Lightweight Network Support for Scalable End-to-End Services*

Kenneth L. Calvert James Griffioen Su Wen
Laboratory for Advanced Networking
University of Kentucky
{calvert,griff,suwen}@netlab.uky.edu

ABSTRACT

Some end-to-end network services benefit greatly from network support in terms of utility and scalability. However, when such support is provided through service-specific mechanisms, the proliferation of one-off solutions tend to decrease the robustness of the network over time. Programmable routers, on the other hand, offer generic support for a variety of end-to-end services, but face a different set of challenges with respect to performance, scalability, security, and robustness. Ideally, router-based support for end-to-end services should exhibit the kind of generality, simplicity, scalability, and performance that made the Internet Protocol (IP) so successful. In this paper we present a router-based building block called *ephemeral state processing* (ESP), which is designed to have IP-like characteristics. ESP allows packets to create and manipulate small amounts of temporary state at routers via short, predefined computations. We discuss the issues involved in the design of such a service and describe three broad classes of problems for which ESP enables robust solutions. We also present performance measurements from a network-processor-based implementation.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Network

General Terms

Design

*Work sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0514, by the National Science Foundation under Grant EIA-0101242, and by a grant from Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

SIGCOMM'02, August 19-23, 2002, Pittsburgh, Pennsylvania, USA.

Keywords

ephemeral state, router achitecture, end-to-end services, programmable network

1. INTRODUCTION

The best-effort datagram service of the Internet Protocol has been a remarkably flexible and robust building block for a wide variety of end-to-end services. However, as the Internet matures, there is growing demand for additional network-level mechanisms to support new services and to improve the scalability and performance of existing ones. Broadly speaking, two general approaches to deploying new capabilities in the "waist of the hourglass" have been considered. The first is the common one: target a specific problem and develop a focused network-based solution to that problem. This approach is exemplified by services such as Express [9] and SSM [8] for scalable multicast routing, PGM and others [5, 30] for scalable reliable data distribution, and ECN [18] for early signaling of congestion without packet loss. This approach has the advantage that the business case and engineering tradeoffs are usually clear. Its disadvantage is that over time, the aggregation of one-off solutions decreases the robustness of the network. Moreover, building problem-specific solutions into the network may interfere with future possibilities unforeseen at the time of deployment.

At the other extreme is the approach exemplified by research in active networks, which emphasizes *generality* [7, 13, 29]. The advantage of this approach is that deployment of a sufficiently flexible platform allows all current and future problems to be solved—at least in theory. The disadvantage is that it is not at all clear whether or how a "sufficiently flexible platform" with the desired level of security, performance, and scalability can be engineered or deployed.

We propose to obtain the advantages of both approaches by identifying a set of simple and generic router primitives to support a broad range of new services, but not necessarily every possible service. To that end, we present a general-purpose network-level building-block service called *Ephemeral State Processing* (ESP). ESP supports end-to-end services by allowing packets to create limited amounts of temporary state at routers and invoke simple predefined computations on that state.

As a simple example of how such a service might be used, consider the problem of determining if the paths from a sender to two receivers share a set of common links. This problem might arise, say, in determining where to place a multicast reflector or gateway. If end systems can create

temporary network state and then query that state before it disappears, the sender can send a "marker" message to receiver A to mark that path, and then a "query" message to receiver B that records the last marked node, thereby identifying the common part of the path (see Figure 1).

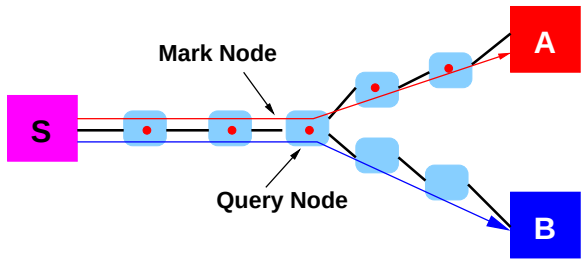

Figure 1: Finding common nodes along two paths.

Our contributions in this paper include the following. First, we present the design of ESP and the engineering goals and considerations behind it. Second, we describe three broad classes of problems that can be solved using ESP, and present example solutions, including an ESP-based reliable multicast service similar to PGM [5] and an aggregation service similar to concast [3]. Third, we describe a network-processor-based implementation, and a design for an *ephemeral state store* that supports fast access to millions of values per node using commodity memory with negligible management overhead. Fourth, we present encouraging initial performance results for this implementation.

The remainder of the paper is organized as follows. The next section presents design goals, architecture, and components of ESP. Section 3 describes ways to use ESP, and Section 4 addresses important issues that may arise in using it, including error handling and security. Section 5 and Section 6 describe our implementation on the Intel IXP1200 network processor and present performance results that illustrate the scalability of the service. Related work is discussed in Section 7, and Section 8 concludes the paper.

2. EPHEMERAL STATE PROCESSING

The remarkable success of the Internet Protocol can be attributed, at least in part, to its simple, generic service abstraction. Our general goal in developing ESP has been to reflect those characteristics that have made IP successful. Before describing the ESP architecture itself, we first highlight what we consider to be the key requirements for its design.

2.1 Design Requirements

The starting point for our extension to the network layer is that it *enable packets to leave information at a router for other packets to modify or pick up.* Although this basic capability is clearly *not* characteristic of IP, it makes possible a number of interesting uses, and we believe it is essential for a general building block. Yet user-controlled network state is something to be approached with extreme caution: the prospect of maintaining state for hundreds of thousands of flows through a core router is rather daunting.

The key observation, however, is that the important quantity when it comes to state is actually the *space-time product* of storage: Little's Law, from queueing theory, tells us that if the average holding time of a storage resource goes down, a system with a given fixed capacity can accommodate a higher average arrival rate of customers. The conventional approach to user-controlled state in the network is called *soft state*. The general concept of soft state is that the resource is reclaimed (only) if it is not "refreshed" periodically. With soft state, the holding time (and thus the space-time product) of the resource is unbounded; this makes it necessary to limit the ability of packets to create and refresh state, which in turn introduces all sorts of requirements for authentication, etc. To avoid these complications, we add a second requirement: *the space-time storage requirement per flow (in fact, per packet) is bounded.*

Users can send IP datagrams at any time, without prior arrangement, and each packet is handled independently of all others. Packets are transmitted anonymously, except for addresses. One reason this is feasible is because the Internet Protocol requires a bounded amount of processing per packet. Moreover, the processing requirement is essentially fixed, so that it can easily be implemented in hardware. To ensure that ESP has these desirable characteristics, we require that *the amount of processing required per packet at each node must be comparable to that of IP*—in other words, it must be "too cheap to meter." And it must be *anonymous*, in the sense that routers do not care which end system is using the service, end systems do not care which routers process their packets, and no central authority need be consulted for permission to use the service.

Finally, for generality we want network-layer independence. Also, ESP should fit comfortably within the architectural context of modern routers and the Internet Protocol, and not reinvent or modify existing network services. Thus *ESP relies on the network layer for forwarding only.*

2.2 ESP Architecture

To achieve these goals, we designed a new network-level building-block service based on ephemeral router state and small bounded per-packet processing costs. The three main components of the system are the *ephemeral state store* (or ESS), in which packets can save and retrieve small amounts of state, the *instruction set*, which defines the computations that packets can invoke, and the *protocol*, which defines the way ESP packets are processed as they are forwarded through the network.

The basic idea is that each ESP packet specifies a single instruction, which operates on information carried in the packet and/or stored at a node. As packets traverse the network, they create, modify, or retrieve small amounts of state at each ESP-capable router along the path from the source to the destination. This state information exists only for a short time (say 10 seconds) and must be used by subsequent packets within that interval. After processing at a router, each ESP packet is either forwarded toward its destination or silently discarded, according to the result of the instruction execution.

The flexibility and generality of ESP stems from the ability to execute sequences of instructions in *both space and time*: a single packet creates a sequence in space as it traverses a path through the network, while an individual node executes a sequence of packets in time.

The scalability of ESP derives from two factors. First, ESP processing can be extensively parallelized: only pack-

ets that belong to the same end-to-end computation need to share state and be processed serially. Second, per-packet resource requirements can be precisely bounded. These factors make it possible to do most ESP processing *in a local interface context*, i.e. on the port cards of the router (as is typically done for IP forwarding). In other words, there is no fundamental architectural reason for ESP packets to be diverted far from the fast path for processing. Moreover, port-card-based implementations only have to process packets at "wire speeds"—or even less, depending on the level of parallelism. Finally, the ephemeral state store in each ESP processing context (whether port-card-based or centralized) can be implemented by multiple separate small stores rather than one large monolithic store, provided that packets that share information are always processed using the same store.

Note that centralized ESP processing *is* required for some kinds of end-to-end services—namely, in situations where the packets of a computation do not pass through a common port at some router. Although our experience so far suggests that port-based processing suffices for most applications, our design nevertheless assumes that every ESP-capable node has a centralized ESP facility *in addition to* a separate ESP-processing facility on each port card. The packet protocol (Section 2.5) allows each packet to specify the context(s) in which it is to be processed at each node.

The remainder of this section considers the three ESP components in greater detail.

2.3 The Ephemeral State Store

Much of the power and scalability of ESP arise from its use of an associative memory called an *ephemeral state store* (ESS) at each node. Like other associative stores, the ESS allows data *values* to be associated with keys or *tags* for subsequent retrieval and/or update. However, a key feature of our approach is that the ESS supports only *ephemeral* storage of (tag, value) pairs; each (tag, value) binding is accessible for only a fixed interval of time after it is created.

The *lifetime* of a (tag, value) binding in the store is defined by the parameter τ, which is required to be approximately the same everywhere in the network. Once created, a binding remains in the store for τ seconds and then vanishes. The value in the binding may be updated (overwritten and read) any number of times during the lifetime. For scalability, we want the value of τ to be as short as possible; for robustness, it needs to be long enough for interesting end-to-end computations to complete. For the purposes of this paper, the lifetime is assumed to be about 10 seconds.

A fundamental principle of our service is that the lifetime cannot be extended. This difference between ephemeral state and soft-state is subtle but important. With soft state, *the user controls when the resource is released*. It follows that the system cannot guarantee any particular rate of resource availability, and is therefore vulnerable to denial-of-service attacks. With ephemeral state, on the other hand, *resources are reclaimed at the same rate they are allocated*. Reclaimed resources are equally available to *all* users. It follows that for any given maximum rate of binding creation and value of τ, the size of store needed to guarantee that every creation attempt succeeds is fixed, and can be determined in advance.

The ESS is modeled as a set of (*tag, value*) pairs where each tag has at most one value bound to it. Both tags and values are fixed-size bit strings. No structure is imposed on either tags or values by the state store; their meaning and structure is defined by the applications. Tag selection is completely distributed—any user can use any tag.

Conceptually, the ESS is accessed via two methods:

put(*tag*, *value*): binds *value* to *tag*.

get(*tag*): returns the value bound to *tag* or *nil* (which differs from every value) if *tag* has no value.

ESP instructions use these two simple methods to create, read, and modify bindings in the ESS. Section 5.1 describes how these operations can be implemented efficiently.

The utility of the ephemeral state store depends on *users choosing tags randomly*, and having a large enough space of tags to choose from that the probability of users choosing the same tag ("colliding") during any interval $[t, t+\tau]$ is small. In addition, it should be impractical for any user to guess another's tag by any brute-force method. If tags are chosen truly randomly and the number of distinct tags is sufficiently large, the effect is that *each user sees a "private" ephemeral state store*. Indeed, tags can be thought of as variable names that have global significance; thus users are motivated to choose their names (i.e. tags) carefully.

Our current design uses 64-bit tags and 64-bit values. For tags of this size, the probability that any given random tag collides with one or more tags in a set of r (randomly-chosen) tags is given by:

$$1 - (1 - 2^{-64})^r$$

The probability that at least one collision occurs in a set of r randomly-chosen tags is given by:

$$1 - \prod_{i=1}^{r-1} (1 - (i/2^{64}))$$

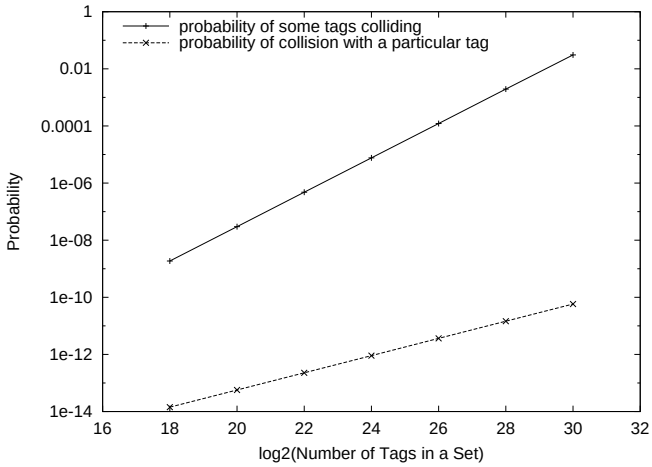

Figure 2: Probability of collision for varying numbers of users with 64-bit tags.

Figure 2 shows the relevant probabilities for group sizes from 2^{18} to 2^{30}. The top curve is the probability of *some* collision occurring in the group. The bottom curve is the probability that *the* particular tag chosen by a user will collide with another tag in the group. Obviously these probabilities are extremely low. If the total number of tags extant

```
COUNT(pkt p)
  α = get(p.C)
  if (α is nil)
    α = 0
  α = α + 1
  put(p.C, α)
  if (α ≤ p.thresh)
    forward p
  else discard p
```

```
COMPARE(pkt p)
  α = get(p.V)
  if (α is nil)
    put(p.V, p.current)
    forward p
  else if (α <op> p.current)
    put(p.V, p.current)
    forward p
  else discard p
```

Figure 3: Example ESP instructions

in the Internet during an interval of τ seconds reached one billion, the probability of a collision involving tags *anywhere* in the Internet would still be only 3%. In practice, collisions only occur when different computations use the same tag at the same state store —along the same path, for example.

To summarize: the short lifetime makes it possible to place a strict bound on the space-time resources used by any packet, thereby allowing routers to handle worst-case loads and maintain line-speed processing. The use of randomly-chosen tags to identify state makes it possible to completely decentralize naming, and to create and use a binding in one step.

2.4 ESP Instructions

Each ESP-capable node in the network supports a predefined set of *instructions* that can be invoked by ESP packets (described in the next section) to operate on the ESS. ESP instructions are analogous to the instruction set of a general-purpose computer: each involves a small number of operands and takes a fixed amount of time to complete. *Because each ESP packet invokes a single ESP instruction, the per-packet processing time is known and bounded.* The key differences with traditional machine instruction sets are that (1) sequencing must be achieved by arranging for a sequence of instruction-invoking packets to arrive at the router (i.e., no program counter), and (2) operations can only retrieve values placed in the store within the last τ seconds. Nevertheless, interesting computations can be constructed by transmitting sequences of packets through the network.

Each ESP instruction takes zero or more operands, each of which may be:

- a value stored in the local ephemeral state store (typically identified by a tag carried in the ESP packet);
- an "immediate" value carried directly in the packet;
- a well-known parameter value (e.g. the value of a MIB variable).

ESP instructions affect only the local state where they are executed; they either run to completion or abort (resulting in an error indication). Each instruction executes atomically with respect to the ephemeral state store; upon completion, the initiating packet is either (silently) dropped or forwarded toward its original destination.

We envision routers supporting a standard set of a perhaps a few dozen ESP instructions. Here we describe two instructions needed for examples in the next section; others will be introduced as needed later in the paper.

The COUNT instruction takes two operands: a tag C identifying a "counter" value stored in the ESS, and an immediate value, *thresh*. As the name implies, the COUNT instruction can be used to count packets passing through the router. Once the count C reaches the value *thresh*, subsequent count packets will increment the counter but will not be forwarded. This is useful, for example, when counting the number of neighbors sending packets through a router. COUNT is also useful as a "setup" instruction for subsequent instructions that collect information.

The COMPARE instruction takes three operands: a tag V identifying the value of interest in the ESS, an immediate value *current* which is the user-supplied value carried by the packet, and an immediate value `<op>` that specifies a comparison operator to apply (e.g., $<$, $\geq$, etc). The COMPARE instruction tests whether the relation specified by `<op>` holds between the value carried in the packet and the value in the ESS. If so, the value from the packet replaces the value in the ESS and the packet is forwarded. The COMPARE instruction is particularly useful in situations where only packets containing the highest or lowest value seen by the node so far should be allowed to continue on. Pseudocode for both instructions is shown in Figure 3.

2.5 ESP Packets

The third component of the ESP service, which coordinates and ties together the functionality provided by the first two components, is the *ephemeral state packet protocol*. As ESP packets travel through the network toward their destination, they are recognized as such by ESP-capable routers and processed hop by hop. (If necessary, ESP datagrams can carry a Router Alert option [11] to indicate that they should be examined by routers for special processing.) Non-ESP-capable routers will simply forward ESP packets as usual.

Two forms of ESP packets are supported: dedicated and piggybacked. A *dedicated* packet consists of an IP datagram whose payload contains the identifier of the desired ESP instruction along with its packet-borne operands. The IP header of the datagram carries a protocol number indicating the ESP protocol. A *piggybacked* ESP packet carries the ESP instruction in an IP option (IPv4) or extension header (IPv6). Piggybacked ESP packets initiate instructions as a side effect of carrying the normal data through the network. They offer the advantage of not adding to the bandwidth requirements of the application; their disadvantage is that the IPv4 option mechanism limits the size of the instructions that can be carried.

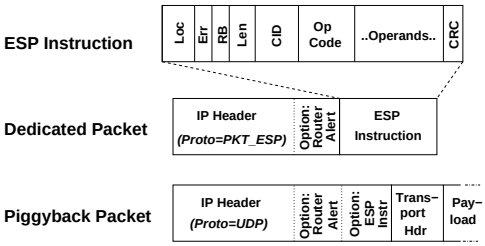

Figure 4: ESP packet formats

Each ESP instruction contains control information, an opcode, and operands. The instruction format and the header context for both types of packets are shown in Figure 4. The 3-bit *Loc* field specifies whether execution occurs on the *input* port card, the *output* port card, in the *centralized* location, or any combination of these three locations (including no processing at all). The *Err* field is set when an

error occurs while processing an ESP packet (e.g., failure to create a binding because the ESS was full). Packets with *Err* are forwarded on to the destination without further processing, allowing the end systems to discover that the operation failed. The *RB* is a "reflector" bit. ESP routers forward packets with the reflector bit set *without* processing them; when the packet reaches its destination, the ESP implementation swaps the source and destination IP addresses, unsets the reflector bit, and sends the packet back to the original source. We have found this capability useful for obtaining information along the path from a network router to an end-system. The *Computation ID* (CID) is a demultiplexing key: different packets that need to access the same state must have the same CID. ESP implementations are required to ensure that instructions bearing the same CID are executed in the same ESS context. (This enables the use of multiple, parallel ESS's on a single port card.) The *Opcode* identifies the ESP instruction to be performed and the *Operands* field carries the opcode-specific operands.

Note that the only parts of an ESP packet that are modified by processing en route are the instruction operands and the control bits (Loc, Err, etc). In particular, the IP header is not changed; thus packets cannot be diverted from their original path. Although the ability to "redirect" ESP packets as a result of processing (e.g. by modifying the destination address based on state) would be useful in some circumstances, it opens up opportunities for abuse, and also violates our general separation of concerns. Instead, a redirection service could be implemented as a separate, but complementary, building block service [27]. The Internet Indirection Infrastructure [25] may be a good match for this purpose.

2.6 Application Programming Interface

Applications running on end systems access the ESP service (i.e., initiate computations and collect results) via an API that allows information to be placed in outgoing ESP packets and extracted from incoming ESP packets. To send or receive ESP packets, the application associates an ESP communication endpoint (e.g., a socket) with a Computation ID, which identifies the computation in which the endpoint is participating. Thereafter the same CID is placed in all outgoing ESP headers, and incoming ESP headers containing CID are delivered to that socket. In addition to the Computation ID, senders must specify the destination to which the ESP packet should be sent. The destination address can be bound to the endpoint or specified on a per-packet basis.

To cause ESP instructions to be piggybacked on the packets of an existing flow, the application simply invokes an API call (e.g., socket option) that arranges for the ESP option to be added. The destination does not need to be specified since it is already known.

The API may also provide general methods to construct (and parse) ESP packets given opcodes and operands (tags and values). In case an ESP error arises during transmission, the API must also inform the application of the specific error that occurred so that it can take corrective measures.

3. APPLICATIONS OF ESP

To illustrate the utility of ESP, in this section we show how it can be used to solve three general types of problems, each of which is difficult to solve using end-system-only approaches. Specifically, we consider the problems of *controlling packet flow*, *simple computations on end system data*, and *discovering topology information*. Because of space limitations our examples are confined to these classes, but we do expect that ESP will prove useful to a wide range of end-to-end services and applications. In the descriptions that follow, loss-free operation is assumed; Section 4.3 discusses error control.

3.1 Controlling Packet Flow

One of the simplest functions the network can perform under application control is simply to *not* forward packets. It turns out that this capability can be useful in a variety of contexts. For example, the scalability of many multicast applications—especially those requiring some form of feedback or reliable delivery—is limited by the twin problems of implosion and wasted bandwidth (arising from feedback to the source and data retransmitted to group members that don't need it). A number of techniques, both network- and end-system-based, have been proposed to regulate packet flow in an attempt to avoid these problems and scale to larger group sizes. Examples include hierarchical aggregation [16], feedback rate control in RTCP [20], randomized delays for multicast NACK suppression [6], and repair subcasting in reliable multicast protocols [5, 14].

The basic idea common to all of these is elimination of unnecessary packets as they pass key locations in the network. This can be accomplished in ESP as follows:

1. Create ephemeral state at router interfaces where packet-pass/drop decisions will be made.

2. Send packets carrying an ESP instruction that makes the pass/drop decision based on the ephemeral state (or absence thereof), and possibly updates the state in the process.

The following sections present two specific examples of the use of this approach to solve real-world problems: feedback thinning and PGM-like NACK suppression and subcasting.

3.1.1 Multicast Feedback Thinning

Group applications often require feedback to be sent from group members to a common destination such as a multicast source. For example, the RTP/RTCP protocol [20] defines feedback messages ("receiver reports") that carry information such as the number of packets lost. The source then uses the information to adjust its transmission rate, encoding scheme, etc. The challenge, as the group size grows, is to avoid implosion while maintaining the timely nature of the feedback.

Let the feedback information of interest be represented by the generic parameter u. In many cases (including RTCP) the source is only interested in the extreme values of the feedback—e.g. the *maximum* value of u transmitted by any group member. Using ESP, the danger of implosion can be substantially reduced by recording the maximal (in whatever ordering is relevant) value forwarded at any point, and only passing packets carrying values that exceed that maximum. Thus, packets are discarded as soon as it is determined that they are not "interesting" to the destination.

Assume, for example, that we are interested in the maximum value of u at any group member. Periodically group members transmit an ESP COMPARE instruction to the source.

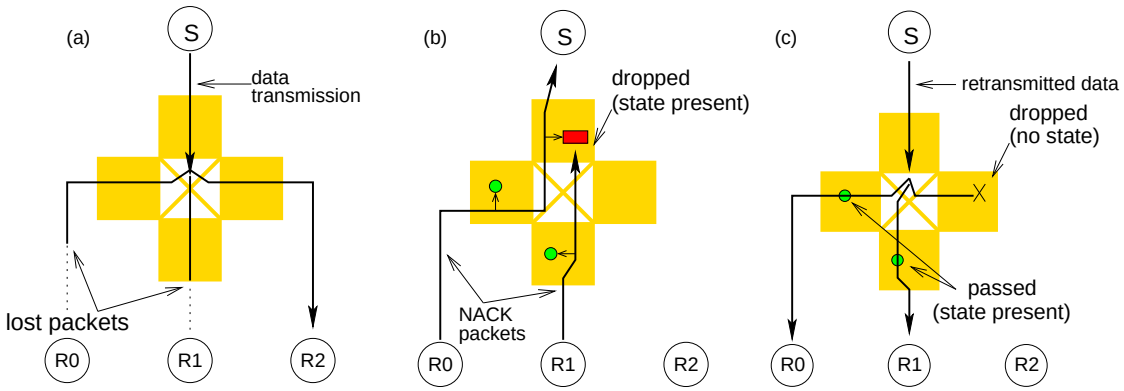

Figure 5: NACK suppression and subcasting using ESP.

Each member includes its value of u as the immediate value to the COMPARE instruction, uses "<" as the compare operation, and t_u as the ESS tag (where the tag name t_u is discovered via some out-of-band mechanism). As packets travel toward their destination, the COMPARE instruction updates the maximum value seen at each node. Packets whose values are not larger than the current maximum are discarded. The result is that the destination application receives a "thinned" sequence of packets, each containing a larger value of u than the previous one. Assuming values arrive at nodes in random order, the result is that the total number of packets arriving at the source is, on average, exponentially smaller than without this filtering capability.

3.1.2 *Enhancing Reliable Multicast*

As a more elaborate example of controlling packet flow with ESP, consider a reliable multicast enhancement service with functionality similar to that provided by PGM, a router-based protocol designed specifically to support reliable multicast [5].

Like PGM, the ESP-based service is based on negative acknowledgements and subcast retransmissions. Multicast receivers send a negative acknowledgement (NACK) to the source, or a designated proxy, whenever they detect a loss; the NACK contains the identifier of the lost packet. Implosion is avoided by discarding duplicate NACKs inside the network. Upon receiving a NACK, the multicast source or proxy re-multicasts the lost packet. To avoid wasting bandwidth delivering retransmissions to all nodes, a form of subcast is implemented by forwarding the retransmission only to the portion of the multicast tree through which NACKs were forwarded.

To implement this service, NACK packets carry an ESP instruction that marks both incoming and outgoing interfaces of each node visited. The mark on the *output* (upstream) side is used to suppress subsequent NACKs for the same data packet. This is implemented with a COUNT instruction with threshold of one (or higher for fault tolerance; see Section 4.3). The mark on the *input* side is necessary for subcasting. Retransmissions carry a piggybacked ESP instruction, processed on the *output* (downstream) side, that allows the packet to pass only if the interface was marked by a NACK for that sequence number. Figure 5 illustrates the operation in a single router with four interfaces. In (a), the multicast source S transmits to three receivers, but the packet is lost before reaching R0 and R1. In (b), R0 and R1 send a NACK with an ESP instruction piggybacked; the instruction deposits a small amount of state (with a tag determined by the sequence number of the missing data) in the input interface. It also checks for the presence of a similar bit of state on the output interface, and the packet is forwarded only if the state is *not* present or the threshold has not been exceeded. (The figure shows operation with a threshold of one.) In (c), the source retransmits the requested data, with a piggybacked ESP instruction that checks for the existence of state at the output interface, and the packet is forwarded through the interfaces toward R0 and R1.

This solution works well *provided* the path followed by NACK packets is the reverse of the path followed by data (and retransmission) packets. However, if the paths are not symmetric, an alternative method is needed to relay NACK packets back up the forward data path. The same situation arises whenever protocols require that network state be accessed on both the forward and backward path, for example in RSVP and PGM. Various solutions for this problem exist, including a protocol designed specifically for this purpose [22].

3.2 Simple Computations on User Data

Distributed applications commonly have need to distill or aggregate data supplied by the participants. Perhaps the simplest example is counting the number of receivers in a multicast group; others include tallying votes, computing the average load, and calculating the amount of work remaining in a system. This kind of aggregation is typically accomplished by applying an associative and commutative operator to values supplied by group members. However, if the group is large, it is impractical for a single node to collect all the input and perform the computation; if the group members are anonymous, it is difficult to impose a structure that would allow the result to be computed by the group in a distributed fashion.

Both of these problems can be solved via tree-structured ESP computations. The basic idea is for a particular node to be designated as the collector or destination; the paths from group members to the destination node form a tree. Each member sends its value to the destination in an ESP packet; each interior node of the tree collects values from its children, performs the computation, and forwards the result to its parent. Eventually the destination receives the (single)

```
COLLECT(pkt p)
  α = get(p.V)
  if (α is nil)
    α = p.val
  else
    α = α ∘ p.val
  put(p.V, α)
  β = get(p.C)
  if (β is nil) abort
  β = β - 1; put(p.C, β)
  if (β == 0)
    p.val = α
    forward p
  else discard p
```

Figure 6: Instruction for aggregation computation

result. Risk of implosion is reduced because each interior node sees packets only from its immediate children in the tree. Associativity and commutativity of the operation enable collection and computation to be interleaved—that is, packet values can be combined with the computation state one at a time as they arrive.

In general, these tree-structured "aggregation" computations are carried out in two phases. In the first phase, each non-leaf node learns of its children in the tree. In the second phase, group members send their values up the tree (toward the destination). Each node computes and forwards its result only after having heard from each of its children. The first phase can be accomplished with the COUNT instruction described in Section 2.4. The second phase can be accomplished with a simple COLLECT instruction, which takes four operands: A tag V, identifying the result of the computation so far in the ESS; an immediate value *val*, which carries the value to be contributed by this packet; a tag C, identifying the child-counter established by the COUNT instruction in the first phase; and an immediate value $\circ$, which specifies the associative and commutative operator to be applied. COLLECT applies the user-specified operation $\circ$ (e.g., addition or subtraction) to the value in the packet and the value stored at the node. It then decrements the "child count" (C). When C reaches 0, all children have reported and the result is forwarded to the next hop.

Note that the two phases of the computation share ephemeral state, so both must be completed at each node within one ephemeral state lifetime. (See also Section 4.2.)

3.3 Discovering Topology Information

Recently researchers have proposed various end-to-end services based on the ability to invoke special (predefined) functionality at particular locations inside the network. By enabling special functions at precisely the right nodes, end systems can control the way their packets are processed en route. Examples of such functions include packet duplication for multicast [26, 28], marking or logging for traceback of denial-of-service attacks [19, 24], ingress filtering [15], and packet redirection for overlays [10, 25]. Although ESP is not designed to provide such "heavyweight" functions itself, it can be used to solve the problem of determining *where* in the network such functions should be invoked.

The basic approach, again, is to first send packet(s) that set up state to distinguish the desired node from others, and then send packet(s) to recognize and collect the address(es) of the distinguished node(s). The example given in Section 1 and shown in Figure 1 illustrates this.

In earlier work, we showed how ESP can be used in combination with the ability to invoke special processing at specific routers, to enable new types of services. We proposed the use of dynamically-invoked *duplication modules* at routers to implement multicast using unicast forwarding [27]. ESP is ideally suited for discovering a good (efficient) location to invoke a duplication module. In other work, we have shown how to use ESP to identify bottleneck links in multicast trees [28], and (in the context of layered multicast) use that information to install thinning modules for congestion control.

4. PRACTICAL CONSIDERATIONS

In this section we address various questions of a pragmatic nature that may arise when designing, deploying and using ephemeral state services.

4.1 Partial Deployment

One cannot assume that every router in the network will simultaneously begin supporting any new capability. Therefore services based on ESP need to operate correctly when only a subset of routers are ESP-capable. Because non-ESP-capable routers simply forward ESP packets, all of the algorithms described in this paper operate correctly with partial ESP deployment, albeit with reduced performance and scalability in some cases. For example, the path-intersection computation described in Section 1 returns the last *ESP-capable* node common to both paths.

In general, the performance and scalability of ESP services improves as the fraction of nodes supporting ESP grows. For some applications, such as the aggregation application described in Section 3.2, we have found that most of the benefit can be obtained even if the functionality is deployed only at the edges of domains [3].

4.2 Timing

Distributed computations using ephemeral state require some level of coordination to ensure packets arrive at all nodes within an interval of duration τ. The simplest method is to have a controlling node transmit a stimulus message to participating hosts, which then respond by sending the appropriate ESP packet. This stimulus message may need to be reliably transmitted.

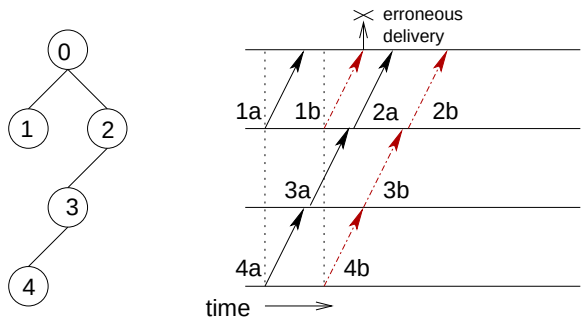

Figure 7: Erroneous aggregation computation: "a" packets establish child counts, "b" packets collect values.

When stimulus-based coordination is inadequate (e.g., reliably disseminating the stimulus introduces too much jitter)

some other coordination method is required. If the participating hosts have a common time source the controlling node can instead transmit the *start time* of the computation to the participants. The start time simply needs to reliably reach the participants in advance of the indicated time. Note that the accuracy required of the common time base is modest—as long as the participants' clocks are within the (assumed) maximum end-to-end network delay of each other, the computation will be coordinated to the same degree as with stimulus messages.

A related issue is the timing between phases of a multiphase computation. The inter-phase delay must be adequate to ensure that all packets of one phase are processed at each node before any packets of the next phase—otherwise errors can result. Figure 7 illustrates this for the aggregation computation: Leaf nodes 1 and 4 start each phase (labeled a and b) of the computation at the same time. However, because the b phase starts at Node 1 before all a packets have made it to Node 0, Node 0 prematurely delivers the result.

Given an upper bound δ on the one-way transit delay through the network, an inter-phase delay of 2δ suffices to ensure that this kind of error does not occur.

4.3 Dealing With Errors

ESP is a best-effort service; like other IP datagrams, packets carrying ESP instructions are subject to various misfortunes including loss, reordering, and duplication. The effect of such errors on end-to-end computations depends on the particular computation. The choice of how best to deal with such errors is ultimately up to the application itself. Our intent here is to consider the effects and highlight some principles for designing computations to be robust against errors. We focus here on losses because ESP-based computations tend to be robust against reordering, and also because the solution for losses protects against duplicates as well.

When an ESP packet is lost, the effect on the computation is one of the following:

- No effect: the computation proceeds correctly to completion in spite of the error (e.g., losing a packet that would have been discarded anyway, like a COMPARE packet that does not contain the extreme value).
- Silent failure: nothing is delivered to the application(s) in the end systems. This result can be detected in the usual way, through a timeout.
- Explicit failure: an instruction aborts at an intermediate node due to the absence of expected ephemeral state (because the state was never established or timed out). In this case the packet that invoked the aborted computation is forwarded with the *Err* bit set and the *Loc* bits cleared.
- Incorrect result: an incomplete or incorrect value is delivered to the application as if it were correct.

Of course, the likelihood of each type of outcome depends on the specific details of the instance. For some computations, an incorrect outcome is unlikely in all cases, while for others, the probability grows with the size of the computation.

Consider the feedback thinning service of Section 3.1.1, for example. Packet loss may affect the number and sequence of values delivered to the destination, but it does not affect the ultimate result unless all packets containing the maximal value are lost. Thus if multiple repetitions of the computation produce the same result it is very likely to be correct.

On the other hand, the two-phase aggregation service described in Section 3.2 will fail silently or produce an incorrect result if even *one* message is lost. For small groups it may be possible to obtain a correct result by simply repeating the computation, but as the number of nodes involved grows, the likelihood of error-free completion drops quickly. We simulated the aggregation computation for a group size of 5000, with 4 randomly-chosen lossy links, each having a loss probability of 10%. Out of 1000 runs, 690 completed successfully (i.e. a result was delivered to the receiving application); of those, only 454 returned the correct answer of 5000. Clearly it is necessary to "build-in" robustness when designing such services.

Our paradigm for making distributed computations robust against loss is based on proactive retransmission. In the rest of this subsection we outline the principles of the approach and apply it to the aggregation computation of Section 3.2 as an example.

4.3.1 Adding Redundancy

Lost ESP packets, like lost TCP packets, are recovered by host retransmission regardless of where in the network they are lost. However, the situation is somewhat more complicated with ESP. One reason is that each ESP packet is (in general) intended to modify the state of all ESP nodes along its path, but a lost packet only affects nodes up to the point where it is dropped. Thus the retransmitted packet should modify only the state of those nodes *after* the point where the original packet was dropped. On the other hand, the packet itself may be modified as it travels through the network; the retransmitted packet should match the original when it arrives at the loss point.

These observations imply that each ESP node must be able to distinguish between original packets and retransmitted duplicates. Moreover, any packet recognized as a duplicate should *not* update the computation state at the node, while the packet itself *should* be updated in the same way as the original packet. Also, duplicates must be forwarded (if forwarding is consistent with the computation state) to ensure that redundancy propagates through the network.

If *all* duplicate packets are forwarded in a tree-structured computation, the redundancy is amplified and implosion may result. Therefore filtering is needed to ensure that redundancy remains at a constant level as packets move up in the tree. We therefore must keep track of, and limit, the number of packets forwarded at each hop, thereby keeping the amount of redundancy constant across the tree.

The foregoing discussion leads to the following general form for instructions in a multiphase, tree-structured computation:

Basic Redundant Instruction Form

```
if the packet is not a duplicate
  record the packet;
  execute the operation;
  update the computation state and packet;
else
  (possibly) update the packet;
increment fwd-count;
if the packet is forwardable and fwd-count ≤ pkt.limit
  forward the packet
else discard the packet
```

Note that it is not necessary for all hosts to originate the same number of packets. The computation may specify an *expected* number of transmissions per host, and each host can transmit transmit with the appropriate probability to achieve that average.

4.3.2 Duplicate Detection

In general the number and identity of nodes originating packets in a computation is not known *a priori*, so we need a means of distinguishing retransmitted packets from original packets without this information. We use *Bloom filters* to solve this problem using a fixed, modest amount of space.

A Bloom filter [1] is a way to record sets of elements using a bitmap of size $m = 2^q$, and k random hash functions $h_0, \ldots, h_{k-1}$. Each element that might be recorded in the Bloom filter has a unique identifier; the hash functions map identifiers to q-bit offsets. An element is recorded in the bitmap by hashing its identifier with each of the k hash functions, and setting the bits at each of the resulting offsets. To test for the presence of the element with identifier i in the set, one checks whether each of the bits at offsets $h_0(i), \ldots, h_{k-1}(i)$ is set in the bitmap. If so, the element is considered to be present; otherwise it is not present.

Bloom filters introduce a tradeoff between the size of the bitmap and the probability of a *false positive*: After a number of elements have been recorded in the bitmap, a new element may be erroneously judged to be present, because all of its bits were set by earlier elements. Given a set of k hash functions $h_0, \ldots h_{k-1}$, we say a set I of identifiers is *collision-free* if, for each $i \in I$,

$$\{h_0(i), \ldots, h_{k-1}(i)\} \not\subseteq \bigcup_{j \in I, j \neq i} \{h_0(j), \ldots, h_{k-1}(j)\}$$

Figure 8 shows the probability, according to this definition,[1] that a randomly-chosen set of identifiers is collision-free. The horizontal axis represents the number of distinct elements being recorded; the curves show the results (obtained by simulation) for various combinations of m and k.

Detection of duplicate packets is implemented by storing the Bloom filter bitmap in the ESS. For simplicity, the following discussion assumes that the Bloom filter bitmap is the size of a single ESS value[2] At the beginning of a computation, each node randomly chooses k offsets from 0 to $m-1$, and sets the corresponding bits in a bitmap (IDmap) the same size as the Bloom filter. Each packet (i.e. instruction) forwarded by that node in that computation carries a tag identifying the Bloom filter, plus this IDmap. To check whether a packet is a duplicate, the receiving node checks whether all the bits set in the IDmap are already set in the stored Bloom filter. If so, the packet is considered a duplicate; otherwise it is considered new, and the IDmap is OR'ed into the Bloom filter bitmap. This method has the advantage of allowing the instruction to be oblivious to the value of k.

[1]This condition is strong, in the sense that it guarantees no false positive will occur regardless of the order in which elements are added. For some sets that do not satisfy this condition, whether a false positive occurs depends on the order in which the elements are added.

[2]It can easily be generalized to larger bitmap sizes by letting a single tag denote a sequence of tags.

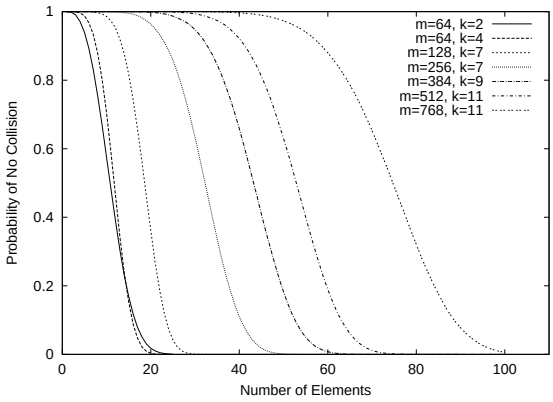

Figure 8: Probability a set of identifiers is collision-free vs. the set size, for various bitmap sizes and numbers of hashes.

4.3.3 Robust Aggregation

To summarize this section, we present two instructions that together implement a robust version of the computation of Section 3.2. The main change is the addition of Bloom filters to each instruction to detect duplicates, and forwarding filters to suppress extra redundancy (note that COUNT already included such a filter). Also, since the Bloom filter has to be present anyway, we use it to keep track of children instead of a counter.

```
RCHLD(pkt p)
  α = get(p.T0)
  if (α is nil) α = 0
  α = α | p.idmap
  put(p.T0, α)
  β = get(p.C)
  if (β is nil) β = 0
  β = β + 1
  put(p.C,β)
  if (β ≤ p.thresh)
    p.idmap = myIDmap
    forward p
  else discard p
```

```
RCOLLECT(pkt p)
  α = get(p.T1)
  if (α is nil) α= 0
  if ((α & p.idmap) != p.idmap)
    α = α | p.idmap; put(p.T1,α)
    ξ = get(p.V)
    if (ξ is nil) ξ = p.val;
    else ξ = ξ ∘ p.val
    put(p.V,ξ)
  β = get(p.T0)
  if (β is nil) abort
  if (α == β)
    ν = get(p.D)
    if (ν is nil) ν = 0
    ν = ν + 1;  put(p.D,ν)
    if (ν ≤ p.thresh)
      p.idmap = myIDmap
      p.val = ξ
      forward p
  else discard p
```

Figure 9: Instructions for robust aggregation

The two instructions are shown in Figure 9. The first, RCHLD, replaces the COUNT instruction (introduced in Section 2.4) and records the identifers in packets received from its children. The instruction takes the following operands: a tag (T_0) identifying the Bloom filter bitmap, the *idmap* of the sender, a tag C identifying the count of forwarded packets, and an immediate threshold *thresh*.

The second instruction, RCOLLECT, is similar to COLLECT, but also checks for duplicate packets. In addition to the operands V, *val*, and "$\circ$" that are used in COLLECT (Section 3.2), it also takes the following operands: a tag T_1 identifying the Bloom filter bitmap for detecting duplicates; an immediate *idmap*, identifying the last node to forward the packet (the same *idmap* used in the RCHLD instruction),

which is used to update value bound to T_1; the tag (T_0) identifying the Bloom filter used in the previous RCHLD instruction; a tag D identifying the count of packets forwarded; and an immediate threshold *thresh* to control the number of duplicated transmissions. The key difference between RCOLLECT and COLLECT is that in RCOLLECT the condition for forwarding is when the two Bloom filters match, rather than when the count is zero.

In a tree-structured computation, the size of the bitmap required and the number of hash functions should be chosen based on the tree node with the maximum degree, so the probability of messages falsely being rejected as duplicates is sufficiently low. The curves in Figure 8 give an idea of the size of bitmap required to achieve this for different numbers packets to be recorded.

We simulated the robust aggregation computation on the same tree used for the simulation of the unreliable computation: 5000 leaves, four randomly-chosen lossy links with loss probabilities of 10%. Using a Bloom filter size of 512 bits (the tree has a maximum branching factor of 40) $k = 11$ hashes, and a forwarding limit of two, 926 of 1000 runs completed successfuly, and of those, 874 obtained the correct result. With a forwarding limit of three—that is, leaves send three copies of each packet, and each packet is forwarded up to three times—all but three runs completed successfully, and 945 runs obtained the correct result.

4.4 Security Considerations

Many of the threats relevant to ESP are common to other network-level services, including IP itself. As with other services, the most effective approach is often to handle security in the application, where it is possible to apply existing end-to-end mechanisms. For example, group feedback values filtered through the network (Section 3.1) can be confirmed at the application layer using group security protocols. However, the special role of routers in providing ESP services, coupled with the need to keep routers and end systems (mostly) oblivious to each others' identities, makes it harder to apply some traditional end-to-end security solutions. Our goal here is not to describe a complete security design for ESP, but rather to show that options exist for applications concerned about security.

We consider three kinds of threats: ESP as a threat to other network applications that do not use it; ESP as a threat to routers that implement it; and threats to applications using ESP.

4.4.1 Attacks using ESP

Because ESP neither duplicates nor spontaneously generates packets—the number of ESP packets leaving any router is at most the number entering—it does not offer any opportunities for new flooding attacks. Only packets carrying ESP headers are processed by ESP; thus ESP is no threat to applications that do not use it. An attacker could cause a non-ESP application's packets to be processed somewhere in the network by inserting a piggybacked ESP header. However, given the ability to modify packets in transit, this is just one of many ways to cause mischief, and ways to protect (or at least detect) such modifications are well-known.

4.4.2 Threats to ESP-Capable Routers

ESP gives routers additional work to do. It is therefore prudent to ask whether that additional workload poses any significant threat to router functionality. For example, could an attacker incapacitate a router by flooding it with ESP packets? Because it is designed to be implemented on line cards and to run at wire speeds, ESP is no worse in this sense than plain old IP processing.

The centralized ESP context (Section 2.2) in a router can potentially be attacked by flooding it with ESP packets. However, this is no different than any other application-level protocol running in a router over UDP or TCP. Moreover, a denial-of-service flooding attack on one of a router's ESP contexts does not affect the others, nor does it affect the router's ability to forward datagrams, except to the extent that the ESP flood consumes bandwidth—which, again, is no different from a "traditional" IP denial-of-service attack. We conclude that the presence of ESP in a router does not introduce any vulnerabilities that are not already present.

4.4.3 Threats Against Users of ESP

The operands of ESP instructions are carried in the clear in each packet. Anyone who can eavesdrop on packets in the network can discover the tags and values occurring in others' computations. If the computation involves sensitive end system information, that information can be compromised. Moreover, given the tags and CID used in an end-to-end computation, an attacker can send packets containing bogus information to cause the computation to be aborted, or to deliver an incorrect result that looks like a correct one.

Clearly this threat can be ameliorated by making eavesdropping and spoofing difficult. For example, groups that want to use ESP securely can set up a virtual private network using IPsec (including the *other* ESP [12]) to ensure that packets are not snooped or forged. This raises the cost of using ESP significantly, but does have the side benefit of providing a way to ensure that paths are routed through ESP-capable routers.

More dynamic cryptography-based approaches may exist. In any case the computational costs of cryptography, coupled with the potentially high administrative costs of establishing and maintaining trust relationships between hosts and infrastructure, give rise to tradeoffs that must be considered very carefully in the context of an extremely lightweight service like ESP. We consider this an important area for future work.

4.5 Route Stability

The ability to leave state at network nodes is only useful if later packets can reliably "find" that state. Some computations using ESP (including most of the examples in Section 3) rely on the fact that successive packets sent to the same destination will follow the same path through the network. We expect that most computations will complete within an interval comparable to the state lifetime. It seems reasonable to expect routes to remain stable over that time. However, load sharing and route flaps may cause problems for some ESP computations—as they do for other services, including TCP [17].

5. IMPLEMENTATION

As a proof-of-concept, we implemented the ESP service on the Intel IXP1200 network processor[3]. We used the

[3]We are also developing a hardware implementation on the Virtex 1000 Field-Programmable Gate Array (FPGA).

Bridalveil evaluation board which contains a core StrongARM processor running at 232 MHz that connects to up to 8 MB of SRAM and 256 MB of SDRAM. The IXP1200 board supports four 100 Mbps Ethernet ports that can be accessed by the processor through a 104 MHz bus. The Intel IXP1200 network processor is designed to support on-board (fast) packet processing via user-loaded software processing modules. The innovation of the the IXP1200 is the six on-chip microengines (μengine) each supporting four hardware threads for parallel processing. Each of the μengines is separately programmable and is supported by a set of hardware instructions specifically designed for packet processing.

Our goal was to utilize the processing power of current network processors to perform at or near wire speeds of 100 Mbps. We focused on the implementation and performance of the ESS, because the high-latency memory accesses to the ESS are the dominant cost of an ESP instruction. In this section we describe the design of the ESS; the next section presents performance results obtained from the IXP1200.

5.1 A Scalable ESS Design

If ESP is to be practical, it is crucial that ephemeral state be scalable and inexpensive to implement. Although conventional *content addressable memories (CAMs)* offer exceptional performance, they do not scale well, being typically relatively small and expensive.

To address this problem, we developed an ESS architecture based on inexpensive commodity memory. Commodity memory is cost effective and offers much larger storage capacity than CAMs. The challenge is minimizing the time required to locate and access (tag,value) pairs in the RAM.

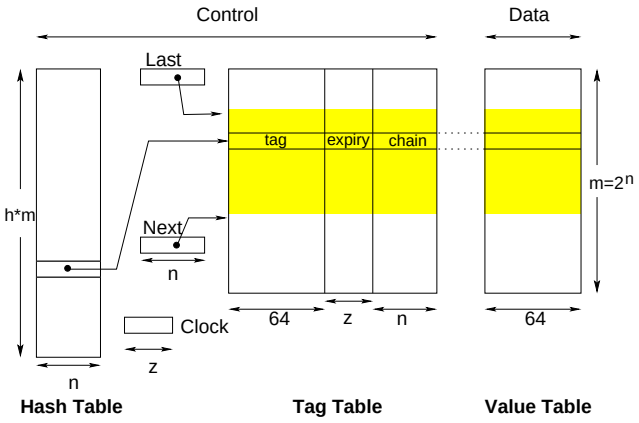

Figure 10: Memory layout of ESS.

Our design for a 2^n-entry store partitions the memory into the three tables shown in Figure 10. The tables may be located in different types of RAM for performance or storage capacity reasons [23] (e.g., fast SRAM vs. large DRAM). The *value table*—64 bits per entry—stores values associated with tags. The remaining two (control) tables, the *hash table* and *tag table*, together implement the associative lookup service.

Hashing is used to reduce the number of memory references needed to locate a tag. Hash collisions are handled via explicit chaining. Pointers into the tag table, rather than the tags themselves, are stored in the hash table to create a level of indirection that allows tags to be stored sequentially in the order they were created, which is also the order they will timeout or expire. The *next* register points to the next available entry, while the *last* register points to the next entry to expire. At any time, the entries from *last* to *next* (modulo 2^n) are "live" and ordered from oldest to youngest. This simplifies the process of removing expired entries and encourages parallelism.

Each tag in the ESS has an entry in the tag table; the associated value is stored at the same index of the value table. Each tag table entry also contains the *expiry time* of the entry (a z bit value), and a *chain* pointer (i.e., an n-bit index of another entry in the tag table). The number of entries in the hash table can be anything, but for efficiency should be at least the size of the tag table. The tag table is the same size as the value table; thus if the value table capacity is 2^n, each hash table entry is n bits wide. A clock register with a resolution of z bits is incremented periodically and is used to calculate the expiry time.

Entries are actively removed by a *cleaner* function, which waits for the clock to equal the expiry field pointed to by *last*. Aggressive removal of timed-out entries reduces the number of bits that have to be stored in the expiry field; lazy removal requires substantially more bits to ensure that the clock doesn't wrap.

The interface to the ESS supports four operations:

handle `find`(*tag*): return a *handle* to the record bound to *tag*. If the *tag* does not exist, return a NULL *handle*.

handle `find_create`(*tag*): this atomic operation checks for the existence of *tag* and creates it if it does not exist. It then returns a *handle* for the record.

status `write`(*handle*,*value*): bind *value* to the *tag* associated with *handle*. Return success or failure.

value `read`(*handle*): return the *value* bound to the *tag* associated with *handle*.

The ESS tables and interface are designed such that a tag need be looked-up only once per ESP instruction. Both `find` and `find_create` map a 64-bit tag to an n-bit handle (i.e., an index into the value table) that can be used for subsequent reads and writes to the tag's value.

5.2 Design Characteristics

Our ESS design has several desirable characteristics. First, it is relatively cheap in terms of space (memory) overhead. At least 128 bits are required per (tag,value) pair; the additional overhead, assuming z-bit timestamps and a hash table to tag table size ratio of h, is $(h+1)n+z$ bits per entry for a 2^n store. Section 6 considers the effect of various h settings.

Second, the design is efficient in time overhead. Given a reasonable hash function and a sufficiently large hash table, only two memory accesses are needed to locate a tag, regardless of how full the tag table is. Moreover, the lookup occurs at most once per instruction. Tag values can then be read or written in a single memory access.

Third, our design requires only a single low resolution clock. Time values can be stored in a small number of bits. For example, assuming 10-second lifetime and a 0.1-second resolution clock, 10-bit time values should more than suffice.

Fourth, the design encourages parallel and pipelined access to the ESS. The `find_create` operation must coordinate with the cleaner when a new entry is allocated, but otherwise they can run independently. Note that the expiration of a tag (and removal by the cleaner) has no effect on instructions that might be accessing the corresponding location in the value table. It is also possible for multiple threads to access the store in parallel. As long as different threads do not access the same tag, there is no need for coordinated access to the data store. If we assume that different computations never use the same tags, the only source of interference is different instructions belonging to the same computation being executed simultaneously. We can prevent this from happening by serializing the execution of packets belonging to the same computation. The Computation ID carried in the ESP packet (Section 2.5) provides the means of achieving this.

6. PERFORMANCE

We implemented ESP processing using a combination of the StrongARM and the μengine processors. To evaluate the per packet processing costs of ESP, we used the MAC-level packet reception and transmission code from the IXP1200 SDK2.0 development system; this code is known to be able to process packets beyond linespeed. We inserted new μengine instructions to detect ESP packets the dispatch them to the core processor for further processing. We used the 64-bit hardware hash instruction to hash the tags carried in the packet. We measured the performance of the system by generating ESP packets and sending them over the 100 Mbps link to the IXP1200.

Our code used the algorithms described earlier for finding, reading, and writing (tag,value) pairs in the ESS. Hash collision were handled through chaining, while new bindings used the *next* pointer and the cleaner used the *last* pointer. A separate thread was scheduled to periodically (e.g. every tenth of a lifetime) check whether the entry pointed to by the *last* pointer had expired. To measure performance when the hash table becomes full, we artificially loaded the table and then measured its performance. As expected, the latency of the find() operation depends on the chain size in the hash table (Table 1). Table 1 shows the cost of the ESS functions for the SRAM implementation when the hash table is essentially empty (i.e., chain lengths of 1). Since the actual access time depends on the length of the chain, we also show the number of memory accesses required as a function of the chain length *cl*. The `find_create`() only takes five more 32-bit access than `find`() because the hash table and the tag/data table must both be modified. The simplicity of ESP instructions means that the majority of the processing costs are attributed to ESS performance. As shown here, our initial numbers indicate that the ESS is capable of operating near or at linespeeds.

We also measured performance using different size hash tables. In particular, we tried h ratios ranging from 0.5 to 2 (h = *hash table size/tag table size*). Fig. 11 shows the effect that different hash table and tag table sizes have on the find() performance. Our results show that even when the store is full, the average access time remains constant if the hash table has twice as many entries as the data table. Even when the hash table is only half the size of the data table, the average access time still remains below 3 μsec per packet.

ESS calls	Time using SRAM	Number of 32-bit Memory Accesses
find()	0.5 μsec	$5 \times cl$
find_create()	1.4 μsec	$5 + 5 \times cl$
read()	0.1 μsec	2
write()	0.1 μsec	2

Table 1: Access times for ESS operations. *cl=Chain Length*

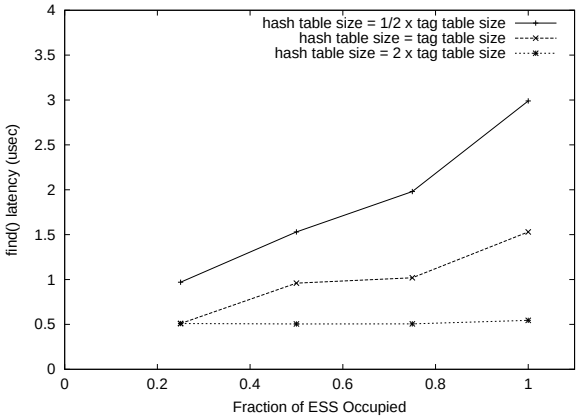

Figure 11: The average find() function latency when fraction of the ESS is filled

The IXP1200 provides different access capability to SRAM and SDRAM memory. The SRAM has a 32-bit wide bus and runs at the core clock frequency (232MHz). The SDRAM has a 64-bit wide bus and operates at half of the SRAM frequency (116MHz). The SRAM's fast access is intended for crucial router state such as the routing table. The SDRAM's wider bus is designed for packet I/O and buffering. Although smaller in size, the IXP1200's SRAM offers the performance we desired for ESS storage and with up to 8 MB of space was sufficiently large to support a reasonable-sized prototype ESS. However, for comparison we measured the performance of our ESS design using the SRAM and then the SDRAM.

	Avg. Latency (μsec)		
Functions	SRAM	SDRAM	SRAM & SDRAM
COMPARE	3.98	6.49	4.99
COUNT	2.62	3.54	3.47

Table 2: Latency of various ESS operations with ESS located in SRAM, SDRAM or both

Table 2 shows the latency ESP instructions incur. These measurements are taken on the StrongARM processor and factor out the cost of transferring data to/from the core, so they are representative of what we expect to see if they were executed at the μengine level.

The table also shows the instruction latency when ESS is implemented in different levels of the memory hierarchy. As we expected, the SRAM implementation performed the best. The SDRAM implementation did not perform as well since SDRAM data is transferred in 64-bit blocks (32-bit for

SRAM) at half of the SRAM frequency. In many cases, the ESS access is to data smaller than 64 bits. To fully utilize the speed of the SRAM, as well as the size of the SDRAM, we placed part of the ESS on the SRAM (the hash table) and the other part on the SDRAM (the tag table). Because access to the tag table usually involves larger than 32-bit reads, SDRAM seems to be more suitable for the tag table. Although the performance is not as good as the SRAM-only implementation, it trades off the benefit of a larger ESS for slightly slower access times.

	Throughput (Kpps)		
Functions	SRAM	SDRAM	SRAM & SDRAM
COMPARE	232	146	188
COUNT	340	259	263

Table 3: Estimated Throughput of ESP instructions

We also measured certain costs of processing on the μengine, namely the ESP packet classification and the hardware hash operations performed on tags. We used the cycle-accurate hardware emulator provided by the IXP1200. The μengine processing times averaged 75 cycles per packet (about 0.32 microseconds). Based on this number and the latency information from Table 2, the estimated packet rates for various ESP instructions in terms of number of minimum-sized packets per second processed are shown in Table 3.

We are currently pushing the code completely into the μengines in order to avoid the performance penalty of transferring data to the core for processing. Our prototype implementation did not take advantage of some μengine features that would almost certainly increase the effective throughput, including multiprocessing, hardware multithreading, and optimized memory access. Nevertheless, our preliminary measurements show that ESP processing can support bandwidths of up to 169 Mbps, even going through the StrongARM core.

7. RELATED WORK

A number of research efforts have aimed to place some form of extended function or programmability in the network. Compared to ESP, these generally either target more specific end-to-end services, or provide "heavyweight" computational capabilities offering greater computational power.

The PGM [5] and BCFS [30] protocols extend router functionality specifically to support end-to-end reliable multicast. Generic Router Assist (GRA) [2] is a set of capabilities intended to generalize these protocols to support other services. These protocols use soft-state techniques to maintain and process state information for multicast sessions. This state information can be used for, among other things, subcasting and NACK suppression. In its current state, GRA is more narrowly focused than ESP, and does not share its scalability characteristics.

Protocol boosters [4] was an early effort to place support for end-to-end services in the network. Unlike ESP, protocol boosters could operate on all parts of the packet, and could generate packets in the middle of the network. Among the example applications was on-the-fly addition of payload redundancy for forward error correction.

Active networks offer an alternative model of network programmability. For example, ANTS [29] provides users with more or less complete control of packet processing, by associating a program to each packet, or *capsule*. ANTS also allows packets to deposit and retrieve information in an associative store for a limited time; however, the time bound is not fixed as in ESP, but depends on other factors such as distance traveled by the packet. Overall, the design goals of ANTS are oriented more toward flexibility than those of ESP.

SNAP [13] allows packets to carry small programs expressed in a restrictive programming language that ensures that the computing resources consumed by a packet are strictly bounded. SNAP's focus is more on computing resources than storage, however. It does not explicitly provide for packets to exchange information in the network.

Some active network services target specific needs, especially for group applications. The *concast* service [3] provides a many-to-one communication channel in which information sent to a common destination from the group is combined in routers along the way, according to a merge specification supplied by the receiver. Compared to the simple packet elimination capabilities possible with ESP, concast can perform more complicated (heavyweight) packet manipulation, possibly generating completely new and different sized payloads.

The "Smart Packets"project from BBN [21] applies active network technology to the problem of network management; it also focused on a small set of primitives to be invoked by packets. However, the designers of the Smart Packets mechanism explicitly ruled out having packets leave state at a node; thus while computations may involve multiple nodes they cannot involve multiple packets.

8. CONCLUSIONS

We have proposed a new approach to placing support for end-to-end services inside routers. Ephemeral state processing is a building-block service designed under minimal assumptions about applications. A key aspect of our design is the use of an associative store with a *fixed lifetime*. This constraint benefits the network by ensuring that state resources are freed at the same rate they are allocated; this makes it possible to build stores that can process allocation requests at "wire speed". It benefits the user (service designer) by ensuring that all trace of a computation disappears from the network within a fixed time.

We have designed ESP to be implemented on, or near, the fast path of a modern router, and to have scalability and robustness characteristics similar to those of the Internet Protocol. Our experience with an initial prototype implementation on the IXP1200 network processor suggests that ESP should easily be able to process packets at line speeds in excess of hundreds of megabits per second. By exploiting available parallelism, pipelining, and special-purpose hardware, we expect performance can be increased substantially beyond that.

Our current design is based on a small set of parameterized instructions that suffice for a broad set of end-to-end services that are primarily of an auxiliary nature. However, we expect that as new ways to use ESP are found, the need for new instructions will arise. This brings up the question of how to add new instructions once ESP is deployed. One possibility would be to allow packets to carry "microcode"—perhaps coded in a restricted language like SNAP [13]—which would be interpreted at each node. However, it is not

at all clear that such flexibility is necessary. An alternative would be to extend the instruction set through a formal standardization process. This would allow the flexibility and simplicity of the instruction set to be carefully maintained. In either case, allowing the instruction set to be extended provides for the ESP service to evolve over time.

In addition to various engineering choices that should be evaluated through experience—the particular value of τ, the size of stored values, and so forth—we find some further issues worthy of future study. One is dynamic adaptation of robust ESP computations to network conditions, so that they use network resources (bandwidth and storage) as efficiently as possible. Another is the development of a security architecture for ESP that provides enhanced protection while preserving the lightweight nature of the service.

Although we are optimistic that ESP is both sound in design and useful in application, it represents just one point on a spectrum of approaches to extending network functionality. We hope others will both consider additional ways to use ESP and also explore different parts of that spectrum.

Acknowledgements

This paper has benefited greatly from the constructive comments of the anonymous referees and especially our shepherd, David Wetherall. The authors also acknowledge with thanks the technical support of the IXP1200 group at Intel Architecture Labs.

9. REFERENCES

[1] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] Brad Cain, Tony Speakman, and Don Towsley. Generic router assist (GRA) for multicast transport protocols, June 2002. Internet Draft (work in progress).

[3] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and implementation of an active network service. *IEEE Journal on Selected Areas of Communications*, 19(3):426–437, March 2001.

[4] D.C. Feldmeier et al. Protocol boosters. *IEEE Journal on Selected Areas of Communications*, 16(3):437–444, April 1998.

[5] T. Speakman et al. PGM Reliable Transport Protocol Specification. RFC 3208, December 2001.

[6] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *ACM SIGCOMM*, Cambridge, MA, September 1995.

[7] Michael Hicks, Pankaj Kakkar, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. 1998. International Conference on Functional Programming.

[8] H. Holbrook and B. Cain. Source-specific multicast for IP, November 2001. Internet Draft (work in progress).

[9] Hugh W. Holbrook and David R. Cheriton. IP Multicast Channels: EXPRESS Support for Large-Scale Single Source Applications. In *ACM SIGCOMM*, Cambridge, MA, August 1999.

[10] John Janotti. Network Layer Support for Overlay Networks. In *IEEE OpenArch*, New York, June 2002.

[11] D. Katz. IP router alert option, February 1997. RFC 2113.

[12] S. Kent and R. Atkinson. IP encapsulating security payload (ESP), November 1998. RFC 2406.

[13] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical Programmable Packets. In *IEEE INFOCOM*, Anchorage, AK, April 2001.

[14] Christos Papadopoulos, Guru Parulkar, and George Varghese. An Error Control Scheme for Large-Scale Multicast Applications. In *Proceedings of the INFOCOM '98 Conference*, 1998.

[15] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *ACM SIGCOMM*, San Diego, CA, August 2001.

[16] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *The IEEE Journal on Selected Areas of Communication*, 1996. (see also the Proceedings of IEEE INFOCOM'96).

[17] Vern Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.

[18] K. K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP, September 2001. RFC 3168.

[19] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[20] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-time Applications, January 1996. RFC-1889.

[21] Beverly Schwartz, Wenyi Zhou, Alden Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart Packets for Active Networks. *ACM Transactions on Computer Systems*, 18(1), February 2000.

[22] Jonathan Shapiro, Jim Kurose, Don Towsley, and Stephen Zabele. Topology discovery service for router-assisted multicast transport. In *IEEE OpenArch 2002*, New York, June 2002.

[23] S. Sikka and G. Varghese. Memory-Efficient State Lookups with Fast Updates. In *ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[24] A.C. Snoeren, C.E. Jones, F. Tchakountio, S.T. Kent, and W.T. Strayer. Hash-Based IP Traceback. In *ACM SIGCOMM*, San Diego, CA, August 2001.

[25] Ion Stoica, Dan Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *SIGCOMM 2002*, Pittsburg, PA, August 2002.

[26] Ion Stoica, T.S. Eugene Ng, and Hui Zhang. REUNITE: A Recursive Unicast Approach to Multicast. In *IEEE INFOCOM*, pages 1644–1653, Tel-Aviv, Israel, March 2000.

[27] Su Wen, James Griffioen, and Kenneth Calvert. Building Multicast Services from Unicast Forwarding and Ephemeral State. *Computer Networks*, 38(3):327–345, February 2002.

[28] Su Wen, James Griffioen, and Kenneth Calvert. CALM: Congestion-Aware Layered Multicast. In *IEEE OpenArch*, New York, June 2002.

[29] David Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[30] Koichi Yano and Steven McCanne. The Breadcrumb Forwarding Service: A Synthesis of PGM and EXPRESS to Improve and Simplify Global IP Multicast. *ACM Computer Communication Review*, 30(2), April 2000.