**AFRL-IF-RS-TR-2004-183**
**Final Technical Report**
**June 2004**


# ACTIVECAST

**University of Kentucky**

**Sponsored by**
**Defense Advanced Research Projects Agency**
**DARPA Order No. H498**



**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

**STINFO FINAL REPORT**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-183 has been reviewed and is approved for publication

APPROVED: /s/

SCOTT S. SHYNE
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 074-0188*



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | JUNE 2004 | Final Jun 99 – Dec 03 |

**4. TITLE AND SUBTITLE**
ACTIVECAST

**5. FUNDING NUMBERS**
C - F30602-99-1-0514
PE - 62301E
PR - H498
TA - 00
WU - 01

**6. AUTHOR(S)**
Kenneth L Calvert,
James N. Griffioen, and
Ellen W. Zegura

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Kentucky
201 Kinkead Hall
Lexington Kentucky 40506-4514

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGA
525 Brooks Road
Rome New York 13441-4505

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
AFRL-IF-RS-TR-2004-183

**11. SUPPLEMENTARY NOTES**

AFRL Project Engineer: Scott S. Shyne/IFGA/(315) 330-4819/ Scott.Shyne@rl.af.mil

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** ***(Maximum 200 Words)***

The Activecast project, with the support of the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory, Air Force Material Command, US Air Force, under agreement number F30602-99-1-0514, has developed programmable network services to improve the scalability and usability of networks in general. Technologies produced by the project are designed to incorporate programmability in a form that could be deployed in the present Interne including: Programmable Any-Multica (PAMcast); Concast and Secure Concast; Ephemeral State Processing and Lightweight Processing Modules (ESP/LWP); and Speccast. In addition, auxiliary technologies have been developed either in support of, or based on these services. In this report we described each service/technology along with its intended use, and the results of any experimental evaluations of the service. Other outputs produced by the project are listed at the end of the report.

**14. SUBJECT TERMS**
Active Networking, Multicase, Anycase, Unicast

**15. NUMBER OF PAGES**
47

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# Table of Contents

# List of Figures



List of Tables

# 1 Overview

Active networks support dynamic generalization of network services by injecting code and/or policy into the shared nodes of the network. Early active networking projects have focused on *platforms* and enabling technology such as programming languages and lower-level architectural features.These projects achieved significant and interesting results in terms of expressive power, flexibility and security. Other projects applied active network technology to solve *application-specific* problems, such as network management, interest-filtering and time management in distributed simulation, and video streaming. The Activecast project was initiated to help bridge the gap between these two extremes. To improve the *useability* of active networks, "application-friendly" APIs for high-level customizable services were needed. These services would be designed to hide the complexity of the programming process while allowing applications to conveniently specify the processing/handling they would like their data to receive.

This report describes the technical results of the Activecast project: *PAMcast*, a programmable form of anycast that uses application-specific criteria to select one (or more) destinations from a set of possible destinations; *Concast* (and Secure Concast), a many-to-one channel that can be programmed to merge messages as they travel through the network from multiple senders toward a single receiver; *Ephemeral State Processing (ESP)*, an ultra-lightweight form of active networking designed to support auxiliary computations in the network; *LightWeight Processing modules (LWP)*, a paradigm for deployment of per-user services in the network; and *Speccast*, a generalized routing and forwarding service in which addresses are replaced by predicates that specify packet destinations.

These services represent a middle point between general-purpose, application-independent platforms (offered by Execution Environments (EEs) and Node Operating Systems) and application-specific solutions (such as reliable data distribution). . 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. We argue that they are *scalable*, both in the sense of supporting applications involving many endpoints, and in the sense of supporting widespread deployment and simultaneous usage by many applications. Concast, PAMcast, LWP and Speccast are EE-agnostic—they could be implemented on any of the currently extant EE platforms—while ESP should be regarded as a very lightweight EE, which is designed to be implemented mainly in special hardware. These services are also generally backward-compatible, in that they could be deployed incrementally in the Internet, given appropriate (specialized) router support.

This report is organized as follows. The next section outlines some lessons learned from the project, including key principles for designing scalable, usable services. We then describe the PAMcast, Concast and Secure Concast, ESP, LWP, and Speccast services, highlighting their embodiment of those principles and giving example uses of the services. We then briefly describe other technology developed in the project, including some that emerged as a by-product, and some whose development is still ongoing. In the final section, tangible outputs of the project are tallied.

## 2 Designing 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 Programmable Any-Multicast

PAMcast is based on the traditional anycast service specified by Partridge, Mendez, and Milliken in RFC 1546, in which the network delivers a packet from a sender to *any* of 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 or closest Autonomous System (AS) count. 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, for example Berkeley's Shared Passive Network Performance Discovery (SPAND) and Georgia Tech's Application-level Anycasting Service. 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 [6].

## 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 [5].

- **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. It is well-known that parallel downloads improve client response time over single-server downloads. 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 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

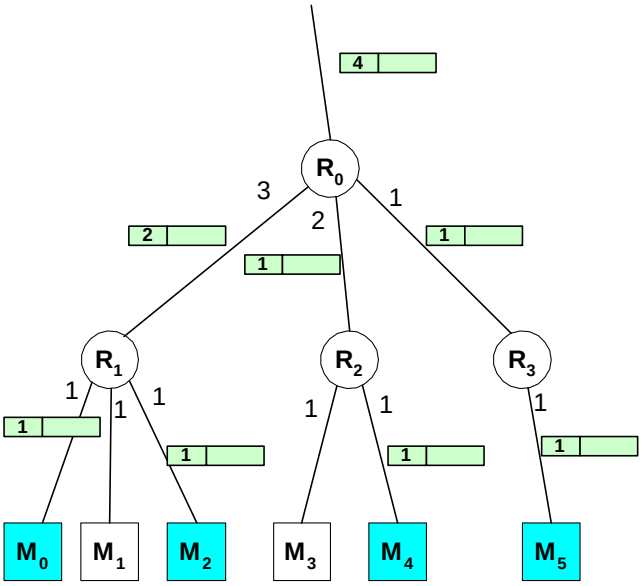



Figure 1: Message delivery on a PAMcast tree

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.

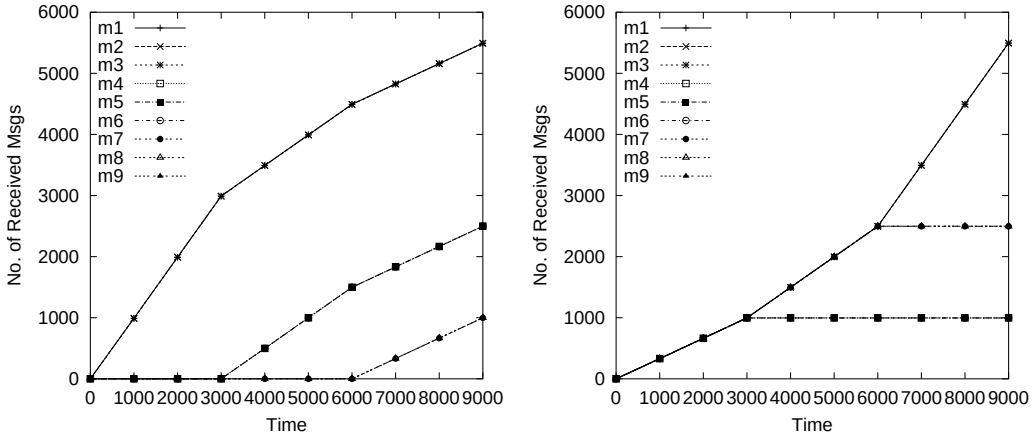
We implement the balanced mode using a weighted counter (WC) for each tree link, which is similar to the virtual clock for weighted fair queueing. The weighted counter keeps track of the total degree of the messages passing through the tree link, which is normalized to the number of downstream group 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 simulation package to model the PAMcast service. For our study, we used five different random topologies of 100 nodes. 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 [6]. First, Figure 2(a) 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




(a) Messages received as members join

(b) Messages received as members leave

Figure 2: Message distribution with WC algorithm

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(a) shows that the WC algorithm works well with the member join activities.

Figure 2(b) shows how the WC algorithm works when members leave the PAMcast group. The graph 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(a) and 2(b) have shown the load-balancing property of the WC algorithm is well-maintained with group join/leave activities.

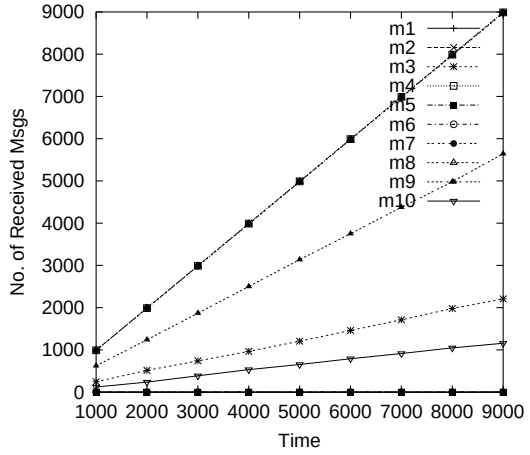



Figure 3: Probabilistic closest with m = 4

Figure 3 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 3.) 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 3, 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.

Table 1: Hop distance of group members in a tree for Figure3

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

Additional performance evaluation can be found in [6]. 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 Application Scenarios

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, that the ability to select a customized 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 [5]. 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.

## 4 Concast

The IP multicast communication paradigm lets a single address represent a *set* of nodes. Nodes signal the network their requests to be included in the set associated with a particular address; the network takes responsibility for delivering a packet addressed to that address to all nodes in the set. 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.

Maintaining this abstraction mechanism when information has to flow from the group back toward a single node 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 historically have resorted to ad-hoc unicast-based solutions that limit scalability. The purpose of the *concast* service is to provide such a many-to-one channel.

Concast embodies the principles of anonymity and specialization exemplified by multicast. In particular, it preserves anonymity when multicast receivers need to send feedback to the multicast sender. Concast allows the 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 simple generic 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 the individual publications [2, 1, 4].

### 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 4: 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 *custom* methods or functions (see Figure 4), which are invoked from a *generic* packet-processing loop (see Figure 5), which is the same for each flow. This generic packet-processing loop is invoked for each incoming packet belonging to the flow.

The concast framework allows users to supply the definitions of the custom methods using a

```
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 5: Network per-packet processing.

(restricted) mobile-code-language. The definitions are provided by the application receiver and pulled down through the network to the appropriate nodes in a "just-in-time" manner, as senders join the group. The *Concast Signaling Protocol* is responsible for this deployment. 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 in a *flow state block* (FSB):

**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 controls the reclamation of the resources associated with the entry; if it expires without being refreshed, the entry is removed. Thus each concast-capable node periodically sends a (unicast) message downstream—that is, toward the flow's receiver—to refresh its soft state. When a flow's UNL contains fewer than two entries, 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, i.e. the intermediate results of processing incoming messages belonging to this flow.

Incoming concast 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 next 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 Concast Signaling

The *Concast Signaling Protocol* (CSP) establishes concast-related state in network nodes. The protocol works by "pulling" the merge spec from the receiver towards senders as they join the group; concast flow state is thus initialized along the path from each sender to the receiver. The receiving application initially installs the merge specification locally (i.e. at the receiving host). Senders join the group by transmitting a *Request for Merge Spec* (RMS) message toward the receiver $R$. All CSP messages are sent as regular IP unicast datagrams with CSP identified in the protocol field, $(G, R)$ in the Concast ID option field, and the IP Router Alert Option (RFC 2113) to flag the packet for hop-by-hop processing at concast-capable routers. Concast-capable nodes process every CSP packet as described below.

Whenever a CSP message is sent downstream, the IP destination address is $R$ (the flow's receiver); CSP messages sent upstream have the IP address of the next upstream router in the tree as destination. The source address of a CSP packet is always the (unicast) IP address of the packet originator, which may be a router.[1]

Each CSP message contains a *flow id* $(R, G)$ in an option in the IP header; $R$ is the already-mentioned receiver address and $G$ is the concast group address. In addition, each message contains an indication of the message type and type-specific information. The message types along with their type-specific information are:

**Join Flow Request (JFR):** This message is transmitted by a sender toward the receiver. It informs the nearest downstream concast-capable router of the existence of a sender for this flow.

**Concast Join Succeeded (CJS):** This message is sent in response to a JFR, after the downstream node has received the merge spec and set up the flow.

**Request for Merge Specification (RMS):** This message is always sent downstream. It indicates to a downstream node the existence of a concast-capable router upstream, and requests that the upstream node be added to the specified flow. RMS messages contain a **Refresh Flag**, which indicates whether the message is being sent to keep an entry alive in the downstream UNL. If the flag is cleared, the sender expects the flow's merge specification to be sent in response to the message.

**Merge Specification (MS):** The MS message is sent upstream, in response to an RMS message (with the Refresh Flag clear), and carries the merge specification for the requested receiver and group.

**Error:** The Error message conveys information about various error conditions related to the concast data flow (e.g. destination unreachable). It may be sent to the originator of a RMS message or to all the members on the Upstream Neighbor List.

The normal progression of messages in the CSP protocol is illustrated in Figure 6. Initially receiver $R$ indicates to the local concast implementation—labeled "RCSPD", for *receive-CSP daemon* in the figure—its request to create (and receive from) the flow $(R, G)$, providing a merge specification as part of the request. The RCSPD creates the FSB for $(R, G)$ with an empty UNL.

[1] This is a crucial from our early designs, which specified that the concast group ID be used as the source ID. The change was necessary to enable connectivity problems in the concast tree to be detected via ICMP. (ICMP messages cannot be sent in response to packets with concast source addresses.)

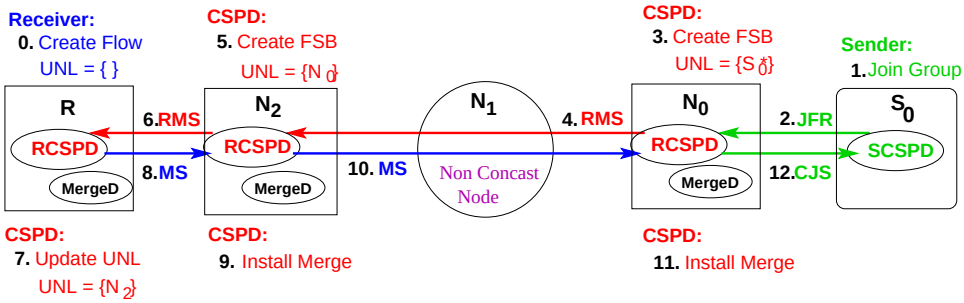



Figure 6: Concast Signaling Message Flow.

Subsequently, sender ($S_0$) indicates to its local SCSPD (*send-CSP daemon*) its request to join the group $(R, G)$. The SCSPD then transmits a Join Flow Request message with source address $S_0$ and destination $R$. At the first concast-capable node on the path between $S_0$ and $R$ (labeled $N_0$ in the figure), the JFR message is processed by the RCSPD, which creates a FSB for the flow, with a UNL containing only $S_0$; the state of the flow is marked "pending", to indicate that it is not yet fully established. Node $N_0$ then sends a RMS message toward $R$. As the figure indicates, non-concast-capable nodes ($N_1$) simply forward the CSP messages normally. At the next concast-capable hop ($N_2$), again a FSB is created and another RMS message is sent. When it reaches $R$, $N_2$ is added to the UNL for $(R, G)$ and the complete merge specification is sent back upstream to $N_2$, the source of the RMS. When $N_2$ receives the MS message, it installs the merge specification, sets the flow state to FORWARDING, and then sends the MS message directly to $N_0$ (because it is in the UNL). Finally, after installing the merge specification, the RCSPD at $N_0$ returns a CJS message to $S_0$, at which point the original "join group" request succeeds, and the sender is free to transmit data on the flow. Processing of concast datagrams belonging to the $(R, G)$ flow is now being performed at $N_0$ and $N_2$, but only one sender belongs to the flow, so any messages sent by $S_0$ are forwarded unchanged to $R$.

When another sender joins the same flow, the CJS/RMS messages progress downstream *only* until they reach a node that is already a member of the flow; that member transmits the merge specification back upstream, extending the flow tree to the new sender.

### 4.3 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. 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 impor-

tant 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—our simulation study showed that under certain conditions, aggregating packets can improve throughput [3].

A variety of group-oriented applications can benefit from concast. 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 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. 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 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. As the number of sensors in the system grows, some form of hierarchy is needed to keep the central server (and its incoming channels) from becoming overloaded. Using a concast service

to aggregate or filter data en route to the collection node automatically delegates processing into the network.

## 4.4 Securing the Concast Service

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, which presumably will be making use of the received data—may not trust some portions of the infrastructure to perform these functions securely. Unfortunately, the use of end-to-end security, which is the standard practice when untrusted nodes handle user data, is not compatible with merging data inside the network. Conversely, it is likely that only a subset of users—say, those who have paid for the privilege—will be authorized to participate in concast sessions. Thus, we need to extend the concast service described above with mechanisms to enforce various policies that govern which nodes can participate in a concast session.

With this in mind, we developed a secure concast design based on the principle that *control over participation in a concast flow (and thus access to user data) is equivalent to possession of the merge specification*. In other words, any node possessing the merge specification will be able to get hold of and possibly modify user data and forward it to the receiver, and nodes without the merge specification will not be able to. It follows that the security of a flow rests upon the secrecy of the merge specification. This has two important consequences:

- The propagation of the merge specification is controlled by the signaling protocol; thus it must be modified to ensure that only trusted nodes (i.e. nodes authorized according to the policies mentioned above) are able to obtain the merge specification.

- Given secure distribution of the merge spec, security of user data can be delegated to the merging process. That is, mechanisms (including secret keys) needed to ensure confidentiality and authenticity of *user data* sent/received on the flow can be placed in the merge specification.

This results in a nice separation of concerns: the application is responsible for the security of its own data *en route*, via the merge spec it supplies; the concast infrastructure is responsible for ensuring that only duly authorized nodes participate in processing that user data according to the given specification. In particular, applications can choose to employ “null” mechanisms in the merge spec if they are not concerned with security.

Implementing secure concast required substantial modifications to the signaling protocol, as well as more modest modifications to the generic merging framework. We describe the changes to the signaling protocol first.

### 4.4.1 Modifications to the Signaling Protocol

The secure concast signaling protocol is required to enforce the following policies:

- User specification of which senders (users) are authorized to join a given flow. This policy is supplied by the receiver as part of the merge specification.

- User specification of which routers are authorized to participate in the flow, i.e. to carry out the merge specification on user data sent toward the receiver. This policy also is supplied by the receiver in the merge spec.

- Service provider specification of which users are authorized to create or join concast flows. Since concast is a "value-add" service, providers want to ensure that only customers who have paid for the service have access to its benefits. This policy is assumed to be installed locally at each router in the network.

- Service provider specification of which routers are permitted to participate in concast flows. Providers may or may not trust each others' routers. For example, a transit provider may only accept merge specifications from (or supply merge specifications to) routers belonging to other providers with which it has "concast peering agreements". This policy is also assumed to be installed locally at each router.

Enforcing these policies is a challenge because *the point of concast is to preserve anonymity*. Requiring the receiver to make an explicit authorization decision on each sender who wants to join a flow would eliminate the scalability benefits that motivate the use of concast in the first place. The only way around this dilemma is to rely on *transitive trust*: the receiver has to rely on upstream routers to enforce its policies. Similarly, intermediate nodes rely on routers upstream to enforce their policies about who can be a sender or merge point. It follows that *to join the flow, nodes must not only be trusted to handle user data, they must also be trusted to enforce all relevant policies*.

Since the security of the flow rests on the secrecy of the merge specification, and being trusted to join the flow implies being trusted to enforce all relevant policies, policies can simply be carried in the merge specification. Thus, the main modifications required in the signaling protocol are:

1. Signaling messages are modified to carry origin authentication information. In particular, JFR messages carry a authenticator for the joining sender; RMS messages carry an authenticator for the requesting router in addition to the sender credential from the original JFR message. MS messages carry authentication information for the receiver (i.e. origin of the merge specification) and the downstream neighbor router (if any) originating the message. We do not specify a particular authentication mechanism; also, participants are assumed to obtain any secrets needed to verify authenticity through out-of-band means. (Our implementation supports both shared-secret and public-key authentication mechanisms.)

2. Before establishing flow state on behalf of an upstream node (in response to a JFR or RMS message), a router applies all relevant policies. In particular, when a JFR or RMS message is received, the local node policy is applied to the (authenticated identity of the) requesting upstream node before establishing the FSB and forwarding the request downstream. When/if a MS message is received in reply, the policies in the merge spec are then applied to the node in the UNL, and only if it is authorized is the flow finally established.

3. To keep the merge specification secret in transit between authorized concast-capable nodes, a step is added to the protocol which establishes an *IPsec tunnel* between a node and its upstream neighbor before the MS message is transmitted. All concast signaling messages between the two routers are transmitted over this tunnel, which is shared by all flows for which the two routers are neighbors.

These modifications result in a somewhat more complex signaling process. This is, however, required to maintain the invariant that every participating node is acceptable according to its neighbors' (in the concast flow tree) policies.

### 4.4.2 Merge Framework Modifications

In addition to changes required to secure the "control plane", changes were also needed in the merging framework in order to support user-defined encryption/decryption in the data plane.

First, we modified the merge specification to include user-defined encryption function and decryption functions, as well as the secret key to be used for encryption and decryption. These functions may be specified via Java byte codes, or by reference to standard, predefined encryption and decryption functions (e.g. AES-ECB), which we made part of the standard merging framework (i.e. `merged`) implementation. We also modified the generic message-processing loop shown in Figure 5 to invoke these functions to decrypt each incoming concast data message, and to encrypt each outgoing concast data message. (Applications that do not wish to secure their data can specify standard "null" encryption methods.) As part of the encryption specification, the framework allows the user to specify whether a MAC (message authentication code) should be included in the encrypted message. If so, the MAC will be checked when the packet is decrypted to verify its integrity.

The second change to the framework allows the per-flow merge daemon at a merging node to distinguish among senders, merging nodes, and the receiver. Merge daemons executing on sender nodes receive packets over a local socket. Because they arrive unencrypted, the decryption function does not need to be invoked (only the encryption must be called). On receiver nodes, incoming packets are processed and delivered straight to the receiver application. In this case, only the decrypt function is required. On intermediate nodes both encryption and decryption are invoked on all incoming and outgoing packets (assuming the flow is in the merging state).

Because the sending and merging operations are controlled by separate policies, the first downstream node from the sender responds to the JFR message with a *partial* merge specification, containing only the secret key and the encryption function; in particular, it does not contain the merge functions.

## 4.5 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. 4 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 daemon. 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 (who is behind a modest-capacity link) 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 with real-time protocol mixers, but in just the locations in the network where the processing is needed (as opposed to specialized servers, which may be off the normal path between participating nodes). The audio merge function is similar: audio streams from different sources are combined and played out at a single location that is behind a bottleneck network link; this could be used for a field briefing from distributed intelligence sources, or for performance of a distributed quartet.

To support this type of application, we designed a simple merge function that "thins" incoming video streams by downsampling (i.e. throwing away pixel values in a controlled way) . 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 down-samples the stream appropriately to fit in the assigned region. 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 experimental topology is shown 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 shows the CPU load caused by merge processing on each of the three merging routers (nodes 1, 2, and 3)[2] 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

[2]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.

(a) Initially there may only be four students whose 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.

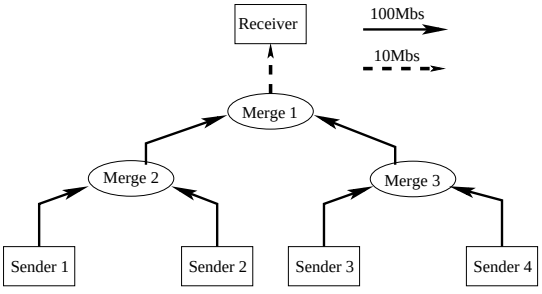



Figure 8: Topology used in the video merging experiments.

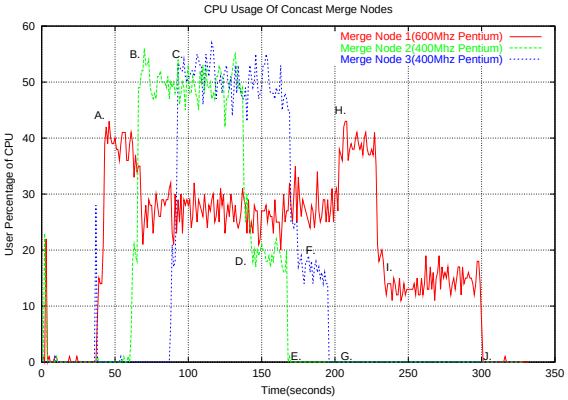



Figure 9: Merge processing load imposed on concast routers.

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[3]. 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).

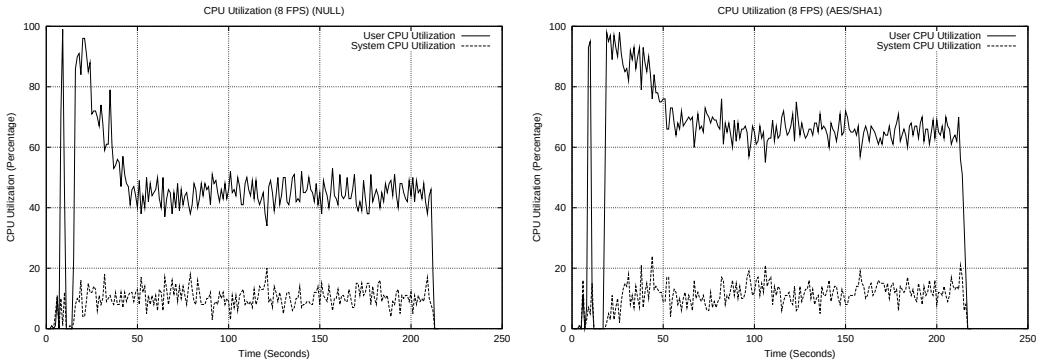
To quantify the cost of security for concast service, we ported the video-merging application to the secure concast framework, and measured the load at an intermediate node (1.4 GHz Pentium) where two 8-frames-per-second streams were being merged into a single stream. The graphs in Figure 10 show the results. The user component of the overall load increased from around 45% using the "null" ciphersuite to around 65% when the Advanced Encryption System was used with a SHA-1-based MAC. The system component of the load remained more or less constant at around 10%.

## 4.6 Discussion

We have designed and implemented a secure version of the concast protocol, to make it feasible to deploy the service in production networks where various users and service providers may not trust each other. We have also designed and implemented a means of specifying, verifying and merging policies using address prefixes. The mechanism allows the policy to define classes of allowed/forbidden nodes based on IP addresses.

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 described in the next section reflects this

[3]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).




(a) Load vs. time, with null security

(b) Load vs. time, with AES/SHA-1

Figure 10: User and system load: video merge at 8 frames/sec

observation.

# 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 scales especially well in the "number of sessions" dimension, because of limits on the amount of state storage available at each node. In the case of Concast, we addressed this problem by providing mechanisms to control access to the service.

A different approach led to a novel approach to active networking based on an ultra-lightweight form of state, which has essentially zero management cost. 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 $[t, t+\mathrm{T}_l]$, 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 $\mathrm{T}_l$ 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 expects 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 (RFC 2113) 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

As a simple example of the use of ESP, 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 $A \rightarrow B$ and $X \rightarrow Y$. 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.

As a building-block service, ESP should be useful to a variety of applications and higher-level services. In other publications we have shown the following uses for ESP:

- In combination with Lightweight Processing Modules that can perform packet duplication, ESP can be used to implement a user-controlled multicast service. That is, the network provides only unicast routing services; ESP is used to determine the best location in the network for duplication to take place [7].
- Again in combination with LWP modules, this time performing filtering, ESP can be used to perform congestion control for multicast applications [12].
- We have also shown how ESP can be used in the construction of a generic *grouping* service, which enables nodes to build more efficient overlay networks.

### 5.4 A Network-Processor-Based Implementation

A key objective of high-performance (backbone) routers is the ability to *handle packets at wire-speeds on as many interfaces as possible*. To achieve this level of performance, current (non-programmable) high-speed routers perform packet processing in hardware using application-specific integrated circuits (ASICs). This approach has been highly successful, producing routers that can switch millions of packets per second.

More recently vendors have introduced programmable *network processors* that can perform packet processing in parallel. To evaluate ESP's ability to scale to highend routers, we used the Intel IXP 1200 network processor[4] as our development platform. The IXP 1200 is designed for high-performance, deep packet processing, and has general characteristics similar to those of other network processors such as programmability, multiple processing engines, and a multi-level memory hierarchy. Consequently, we expect that insights derived from our experience with the IXP 1200 can be applied to other network processor platforms as well.

#### 5.4.1 Mapping ESP to the IXP 1200

The Intel IXP 1200 network processor, illustrated in Figure 11, is designed to facilitate the deployment of new value-added services simply by changing the software running on the hardware. To

[4]Intel Corporation provided additional support for this work.

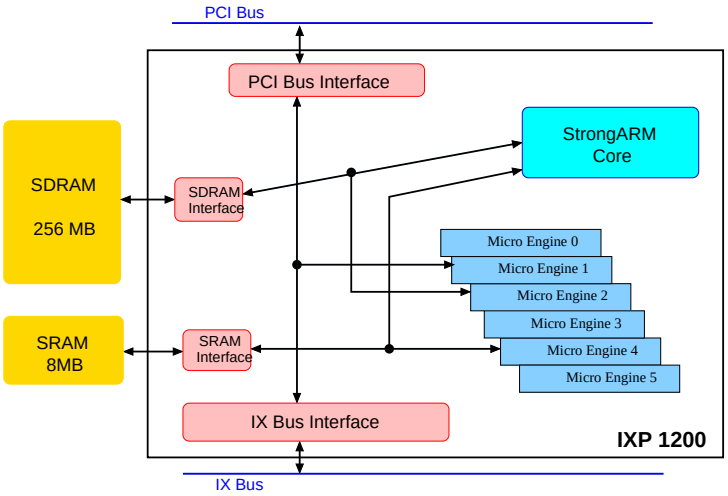



Figure 11: IXP 1200 architecture

achieve line-rate routing/switching, the IXP 1200 supports parallel packet processing on a set of six processors called *micro-engines*. The six on-chip micro-engines each supports four *hardware threads*. Each of the micro-engines is individually programmable and supports a set of hardware instructions specifically designed for packet processing. Like other network processors, the IXP 1200 uses a multi-level memory hierarchy consisting of registers, SRAM, and SDRAM. Registers offer the fastest performance, with SRAM and SDRAM performance being significantly slower; SDRAM being worse than SRAM. However, the SDRAM is much larger than the SRAM which is much larger than the register set.

To achieve the desired level of performance, we defined a set of design goals for our IXP 1200 implementation:

- **Maximize Parallelism/Minimize Synchronization Overhead.**
- **Minimize Memory Accesses.** Map data structures to storage so that ESS information can be retrieved with as few accesses as possible.
- **Maximize ESS Capacity.** Use memory efficiently.
- **Maintain ESP Semantics.** Maintain the ordering and sharing constraints described above.
- **Operate Continuously.**

To achieve parallelism, we distribute the ESP processing across the set of IXP micro-engines. Micro-engine 0 serves as the ingress processor and packet dispatcher, micro-engine 5 collects all outgoing packets and enqueues them for transmission, and micro-engines 1 to 4 handle all ESP processing.

Because we want the ESS to be as large as possible, we store the ESS data itself (tags and values) in SDRAM. Although SDRAM has the highest read/write latency of any of the IXP 1200's storage areas, it is only a bit higher than the latency of the SRAM, and it offers significantly larger capacity. Smaller auxiliary data structures (hash tables) are stored in SRAM for speed.

ESP requires at least one ESS per network interface, but there are advantages to having more ESS's per interface. First, partitioning computations into different ESS's reduces the number of tags per ESS, thereby reducing the probability of tag collisions. Second, reclamation of expired data requires synchronization between the cleaning thread and the packet-processing threads; on the IXP1200, synchronization across micro-engines is significantly more expensive than between threads on the same micro-engine. Therefore in our design, each micro-engine (as opposed to each interface) has its own ESS, used exclusively for packets processed by that micro-engine. To ensure that packets are processed in the correct ESS context, we use a hash of the computation ID to assign packets to ESSs.

Although parallelizing the processing across multiple micro-engines results in improved performance, we took it a step farther, by using multiple threads per micro-engine. Although all threads execute on the same CPU, the ability to switch to a different thread while blocked waiting for a memory access to complete (say to SRAM or SDRAM), allows us to mask the high latency associated with these operations. However, concurrent processing by multiple threads raises several interesting issues. First, to ensure that packets from the same computation are not processed concurrently by different threads, the packet demultiplexing code dispatches all packets having the same computation id to the same one of four input queues (not just the same micro-engine). Each thread can service any input queue; however, we use local thread synchronization techniques to ensure that no two threads work on the same queue at the same time. Second, given the challenges of synchronizing multiple concurrent threads, it might make sense to further subdivide the ESS into subESS's, with one subESS per thread. The downside of this approach is that memory can become fragmented, wasting valuable ESS space. Although it would simplify synchronization, we decided to stick with a single ESS per micro-engine (shared by all threads).

To support continuous operation of the router, we used one thread on each micro-engine as a *cleaner thread*, reclaiming expired (tag,value) pairs from the ESS. Because the cleaner examines and modifies the state of the ESS, there is potential for interference between it and a normal ESP instruction being executed concurrently by another thread. To address this issue we designed an efficient synchronization method that allows ESS accesses by both the cleaner and instruction-processing threads to be interleaved with minimal synchronization.

### 5.4.2 IXP Performance Results

To evaluate the performance, we measured the throughput achievable with multiple threads. All threads executed on the same micro-engine, so there was no direct increase in parallelism; however, the use of multiple threads resulted in overlapping SDRAM accesses, hiding (at least some of) the latency of reading and writing. To generate enough traffic to keep the threads busy, we ran eight PCs simultaneously transmitted minimum-sized packets as fast as possible. Each packet carried a **count-with-threshold** instruction; 50% of the packets created a new tag, and 50% simply incremented an existing tag. Because packets were minimum sized (requiring an inter-packet gap), the total offered load we were able to generate was (76.25 Mbps $\times$ 4 lines) = 305 Mbps.

Table 2 shows the throughput, both in packets per second and megabits per second, for 1, 2, and 3 threads. The fourth thread is the cleaner. The "No Cleaner-assist" table shows performance when the cleaner does nothing but clean. The "With Cleaner-assist" table shows performance when the cleaner uses its spare cycles to assist with packet processing. Clearly, at lower loads, the cleaner has more cycles to assist with packet processing than at high loads, and the throughput with one

| No Cleaner-assist | |
|---|---|
| No. Threads | Throughput |
| 1 | 234 Kpps = 120 Mbps |
| 2 | 420 Kpps = 215 Mbps |
| 3 | 547 Kpps = 280 Mbps |

| With Cleaner-assist | |
|---|---|
| No. Threads | Throughput |
| 1 | 212 Kpps = 108 Mbps |
| 2 | 410 Kpps = 210 Mbps |
| 3 | 507 Kpps = 260 Mbps |
| 4 | 564 Kpps = 290 Mbps |

Table 2: Throughput of a single micro-engine, with and without cleaner-assist.

packet-processing thread plus the cleaner assist is comparable to the throughput with two packet-processing threads. Recall that the maximum achievable throughput (with minimum-size packets) is 305 Mbps due to Ethernet limitations. With three threads (plus the cleaner in its spare time) processing packets, the IXP comes within 5% of the full 305 Mbps. In fact, when we changed the mix of instructions so that all packets carry the same tag, the IXP operated at the optimal 305 Mbps rate. Note that this level of performance is achieved with only *a single micro-engine* doing packet processing.

### 5.5 A Modular Software ESP Implementation

Not all routers emphasize performance and scalability. For example, "broadband router/gateway" products for the home or office support small-scale systems, but more advanced functionality (e.g., firewall protection). Similar goals apply for military network systems designed for quick field deployment. The design considerations for such routers are quite different from those of the high-end routers; instead of performance and parallelization, these systems focus on robustness, flexibility, and minimizing cost. To achieve this flexibility, many of these routers employ standard processors executing conventional operating systems (like Linux), modified to support the router features the vendor desires.

Click is a fine-grained modular software approach for implementing routers with advanced and evolving packet-processing features; as such, it is ideal for our purposes. The basic paradigm of click is to decompose router functionality into small functions that can be performed more or less independently, and encapsulate them in individual modules called *elements*. Elements are organized into a processing graph.

To incorporate ESP processing in routers such as click, we divided the ESP functionality into modules (click elements) and incorporated them into the click's packet flow graphs. One of the main problems we faced was how to modularize ESP. The click philosophy is for modules to be small and simple; this would suggest separate modules to validate the ESP header, retrieve values from the ESS, execute ESP instructions, and classify packets. However, a highly-decomposed implementation raises issues about performance and sharing of data between modules.

We implemented most of the ESP functionality, including instruction execution and state store management, in a single module called *ESPExec*. Some of the benefits of modularity are given up this way, but probably not too many. For example, it is not clear that there is much need for re-use of functional modules within ESP. In addition, the standard composition technique in click has well-known performance costs, and an integrated implementation avoids them.

To evaluate the overhead of our click implementation, we compared against the cost of basic IP processing in click. The click authors report the time to process an IP packet is roughly 2900 ns on

a PIII running at 700 MHz for a total forwarding rate of 345 Kpps. To baseline our system, a P4 running at 1.8GHz, we reran the same experiment as the click authors, and found the time to process an IP packet to be roughly 1300 ns for a total forwarding rate of 765 Kpps. Given this base-level performance, we are able to gauge the overhead required by ESP.

| count | compare | collect | rcollect |
|---|---|---|---|
| 660 ns | 702 ns | 1064 ns | 1525 ns |

Table 3: Added cost of ESP processing.

The additional ESP processing cost depends on the ESP instruction. The **count-with-threshold** and **compare** instructions are only 55% more costly than standard IP processing. The ESP **collect** instuction, on the other hand, involves more tags and thus increases the overhead by 81%. For the more complex **rcollect**, the overhead is about 116%. Although the overhead is substantial, the worst case throughput (with minimum sized packets) is still 354 Kpps or 181 Mbps, which is sufficient to keep all lines on a home router busy.

# 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.

## 6.1 LWP Specification

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.

## 6.2 Usage Scenarios

Using this and other LWP modules together with ESP, we have shown how to implement multicast based on unicast routing [7]. 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 [7].

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 12.

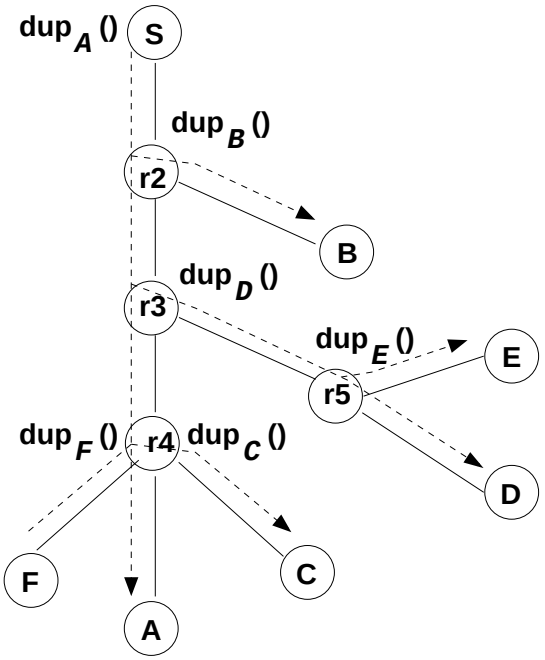



Figure 12: A ESP/LWP multicast tree.

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 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.

The details of our approach along with results are presented in [12].

# 7 Speccast: Generalized Routing and Forwarding

The speccast service is defined as follows. Let **N** denote a set of nodes, and let **P** be a set of predicates on nodes, i.e. functions from **N** to $\{true, false\}$. Each packet $m$ carries a predicate $m.dest \in \mathbf{P}$ called its *destination predicate*. The predicate specifies the set of nodes to which $m$ is to be delivered. That is, the nodes of the network conspire to deliver $m$ to all nodes $n$ such that $m.dest(n) = true$.

The speccast service is *best-effort*; it delivers the message $m$ to nodes satisfying $m.dest$, provided certain conditions are met (e.g., the network is not overloaded). Moreover, each packet is forwarded and delivered independently of all others, i.e. speccast is a *datagram* service.

This problem statement is very general. It says nothing about the nature of the predicates used to specify destinations, nor does it assume anything about which nodes satisfy what properties. In particular, no relationship is assumed between the predicates satisfied by a node and that node's location in the topology, and no underlying routing or addressing mechanism is assumed. Defining a solution to this problem involves specifying (i) the information carried in each packet in addition to the destination predicate; (ii) the information stored at each node, i.e. the "forwarding data structure"; (iii) the algorithm used by nodes to originate/forward packets, which takes as inputs the forwarding data structure and the information from the packet, and returns the set of channels on which the packet should be transmitted; and (iv) distributed algorithms for initializing the forwarding data structures at each node and updating them across changes in network topology (and possibly in predicate set, i.e. which nodes satsify which predicates). In general, the details of these algorithms will depend on the properties of **P**, and their efficiency will depend on which nodes in the topology satisfy which predicates.

Depending upon the properties of **P** and which nodes satisfy what predicates, the speccast service abstraction can subsume many traditional and emerging network-layer addressing/routing services. For example, *unicast* service is supported if **P** contains a *point predicate* for each node to which messages need to be delivered. (A point predicate is satisfied by exactly one node and thus can be used as an identifier.) Classical (any-source) *multicast* service can be supported by having a predicate in **P** for each multicast group address, which is satisfied by exactly the nodes belonging to that multicast group. A *publish and subscribe* service can be provided if there is a predicate in **P** for each published element, which is satisfied by all nodes subscribing to receive information corresponding to that element. In short, the flexibility of predicates allows the speccast service to be used in a wide variety of ways. In addition, this generalized service is useful for studying the relationships among address structure, locality, and routing/forwarding efficiency. An understanding of these relationships is crucial for the design of future network architectures.

## 7.1 Solution Evaluation Metrics

To compare solutions to the speccast problem, we define several measures of the cost of delivering a packet carrying a given destination predicate $p$ from a given source node $n$ to all nodes that satisfy $p$:

- **Network load**, i.e. the number of hops (links) over which a message is transmitted. Note this includes all links over which the message is forwarded, even if the link does not lead to

any node that satisfies $p$. We define the *network load ratio* to be the ratio of the total number of edges crossed by a message versus the number of links in the shortest-path tree from the source $n$ to all nodes satisfying $p$.

- **Delay**. We define the *delay cost* of a routing solution to be the sum, over all nodes $d$ satisfying $p$, of the number of edges crossed on the way from $n$ to $d$, when that solution is used. We define delay *stretch* to be the ratio of the algorithm's measured delay cost versus the minimum possible delay cost.

- **Network State**. We measure the *state cost* of a solution as the amount of information stored at all nodes combined. In general this is affected by the size of **N** and the structure of **P**.

- **Forwarding-time Computational Cost.** We measure the total *computational costs* as the sum across all nodes of the forwarding processing cost (i.e., the number of computation steps needed to decide where/how to forward the message). This cost depends heavily on the properties of **P**.

We have investigated two different solutions to the speccast problem. We describe each one in turn.

## 7.2 A Layered Solution

We have developed and evaluated a speccast solution for a particular predicate language: positive combinations of atomic propositions in disjunctive normal form. Examples of possible destination predicates in **P** for this language include $b$, $a \wedge b \wedge c$, $a \vee d$, and $(a \wedge b) \vee c$. Here $a$, $b$ and $c$ are atomic propositions, each of which is either satisfied or not by each node; such atomic propositions can represent attributes such as "command node", "altitude $> 5000$", etc. Negation is not included, but could be added fairly straightforwardly at the cost of additional state.

The basic idea of our approach for this language is to decompose the problem into two subproblems, and solve each subproblem with a separate *layer*. The first, lower-level subproblem is to deliver a packet to all nodes satisfying a single atomic proposition $b$. We refer to the solution to this subproblem as the "base layer". The second, higher-level subproblem is to use the base layer to deliver a packet to the set of nodes satisfying any given disjunction of conjunctions (i.e. any boolean expression in disjunctive normal form) while minimizing delivery to nodes that do *not* satisfy the predicate. We refer to the solution to the second subproblem as the "composition layer"; the composition layer runs on top of the base layer.

### 7.2.1 Base Layer: Deliver to Atomic Propositions

The problem of delivery to nodes satisfying each atomic proposition can be solved using an any-source multicast service (i.e. classical IP multicast), simply by using a multicast group address per atomic proposition and having each node that satisfies an atomic proposition join the corresponding multicast group. Unfortunately, for efficient delivery, both the base layer *and* the composition layer need to process the packet at each hop. Because this is not possible using a legacy multicast service, we developed our own base layer, which constructs a spanning tree per atomic proposition $b$; this spanning tree contains all the nodes that satisfy $b$. In general this tree will also include nodes that do not satisfy $b$, i.e. it is a Steiner tree.

Every node in the network maintains a base layer forwarding table containing one entry per atomic proposition. If node $n$ is part of the tree for attribute $b$, the table entry for $b$ indicates $n$'s neighbors in the tree. If $n$ is *not* part of the tree for $b$, the entry for $b$ indicates the neighbor that is "closest" to that tree among all of $n$'s neighbors.

When a packet needs to be forwarded to nodes satisfying $a$, the node looks up the entry for $a$ and transmits a copy of the packet over all associated interfaces, except the one on which the packet arrived, if any. Thus, once a packet reaches the tree for atomic proposition $a$, it is forwarded to all nodes satisfying $a$. The use of a shared tree minimizes overall network load and network state is reduced, possibly at the cost of increased delay.

The forwarding table information for the base layer can be established using a simple distance-vector-type protocol in which each node announces to its neighbors the information it knows about every attribute tree: the attribute, the distance from the node to the tree (0 if the node satisfies the attribute, infinity if it does not know of a path to the tree), and a tree ID (which can be chosen randomly). Initially each node considers itself the only member of the tree for exactly those atomic propositions that it satisfies. When a node learns of another tree for the same atomic proposition with a lower ID, it "joins" that tree; the result is that eventually the two trees merge into one (for details, refer to [21]). Nodes on the tree also include an estimate of the tree size; this is used in the composition layer. This process does not produce a minimal tree; however, the problem of constructing a minimum Steiner tree is well-known to be computationally hard.

### 7.2.2 Composition Layer

To reach the set of nodes satisfying a boolean expression in disjunctive normal form—recall that any arbitrary boolean expression can be expressed in disjunctive normal form—the composition layer uses the base layer in a two-step process. First, the originating node selects a single atomic proposition from each conjunction in the expression. Then a copy of the packet is forwarded to all nodes that satisfy each such selected atomic proposition, using the base layer. This process of "weakening" the destination predicate results in some nodes receiving the packet that do not satisfy its original destination predicate, but every node that satisfies the original predicate satisfies the weaker version and will receive the packet. Thus we are trading some additional network load for a reduction in the amount of forwarding state needed.

For example, the predicate $D = (a \wedge b) \vee c$ might be weakened to $a \vee c$ or to $D' = b \vee c$. The services of the base layer are then used to send the packet to nodes satisfying the weaker predicate, i.e. all nodes satisfying $b$ and all nodes satisfying $c$. Forwarding by the base layer is based on $D'$, although the original predicate $D$ remains in the packet. At each node $m$, the predicate $D$ is checked to see whether it is satisfied by $m$; if so, the packet is delivered to the applications layers of the node.

To decide which atomic proposition to select from each conjunction, we use a heuristic, based on the combination of tree size and distance from the originating node to the tree; both are approximately proportional to the network load (number of links traversed) that will result from delivering the packet via that tree. Therefore we choose the atomic proposition from each conjunction that minimizes the sum of these values. After this weakening step, the resulting list of selected atomic propositions is placed in the packet along with the original destination predicate, and the packet is forwarded to each, using the base layer. (Recall that each node has a forwarding table entry for each attribute, regardless of whether it satisfies that attribute or not.)

As an example, consider the predicate $D = (a \wedge b) \vee (c \wedge d)$. Suppose that $a$ and $c$ are the

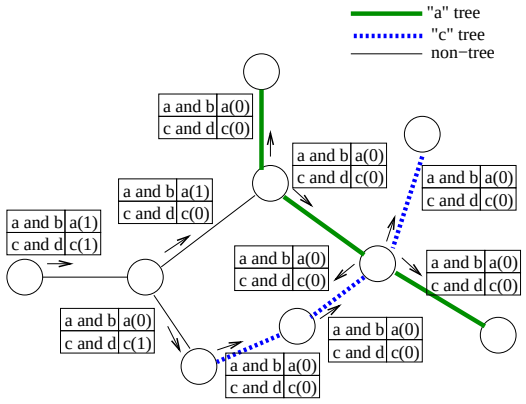



Figure 13: Packet forwarding example

atomic propositions with the smallest sum of distance-to-tree and tree size. The packet is forwarded with a header containing a list of disjunctions, each consisting of a conjunction and an indication of the atomic proposition selected from that conjunction, plus a *pending bit* for that atomic proposition (indicated in the figure by the number in parentheses):

| $a \wedge b$ | $a(1)$ | $c \wedge d$ | $c(1)$ |
|---|---|---|---|

The pending bit for $a$ indicates whether the packet still needs to reach the tree for $a$.

When forwarding a packet, the composition layer at node $n$ carries out the following steps.

1. For each selected atomic proposition $a$ such that $n$ is part of the tree for $a$: forward the packet out all interfaces belonging to that tree (except the one on which the packet arrived), clearing the pending bit for $a$ in the process.

2. For each selected atomic proposition $b$ such that $n$ is *not* part of the tree for $b$ *and* the pending bit is set: add the interface toward $b$'s tree to the list of outgoing interfaces for the packet.

3. For each interface in the list created above, forward a copy of the packet on that interface, clearing the pending bits for each atomic proposition whose tree does *not* lie in that direction.

These steps ensure that the packet follows the shortest path to the tree for each selected atomic proposition. They also allow a single copy of a packet to follow a single path toward multiple trees until the paths diverge.

Note that the first step above results in packets traversing the *union* of trees that intersect. That is, if a packet is traversing one spanning tree and reaches a node that also belongs to the tree of one of the other selected atomic propositions, the composition layer duplicates the packet and forwards the duplicate(s) along the latter tree as well. This lets packets reach the nodes satisfying $a$ faster by taking multiple routes: by following the "gradient" of nodes that are not on the $a$ tree, and via the trees of other selected atomic propositions that intersect the $a$ tree. (It is this optimization that requires that all selected atomic propositions be carried in all copies of the packet.) Figure 13 illustrates the use of this optimization, as well as the use of the pending bits.

Because packets can encounter a selected atomic proposition's tree multiple times, care must be taken to avoid excessive duplication and waste of bandwidth. (For example, trees that intersect

in two places would result in packets looping forever.) For this purpose our approach requires a *duplicate suppression* mechanism that keeps packets from being forwarded over the same link more than once in the same direction. This mechanism can be implemented in several ways. One possibility is to use *ephemeral state processing* (see Section 5): Each packet is originated with a unique tag; a packet is forwarded, it leaves a bit of ephemeral state on each interface it traverses; packets carry an ESP instruction which causes a packet to be dropped if it reaches an interface where its tag is already stored. Other well-known solutions are to record each interface traversed in the packet, say using a Bloom filter, or to have packets carry a hop-limit field that is decremented each time the packet is forwarded, and drop packets when the hop-limit field reaches zero.

### 7.2.3 Evaluation of Layered Approach

To evaluate the performance and overhead costs of this implementation approach, we conducted simulation experiments on two versions. The first is the basic two-layer service, with an optimization that uses additional *filter* state to keep track of which branches of the tree for one atomic proposition lead to nodes that do not (or do) satisfy other atomic propositions. The second version did not use filters, but did incorporate support for *hierarchical atomic propositions*, i.e. attributes that are satisfied *only* by nodes satisfying other attributes. These can be thought of as address prefixes: a prefix corresponds to an attribute that is satisfied by nodes whose addresses match the prefix. The advantage of structuring the predicate set in this way is that it gives the ability to refer to groups of nodes with higher-level atomic propositions, while still retaining the ability to specify any node individually. We call the former version *non-hierarchical speccast*, and the latter *hierarchical speccast*. (Note that the total number of atomic propositions is exponentially larger in the hierarchical case.)

To measure the performance and cost of each of this speccast service, we carried out two experiments. The first experiment investigates the cost of using the various services to implement *unicast*; the second considers the use of the services for *multicast*. In both cases we measured the network delay, network load, and network state. For comparison, in addition to our two versions of speccast, we also simulated other routing and forwarding approaches, including: a generic shortest path first routing algorithm (SPF) that simulates traditional unicast (and IP multicast) routing; a generic publish-subscribe service based on a *broker overlay*, in which a packet is first transmitted to the nearest broker, which forwards it to all nodes it knows about that satisfy the destination predicate; and a structured (peer-to-peer) overlay similar to Tapestry.

In the unicast experiment, each node in the network was assigned a unique "address"—that is, a combination of attributes satisfied only by that node. How the address was interpreted depended on the routing approach. Both the Broker and SPF model treat addresses as "flat" (unstructured) bits strings. In both the Speccast services and Tapestry, the address is regarded as a sequence of base $b$ digits. Since we have a fixed number of nodes $N$, the number of digits needed in the address is $d = log_b N$. In the Tapestry simulation, the digits are used directly by the routing algorithm as described above. In the non-hierarchical speccast simulation, we associate one atomic proposition with each (digit, position) pair. Each address can thus be uniquely represented as a conjunction of $d$ atomic propositions. Thus the number of atomic propositions in the system is $b \times d$. To measure the effect of the base $b$ on the network state, load, and delay, we varied $b$ from 2 to 30 in our experiments.

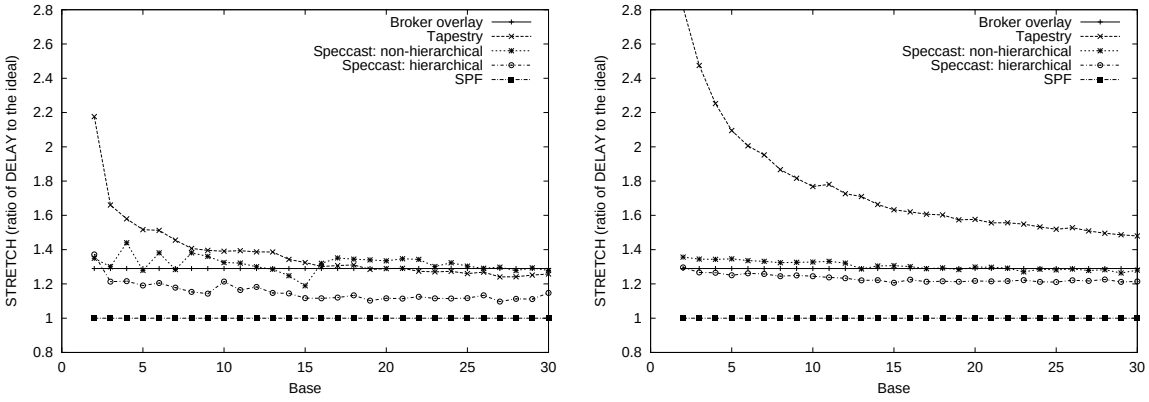
In the hierarchical speccast simulations, the set of atomic propositions contains a single atomic proposition for each string of $k$ digits, for $k$ between 1 and $d$. These atomic propositions are ar-

ranged hierarchically in essentially the prefix-based structure described earlier. So the attribute corresponding to the string 1223 has a "parent" attribute corresponding to the string 122.

Before presenting the results, we must explain one other concept. *Locality* is a correlation between a node's position in the topology and the set of predicates satisfied by a node. With locality, knowing that node $n$ satisfies atomic proposition $a$ removes some uncertainty about the probability that other nodes near $n$ in the topology will also satisfy $a$.

To evaluate the affect of locality on performance and overhead, we ran two versions of each experiment using transit-stub graph models generated with the well-known GT-ITM topology modeling package. In the first test, we randomly assigned addresses to nodes. As a result, the probability of nodes with "similar" addresses being near one another was relatively low (i.e., no locality). In the second test, addresses were assigned according to node numbers in the GT-ITM graph structure. Because of the way transit-stub graphs are constructed, nodes near each other in the topology have similar node numbers, and so received similar addresses in the second test. As a result, nodes in the same network domain usually have a common address prefix. Each topology consisted of 3080 nodes comprising four transit domains averaging 14 nodes each, with each transit node connecting to 3 stub domains averaging 18 nodes each. Each graph also contained 12 extra transit-stub edges and 56 additional stub-stub edges; these extra edges increase the number of paths between stub nodes in the graph, and thus provide more diversity of routing.

We also simulated the use of speccast to provide multicast service in a straightforward way using atomic propositions in place of multicast addresses. To send a packet to the group, a sender simply addresses the packet to the group's atomic proposition. Unlike conventional multicast, our multicast service is more expressive and could also be used to send packets to the unions or intersections of groups. Using hierarchical speccast service, groups could also be partitioned into subgroups, which might be useful, say, for subcasting.

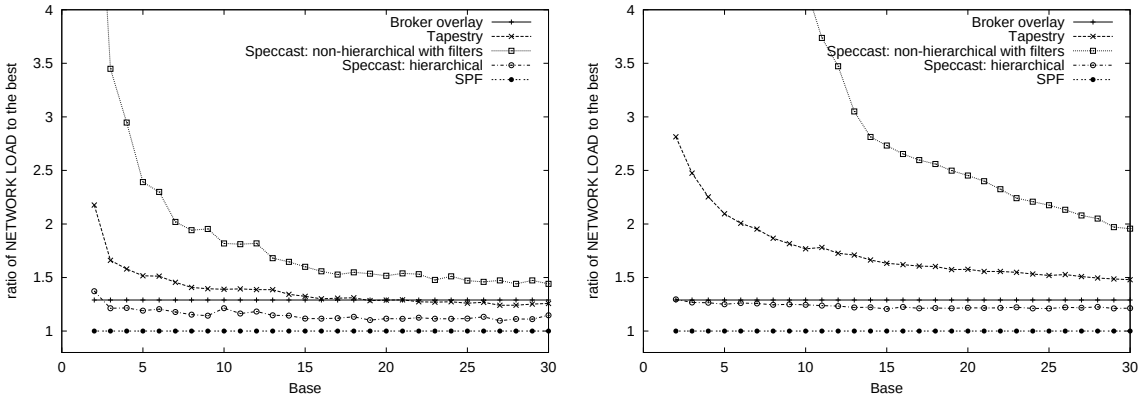



(a) With locality

(b) Without locality

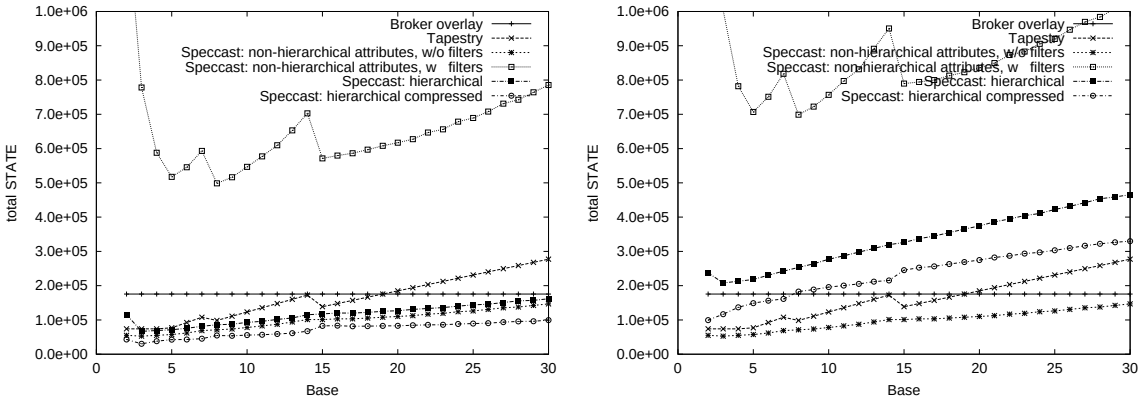
Figure 14: Normalized network delay for unicast delivery.

The graphs in Figures 14-16 plot the metrics discussed earlier as a function of the "base" of the (unicast) addresses. Figure 14 shows the network delay stretch, computed as the ratio of the measured delay to the optimal (shortest path) delay, when addressed are assigned with and without locality. Figure 15 shows the network load (measured in number of packet-hops) imposed on the




(a) With locality (b) Without locality

Figure 15: Normalized network load for unicast delivery. methods




(a) With locality (b) Without locality

Figure 16: Network state required for unicast delivery.

system by each algorithm. Unlike delay, the network load is noticeably affected by locality. Finally, Figure 16 shows the amount of state each algorithm requires in the network. Although speccast competes fairly well in terms on delay and network load, its greater flexibility comes at the cost of additional forwarding state kept in the network. Without locality, each tree contains many nodes that do not satisfy the atomic proposition, but yet must maintain the state. With locality, the number of such nodes drops significantly, as does the network state.

Figure 17 shows the network delay and network load when speccast is used for multicast delivery. The results confirm our expectations: Because speccast constructs a shared distribution tree,

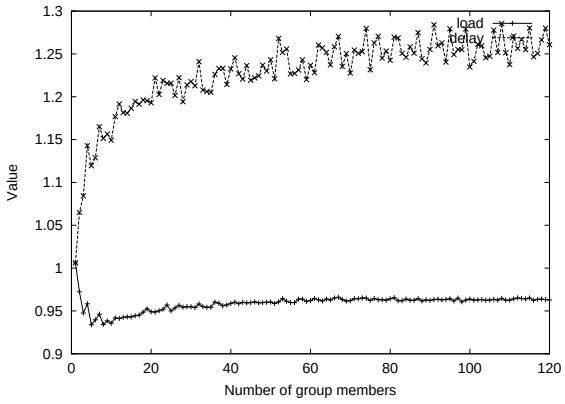



Figure 17: Normalized delay and network load for delivery to groups of receivers.

it performs much like a core-based multicast tree, reducing the network load (even below 1) at the expense of longer delays (stretch $> 1$).

## 7.3 A Link-State Implementation

In addition to the distance-vector approach described above, we have implemented speccast using a different approach based on a link-state routing protocol and source-routing. The advantages of this approach are that (i) it requires very little knowledge about the language used to specify destination predicates; (ii) it does not assume or require the existence of *any* globally-assigned node identifiers; (iii) it can support the aggregation of node predicates within areas, making a subgraph of the network look like a single node, and thus can in principle support large-scale networks (though we have not implemented this capability); and (iv) it admits the possibility of representing destination predicates as simple programs in packets.

Our implementation, which is written in Java, comprises code (i.e. a seperate class) for each of the routing and forwarding components, as well as a *Hello protocol*. The Hello protocol enables a node to locate its neighbors in the network, and assigns (random) identifiers to each link. The *routing algorithm* performs two functions. First, each node periodically transmits local routing information—in the form of a link-state announcement, or LSA—to its neighbors. Each LSA contains the node predicate for the originating node, and a list of links to which it is connected. Second, information from incoming LSAs is used to construct a *network map*, including each node and the predicate it satisfies, as well as the links connecting nodes to each other.

Packets are forwarded using a source routing approach. To originate a packet, a node consults

its network map to determine which node(s) satisfy the destination predicate. It then determines a (branching) path to be followed by the packet to all such nodes. This tree is encoded in the packet header. When a packet arrives at a node, the destination predicate is checked to see whether the node satisfies it; if so it is delivered to the higher level. Then the packet is forwarded on the interfaces indicated by the tree contained in the packet.

Our implementation uses the simple predicate language described in the previous subsection, i.e. both node and destination predicates are in the form of disjunctions of conjunctions of attributes. A packet is delivered to a node if (and only if) it is *possible* that the node satisfies the destination predicate, i.e. if at least one conjunction in one of the predicates contains all attributes of some conjunction of the other. Attribute names are encoded as case-sensitive strings of letters and numbers. Conjunction and disjunction are indicated by `&` and `|`, respectively (e.g. "`abc&def | qwe & asd`").

The protocol is implemented as a Java Class ("Node"), which includes an API with `send()`, `setPredicate()` and `registerReceiver()` methods. (The receiver interface must be implemented by the higher-level protocol.) An instance of the node class, when started, broadcasts messages to its neighbors on all configured interfaces, and is then ready to forward packets. Our code contains several sample applications: *NetTest*, (to test with a network topology created by GT-ITM) and a simple 3 node topology test.

# 8 Other Technologies

In addition to the primary technologies described above, the ActiveCast project has produced other technologies, which we describe next.

## 8.1 FPAC: Fast, Fixed-cost Packet Authentication

One of the problems that arises in providing enhanced services like example concast, or any other service in which some packets receive service beyond the normal best-effort forwarding, is that of ensuring that only packets that are entitled to the service receive it. In particular, if resources (e.g. bandwidth) are reserved for one user's packets, and some other (malicious) user sends "spoofed" packets, impersonating the first user, the spoofed packets may use up the reserved resources, thus denying service to the legitimate user. The usual approach to detecting spoofed packets is to use a cryptographic message authentication check (MAC) code. However, any per-packet check must be (i) capable of being performed at wire speed for minimum-sized packets, and (ii) immune to replay attacks. If packets cannot be processed at least as fast as they arrive, then the check itself is vulnerable to a denial-of-service attack.

We therefore developed and implemented a lightweight authentication code called FPAC, which can be implemented for a fixed per-packet cost. FPAC, which is described in a paper presented at the INFOCOM 2002 conference [11], is designed specifically to protect against unauthorized access to resources that are consumed in an amount proportional to a packets length. The cost is fixed because only a fixed-size portion of the packet (i.e. a "shim" header between the IP header and the transport header) is actually checked for authenticity. The advantage of this approach is that it can be implemented without requiring the entire packet. The tradeoff is that FPAC cannot be used for end-to-end integrity checks; however, there are well-known good solutions to the problem of end-to-end authenticity/integrity verification.

FPAC uses a symmetric-key cipher to hash a shared secret. It requires that the shared secret be distributed to all nodes that have reserved resources for a flow. We did not develop a specific solution for the key-distribution problem, although we believe the secure concast signaling protocol could be used for the purpose.

### 8.2 A Generic Grouping Service

We have also studied the design and implementation of a generic, distributed *grouping* service. The service is intended to help large-scale applications distribute participants among groups. In recent years, the design and use of various forms of *overlay networks* has been a topic of interest in the networking field. Overlay networks iterate the "catenet" idea that led to the original Internet, by treating the Internet itself as a lower-level network and building a higher-level topology using Internet-level "virtual links". Such topologies, however, can be very inefficient if they are constructed in such a way that neighbor relationships at the overlay level do not respect the topology of the underlying network. Our grouping service is intended to help with this problem—and related problems, such as determining which multicast receivers should use which recovery server—by taking application-level information from each participant, combining it with network-level information, and returning an indication of which participants should be in the same groups. More precisely, the service allows the application to define a "distance" metric on application information, and combines that with a network-level metric, in such a way that some combination (also specified by the application) of the two metrics is minimized.

The grouping services we have studied so far are designed to leverage advanced/programmable network services. Along with the design of a general "grouping API", we have published descriptions of both concast-based [14] and ESP-based [17] grouping service implementations.

## 9 Summary of Work Products

In addition to the technologies and prototype implementations[5] already described in this report, this project has produced the following outputs:

- 20 refereed or invited publications, which have appeared in top venues for networking research. All publications which have appeared to date are listed below under the heading "References"; at least two more are currently in preparation for submission.
- One patent application was filed, for "System and Process for Providing Auxiliary Information for a Packet-Switched Network of Shared Nodes Using Dedicated Associative Store", inventors Kenneth L. Calvert and James N. Griffioen.
- Work done as part of this project has made up or will appear in the PhD thesis of at least four students: Su Wen, Amit Sehgal, and Leonid Poutievski of the University of Kentucky, and Youngsu Chae of the Georgia Institute of Technology.
- Work done on various aspects of this project formed the Masters Degree Project for two students, Srini Venkatraman and Chetan Singh Dillon of the University of Kentucky.

Some of the text of this report was contributed by coauthors of papers in the list below.

[5] All of which are available at the project web site ⟨`http://protocols.netlab.uky.edu/~acast/`⟩

## References


[1] 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.

[2] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen. Concast: Design and Implementation of a New Network Service. In *Proceedings of the 1999 IEEE International Conference on Network Protocols*, Toronto, November 1999.

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

[4] 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 2001.

[5] Y. Chae, K. Guo, M. Buddhikot, S. Suri, and E. Zegura. Silo, rainbow, and caching token: Schemes for scalable, fault tolerant stream caching. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(7), September 2002.

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

[7] Su Wen, J. Griffioen, and K. Calvert. Building Multicast Services from Unicast Forwarding and Ephemeral State. In *Proceedings of the 4th IEEE International Conference on Open Architectures and Network Programming* (OPENARCH '01), Anchorage, Alaska, April 2001.

[8] K. Calvert, J. Griffioen, S. Natarajan, B. Mullins, L. Poutievski, A .Sehgal, S. Wen. Leveraging Emerging Network Services to Scale Multimedia Applications. In *Proceedings of 2001 IEEE International Conference on Computer Communications and Networks*, Scottsdale, AZ, October 2001.

[9] Su Wen, J. Griffioen, and K. Calvert. Building Multicast Services from Unicast Forwarding and Ephemeral State. *Computer Networks* 38(3), February 2002.

[10] M. Bond, K. Calvert, J. Griffioen, B. Mullins, S. Natarajan, L. Poutievski, A .Sehgal, S. Venkatraman, S. Wen. ActiveCast: Toward Application-Friendly Active Network Services. In *Proceedings of DARPA Active Networks Conference and Exposition*, San Franciso, May 2002.

[11] K. Calvert, S. Venkatraman, J. Griffioen. FPAC: Fast, Fixed-cost Authentication for Access to Reserved Resources. In *Proceedings of IEEE INFOCOM 2002*, New York, June 2002.

[12] 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), New York, June 2002.

[13] K. Calvert, J. Griffioen, S. Wen, Lightweight Network Support for Scalable End-to-End Services. In *Proceedings of ACM SIGCOMM 2002 Symposium*, Pittsburgh, August 2002, pp. 265–278.

[14] A. Sehgal, K. Calvert, J. Griffioen. A Flexible Concast-based Grouping Service. *Proceedings of the 2002 International Working Conference on Active Networks* (IWAN '02), Zürich, December 2002.

[15] M. Bond, J. Griffioen, C. Dillon, K. Calvert. Designing Service-Specific Execution Environments. In *Proceedings of the 2002 International Working Conference on Active Networks* (IWAN '02), Zürich, December 2002.

[16] Su Wen. Supporting Group Communication on a Lightweight Programmable Network. Ph.D. Thesis, University of Kentucky, January 2003.

[17] A. Sehgal, K. Calvert, J. Griffioen. A Generic Set-formation Service. In *Proceedings of the 6th IEEE International Conference on Open Architectures and Network Programming* (OPENARCH '03), San Francisco, April 2003.

[18] K. Calvert, J. Griffioen, B. Mullins, S. Natarajan, L. Poutievski, A .Sehgal, S. Wen. Leveraging Emerging Network Services to Scale Multimedia Applications. *Software: Practice and Experience*, 33:1377–1397, 2003.

[19] B. Mullins, J. Griffioen, K. Calvert. Multicast TCP via Concast Merged Acknowledgements. In *Proceedings of 2003 IEEE International Conference on Computer Communications and Networks*, Dallas, October 2003.

[20] N. Imam, J. Li, K. Calvert, J. Griffioen. Challenges in implementing an ESP Service. In *Proceedings of the 2003 International Working Conference on Active Networks* (IWAN '03), Tokyo, December 2003.

[21] L. Poutievski, K. Calvert, J. Griffioen. Speccast. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, March 2004.

[22] M. Muthulakshmi, J. Heath, K. Calvert, J. Griffioen. ESP: A Flexible, High-Performance, PLD-Based Network Service. In *Proceedings of 2004 IEEE International Conference on Communications* (High-Speed Networks Symposium), Paris, June 2004.