An Architecture for Packet-Striping Protocols

ADISESHU HARI
Lucent Bell Labs
GEORGE VARGHESE
University of California
and
GURU PARULKAR
Washington University

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

George Varghese is supported by NSF Grant NCR-9405444 and an ONR Young Investigator Award.
An earlier version of this article appeared in the *ACM Sigcomm'96 Conference.*
Authors' addresses: A. Hari, Networking Techniques Research, Lucent Bell Labs, 101 Crawfords Corner Road, Holmdel, NJ 07733; email: hari@bell-labs.com; G. Varghese, Computer Science and Engineering, University of California, San Diego, CA; email: varghese@cs.ucsd.edu; G. Parulkar, Applied Research Lab, Washington University, St. Louis, MO; email: guru@arl.wustl.edu.

improvements over conventional round-robin striping schemes and striping schemes that do not resequence packets. Some aspects of our solution have been implemented in Cisco's router operating system (IOS 11.3) in the context of Multilink PPP striping.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architecture* (OSI model)

General Terms: Algorithms, Design, Measurement, Performance, Theory

Additional Key Words and Phrases: Causal fair queuing, fair queuing, load sharing, Multilink PPP, packet striping, stripe protocol, striping

1. INTRODUCTION

Parallel architectures are attractive when scalar architectures with the required performance are unavailable or have poor cost-to-performance ratios. 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 [Traw and Smith 1995] 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 trade-offs, in 1998 a T1 line (1.5Mbps) from a typical long-distance carrier had a per-circuit cost of US $3600 and a per-mile cost of US $4.30 while a T3 line (45Mbps) from the same carrier had a per-circuit cost of US $28,895 and a per-mile cost of US $67.66. The price differential between T1 and T3 lines makes striping across T1 links attractive, especially when the customer bandwidth requirement begins to grow beyond 1.5Mbps but is still much less than 45Mbps.

On the other hand, many of the gigabit testbeds [Theoharakis and Guerin 1993] have resorted to striping because of the unavailability of high-speed equipment: for instance, the IBM SIA adaptor [Theoharakis and Guerin 1993] emulates a SONET STS-12 line using four STS-3c lines, and the Washington University Gigabit Switch [Richard and Figerhut 1998] uses two 1.2 Gbps links to emulate a 2.4 Gbps link. Similarly, transmitting data over multiple phone lines is one way "power users" increase their effective bandwidth to their ISPs (Internet Services Providers) in the absence of high-capacity data connections to the home.

One may argue that striping is only a temporary solution to transient bandwidth problems that will eventually go away. We contend, however, that at any level of technology there will always be such bandwidth problems. First, bandwidth comes in discrete granularities, and striping allows the gaps between discrete steps to be bridged more evenly. Second, if

the maximum bandwidth available over a single physical channel is B, striping provides the only way to provide bandwidths greater than B.

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*) in the channels. In many applications, the receiver must reconstruct the sender sequence from the parallel packet streams. This reconstruction 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 7, 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 article, 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 article 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 article include an architectural model, a working software implementation of the model and the striping algorithm, and measurements and evaluation of the striping algorithm.

1.1 Overview of Solution Components

In Section 2, we present a model of the load-sharing problem. To provide load sharing in the presence of varying-length packets, we use *fair load-sharing* algorithms. We define fair load sharing to be load sharing proportional to the capacities of the individual channels. 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 synchro-

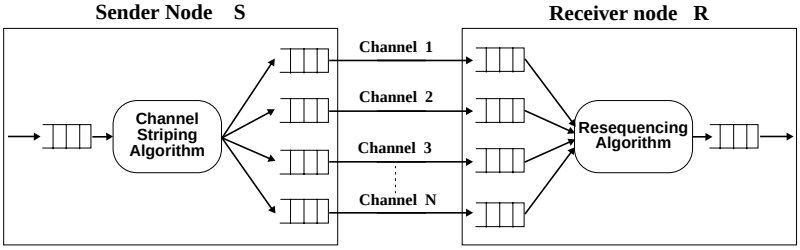

Fig. 1. Channel-striping configuration.

nization. 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 two of our ideas: SRR versus round robin, and logical reception versus no resequencing. We discuss related work in Section 7, and in Section 8 we present our conclusions regarding the applicability of our work.

2. MODEL

To allow our algorithms to be widely applicable, we use a broad definition of a channel. For the rest of this article, we define a *channel* to be a logical FIFO path at either the physical, data link, network, or transport layers. We use the term *packet* to refer to the atomic unit of exchange between two entities communicating across a channel. The generic channel-striping configuration is depicted in Figure 1.

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 arriving at 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 occasional 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. Variable skew is important to model realistic network channels. Variable skew 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 provides FIFO delivery between a given sender and receiver.[1] Network layer channel examples include ATM or X.25 virtual circuits. Even in datagram networks, it is possible to construct "network" channels (e.g., by using IP encapsulation or strict IP source routing to set up multiple paths between two IP endpoints). 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 common examples appear to be data link and virtual circuit channels.

2.1 Goals

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. Recall that fair load sharing (FLS) refers to load sharing across multiple channels in such a manner that the amount of data sent over each channel is proportional to its capacity. In the case when the traffic is overloaded, FLS ensures that the bandwidth is fully utilized. One assumption we make at this point is that packet striping at the sender is followed by link queuing rather than link transmission, i.e., there exists a transmit buffer on each link or channel. This is shown in Figure 1. This per-channel transmit buffer is needed to allow the load-sharing algorithm to insert packets in one channel queue and move quickly to another channel queue without being blocked by the transmission of a previous packet on that channel. In addition, the channel-striping scheme should be robust enough to recover synchronization in case of bit and burst errors, and be scalable enough to impose little computational and storage overhead.

To understand why this combination of goals may be difficult, consider round-robin striping, in which 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 order of arrival of packets at the receiver may differ from their

[1]A LAN is not strictly FIFO, but is usually close enought that it can be modeled as a FIFO channel with occasional errors.

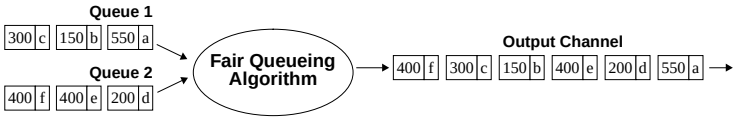

Fig. 2. Example of fair queuing.

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, adding sequence numbers 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, the packets can be split into fixed-size striping units of data, which can 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 is often done at the hardware level using devices known as inverse multiplexers. We discuss two hardware inverse multiplexing schemes in detail in the section on related work. In contrast, our work focuses on striping scenarios where the packets cannot be reformatted.

3. USING FAIR QUEUING 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*.

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

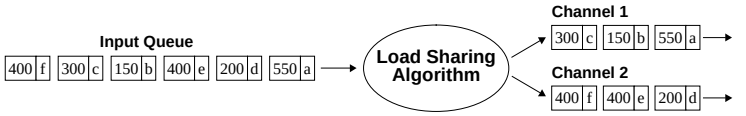

Fig. 3. Example of load sharing.

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 "time reversal" to be a useful 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 Noncausal Fair-Queuing Algorithms

Consider a node running an 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 WFQ algorithm [Demers et al. 1989] depends on the packets at the head of each queue in order to simulate bit-by-bit round robin.

In the backlogged case, if an 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.[2] All other FQ algorithms are called noncausal algorithms. Thus the WFQ fair-queuing algorithm [Demers et al. 1989] is noncausal, while ordinary round robin is causal.

Why do we restrict ourselves to backlogged FQ behavior? In the nonbacklogged case, most FQ algorithms maintain a list of active flows as part of their state. This mechanism allows them to skip over empty queues. However, this mechanism also makes almost all FQ algorithms noncausal. Thus, for our transformation we restrict ourselves to the backlogged behavior of an 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.

[2]more precisely, *strictly causal.*

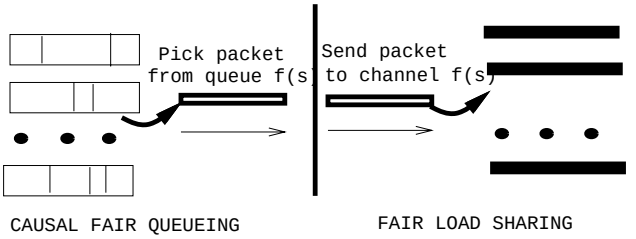

Fig. 4. Consider a backlogged execution of a fair-queuing algorithm. If the algorithm is causal we first apply a function $f(s)$ to select a queue. We transmit the packet p at the head of the selected queue and then update the state using a function $g(s, p)$. We can obtain a fair load-sharing algorithm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

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 function 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 CFQ Algorithms for Load Sharing

The transformation from fair queuing to fair load sharing is illustrated in Figure 4. We start on the left with a 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.

3.3 Evaluating the Transformation

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 number of 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 Transformation Theorem

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

THEOREM 3.1 *Consider a 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. 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 execu-

tion 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 in 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, since A is fair, we know that 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 1-1 correspondence 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. We describe two instances of CFQ algorithms.

3.5 Surplus Round Robin (SRR)

SRR is a specific example of a CFQ algorithm to which the transformation theorem can be applied. SRR is based on a modified version of DRR [Shreedhar and Varghese 1995]. SRR is also identical to an FQ algorithm proposed by Floyd and Van Jacobson [1995].

In the SRR algorithm, each queue i is assigned a quantum of service, measured in units of data, and is associated with a counter called the Deficit Counter (DC_i), which is initialized to 0. Queues are serviced in a round-robin manner. When queue i is picked for service, DC_i is incremented by the quantum for that queue. As long as DC_i is positive, packets are sent from that queue, and DC_i is decremented by the size of the transmitted packet. Once DC_i becomes nonpositive, the next queue in round-robin order is selected for service. Note that 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, and 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 nonpositive, packets are sent from the next queue.

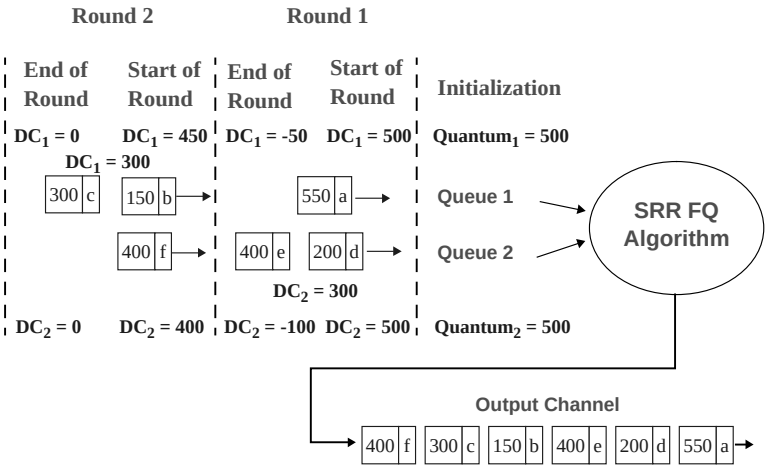

Fig. 5. Example of SRR Fair Queuing. Each queue has a quantum of 500 bytes.

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 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 $Quantum_i$. Let the maximum packet size be Max.

LEMMA 3.2 *For any backlogged execution of the SRR FQ algorithm, the following holds for each Channel i at the end of each round:* $-Max < DC_i$.

PROOF. By definition, at the end of each round, DC_i is nonpositive; hence it is bounded above by 0. Since only one packet can be transmitted to cause DC_i to transit from a positive value to a nonpositive value, and since no packet is larger than Max by definition, DC_i is strictly bounded below by $-Max$. □

THEOREM 3.3 (FAIRNESS OF SRR). *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, *and thus SRR is fair.*

PROOF. Let $DC_i(k)$ be the value of DC_i at the end of round k. Let $Bytes_i(k)$ be the bytes sent from Queue i in round k. Let $Send_i(k)$ be the total number of bytes sent from Queue i in rounds 1 to k. Thus $Send_i(K) = \Sigma_{k=1}^{K} Bytes_i(k)$.

By definition the value of DC_i at the end of round $k - 1$ is $DC_i(k - 1)$, and the value of DC_i at the beginning of round k is $DC_i(k - 1) + Quantum_i$. Assuming a backlogged execution, during round k, DC_i decrements by the number of bytes sent in that round which is $Bytes_i(k)$, finally resulting in a value of $DC_i(k)$ at the end of the round. Equating the two, we get the following:

$$Quantum_i + DC_i(k - 1) = Bytes_i(k) + DC_i(k)$$

This equation reduces to

$$Bytes_i(k) = Quantum_i + DC_i(k - 1) - DC_i(k).$$

Summing over K rounds, we get a telescoping series: $\Sigma_{k=1}^{K} Bytes_i(k) = K \cdot Quantum_i + DC_i(0) - DC_i(K)$. Since $Send_i(K) = \Sigma_{k=1}^{K} Bytes_i(k)$, and by definition, $DC_i(0) = 0$, we get $Send_i(K) = K \cdot Quantum_i - DC_i(K)$. This equation can be written as $|Send_i(K) - K \cdot Quantum_i| = |DC_i(K)|$.

By Lemma 3.2, $|DC_i(K)|$ is bounded by Max. □

3.6 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 nonpositive. 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 nonpositive, and then the next channel is selected, and so on.

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 3.4 *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.*

The SRR load-sharing scheme has a number of nice properties that makes it appropriate for use in a practical packet-striping algorithm. It

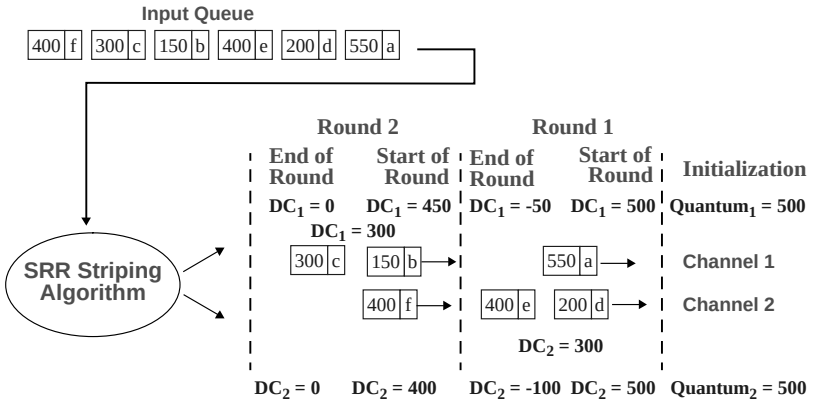

Fig. 6. Example of SRR Load Sharing. Each channel has a quantum of 500.

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.

3.7 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.[3]

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 index 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.

When all queues are backlogged, it is easy to show that 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 algorithm 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 causal.

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

[3]Randomized Fair Queuing is not related to Stochastic Fair Queuing [McKenney 1991], which refers to a class of algorithms that are probabilistic variants of the WFQ algorithm. RFQ is based on a suggestion made by Adam Costello, Washington University in St. Louis.

the channel, such that $prob(i) = B_i/\Sigma_{j=1}^{N} B_j$. 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 because of the single-step process for selecting the channel to serve. It might be useful in cases where the per-link DC state of the SRR algorithm is not needed.

4. FIFO Delivery Using Logical Reception

The previous section described how we could use ideas from fair queuing to implement load-sharing algorithms at the sender. (See Section 2.1 on Goals.) This section describes techniques for ensuring FIFO delivery at the receiver without adding sequence numbers to packets. 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 suppose 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, this is expressed in Theorem 4.1.

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, then 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. CFQ algorithms match this property by definition.

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 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 in Section 2.1) 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 mode 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 the receiver reinitializes the channel, when either the sender or the receiver goes down and comes up, 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., Multilink PPP) 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, while D is the value of the DC before the packet is sent. We assume that R and D are abstract numbers which can grow unboundedly. In real implementations, since R and D can only take on finite values before rolling over, they must be selected to be sufficiently larger than a burst loss to prevent misinterpretation of their current values. 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.

A simple example illustrating the idea behind synchronization recovery is illustrated in Figures 6 to 12. We consider the configuration shown in Figure 7. The SRR CFQ algorithm is used for resequencing at the receiver,

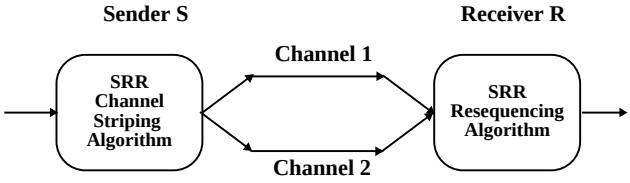

Fig. 7. Configuration to illustrate synchronization recovery.

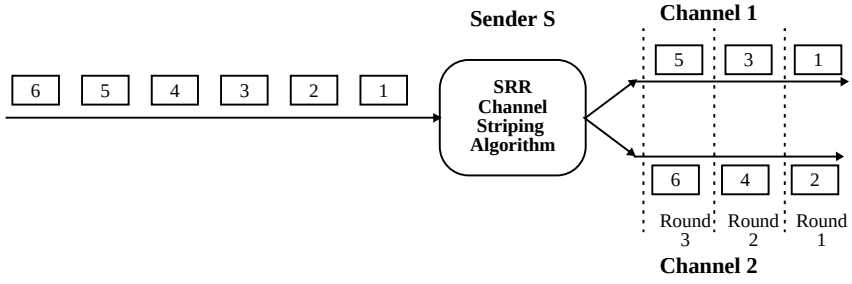

Fig. 8. Sender sends packets.

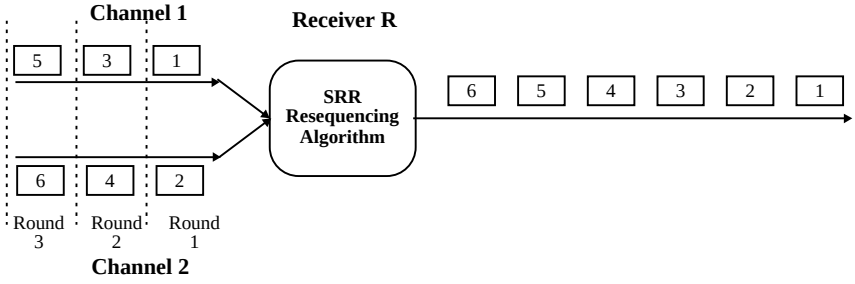

Fig. 9. Receiver in synchronization with the sender.

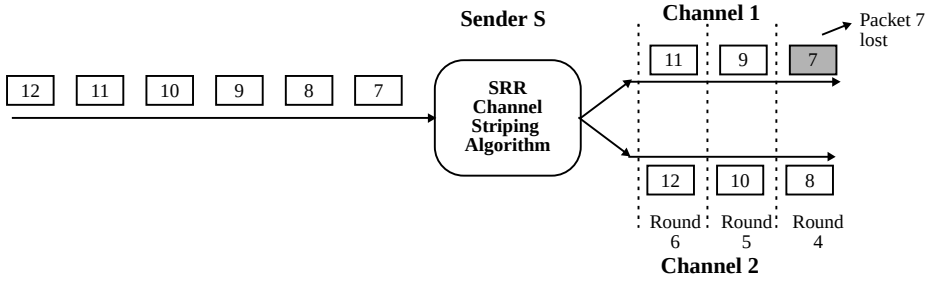

Fig. 10. A channel loses a packet.

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 (Round Robin).

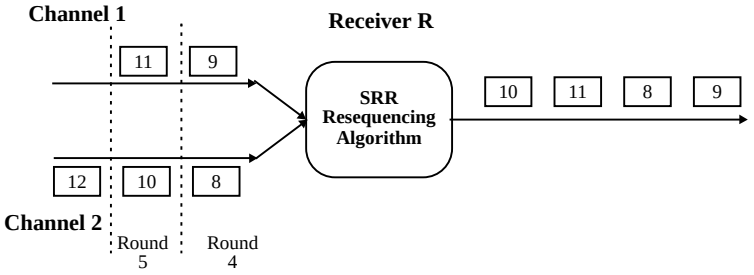

Fig. 11. Receiver out of synchronization.

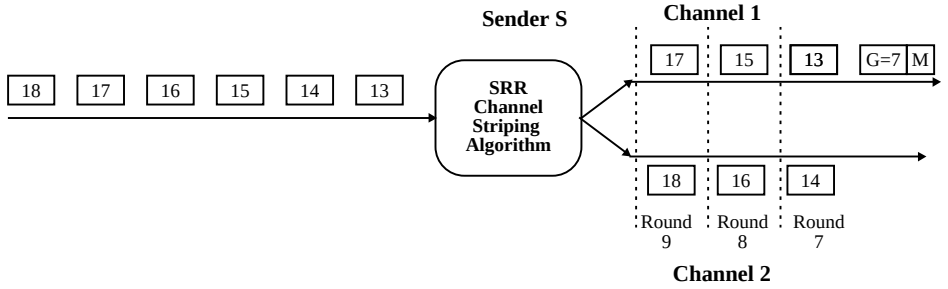

Fig. 12. Sender sends a marker packet.

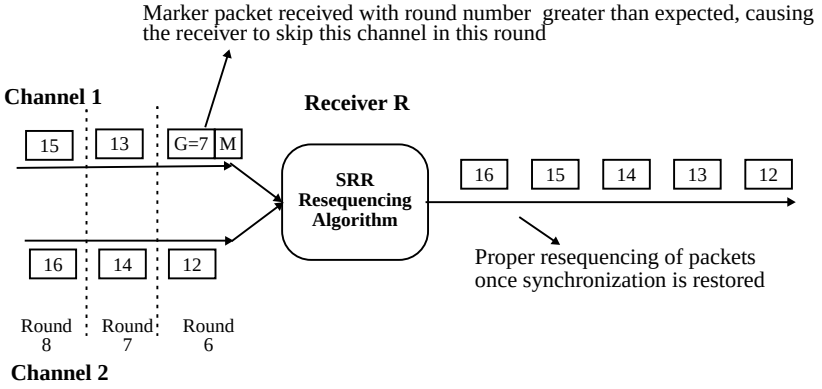

Fig. 13. Synchronization restored at receiver.

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 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. As a result, the receiver goes out of synchronization with the sender, and starts 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 difference in round number causes the receiver to skip this channel in the current round and proceed to the next channel. This skipping 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, 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 if both visit channels in increasing channel number order during a round-robin cycle. C2 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 and

—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 global round number 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 action 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 is greater than 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 skipping clearly maintains condition C1.

For each channel i, assume that $Quantum_i \geq Max$ (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 these observations, we can prove the following lemma (see the Appendix for the proof):

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 errors are either 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 [Chandy and Lamport 1985] and then doing a reset [Varghese 1993]. We deal with sender or receiver node crashes by doing a reset.

In summary, 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, which also

includes the relevant state (i.e., the DCs), we can synchronize each channel independently. The only global condition that needs to be enforced is condition C1, which is implemented easily by skipping channels that have lower round numbers than incoming marker packets. The pseudocode for the algorithms is described in the Appendix.

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 an architectural framework 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 striping 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.

6.1 An Architectural Framework 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 multiaccess interfaces like Ethernets and Token Rings. The framework is as shown in Figure 13. 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. Sending to the stripe layer is accomplished 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.

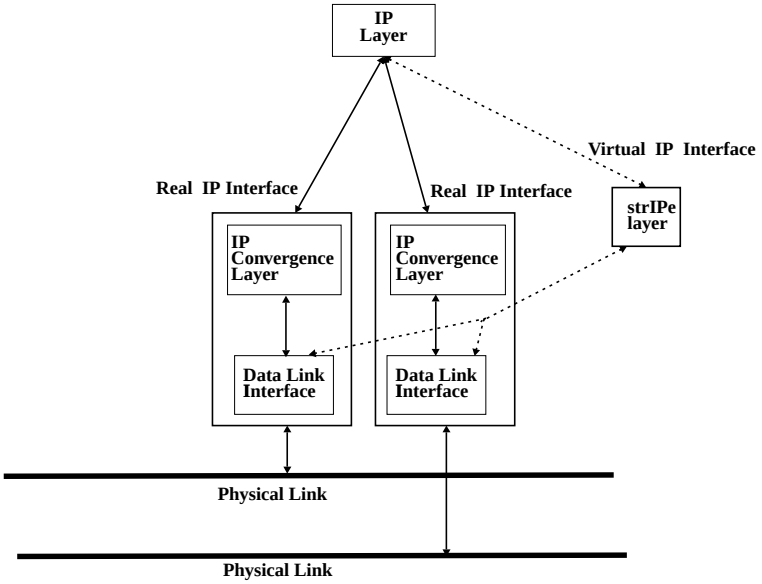

Fig. 14. The position of the stripe layer. The dotted lines indicate the data flow between IP and the data link layer via the stripe layer.

At the receiving end, the data link interfaces hand over striped packets to the stripe layer for resequencing. This handover 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. We note that 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 1.2/i386 kernel. Our setup consisted of two Pentium workstations, each with two IP interfaces. To show that our protocol can handle different links, we chose two completely different data link interfaces. One interface was a 10Mbps 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 14 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. We used Ethernet and ATM interfaces because we wanted to study the effects of striping across dissimilar interfaces.

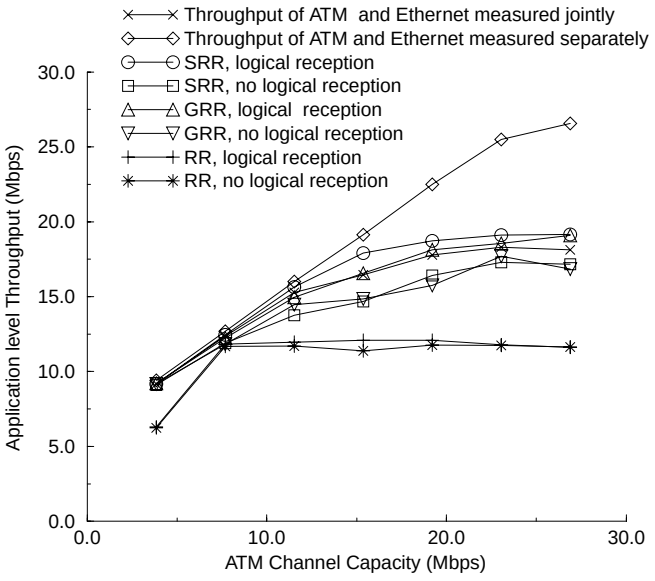

Fig. 15. Performance of SRR as the capacity of the ATM PVC is varied.

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. Note in this case that only one interface is used for sending data at a time. We then opened two TCP connections between the two workstations, one over the Ethernet link, and the other over the ATM link. Data were sent over both connections at the same time, and the throughput was measured. This measurement gave us the throughput of the ATM and Ethernet interfaces when packets are sent simultaneously across both interfaces: this measurement was done to estimate the overhead of servicing interrupts from two interfaces at a time, as opposed to the previous case. 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 14 we have a throughput upper bound, as well as four variants of `stripe` with one or more of its features disabled. Figure 14 plots the throughput of each of the variants as the raw bandwidth of the ATM VC is varied from 3.8Mbps to 26Mbps.

Table I presents the throughput of each scheme when the ATM VC has a capacity of 23.8Mbps. We first note, that, even in the absence of striping, the throughput while sending packets jointly over both interfaces (18.3Mbps) is less than the sum of the individual throughputs over each of

Table I. Throughput of Various Striping Algorithms as Measured at the Application Layer Using the `stripe` Framework. Striping is over an ATM VC and 10Mbps Ethernet. The ATM VC has a BW of 23.8Mbps.

Scheme	Throughput (Mbps)
Throughput of ATM and Ethernet measured jointly	18.3
Throughput of ATM and Ethernet measured separately	25.5
SRR, logical reception	19.1
SRR, no logical reception	17.3
GRR, logical reception	18.5
GRR, no logical reception	17.7
RR, logical reception	11.8
RR, no logical reception	11.7

the interfaces (25.5Mbps). This result is because the workstation has to service more interrupts when both interfaces are active. Consequently, there is a significant increase in the processing overhead, and corresponding decrease in the throughput. Note that the bottleneck is in the interrupt driver processing, as opposed to the striping overhead.

We note that the throughput of `stripe` (19.1Mbps) is about the same[4] as the throughput obtained by sending unstriped packets jointly over both the ATM and the Ethernet interfaces (18.3Mbps): this result clearly shows that the striping and resequencing algorithm introduces virtually no overhead. The throughput of `stripe` is consistently better than the variants that disable features. Its throughput is better than the variant that uses GRR (18.5Mbps), and better than the variant that disables logical reception (17.3Mbps). 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 in Figure 14 is due to the fact that, at those points, the rate of the ATM PVC is less than that of the Ethernet. Note that RR is commonly used in existing striping protocols, as discussed in Section 7.

There is a throughput gain using SRR over GRR, although the difference is not marked in Figure 15 (around 600kbps). This marginal gain 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 7.6Mbps, so that the ATM interface gave the same throughput as the Ethernet (6Mbps). Note that in this case GRR reduces to RR. Then packets were sent in deterministic fashion, with the bigger (1000 bytes)

[4]actually a little better, since the striping case uses only one TCP connection and hence has less computational overhead.

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 11.2Mbps. With GRR, the bigger packets are all sent on one interface, and the smaller packets on the other, so the throughput drops dramatically to 6.8Mbps.

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 140Mbps over an ATM interface using 8KB 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 Implementation Issues with CFQ Algorithms

We have experimented with our striping scheme both at the IP level using the `stripe` protocol as well as at the transport level through our simulations. We have not, however, exhaustively explored all the issues involved with implementing a complete striping solution in a real-world environment. For example, we have assumed an infinite buffer space at receivers and that channel conditions are fixed, i.e., the capacities do not fluctuate. We have also not analyzed the performance impact of channels with unequal skew. We briefly examine these issues in this section.

6.3.1 *Flow Control*. For channels not providing flow control, e.g., UDP channels, a simple credit-based flow control scheme proposed by Kung and Chapman [1993] proved very effective in eliminating packet loss due to channel congestion. In this scheme, the receiver periodically transmits its buffer availability on each link in control packets. The buffer availability is in the form of credits which the sender consumes for each packet it transmits. The periodic transmission of credits from the receiver allows the sender to maintain proper flow control even in the presence of packet loss. This scheme is particularly well suited to our striping scheme, since the credits can be piggybacked on the periodic marker packets for data traffic in the reverse direction from the receiver to the sender.

6.3.2 *Load Sharing over Shared Media Channels*. The amount of data sent through each channel by the channel-striping algorithm is proportional to the quantum associated with that channel, which in turn is assumed to be proportional to the rated bandwidth (BW) of the channel. If the actual BW of a channel were to vary below the rated BW, as often happens with shared media channels like Ethernets, 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 with CFQ-based striping, the sender blocks on this channel to transmit data.

Assume the quantum associated with each channel is proportional to its rated BW. If the rated BW of channel i is $B_{rated,\,i}$, and the actual BW of Channel i 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 $(B_{act,\,j}/B_{rated,\,j})$ $\Sigma_{i=1}^{N} B_{rated,\,i}$, which is less than $\Sigma_{i=1}^{N} B_{act,\,i}$.

The above analysis indicates that it is important to assign a value to the quantum associated with each channel close to the actual BW of the channel. If the channel has a constantly varying instantaneous BW, e.g., a shared Ethernet segment, then the load sharing will not be very effective, depending on the mismatch between the actual BW and the rated BW.

6.3.3 *Buffering and Delay 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, as required by logical reception. This blocking increases the delay of all delivered packets.

If there are N channels, all operating at the same rate, and one channel has a latency T seconds greater than the other channels, corresponding 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 that 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, . . . , $N - 1$ be T_1, T_2, . . . , T_{N-1}, respectively. Let B_i is the bandwidth of channel i. Then the total amount of buffering required is $\Sigma_{i=1}^{N-1} B_i T_i$, with $B_i T_i$ amount of buffering required at channel i.

All incoming packets are delayed T_{N-1} seconds 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 applications which need channel striping tend to be throughput intensive, rather than delay sensitive.

6.3.4 *FIFO Channels*. The overall framework for packet striping presented in Figure 1 assumes FIFO channels. However, this assumption of FIFO channels is needed only if the logical reception and synchronization

recovery algorithms described earlier are used at the receiver. In cases where no such algorithm is used at the receiver, the load-sharing algorithms can still be used at the sender without the necessity of FIFO channels. For example, it is possible to do fair load sharing over arbitrary channels when our load-sharing algorithms are used in conjunction with Multilink PPP framing, as in Section 7.2.

6.3.5 *Work Conservation*. Another aspect of our load-sharing algorithms which needs to be noted is that while these algorithms provide fair load sharing in the form of proportionate loading of channels, they are not work conserving. For example, on startup the SRR striping algorithm will transmit all incoming packets on one channel untill the channel's assigned quantum is exceeded, and will then proceed transmitting from the next channel. It is therefore possible for multiple packets to be sent over one link while other links are not used. In practice, this delay can be greatly reduced by keeping per-link transmit buffers, as shown in Figure 1. This allows the striping algorithm to quickly move from one channel queue to another, unconstrainted by the channel transmission rate. Unfortunately, it does not appear to be possible to devise a packet-striping scheme which provides FIFO delivery of packets while being work conserving [Bennett et al. 1999].

7. RELATED WORK

Traw and Smith [1995] describe a model of load striping, and summarize various approaches to the problem. We discuss some solutions at the physical, data link, and network layer, and describe a commercial implementation of our work.

7.1 Existing Implementations

7.1.1 *Bonding*. Inverse multiplexers which operate on 56kbps and 64kbps circuit-switched channels are commercially available. Industrywide standardization of inverse multiplexers has been initiated by the BONDING [Duncanson 1994; Fredette 1994] consortium, which has issued standards for a frame structure and procedures for establishing a wideband communications channel by combining multiple switched 56kbps and 64kbps 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. Since BONDING is exclusively designed to provide an interoperable standard for the inverse multiplexing of synchronous digital access lines, it cannot be used in other scenarios, for example, analog access lines, or multiaccess networks like LANs. Furthermore, since the implementation of delay compensation and aggregation is typically implemented in hardware, there is a fixed limit to the amount of skew compensation that an inverse multiplexer can perform.

7.1.2 *Inverse Multiplexing over ATM IMA*. The ATM Forum is considering a standard for ATM cell striping called IMA [ATM Forum Technical Committee 1999] based on round-robin striping of ATM cells over multiple point-to-point links. As with BONDING, IMA works with special hardware at either end of the ATM links and only when the skew can be bounded tightly. For example, the current standard works with a maximum skew of 25 milliseconds. Furthermore, the current version of IMA (V1.0) works only with links having the same nominal link cell rate (LCR).

7.1.3 *Gigabit Testbed Network Adaptors*. 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 [Theoharakis and Guerin 1993] does striping over four STS-3c channels. The Bellcore HAS adaptor[5] stripes HIPPI packets over SONET lines using a first-come-first-serve (FCFS) striping policy, while the CASA gigabit testbed uses round-robin striping at the byte level. The OSIRIS Adaptor [Druschel and Peterson 1994] 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 they rely on extra information for resynchronization (e.g., information embedded in SONET or ATM headers). Thus none of these schemes meet all our goals.

7.1.4 *Upper-Level Striping*. Existing striping schemes which operate at higher levels usually sacrifice either fair load sharing or FIFO delivery. For example, the Random Selection scheme offered by one router vendor 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 used by a router vendor relies on hashing packet addresses to channels to route packets destined for the same address over the same channel. This scheme provides FIFO delivery of packets destined for the same address, but does not provide load sharing for packets addressed to any given destination.

7.1.5 *Multilink PPP*. The Internet standard RFC1990 specifies MP (Multilink PPP). MP provides a framework and packet formats for striping across multiple PPP links by attaching a MP header to each packet. The MP header contains a sequence number which allows the receiving endpoint to restore FIFO order. In addition the MP header also describes a fragmentation protocol which allows packets to be transmitted as multiple fragments over multiple links and reassembled at the other end. Thus, one

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

way of reducing the latency of a big packet is to split it into multiple fragments and to transmit the fragments over separate links.

The base MP's monotonically increasing sequence numbering does not allow the suspension of the sending of a sequence of fragments of one packet in order to send another packet. It is, however, possible to send intervening packets with only PPP headers without MP headers. This ability allows MP to support two levels of priority.

The ISSLL WG has defined a Multi-Class Extension to Multilink PPP [Borman 1999] to provide multiple levels of priority. In this extension, MP is extended by adding a class number in addition to the sequence number to the MP header. This extension allows fragments of different classes to be interleaved, thereby allowing transmission of fragments of packets of one level to be superseded by transmission of packet fragments from another level. In addition, real-time frame formats have been defined to allow the transmission of a frame to be suspended in order to transmit a higher-priority frame.

Our `stripe` protocol described in Section 6.1 differs from MP 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 data packet. Not adding new headers is essential for striping over high-speed interfaces. Finally, MP 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 the gigabit testbeds 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. In 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.

7.2 Commercial Implementation of SRR

We now describe the application of our work (SRR) in a Multilink PPP striping solution[6] provided by a commercial router vendor (Cisco). In this scheme, the router maintains two separate queues, one for real-time packets and another for non-real-time packets. Non-real-time packets are sent over the Multilink bundle using Multilink PPP encapsulation and SRR striping algorithm. The real-time packets are interleaved using PPP encapsulation. The receiver uses the Multilink PPP sequence numbers to resequence the non-real-time packets. The scheme is shown in Figure 15.

To bound the queuing delay experienced by real-time packets, the user specifies a configurable parameter called the *fragment delay* which is set to

[6]Personal communication, January 1998.

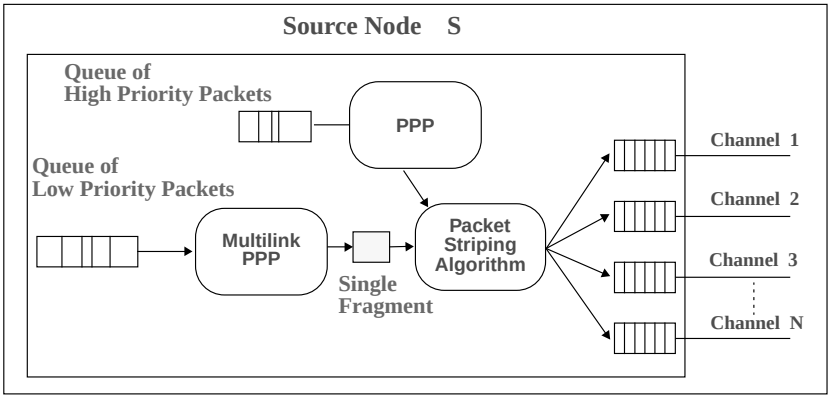

Fig. 16. Commercial implementation of SRR striping algorithm.

a default value of 30 microseconds. This delay is translated into the quantum for each link by the formula:

Quantum of individual link (in bytes) = Bandwidth of link (in bytes) *

fragment delay (in seconds)

Recall that Multilink PPP provides both sequence numbering as well as a link-level fragmentation format. The minimum quantum across all the links is used to set the maximum fragment size of non-real-time packets. The non-real-time packets are fragmented to this minimum size, and the fragments are sent using SRR across the links in the bundle. As can be seen, if the queue length at each link is kept to within the quantum, the latency for a real-time packet is bounded to within the fragment delay.

8. CONCLUSION

This article describes a family of efficient channel-striping algorithms that solve both the variable-packet-size problem and the FIFO delivery problem 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. In order for receiver simulation to work, it is important to note that it is only necessary that the sender algorithm be causal, which is guaranteed by our fair load-sharing schemes. While the SRR scheme is very similar to the DRR fair-queuing scheme [Shreedhar and Varghese 1995], our major contribution is to show a general connection between fair queuing and load sharing, and to introduce

the idea of logical reception which is not present in Shreedhar and Varghese [1995]. SRR striping has been implemented in router vendor Cisco's Internetworking Operating System (IOS) 11.3 as part of Cisco's Multilink PPP implementation.

A major contribution of our article is a loss recovery protocol. 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. In particular, we feel that *Channel Bonding* is a very promising application. Channel bonding refers to using multiple phone lines to communicate from a home PC to an ISP. Because of the low bandwidths of the individual phone lines, it is very important to utilize the bandwidth efficiently. 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 network; however, these boundaries are needed in order to implement early discard policies [Romanov and Floyd 1994].

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

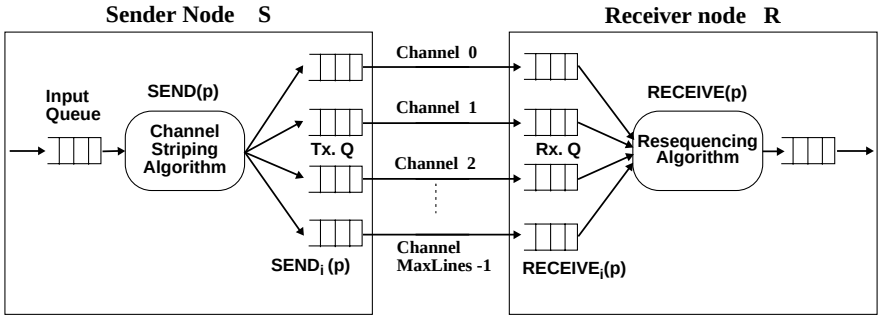

Fig. 17. Reference configuration for the striping and resequencing algorithms.

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.

APPENDIX

A. SENDER AND RECEIVER PSEUDOCODE

The reference configuration is as shown in Figure 17. At the sender, we have a single input queue, from which packets are taken in FIFO order, and striped across the different channels by being placed in the individual channel transmit queues. This is the task of the **send** function. The **send**$_i$ function is used to transmit packets from each of the channel queues. Packets sent on each channel include both regular data packets and periodic marker packets.

The following variables are maintained at the sender. There is a global variable sp (sender pointer) which is initialized to 0. This tracks the transmit queue from which the sender is currently sending packets. The currently active channel at the sender is defined as the channel pointed to by the sender pointer.

There are $MaxLines$ channels, numbered from 0 to $MaxLines - 1$. At the sender s for each channel i, there are two corresponding local variables, the deficit counter DC_i^s and the current round number R_i^s for channel i. Initially, $DC_i^s = q_i$ for all i, where q_i is the quantum associated with channel i.

R_i^s keeps track of the round number of the next packet sent on channel i. Initially, $R_i^s = 1$ for all i, $0 \leq i \leq MaxLines - 1$.

We assume that the quantum is large enough to transmit a maximum-size packet; formally, $q_i \geq MaxPacket$ for all i. We assume that any data packet p belongs to a data packet alphabet P; formally, $p \in$ data packet alphabet P. We define a message m to be either a data packet or a marker; formally $m \in P \cup$ Marker.

The code for the **Send** and the **Send**$_i$ function follows:

MACRO: **AdvanceSenderPointer** /* macro used by main events below*/
  $R^s_{ps} = R^s_{ps} + 1$
  $sp = (sp + 1)$**mod** $Maxlines$
Send *(p)*
Precondition: packet p is at head of the input queue
Effect:
  $Addq(p, sp)$ /* remove packet p from channel queue */
  $DC^s_{sp} = DC^s_{sp} - Size(p)$ /* adjust deficit counter */
  If $DC^s_{sp} \leq 0$ /* if nonpositive */
    $DC^s_{sp} = DC^s_{sp} + q_i$ /* add a quantum for next round */
    AdvanceSenderPointer /*advance round numbers and sender pointer */
SendMarker *(i)* /* send periodic marker on channel i */
Effect:
  Let $M = (Marker, R^s_i, DC^s_i)$ /* The Marker packet has two fields: the first field */
/* carries the sender round number and the second the deficit counter in channel i*/
$Addq(M, i)$/* Add marker to end of channel i queue */
Send$_i$*(m)*
Precondition: message m is at the head of the transmit queue
Effect: *Transmit (m,i)* /* Transmit m on physical channel i */

At the receiver, there are multiple input queues, one per channel, on which packets are received and stored by function **Receive**$_i$. The resequencing algorithm resequences packets from these input queues and places them in the single output queue. This task is done by function **Receive**$_i$, which is called when a data packet is received on the currently active channel, and function **ProcessMarker**, which is called when a maker is received on the currently active channel. The currently active channel at the receiver is defined as the channel pointed to by the receiver pointer.

The following variables are used at the receiver. Analogous to sp at the sender is the receiver pointer rp, which is initially 0 and points to the receiver channel queue that the receiver is currently removing packets from. Again, analogous to the sender, for each channel i, the receiver maintains two variables. The first is a deficit counter DC^r_i (which is initially equal to q_i for all i, $0 \leq i \leq Maxlines - 1$). There is also the round number variable R^r_i initially equal to i for all i, $0 \leq i \leq Maxlines - 1$.

In addition, there is a third variable per channel (that has no correspondence at the sender end). This is the saved round number variable R^{save}_i which is the round number sent in the last marker received on channel i. This variable is initialized to 1 for all i, $0 \leq i \leq Maxlines - 1$. With these variables, the receiver code is as follows:

MACRO: **AdvanceReceiverPointer** /* Macro to increment receiver pointer */

$R^r_{rp} = R^r_{rp} + 1$ /* increment round number of channel rp

$rp = (rp + 1)$ **mod** $MaxLines$ /* increment rp */

Receive$_i(m_{m \in P \cup Marker})$

/* Physically receive a data packet or a marker on channel i */

$Addq(p, i)$ /* buffer the packet in channel i queue for later logical reception */

Receive *(p)*

/* Delivers packet p to receiving application */

Precondition: Data packet at head of queue rp and $R^{save}_{rp} \leq R^r_{rp}$

Effect:

$Removeq(p, rp)$ /* remove packet p from channel queue */

$DC^r_{rp} = DC^r_{rp} - Size(p)$ /* adjust deficit counter */

If $DC^r_{rp} \leq 0$ /* if deficit counter is nonpositive */

$DC^r_{rp} = DC^r_{rp} + q_i$ /* add a quantum for next round */

AdvanceReceiverPointer /* increment receiver pointer */

Processmarker *(r, c)* /* process arrival of marker */

Precondition: (*Marker*, *r, c)* at head of queue rp

Effect:

$DC^r_{rp} = c$ /* put deficit counter in marker in channel deficit counter */

$R^{save}_{rp} = r$ /* save round number in marker in the saved round number variable */

if $R^{save}_{rp} > R^r_{rp}$ then **AdvanceReceiverPointer**

/* skip to next line if saved round number is larger */

Skipline

Precondition: Data packet at head of queue rp and $R^r_{rp} < R^{save}_{rp}$

Effect: **AdvanceReceiverPointer**

/* skip to next line if saved round number is larger */

B. PROOF OF THE MARKER RECOVERY THEOREM

Each channel can be modeled by a simple state machine that models a FIFO queue which can potentially lose messages (markers or packets) and is initially empty. Consider the composite system consisting of the sender protocol state machine, receiver state machine, and $MaxLines$ FIFO channel state machines. We will prove properties of the system. Formal definitions of composition and executions of such state machines can be found in Lynch and Tuttle [1989].

Consider any execution E of the system. With respect to E, we make the following definitions. While packets do not carry any sequence numbers in our protocol, we will add the following fictitious numbers to each packet that is convenient for the proof. Define $T(P)$, the transmit number of a packet P sent in E to be the three-tuple (R, i, D), where R is the round number in which the packet P is transmitted, i is the link number, and D is the value of DC^s_i just before the packet is transmitted.

Similarly, define $R(P)$, the receive number of a packet received in E to be the three-tuple (R, i, D), where R is the round number in which the packet

is received, i is the link number, and D is the value of DC_i^r just before the packet is logically received on link i, i.e., just before P is handed to the application.

Comparisons between send numbers as well as receive numbers are done lexicographically. Formally, we say that $(R, i, D) < (R', i', D')$ if $R < R'$, or $(R = R') \wedge (i < i')$, or $(R = R') \wedge (i = i') \wedge (D' > D)$. It is easy to verify this.

Fact 1. In E, if P_1 is transmitted before P_2, $T(P_1) < T(P_2)$. This follows intuitively from the code because the sender sends packets in earlier rounds before later rounds; within a round the sender sends packets with smaller channel numbers first; finally, within a link and a given round, packets with larger deficit counters are sent earlier than others with smaller deficit counters.

Similarly, we have Fact 2:

Fact 2. In E, if $R(P_1) < R(P_2)$, then P_1 is received before P_2. This follows intuitively from the code because the receiver logically receives packets in earlier rounds before later rounds; within a round the receiver receives packets from smaller channel numbers first; finally, within a line and a given round, packets with larger deficit counters are logically received earlier than others with smaller deficit counters.

From these two simple facts, it is easy to see that FIFO order will be preserved if the transmit number of a packet $T(P)$ is the same as its receive number $R(P)$. We will show that this is indeed true in correct operation (no loss), and will become true after all loss stops and after a marker is sent on every channel.

Let $TransitBytes_i$ be a variable which counts the bytes in transit to the receiver on channel i, which includes (1) the bytes buffered at the sender in sender channel queue i, (2) the bytes in transit on the physical channel i, and (3) the bytes buffered at the receiver in receiver queue i. For any message M in transit on channel i, define $Transit(M)$ to be the bytes in transit on channel i that are ahead of message M.

We define channel i to be *correct* at any instant if the following three conditions all hold:

—$R_i^s \cdot q_i - DC_i^s = TransitBytes_i + R_i^r \cdot q_i - DC_i^r$.

—If there is a message M (either a marker $M = (Marker, R, D)$ or a packet with $T(M) = (R, i, D)$) in transit on channel i) then $R \cdot q_i - D = Transit(M) + R_i^r \cdot q_i - DC_i^r$.

—$R_i^{save} \leq R_i^r$.

Intuitively, the first equation is just a conservation-of-bytes equation that must be true if the channel is lossless, and must be made true by markers after a lossy period. The left-hand side represents the amount

transmitted by the sender on channel i (the current round number times the quantum minus the current value of the deficit counter at sender). The first term on the right-hand side represents the bytes in transit. The second term on the right-hand side represents the amount received by the receiver on channel i (the current round number times the quantum minus the current value of the defict counter at receiver),

Intuitively, the second equation is a similar "conservation of bytes" for the bytes ahead of any message in transit but using the round number and deficit counter at the point the message was sent as opposed to the corresponding sender variables. Intuitively, the third equation just ensures that the **Skipline** function will not be executed in normal operation.

We define a *good period* of execution E to be a period over which all channels are initially correct, and over which no packets are lost on any link. For simplicity, we will only consider executions E that start with a period of arbitrary loss and then all loss stops (i.e., E ends with a good period). We will see that FIFO delivery is restored after some time T after the end of a loss period. In actual practice, of course, good and bad periods can alternate; as long as the good periods are larger than T, FIFO delivery will be restored before the end of the good period. The larger the good periods are when compared to T, the better will be the performance of the protocol. However, we prefer to keep our model simple and assume only one good period, realizing that our results can be simply extrapolated to the more realistic model.

LEMMA B.1. *During a good period, all channels are correct.*

PROOF. The proof is by induction on the sequence of events in the execution E. We know by definition that all channels are correct at the start of a good period. We show by induction that, assuming no packet loss, all other actions of the protocol will preserve the correctness of a channel. Consider the first of the three correctness conditions first. The **Send** action will increase $\mathbf{TransitBytes}_i$ but will decrease DC_i^s by the same amount to compensate. Also, at the end of a round for channel i at the sender, DC_i^s increases by q_i, but R_i^s increases by 1 to compensate. The **Receive** action can similarly decrease $\mathbf{TransitBytes}_i$ but will decrease DC_i^r by the same amount to compensate. Again, at the end of a round for channel i at the receiver, DC_i^r increases by q_i, but R_i^r increases by 1 to compensate.

There are only two other actions to be considered. The receiving of a marker can cause the deficit counter and saved round number to change, and the **Skipline** function can cause the receiver round number for channel i to change without changing the other variables. So consider a marker $M = (Marker, R, D)$ sent during a good period on channel i. When M reaches the head of the channel i queue at the receiver, by the second correctness condition since $Transit(M) = 0$, the receiver round number and deficit counters are equal to R and D respectively. Thus, processing a marker in a good period does not affect any of the variables. Similarly, the

third correctness condition and the code assure us that during a good period the **Skipline** function will never be executed.

The inductive proof of the second correctness condition is very similar to the proof of the first condition except that there are fewer cases, since the sender actions do not affect the message M in transit. The proof of the third condition follows from the fact that the R_i^{save} only changes when a marker is processed. We have just argued that processing a marker in a good period can only cause $R_i^{save} = R_i^r$ at the instant the marker is received. Subsequent events leave R_i^{save} unchanged until the arrival of the next marker, but can increase R_i^r, which leaves the third condition true. □

LEMMA B.2. *In a good period of E, for all packets P that are sent and received,* $R(P) = T(P)$.

PROOF. Let P be some packet transmitted (and received) on say channel i during this period. Let the values of the deficit counter and round number at i just before P was transmitted be D and R respectively. Then, $T(P) = (R, i, D)$ by definition. By Lemma B.1, channel i is correct when P is transmitted and is correct when P is received. By the second correctness condition, when P is received (which we assume is in a good period), $Transit(P) = 0$, and therefore the receiver round number and deficit counter are equal to R and D respectively. Since i remains the same as well, $R(P) = T(P)$ for all packets P sent and received in a good period. □

THEOREM B.3. *In a good period, FIFO delivery is maintained.*

PROOF. If P_1 and P_2 are sent and both are received and P_1 is sent before P_2, then we wish to show that P_1 is logically received before P_2. From Fact 1, $T(P_1) < T(P_2)$. From Lemma 4, $R(P_1) = T(P_1) \wedge R(P_2) = T(P_2)$. Therefore $R(P_1) < R(P_2)$. Therefore, by Fact 2, P_1 is logically received before P_2. □

Thus if all links are initially correct and the links remain lossless, FIFO behavior is preserved (which is intuitively clear). All we have left is to show, after all loss stops, that in bounded time after loss stops, a good period (and hence FIFO reception) resumes. We define a marker to have been *processed* on a link i after the marker has been been received and removed from receiver queue i and $R_i^{save} \leq R_i^r$. Thus, if a marker reception causes $R_i^{save} > R_i^r$, we do not say that the marker has been processed until channel i has been skipped until $R_i^{save} = R_i^r$.

LEMMA B.4. *After arbitrary loss in execution E stops, a good period starts after a marker has been received and processed on all links.*

PROOF. Let M be the first marker transmitted (and received) on, say, channel i after loss stops. Let the values of the deficit counter and round

number at i just before M was transmitted be D and R respectively. Then $M = (Marker, R, D)$ by the code. By the time M is received, the sender may have sent *TransitBytes* more bytes which are now in transit behind M. It is easy to verify from the code that sending *TransitBytes* causes the receiver round number and deficit counter variables to change from R to R' and from D to D' such that $R' \cdot q_i - D' = TransitBytes + R \cdot q_i - D$. This establishes the first correctness condition. A very similar argument establishes the second correctness condition.

For the third correctness condition, we start by observing that loss can only lead to the round number at the receiver being less than the correct value if no loss had occurred. More precisely, even during loss, it is always true that $R \cdot q_i - D \geq TransitBytes + R' \cdot q_i - D'$, where R, D and R', D' are the variables at sender and receiver respectively. (The proof is identical to that of Lemma B.1 except that packet loss can lead to decreasing *TransitBytes* without affecting other variables.) Thus, when a marker is received and acted upon, it can only result in $R_i^{save} > R_i^r$. Thus, when a marker is actually removed from the queue, the third correctness condition may not be true. However, it will be true after the marker is processed (i.e., after the **Skipline** function has been executed a sufficient number of times and channel i is skipped over until the third condition becomes true).

To prove the last statement carefully, we have to also show that the receiver protocol cannot deadlock and prevent a marker from being processed. Suppose $R_i^{save} > R_i^r$ and the receiver pointer rp is stuck on line j. Then, either a sufficient number of data packets will arrive on line j (which will cause rp to advance), or a marker will arrive at the head of the channel j queue with a round number value at least equal to R_j^r) (which will cause the **Skipline** function to be executed on line j). □

THEOREM B.5. *After arbitrary loss in execution E stops, FIFO delivery will resume in bounded time.*

PROOF. By Lemma B.4, after arbitrary loss in E stops, all links will be correct after a marker is received and processed on all links. When this happens, a good period starts. Thus by Theorem 5 FIFO delivery resumes. The time for this to happen is the worst-case period between marker transmissions plus the time for a marker to be transmitted between the sender and receiver. □

REFERENCES

BENNETT, J. C. R., PARTRIDGE, C., AND SHECTMAN, N. 1999. Packet reordering in not pathological behavior. *IEEE/ACM Trans. Netw. 7*, 6 (Dec.), 789–798.

BORMAN, C. 1999. The multi-class extension to Multi-Link PPP. RFC 2686.

CHANDY, K. M. AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb.), 63–75.

DEMERS, A., KESHAV, S., AND SHENKER, S. 1989. Analysis and simulation of a fair queuing algorithm. In *Proceedings of the ACM Conference on Communications Architectures and*

Protocols (SIGCOMM '89, Austin, TX, Sept. 19–22), L. H. Landweber, Ed. ACM Press, New York, NY, 3–12.

DRUSCHEL, P., PETERSON, L. L., AND DAVIE, B. S. 1994. Experiences with a high-speed network adaptor: A software perspective. *SIGCOMM Comput. Commun. Rev. 24*, 4 (Oct.), 2–13.

DUNCANSON, J. 1994. Inverse multiplexing. *IEEE Commun. Mag. 32*, 4 (Apr.), 34–41.

FLOYD, S. AND JACOBSON, V. 1995. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Netw. 3*, 4 (Aug.), 365–386.

FREDETTE, P. H. 1994. The past, present and future of inverse multiplexing. *IEEE Commun. Mag. 32*, 4 (Apr.), 41–47.

KUNG, H. T. AND CHAPMAN, A. 1993. The FCVC (Flow Controlled Virtual Channel) proposal for ATM networks. In *Proceedings of the International Conference on Network Protocols* 116–127.

MCKENNEY, P. E. 1991. Stochastic fair queueing. *Internetworking Res. Exp. 2*, 113–131.

LYNCH, N. A. AND TUTTLE, M. R. 1989. An introduction to input/output automata. *CWI Q. 2*, 3, 219–246.

ROMANOV, A. AND FLOYD, S. 1994. Dynamics of TCP traffic over ATM networks. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM '94). ACM, New York, NY, 79–88.

SHREEDHAR, M. AND VARGHESE, G. 1995. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev. 25*, 4 (Oct.), 231–242.

THE ATM FORUM TECHNICAL COMMITTEE. 1999. Inverse multiplexing for ATM (IMA) specification. Version 1.1. ATM Forum Doc. AF-PHY-0086.001.

THEOHARAKIS, V. AND GUERIN, R. 1993. SONET OC12 interface for variable length packets. In *Proceedings of the 2nd International Conference on Computer Communications and Networks* (June). 28–30.

TRAW, C. B. S. AND SMITH, J. 1995. Striping within the network subsystem. *IEEE Network 9*, 4 (Jan.), 22–29.

VARGHESE, G. 1993. Self-stabilization by local checking and correction. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

RICHARD, W. D. AND FIGERHUT, J. A. 1998. The gigabit switch (WUGS-20) link interface specification. Tech. Rep. ARL-94-17. Computer Science Department, Washington University, St. Lous, MO.

Received: March 1999; revised: October 1999; accepted: October 1999