A Reliable and Scalable Striping Protocol*

Hari Adiseshu, Guru Parulkar and George Varghese
hari@dworkin.wustl.edu, guru@cs.wustl.edu, and varghese@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA
TEL: (314) 935-6160, FAX: (314) 935-7302

Abstract

Link striping algorithms are often used to overcome transmission bottlenecks in computer networks. 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 solves both problems. Our scheme applies to any layer that can provide multiple FIFO channels.

We deal with variable sized packets by showing how fair queuing algorithms can be transformed into load sharing algorithms. Our transformation results in practical load sharing protocols, and shows a theoretical connection between two seemingly different problems. The same transformation can be applied to obtain load sharing protocols for links with different capacities. We deal with the FIFO requirement for two separate cases. If 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, we show how to provide quasi-FIFO delivery. Quasi-FIFO is FIFO except during occasional periods of loss of synchronization. We argue that quasi-FIFO is adequate for most applications. We also describe a simple technique for speedy restoration of synchronization in the event of loss.

We develop an architectural framework for transparently embedding our protocol at the network level by striping IP packets across multiple physical interfaces. The resulting *strIPe* protocol has been implemented within the NetBSD kernel. Our measurements and simulations show that the protocol offers scalable throughput even when striping is done over dissimilar links, and that the protocol synchronizes quickly after packet loss. Measurements show performance improvements over conventional round robin striping schemes and striping schemes that do not resequence packets.

*Hari Adiseshu and Guru Parulkar were supported in part by ARPA, National Science Foundation, and an industrial consortium of Ascom Timeplex, Bellcore, BNR, Goldstar, NEC, NTT, SynOptics, and Tektronix. George Varghese was supported in part by NSF Research Grant NCR-9405444 and an ONR Young Investigator Award.
Copyright © 1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., fax +1 (212) 869-0481, or (permissions@acm.org).

1 Introduction

Parallel architectures are attractive when scalar architectures with the required performance are unavailable or have poor cost-performance. Examples include multiprocessors and RAID systems that use disk striping. Parallel solutions, however, have additional costs for *synchronization* (e.g., the need to keep multiprocessor caches coherent) and *fault-tolerance* (e.g., the need for parity disks in disk arrays).

Similar considerations apply to computer networks [TS95] because of transmission and processing bottlenecks. High end workstations and servers can easily saturate existing Local Area Networks (LANs). Such devices may obtain increased throughput by "striping" data across multiple adaptors and multiple LANs. Solutions that use striping may even be cheaper than the alternatives.

As an example of cost-performance tradeoffs, a 155 Mbps multimode fiber together with transmitter/receiver optics costs about $75 today, while a 622 Mbps single mode fiber costs about $700. For a wirelength of a mile or so, striping data across four 155 Mbps fibers may be cheaper than using a 622 Mbps link. Similarly, in the wide area the price differential between T1 and T3 lines makes striping across T1 links attractive. On the other hand, many of the Gigabit testbeds [TG93, JD93] have resorted to striping because of the unavailability of high speed equipment: for instance, the IBM SIA adaptor [TG93] emulates a SONET STS-12 line using four STS-3c lines.

Thus channel striping, also known as load sharing or inverse multiplexing, is often used in computer networks. However, as in other parallel solutions, there are *synchronization* and *fault-tolerance* costs that are inherent to channel striping. If a FIFO (First-In-First-Out) stream of packets is striped across multiple channels, packets may be received out of order at the receiver because of different delays (called *skews*) on channels. In many applications the receiver must reconstruct the sender sequence from the parallel packet streams. This adds a synchronization cost. In addition, channel striping must also be resilient to common faults such as bit errors and link crashes.

As we will see in Section 2.1, earlier solutions to the synchronization and fault-tolerance problems are expensive, inefficient, or dependent on assumptions that make them infeasible in certain application domains. Our paper, on the other hand, describes a new family of channel striping algorithms that is both general and efficient. Our striping schemes are based on a combination of two novel ideas: *fair load sharing* and *logical* FIFO *reception*.

The theoretical contributions of this paper include: a new connection between fair queuing and load sharing, the idea of logical reception, and a novel distributed algorithm to restore synchronization in the face of loss. The practical contributions of this paper include: an architectural model, a working software implementation of the model and the striping algorithm, and measurements and evaluation of the striping algorithm.

Overview of Solution Components

In Section 2, we present a model of the load sharing problem, and review previous work. To provide load sharing in the presence of varying length packets, we use *fair load sharing* algorithms. We show such fair load sharing algorithms can be automatically derived by transforming a class of *fair queuing* algorithms. In Section 3, we develop a criterion for this transformation, and provide an instance of fair load sharing algorithms.

In Section 4, we deal with the FIFO delivery problem. Our solution is 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; we then have the receiver simulate the sender algorithm in order to remove packets in FIFO order from channel buffers. We show how to achieve *quasi*-FIFO delivery at the receiver without any modification of the transmitted packets, by using logical reception.

We define quasi-FIFO delivery as FIFO delivery except during periods of loss of synchronization between the sender and the receiver. Undetected loss of packets between the sender and receiver may cause loss of synchronization. We show in Section 5 how to quickly detect and recover from such loss of synchronization.

In Section 6 we present the details of our prototype implementation. We first present a framework for striping IP packets over multiple IP interfaces in Section 6.1. We then present experimental verification of the load sharing and FIFO delivery properties of our channel striping scheme in Section 6.2. We show how an implementation of our striping algorithm over two dissimilar links can provide the aggregate throughput of the individual links. We also study the individual impact of our two ideas: SRR versus round robin, and logical reception versus no resequencing.

2 Model and Related Work

To allow our algorithms to be widely applicable, we use a broad definition of a channel. For the rest of this paper, we define a *channel* to be a logical FIFO path at either the physical, data link, network, or transport layers. 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.

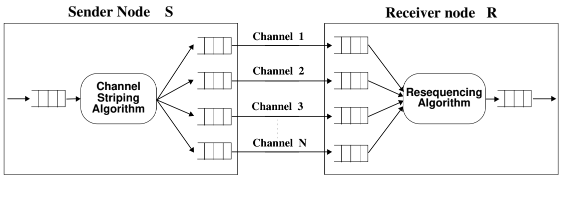

Figure 1: Channel striping configuration

As seen in Figure 1, there are N channels between the sender S and the receiver R. For simplicity, we consider traffic in only one direction; the same analysis and algorithms apply for the reverse direction. Node S implements the striping algorithm to stripe outgoing traffic across the N channels, and node R implements the resequencing algorithm to combine the traffic into a single stream. We will sometimes assume, for throughput analysis, that sender S is backlogged, i.e., it always has packets to transmit. However, our algorithms work for any traffic pattern offered to the sender.

All channels are assumed to be FIFO. Channels can be subject to packet loss and corruption. Channels that occasionally deviate from FIFO delivery can also be modeled as having burst errors. Finally, we allow the end-to-end latency or skew across each channel to be potentially different and to vary on a packet to packet basis. This is important to model realistic network channels. This also rules out simple solutions to the resequencing problem based on skew compensation, if the skew cannot be bounded or characterized.

The simplest example of a channel at the data link layer is a point-to-point link that connects two devices, where the two devices could be workstations, switches, routers, or bridges. A less obvious example of a data link channel is a LAN (e.g., Ethernet), that guarantees FIFO delivery between a given sender and receiver. Network layer channel examples include ATM or X.25 virtual circuits. Even in datagram networks, it may be possible to construct "network" channels (e.g., by using strict IP source routing to set up multiple paths between two IP endpoints), but these examples seem contrived. Finally, since most transport protocols like TCP provide a stream service, it is possible to think of a channel as a transport connection. A fast CPU may achieve higher throughput by striping data across multiple "intelligent" adaptors, each of which implements a TCP connection. However, the most useful examples appear to be data link and virtual circuit channels.

Given a set of FIFO channels, the desirable properties of a channel striping scheme include fair load sharing with variable sized packets and variable capacity channels, FIFO delivery of packets at the receiver, and applicability to a wide variety of channels without any modification to existing channel packet formats or equipment. In addition the scheme should be robust enough to recover from bit and burst errors, and be scalable enough to impose little overhead.

2.1 Existing Channel Striping Algorithms

[TS95] describes a model of load striping, and summarizes various approaches to the problem. Table 1 compares the features of some solutions to striping.

The simplest channel striping scheme is round robin striping — the sender sends packets in round robin order on the channels. Round robin provides for neither load sharing with variable sized packets, nor FIFO delivery without packet modification. Fair load sharing does not hold if the sender alternates between big and smaller packets and stripes over two channels. In this case, all the big packets go over one channel. Also, since the channels may have varying skews, the physical arrival of packets at the receiver may differ from their logical ordering. Without sequencing information, packets may be persistently misordered.

Round robin schemes can be made to guarantee FIFO delivery by adding a packet sequence number which can be used to resequence packets at the receiver. However, this violates the goal of working over existing channels which do not allow header modification. 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 cells between two switches), and yet use existing equipment. Even channels that allow variable sized packets (e.g., Ethernets) have a restriction on the maximum packet size. We cannot add an extra header if the packets that the sender wishes to send are already maximum sized packets.

Both the variable packet size problem and the FIFO problem can be solved if the channel striping algorithm can modify the equipment (typically hardware) or reformat the packets at the endpoints of a channel. For example, in the First Come First Serve scheme, the packets are split into fixed size striping units of data, which are then striped round robin across the channels. The striping unit can be a bit or a byte or a bigger aggregation. Bit or byte interleaving

Scheme	FIFO **delivery**	**Load sharing with Variable Length Packets**	**Target Environment**
Round-Robin, no header	May be non-FIFO	Poor	At all levels
Round-Robin with header	Guaranteed FIFO	Poor	Only if we can add headers
BONDING	Guaranteed FIFO	Good	Only over synchronous serial channels
Fair Queuing algorithm with header	Guaranteed FIFO	Good	Only if we can add headers
Fair Queuing algorithm, no header	Quasi-FIFO	Good	At all levels

Table 1: Features of some channel striping solutions. The first three rows describe existing schemes, and the last two rows describe the features of our new schemes.

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. Industry wide standardization of inverse multiplexers has been initiated by the BONDING [Dun94][Fre94][Gro92] consortium, which has issued standards for a frame structure and procedures for establishing a wideband communications channel by combining multiple switched 56 and 64-kbps channels. The BONDING scheme uses a fixed size frame structure and skew compensation for reordering, together with frame sequence numbers to recover from errors. The BONDING scheme requires special hardware at the sender and receiver.

The ATM Forum is considering a standard for ATM cell striping called AIM (ATM Inverse Multiplexing) based on delay compensation. As with BONDING, this works only when the skew can be bounded tightly.

The establishment of Gigabit testbeds led to the design of several network adaptors which striped data across multiple slower speed ATM links to achieve gigabit throughputs. As mentioned earlier, the IBM SIA adaptor [TG93] does striping over 4 STS-3c channels. The Bellcore HAS adaptor [Joh95] stripes HIPPI packets over SONET lines using a First-Come-First-Serve (FCFS) striping policy, while the CASA Gigabit testbed [JD93] uses round robin striping at the byte level. The OSIRIS Adaptor[DP94] does 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. All these schemes either rely on extra hardware to do load sharing (e.g., using byte striping), or rely on extra information for resynchronization (e.g., information embedded in SONET or ATM headers). Thus none of these schemes meet all our goals.

Existing striping schemes which operate at higher levels usually sacrifice either fair load sharing or FIFO delivery. For example, the Random Selection scheme [Bay95] relies on random assignment of channels to packets to ensure load sharing, but does not provide FIFO delivery. The same is true for the Shortest Queue First scheme used in the EQL serial line driver in the Linux operating system: in this scheme, the channel with the smallest queue is selected for transmitting the next packet. On the other hand, the Address-based Hashing scheme [Bay95] relies on hashing packet addresses to channels to route packets destined for the same address over the same channel. This provides FIFO delivery of packets destined for the same address, but does not provide load sharing for packets addressed to any given destination.

The Internet standard RFC1717 specifies MPPP(PPP Multipoint). This provides a framework and packet formats for striping across multiple PPP links. However, no algorithm is specified for either the sending or the receiving end. In addition, the sender modifies each packet by adding sequence numbers to it.

Our strIPe protocol described later differs from MPPP in three fundamental ways. First, it works transparent to IP over any interface, not just a PPP interface. Second, there is no modification of any data packet, since no new header is tagged along with each packet. This is essential for striping over high speed interfaces. Finally, MPPP supplies no algorithm for striping at the sender and resequencing at the receiver, while our strIPe protocol does.

While some of the solutions described in [TG93, Joh95, JD93] look superficially similar to ours (e.g., the use of queues at the receiving ends of channels), these schemes rely on extra information such as SONET framing for synchronization, which is unavailable for many channels. Further, they either do not provide general mechanisms for fair load sharing or rely on mechanisms like byte striping that are infeasible in many contexts. By contrast, we use a distributed algorithm to restore synchronization, and a transformation of a fair queuing algorithm to provide fair load sharing. Our algorithms are applicable to a wide variety of channels.

3 Using Fair Queuing algorithms for Load Sharing

We 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. Henceforth we will refer to such algorithms as FQ algorithms. In FQ, we 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*.

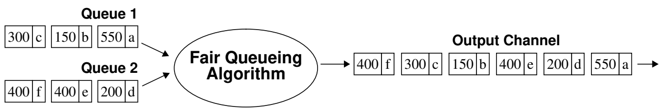

Figure 2: Example of fair queuing

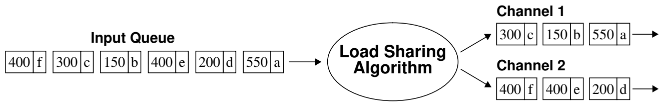

Figure 3: Example of load sharing

Figures 2 and 3 explain the intuitive relationship between fair load sharing and fair queuing. In Figure 2, an arbitrary FQ algorithm feeds 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. Notice that the bandwidth of the channel is partitioned roughly equally among the channels. They have the same fair share of 500 bytes each. 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. 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. We believe this to be an important insight, since it suggests that the considerable amount of work done in the FQ area can be directly applied to load sharing. However, as we shall see, only a subset of FQ algorithms can be used for load sharing.

3.1 Causal and Non Causal 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 packets at the head of each queue, the current queue sizes, and so on. For instance, the DKS algorithm [DKS89] 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 a *Causal* FQ (CFQ) *algorithm.* All other FQ algorithms are called non-causal algorithms. Thus the DKS fair queuing algorithm [DKS89] is non-causal, while ordinary round robin is causal.

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-causal. 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, CFQ 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: $f(s) = s$; finally, the function $g(s,p)$ merely increments the pointer to the next queue.

3.2 Use of CFQ 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 CFQ algorithm and end with a fair load sharing algorithm on the right.

The CFQ 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 input channel (see the right of Figure 4), the load sharing algorithm sends packet p to low speed output 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.

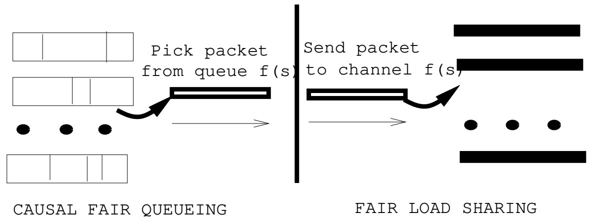

Figure 4: Consider a backlogged execution of a fair queuing algorithm. If the algorithm is causal we first apply a function $f(s)$ to select a queue. We transmit the packet p at the head of the selected queue and then update the state using a function $g(s,p)$. We can obtain a fair load sharing algorithm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

To precisely evaluate fair sharing, we define throughput fairness for both deterministic and probabilistic fair queuing schemes. In discussing throughput fairness it makes sense to only consider the case when all input queues are backlogged.

Consider a fair queuing scheme with several input queues. In the start state each queue contains a sequence of packets with arbitrary packet lengths. Define a *backlogged execution* to be an execution in which no input queue is ever empty. There are an infinite number of possible backlogged executions corresponding to the different ways packets, especially packet lengths, can be assigned to queues in the start state. In a backlogged execution we 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 causal. Thus there is no loss of generality in considering only backlogged executions.

3.4 The Transformation Theorem

We show that a load sharing algorithm obtained by transforming an CFQ algorithm as shown above has the same fairness properties as the original CFQ algorithm.

Theorem 3.1 *Consider an* CFQ *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 CFQ 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, e.g., the execution shown in Figure 3. From execution E we generate a corresponding execution E' (e.g., the execution shown in Figure 2) of the original CFQ 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 CFQ 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 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. □

The theorem can be used to convert causal fair queuing algorithms into load sharing algorithms. A simple example is a randomized fair queuing (RFQ) scheme that randomly picks a queue to service. RFQ can be transformed into a randomized load balancing algorithm that keeps the expected number of bytes transmitted on each line the same. However, a more useful example is the SRR scheme that we describe next.

3.5 Surplus Round Robin (SRR)

We turn to a specific example of a CFQ algorithm, which we call Surplus Round Robin (SRR), to which the transformation theorem can be applied. 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.

Figure 5 graphically illustrates the operation of the SRR CFQ algorithm. In the figure, we see two input queues, one containing packets labeled a, b, c, in that order, and the other containing packets d, e and f. Both queues are assigned a quantum of 500 each. In addition to its label, each packet is also marked with its size. The figure shows the values of the DCs associated with each queue as the SRR algorithm executes. Note that a *round* is a sequence of visits to consecutive channels, before returning to the starting channel. The DC of each queue is incremented by the quantum associated with that queue in each round. When the DC becomes non-positive, packets are sent from the next queue.

In Figure 5 the DC of channel 1 is initially the quantum size (500). After sending out packet a (of size 550), the DC of channel 1 becomes $500 - 550 = -50$ which is negative. Thus the round robin

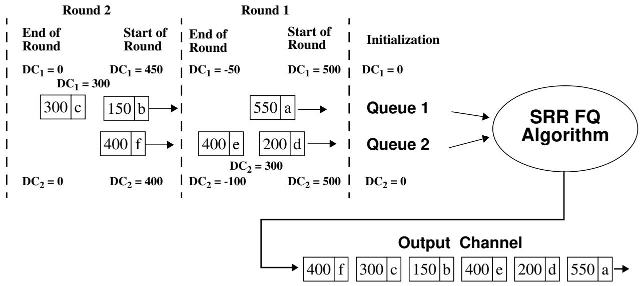

Figure 5: Example of SRR Fair Queuing. Each queue has a quantum of 500 bytes

pointer moves on to channel 2, where two packets, d and e, with combined size 600, are sent before the DC of channel 2 becomes $500 - 600 = -100$. At this point, the round robin scan returns to channel 1 to start round 2. A fresh quantum of 500 is added to the DC for channel 1, leaving a value of 450, which now allows packets b and c to be sent out in the second round.

As can be seen, SRR sends roughly the same amount of data from each queue. It is possible to precisely characterize throughput fairness for the SRR FQ algorithm. Let the quantum of service assigned to queue i be $Quantum_i$. Let the maximum quantum among all the channels be $Quantum$. Let the maximum packet size be Max.

Theorem 3.2 *Consider any execution of the* SRR FQ *algorithm in which queue i is backlogged. After any K rounds, the difference between the bytes that queue i should have sent, i.e., $K \cdot Quantum_i$, and the bytes that queue i actually sends is bounded by $Max + 2 \cdot Quantum$.*

The proof is similar to the proof of fairness of DRR [SV94].

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, 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.

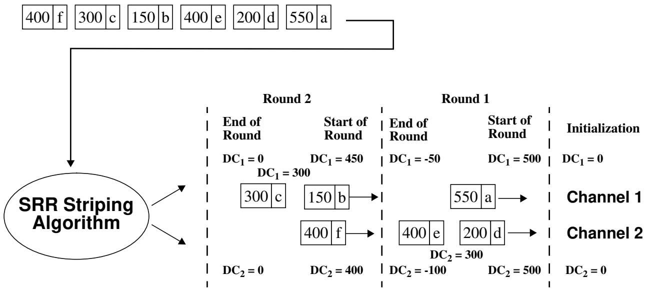

Figure 6: Example of SRR Load Sharing. Each channel has a quantum of 500

Figure 6 illustrates the operation of the SRR load sharing algorithm. The load sharing algorithm preserves the same fairness bounds as the FQ algorithm. Using the terminology of the previous theorem:

Lemma 3.3 *Consider any execution of the* SRR *load sharing algorithm. After any K rounds, the difference between the bytes that should have been sent on Channel i, i.e., $K \cdot Quantum_i$, and the bytes actually sent on Channel i is bounded by $Max + 2 \cdot Quantum$.*

The SRR load sharing scheme has a number of nice properties that makes it appropriate for use in a practical packet striping algorithm. It divides the bandwidth fairly among output channels even in the presence of variable length packets. It is 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. It 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.

4 FIFO Delivery using Logical Reception

This section describes techniques for ensuring FIFO delivery at the receiver. Our main idea is what we call *logical reception*.

Logical reception combines two separate ideas: *buffering at the receiver* to allow physical reception to be 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 resequencing algorithm. Notice also that if we look at the picture at the receiver node, it is clear that the receiver is performing a fair queuing function. But we have already seen a connection between channel striping and fair queuing schemes. Thus the main idea is as follows. The receiver can restore the FIFO stream arriving to the sender if it uses a fair queuing algorithm that is derived from the channel striping algorithm used at the sender.

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 a Causal Fair Queuing (CFQ) algorithm, then the receiver can run the CFQ 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, coupled with the buffering and receiver blocking, ensures *logical* FIFO *reception*, irrespective of the nature of the skew present between the various channels. Formally:

Theorem 4.1 *Let B be the load striping algorithm derived by transforming an* CFQ *algorithm A. If B is used as a channel striping algorithm at the sender and A is used as the resequencing 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.*

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 the sender must periodically resynchronize with the receiver. Such synchronization can be done quite easily as shown in Section 5. If packets are lost infrequently and periodic synchronization is done quickly, logical reception works well. We discuss performance simulations in Section 6.

Why is it necessary for the fair queuing algorithm to be causal? 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 CFQ algorithms.

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.

Even in the case when sequence numbers can be added to packets, logical reception can help simplify the resequencing implementation. Some of the hardware implementations for resequencing, e.g., [McA93], rely on hardware to sort out of order packets and modified packet formats. Logical reception can be used to avoid such sorting. The sequence number inserted by the sender is now needed only for confirmation, since logical reception suffices for FIFO delivery. The sequence numbers, however, provide sequencing of packets even when the sender and receiver lose synchronization, and *guarantee* FIFO reception.

The most important application of logical reception (see goals listed earlier) is the case when sequence numbers *cannot* be added. Unfortunately, in this case we cannot guarantee FIFO delivery always. We refer to this mode of packet reception, in which the receiver maintains FIFO delivery, except during periods of loss as *quasi-*FIFO *reception*. This is in contrast to guaranteed FIFO reception. For quasi-FIFO reception to be of practical significance, we need to restore synchronization periodically, or the receiver will continue to deliver packets out of order. We now describe the synchronization protocol.

5 Synchronization Recovery at the Receiver

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. Sending marker packets does not require modifications to the data packets, which is one of the desirable properties of a striping scheme. The only requirement is that the lower level protocol provides a distinct codepoint (i.e., demultiplexing information) for the marker packets, to distinguish markers from normal data packets. Such codepoints are available for ATM virtual circuits, e.g., OAM cells or LLC/SNAP encapsulation, and for most existing links. For example, on Ethernet, codepoints for marker packets are available simply by using a different packet type field. Note that using a different type field for marker packets does not alter ordinary data packets or link packet formats in any way, as opposed to existing striping schemes (e.g., MPPP) which require a modified link packet format for all packets.

We now describe a marker synchronization scheme for the striping scheme using the SRR striping algorithm. As previously defined, a *round* is a sequence of visits to consecutive channels before returning to the starting channel. In each round, the sender sends data over all channels. Similarly, in each round, the receiver receives data from all channels.

The state at the sender can be fully specified by specifying the current round, and the value of the SRR Deficit Counters (DCs) at each channel. Similarly, the state at the receiver consists of the current receiver round number, and the value of the SRR DCs at each of the channels as seen by the receiver. In the absence of packet loss or corruption, the state at the sender would correspond to the state at the receiver, modulo the packets in transit, and the receiver would stay in synchronization. However, if there were a packet loss, then the two states would differ, and the receiver would run out of step with the sender.

Intuitively, each packet sent can be *implicitly* numbered with a tuple (R, D), where R is the round number before the packet is sent and D is the value of the DC before the packet is sent. Similarly, at the receiver, a received packet can be implicitly numbered by the round number and DC before the packet is received. If the (implicit) receive and send numbers for each packet are identical, then the receiver will deliver packets in the correct order.

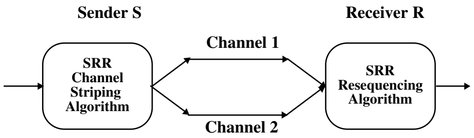

Figure 7: Configuration to illustrate synchronization recovery

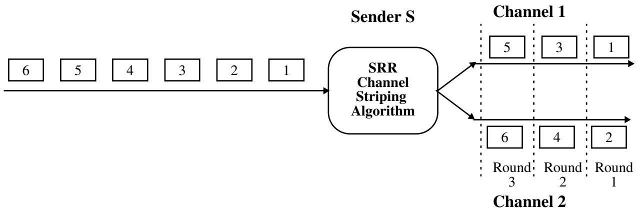

Figure 8: Sender sends packets

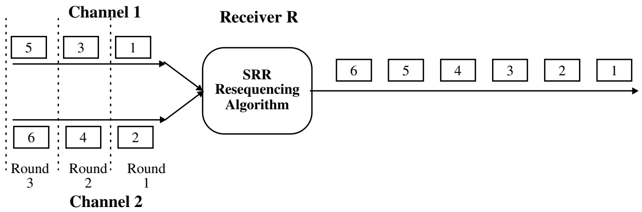

Figure 9: Receiver in synchronization with the sender

A simple example of the illustrating the idea behind synchronization recovery is illustrated in Figures 7 to 13. We consider the configuration shown in Figure 7. The SRR CFQ algorithm is used for resequencing at the receiver, while the transformed version of the algorithm is used as the striping algorithm at the sender. There are two channels of equal capacity linking the sender to the receiver. We assume that all packets are of equal size, and that the quantum of service for both channels is the same and equal to the packet size. In such a scenario, SRR reduces to RR.

Figure 8 shows packets arriving at the sender, and being striped across the two channels. Each packet is numbered in the order of arrival. As can be seen, packets 1 and 2 are sent in the first round, packets 3 and 4 in the second round, and so on. Figure 9 shows

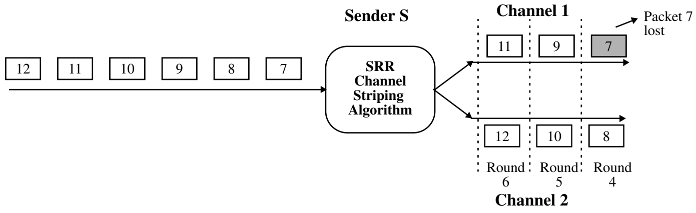

Figure 10: A channel loses a packet

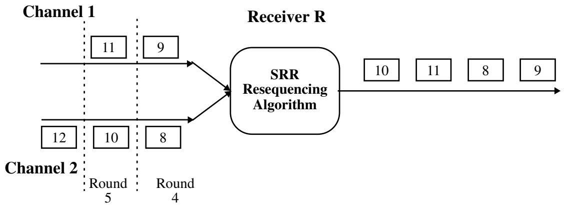

Figure 11: Receiver out of synchronization

the operation of the receiver. In the first round, the receiver picks one packet from channel 1, followed by one packet from channel 2. Similarly in the second round, the receiver picks packets 3 and 4 from channels 1 and 2 respectively. Thus, the receiver delivers packets in the same order as the sender receives them.

Figure 10 shows packet 7 being lost in one of the channels. We assume that any packet corruption causes the packet to be discarded, and not handed over to the resequencing algorithm. The effect of this loss is to cause the state maintained at the receiver to differ from the state maintained at the sender. As can be seen in Figure 11, in round 4, the receiver expects packet 7 on channel 1, but instead picks up packet 9, since packet 7 is lost and packet 9 is the next packet sent on channel 1. This causes the receiver to go out of synchronization with the sender, and start delivering packets out of order.

The sender periodically sends marker packets on each channel, containing its state, which in this simple case consists of the round number G. As shown in Figure 12, the sender sends a marker packet labeled M before round 7, containing the round number G set to 7. When the marker packet reaches the receiver, as shown in Figure 13, the receiver sees that there is a difference between the round number maintained by the receiver, which is currently 6, and the round number carried in the marker packet, which is 7. This causes the receiver to skip this channel in the current round, and proceed to the next channel. This is because the difference in round numbers is caused by missing packets, indicating that the receiver has skipped ahead out of turn on this channel, and therefore needs to wait that many rounds before visiting that channel again. Hence in round 6, the receiver skips channel 1. By round 7, the receiver is fully in synchronization with the sender, as can be seen in Figure 13.

We now describe the reasoning behind the channel skipping done by the receiver. Suppose the sender sends packet p on channel c,

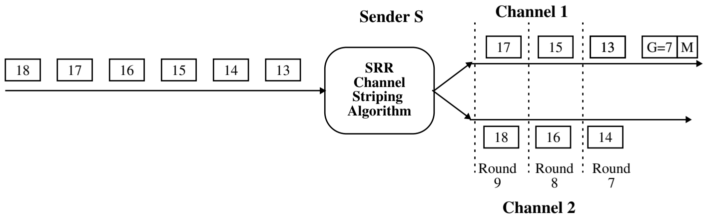

Figure 12: Sender sends a marker packet

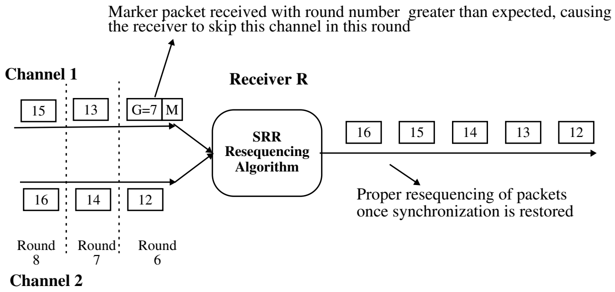

Figure 13: Synchronization restored at receiver

before sending packet q on channel c'. Then either q's send round number is greater than that of p, or the round numbers will be the same and c is visited before c' in the round robin cycle. Thus if the receiver has the same receive numbers for p and p' the receiver will deliver p before p' as long as receiver delivery meets two conditions: **C1:** The receiver never delivers a higher round number packet before a lower round number packet. **C2:** The receiver visits channels in the same order as the sender in the round robin cycle.

Condition C2 can easily be enforced if the receiver and the sender number the channels in the same way, and both visit channels in increasing channel number order during a round robin cycle. This can be guaranteed by having each marker carry the sender number of the channel which can be adopted by the receiver. It remains to ensure that:

- Eventually all packets have the same send and receive numbers after packet loss stops .
- Condition C1 is enforced.

To ensure synchronization of send and receive numbers, we maintain explicit packet numbers for both sender and receiver. Both sender and receiver maintain a global round number G that is incremented after one round robin scan over the queues. This together with the DCs provides explicit packet numbers for each packet, though these are not carried in packets. The next step is obvious: each periodic marker packet on a channel c carries the packet number for the next packet to be sent on channel c. The receiver also maintains a local round number r_c for each channel c. When the receiver receives a marker packet (r, d) for channel c, it sets $r_c = r$ and the DC of channel c to be d. After packet loss stops, it is obvious that this will synchronize the packet sender and receiver numbers for all future packets on channel c.

Finally, to maintain condition C1, the receiver maintains a global round number G that is incremented on every round robin scan. When the receiver reaches a channel whose value of $r_c > G$, it simply skips that channel in the current round robin scan. The intuition is that the receiver has lost some earlier packets on channel c and has arrived "too early" at scanning this channel. Channel c will continue to be skipped until $G = r_c$ at which point channel c is serviced with the usual SRR algorithm. This clearly maintains condition C1.

Assume that for each channel i, $Quantum_i \geq Max$ (i.e., the quantum assigned to each channel allows the sending of one maximum sized packet). This assumption prevents channels from being skipped in the round robin order because their Deficit Counters do not allow the sending of a packet. Using a formal model, we can prove that:

Theorem 5.1 Marker Recovery: *Let t be the first time after all channel errors stop that a marker is delivered on every channel. The marker algorithm restores* FIFO *delivery after t.*

Thus the algorithm recovers from errors very quickly (time between sending the marker plus a one-way propagation delay). Note that in practice, channel errors never stop; the theorem says that if the errors stop for a period longer than the recovery time, then the system will be resynchronized. We have implemented this algorithm and found that it works well, by providing quick restoration of FIFO delivery, even for fairly high error rates. The marker recovery theorem assumes that the only channel error is detectable packet corruption or packet loss. It is also possible to make the marker algorithm self-stabilizing (i.e., robust against any error in the state) by periodically running a snapshot [CL85] and then doing a reset [Var93]. We deal with sender or receiver node crashes by doing a reset.

The main idea behind the marker recovery protocol is a way of numbering packets on a channel that depends only on the data sent or received on a channel. A global numbering scheme such as a global sequence number appears to require expensive global synchronization across all channels. By using a per-channel numbering scheme, that also includes the relevant state (i.e., the DCs) we can synchronize each channel independently. The only global condition that needs to enforced is condition C1, which is implemented easily by skipping channels that have lower round numbers than incoming marker packets.

6 Implementation and Performance Evaluation

Having looked at the theory underlying the use of CFQ algorithms for packet striping and resequencing, we now turn to implementation issues. We propose a model for striping IP packets over multiple data link interfaces in Section 6.1. We implemented our scheme in the NetBSD kernel and measured its throughput gains when striping was done over a combination of an ATM and an Ethernet link. This allowed us to see the effects of SRR versus round robin, and the effects of using logical reception versus no resequencing at all. The implementation is discussed in Section 6.2.

Finally, in Section 6.3, we discuss other experiments and simulations we performed to verify the load sharing and FIFO properties of our scheme. By implementing the SRR striping protocol above the transport layer and by simulating packet loss, we were able to study the effect of packet loss on throughput and various parameters of the marker recovery scheme. We also evaluated the effect of the quasi-FIFO delivery of our scheme, during losses, on a video application.

6.1 A Model for Transparent IP Striping on a LAN

We present a simple architectural framework for striping IP packets over multiple data link interfaces, which can include multi-access interfaces like Ethernets and Token Rings. The framework is as shown in figure 14. We create a virtual interface, which we term the strIPe interface, between IP and the actual data link interfaces which are to be striped. In this fashion, IP striping can be totally transparent to both IP and upper level protocols and applications. We refer to this technique of striping IP packets as the strIPe protocol.

In current protocol stacks, the IP protocol sends and receives IP packets from multiple IP *interfaces*, which are composed of IP convergence layers on top of the data link layers (see Figure 14). The convergence layer is responsible for mapping IP addresses to data link addresses, and encapsulating the IP packet in a data link frame. For example, for Ethernet interfaces, the convergence layer performs ARP. The strIPe layer becomes one such convergence layer below IP and above the data links that the packets will be striped over. The strIPe layer implements the sender side striping algorithm and the receiver side resequencing algorithm. In our case, both algorithms are based on SRR.

Whenever a sending host sees that a packet is to be routed to one of the IP addresses corresponding to the receiver with multiple channels, it sends the packets to the strIPe layer. This is accomplished

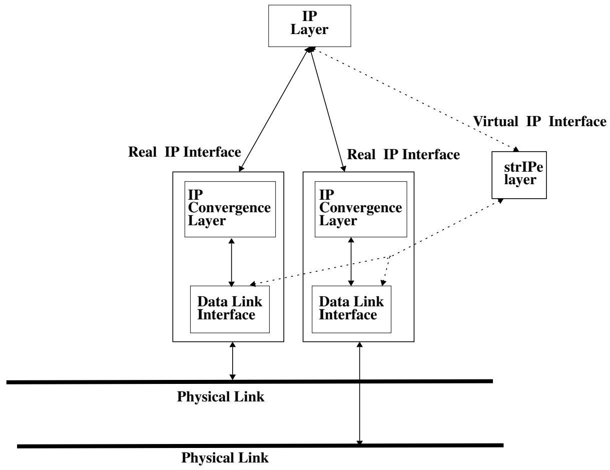

Figure 14: The position of the strIPe layer. The dotted lines indicate the data flow between IP and the data link layer via the strIPe layer.

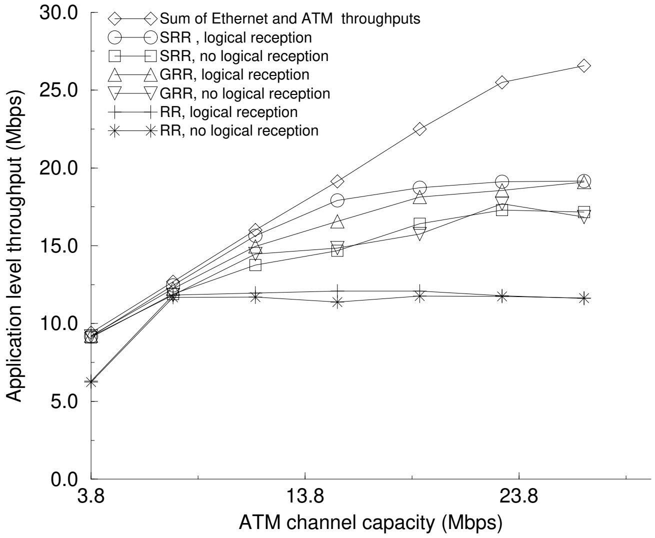

Figure 15: Performance of SRR as the capacity of the ATM PVC is varied

by modifying the routing table of the sending host. Recall that it is possible for host specific routes to override network specific routes. Thus, if the two ethernets are on IP networks Net1 and Net2, and if the receiving host's two IP addresses are Net1.B and Net2.B, then we simply make entries in the sending host's routing table, asking it to route packets to Net1.B and Net2.B to interface C, which corresponds to the strIPe interface.

At the receiving end, the data link interfaces hand over striped packets to the strIPe layer for resequencing. This is accomplished by using a different codepoint in the data link layer header for striped IP packets. The strIPe layer at the receiving end then resequences the packets before handing them to IP. There are other subtleties to do with ARP handling and differing MTU sizes that we defer for lack of space. We do note, however, than our model restricts the maximum packet size, or the Maximum Transmission Unit MTU of the strIPe interface to the minimum MTU of the underlying physical interfaces.

6.2 Performance of the NetBSD Implementation

The strIPe protocol was implemented in the NetBSD/i386 kernel. Our setup consisted of two Pentium workstations, each with two IP interfaces. One interface was 10 Mbps Ethernet, and the other was an ATM interface, which sent IP packets through a Permanent Virtual Circuit (PVC). The bandwidth of the PVC could be modified in hardware. Figure 15 depicts the performance of our strIPe protocol when used to stripe IP packets across the Ethernet and ATM interfaces. The bandwidth of the PVC used for IP traffic was varied, and the effect on striping throughout was studied. The throughput measurements were carried out at the application level, using a sending program which sent a random mixture of small and large packets to the receiving program on the other workstation over a TCP connection.

Besides the throughput of our strIPe protocol, we also implemented and measured the performance of four other striping variants to gain insight into the advantages of SRR versus round robin, and logical reception versus no resequencing. We first measured the throughput of the ATM and Ethernet interfaces separately, for each value of PVC bandwidth, and calculated the sum of the individual throughputs: clearly this is an upper bound on striping performance. Note that in this case, only one interface is used for sending data at a time, as opposed to the striping case, in which two interfaces are used. Second, we replaced SRR by generalized round robin (GRR), which allocates packets to interfaces based on the closest integer ratio of their bandwidths. Third, we implemented GRR without logical reception — i.e., no resequencing is done. Finally, we implemented ordinary round robin (RR), which just alternates between channels. Thus, besides the throughput of our strIPe protocol, in Figure 15 we have a throughput upper bound, as well as four variants of strIPe with one or more of its features disabled.

The throughput upper bound was measured by sending packets separately over the Ethernet and ATM interfaces, and by adding the individual throughputs. The packet sizes were kept the same for both interfaces. We observe that the throughput upper bound increases linearly before starting to fall, as the CPU cannot keep up with the network at higher speeds with the selected range of packet sizes. The throughput with strIPe (SRR + logical reception) is approximately equal to the throughput of the sum of the ATM and Ethernet interfaces till the ATM PVC bandwidth is set to 14 Mbps, after which it starts flattening. This is because the workstation has to service more interrupts in the striping case. Note that in measuring the upper bound, only one interface is used at a time. With a single interface under heavy load, multiple packets can be received in a single interrupt routine. This effect is less pronounced with striping, where interrupts are received from multiple interfaces. Consequently, there is a significant increase in the number of interrupts, and correspondingly in the processing overhead. Note that the bottleneck is in the interrupt driver processing, as opposed to the striping overhead.

The throughput of strIPe is consistently better than the variants that disable features. Its throughput is better than the variant that uses GRR, and better than the variant that disables logical reception. We note that the throughput of GRR is higher than that of GRR without resequencing. RR is consistently worse than the other variants: as the throughput of the ATM interface increases, RR performance is still limited by the slowest speed (i.e., Ethernet) interface. Thus, increasing the speed of the ATM interface does not improve throughput beyond a critical point. The initial increase in RR throughput is due to the fact that at those points, the rate of the ATM PVC is less than that of Ethernet. Note that RR is commonly used in existing striping protocols, as discussed in Section 2.1.

There is a throughput gain using SRR over GRR, although the difference is not marked in Figure 15. This is because roughly the same amount of data is carried over both interfaces, and TCP is able to keep the transmit queues of both interfaces full. The advantage of SRR over GRR, of course, is that it is always possible to construct a worst case sequence which will cause GRR to perform badly, while SRR does not have any such drawback. To show that GRR does not work well in all situations, the following experiment was conducted. The rate of the PVC was set to 7.6 Mbps, so that the ATM interface gave the same throughput as the Ethernet (6 Mbps). Note that in this case GRR reduces to RR. Then packets were sent in deterministic fashion, with the bigger (1000 bytes) packets alternating with the smaller (200 bytes) ones. With SRR, the packet arrival sequence did not have any effect on throughput, yielding a striped throughput of 11.2 Mbps. With GRR, the bigger packets are all sent on one interface, and the smaller packets on the other, so the throughput drops dramatically to 6.8 Mbps.

We note that the throughput on the single ATM interface can be improved considerably by using a large MTU size for the ATM interface. For example, we obtain throughputs in excess of 70 Mbps over an ATM interface using 8 KB sized packets. However, our striping algorithm restricts the MTU size used for a collection of links to be the smallest MTU size, which in this case is that of the Ethernet interface. This problem does not appear to be specific to our scheme, but seems to apply to any striping algorithm that does not internally fragment and reassemble packets. Since the overall throughput is considerably dependent on MTU size, we recommend that striping be done on links with similar MTU sizes. Our experiments should be viewed as a validation of our algorithms; they *do not* indicate that striping across an ATM and an Ethernet interface is a good idea.

6.3 Transport Layer Simulations and Experiments

In addition to implementing the strIPe protocol in the NetBSD kernel, a striping protocol was also implemented at the transport layer by striping packets across multiple application sockets using the same SRR striping and resequencing algorithm. The aim of this experiment was to study the effect of marker position and frequency on synchronization recovery, and to study the effect of packet loss on applications, using quasi-FIFO delivery of packets at the receiver.

The main findings of our experiments were as follows:

- For arbitrary levels of packet loss (measured up to 80%), the marker based resynchronization scheme was able to restore FIFO delivery once packet losses stopped.
- For a given loss rate, increasing the frequency of marker packets decreased the occurence of out of order delivery of packets.
- For a given loss rate, the position of the marker packet within a round had an effect on the number of out of order deliveries, with the minimum number of out of order deliveries occuring when the marker was sent either at the beginning or end of the round.
- For channels not providing flow control, e.g., UDP channels, a simple credit based flow control scheme proposed by Kung et. al. [KC93] proved very effective in eliminating packet loss due to channel congestion. This scheme was particularly well suited to our striping scheme, since the credits could be piggybacked on the periodic marker packets.
- To see the tolerance of real world applications to possible packet reordering introduced by quasi-FIFO delivery, video traces sent by the NV video conferencing application were captured. The stored traces were then striped over multiple UDP channels with a controlled amount of loss. The received traces, with some reordered packets, were fed to the NV application. Only at packet loss levels of 40% and above were any perceptible differences found in the NV playback, as compared to the original packet stream. Incidentally, pure packet loss of 40% (without any reordering), produced the same qualitative difference, suggesting that the effect of packet reordering was insignificant compared to the effect of packet loss.

7 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 Causal Fair Queuing (CFQ) algorithms into load sharing algorithms. This transformation also provides load sharing for channels having different capacities. We solve the FIFO problem using logical reception, which combines the two ideas of receiver buffering and receiver simulation of the sender algorithm. It is important to note that in order for receiver simulation to work it is only necessary that the sender algorithm be causal, which is guaranteed by our fair load sharing schemes.

Logical reception must be augmented with periodic resynchronization to handle packet losses. We have described an elegant resynchronization scheme that restores synchronization quickly in approximately a one-way propagation delay, as opposed to a conventional reset based scheme which would have taken a round-trip delay. We have formally proved our protocol correct. We have implemented and simulated this protocol for various values of error rates. We found that the scheme works well for error rates up to 80%. We also found by experiment that the best position to place a marker was at the end of a round.

We implemented the basic ideas at the transport level, and then developed a framework to transparently incorporate them into the IP protocol stack. We then implemented this protocol, that we called strIPe, in the NetBSD kernel. Our experiments indicate that strIPe is capable of providing nearly linear speedup with dissimilar links. We also confirmed that the use of SRR was better than RR because of the guaranteed performance improvement. The performance improvement for resequencing data packets is sensitive to whether the receiver is a bottleneck: for fast receivers, the cost of dealing with out-of-order data packets may not be an issue. However, for applications that require in-order packet delivery, e.g., MPEG video, resequencing is crucial.

We have also described and 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].

We believe the strIPe protocol based on SRR, logical reception, and periodic resynchronization is suitable for practical implementation, even in hardware. SRR requires only a few extra instructions to increment the Deficit Counter and do a comparison; the marker based synchronization protocol is also simple since it only involves keeping a counter and sending a marker containing the counter.

References

[Bay95] Bay Networks Web Page. http://www.wellfleet.com/Products/Routers/Protocols/Traffic2.html, 1995.

[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.

[JD93] W. St. John and D. DuBois. CASA Gigabit Testbed Annual Report. Technical report, 1993.

[Joh95] C. Johnston. Presentation at CNRI Gigabit Testbed Workshop , June 1993, June 1995.

[KC93] H.T. Kung and Alan Chapman. The FCVC (Flow Controlled Virtual Channel) proposal for ATM networks. In *Proceedings of the International Conference on Network Protocols*, October 1993.

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

[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.

[TG93] V. Theoharakis and R. Guerin. SONET OC-12Interface for Variable Length Packets. In *Proceedings of the Second International Conference on Computer Communications and Networks*, pages 21–25, June 28-30,1993.

[TS95] C. Brendan S. Traw and Jonathan Smith. Striping within the Network Subsystem . *IEEE Network*, pages 22–29, 1995.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.