ActiveCast: Toward Application-Friendly Active Network Services*

M. Bond, K. Calvert, J. Griffioen, B. Mullins, S. Natarajan,
L. Poutievski, A. Sehgal, S. Venkatraman, S. Wen
Department of Computer Science
University of Kentucky, Lexington, KY 40506

E. Zegura, Y. Chae
College of Computing
Georgia Institute of Technology, Atlanta, GA 30332

Abstract

The next step in the evolution of active networks—one that will support radical new uses of the network and increased scalability—is to package the power of a programmable network platform into customizable active services that are easy for applications to use. The Activecast project has developed and is evaluating a set of active services that will not only enhance the "application-friendliness" of active networks, but will also improve the scalability and usability of networks in general.

This paper discusses the challanges of programming active networks and then presents four new active network services, PAMcast, Concast, ESP, *and* LWP, *that simplify the task of programming active networks.* PAMcast *services allow messages to be sent to any node(s) satisfying a set of user-supplied selection criteria. The* Concast *service provides the logical inverse of multicast, gathering and merging data from a set of senders. Finally, the ESP and LWP services provide extremely lightweight building-blocks on which additional higher-level semantic services can be constructed. For each service, we describe the service abstraction, the ways in which users can customize the service, and its ease of use (i.e., how the customized service is automatically distributed across and "programmed" into the network on the user's behalf). We also present results from simulation models and actual implementations of the new services that demonstrate the scalability and performance of the services.*

*This work sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0514; and by the National Science Foundation under grant numbers EPS-9874764 and EIA-0101242. Views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

1 Introduction

Active networks support dynamic generalization of network services by injecting code and/or policy into the shared nodes of the network. Several active networking projects have focused on *platforms* and enabling technology such as programming languages and lower-level architectural features [32, 13, 3, 1]. These projects have achieved significant and interesting results in terms of expressive power, flexibility and security. A few other projects have applied active network technology to solve *application-specific* problems, such as intrusion detection and response, or interest-filtering in distributed simulation [28, 35]. However, there has been relatively little work done to bridge these two programming extremes. To improve the *useability* of active networks, "application-friendly"APIs for high-level customizable services are needed. These services must hide the complexity of the network but yet allow applications to conveniently specify the processing/handling they would like their data to receive.

The Activecast project aims to develop active network services that are convenient to use and provide tangible benefits to applications. This paper describes several such services: *PAMcast*, a programmable form of anycast that uses application-specific criteria to select one (or more) destinations from a set of possible destinations; *Concast*, a many-to-one channel that can be programmed to merge messages from multiple senders (enroute to the same receiver) in the network; *Ephemeral State Processing (ESP)*, an ultra-lightweight form of active networking designed to support auxiliary computations in the network; and *LightWeight Processing modules (LWP)*, a paradigm for deployment of per-user services in the network. The first two are packet delivery services that offer specialized programming interfaces. The latter two services work together in a novel paradigm to assist applications in the deployment of user-specific functionality (i.e., ESP can determine the nodes in the topology where lightweight processing modules can be deployed directly by the user).

Our services represent a middle point between general-purpose, application-independent platforms (offered by EEs and Node Operating Systems) and application-specific solutions (such as interest filtering [35] or reliable data distribution [24]). They are designed to provide high-level customizable services that applications, particularly large group applications, can use to perform topology-sensitive processing without knowing the details of the network. The services hide the complexity of active networks and achieve scalability via features such as anonymity (e.g., programmable communication with unknown end-systems over unknown network routers), automatic code deployment (to only "the right nodes"), simplified router state management, and enforcable security and admission control. Concast, PAMcast, and LWP are EE-agnostic—they could be implemented on any of the currently extant EE platforms—while ESP should be regarded as a special-purpose EE. Our services are also mostly also backward-compatible, in that they could be deployed incrementally in the Internet.

The rest of this paper is organized as follows. In the next section we outline some principles we believe are key to designing scalable, usable services. We then describe the PAMcast, Concast, ESP, and LWP services, highlighting their embodiment of those principles and giving example uses of the services. Section 7 concludes the paper.

2. Challenges for Application-Friendliness

To bridge the gap between general-purpose programmable platforms and application-specific solutions, active networks should support *application-friendly* services: high-level services that improve the *usability* and *scalability* of active networks.

Usability refers to the amount of effort required by the application programmer to make use of the service. It seems clear that end users are no more likely to program the network than they are to program their home computers. To be attractive, active services should expose *only the details that are relevant to the application*. For example, services that require the application programmer to explicitly identify many nodes, participants, or channels, or to know the specific topology of any significant part of the network, are inherently *difficult to program* and *do not scale*.

Scalability is important in two ways. First, an active service should support applications involving large numbers of end systems and network nodes. It is widely recognized that large-scale group applications are impractical without some form of network support. Although the active code needs to be deployed and run on a large number of routers and end-systems, the application should not be required to know how many or which nodes are involved in the computation. Second, the system should scale in terms of the number of simultaneous active services that can be supported. Because active services consume router state and processing cycles, it is important that the service regulate/police the resources consumed by the application using the active service so that the scalability is maximized and denial-of-service (either intentionally or accidentally) is prevented. Resource management and admission control add complexity in the form of policies governing which users are allowed to invoke which services. The admission control mechanism itself must have resource bounds, lest a denial-of-service attack be mounted by saturating the admission control mechanism.

The following highlights characteristics of services that we believe are necessary for usability and scalability:

Anonymity: Anonymity improves scalability both in terms of network state maintained and ease of programming. Services that distinguish among users (or packets/nodes) require (explicit or implicit) *state* and/or *policies* that define how each user (or packet/node) is treated. Services that deal with "groups" of users/packets/nodes are also easier to program. For example, it is more convenient to transmit a single IP multicast packet than N unicast packets. Similarly, treating all packets in a class or group the same reduces network state and packet processing overhead, which is important if nodes must process packets at line speed.

Locality and Automatic Deployment: Deploying flow-specific state and/or processing at every node along a flow's path does not scale well. Flow state and processing should only be carried out where needed. Services that automatically determine nodes where code is needed and deploy code at those points have better usability and scalability than services that require the application to do this. Moreover, determination of whether processing is needed at a node should be done periodically, not per-packet.

Specialization: A narrow but customizable programming interface is ideal. Application-designers do not want to deal with complex distributed algorithms. The service should provide a restricted interface, with support functionality (e.g. code deployment) hidden from the user.

Automatic State Management: Any "interesting" active network service requires network state. Requiring end-systems to manage/police this state is not desirable or scalable. State should be set up, policed, and destroyed by the service, not the application.

Security and Authentication: Complex trust models in which packets, intermediate nodes, etc must be trusted and authenticated are difficult to make secure. Services that use simple trust models (e.g., only trusting end-systems) are easier to implement and require less overhead.

Best Effort: Best-effort services may place additional burdens on the end systems, forcing them to recover from packet loss. However, such services exhibit better scalability and soft-state techniques can be used to help manage network state.

The following sections present four active network services designed to ease the task of programming active networks (usability) and improve scalability.

3 PAMcast

PAMcast is based on the traditional anycast service, which delivers a packet from a sender to *any one* in a set of receivers. The realization of such a service has been considered at the network layer, where the receiver is selected based on closest hop count [19] or closest AS count [16]. Using hop count or AS count as the selection criteria aims to reduce network resource usage. However it is clear that hop-count-oriented selection is limited and does not meet the needs of diverse applications.

One can envision at least two ways to generalize the traditional anycast service. First, one might provide more flexibility in the specification of how the receiver is selected. Several projects have pursued this direction using application-layer realizations of anycasting (e.g., SPAND [27] and ALAS [25]). Second, one might provide more flexibility in the *number* of receivers that are selected, allowing more than just one to receive a packet. If the set of receivers has size n, selecting any one member corresponds to traditional anycast service, while selecting any n corresponds to traditional multicast service. Values between 1 and n represent a new form of service (sometimes called partial multicast).

We are exploring both forms of generalization, targetting a service that allows flexibility in both the number and method for selecting receivers from a set. A more complete description of our work can be found in [10].

3.1 PAMcast Service Overview and Examples

We propose a new packet delivery service — programmable any-multicast or PAMcast — which generalizes both anycast and multicast services, by providing for delivery to any m out of n group members, $1 \leq m \leq n$. Such a service has applicability for a wide range of applications. For example:

- **Fault tolerant repositories.** Consider the problem of storing a data item in a repository. For fault tolerance, the repository service offers a number of geographically diverse storage locations. One might PAMcast the data item to m of the n storage locations, with m selected based on the importance of the data, the cost of storage in multiple locations, and the perceived reliability of the network and storage servers. The choice of the particular m locations might attempt to balance the load over time.

- **Parallel cache queries.** Suppose a group of caches store data items. A client might PAMcast a query to m caches in the group, in the hope that at least one has the desired item. The choice of m must balance the penalty associated with not finding the item with the overhead associated with query to and delivery from multiple caches. It should also take into account the probability that each cache has the desired item; when an item is more densely replicated, a smaller value for m can be used [9].

- **Parallel download.** Suppose multiple servers have the same content. A client might PAMcast queries to m of the servers, requesting that each transmit a portion of a particular file. Parallel downloads have been shown to improve client response time over single-server downloads [2]. The value of m might depend on the file size; the particular m servers might be chosen to be relatively close to the client.

As the sample applications indicate, some additional control over the delivery (beyond the size m) is desirable. Most generally, one can envision that each packet contains a program (or reference to a program) that controls how the m receivers are selected. We consider the implementation of several specific *modes* of delivery, to explore what is possible with limited state and computation at the routers. Thus, the PAMcast service is programmable in two dimensions: the number m of receivers and the mode of selection.

3.2 PAMcast Architecture

A naive implementation of the PAMcast service would transmit a packet to all n group members (as in multicast), then filter at the receivers to deliver to m members. Such an approach, however, makes poor use of network resources when $m << n$. It also must address the problem of determining filters that select m. An alternative approach is to construct independent multicast groups, one for each value of m. However, the number of groups is potentially very large (2^n if all possible groups of all sizes are supported).

We design an architecture for scalable realization of the service using selective copying at branch points in a tree that includes all n receivers. The architecture can be implemented either within an application-layer overlay or within network routers, though we discuss only the router implementation in this paper. If implemented in routers, partial deployment is feasible, with tunneling between capable routers. The basic service is *best-effort* in the sense that it cannot guarantee delivery to exactly m receivers.

Our architecture is based on a shared tree. We use the term "core" to denote the root of the shared tree. The tree-based architecture provides a scalable means to deliver a specified number of copies of a message to group members. Unlike a multicast tree, the PAMcast tree has an additional attribute, *group size*, maintained on a per group and per tree link basis. The group size denotes the number of downstream group members that are reachable through the link, where "downstream" means "in the direction away from the root".

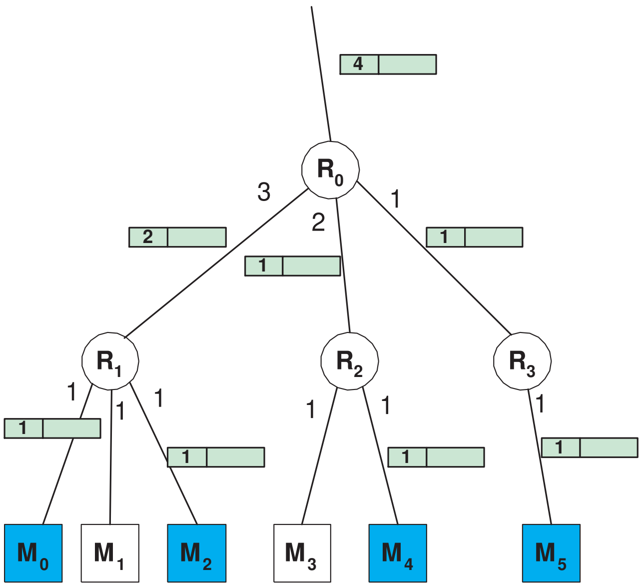

Figure 1. Message delivery on a PAMcast tree

Figure 1 shows how messages are delivered through a PAMcast tree. PAMcast messages have three header fields: *group address*, *degree* and *mode*. The degree field is the only one shown in the figure, since it is the only one modified during transmission. The *group address* field indicates the target group of members to which the message should be delivered. The *degree* field indicates the target number of group members that are supposed to receive the message. The *mode* field indicates how the copies are made, either by specifying a built-in method or providing a reference to a program. End applications specify the three attributes on per-message basis according to their specific needs.

When an application sends a PAMcast message, the message is first routed to the core router of the group. Upon receiving a PAMcast message, the core router identifies which delivery mode is used and what the degree of the message is. Using the mode and the degree information, the core router determines i) a subset of incident tree links through which duplicated messages are to be routed and ii) the degree of each duplicated message. The mode of the message controls how to select the subset of the tree links and the distribution of degree among the selected tree links. The above operations are repeated on each tree router while PAMcast messages are traveling along the tree.

For PAMcast tree management, we propose a PAMcast Group Membership Protocol (PGMP), which is similar to Internet Group Membership Protocol (IGMP). PGMP has three control messages, *group join*, *group leave*, and *group refresh*. There are two main differences between IGMP and PGMP in processing the control messages. First, PGMP messages are routed all the way to the core router. In IGMP, however, messages are terminated at the first router that is part of a delivery tree for the group. The purpose of relaying the PGMP messages up to the core router is to update the group size attributes on every tree links along the path from the joining host to the core router. Second, PGMP messages contain a *group size* field. The group size field of an incoming control message represents the total number of downstream group members reachable through the incoming tree link for the control message. A router that receives a PGMP message will use the group size field to set the group size value for the incoming link. Each member of the PAMcast group will periodically send a refresh message to reinstantiate the group size attribute along the path to the core.

3.3 PAMcast Performance Evaluation

We evaluate the performance of two specific methods for selecting the m receivers:

- **Balanced Mode.** The goal of this mode is to achieve equal distribution of messages over the set of group members. The rationales for the balanced mode are i) increased load-balancing among group members, ii) increased fault-tolerance to link or node failures by avoiding a concentration of messages on group members that share the same tree links/nodes, and iii) random selection of representative group members that are located sparsely on the underlying network.

  We implement the balanced mode using a weighted counter (WC) for each tree link, which is similar to the virtual clock [36, 12] for weighted fair queueing. The weighted counter keeps track of the total degree of the messages passing through the tree link, which is normalized to the number of downstream group members. The load-balancing using the weighted counters is accomplished by balancing the counters of the tree links.

- **Closest Mode**

  The *closest* mode delivers a message to the k group members that are closest (in hop count) to the core router. The underlying rationale of the closest mode is to reduce the network latency and the total network bandwidth usage by selecting the m-closest group members from the core router. We consider two different approaches to the closest mode. First, a deterministic closest mechanism delivers messages to precisely the closest group members. The deterministic mechanism requires that a router should maintain per-member distance information for all downstream group members. This can be achieved by adding one more field, hop-distance, into PGMP messages. Upon receiving a PGMP message, a router collects the distance information of the sending group member, increases the hop-distance by one and passes the message to the parent router as usual. The state required by the deterministic mechanism is linear in the number of downstream group members, potentially prohibitive for large and concentrated groups.

  Alternatively, a probabilistic closest mechanism delivers messages to the closest group members with high probability (that is, with high probability the members selected are the closest to the core). This mechanism uses a constant amount of state per on-tree router. We use a degree distribution mechanism based on the Chebychev inequality for the probabilistic closest mode.

To analyze performance, we used the ns-2 [29] simulation package to model the PAMcast service. For our study, we used five different random topologies of 100 nodes generated by GT-ITM[7]. The simulation model consists of two separate modules: a data forwarding module and a tree management module. The data forwarding module at each router handles PAMcast messages and provides appropriate degree distribution and message copying/forwarding depending on the delivery mode of messages. The tree management module implements the PGMP protocol.

What follows are some sample results; for more extensive evaluation, see [10].

First, Figure 2 shows how the WC algorithm works when new members join a PAMcast group. At the beginning, members m1, m2 and m3 join the group. During $0 \leq t < 3000$, the three members receive the equal number of messages. At $t = 3000$, three new members m4, m5 and m6 join the group, resulting in the total six group members. During $3000 \leq t < 6000$, the six members equally receive the messages. WC algorithm guarantees that the new three group members receive messages with the same rate of the previously joined members. At $t = 6000$, other three members m7, m8, m9 join the group. During $t > 6000$, all the nine group members receive the same number of messages. In brief, Figure 2 shows that the WC algorithm works well with the member join activities.

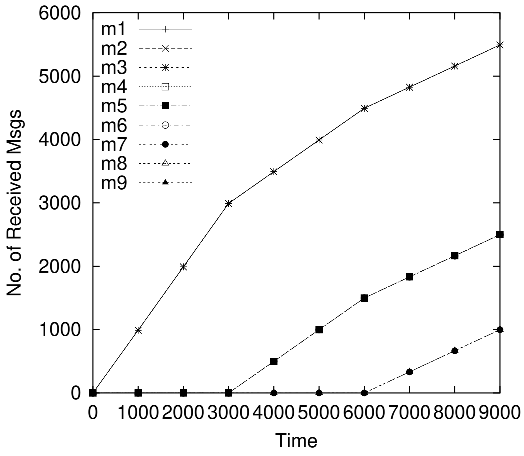

Figure 2. Member Join in WC

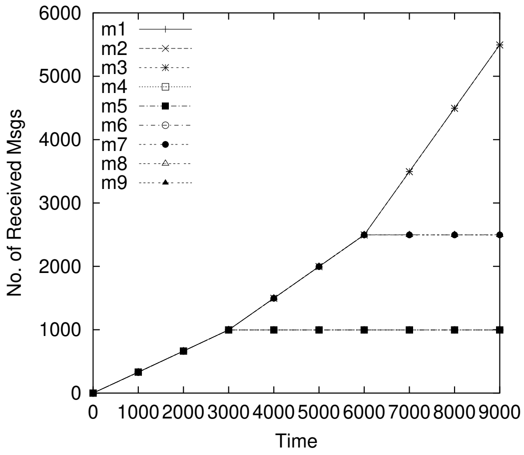

Figure 3. Member Leave in WC

Next, Figure 3 shows how the WC algorithm works when members leave the PAMcast group. The WC algorithm shows similar performance with the member join case. At the beginning nine group members join the group. At $t = 3000$, three members m4, m5 and m6 leave the group, resulting in the total six group members. During $3000 \leq t < 6000$, the message delivery rate at each group member increases accordingly, compared to that of $t < 3000$. Figures 2 and 3 have shown the load-balancing property of the WC algorithm is well-maintained with group join/leave activities.

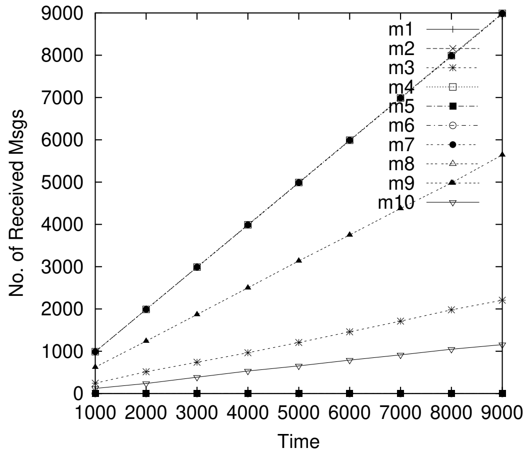

Figure 4. Probabilistic closest with m = 4

Table 1. Hop distance of group members in a tree for Figure4

Hop distance	Group Members
2	m4 and m7
3	m8
4	m3, m9 and m10
5	m1, m2, m5 and m6

Figure 4 shows how PAMcast messages are delivered in the closest mode. The total group size is ten and the target degree of PAMcast messages is four. In addition, Table 1 shows the hop distance distribution of group members on the PAMcast tree for Figure 4. Since the target degree is four, in an ideal closest mode, members m4, m7 and m8 always receive messages and one of the hop-distance four members, m3, m9 and m10, would receive remaining messages. The probabilistic closest always delivers messages to member m4, m7, and m8 as depicted in Figure 4, which is same with the ideal closest. Among the hop-distance four members, preference goes in the order of m9, m3 and m10. The remaining group members receive no messages at all.

Additional performance evaluation can be found in [10]. For example, we consider the effect of stale group size information on the performance of the balanced mode. We also consider the ability of the probabilistic closest algorithm to reach the closest receivers.

3.4 Ongoing Work

Traditional anycast service has received much attention because of its applicability for selection from a set of replicated servers. As services become more complex and involve distributed groups of servers, we envision that the ability to select a subset of servers (as supported by PAMcast) will be of increasing interest. Within our own work, we have explored the use of PAMcast in a multi-media stream caching environment, to support the distribution of variable numbers of copies of objects into independent caches [9]. The number of copies depends on an estimate of the popularity of the object. The balanced mode is most appropriate, since a global goal is to distribute the contents evenly over the set of caches.

Several variations on the basic service may broaden the applicability of PAMcast. We have begun consideration of both a reliable PAMcast and a PAMcast suitable for streaming uses (i.e., involving delivery of multiple packets). We are also considering the implementation of PAMcast within the CANEs execution environment over the ABone.

4 Concast

The IP multicast communication paradigm lets a single address represent a set of nodes, namely the set containing all nodes that are interested in receiving messages sent to that address. This multicast *abstraction* is truly scalable in that it minimizes network traffic and simplifies the programming model: a sender can treat any number of receivers as a single entity, without knowing about individuals or even how many receivers there are. To maintain this abstraction for information flow in the other direction requires an analogous many-to-one channel: one that enables a receiver to communicate with an arbitrary number of senders as if they were a single entity. In the absence of such a channel, group applications are forced to implement ad-hoc unicast-based solutions that limit scalability [34, 20, 22]. The goal of our *concast* service is to provide such a channel.

Concast embodies the principles of anonymity and specialization. It preserves anonymity when multicast receivers need to send feedback to the multicast sender. It allows the active application programmer to supply a *merge specification*, which describes how to combine or aggregate messages inside the network, as they travel from the many (senders) to the one (receiver). Programmability is crucial to the utility of concast, because the nature of the merging operation depends upon the semantics of the application's messages—unlike the the inverse operation provided by multicast (i.e. "duplication"), which is useful to many applications. At the same time, the merge specification fits into a specialized interface designed specifically to limit the resources available for processing each packet. For example, the merge specification template ensures that at most one packet is forwarded for each incoming packet. The remainder of this section provides a brief overview of our service and its implementation. For a more detailed description of the concast service, the interested reader is referred to [4, 6, 8].

4.1 The Concast Service and Programming Interface

Concast messages are sent from a *group of senders* to a single receiver, R. A *concast flow* is associated with a pair (G, R), where G is the group identifier. The packets delivered to R are a function of the packets sent by the members of G. Concast packets are ordinary IP datagrams with R in the destination field and G in the IP options field (in a *Concast ID* option). The IP source address carries the unicast address of the last concast capable router that processed the packet. Concast capable routers intercept and divert for processing all packets that use the Concast ID option.

getTag(m)**:** a *tag extraction* function returning a hash or key identifying the message. Messages m and m' are eligible for merging iff *getTag*(m) = *getTag*(m').

merge(s, m, f)**:** the function that combines messages together. The first parameter is the current *merge state* (i.e., information representing messages that have already been processed). The second parameter is the incoming message to be merged into the state s. The third parameter is the "flow state block" containing information about the concast flow to which m belongs.

done(s)**:** the predicate that checks s, the current merge state, and decides whether a message should be constructed (by calling *buildMsg*) and forwarded to the receiver.

buildMsg(s)**:** the *message construction* function, which takes the current merge state, s, and returns the payload to be forwarded toward the receiver, along with the updated state.

Figure 5. Merge Specification Methods.

The concast abstraction allows applications to specify the mapping from sent messages to delivered message(s), which is carried out in the concast-capable routers along the paths from senders to R. This mapping is called the *merge specification*; it controls (1) the relationship between the payloads of sent and received datagrams, (2) the timing of message delivery, and (3) packet identification (i.e., which packets are merged together). The merge specification is defined in terms of four methods (see Figure 5), which are invoked from a generic packet-processing loop (see Figure 6) that is the same for each flow. The packet-processing loop is invoked for each incoming packet belonging to the flow.

The concast framework allows users to supply the definitions of these functions using a (restricted) mobile-code-language. These definitions are supplied by the application

```
ProcessPkt(Receiver R, Group G, IPDatagram m) {
    FlowStateBlock fsb;
    DECTag t;
    MergeStateBlock s;

    fsb = LOOKUP_FLOW(R,G);
    if (fsb != NULL) {
       t = fsb.getTag(m);
       s = GET_MERGE_STATE(fsb,t);
       s = UPDATE_TTL(s,m);
       s = fsb.merge(s,m,fsb);
       if (fsb.done(s)) {
         (s,m) = fsb.buildMsg(s);
         FORWARD_DG(fsb,s,m);
       }
       PUT_MERGE_STATE(fsb,s,t);
    }
  }
```

Figure 6. Network per-packet processing.

receiver and pulled down through the network to the appropriate nodes via the *Concast Signaling Protocol* as senders join the group. For each concast flow (G, R), each concast-enabled router on the regular unicast path from some member of G to R maintains the following information:

Upstream Neighbor List (UNL): Each item in the UNL represents a concast-capable router or sender that forwards packets toward R along a path that goes through this node. Each entry in the UNL list contains a node identifier and a softstate timer. The softstate timer decreases monotonically over time; if it reaches zero, the entry is removed. Thus each concast-capable node periodically sends a (unicast) message downstream to refresh its soft state. When a flow's UNL contains fewer than two neighbors, packets belonging to the flow are forwarded without being processed.

Merge Specification: the definitions of the *getTag()*, *merge()*, *done()*, and *buildMsg()* functions.

Per-message State List: A list of in-progress "merge states" indexed by message tags.

Incoming packets are classified into flows based on (G, R). If no FSB is found for a packet, it is discarded. Otherwise the *getTag* function is obtained from the FSB and applied to the packet to obtain a tag that identifies the equivalence class of packets to which the packet belongs. The *merge* function is invoked on the current merge state for the tag (i.e., the merged state from all messages already received) to compute the new merge state. The *done* predicate then determines indicates whether the merge operation is complete. If so, *buildMsg* is invoked to construct an outgoing message from the merged state that is forwarded toward the receiver.

4.2 Application Scenarios and Benefits

Merging packet flows inside the network offers several benefits. Some have already been described when we discussed the principle of anonymity, above. Here we present some other example benefits.

Implosion Avoidance: *Packet implosion* occurs when a large number of packets must be received and processed by the destination over a short interval [21]. As the number of senders increase, buffer space requirements at the receiver and the processing load on the receiver increases. This may ultimately result in lost packets at the receiver and an overall drop in throughput. Implosion is especially a problem in reliable multicast, which is useful for information dissemination applications.

Reduced Bandwidth: Merging messages together near their point of origin (i.e., near the senders) can substantially reduce the traffic load imposed on the network. This is particularly important in WAN settings such as the Internet where bandwidth is shared, and thus is a valuable resource. An example of an application to which this benefit is applicable is an audio-video conference, in which some participants are behind limited-bandwidth links. By combining audio and video streams in the network (see the next section), the conference remains fully distributed and each participant sees a single composite stream.

Larger Packets: Router performance is typically given in terms of packets/second, rather than bits per second. In other words, small packets present more of a performance challenge than large ones. By concatenating or merging multiple small messages into a larger single message, the fixed per-packet cost can be amortized over a larger (aggregate) packet. For applications in which large numbers of small packets travel from many senders toward a server—for example, TCP acknowledgement packets traveling toward a busy web server—aggregating packets can improve throughput if queue capacities are denoted in packets rather than bytes [5].

The number of group-oriented applications that would benefit from concast is growing rapidly. One class of such applications are *group-request-reply* applications, in which a single machine issues a request to a group of machines (typically via multicast) and then waits for a response from all group members (historically implemented via unicast). This model of interaction is used in network transport protocols, and in application-level communication. For example, sender-initiated reliable multicast protocols [34, 20, 33] transmit data to receivers via multicast and then wait for receivers to send ACK messages to the sender. Similarly, application-level concast ACK messages are required by a variety of applications to ensure end-to-end reliability [23]. Other applications need to gather distributed state information before making a decision. Distributed consensus algorithms may request a vote from all members.

Two other classes of application that exhibit concast communication patterns are *client-multiserver* and *multiclient-server* applications. To ensure reliability or availability, *client-multiserver* applications replicate server functionality across multiple machines. Client requests are multicasted to the servers who send simultaneous responses to the client. The client often needs only one, or K out of N, responses. For example, a distributed database may require multiple, K out of N ACKs when storing replicas of a newly written record. *Multiclient-server* refers to the classical client-server model, in which a single server responds to requests from any number of client machines. In this model, N clients transmit request messages, often simultaneously, to a single server. In many cases, the requests are similar or even the same. Consider a web server at a popular web site. It can service hundreds or thousands of requests each second, often for the same page. Logically these requests can be viewed as concast communication.

Another class of applications that are characterized by concast communication patterns is the *report-in* style applications in which distributed machines, sensors, robots, or other devices periodically transmit data to a centralized control or monitoring system. In some cases, transmission from the concast senders may be continuous such as stereo video feeds sent to a centralized video processing engine that may reconstruct 3D models, extracts depth, perform motion detection, target tracking, etc.

Distributed sensor networks represent another class of applications that can benefit from concast. Many of the current systems rely on a scheme in which data produced by the distributed sensors is collected at a single node that performs image processing, interpretation, and display of the data (for example, the Operator Control Unit in [15]). As the number of sensors in the system grows, the central processor is overburdend with data and a human operator is unable to accurately monitor the incoming information. Using a concast service to aggregate or filter data near the source avoids the problem of information (if not packet) implosion.

4.3 Implementation and Results

We have implemented concast in the Linux operating system. Our implementation has three components:

CSP Daemon

The CSP daemon (CSPd) is responsible for running the concast signaling protocol (CSP), and maintaining the concast state on routers. Two types of the CSPd are implemented, one for the receiver, one for the sender. The sender CSPd runs on sender nodes to extend and maintain the concast session and pull the merge specification down along the concast path. The receiver CSPd runs on the receiver and concast-capable nodes along the concast path. It is responsible for spawning the merge daemon (MERGEd) and signaling for the merge specification to be downloaded. The implementation of the two CSPd's consists of approximately 10,000 lines of C++ code.

Merge Framework

The merge daemon (MERGEd) that is spawned by CSPd on the concast path is responsible for locating and applying user-defined merge specification to concast packets. Currently, the merge specification defined in Fig. 5 can be implemented in Java or Tcl. The MERGEd is implemented in slightly more than 2,000 lines of Java code.

Kernel Modification

A concast kernel module supports socket options that allow users to mark sockets as concast sockets, and to join and leave concast groups. It also supports the intra-node communication between merge daemons and CSP deamon. The packet path is modified so that concast packets are recognized and shunted to the appropriate daemon for processing. The concast module implementation consists of less than 3,000 lines of code.

We have developed two test applications on this concast implementation: video merging and audio merging. In the video merging application, we emulate a distant learning environment, where the instructor sees the video feed from each student, while each student only needs to see the instructor. In this type of application, the video flows from students to instructor result in implosion and poor video quality if the bandwidth is not managed carefully. Using concast, flows can be merged and thinned in a manner similar to that used by RTP mixers [26], but in just the locations in the network where the processing is needed.

To support this type of application, we designed a simple merge function that thins incoming video streams by downsampling. It combines packets so frames from different incoming streams are combined into a single composite (tiled) frame made up of reduced-size frames; thus the outgoing link from each concast-capable router carries a single video stream. At each hop, the merge function keeps track of the number of incoming streams and the number of original streams contained in each. It then assigns a region of the outgoing frame to each incoming video stream and downsamples the stream appropriately to fit in the assigned re-

(a) Initially there may only be four students who's video is merged into a single video stream that is displayed.

(b) As more people join the class, the concast merge function dynamically adjusts the video to make room for the new students.

Figure 7. Illustration of a Distance Learning Application: The video stream from each source is down-sampled at the merge point, resulting in the viewing window size at the receiver remaining constant while the number of sub-windows increases (i.e. more participants), and the size of sub-windows decreases.

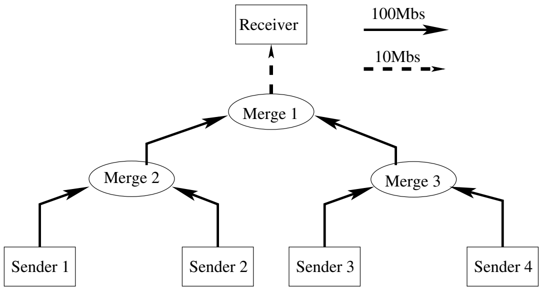

Figure 8. Topology used in the video merging experiments.

gion. As new students join the class, the other images are adjusted to make room for the new student (see Figure 7). Each composite stream carries information about how many original streams it contains, and how they have been last combined so that each node can determine how to combine its incoming streams.

We measured the load incurred on the routers while performing the video merge on a single concast flow. The experienmental topology is show in Figure 8. We used four video senders, each transmitting a baseband video stream of five 320x240 frames per second, using 8-bit grayscale, for a rate of about 3 Mbps.

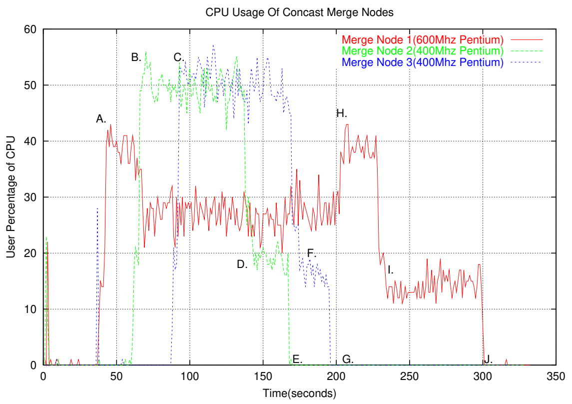

Figure 9. Merge processing load imposed on concast routers.

Figure 9 shows the CPU load caused by merge processing on each of the three merging routers (nodes 1, 2, and 3)[1] Our concast merge specification was written in Java and runs in a user-level JVM, which accounts for the majority of the load. Initially, video sources are started on senders 1 and 3. This causes the the load on merging node 1 to increase (see point A in the graph). Video sources were then started on senders 2 and 4 (points B and C), causing the load to increase on merging nodes 2 and 3 respectively[2]. Note that when a node only has one upstream neighbor, packets are forwarded as normal without invoking the merge processing. Despite being implemented in Java, the merging code (which is merging two 3 Mbps incoming streams into a single outgoing 3 Mbps stream) does not exceed a 60% CPU load. At the end of the test, the senders terminate (points D, F, and I) and after the concast softstate times out, the CPU loads again return to zero (points E, G, and J).

We have done similiar experiments with an audio merge application (e.g. emulating a distributed quartet concert). The source code of our concast implementation has been packaged for release, and is available on http://protocols.netlab.uky.edu/˜acast.

4.4 Securing Concast

Applications benefit from concast by being able to apply merge processing at the location in the network where it provides maximum leverage—where packet streams converge. However, this requires that end users trust the infrastructure to perform certain critical operations, namely (i) admit users to the concast session, and (ii) examine and possibly modify user data. In some cases users (in particular the receiver) may not trust some portions of the infrastructure to perform these functions securely. Conversely, it is likely that only a subset of users will be authorized to participate in concast sessions. Thus we need mechanisms to enforce two kinds of policies: user (i.e. receiver) policies authorizing interior nodes to participate as merge points, and network provider policies, authorizing users to create or join concast sessions.

These mechanisms operate primarily in the concast signaling protocol. Before a node (initially the receiver) supplies the specification to an upstream neighbor requesting it, it applies the receiver policy to verify that the potential neighbor is authorized to receive the merge specification. When an arbitrary interior node receives a request from a sender to join a concast flow, it applies local policy to check that the sender is authorized to use the service as a sender. Then it requests the merge specification from downstream; when the response arrives, it again applies local policy to check that its downstream neighbor (which may be the receiver) is authorized to supply a merge specification. Finally, it extracts the receiver policy from the merge speci-

[1] Merging node 1 happened to be a 600 MHz Pentium, whereas the other nodes were 400 MHz Pentiums – so the load appears to be lower on node 1 even though it is doing the same processing as the other nodes.

[2] This actually reduces the load on merging node 1 slightly because node 1 switches from horizontal down-sampling to vertical down-sampling (which is a better match for the data structures used.) When senders 2 and 4 terminate the load returns (point H).

fication and verifies that the receiver policy allows the particular sender to join the concast group. Only if all these checks succeed is the necessary flow state established.

4.5 Discussion

We have designed the concast signaling protocol with hooks for the authorization checks described above, as well as the cryptographic authentication mechanisms required. We have also implemented a means of specifying, verifying and merging policies using address prefixes. The mechanism allows the policy to define classes of allowed nodes, with all others being forbidden, or classes of forbidden nodes, with all others being allowed. In either case, one level of exceptions can be specified.

It is fairly clear that the need to specify these policies represents at least a potential violation of the principle of anonymity: receivers are required to identify specific nodes that are trusted (or not trusted). Thus, there is a fundamental conflict between access control and anonymity, which implies that for services to be anonymous, they must be "throwaway", that is, sufficiently lightweight to be made available to all comers. The design of the service we describe in the next section reflects this observation (as have other active network platforms [13]).

5. Ephemeral State Processing

Most interesting active services require per-user (or per-application) state in the network. Indeed, the ability for packets to exchange and collect information as they travel through the network is one of the interesting capabilities of active networking. However, the need to set up, manage, and reclaim state is a major impediment to scalability. For example, although they are very scalable in terms of the number of users per session, neither PAMcast nor concast scale as well in the "number of sessions" dimension, because of limits on the amount of state storage available at each node.

We have developed a novel approach to active networking based on an ultra-lightweight form of state. This approach, which we call *ephemeral state processing* (ESP), is simple enough to be implementable directly in hardware, and to support packet processing at line speeds. (In other words, it is designed to scale in the "number of simultaneous sessions" dimension.) ESP allows information carried in packets to be temporarily stored in the network, combined with information from other packets, and forwarded to a destination. All of this occurs under direct user control, with no out-of-band signaling or control setup required.

Processing in ESP is carried out in short computations called *operations* that are initiated by packets as they pass through the network. Each ESP-capable node supports a set of these computations, which may update the node state and/or fields of the packet. One way to think about ESP computations is that a single packet initiates a spatial sequence of operations—one per node that forwards it—while a series of packets initiates a temporal sequence of operations at a single node. Interesting distributed computations can be constructed by judicious arrangement of operation sequences in time and space.

The lightweight nature of ESP stems from the simplicity and fixed cost of the operations, and from the fact that stored state persists at a node for only a short, fixed time—on the order of seconds. Little's result says that for a given size store, the maximum rate of state usage that can be sustained is inversely proportional to the holding time. Thus by halving the "holding time" for state storage, we double the sustainable rate of usage. Because the resources consumed by each ESP operation are small and fixed, there is no need for access control—we believe it is quite feasible for nodes to process ESP packets at line speeds.

ESP is especially useful for auxiliary computations, that is, processing in the network that supports enhanced services. In particular, it offers a solution to the problem of determining *where* enhanced functionality should be installed, by allowing end systems to extract a limited amount of information about topology from the network.

In the next sections we describe the components of ESP: the Ephemeral State Store, the operations initiated by packets, and the wire protocol.

5.1. Ephemeral State Store

Much of the power and scalability of our approach stem from the availability of an associative memory called an *ephemeral state store* at each node. An associative memory allows fixed-size bit strings to be associated with keys or *tags* for subsequent retrieval and/or update. We model the store as a set of (*tag, value*) pairs; 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. In what follows, x denotes an arbitrary tag, while e denotes an arbitrary value.

The ephemeral state store is accessed through the following operations:

1. **put**(x, e): bind the value e to the tag x. After this operation, the pair (x, e) is in the set of bindings of the store.

2. **get**(x): Retrieve the value bound to tag x, if any. If no pair (x, e) is in the store when this operation is invoked, the special value $\perp$, which differs from every legitimate value, is returned.

The ephemeral state store has two distinctive characteristics. The first is that bindings are ephemeral: each (tag,value) pair is accessible for only a fixed interval of time after it is created. The parameter T_l is the *lifetime* of a binding in the store; once created, a binding remains in the store for T_l seconds and then vanishes. Note that bindings cannot be prevented from disappearing: there is no way to refresh ephemeral state. Also, the value of T_l should be approximately the same for every node, so that users can, when necessary, be assured that bindings have indeed expired.

The importance of the finite lifetime is that it allows the resource requirements of computations using the store to be precisely bounded. The flip side of this is that any value in the store must be retrieved within the state lifetime or lost. For scalability, we want the value of T_l to be as short as possible; for robustness, it needs to be long enough for interesting end-to-end services to be completed. A detailed description of how the value of T_l is set is outside the scope of this paper. For now the reader may assume that it is on the order of a few seconds. In other words, ephemeral state computations must complete within a few round-trip times through the network.

It is important to note that the capacity of an ephemeral state store is determined by the amount of memory available at the node. We want the tag space to be large enough so that users can *choose tags at random* and be assured that, with high probability, a tag chosen at time t will not be in use by any other user during the interval $[t, t + T_l]$, nor can any user "guess" another user's tag by any brute-force method. If each user chooses tags randomly, for storing values and the number of distinct tags is sufficiently large, the effect is that *each user sees a "private" ephemeral state store*. Because a tag collision is only a problem if two users use the same tag *within an interval of length T_l*, tag values of, for example, 64-bits are certainly within reason and offer reasonable protection/privacy.

5.2. Local Operations with Ephemeral State

The set of *operations* defines the computations that can be performed using ephemeral state. Each operation is carried in a specially marked packet that causes the operation to be invoked at ESP-capable routers during forwarding. These operations 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. Interesting computations can be constructed by sequencing instructions so that later ones make use of values left in the state store by earlier ones. The key differences here are that (1) sequencing must be achieved by arranging for a sequence of operation-initiating packets to arrive at the router (no program counter), and (2) each operation can only access values placed in the store within the last T_l seconds.

Operations can have zero or more operands of the following types:

- a value stored in the local ephemeral state store (i.e. bound to a tag carried in the initiating packet);
- an "immediate" value, i.e. one carried directly in the packet;
- a well-known parameter value (for example, the value of a MIB variable).

Operations are entirely local, and either run to completion or abort. An operation that completes successfully produces zero or more outputs that are either placed in the packet or bound to a tag in the ephemeral state store or both, depending on the operation. Upon successful completion, depending on the values computed by the operation the packet that initiated the operation is either silently dropped or forwarded toward its original destination—possibly carrying outputs from the computation, to be used as inputs at the next ESP-capable node. Each operation executes atomically with respect to the ephemeral state store; in particular, no binding can expire during an operation.

We envision that a standard set of a few dozen operations will suffice for a large class of interesting computations; here we describe three example operations that we have found useful in building active network services.

COUNTCH

The COUNTCH operation counts the number of times the operation has been carried out, and filters (blocks) packets based on that number. It takes one operand, a single tag carried in the initiating packet. If no value is currently bound to the tag, it is bound to the value 1 and the packet is forwarded. Otherwise, the value bound to the tag is increased by 1, and the initiating packet is silently discarded. When COUNTCH-initiating packets carrying the same tag are sent from a group of hosts to a common destination, the paths followed by the packets induce a tree. The effect of the operation is to bind to that tag, at each ESP-capable router, a count of the number of "children" it has in that tree. Each "child" is an ESP-capable router or sender. This operation is often used as a "set up" step for other operations.

COLLECT

The COLLECT operation applies an associative and commutative operator (e.g., max, min, sum, etc.) to values carried in COLLECT-initiating packets. A COLLECT-initiating packet contains two tags (say x and y), two values (say a and b), and an operator code *op* (the commutative/associative operation to perform). COLLECT expectes one of the tags x to already exist in the router's ESS. If x does not exist in the ESS, the operation aborts. If the other tag y does not exist, it is initialized and bound to the value a in the ESS. If tag y is found, the operation *op* is applied to value of y and value b, the result is bound to tag y. After each successful operation, the value bound to tag x is decremented by 1. The initiating packet is only forwarded if the value bound to x becomes 0; otherwise it is discarded.

When COUNTCH and COLLECT are sent by multicast receivers to the multicast sender, they can help the multicast sender to discover the number of receivers in the multicast group. When the *op* specified is addition, a COLLECT-initiating packet sent after COUNTCH-initiating packets sent by all multicast receivers will reach the multicast sender containing the total number of the receivers in the multicast tree (assuming no losses occur).

FMAX

The FMAX operation tests whether a value in the store at a given tag is greater than or equal to the value carried in the packet. If so, the value of a read-only system variable at the node, namely **nodeid** (e.g., the node's IP address), is stored in a field in the ESP packet. If no binding for the given tag exists, the packet is forwarded anyway. This operation is useful for locating nodes in the network that have desired properties, for example branch points of multicast trees.

FMAX illustrates the use of well-known (read-only) global variables in computations—in this case, **nodeid**. Each ESP-capable node provides access to this variable. Other well-known global variables may also be useful. In the limit, access to MIB variables might be supported.

Whenever an ESP packet arrives at a node (either for forwarding or because it is addressed to that node), it is recognized as such and passed off to the ESP module for processing.

Two forms of ESP are supported: dedicated and piggybacked. A *dedicated* ESP datagram consists of an IP datagram whose payload contains the opcode of the desired operation along with its packet-borne operands. The IP header of the datagram carries the Router Alert option [17] and the Protocol Number of the ESP protocol.

A *piggybacked* ESP datagram carries the opcode and operands in an IP option, along with a protocol number and payload of some regular application. Piggybacked ESP packets initiate operations as a side effect; their advantage is that they do not add signficantly to the bandwidth requirements of the network.

Ephemeral state processing at each network node is implemented as an adjunct to the Internet Protocol (or other network-layer protocol) similar to ICMP or IGMP. End systems must implement the mechanism. Interior network nodes should implement the mechanism, but the service will function correctly even if only a subset of interior nodes implement it.

5.3. Usage Scenario

The operations of the previous section can be used to solve problems and enhance applications. As a simple example, we present a method of determining the identity of a node in the intersection of two paths through the network, if any.

Given are four nodes A, B, X, and Y. We wish to find the address of the ESP-capable node closest to B that is on *both* the paths $A \to B$ and $X \to Y$. The solution has two steps, and requires the cooperation of A, X, and Y.

First, A transmits a COUNTCH-packet to B. This has the effect of binding the value 1 to the tag z at all ESP-capable nodes along that path. A short time later, X transmits a FMAX-packet to Y. This packet is sent with the value 0, the tag z, and X's ID. If it encounters a larger value (i.e. 1) bound to z at any node along the path, it collects the ID of that node and carries it on to Y. Because the comparison test in FMAX succeeds on equality, the node ID received at Y is that of the *closest* node to Y on both paths.

5.4. Implementation

We are developing two low-level implementations of ESP. One is based on the Intel(tm) IXP1200 network processor, a special-purpose platform with a strongARM core and multiple "micro-engines" for packet processing. The other is an FPGA-based implementation of the ESS, using an external RAM to store the actual tags and bindings.

6. Lightweight Processing Modules

LWP is a paradigm for providing enhanced services under user control. It is designed to leverage the capability provided by ESP, and is based on two ideas:

- Certain common and very simple functionality is useful for implementing a variety of packet-processing services. For example, the ability to extract packets matching a particular pattern and duplicate, redirect, or drop them is a useful building block from which many services can be built. (This observation also motivates the design of many "network processor" devices, such as the Intel IXP1200.)

- It is simpler for an application to communicate directly with a node in the middle of the network to invoke enhanced functionality than it is to have the invocation request propagate from node to node, across all intervening domains, subject to each node's policies. The "relay" model is appropriate when the functionality must be installed at a large number of nodes, as in concast. However, as we saw earlier, it does require that the application trust intermediaries to convey the request and any response correctly. When functionality is required at one or a small number of nodes, it is better for the application to negotiate directly with the node or nodes concerned.

An additional benefit of this paradigm is a simplified payment model: the requesting application can negotiate directly with the providing node regarding payment. This allows for the possibility of domain-specific charging arrangements, whereas in the request-relay model, the request must already contain adequate billing information for whatever node/domain ends up providing the service.

As we have seen, ESP allows end systems to identify regions in the network that are strategically useful for an application—a path intersection point, a congested link, or the confluence of multiple streams.

LWPs are predefined processing modules supported by network routers that can be activated by end systems via control messages. Each processing module applies some function (such as duplication) to packets passing through the router that match a specified pattern. The function is controlled through parameters specified at setup time. Only an authorized end user can instantiate an LWP for packets associated with the end system. Each LWP has an associated end system that is responsible for its existence (e.g. that refreshes the LWP state). This model simplifies the security and resource protection of the LWP system. We expect that a small set of these parameterized modules, which may be sequenced for packet processing, will suffice for a variety of network services.

Each processing module is defined by the following four components:

- *Module ID:* the function to execute at the router (or actual code).
- *Classifier:* a packet filter that identifies which packets this module should intercept, for example a value indicating the multicast group ID to intercept for duplication.
- *Parameters:* a set of configuration parameters that control the code's execution. For example, an instantiation parameter might specify the unicast address where packets output by this module are to be sent.
- *Timeout:* a timeout value indicating when the module should be automatically removed from the router. There may be a system-wide maximum timeout value imposed on all modules. Refresh messages are used to extend a module's lifetime.

Lightweight processing modules operate by identifying packets that match a particular classifier, applying the specified processing to those packets, and (possibly) forwarding the result(s) using standard unicast routing. A processing module may be instantiated multiple times on a router, where each instantiation differs in either the classifier, parameters, or timeout. Modules can be instantiated, terminated, and refreshed via the control mechanism. The refresh operation may dynamically alter the parameters used by a module. Note that a packet may match multiple lightweight processing modules at a router, and thus be processed multiple times.

An example of LWP is the `dup()` module, which replicates any packet matching its classifier filter and forwards the duplicate to a unicast address specified at module instantiation time as a configuration parameter. The original packet is simply forwarded normally to its original (unicast) destination.

Using this and other LWP modules together with ESP, we have shown how to implement multicast based on unicast routing [30]. In our scheme, multicast packets are directed to a unicast address (i.e., the destination address is always one of the receivers), but carry a *multicast group ID* in an option of IPv4 header (or an extension header of an IPv6 packet). The classifier of the `dup()` function specifies this multicast group ID. The function makes a copy of each matching packet, replaces the copy's original IP destination address with its own configured destination address, and forwards both the original and the copy normally. Thus, `dup()` functions correspond to "branch points" in a multicast delivery tree. By placing the `dup()` functions strategically, end systems can create application-specific multicast delivery trees. ESP is used to determine where to place the `dup()` functions. We have developed two schemes for creating application-level multicast trees: one using a centralized approach (sender-controlled) and a second, receiver controlled approach [30].

In the receiver-controlled approach, the receivers collaborate using ESP to identify the branch points in the existing multicast tree. A new receiver then adds a dup() function to an existing branch point. Periodically the sender transmits tree optimization messages that will result in the new receiver's dup() function being moved to the optimal location in the tree. An example multicast tree built in this manner is shown in Fig 10.

This ESP/LWP multicast tree structure can be easily extended to support layered multicast. Layered multicast has been proposed for video transmission, where the sender encodes the multicast content into different layers and then sends them on different multicast groups. Usually each

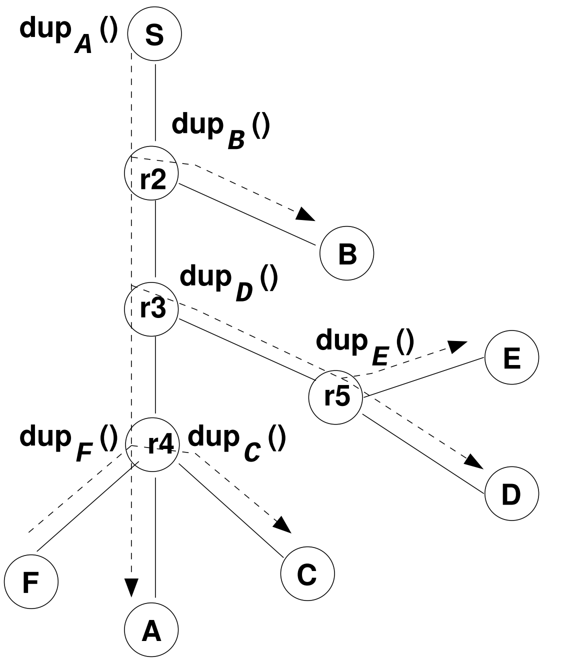

Figure 10. A ESP/LWP multicast tree.

layer improves the video quality incrementally. For example, receiving the base layer gives the lowest quality video and each additional layer improves the video quality. In traditional layered multicast, receivers join as many multicast groups as possible to receive the video streams at the fastest rate supported by the receiver's connection.

In our LWP multicast implementation, layers are identified not by different multicast groups, but by header or IP option on which the dup() function classifies; thus a single dup() function can duplicate multiple layers of data. Using drop() functions in addition to dup(), a receiver in a layered multicast session can dynamically adjust the arriving data rate. The multicast sender and receivers also monitor the congestion on the multicast path using ESP computation in a collaborative effort. We have designed such a layered multicast approach called Congestion-Aware Layered Multicast (CALM). Compared to traditional layered multicast implementations, CALM receivers can more accurately detect congestion on the multicast path, and react to congestion in a more timely manner. We have shown that CALM also provides more stable receiving quality than a traditional approach [18]. The details of our implementation and results are presented in [31].

7 Conclusions

In this paper we presented four new "application-friendly" active network services, *PAMcast, Concast, Ephemeral State Processing*, and *Lightweight Processing Modules*, designed to make it easier for applications, particularly group communication applications, to access and utilize active network services in a scalable way. We presented characteristics that application-friendly active network services should ideally exhibit and then showed how each of the above services attempted to achieved those goals.

All four services achieve, in one form or another, our primary goals of anonymity and opaqueness, and hide the complexity of the active network (topology information, number of users/end-systems, traffic patterns, etc.) from the application. Both Concast and ESP assist applications in selecting the "right" right set of nodes for their specific purpose. PAMcast automatically selects the "right" servers based on application-specified criteria.

ESP and LWP demonstrate two additional desireable characteristics, namely automatic network state management and enforcable security and resource protection models. ESP's ephemeral state store frees the application from the task of managing network state by providing an ultra-lightweight resource that is "too cheap to meter," and thus need not be protected. LWP provides simple, lightweight, fixed cost processing routines that are enabled by end-systems that can be easily authenticated. We have also shown that these building-block services can also enable new types of application-friendly services such as application level multicast and layered multicasting.

In short, we have shown that efficient and application-friendly active network services are feasible. We believe they are a necessary step in moving active networks from the laboratory to production deployments.

References

[1] S. Berson, R. Braden, T. Faber, and B. Lindell. The ASP EE: An Active Network Execution Environment. In *Proceedings of the DANCE Conference*, May 2002.

[2] J. Byers, M. Luby, and M. Mizenmacher. Accessing multiple mi rror sites in parallel: Using tornado codes to speedup downloads. In *IEEE Infocom*, March 1999.

[3] K. L. Calvert and E. W. Zegura. Composable active network elements. http://www.cc.gatech.edu/projects/canes/.

[4] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen. Building a Programmable Multiplexing Service Using Concast. In *Proceedings of 2000 International Conference on Network Protocols*, November 2000.

[5] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen. Concast: Design and Implementation of a

New Network Service. In *Proceedings of 1999 International Conference on Network Protocols*, Osaka, Japan, November 2000.

[6] Ken Calvert, Jim Griffioen, Billy Mullins, Amit Sehgal, and Su Wen. Implementing a Concast Service. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.

[7] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[8] Kenneth L. Calvert, James Griffioen, Billy Mullins, Amit Sehgal, and Su Wen. Concast: Design and Implementation of an Active Network Service. *IEEE Journal on Selected Area in Communications (JSAC)*, 19(3), March 1998.

[9] Y. Chae, K. Guo, M. Buddhikot, S. Suri, and E. Zegura. Silo, rainbow, and caching token: Schemes for scalable, fault tolerant stream caching. *To appear in IEEE JSAC*, Spring 2002.

[10] Y. Chae and E. Zegura. PAMcast: Programmable anymulticast for scalable message delivery. Technical report, Georgia Tech, College of Computing, 2001.

[11] S. Floyd, V. Jacobsen, S. McCanne, C-G Liu, and L. Zhang. A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. In *Proceeding of the ACM SIGCOMM'95 Conference*, November 1995.

[12] S. Golestani. A self-clocked fair queueing scheme for high speed applications. In *IEEE Infocom*, 1994.

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

[14] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceeding of the ACM SIGCOMM'95 Conference*, November 1995.

[15] T. Kanade, R. Collins, A. Lipton, P. Burt, and A. Wixson. Advances in Cooperative Multi-Sensor Video Surveillance. In *Proc. DARPA Image Understanding Workshop*, pages 3–23, 1998. Monterey, CA.

[16] M Dina Katabi and John Wroclawski. A Framework for Scalable Global IP-Anycast (GIA). In *SIGCOMM*, Stockholm, Sweden, August 2000.

[17] D. Katz. IP Router Alert Option, February 1997. RFC 2113.

[18] S. McCanne and V. Jacobson. Receiver-Driven Layered Multicast. In *Proceedings of the ACM SIGCOMM '96 Conference*, October 1996.

[19] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. *RFC 1546*, November 1993.

[20] Sanjoy Paul, Krishan K. Sabnani, John C. Lin, and Supratik Bhattacharyya. Reliable multicast transport protocl. *IEEE Journal on Selected Areas in Communications, special issue on Network Support for Multipoint Communication*, 1996.

[21] Sridhar Pingali, Donald F. Towsley, and James F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *the Proceedings of the ACM Sigmetrics Conference*, 1994.

[22] J. Rosenberg and H. Schulzrinne. Sampling of the Group Membership in RTP, February 2000. RFC 2762.

[23] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments In System Design. *ACM Transactions on Computer Systems*, 2:277–288, November 1984.

[24] Matt Sanders, Ken Calvert, Mark Keaton, Samrat Bhattacharjee, Stephen Zabele, and Ellen Zegura. Active Reliable Multicast on CANEs: A Case Study. In *Proceedings of the 4th IEEE International Conference on Open Architectures and Network Programming (OPENARCH'01)*, Anchorage, AK, April 2001.

[25] Christoph Schuba. Project Orion: Transparent Server (Cluster) Selection. In *Sprint Applied Research parTners Advanced Networking (SPARTAN) Symposium*, Lawrence, Kansas, May 1998.

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

[27] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[28] Dan Sterne. Adaptive Network Defense: Active Network Intrusion Detection and Response, 2001. http://www.pgp.com/research/nailabs/adaptive-network/active-networks.asp.

[29] UCB, LBNL, and VINT. Network Simulator. http://www.isi.edu/nsnam/ns/.

[30] S. Wen, J. Griffioen, and K. Calvert. Building Multicast Services from Unicast Forwarding and Ephemeral State. In *Proceedings of the OpenArch 2001 Conference*, April 2001.

[31] Su Wen, James Griffioen, and Ken Calvert. CALM: Application-Aware Layered Multicast. In *Proceedings of the 5th IEEE International Conference on Open Architectures and Network Programming (OPENARCH'02)*, June 2002.

[32] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[33] B. Whetten, S. Kaplan, and T. Montgomery. A High Performance Totally Ordered Multicast Protocol, August 1994. ftp://research.ivy.nasa.gov/pub/docs/RMP/RMP_dagstuhl.ps.

[34] Rajendra Yavatkar, James Griffioen, and Madhu Sudan. A reliable dissemination protocol for interactive collaborative applications. *IEEE Journal on Selected Areas in Communications, special issue on Network Support for Multipoint Communication*, 1996.

[35] S. Zabele and T. Stanzione. Interest Management Using an Active Networks Approach. In *Simulation Interoperability Workshop (SIW)*, March 2000.

[36] L. Zhang. Virtualclock: A new traffic control algorithm for packet switching networks. In *ACM Sigcomm'90*, 1990.