Assume all queues are backlogged, i.e., have packets to send. The fair queuing problem lies in selecting the queue from which the next transmitted packet should originate. This decision can depend not only on the previously transmitted packets, but also on other parameters, like the size of the packets in the head of each queue, the current queue sizes, and so on. For instance, the DKS algorithm depends on the packets at the head of each queue in order to simulate bit-by-bit round robin.

In the backlogged case, if a FQ algorithm depends only on the previous packets sent to choose the current queue to serve, then we call the algorithm an *oblivious FQ (OFQ) algorithm*. All other FQ algorithms are called non-oblivious algorithms. Thus the DKS algorithm is non-oblivious while ordinary round robin is oblivious.

Why do we restrict ourselves to backlogged FQ behavior? In the non-backlogged case, most FQ algorithms maintain a list of active flows as part of their state. This allows them to skip over empty queues. However, this mechanism also makes almost all FQ algorithms non-oblivious. Thus for our transformation we restrict ourselves to the backlogged behavior of a FQ protocol. Notice that any FQ algorithm must handle the backlogged traffic case. Intuitively, in load sharing there is no phenomenon corresponding to empty queues in fair queuing; this anomaly is avoided by considering only the backlogged case.

In the backlogged case, OFQ algorithms can be formally characterized by repeated applications of two functions in succession. One function $f(s)$ selects a queue, given the current state s of the sender. This is illustrated on the left in Figure 4. After the packet at the head of the selected queue is transmitted, another function g is invoked to update the sender state to be equal to $g(s,p)$ where p is the packet that was just sent. For example, in ordinary round robin the state s is the pointer to the current queue to be serviced; the function $f(s)$ is the identity function: i.e., $f(s) = s$; finally, the function $g(s,p)$ merely increments the pointer to the next queue.

In OFQ algorithms, the current state of the algorithm is essentially encoded in the packets already sent, since OFQ algorithms do not depend on the packets currently queued in order to select the next queue to be serviced. Also, for OFQ algorithms, the function g to update the state depends only on the previous state and the currently transmitted packet.

The conversion of OFQ algorithms to load sharing algorithms, and related theorems can be intuitively described as we show in the next section.

3.2. Use of OFQ Algorithms for Load Sharing at the Sender

The transformation from fair queuing to fair load sharing is illustrated in Figure 4. We start on the left with an OFQ algorithm and end up with a fair load sharing algorithm on the right.

The OFQ algorithm is characterized by an initial state s_0 and the two functions f and g. To obtain the fair load sharing algorithm we start the load sharing algorithm in state s_0. If p is the latest packet received on the high speed channel (see the right of Figure 4), the load sharing algorithm sends packet p to low speed line $f(s)$. Thus while the fair sharing algorithm uses $f(s)$ to *pull* packets from *input* queues, the load sharing algorithm uses $f(s)$ to *push* packets to output channels. In both cases, the sender then updates its state by applying the function g to the current state and the packet that was just transmitted. Notice that there is no requirement for the load sharing algorithm to work only in the backlogged case; if the queue of packets from the input high speed channel is empty, the load sharing algorithm does not modify its state further until the next packet arrives.

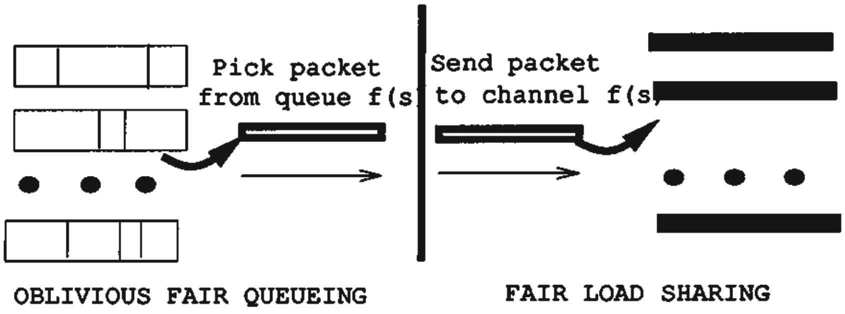

Figure 4: Consider a backlogged execution of a fair queuing algorithm. If the algorithm is oblivious we first apply a function $f(s)$ to select a queue. We transmit the packet p at the head of the selected queue and then update the state using a function $g(s,p)$. We can obtain a fair load sharing algorithm by using the same function f to pick a channel to transmit the next packet on, and update the state using the same function g.

3.3. Evaluating the Transformation: Throughput Fairness

What good is this transformation? It produces useful load sharing algorithms with similar fairness properties (in terms of throughput) to that of the fair queuing algorithms used as inputs to the transformation. To make this precise we need to define the notion of throughput fairness for both deterministic and probabilistic fair queuing schemes. Since we are talking about *throughput fairness* it makes sense to define this quantity in the case when all input queues are backlogged.

Consider a fair queuing scheme with several input queues, where in the start state each queue contains a sequence of packets with arbitrary packet lengths. Define a *backlogged execution* of the fair queuing scheme to be an execution in which no input queue is ever empty during the execution. Notice that there are an infinite number of possible backlogged executions corresponding to the different ways packets (and especially packet lengths) can be assigned to queues in the start state. Notice also that in a backlogged execution we can assume, without loss of generality, that all packets that are serviced arrive in the start state.

An execution will produce as output a finite or infinite sequence of packets taken from each input queue. The bits allocated to a queue i in an execution E is the sum of the lengths of all packets from queue i that are serviced in execution E.

We say that a deterministic fair queuing scheme is *fair* if over all backlogged executions E, the difference in the bits allocated to any two queues differs by at most a constant (for instance, the difference cannot grow with the length of an execution.) We say that a randomized fair queuing scheme is *fair* if over all backlogged executions E, the expected number of bits allocated to any two queues is identical.

We can make analogous definitions for load sharing algorithms. A backlogged execution now begins with an arbitrary sequence of packets on the high speed channel. The bits allocated to a channel i in an execution E is the sum of the lengths of all packets that are sent to channel i in execution E. The fairness definitions for load sharing and fair queuing are then identical except with the word "channel" replacing the word "queue".

Note that any execution of a load sharing algorithm can be modeled as a backlogged execution as long as the load sharing algorithm is oblivious and therefore makes striping decisions based only on the packets seen so far. Thus, at least for the load sharing algorithms considered in this paper, there is no loss of generality in considering only backlogged executions.

3.4. The Transformation Theorem

We now show that a load sharing protocol that is obtained by transforming an OFQ algorithm (as shown above) has the same fairness properties as the original OFQ algorithm.

THEOREM 3.1. *Consider an OFQ algorithm A and a fair load sharing algorithm B that is produced by the transformation described above. Then if A is fair, so is B.*

Proof: (Idea) Notice that the theorem applies to both randomized and deterministic OFQ algorithms. The main idea behind the proof is simple and is best illustrated by Figure 1. We use the initial state s_0 and the functions f and g of A and define B as we described earlier. Now consider any execution E of the resulting load sharing protocol B (for example, the execution shown in Figure 3). From execution E we generate a corresponding execution E' (for example, the execution shown in Figure 2) of the original OFQ algorithm A.

To construct E' from E we consider the outputs of the load sharing algorithm in E to be the inputs for E'. More precisely, we initialize queue i in E' to contain the sequence of packets output for channel i in E. We then show that if the OFQ algorithm A is run on this output, it produces the execution we call E', and the output sequence in E' is identical to the input sequence as E. Thus the input of E corresponds to the output of E', and vice versa. This correspondence can be formally be verified by an inductive proof.

Finally, we know that since A is fair, the output sequence in E' contains approximately the same number of bits from every queue. Thus since there is a 1-1 correspondence between outputs and inputs in E and E', we see that the output sequence in E assigns approximately the same number of bits to every output channel. Since this is true for every execution E of B, B is also fair.

Note that the correspondence does not work in the reverse direction. It is hard to construct a load sharing execution from an OFQ execution. However, what we have shown is sufficient for the proof. □

3.5. Surplus Round-Robin (SRR)

We now turn to two specific examples of the transformation theorem. Our first example is a transformation of an oblivious fair queuing algorithm that we call Surplus Round Robin (SRR). SRR is based on a modified version of DRR [SV94]. SRR is also identical to a FQ algorithm proposed by Van Jacobson and Sally Floyd [Flo93].

In the SRR algorithm, each queue is assigned a quantum of service (measured in units of data), and is associated with a counter called the Deficit Counter (DC), which is initialized to 0. Queues are serviced in a round robin manner. When a queue is picked for service, its DC is incremented by the quantum for that queue. As long as the DC is positive, packets are sent from that queue, and the DC is decremented by the size of the transmitted packet. Once the DC becomes non-positive, the next queue in round robin order is selected for service. Thus if a queue overdraws its account by some amount, it is penalized by this amount in the next round. It can be proven that SRR is a deterministic and oblivious fair queuing scheme, and is also fair (in the sense defined above).

Transforming SRR into a load sharing algorithm. The corresponding load sharing algorithm works as follows. Each channel is associated with a Deficit Counter (DC), and a quantum of service, (measured in units of data), proportional to the bandwidth of the channel. Initially, the DC of each channel is initialized to 0, and the first channel is selected for service, i.e. for transmitting packets. Each time a channel is selected, its DC is incremented by the quantum for that channel. Packets are sent over the selected channel, and its DC is decremented by the packet length, in bytes, till the DC becomes non-positive. The next channel is then selected in a round-robin manner, and its quantum is added to its DC. Packets are sent over this channel till its DC becomes non-positive, and then the next channel is selected, and so on.

We now describe a simple example of the operation of the SRR load sharing algorithm. Refer to Figure 3. In the figure, we see packets labeled *a*, *d*, *e*, *b*, *c*, and *f* at the input queue to the load sharing algorithm. We now explain how the SRR algorithm stripes packets across the two output channels.

Both Channel 1 and Channel 2 are associated with a quantum of 500 bytes. Initially, the Deficit Counters of the two channels, $DC1$ and $DC2$ respectively, are initialized to 0. Channel 1 is first selected for service, and its DC is incremented by the value of the quantum for the channel, which is 500 bytes. Since $DC1$ is positive, the first packet in the queue, namely packet *a*, is sent on Channel 1. $DC1$ is decremented by the size of *a*, which is 550 bytes. $DC1$ therefore becomes -50. Since $DC1$ is now less than 0, we proceed to the next channel, which is Channel 2. As it has been selected for service, $DC2$ is incremented by its quantum, which is 500 bytes. The next packet in the input queue, packet *d*, is sent on Channel 2, thereby decrementing $DC2$ to 300 bytes, since *d* is of size 200 bytes. The positive value of $DC2$ permits the next packet, packet *e*, to be also sent on this channel.

After sending packet *e*, $DC2$ is reduced to -100. $DC2$ going negative causes the SRR algorithm to select the next channel, which is Channel 1, for service and to increment $DC1$ by 500 bytes, thereby setting $DC1$ to 450 (It was previously -50). This value of $DC1$ allows us to send packets *b* and *c* out on Channel 1, at which point $DC1$ is reduced to 0. This causes Channel 2 to be selected for service, thereby increasing $DC2$ by 500 bytes to 400. This suffices to put packet *f* out on Channel 2. Thus at the end of two rounds, both $DC1$ and $DC2$ are 0, and packet *a*, *b* and *c* are sent on Channel 1,and packet *d*, *e*, and *f* are sent on Channel 2.

The SRR load sharing protocol has a number of nice properties that should make it appropriate for practical usage. It divides the bandwidth fairly among output channels even in the presence of variable length packets. It is also extremely simple to implement, requiring only a few more instructions than the normal amount of processing needed to send a packet to an output channel. Note that is also possible to generalize SRR to handle channels with different rated bandwidths by assigning larger quantum values to the higher bandwidth lines. This corresponds to weighted fair queuing.

3.6. Randomized Fair Queuing (RFQ)

Our second specific example of a transformation is based on a simple fair queuing algorithm that we call Randomized Fair Queuing[6] OFQ algorithm.

This algorithm needs even less state than SRR; however, it provides probabilistic, rather than deterministic fair queuing. Briefly, the queue to be served is chosen by selecting a random queue index (this can be implemented by a pseudorandom sequence). In general, to implement weighted fair queuing, we can choose Queue i with a probability of occurrence that is proportional to the weight associated with that of Queue i.

It is easy to show that when all queues are backlogged, this algorithm is fair in the probabilistic sense defined above - i.e, the expected amount of data sent from any two queues is the same for any execution. This assumes that the average packet size of each queue is the same. In case the average packet size is different , we have to assign weights to each queue inversely proportional to the average packet size for that queue. The RFQ algorithm is also oblivious.

When we apply the transformation theorem to this fair queuing protocol, the resulting load sharing protocol is particularly simple. Each Channel i is associated with a probability $prob(i)$ corresponding to the bandwidth B_i of the channel, such that $prob(i) = B_i / \sum_{j=1}^{N} B_j$. A pseudorandom sequence, ranging from 1 to N (such that the probability of occurrence of i is the same as the probability associated with the Channel i) is used to select the channel over which the next packet is sent.

From a practical viewpoint it seems preferable to rely on SRR as it provides guaranteed load sharing. However, the randomized channel sharing protocol is slightly simpler to implement and may be preferable in some cases.

4. Solving the FIFO Delivery using Logical Reception

This section describes techniques for ensuring FIFO delivery at the receiver. Recall that this is the second major problem to be solved by a channel striping scheme. The techniques described in this section work in conjunction with the load sharing algorithms presented in the previous section to provide a complete solution. The main idea that we propose is what we call *logical reception.*

Logical reception is a combination of two separate ideas: *buffering at the receiver* to allow physical reception to distinguished from logical reception, and *receiver simulation* of the sender striping algorithm. Logical reception can be explained very simply using Figure 1. Notice that there are per-channel buffers shown between the channel and the reassembly algorithm. Notice also that if we look at the picture at the destination

[6] Randomized Fair Queuing is not related to Stochastic Fair Queuing [McK91], which refers to a class of algorithms that are probabilistic variants of the DKS algorithm

node, it is clear that the receiver is performing a fair queuing function. But we have already seen a nice connection between channel striping and fair queuing schemes. Thus the main idea is that *the receiver can restore the FIFO stream arriving to the source if it uses a fair queuing algorithm that is derived from the channel striping algorithm used at the source.*

An example will help clarify the main idea. Suppose in Figure 1 that the sender sends packets in round robin order sending packet 1 on Channel 1, packet 2 on Channel 2, . . . and packet N on Channel N. Packet $N + 1$ is sent on Channel 1 and so on. The receiver algorithm uses a similar round robin pointer that is initialized to Channel 1. This is the channel that the receiver next expects to get a packet on. The main idea is that the receiver will not move on Channel $i + 1$ until it is able to remove a packet from the head of the buffer for Channel i. Thus suppose Channel 1 is much faster than the others and packets 1 and $N + 1$ arrive before the others at the receiver. The receiver will remove the first packet from the Channel 1 buffer. However, the receiver will block waiting for a packet from Channel 2 and will not remove packet $N + 1$ until packet 2 arrives.

In general, if the sender striping algorithm is a transformed version of an oblivious fair queuing (OFQ) algorithm, then the receiver can simulate the sender algorithm. The receiver can run the OFQ algorithm, to know the channel over which the next packet is to arrive from the sender. The receiver then blocks on that channel, waiting for the next packet to arrive, while buffering packets that arrive on other channels. The simulation of the sender algorithm, coupled with the buffering of packets on each channel and receiver blocking on the expected channel, ensures *logical FIFO reception*, irrespective of the nature of the skew present between the various channels. This can be stated as the following theorem:

THEOREM 4.1. *Let B be the load striping algorithm derived by transforming an OFQ algorithm A. If B is used as a channel striping algorithm at the sender and A is used as the fair queuing algorithm at the receiver, and no packets are lost, then the sequence of packets output by the receiver is the same as the sequence of packets input to the sender.*

Of course, the synchronization between sender and receiver can be lost due to the loss of a single packet. In the round-robin example shown above if packet 1 is lost, the receiver will deliver the packet sequence $N + 1, 2, 3, \ldots, N, 2N + 1, N + 2, N + 3, \ldots$ and permanently reorder packets. Thus there is a need for the sender to periodically resynchronize with the receiver. Such synchronization can be done quite easily as shown in Section 4.3. Intuitively, if packets are lost infrequently and periodic synchronization is done in a timely manner, logical reception will work well. We will discuss the performance implications of this scheme in more detail later.

Given the fact that the receiver runs the same fair queuing algorithm to reassemble incoming packets, as the sender uses for channel striping, it is easy to appreciate why it is necessary for the fair queuing algorithm to be oblivious. For the receiver to simulate the sender, it is necessary for it to know the channel over which the next packet is going to arrive. This decision has to be made based on the current state, which can encode only the previous arrivals. By definition, this is the property of OFQ algorithms.

Note that buffering of packets often does not introduce any extra overhead because once the packets are read in, they do not have to be copied for further processing– only pointers to the packets need be passed, unless the packet has to be copied from one address space to another (e.g. from the adaptor card to the main memory), in which case a copy is needed in any case.

We now consider two cases. The first is the case when sequence numbers can be added to packets. In this case, logical reception may be a way of simplifying the packet reassembly algorithm while the sequence numbers are only used as a backup to detect FIFO violation. The most important application of the logical reception idea, however, is the case when sequence numbers cannot be added. In this case, we cannot guarantee FIFO delivery during periods of resynchronization but have to settle for quasi-FIFO delivery. We argue why we believe that quasi-FIFO delivery is adequate even for ATM applications in Section 4.2

Our plan for the rest of this section is as follows. First, we describe the logical reception idea, ignoring the need for synchronization. We start with the case when headers can be added, and then consider the case when headers cannot be added. Finally, we describe our solutions to the synchronization problem in Section 4.3.

4.1. Logical Reception to Speed Up Packet Processing

As discussed earlier, if headers can be added to the transmitted packets, we can include a sequence number with every packet, which can be used for resequencing at the receiver. A naive implementation at the receiver might sort through the incoming packets to resequence them, incurring a sorting cost for each packet received. Sorting would be particularly expensive if there were random delays and skews between different channels leading to sorting amongst a large number of packets. Hardware implementations have been proposed which can be used for reassembly of packets arriving on multiple links [McA93]. This scheme requires specialized hardware (to sort out of order packets) and modified packet formats.

If we use logical reception we can (potentially) have a very simple reassembly scheme. Receiver simulation of the sender will work with any amount of skew, with graceful degradation in the average delay experienced by each packet. The sequence number inserted by the sender is now needed only for confirmation, since in the normal state, when there is no packet loss, the receiver delivers packets in FIFO order. The sequence numbers added by the sender to each packet serve to detect, and to provide sequencing of packets even if the sender and receiver lose synchronization, thereby providing *guaranteed FIFO reception.*

4.2. Quasi-FIFO Reception at the Receiver

We observe that even if we dispense with the sequence numbers, the receiver can still provide FIFO delivery of packets, provided there is no loss of packets. We refer to this mode of packet reception, in which the sender does not send any explicit sequence numbers with each packet, as *quasi-FIFO reception*, as opposed to guaranteed FIFO reception. We argue that in cases where sequence numbers cannot be added to packets, or their addition is expensive to implement, quasi-FIFO provides more than adequate performance.

Motivation for quasi-FIFO reception. Existing datagram networks, e.g. the Internet, do not guarantee FIFO delivery of packets. In these networks, FIFO delivery is a performance, not a correctness issue. These networks do not have any problem with quasi-FIFO reception, except for performance degradation during loss of FIFO reception. However, ATM networks mandate FIFO delivery of packets (cells). There are two perceived drawbacks involved in the use of quasi-FIFO reception in ATM networks:

- **Quasi-FIFO destroys correctness for ATM applications:** In ATM networks, packets are encapsulated in ATM Adaptation Layer (AAL) frames. We deal with applications that run above the commonly used AALs (AAL3-4 and AAL5). In these AALs, individual cells do not carry any error checking or CRC for the data portion of the cell. In that case, reordering of cells is detected by the CRC, and is equivalent to a packet loss. ATM does not provide guaranteed cell delivery, so applications have to be able to deal with packet loss in any case. Out of order reception, therefore is similar to the occurrence of a burst error with FIFO reception.

- **Quasi-FIFO has bad performance implications:** A preliminary look at quasi-FIFO seems to imply that a single bit error can be magnified by a factor of N, where N is the number of cells received out of order till synchronization is restored, as described in Section 4.3. However, this is not usually the case, since loss of synchronization is caused by packet *loss*, not packet *corruption*, which is what results from bit errors. So a single bit error, or even a burst error, is unlikely to cause the receiver to lose synchronization over a physical point to point channel, provided the physical channel hardware has the capability of reporting cell errors.

  Over a logical channel, which is composed of multiple physical channels linked through switches, undetected cell loss can take place. However, as we describe later in this section, we can bound the value of N (out of order cells) by sending periodic marker packets. For example, an IP packet of maximum size sent over ATM [Lau94] would occupy more than 180 ATM cells. If we send periodic marker cells after every 180 data cells, then during loss of synchronization, we would lose approximately the same number of cells as exist in an single maximum size IP packet. We note that even in the absence of striping, the loss of a single cell on an ATM connection would cause the entire packet to be lost. The packet loss of striping over logical channels therefore is not significantly more than that of an unstriped connection, and is more than made up by the increased bandwidth obtained.

For quasi-FIFO reception to be of practical significance, we need to restore synchronization between the sender and the receiver in case of loss of synchronization, else the receiver will continue to deliver packets out of order indefinitely. The restoration of synchronization is needed for guaranteed FIFO reception to work efficiently also. The next section discusses protocols for the detection of and recovery from this loss of synchronization.

4.3. Synchronization in the Face of Errors

The techniques described below utilize special marker packets, which the receiver can distinguish from the normal data packets. We assume that when either the sender or the receiver goes down and comes up, it reinitializes the channel, thus restoring synchronization. So the error cases that we have to deal with are channel errors which cause packet loss, and hardware/software errors at either the sender or receiver. Note that sending marker packets does not require modifications to be made to the data packets, which was listed as one one of the desirable properties of a striping scheme in section 1.

Detecting packet errors by sending packet counts. A simple way for the receiver to know whether it is in synchronization with the sender is for the sender to periodically send over each channel a marker packet containing the count of the number of packets it has sent over that particular channel. The receiver compares the marker value to its locally maintained value. In case the two values match on all channels, then the receiver can assume that it is in synchronization till that point. This of course depends on the markers following the data packets in FIFO order, which is ensured by FIFO channels.

This technique, though simple to implement, is not fully self stabilizing, i.e., it cannot recover from arbitrary errors including link errors and hardware errors that corrupt registers. To see why, consider a simple scenario in which round robin striping is done with equal sized packets across two channels. Assume the sender transmits the first packet over the first channel. If due to some local hardware error, the receiver starts reception at the second channel, it would pick the incoming packets out of sequence, and this would not be detected by this technique, since the packet counts carried by the marker packets on both channels would be the same as the locally maintained values at the receiver.

Use of Snapshots for Synchronization. A technique for synchronization detection which is self stabilizing is for the the sender to embed its current state in the marker packets which it periodically sends. This provides a snapshot of the system, which can be proven to be self stabilizing [CL85]. For example, if SRR is used for channel striping, the state would include the current value of the Deficit Counters, and the channel currently selected for service.

Synchronization recovery. The previous techniques only ensure error detection, but not error recovery. To ensure error recovery, it is necessary for the sender and the receiver to resynchronize by reinitializing. This can be done by the receiver sending a *reinitialize-request* packet to the sender upon detecting loss of synchronization, which in turn causes the sender to send a *reinitialize-confirm* packet to the receiver on all channels, following which the sender and the receiver reinitialize, and are back in synchronization.

The scheme described above adds a round trip delay to synchronization recovery, which is clearly undesirable over channels with high latencies. An alternate scheme, currently being worked out for the SRR based striping scheme, involves using the information contained in the marker packets to restore synchronization at the receiver, without the use of explicit reinitialization message exchange between the receiver and the sender. This scheme introduces *ghost* packets in the receiver input queues to compensate for missing data packets. The receiver reassembly algorithm treats the *ghost* packets just as regular data packets, with the difference that the *ghost* packets are not delivered to the output queue – they serve purely to restore synchronization at the receiver. This scheme has been verified experimentally, and work is in progress to formally prove that the scheme restores synchronization at the receiver.

5. Implementation and Performance Issues

The proposed channel striping schemes were implemented above the transport layer to verify their load sharing and FIFO delivery properties. The implementation utilized three components:

- **Channels:** The FIFO channels were provided by TCP connections between the sender and receiver programs running on two different workstations across an Ethernet network. To simulate a channel of a given capacity, the connections were throttled, by permitting the sender program to send only a given number of bytes per unit time on each connection. All channels had equal bandwidths.
- **Striping algorithms:** For the channel striping algorithms, the SRR and RFQ algorithms presented in section 3 were implemented at the sender and receiver.
- **Packet generation at the sender:** The packet stream at the sender was modeled as a continuous stream of packets of uniformly distributed sizes.

The sender program incorporated the packet generator, the load sharing version of the OFQ algorithm, and the sending end of the TCP connections. The receiver program incorporated the other end of the TCP connections, and the OFQ algorithm for packet resequencing. The only complexity in the programs was in the receiver code to reassemble complete packets from the byte stream arriving at the receiver, as the socket based TCP connections did not preserve packet boundaries. The actual OFQ algorithms took only a few lines of code to implement. Two sets of sender and receiver programs were created, one for the SRR, and other for the RFQ algorithm. The implementation verified that packets were transmitted on all channels at the same data rate by the sender algorithm. The implementation also verified that the receiver algorithm restored the FIFO delivery order.

Figure 5 shows the performance of the SRR and RFQ algorithms, when applied to striping in the configuration described above. The figure shows the variation in throughput for different channel capacities. As can be seen, the striping algorithm provides scalable throughput as the number of channels increases. The decrease in throughput with increasing channel rates is due to the saturation of the processing capacity at either workstation.

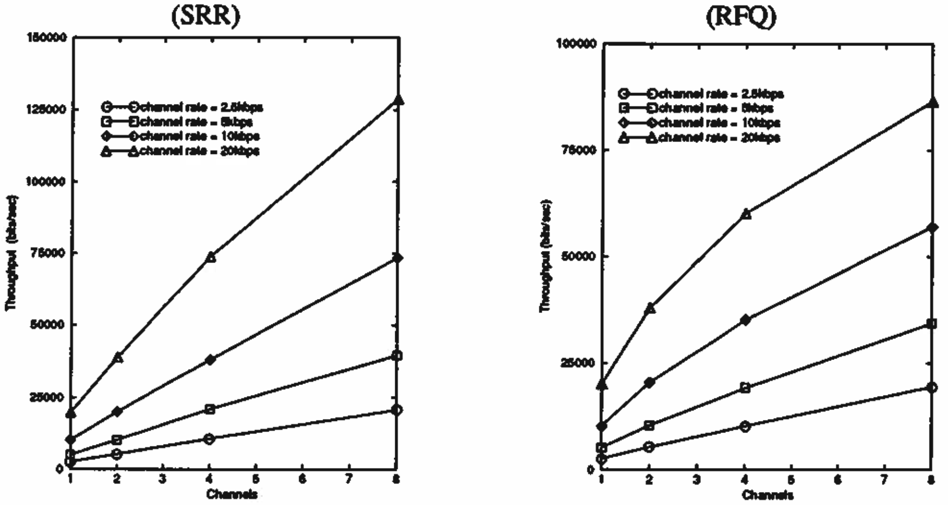

Figure 5: Performance of SRR and RFQ algorithms

5.1. Performance Model

We present performance analysis for channel striping schemes based on fair queuing algorithms. We assume full channel loading in the cases discussed below.

Cost of Load Sharing using OFQ Algorithms. The amount of data sent through each channel by the channel striping algorithm is proportional to the rated bandwidth(BW) of the channel. We assume that the sender cannot transmit at a rate greater than the actual BW of the channel. In such a case, if the actual BW were to vary below the rated BW, the actual throughput at the sender would not only be less than that of the aggregate of the rated BWs, but also less than the aggregate of the actual BWs. This somewhat surprising result is due to the nature of the load sharing operation of the channel striping algorithm. The slowest channel effectively constrains the rate of transmission over all other channels, since the sender blocks on this channel to transmit data.

If the rated BW of Channel i is $B_{rated,i}$, and the actual BW of each channel is $B_{act,i}$, then the *amount* of data sent over Channel i is proportional to $B_{rated,i}$, while the maximum *rate* at which it is can be sent over Channel i is $B_{act,i}$. Let the ratio $B_{act,i}/B_{rated,i}$, measured over all channels, be minimum for Channel j. Then it can be shown that the maximum aggregate rate at which data is sent by the sender is $(B_{act,j}/B_{rated,j}) \sum_{i=1}^{N} B_{rated,i}$, which is less than $\sum_{i=1}^{N} B_{act,i}$.

The above analysis indicates that it is important to assign a value to the rated BW of the channel close to the actual BW.

Buffering Requirements. If one of the channels has higher delay than the other channels, then we have to buffer the data from the other channels while we wait to receive packets from this channel, in order for the proper working of the fair queuing algorithm. This increases the delay of all delivered packets.

If there are N channels, all operating at the same speed, and one channel is slower than the others by T secs, which corresponds to β bytes of data, then we have an extra $(N-1)\beta$ bytes of buffering introduced at the receiver.

Assume, in the general case, the channels are numbered from 0 to $N-1$, in the order of decreasing delay. Then, relative to the channel with the maximum delay (channel 0), let the skew in time for channels $1, 2, \ldots, N-1$ be $T_1, T_2, \ldots, T_{N-1}$, respectively. Let B_i is the bandwidth of channel i. Then the total amount of buffering required is $\sum_{i=1}^{N-1} B_i T_i$, with $B_i T_i$ amount of buffering required at channel i.

All incoming packets are slowed T_{N-1} extra by this buffering. In case it is possible to distinguish packets belonging to different classes, it is possible to run two instances of the channel striping algorithm in parallel—one utilizing only the fast channels for time sensitive traffic, and another utilizing all channels for delay insensitive traffic. We do note however, that application which need channel striping tend to be throughput intensive, rather than delay intensive.

Cost of a Link Error. Whenever a packet is lost, there is the possibility of the receiver going out of synchronization with the sender. In such a case, while succeeding packets are not lost, FIFO order is lost till the synchronization detection and recovery algorithm runs. The synchronization detection algorithm depends on the sender sending periodic marker packets on all channels. By adjusting the period of this algorithm, we can bound the loss of FIFOness. For example, by transmitting a marker packets after every N data packets, we can ensure that loss of FIFO is restricted to a very small number of packets (around $N/2$ on the average, depending on N). Also, in some cases, it might be possible for the receiver to detect packet loss even before the arrival of the marker packets. For example, if the receiver is an ATM end node, since it reassembles the incoming cells, it can conclude from sustained CRC errors that it is out of synchronization with the sender. It must be noted, however, that a tradeoff exists between the frequency of marker packets and the overhead involved in sending and processing these packets.

6. Conclusion

This paper describes a family of efficient channel striping algorithms that solve both the variable packet size and the FIFO delivery problems for a fairly general class of channels. The channels can lose packets and have dynamically varying skews. Thus our schemes can be applied not only at the physical layer, but also at higher layers.

We solve the variable packet size problem by transforming a class of fair queuing algorithms called Oblivious Fair Queuing (OFQ) algorithms into load sharing algorithms. We solve the FIFO problem by the idea of logical reception, which combines the two ideas of receiver buffering and receiver simulation of the sender algorithm. It is pleasing that in order for receiver simulation to work it is only necessary that the sender algorithm be oblivious, which is guaranteed by our fair load sharing schemes. The requirement for obliviousness arises for similar reasons in the two cases: in the fair queuing transformation it arises because channel striping decisions cannot depend on packets that have not yet arrived; the same is true for receiver simulation. Logical reception must also be augmented with periodic resynchronization to handle packet losses.

We believe the channel striping algorithm based on SRR, logical reception, and resynchronization (using packet counts) is quite suitable for practical implementation, even in hardware. SRR requires only a few extra instructions (to increment the deficit counter and do a comparison); the packet count synchronization protocol is also simple since it only involves keeping a counter and sending a marker containing the counter. Of course, if synchronization loss is detected, we have to reinitialize the system; however, that complexity is needed anyway to handle the case when channels are added and removed. For simple channel striping algorithms like round robin (which is appropriate for striping fixed size packets like cells), we are working on an even simpler reinitialization scheme which involves the receiver skipping a channel n times whenever n packet losses are detected.

We have also described (and hopefully defended!) the notion of quasi-FIFO reception. Without the addition of sequencing information, the receiver can only provide quasi-FIFO delivery. We believe that quasi-

FIFO performance is adequate for most datagram applications and even for ATM, especially in cases where adding a sequence number to each packet is either not possible, or is expensive to implement.

We believe that striping on physical links and striping across virtual circuits are the most important applications of our techniques. For an ATM virtual circuit, it appears feasible to implement markers using OAM cells that are sent on the same Virtual Circuit that implements the channel. When striping end-to-end across ATM circuits, it seems advisable to stripe at the packet layer. Striping cells across channels would mean that AAL boundaries are unavailable within the ATM networks; however, these boundaries are needed in order to implement early discard policies [RF94].

There are a number of detailed issues that we have not described for lack of space. In the paper so far, we have assumed that the receiver buffers (required for logical reception) do not overflow. This is a good assumption if we can bound the worst-case skew (note this does not rule out dynamically varying skews), the channel bandwidths, and the receiver processing rates. In more general cases, it might be necessary to implement a flow control scheme as part of the channel striping scheme. A simple scheme would involve the sending of Xon and Xoff messages. We have also not described dynamic algorithms for detecting channel failures or reductions in effective channel bandwidth: these can easily be added and even used to dynamically vary the load sharing weights assigned to channels. The actual synchronization required to change channels or synchronization weights is similar to that needed for reinitialization. Regardless of these details, we hope the simplicity and elegance of the schemes described in this paper will lead to a wider deployment of channel striping schemes.

Acknowledgement

We acknowledge the help of Adam Costello of Washington University in discussions leading to the formulation of the Randomized Fair Queuing algorithm.

References

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of a Distributed System. *ACM Transactions on Computer Systems*, pages 63–75, February 1985.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM SIGCOMM*, pages 3–12, 1989.

[DP94] Peter Druschel and Larry L. Peterson. Experiences with a High-Speed Network Adaptor: A software Perspective. In *Proceedings of the ACM SIGCOMM*, 1994.

[Dun94] Jay Duncanson. Inverse Multiplexing. *IEEE Communications Magazine*, 32(4), April 1994.

[Flo93] Sally Floyd. Notes on Guaranteed Service in Resource Management. Unpublished Note, 1993.

[Fre94] Paul H. Fredette. The Past, Present and Future of Inverse Multiplexing. *IEEE Communications Magazine*, 32(4), April 1994.

[Gro92] Bandwidth ON Demand INteroperability Group. Interoperability Requirements for Nx56/64 kbit/s Calls, September 1992.

[HS94] M.V. Hegde H. Saidi, P.S. Min. Non-blocking Multi-channel Switching in ATM Networks. In *Proceedings of SUPERCOM/ICC94*, May 1994.

[Jac90] V. Jacobson. 4BSD Header Prediction. *ACM Communication Review*, April 1990.

[Lau94] M. Laubach. Classical IP and ARP over ATM. RFC1577, January 1994.

[McA93] A. J. McAuley. Parallel Assembly for Broadband Networks, 1993.

[McK91] Paul E. McKenney. Stochastic Fair Queueing. *Internetworking: Reserch and Experience*, pages 113–131, 1991.

[RF94] Allyn Romanov and Sally Floyd. Dynamics of TCP Traffic over ATM Networks. In *Proceedings of the ACM SIGCOMM*, pages 79–88, 1994.

[SV94] M. Shreedhar and G. Varghese. Efficient Fair Queueing by Deficit Round Robin. Technical Report WU94-17, Washington University, 1994.