Report Number: WUCS-94-16

1994-01-01

Trading Packet Headers for Packet Processing

George Varghese

In high speed networks, packet processing is relatively expensive while bandwidth is cheap. This begs the question: what fields can be added to packets to make packet processing easier? By exploring this question, we device a number of novel mechanisms to speed up packet processing. With the advent of new standards for hte Data Link, Network, and Transport lyaers, we believe there is an opportunity to apply these techniques to improve the performance of real protocols. First, we suggest adding a data manipulation header to an easily accessible portion of each packet. This header contains pointers to fields (in... **Read complete abstract on page 2.**

Trading Packet Headers for Packet Processing

George Varghese

Complete Abstract:

In high speed networks, packet processing is relatively expensive while bandwidth is cheap. This begs the question: what fields can be added to packets to make packet processing easier? By exploring this question, we device a number of novel mechanisms to speed up packet processing. With the advent of new standards for hte Data Link, Network, and Transport lyaers, we believe there is an opportunity to apply these techniques to improve the performance of real protocols. First, we suggest adding a data manipulation header to an easily accessible portion of each packet. This header contains pointers to fields (in various layers) required for data manipulation. This information allows implementatinos to efficiently combine data manipulation steps (e.g., encryption and copying) in a structured fashion. Second, we suggest adding index fields to protocol identifiers at all layers (e.g., connection identifiers, network addresses, DSAPs) to reduce lookup costs and generic protocol processing. Several new ideas to utilize these index fields (threaded indexing, index passing, and source hashing) are proposed. Virtual Circuit Idenfifiers (VCIs) have been long used to simplify lookup and packet processing in virtual circuit. In source hashing and threaded indexing, the added indices essentiallly serve as VCIs, but for flows in a datagram network. In source hashing, for example, the "VCI" is a consistent random label chosen by the source. Our new methods provide the benefits of normal VCIs withut requiring a round trip delay for set up. Our methods can lower worst case datagram lookup times from O(log(n)) to O(1), which may be important for Gigabit routers.

Trading Packet Headers for Packet Processing

George Varghese

WUCS-94-16

July 1994

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

Trading Packet Headers for Packet Processing

George Varghese *

June 29, 1994

Abstract

In high speed networks, packet processing is relatively expensive while bandwidth is cheap. This begs the question: what fields can be added to packets to make packet processing easier? By exploring this question, we devise a number of novel mechanisms to speed up packet processing. With the advent of new standards for the Data Link, Network, and Transport layers, we believe there is an opportunity to apply these techniques to improve the performance of real protocols.

First, we suggest adding a data manipulation header to an easily accessible portion of each packet. This header contains pointers to fields (in various layers) required for data manipulation. This information allows implementations to efficiently combine data manipulation steps (e.g., encryption and copying) in a *structured* fashion.

Second, we suggest adding index fields to protocol identifiers at all layers (e.g.,, connection identifiers, network addresses, DSAPs) to reduce lookup costs and generic protocol processing. Several *new* ideas to utilize these index fields (threaded indexing, index passing, and source hashing) are proposed. Virtual Circuit Identifiers (VCIs) have been long used to simplify lookup and packet processing in virtual circuit. In source hashing and threaded indexing, the added indices essentially serve as VCIs, *but for flows in a datagram network.* In source hashing, for example, the "VCI" is a consistent random label chosen by the source. Our new methods provide the benefits of normal VCIs without requiring a round trip delay for set up. Our methods can lower worst case datagram lookup times from $O(\log(n))$ to $O(1)$, which may be important for Gigabit routers.

1 Introduction

Networks are getting faster because of advances in transmission and switching. Optical fibers, for instance, allow raw bandwidths of up to a Terabit/sec while Gigabit switching systems are

*Washington University in St. Louis.

being prototyped [Par93]. CPUs are certainly getting faster, but the rate at which CPUs are getting faster does not match the bandwidth increase rate. CPU processing is also limited by other factors such as caches, memories, busses, and disks. Thus packet processing appears to be the bottleneck while network bandwidth is inexpensive. This begs the question: *what fields can we add to packet headers to make packet processing faster?* This is the topic of our paper.

We propose several new and *general* techniques that can be used to reduce key packet processing bottlenecks, especially data manipulation and lookup overhead. Our techniques *require* the addition of fields to packet headers. Our techniques seem to apply to various types of protocols and at all layers of the protocol hierarchy. They can be added to existing protocols without major changes. We emphasize that this paper is not about new protocols but about new techniques that can be profitably added to protocols, both new and old.

Clark *et al* [CJRS89] argue that clever implementation of existing protocol suites (e.g., TCP/IP) can provide high throughput at the transport layer, using reasonably cheap CPUs and large packet sizes. So why not stay with existing protocols (and their headers) but focus on clever implementation? There are three reasons to continue to look for new techniques. First, the measurements of [CJRS89] were at the transport level and did not consider application throughput. Second, as we scale to Gigabit/sec and Terabit/sec, it appears that their techniques, while valuable, will not be enough. Third, significantly reduced packet processing overhead is always useful: it allows the CPU to do other things and reduces the packet size required for Gigabit throughput.

With the *need* for efficient packet processing in Gigabit networks, there is also an *opportunity.* Protocol standards are in transition[Par93] to accommodate high speeds and new services. *Data Link* standards are being affected by emerging ATM specifications; in the near future, wavelength division multiplexing may prove successful. Efforts are underway to modify the Internet *routing* protocols to increase addressing capability and to allow new services (e.g., congestion control, delay guarantees, support for mobile hosts). Finally, new multimedia applications may require multicast transmission which in turn may require multipoint *transport* protocols. Since protocols (and their headers) are changing or being considered for change, this is a good time to reexamine the use of additional fields.

Three other networking trends are relevant:

- **Networks are becoming bigger:** Networks are becoming global and the number of computers is increasing. Both factors increase the number of endpoints that must be addressed. There is a great disparity between the number of *potential* addresses (consider OSI 20 byte network addresses) and the *actual* number of addresses that are in use at a given time. This trend could exacerbate lookup times required to retrieve context at

various layers. Current industry bridges and routers seem to be pushing the limits of technology to provide reasonable cost lookups of packets at link rates.

- **Networks are becoming more diverse:** To quote from [CT90]: "protocols of tomorrow must operate over the range of coming network technology" and "a single end system will be expected to support applications that orchestrate a wide range of media . . . and access patterns." For example, there are already several standards for high speed Data Links (e.g., FDDI, HIPPI, Fiber Channel[Par93]) besides the ATM standards.

- **Network Software will continue to be hard to implement and maintain:** Since crucial portions of networking software often reside in the operating system kernel, designing and maintaining this software is difficult. Approaches like Integrated Layer Processing ([CJRS89]) — which makes protocol processing more efficient by customizing each protocol stack — can exacerbate this problem. We believe that network diversity will lead to an increasing number of protocol stacks.

In summary, adding headers to packets to save packet processing is a good tradeoff for high speed networks. The current climate of transition will probably result in headers being modified anyway for other reasons. The resulting techniques should address generic packet processing bottlenecks, especially data manipulation as well as increased lookup costs for very large networks. It is desirable that the techniques be general and flexible; the techniques used at a layer should continue to work as other protocol layers are changed in response to new applications and technology. We would also like *structured* approaches to improve protocol performance that have the benefits of techniques like Integrated Layer Processing without their disadvantages.

2 Previous Work and Paper Organization

In [CT90], a distinction is made between two kinds of protocol processing: *data manipulation* (functions that read or modify data) and *control* (functions that regulate the transfer of data). The main data manipulation functions are copying data, checksumming, encryption and presentation formatting. Control functions include multiplexing, framing, flow and congestion control, and error recovery. We describe previous work in reducing control and data manipulation overhead, and their limitations.

Reducing Data Manipulation Overhead

It appears ([CT90, Par93]) that data manipulation functions are a principal bottleneck. A sequence of data manipulations that involve x separate passes through the data bytes is often

limited to a throughput of B/x where B is the speed of the endsystem bus. To quote from [CT90]:

> ... data manipulations ... are usually thought of as distinct operations, since they occur in different layers. But this is not necessarily the most efficient design approach. With current RISC chips, which pay a very high cost to read or write memory, it is more efficient to read the data once and perform as many manipulations as possible while holding the data in cache or registers.

Clark and Tennehouse [CT90] call this technique (i.e., combining data manipulations) *Integrated Layer Processing*. For example, speedups have been reported by doing the checksum and copy operations for TCP at the same time [CJRS89]. Another idea, used to reduce the number of copies made of the data, is to add buffer names [KLS86],to packets; the names provided a pointer to the packet's destination in memory. Clark and Tennehouse [CT90] generalize this idea to Application Level Data Units (ADUs) and suggest that the sender add information about the eventual location of each ADU.

Unfortunately, as the information required for Integrated Layer Processing is often in different layers, it requires sequential parsing through intervening layer headers before this information can be found. Current implementations are handcrafted and combine processing at all layers; hence they have access to the information required for data manipulation. However, the diversity of protocol stacks and the difficulty of maintenance makes it desirable to provide more structured access to this information.

Recently, Abbott and Peterson [AP93] have suggested a general technique to integrate protocol processing while preserving modularity. They also suggest adding a pointer to the start of the application data (in a lower layer header) in order to avoid the need for sequential parsing. Our data manipulation header (see Section 3) is a generalization of their idea.

Reducing Control Overhead

We concentrate on one aspect of control overhead: reducing the cost of state lookups *at all layers of the protocol hierarchy*. Some of our techniques are also useful for reducing what we call *generic protocol processing*. This includes updating time stamps, sequence number book-keeping, updating flow control parameters etc.,

State lookups occur at all levels of the protocol hierarchy. They are required at the Data Link for protocol demultiplexing and address filtering, at the Routing Layer for address lookup, and at the transport layer to lookup connection state. Without the techniques in this paper, lookups have a worst case cost of $O(\log n)$, where n is the number of state table entries. To

quote [Par93], lookups "represent a very large faction of the cost of protocol processing, and finding ways to minimize lookup costs are important." Similarly, generic techniques to reduce packet processing are also important.

It is hard to characterize the cost of lookups because they are implementation dependent. For instance, hashing by exclusive-OR folding of address octets seems effective and easy to implement [Jai92]. In [CJRS89], the cost of hashing (in the best case) is taken to be 25 instructions. However, we believe the lookup techniques described in this paper are useful because: **1)** Our techniques do not depend on traffic patterns. **2)** The lookup costs accumulated over several layers do add up. Lookups are also critical for bridges and routers. **3)** Worst-case guaranteed lookup times is sometimes important (e.g., for packet filtering and monitoring applications) and makes some hardware implementations easier. **4)** If we wish to provide guaranteed performance to *flows* in a datagram network ([Par93]), one must either assign *flow IDs* to packets (which is not a solved problem) or infer a flow by hashing multiple address fields (e.g., source-destination addresses) which exacerbates the lookup problem.

An existing technique to reduce the cost of state lookups is *caching* [Par93, Jai92]. A generic technique used to reduce generic protocol processing is *Header Prediction*[CJRS89]. These existing techniques depend on traffic patterns ([Jai86]) and do not guarantee performance improvement. By contrast, our schemes use extra packet headers to provide performance improvements that are independent of traffic patterns. Another common technique [Tan81] used in *virtual circuits* is to first set up a virtual circuit identifier (VCI) for a packet flow; all packets with the same VCI can be processed similarly. The major disadvantage with this approach is the extra latency required to set up VCIs.

There are several hardware techniques for lookups including tree based searches (trie and binary search schemes [CLR90]) and Content Addressable Memories (CAMs). However, special hardware is expensive and inflexible; tree based schemes that require a logarithmic number of probes to memory do not scale well because memory is a bottleneck.

Summary of Previous Work and Paper Organization

In summary, valuable work has been done on data manipulation and Integrated Layer Processing, but most existing implementations are handcrafted. It is desirable to have a structured and general [AP93] approach to this problem. Similarly, many existing techniques for reducing lookup times and packet processing depend on traffic assumptions, and do not guarantee worst-case improvements. Techniques like VCI assignment (in Virtual Circuit networks) require reliable set-up protocols that add complexity and latency to data transfer. Hardware techniques are expensive and may not scale well to Gigabit speeds. Thus new software techniques that rely on extra packet headers may provide a real benefit.

The rest of this paper is organized as follows. Section 3 describes the use of a *Data Manipulation Layer* which allows a structured approach to Integrated Layer Processing. The next section describes several techniques for reducing the overhead of state lookup and generic protocol processing. We introduce two new techniques, *source hashing* and *threaded indices*. Section 4.1 describes *source hashing*: it shows that a simple random index chosen by a source in a datagram network can have many of the properties of a virtual circuit identifier (VCI) without the latency needed to set up VCIs. Section 4.2 describes *threaded indices*: this is an abstraction of the hop-by-hop mapping used in virtual circuits but is applied to destination addresses in datagram networks. Section 5 describes other techniques to obtain cheap lookups and some interesting combinations. We state our conclusions in Section 6.

3 Data Manipulation Header

We propose adding the information required for Integrated Layer Processing to a *Data Manipulation Header* that is placed in an easily accessible portion in the front of the packet. One possibility is to add this header directly after the Data Link header and before any Transport or Routing headers. This is convenient because the hardware or software driver that copies packets from the network to the host typically knows how to parse the Data Link header and hence can easily access the Data Manipulation Header (DMH). It is particularly convenient to add a DMH when a new Data Link protocol (for example the ATM AAL-5 adaptation layer) is being standardized; adding a DMH amounts to lengthening the Data Link Header.

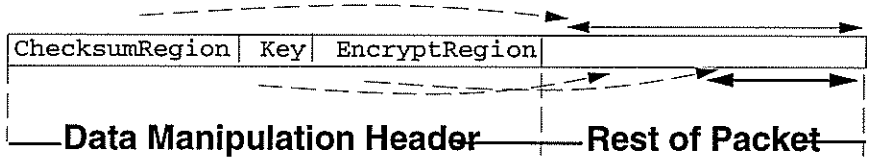

Figure 1: The Data Manipulation Header at the start of a packet contains pointers that allow data to be manipulated while it is being received. A region consists of an offset and a length. The figure shows three potentially useful regions for encryption, checksumming and minimizing data movement.

A DMH consists (Figure 1) of a sequence of variables called *regions*. A region consists of an offset, a length and a type field. The offset and length fields demarcate a sequence of bytes in the packet. The sequence of bytes represent the region of data to be manipulated (e.g., the transport data that is checksummed) or a region that provides information within the packet that is required for manipulation (e.g., a field that can be looked up in a table to yield an encryption key). The type provides information on the type of manipulation.

At the sending end system, it is easy to calculate the region descriptions with only a slight change to layer implementations. The idea is that the layer responsible for calculating a region passes down the region description (including the offset) to the next lower layer. Each layer n then adjusts the offsets of all regions created by higher layers (i.e., layers $n+1$ and above) to add in the length of the Layer n header. This simple structured technique of *accumulating offsets* deals with variable size headers and allows layer implementations to be changed easily.

We turn to specific data manipulations. Checksumming of transport layer data is used as an end-to-end check [SRC84] against undetected corruption. Checksumming only requires the checksum region to be demarcated. A reasonable solution for end-to-end data integrity and privacy is encryption at the transport layer. Encryption requires two regions: a region that can be used to extract the decryption key (when receiving a packet) and the actual region to be decrypted. For example, if each connection has a separate key, the decryption key can be found by extracting the connection ID and using that to lookup the decryption key.

It is desirable to avoid data copying by having packets be copied directly from the network to their final destination in application space [KLS86, CJRS89, BP93]. It seems desirable to have two regions for data copying. A *memory destination* region which is a field that can be used to find the location in memory for the data bytes, and another region to demarcate the data bytes themselves. It may also be desirable to have regions that demarcate regions that require presentation conversion [CT90] and specify the type of conversion required. However, it is infeasible to describe the conversions required for the general case (for example in an RPC application where each parameter has a separate type) using a single type field.

Adding a DMH makes it potentially possible for network adaptor hardware to do some of the data manipulations at the same time without understanding a great deal about higher layer protocols. It also allows various portions of the data to be dispatched to separate processors in a multiprocessor system without requiring a sequential parsing of layer headers. [CT90] points out that the traditional approach corresponds to "interpreting" the protocol suite. On the other hand, adding a DMH corresponds to partially "compiling" the protocol suite.

Our Data Manipulation Header can be considered as a generalization of the technique of [AP93] in which a single pointer (to the start of the application data) is added to a lower layer header. Our scheme is more general because it allows a *set* of regions to be demarcated. The regions can be in arbitrary portions of the packet, which allows further layers of application protocol hierarchy. While our scheme does not preclude software implementations, it *allows* hardware implementations to combine data manipulations without sequential parsing. We believe our structured technique of using accumulated offsets to calculate regions is also useful. On the other hand, we have not addressed a number of other issues relating to modularity that are well treated in [AP93]. In [AP93], modularity is preserved by "automatically synthesizing

the integrated implementation from independently expressed protocols." This is an important idea, and is equally applicable to protocols that use a Data Manipulation Header.

4 Lookup techniques

In the next three sections we concentrate on techniques to reduce lookup costs.

4.1 Source Hashing

In normal hashing, the consumer of an identifier (e.g., a compiler looking up variables in a symbol table) uses a determinisitic function to convert the identifier into an index into a hash table. The function must be deterministic so that all accesses to the identifier will index into the same position. However, it is hoped that the indexes produced by hashing various identifiers will be "random" so that hash collisions will be rare. In *source hashing*, by contrast, the producer of an identifier (e.g., the programmer!) picks a random index for each identifier and consistently tags every instance of the identifier with the same random index. The random index is used (in place of the output of a hash function) to index into the consumer's hash table.

Source hashing is certainly absurd in the simple compiler example described above. However, it is more interesting if the producer is the source of a packet and the consumer is the packet destination which must lookup a state table entry (corresponding to the source) in order to process the packet. Consider 6 byte Ethernet (more accurately, these are administered by the IEEE 802.1 committee) addresses. A number of network devices (most notably bridges), have hardware support to lookup state tables that are keyed by such 6 byte addresses. Some bridges use hashing for lookups. A bridge might use a hash function to convert the 6 byte address into a smaller index (say 12 bits); this hash index is used as an index into a hash table. If hashing with chaining is used, then each index entry points to a list of entries for the various addresses that have hashed into this bucket.

In source hashing, however, we suggest that the 6 byte address is increased to an 8 byte address: 6 bytes are still used for uniqueness, but the remaining 2 bytes is used as a random index that is *supplied by the source.* If it were possible to change the manufacturing process (which assigns unique addresses in ROMs), it is conceivable that the manufacturing process could even produce these extra 2 byte random numbers. However, if that is not possible it is easy for the source to provide the random 2 bytes using a pseudorandom number generator whose seed depends on the source address. All we need is that the same 2 byte random index

is associated with the same 6 byte UID; this association can be maintained by the source or through the manufacturing process. The source can change the association; the only cost is performance, and not correctness.

What we gain by this is that the *receiver does not need to compute a hash function*; the hash index is, in a sense, precomputed by the source. Computing a hash function was a trivial part of overall packet processing at low speeds; however, at higher (especially Gigabit) speeds the computation of the hash function is expensive in terms of either extra hardware or time.[1]. A second advantage is *the hash indices are guaranteed to be more truly random than those computed by hash functions.* In ordinary hashing, there is a tradeoff between the complexity of the hash function and the randomness of the resulting indices: good hash functions based on CRC-like functions are expensive; cheaper hash functions typically are not as good or need to be tuned to the details of the particular addressing format. These problems are avoided with source hashing.

Source hashing is a very general idea and is applicable to lookups at all layers, as we argue below.

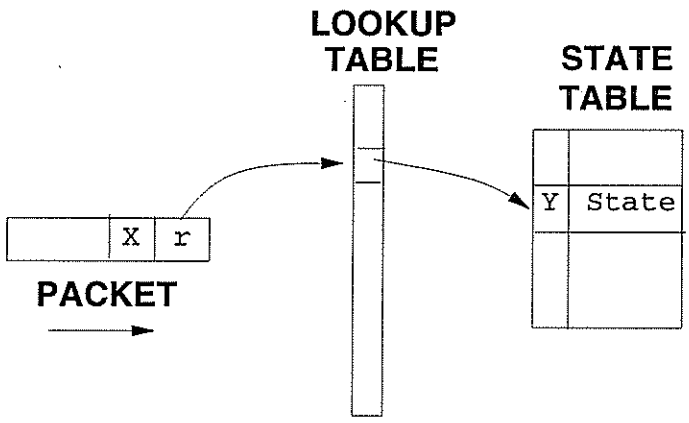

Figure 2: In source hashing, a random index r in the packet is used to index into a lookup table to yield a pointer into the state table. If the IDs match (i.e., $X = Y$) we are done; otherwise the state table is searched to find X's entry. All packets in a flow carry the same random index.

At the destination (Figure 2) the random index r is used to index into a *lookup table* (that corresponds to a normal hash table). The lookup table contains a pointer to a state table entry. If the pointer is null or the identifier Y in the state table entry is not equal to S, then we have a "miss". On a "miss" a search is done (perhaps by normal hashing) of the state table

[1]For example, DEC's GIGAswitch product, which is a multiport FDDI bridge, uses a semi-custom chip to do hashing at 100 Mbps

for the entry corresponding to S. However, if there is a "hit" (i.e., $S = Y$) then we simply access the state associated with the entry for Y.

The terms "hit" and "miss" were used deliberately to draw an analogy with caching. As in caching, we gain considerably if we get a "hit", and we lose slightly (because of the extra lookup and compare) if we have a "miss". However, caching performance depends on access patterns while we can make performance guarantees for source hashing that are independent of access patterns. Its basic performance depends only on the size of the lookup table and the number of sources that simultaneously send packets to the destination.

Also, in many software implementations the cache must be *searched*, often using a sequential search through a linear list[2]. (If there was a faster algorithm to search the cache, we could have used it to search the state table directly.) Thus the cache size is typically small. However, in source hashing since we use table lookup the "cache" size can be very large: in fact, as the size of the lookup table increases, the probability of a "hit" increases without increasing the cost of a "miss". Hardware cache implementations sometimes use Content Addressable Memories (CAMs); however CAMs are expensive and may not scale well as we move to higher speeds. One can think of source hashing as providing a "software approximation" of a CAM.

How do we deal with collisions? Two approaches present themselves immediately. The first, which we call *Newest Contender Wins (NCW)*, works as follows. When a new ID S enters to find that its lookup table slot r is occupied, a search is done for S's state table entry, and the pointer in slot r is made to point to S's slot table entry. The second approach, which we call *Oldest Contender Wins (OCW)* is the opposite of NCW; we do not change the entry in slot r. However, in order to age out old entries we now need an extra "age" bit in each lookup table slot r; this bit is cleared periodically and is set whenever there is an access to the state table entry being pointed to by slot r. If a certain amount of time expires without finding the "age" bit set, the entry is timed out by setting the slot pointer to some null value.

NCW is simpler (it does not require an "age" bit) and works well if IDs rarely collide in time and space. However, suppose two packet streams with different IDs pick the same random index r and their access patterns strictly alternate. Then NCW will provide poor performance to both streams but OCW will guarantee that one of the two streams will get good performance. NCW has the interesting property that it will never behave worse than a cache of size 1 (as is used in say header prediction [CJRS89]) but it is highly probable (assuming only that the number of simultaneous packet streams is much smaller than the lookup table) than the performance will be much better.

Sources can use pseudorandom numbers, preferably with seeds that depend on a unique

[2]In header prediction [CJRS89, Par93] the list is of size 1

source address. Source hashing does not require truly random random numbers that meet all statistical tests. To save computing time, a small queue of random numbers can be computed when a source first comes up. When a flow starts, the source picks the random number at the head of the queue; random numbers can be reused by placing them at the end of the queue when a flow terminates or becomes idle. What size random numbers should be used? The simplest approach is to require sources to provide fairly large random numbers (say 16 or 24 bits). Destinations that use smaller lookup tables can (assuming that the lookup table size is a power of 2) efficiently extract the right number of random bits — e.g., a destination with a lookup table of size 256 could use the low order 8 bits of the source random index.

Simple probabilistic analysis suffices to understand the behavior of source hashing, similar to the analyses done for hashing with chaining[CLR90]. The probability of a collision depends on the size of the hash table and the number of concurrent packet streams. We omit the analysis for lack of space. The typical analyses for hashing *assume* hashed values are uniformly distributed. In source hashing (given reasonable pseudorandom numbers), this assumption should be quite valid.

We now briefly describe three applications of source hashing: providing consistent random labels for congestion control and load balancing, for flow ID assignment and for transport protocol state lookup. In the process we will see that source hashing is useful for reducing packet processing as well as for reducing lookup costs. The applications will also use variations of the basic structure of Figure 2.

Consistent Random Labels and Fair Queueing

In the rest of the paper, we use the term *flow* [Zha91] to denote a stream of packets that traverse the same route from a source to a destination and have similar processing requirements. A flow could be identified by a *virtual circuit* in virtual circuit networks, or by packets with the same source and destination addresses in a datagram network; in all cases, the network must be able to distinguish different flows based on packet headers. In some cases, in a datagram network the source may wish to provide finer discrimination for flows than the implicit flows defined by source-destination addresses.

One important problem in datagram networks is to provide fair treatment [DKS89] to flows so that badly-behaved flows cannot penalize other flows: at the minimum we wish for firewalls between flows. In Figure 3 for examples assume that all four flows $F1$ - $F4$ wish to flow through link L to the right of node D, and that all flows always have data to send. If node D does not discriminates flows, node D can only provide fair treatment by alternately serving traffic arriving on its input links. Thus flow $F5$ gets half the bandwidth of link L and all other flows combined get the remaining half. A similar analysis at node C shows that $F3$

gets half the bandwidth on the link from C to D. Thus without discriminating flows, $F4$ gets 1/2 the bandwidth of link L, $F3$ gets 1/4 of the bandwidth, $F2$ gets 1/8 of the bandwidth, and $F4$ gets 1/16 of the bandwidth. In other words, the portion allocated to a flow can drop exponentially with the number of hops that the flow must traverse. This is sometimes called the *parking lot* problem because of its similarity to a crowded parking lot with one exit.

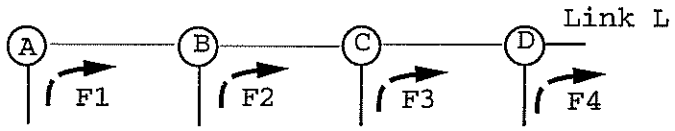

Figure 3: Solving the parking lot problem using source hashing to produce consistent random labels.

Most datagram networks do not discriminate flows and so this potential unfairness exists. Nagle [Nag84] proposed an approximate solution to this problem for datagram networks by having routers discriminate flows and then providing round-robin service to flows for every bottlenecked resource. Nagle proposed identifying flows using source-destination addresses and then using separate output queues for each flow; the queues are then serviced in round-robin fashion. Nagle's scheme was later refined and generalized under the category of *fair queueing* [DKS89] and *virtual clock* [Zha91] schemes. With fair queueing, for example, it is possible for each flow in Figure 3 to receive 1/5 of the bandwidth of link L.

Source hashing offers a simple alternative to Nagle's scheme of discriminating flows by source-destination addresses. Recall that in source hashing the source uses a consistent random index r for each flow. Thus in Figure 3 assume that each flow uses a different random index with high probability. Then we can use the random index to discriminate flows. We modify Figure 2 to have the lookup table entry pointing to an output queue for that random index. The output queues are then serviced in round-robin fashion. Also the simplest approach to dealing with collision, suggested by McKenney, is to ignore the problem. If two flows choose the same random index, they will be treated as one flow with some amount of unfairness. However, this is improbable and perfect fairness is probably not required anyway. Note that sources can obtain an arbitrarily fine level of discrimination for flows by assigning different random indexes for each flow.

Simple round-robin servicing (as suggested by Nagle) does not provide perfectly fair service because the packet sizes used by the different flows could be different. In [Shr94] we show how to overcome this deficiency by a technique called *Deficit Round Robin*. Details can be found in [Shr94].

Load Balancing

Another use for consistent random labels is in *load balancing* as shown in Figure 4. In this scenario, two routers connect two high speed speed LANs (e.g., 100 mbit/sec) using lower speed (say 1 Mbit/sec T1 lines) trunk lines. This is quite common because high speed, long distance, trunk lines are very expensive; it is often cheaper to use a number of lower speed trunk lines. In order to utilize these trunk lines, the router should *load balance* LAN traffic on the various output links. The simplest scheme is for $R1$ to send packets arriving on the LAN alternately on the trunk links. However, this can cause packets within a flow to be reordered which can lead to incorrect operation for some LAN protocols and to inefficient operation for other protocols[3]. If each flow carries a consistent random label, however, we can map each incoming label consistently to the same output line; this avoids reordering. We use each lookup table entry of Figure 2 to point to an output line; by controlling the number of times an output line appears in the lookup table we can control the amount of traffic that goes to an output line.

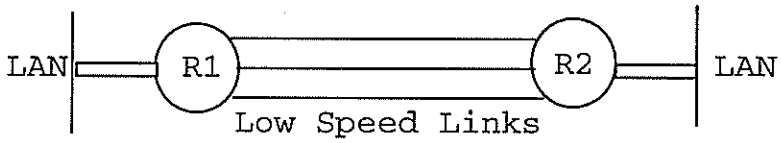

Figure 4: Two routers that connect two high speed LANs using a set of lower speed backbone links. Consistent random labels can be used to load balance traffic across backbone links without misordering traffic for any flow.

We will report the results of some simple simulation experiments in the final paper.

Assigning Flow IDs in Datagram Networks

In a virtual circuit network, a virtual circuit identifier (VCI) can be used [Par93] to distinguish flows at the cost of a set up delay. VCIs are typically small integers that can be easily looked using a simple lookup table. VCIs are also useful because they **a)** simplify packet processing (all flows with the same VCI are forwarded in identical fashion; in particular lookup results are the same) **b)** allow efficient discrimination of flows so that performance guarantees can be provided for each flow.

Because of these advantages, many similar proposals have been made [Par93] to assign flow IDs to flows in a *datagram* network by enhancing an existing internet protocol like IP. A strawman paper for assigning flow IDs presented in [Par93] is as follows. The source picks a unique (easily done by concatenating the source unique ID with an ID that is unique per flow at the source) flow ID for the flow and sends regular IP datagrams. The first datagram carries the flow ID as well as a *flow spec* (i.e., a specification of how the flow is to be forwarded).

[3]For example, some transport implementations do not cache out-of-order packets and may well be optimized for the case in which packets come in order.

Subsequent packets carry only the flow ID. If packets with an unknown flow ID are received the router sends a request back to the source asking for the flow spec.

We cannot use a consistent random label to replace the flow ID because two flows may choose the same label and have different forwarding requirements. Instead, we could use source hashing in the usual way to add a random index to the unique flow ID. This makes looking up flow IDs more efficient. If bandwidth is not a consideration, all packets in the flow could carry the flow spec, the flowID and the random index. The lookup table entry in Figure 2 now consists of a flow ID and some specification of how to forward the packet. On receiving a packet, we can use the random index to access the lookup table; if the flowID matches we use the stored specification of how to forward the packet. Otherwise, we use the flow spec in the packet to compute how to forward the packet, and store this information along with the packet's flowID in the lookup table entry.

The strawman proposal for assigning flow IDs has many problems; for instance, it does not address issues like heterogeneous receivers in a multicast flow, advance booking of flows etc. [Par93]. Thus we believe the application of source hashing to this problem is an interesting research area. Note also that the simple strawman proposal is a way to reduce packet processing for ordinary IP datagrams, *where the IP header itself is the flow spec.* The first packet in each flow will be processed the usual way; the remainder (with high probability) are processed much faster. Once again this is a generalization of header prediction to allow several simultaneous flows to be processed efficiently.

Timer-Based Transport Protocols

Source hashing is also applicable to looking up state in transport protocols. Transport protocols that use a handshake to set up connection [Tan81] can use *Round-Trip Index Passing.* The general idea is that both endpoints pass indices to each other. Each endpoint stores the index of the other endpoint and places this index in any subsequent packets sent to the endpoint. We call this *round trip index passing* because it takes one round trip delay before an endpoint can be sure that the other endpoint has received its index. This delay is no problem for Transport protocols that use 3-way handshakes to set up a connection. However, it is not useful for RPC Transport protocols (Section 4.1) which send data packets without waiting for a handshake.

Thus source hashing is most useful for timer-based transport protocols [Tan81] that avoid the extra round-trip latency. Such protocols are crucial for lightweight RPC implementations[BN84]. In Birrell and Nelson's RPC [BN84], each RPC call packet sent by the source carries a unique identifier (a unique network wide identifier of the process that initiates the call concatenated with a unique sequence number). Sequence numbers increase monotonically for each process.

The RPCruntime at the destination maintains a state table giving the sequence number of the last call made each calling process. When a call packet is received, its call identifier is looked up in this table; if the sequence is not greater than the last sequence number recorded for this process, the call packet is discarded as a duplicate. State information can be discarded at the receiver after a period equal to the maximum time a duplicate packet can live in the network. Thus if there is no information about a process in the state table, any new packet for that process is accepted and its sequence number must be recorded in the state table.

We can use source hashing to improve the lookup performance of such a transport as follows. First, we modify the lookup table in Figure 2 such that the lookup entries point to a *list of pointer blocks* much as in hashing with chaining [CLR90]. Each pointer block contains a pointer to a state table entry; state table entries contain a Process ID, the last sequence number for that process and any other state. A source process adds a consistent random index r with each of its call packets that is used to index the lookup table. Then the list of pointer blocks is searched to find an entry corresponding to the Process ID; if no entry is found, a new entry is made in the state table and a new pointer block is placed in the list.

This is very similar to doing hashing (with chaining) on the process ID except that the random index is a true random number chosen by the source. Unlike the general source hashing paradigm, there is no need for a slow search of the state table when an entry is not found in the pointer block list.[4] We plan to do experimental comparisons of the benefits of ordinary hashing versus source hashing in this context.

4.2 Threaded Indices

Suppose each node in a network could be assigned a unique 16 bit unique index along with (say) a 48-bit unique identifier. The unique identifier is unique to the node for all time; it can be assigned using some administrative procedures (as is done for Ethernet addresses). However, the index is *only unique for a particular network topology* to which the node currently belong, and can be changed whenever the topology changes. For instance, a two node network with nodes A and B can assign index 1 to A and index 2 to B. If we now add a third node C, we may assign A index 3, B index 2 and C index 1. If these unique indexes were placed in packets, we could lookup state information quickly using these indexes. Unfortunately, we require a distributed algorithm to assign unique indexes each time the topology changes.[5]

[4]There are some design details that need elaboration; for example, we need to deal with source and destination crashes as described in [BN84].

[5]Source hashing offers one approach. Each time a new end node comes up at a router, the router could pick a random index for the endnode and broadcast this proposed index to other routers. On collisions, it could retry this procedure. However, the details are complex.

However, if we look at Virtual Circuit Identifiers (VCI's), we realize [Tan81] that a VCI is not a unique network wide index; *instead a VCI is only unique per network link.* When forwarding a virtual circuit packet, the VCI of the incoming packet is used to index a virtual circuit state table that yields both the output link as well as the outgoing VCI. The outgoing VCI is then placed in the packet before the packet is forwarded on the output link. Thus the allocation of VCI's is no longer a global problem; one of the two end-points of each link can be responsible for allocating VCI's during set up.

Exactly the same idea can be used for datagram addresses. There could be a short index (in addition to the normal network address) that is mapped between routers. We call such an index a *threaded index.* Figure 5 illustrates the idea. Before the source S can send a packet to the destination D it first sends a query to the next router A asking for D's index (which is i at node A). S then caches index i and uses it on all subsequent packets to D. When a packet from S arrives at A, A uses the index i to index into its lookup table. The table entry contains the next hop (i.e., B) as well as the index into B's table (i.e., j). The new index j now is placed in the packet before it is sent to B. The process of following threaded pointers continues until the last hop, where no index is required.

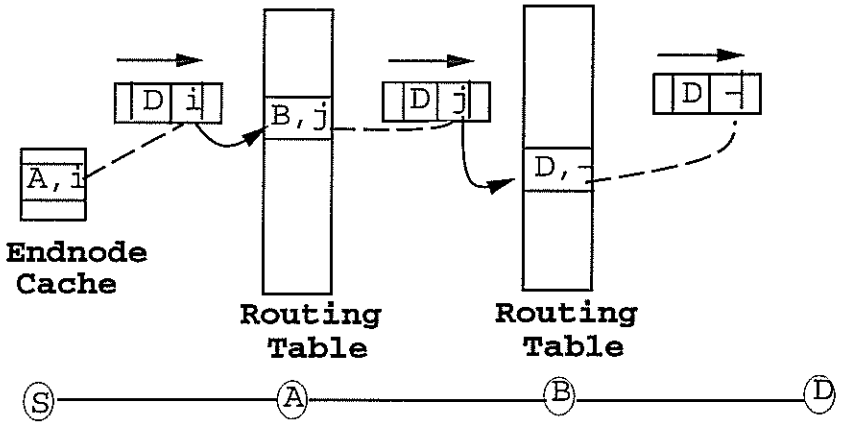

Figure 5: Threaded indices connecting routing entries in a path from a source S to a destination D through two routers A and B. The routing entry at each node contains the next hop node *as well* as an index (i.e., i, j) into the next hop node's routing entry.

There are two differences between *threaded indices* and VCIs. In a virtual circuit, the VCI is set up when a flow starts based on an explicit request from the flow. By contrast, threaded indices are *set up by the routing protocol whenever the topology changes.* (We describe an example algorithm below.) This is an advantage for threaded indices as it will typically take lower latency to start a flow. We need only one hop latency to do a query, a latency that is often required by Address Resolution Protocols (ARPs) anyway. VCI set up requires a

round trip delay through the network. The second difference is that the size of virtual circuit tables need only be proportional to the maximum number of simultaneous virtual circuits that traverse a node at the same time. Threaded indices require larger table sizes proportional to the number of network nodes (as in all datagram networks).

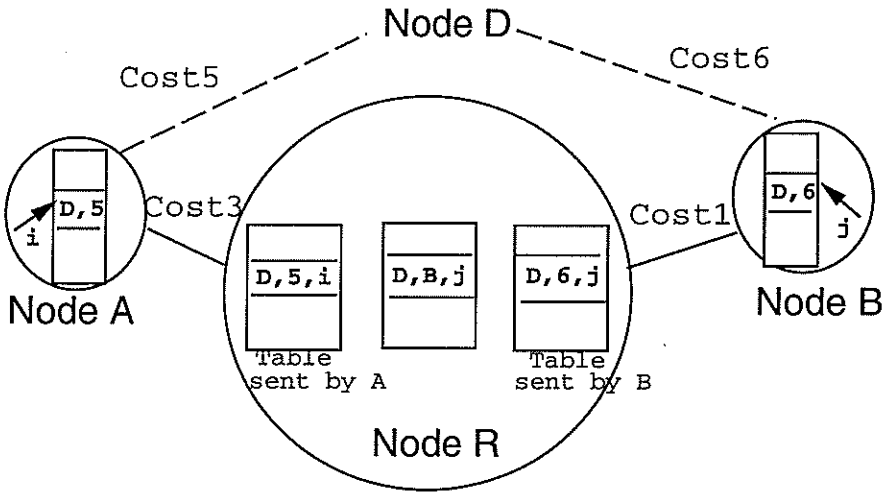

Figure 6: In Bellman-Ford, router R calculates its best path to destination D by choosing the neighbor (i.e., B) that yields the shortest cost ($6+1=7$) path to D. Each neighbor also passes an index into its routing table (e.g., j for B) together with its cost to D. R chooses its threaded index as the index (i.e., j) of its minimum cost neighbor (i.e., B).

We give a quick example of how to modify an existing routing protocol to calculate threaded indices. Many routing protocols (e.g., RIP) are based on the Bellman-Ford algorithm [Tan81, Per92]. The modifications required for Bellman-Ford are illustrated in Figure 6. Normally each router sends its shortest cost to each destination D to all its neighbors. A router R calculates its shortest path to D by taking the minimum of the cost to D through each neighbor. The cost through a neighbor like A is just A's cost to D (i.e., 5) plus the cost from R to A (i.e., 2). In Figure 6 the best cost path from R to D is through router B. We modify the basic protocol so that *each neighbor reports its index for a destination in addition to its cost to the destination.* Then each router uses the index of the minimal cost neighbor for each destination. In Figure 6, R uses B's index (i.e., j) in its routing table entry for D.

5 Combining Techniques

So far we have been concentrating on adding header information to packets that flow between nodes. We have described two new techniques (source hashing, threaded indices) and one

known technique (round-trip index passing). A last useful technique that complements the other three is what we call *passing indices* between layers of a *single node.* The idea is to modify the the header information passed between layers to reduce lookup costs. As an example, consider what is required to reduce the cost of looking up the endnode cache in Figure 5. Endnode caches are used by many routing layer protocols (e.g., IP, OSI [Per92]) in order to fill in information required to send a packet. Some routing protocols (e.g., DECNET Phase IV [DNA82]) even require updating this cache when packets are received.

[Par93] describes a simple idea in which the transport protocol (which keeps state anyway per connection) keeps an index into the routing cache. To implement this in a structured way, when transport calls routing it includes the index (say i) in the call as well as the destination address (say D). This index is used as a hint [Lam84] by routing. Routing uses i to lookup its cache. If the entry contains information about D that information is used; otherwise a normal search of the cache is done to locate the true index. The true index is then returned to transport to store as part of its connection state. In the usual case, there will be no need for any cache searching. Routing also can change its cache entries at any time without notifying Transport.

Suppose the routing protocol also needs to lookup the endnode cache when receiving packets ([DNA82]). Consider the following more bizarre application of index passing. Routing receives and validates a packet; Routing then makes an upcall to Transport with a portion of the transport header. Transport looks up the connection ID and finds the corresponding routing index. In the third step, transport passes the index down to routing which then uses this index to update the cache entry. While this may be too involved to be actually useful, it illustrates the generality of the idea. The general idea, of course, is to have higher layers (even applications) store indexes to lower layer state; the indexes can be passed down when a packet is sent or when an explicit request is made by the lower layer.

There are many obvious combinations of the four lookup techniques we have described. For example, for Transport Protocols (Section 4.1) we can combine source hashing and round trip indexing by using source hashing on any packets sent initially. However, after one round trip delay the destination returns a *round trip index*; this index is used in all subsequent packets. The index in the packet can be coded to distinguish between the random index (which indexes the lookup table in Figure 2) and the round trip index (which indexes the state table in Figure 2). Similarly we can have threaded indexes where the index used on each link is a random index (chosen by source hashing on each link). There many similar combinations that we leave for future work.

6 Conclusions

It is generally accepted that packet processing is more expensive than network bandwidth. In this proposal, we make two somewhat radical suggestions for additional packet fields to make packet processing faster. First, we propose adding a data manipulation header to an easily accessible portion of each packet. Second, we propose adding additional index fields to protocol identifiers at all layers. Examples of such identifiers include connection identifiers, Network Addresses and Data Link type fields.

The data manipulation header contains information about key fields in other layers; this information allows implementations to efficiently combine several data manipulation steps in a structured manner. On the other hand, identifier index fields can be used in various ways to reduce lookups and packet processing. Perhaps the simplest idea is to standardize such fields but to let implementations decide which of several possible techniques they wish to use. We have described several specific techniques in this proposal. Some of these techniques, especially *source hashing* and *index passing*, are new. To the best of our knowledge, they have not appeared before in the literature.

The use of a consistent random label for Source Hashing has other uses besides a way of reducing lookups. It can be used to assign flow Ids, for fair queueing, and for load balancing. While these can all be approximately done by hashing various address fields, source hashing provides more efficient and predictable performance. It is more predictable because even pseudo-random numbers picked by a source should be more "random" than the numbers produced by simpler hash schemes.

In conclusion, we note that the current climate of transition (in which transport, routing, and data link protocols are changing in order to support new technologies and applications) makes this a good time to consider adding additional header fields. Also, if some implementations disregard these extra header fields, this will only affect performance and not correctness. Thus it is easy to add these headers gradually to existing implementations while remaining compatible with older implementations. For these reasons, we hope that there will be opportunities to apply our techniques to influence the new generation of protocols that are swiftly emerging.

References

[AP93] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[BP93] D. Banks and M. Prudence. A high-performance network architecture for a pa-risc workstation. *IEEE Journal on Selected Areas in Communications*, February 1993.

[CJRS89] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, June 1989.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[CT90] D.D. Clark and D.L. Tennehouse. Architecural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings of the Sigcomm '89 Symposium on Communications Archtectures and Protocols*, 19(4):1–12, September 1989.

[DNA82] Digital Equipment Corporation. *DECnet Digital Network Architecture (Phase IV) General Description*, 1982. Order No. AA-N149A-TC.

[Jai86] Raj Jain. Packet trains: Measurements and a new model for computer traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):1162–1167, May 1986.

[Jai92] Raj Jain. A comparison of hashing schemes for address lookups in computer networks. *IEEE Transactions on Communications*, COM-40(10):1570–1573, October 1992.

[KLS86] N.P. Kronenberg, H. Levy, and W.D. Strecker. VAXClusters: a closely coupled distributed system. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[Lam84] Butler Lampson. Hints for computer system design. *IEEE Software*, 1:11–29, October 1984.

[Nag84] John Nagle. Congestion control in TCP/IP internetworks. *Computer Communication Review*, 14(4), October 1984.

[Par93] Craig Partridge. *Gigabit Networking.* Addison-Wesley, Reading,MA, 1993.

[Per92] Radia Perlman. *Interconnections.* Addison-Wesley, Reading,MA, 1992.

[Shr94] M. Shreedhar and G. Varghese. Efficient Fair Queuing using Deficit Round Robin Computer Science Technical Report WUCS-94-17, Washington University, July 1994.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[Tan81] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

[Zha91] Lixia Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. *ACM Transactions on Computer Systems*, 9(2):101–125, May 1991.