Building Multicast Services from Unicast Forwarding and Ephemeral State

Su Wen, James Griffioen, Kenneth L. Calvert
Department of Computer Science
University of Kentucky, Lexington, KY 40506-0046

Abstract— **We present an approach to building multicast services at the network layer using unicast forwarding and two additional building blocks: (i) ephemeral state probes, *i.e.* extremely lightweight distributed computations based on a time-bounded associative memory; and (ii) the ability to inject or enable packet processing functions that modify router behavior in a very limited way. In our approach, senders and receivers use ephemeral state probes to determine where to inject functionality. A special function that duplicates packets matching a particular pattern and forwards them to a specific destination is then instantiated at the desired network location.**

Our approach eliminates the need for sophisticated multicast routing protocols and gives the end-systems control over the multicast service, allowing the application to tailor the service to its needs. At the same time, our approach creates efficient forwarding paths by using ephemeral state probes to determine (only) the relevant aspects of the network and group topology.

We present two multicast implementations: one builds a multicast tree with centralized control, another provides the traditional IP multicast abstraction. Both implementations can be done in a simple and scalable manner with minimal added functionality in the routers beyond unicast forwarding.

Keywords: multicast, ephemeral state, programmable networks, active networks

I. Introduction

Conventional Internet multicast is a useful service abstraction for many applications, but for various reasons it has been slow to be widely deployed. Among the problems with conventional IP multicast are the complexity and variety of routing protocols used, application-to-abstraction mismatches, lack of access control and so on.

To overcome some of these problems, various researchers have proposed and are exploring new one-to-many service abstractions that scale better than conventional IP multicast [11], [1]. A one-to-many abstraction is often a more appropriate service for applications that have a single source and want to record information about receivers, or regulate receiver participation. However, these approaches also assume that the implementation of the service is wired into the network infrastructure, making it difficult to change, extend, and evolve.

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

Other researchers are exploring application-level approaches [10], [4], [12], which implement multicast above the network layer, using only unicast forwarding. Application-level multicast has the advantage that the application has complete control over the way the service is implemented. Unfortunately, these schemes suffer from other problems — including limited scalability, inefficient distribution trees, and vulnerability to node failure — that stem from the creation of whole new topology above the network-layer "cloud".

An ideal solution would combine the flexibility and control of the application-level solutions with the robustness and efficiency of the network-based solutions. In this paper we present an approach that uses a mechanism for probing network topology together with a very simple code-instantiation interface, that achieves a balance of flexibility, efficiency, and robustness. The basic idea of the approach is that the topology-probing mechanism identifies a point for a new receiver to be grafted onto the multicast tree; a small piece of code is then instantiated at that point; the code duplicates packets flowing on the existing tree branch and forwards them to the new receiver.

The approach is flexible because maintenance of the multicast tree (and the exact nature of the installed duplication code) is left to the end systems; it is as robust and efficient as the underlying unicast forwarding. Depending on how code instantiation is handled, it can also be made scalable.

Our methodology is based on two building blocks whose existence we posit: (1) *ephemeral state probes* [3], a facility for very light-weight computations that collect information about the network (in particular, topology information), and (2) *lightweight packet processing modules*, a special set of functions that can be instantiated by end-host applications through a simple code activation mechanism to modify the behavior of strategic routers in the network. Given these two building blocks, we show how information collected via ephemeral state probes can be combined with unicast forwarding to "bootstrap" application-controlled multicast services.

The flexibility of our service allows applications to define application-specific multicast abstractions/services and routing algorithms rather than being confined to a predefined IP multicast abstraction [8] and a hardwired multicast routing algorithm (e.g., DVMRP [15], PIM [9], [7], BGMP [17], Express [11], Simple Multicast [13]).

The rest of this paper is organized as follows. In the next two sections, we describe the building blocks of our system, namely the ephemeral state probe facility and the lightweight packet processing modules. Next we show two different application-controlled implementations of single-source multicast: a sender-oriented protocol in Section IV, where the sender controls the instantiation of the duplication/forwarding code; and a receiver-oriented protocol in Section V, where receivers control the instantiation. The characteristics of the receiver-oriented protocol are analyzed and evaluated in Section VI. Section VII discusses related work and Section VIII concludes the paper.

II. Ephemeral State Probes

The first building-block service, *ephemeral state probes* (ESP), is a simple mechanism for collecting information about the network. Ephemeral state probes (ESPs) are packets sent by end-systems that can initiate computations on network routers and return information about the inside of the network. In particular, ESP computations can provide answers to simple topology-related questions such as "At what node, if any, do the paths from A to B and from X to Y intersect?", and "What is the nearest branch point to multicast receiver R?". Because the mechanism is supported by routers that cooperate to produce the answer, it is efficient for the extraction of *small* amounts of network information through sending and receiving individual packets.

The use of ephemeral state for collecting information from the inside of the network has been proposed elsewhere [3]; here we merely sketch the ideas and capabilities of the service. We assume that each node participates in a unicast routing protocol and can make the routing decisions needed to get a packet to its destination.

The ESP mechanism has two components: the ephemeral state store, which makes it possible to accumulate data from different packets inside the network without overhead; and the packet-initiated probe computations that use the store to accumulate and process data in the network.

A. Ephemeral State Store

The ephemeral state store is an *associative memory* that binds fixed-size *values* to fixed-size *tags*. It is called ephemeral because the binding persists for only a fixed (short) time, called the *ephemeral state lifetime*. Each node in the network maintains its own independent ephemeral state store, and all nodes use the same ephemeral state lifetime.

The benefit of ephemeral state is that these bindings can be used as dynamically allocated and garbage-collected *variables* in packet-initiated computations. Tags are large enough so that the number of possible tags is vastly larger than the capacity of the store. Tags are chosen by end systems randomly; the choice should be truly random to ensure that (with high probability) no other computation is using the same tag. Unlike soft state [5], once a (tag, value) binding is created, it cannot be refreshed to extend its lifetime. Because the binding disappears after a short time, there is zero management overhead. The only constraint is that a computation must run to completion before the binding disappears; for the rest of this paper we assume that the ephemeral state lifetime is sufficiently long that this constraint can always be satisfied. (Earlier we have investigated the conditions under which this assumption is justified, and shown that they are reasonable [3].)

B. Probe Computations

The ephemeral state store provides the ability to assign, retrieve, and modify the values associated with tags. The other part of the ESP mechanism is a set of probe computations that make use of those values. Each probe computation is a simple calculation that terminates in a bounded time. These computations are initiated by ESP packets, which carry information needed by the computation. Each ESP packet contains, an *opcode* identifying the computation to be initiated, zero or more *tags* (variable names) to be used in the computation, and zero or more *immediate values* to be used in the computation.

ESP computations may also involve other information supplied by the router, such as a router identifier (i.e. one of its IP addresses) or information about its state (e.g. queue occupancy).

As an ESP packet travels through the network, each router recognizes it as such and the specified computation (which must be pre-installed at the router) is carried out.[1] As part of the resulting computation, the packet is either forwarded toward its destination or dropped; ephemeral state probes do not create new packets, nor can they modify the IP header fields of a packet. It may be useful to think of the set of probe computations as the instruction set of a virtual machine. By sequencing the instructions properly, interesting and useful distributed computations can be constructed.

C. An Example

Throughout this paper we use the following notation. If t is a tag, then $(*t)$ denotes the value bound to t; the absence of any value will be modeled as t being bound to the special value $\perp$. Thus $(*t) = \perp$ indicates that the "tag name" t is unused. The notation $(*t) := e$ denotes the following atomic operation: evaluate the expression e (which may contain $(*t)$), and bind the resulting value to t.

In what follows, tags will often be carried as fields in the packets that initiate computations. In that case we use the notation *pkt.field* to denote the tag name (i.e. the name of the variable). Please note that $(*pkt.field) := e$ binds a value to the tag named by *field*, and does *not* change the packet. To change the contents of the packet we use the notation $pkt.field := e$, which writes the value e into the *field* location in the packet.

We explain the use of ESP computations through an example. Suppose we want to determine whether two (unicast) paths through the network, say the path from A to B and the path from X to Y (illustrated in Fig. 1), have any routers in

[1] For many purposes it is not necessary for *all* routers to support the ESP facility, but for simplicity we assume here that they do.

common. If the paths intersect, we want to know the address of the first router in the intersection. The solution involves two ESP computations. The first is a "setup" computation initiated by a packet sent from A to B; it assigns the value 1 to the tag T in the state stores of all nodes on the path from A to B. The "setup" computation is shown in Fig. 1a.

if $(*pkt.T) == \perp$
    $(*pkt.T) := 1;$

At each node, if the tag T does not have a value, it is assigned the value 1. The ESP packet is forwarded at the end of the computation.

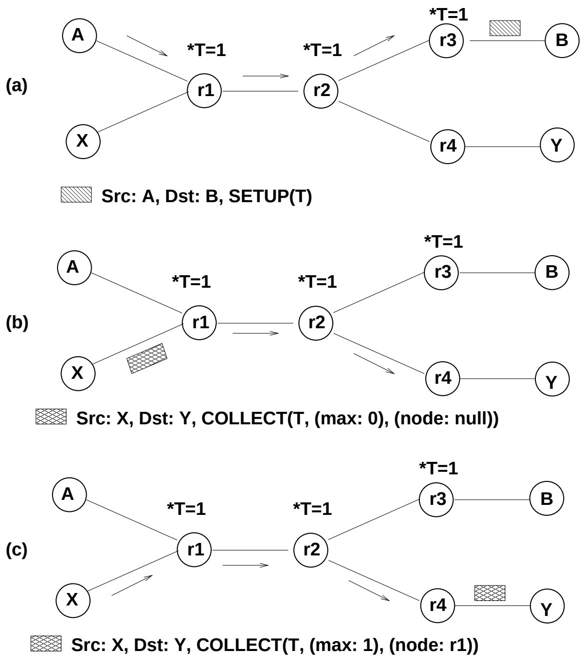

Fig. 1. Finding path intersection

The second computation uses the bindings left by the first computation to identify a node in the intersection, if one exists. A short time after the "setup" ESP is sent, an ESP packet is sent from X to Y to initiate a "collect" computation at routers along the path from X to Y. The ESP packet contains two variables, max and $node$, initially set to 0 and NULL respectively. During the "collect" computation, the value bound to tag T at each node is compared to the value max carried in the ESP packet. If $(*T) > max$, it becomes the new value of max, and the address of the current node is placed in the packet; in any case the packet is forwarded:

if $(*pkt.T) \neq \perp$ and $(*pkt.T) > pkt.max$
    $pkt.max := (*pkt.T);$
    $pkt.node := this\_node\_addr;$

When the collect message arrives at Y, it contains the address of node $r1$, the node in the path intersection closest to X (The ID of the node closest to Y can be obtained by replacing ">" with "$\geq$" in the collect computation).

This is just one simple example to illustrate the use of computations to gather information from multiple nodes. Clearly, for the mechanism to be practical, some coordination between the end-system nodes is needed. In Sections IV and IV we introduce other ESP computations needed for constructing the multicast tree.

III. Lightweight Processing Modules

The point of the ESP facility is to enable end systems to locate the best points to invoke specific functionality. In our approach, the available functionality is defined by router-supported *lightweight packet processing modules*. Our expectation is that a small set of predefined processing modules will suffice for a variety of network services. The code that makes up a processing module might be built into a router, or dynamically loaded via a code installation mechanism [2]. In any case, processing modules are intended to be small, authenticated, code segments that terminate in bounded time.

Lightweight processing modules operate by identifying packets that match a particular filter pattern, applying basic processing (e.g. duplication) to those packets, and then forwarding the zero or more resulting packets using standard unicast routing. Each processing module is defined by the following information
- `Code:` instructions to execute.
- `Classifiers:` a list of packet filters that identify which packets this module should intercept.
- `Parameters:` a set of 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.

A processing module may be instantiated multiple times on a router, where each instantiation differs in either the classifier, parameters, or timeout. Each processing module is treated as soft state in the router. If the module is not "refreshed" within the *timeout* period it is eligible for, it will be automatically removed.

Fig. 2 illustrates the point at which module processing occurs in the router. Note that both ESP and lightweight module processing occur on the input side, prior to forwarding. This is important for our use of ESP to "bootstrap" the multicast tree configuration. Also, we require that the ESP facility have access to certain information about the activated lightweight modules, in particular whether a function with specific parameters is installed.

The basic operations related to lightweight processing modules are:
- `Instantiate Module:` Instantiate a module with a specific classifier, parameters and timeout. This might simply

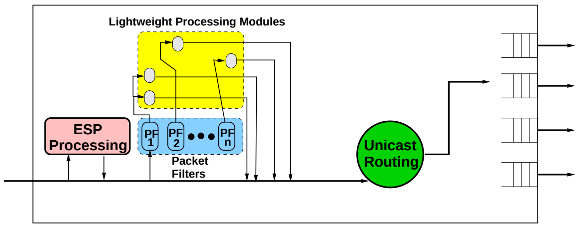

Fig. 2. Router Support: Ephemeral State Probe computations and Lightweight Processing Modules are both implemented on the input side.

involve a control message from the end-system to the router to enable the built-in module, or it might involve dynamically loading the code from the end-system [2].

- `Terminate Module`: Terminate a particular instantiation of a module.
- `Replace Module`: Atomically replace a module with a different one. This is equivalent to a `Terminate Module` operation followed by an `Instantiate Module` operation.
- `Refresh Module`: Refresh the timeout value of a module.

End-systems issue control messages that invoke these operations on the desired routers. By allowing (trusted) end-systems to enable and disable data-path processing at internal nodes in the network, protocols are administered from the edges of the network as opposed to being hardwired into the network itself. This separation of data and control facilitates application-specific and application-optimized implementations of historically hardwired network services. In sections IV and V, we show that these basic operations can be used to implement different flavors of multicast.

A. Dup(): An Example Processing Module

The basic lightweight processing module dealt with in this paper is the duplication function *dup()*. The purpose of this function is to copy each packet that matches the classifier filter specified at instantiation, and forward the copy to a particular unicast address (also specified at instantiation time).

In our scheme, all transmissions are done with unicast packets. The dup() function classifier looks into the IP option field of the IPv4 header (or the extension header in IPv6) to find multicast identification information.[2] A simple example of a multicast group ID is the (IP address, port number) pair. However, the format of the multicast group ID can be defined by the application. It may contain other information such as the packet type (e.g. DATA or CONTROL), sender address (which may or may not be the same as the IP source address), etc. When a dup() function in instantiated at a node, the application specifies the classifier to use (i.e. which packets to duplicate). The classifier may use the application-specific parts of the multicast group ID to selectively duplicate/forward packets.

The dup() function snoops passing IP packets. If the packet matches the classifier filter, the dup() function replicates the IP payload, builds a new IP header with current node as the IP source and the desired receiver (or next hop) as the IP destination, and then passes the packet to the standard unicast routing and forwarding mechanism.

B. Echo Processing

As we will see in later sections, it is also useful to assume that network nodes can act as a "transponder" for specially marked ESP probe messages. We call these *echo ESPs* because the network node will "echo" the ESP probe back to the source of the probe. That is, an end-system can stimulate a node in the network to echo the stimulus message back to the initiating end-system. Basic echo processing in very simple; the source and destination need to be exchanged and the echo field disabled. The echoing network node does not even need to inspect the contents of the stimulus message. Note that echo ESPs only invoke hop-by-hop processing in the reverse direction, not in the forward direction.

Echo ESPs are particularly useful when an end-system, A, wants to learn information about the path from a network router, R, to A. Because network paths are often assymetric, sending an ordinary ESP message from A to R may not follow the same path as packets flowing from R to A. Echo ESPs allow end systems to discover information about the reverse path.

There are several ways in which Echo ESPs can be implemented with minimal overhead. For example, they might be handled similar to ICMP echo packets[14]. Alternatively they could be tunneled from A to R and then forwarded normally from R to A. The particular implementation method used is not important to our discussion.

IV. Sender-managed Groups

In this section, we present a centralized multicast implementation where the multicast tree information is controlled by one host in the group. Often the sender in single source multicast applications needs to know all the group members for reasons such as reliability, security, limiting access, billing and so on. Therefore, the sender is naturally the controlling host. Although such applications have limited scalability (i.e., ability to deal with very large groups), the application-specific requirements dictate a multicast service that informs the sender of all join/leave requests.

In a sender-managed group, all group membership information is known to the sender, which tracks the receiver tree topology and handles tree construction and maintenance. The basic algorithm works as follows. All join and leave requests are sent directly to the sender. Each time the sender receives a join request, it must identify the best "graft point" and updates the duplication functions appropriately. Each time the sender

[2] Another alternative is to place the multicast group ID the non-fragmentable part of the IPv6 payload.

receives a leave request (or times out a receiver), the sender must decide whether the branch point closest to the receiver can be removed or not. If possible, the sender removes the duplication function from the branch point and adjusts the parent and/or child branch points surrounding the removed branch point.

Multicast packets are transmitted hop-by-hop: every branch point has a duplication function for each of its "child" receivers and branch points. Each multicast packet carries the (unicast) address of either a branch point or a receiver. The sender initially transmits the packet to each first-hop branch point. When a packet arrives at a branch point, it is duplicated and forwarded to the destination configured for each installed duplication function.

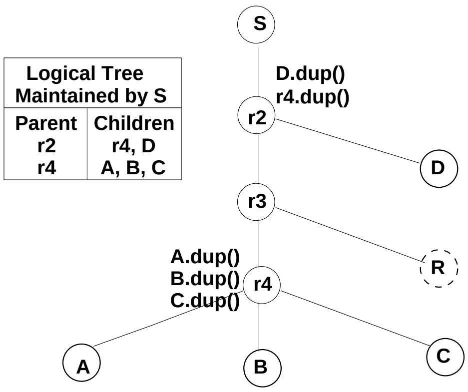

Fig. 3. Sender managed tree

The key to the algorithm is finding the best location in the existing multicast tree to graft on a new branch, without requiring the sender to know the entire network topology. ESP messages can provide the functionality needed to identify the desired branch point without exploring the whole topology. Assume the sender maintains a model of the logical topology that describes the relative location of each receiver and existing branch point, such as the one shown in Figure 3. A variety of ESP-based algorithms for identifying branch points are possible; the following presents one such algorithm.

When a new receiver R sends an application-level join request (not an ESP message) to the sender S, S initiates the following *Branch Point (BP) identification* procedure:

1. S sends an ephemeral state probe to R to determine $BP_{closest}$, the closest existing BP to R.[3] This state probe simply collects the ID of the most recent node that contains a duplication function for this multicast address.

2. R sends an application-level message to S containing $BP_{closest}$.

3. S then sends a "marker" probe to each child (BP or receiver) directly downstream from $BP_{closest}$. Each probe contains the destination address $pkt.dstID$ and performs the following computation:

$(*pkt.val) := pkt.dstID;$

The computation marks the path to each downstream branch point with the ID of the branch point. This will be used in the next step to identify the branch point that is downstream of the new branch point.

4. After a short time period, S sends another ephemeral state probe to R to find the node on the current tree closest to R, which will be the new BP. This probe initiates the following computation at each node:

$$
\begin{aligned}
&\text{if } ((*pkt.val) \neq \perp) \\
&\quad pkt.BP_{next} := (*pkt.val); \\
&\quad pkt.BP_{best} := this\_node\_addr; \\
&\}
\end{aligned}
$$

5. When R receives pkt.BP_{best} it sends a message to S containing pkt.BP_{best} and BP_{next}.

6. S initializes the new branch point, BP_{best}, making it a child of $BP_{closest}$, and BP_{next} the child of BP_{best}, via the steps below.

Initializing a new branch at BP_{best} involves the following steps:

• Modify the logical tree maintained at the sender to include BP_{best} and R.

• Insert a duplication function at BP_{best} that captures, duplicates, and forwards multicast messages to the receiver R and possibly also to an existing BP downstream in the tree (BP_{next}).

• Insert a new duplication function at $BP_{closest}$ that sends to BP_{best}, and remove the duplication function at $BP_{closest}$ that was sending to BP_{next}.

In Fig. 3, the first ESP message from S to R determines the $pkt.BP_{\text{closest}}$ is $r2$. S knows $r2$ has children $r4$ and D, therefore it unicasts "marker" probes to both $r4$ and D, as illustrated in Fig. 4a. After a short while, the sender S unicast a "collect" ESP to R. When the collect ESP arrives at R, $pkt.BP_{best} = r3$ and $pkt.BP_{next} = r4$ (Fig. 4b). When this information is forwarded to S, S replaces $r4.dup()$ with $r3.dup()$ at $r2$, and instantiates $r4.dup()$ and $R.dup()$ at $r3$. The resulting tree structure is shown in Fig. 5.

Leaving the multicast tree is straightforward. The multicast receiver (R) sends an application-level leave request to the sender (S). S removes R from its logical tree structure, and deletes the $R.dup()$ function at R's BP. If removing R results in a BP without any children, the sender recursively removes that BP from the tree.

The multicast sender maintains connectivity by periodically multicasting a control message down the tree to refresh all dup() functions on the BP nodes. Multicast receivers can be tell they are connected to the tree by receiving the refresh message periodically. In addition, the sender should detect receiver failures in order to adjust tree structure accordingly. This can be achieved by having receivers periodically send "heartbeat" messages to the multicast sender. Since sender-managed groups are intended for limited-scalability applications, this requirement does not lead to "implosion" at the sender. An optimization measure can be taken to eliminate

[3] Note that the closest BP may be S.

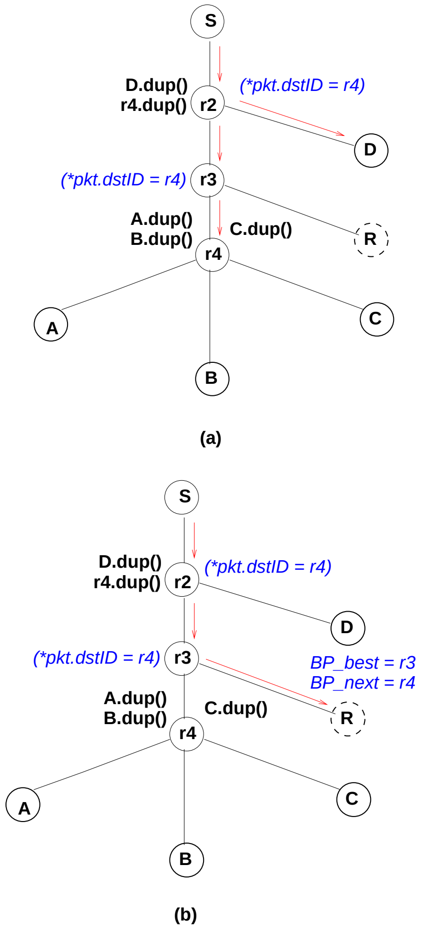

Fig. 4. (a) setup ESP sent in multicast packet. (b) collect ESP unicast to R

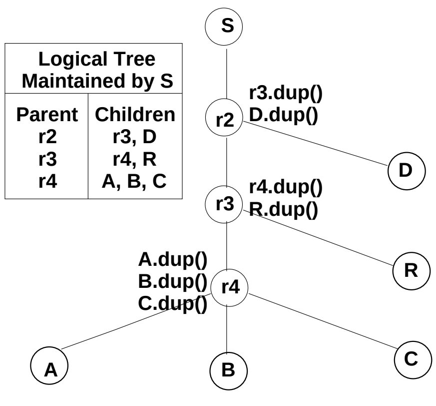

Fig. 5. Sender managed tree (after R joins the tree)

some "heartbeat" messages by adding functionality to aggregate the "heartbeats" in the network. The implementation of this optimization is described in [18].

V. Decentralized Multicast Implementation

In this section we present a scalable, receiver-oriented algorithm for constructing multicast trees. It differs from the previous algorithm in that the sender is unaware of who the receivers are or even how many receivers are currently in the group (much like the standard IP multicast abstraction). Moreover, the receivers (and not the sender) are responsible for most of the multicast tree management.

Like the sender-managed algorithm, multicast is implemented by placing dup() modules at strategic routers in the network. However, unlike the sender-oriented algorithm in which the dup() functions forwarded packets to the next hop in the multicast tree (i.e. not necessarily to a receiver), each dup() function in our receiver-oriented approach forwards packets to a specific receiver. In other words, the IP destination address of every packet belonging to the flow will be that of a receiver in the group rather than the next hop in the multicast tree. Each receiver in the group installs its own dup() function somewhere on the path from the source to the receiver.

We describe the algorithm for "single source" multicast groups. However, with straightforward modifications, the algorithm can be extended to handle multiple sources via a single-source rendezvous point similar to PIM-SM [6].

A. Branch Point Discovery

Our receiver-oriented multicast tree construction algorithm has two parts: *Existing Branch Point Discovery* (described below) followed periodically by a *Tree Optimization* operation (Section V-B).

When a receiver wants to join the group, it begins by locating the nearest point on the existing multicast tree where branching already occurs (i.e., dup() functions have been installed), which may be S. Having identified the nearest existing branch point, the new receiver will install its own dup() function at that branch point. This may not be the optimal branch point for the new receiver, but since packet delivery is unicast from the branch point, data will correctly reach the receiver. For example, consider the tree shown in Figure 6a. When receiver C joins, it finds existing branch point $r2$ and installs its dup() function at $r2$ (see Figure 6b). Clearly $r3$ is a better branch point which will be discovered in the Tree Optimization phase.

A receiver locates the nearest existing branch point (BP_{closest}) via an ephemeral state probe sent (echoed) from the multicast sender back to the receiver. Specifically, the ESP message carries two values, $pkt.BP_{closest}$ and *pkt.hopcount*; the hopcount is used later in an optimization. The echo ESP computation is described as follows:

if (a dup() exists for this group) {
  $pkt.BP_{closest}$= this_node_addr;

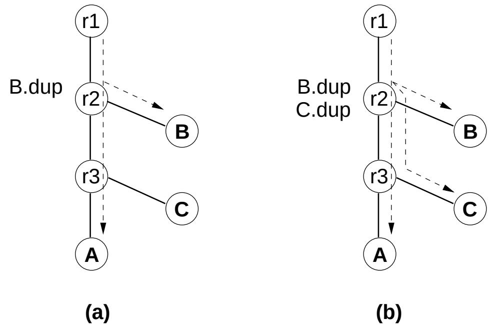

Fig. 6. Existing Branch Point Discovery.

Counting Probe Computation

if $((*pkt.count) \neq \perp)$
  $(*pkt.count) := (*pkt.count) + 1;$
else
  $(*pkt.count) := 1;$

Collection Probe Computation

if $(pkt.newBP \neq NULL)$
  $pkt.hopcount := pkt.hopcount + 1;$
if $((*pkt.count) > 1)$ {
  $pkt.newBP$:= this_node_addr;
  $pkt.max\_redundant :=$
    max($pkt.max\_redundant$, $(*pkt.count)$);
  $pkt.hopcount := 0;$
}

Fig. 7. Counting and Collection probes initated periodically by the source.

    pkt.hopcount= 0;
  }
  pkt.hopcount+= 1;

Upon receiving the echo ESP, the new receiver knows the nearest existing branch point and the distance (number of hops) it is away from that point. It then issues a control message to instantiate a dup() function at the branch point that will duplicate and then forward the duplicate to the new receiver. Once the dup() function has been installed, it must be "refreshed" periodically by the receiver.

B. Tree Optimization

The goal of the tree optimization algorithm is to improve efficiency of the distribution tree by identifying and modifying links that carry redundant traffic.

B.1 Finding Better Branch Points

The first step is to identify better branch points. To help receivers identify better branch points and reassure them they are still in the multicast tree, the sender periodically multicasts (along the existing multicast tree) a pair of ephemeral state probes, namely a *counting ESP*, followed by a *collection ESP*. The computation associated with each probe is described in Fig. 7. Initially the *pkt.newBP* field in the probe message is set to NULL and *pkt.max_redundant* is 0.

The purpose of the counting ESP is to record the number of duplicated messages going through each router. For example, the number of duplicate message passing through router $r3$ in Figure 6b is 2. The multicast collection ESP message uses the information left by the counting ESP to identify potentially better branch points for each receiver. If the *pkt.hopcount* of the discovered branch point *pkt.newBP* is smaller than the hop count to the current branch point, the discovered branch point *pkt.newBP* is potentially better than the current branch point and becomes the *target branch point* where we would like to relocate the dup() function. All multicast group members that find a potentially better branch point are called *reorganizing receivers* and will participate in the next phase of the algorithm; deciding which receivers should move their dup() functions to their target branch points, and which should not. The value *pkt.max_redundant* is included as an optimization that will be described in section V-E.

B.2 To Move or Not to Move

The next step is for the *reoganizing receivers* to decide which of them will move to their target branch points and which of them will remain at the existing branch point.

Before describing which receivers should move their dup() functions, we need to consider the different scenarios under which a link might be traversed multiple times. There are two basic scenarios that cause non-optimal tree paths (see Figures 8 and 9). In the first scenario (Figure 8), links that experience duplicate traffic are the result of two or more dup() functions residing on the same node. In the second scenario, (Figure 9), links that experience duplicate traffic are the result of dup() functions located at different nodes. The first scenario can be solved by having either node A or B (but not both) move its dup() function to $r2$. In other words, we need to "elect" one of the two to move. The second scenario, shown in Figure 9, can only be resolved if the dup() function that is farther downstream, namely C's dup() function as opposed to A's, moves to the target node 4. In other words, the receiver whose dup function is closest to the multicast source should not move, while other receivers that have the same target node are free to move.

In order for the *reorganizing receivers* to decide whether to move or not, they need to know the potential moves of other receivers in the reorganizing set. To inform other members in a scalable way, each reorganizing receiver announces its intention to move by sending an echo ESP to the node where its dup() function currently resides. The probe message contains the receiver's unicast address in the IP address field, but contains the group multicast address in the multicast header. As a result, the ESP echo'ed by the router will be sent to all receivers in the multicast subtree.

This "subcast" echo ESP serves as a move announcement and contains three values: *src*, the originator of the move mes-

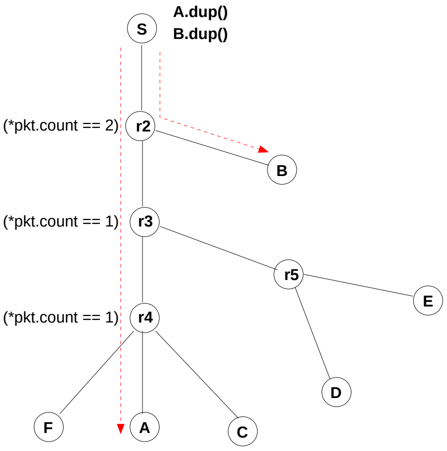

Fig. 8. Non-optimal tree configuration example 1.

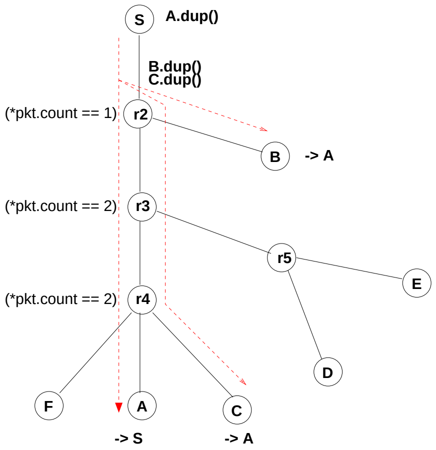

Fig. 9. Non-optimal tree configuration example 2.

sage, *current*, the current location of the *src's* dup() function, and *target*, the node where *src* intends to move its dup() function.

Given the two possible scenarios shown in Figures 8 and 9, a receiver R will move its dup() function to its target node if either of the following conditions is true:

- R learns that it has the same *current* and *target* nodes as another receiver (e.g., Figure 8) whose ID (e.g., IP address) is *larger* than R's ID, OR
- R learns that it shares a link with a node whose dup() function is closer to the multicast source than R's dup() function (e.g., Figure 9). Any move announcement containing value of *current* that differs from R's own value of *current* is from a node whose dup() is upstream of R's dup(), so R can move its dup function.

In Figure 9, node C will receive A's move announcement, but A will not receive C's move announcement because C's echo ESP move announcement is sent to $r2$ rather than S. Receivers that do not receive a move announcement (e.g., node A in Figure 9) do not take any action.

In both cases, the reorganizing receiver that moves learns the identity of the receiver whose packets it will be "snooping" (dup()ing from). We say that C *depends on* A if C's dup() function "snoops" packets sent to A, i.e. if the filter with which it was installed includes "destination=A". This information is recorded and used when a receiver decides to leave the tree.

C. An Example

Consider the topology in Figure 8 where the receivers join in the order $A, B, ..., F$ and assume that the node ID of $A > B > C > D > E > F$.

When node A joins, it finds the sender (node S) as the closest existing branch point and installs its dup() function there. A only depends on the sender (denoted $\rightarrow S$). When B joins, it too finds S as the nearest existing branch point and installs its dup() function at S, recording its dependency on A.

Assume the source issues a periodic counting ESP followed by a collection ESP to optimize the tree. The counting ESP leaves the values shown in Figure 8, which are examined by the collection ESP, ultimately informing A and B that $r2$ is their target node. Node A and B then issue move announcements and discover one another. B then moves its dup() function to $r2$ because they have the same *src* and *target* and $B < A$.

When C joins, it finds that $r2$ is the nearest existing branch point and installs its dup() function at node $r2$. The next round of tree optimization messages leaves the ephemeral state shown in Figure 9 and informs A and C that $r4$ is their *target*. A then stimulates a move announcement from S and C stimulates a move announcement from node $r2$. C receives (*src*=A, *current*=S, *target*=$r4$) and (*src*=C, *current*=2, *target*=$r4$) while A only receives its own message (*src*=A, *current*=S, *target*=$r4$). Consequently C moves its dup() function to node $r4$ and records its dependency on A.

Similarly, when D joins, it first installs its dup() function at $r2$ but then moves it to $r3$ during the next round of tree optimization. When E joins, it joins at $r3$ and then moves to $r5$ during the next round of tree optimizations because E and D have the same *src* and $target$ and $E < D$. Finally, F joins and immediately finds node $r4$. Future tree optimization messages do not identify any better branch points. Note that F will not know who it depends on. The final tree appears in Figure 10.

D. Member leaving or failure

A receiver can leave the multicast tree in one of two ways: (1) by explicitly removing its dup() function, or (2) letting its dup() function time out (i.e., not be refreshed). To leave gracefully, a receiver stimulates an echo control message from its dup() node indicating its desire to leave the group; this message is multicast to the subtree rooted at the dup() node. The "leave" message contains *current*, the location of the dup() function to be removed, *owner*, the node ID the dup() function belongs to, and *depends*, the ID of the node the dup() function

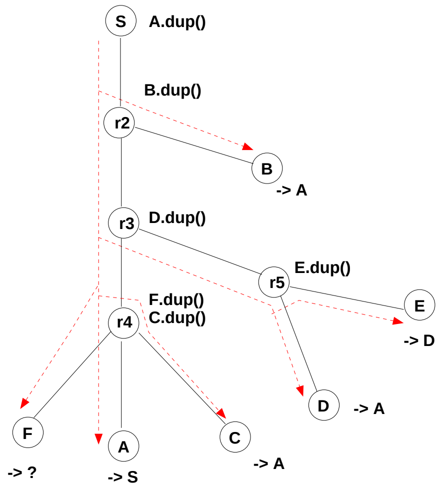

Fig. 10. The final multicast tree.

depends on.

The "leave" message informs all other receivers in the subtree that they may need to reposition their dup() functions in response to the recently departed receiver. Only receivers that depended in the departed receiver need to react. Receivers that do not know who they depend on will also adjust their dup() location. The actions taken by the dependent receivers is simple: (1) move their dup() function to the node previously occupied by the departing receiver, (2) wait for the tree optimization ESPs to reorganize the tree. Figure 11 illustrates the reorganization steps that occur when node A (from Figure 9) leaves the group.

E. Scalability Optimization

In the previous section, we presented the tree construction and optimization process that produces an optimal tree structure. Based on the algorithm characteristics, several optimizations can be applied to reduce the amount of control traffic, namely, the move announcement traffic, making the tree construction process more scalable.

Although the move announcements are subcast and thus have limited scope, the number of move announcement messages can be large and far reaching if many receivers join at the same time (possibly as the result of a receiver leaving). However, there are various optimizations that can significantly reduce the amount of "move" control traffic:

1. *Announcement Suppression*
One way to reduce the control traffic overhead is to have receivers suppress their move announcement if they received a move announcement from a node with the same *target* value during the same round. The receiver should suppress its announcement and immediately move its dup() function to *target* node since it knows there will be a receiver who will not move its dup() function either because the receiver does not hear any other move announcements, or because the move decision algorithm indicates it is the one that should not move.

2. *Rate Reduction*
In case a receiver's *target* dup() location is upstream from where the other reorganizing receivers are trying to move their dup() function, it is not necessary for the receiver to keep sending move announcement in each round, since a competing mover may not yet exist. If a receiver's previous move announcement did not result in a move decision, and in the following round the same *target* node is found, it is experiencing a *wait* condition. If there is no loss in the network, the receiver under the *wait* condition can in fact stop sending move announcements, and a "default" decision will be made as described in 1 above when a competing announcement eventually arrives. Under realistic conditions where loss may occur, the receiver may choose to reduce the sending rate of its move announcement to one every 2 or 3 rounds. Slowing down the sending rate will reduce control traffic, but not stop the tree optimization progress if move message loss occurs.

3. *Volume Limitation*
Another way to reduce the number of move announcements is to have receivers systematically suppress their move announcements so that no more than K receivers send move announcements during each round. Because the tree optimization ESP messages contain *max_redundant*, the maximum number of redundancy on the path, each receiver knows the potential number of move announcements it is going to receive. If the desired maximum number of move announcement per round on the tree is K, and *max_redundant*$> K$, each receiver can send with probability $K/$*max_redundant*. This will result in a limit of approximately K move messages in the tree per round.

VI. Simulation Experiments

We ran simulations to evaluate the effectiveness and scalability of the receiver-oriented tree construction algorithm. Our simulations only considered members joining the group, not leaving. However, leave events causes the system to behave as if many receivers joined simultaneously, so evaluating the join overhead gives insight into the performance of leaving as well.

To evaluate the performance of our algorithm, we used three metrics:

Average Link Stress: Link stress is defined as the number of identical copies of a packet carried over a physical link [4]. The average link stress is measured by summing the stress level of all links in the multicast tree and dividing by the total number of links in the tree.

Maximum Link Stress: The largest link stress value on the tree.

Average Control Cost: Control cost is the number of move messages seen by receivers during each round of tree optimization messages. The average control cost is calculated by summing the total number of move messages each receiver sees per round, then dividing it by the total number of receivers in the group.

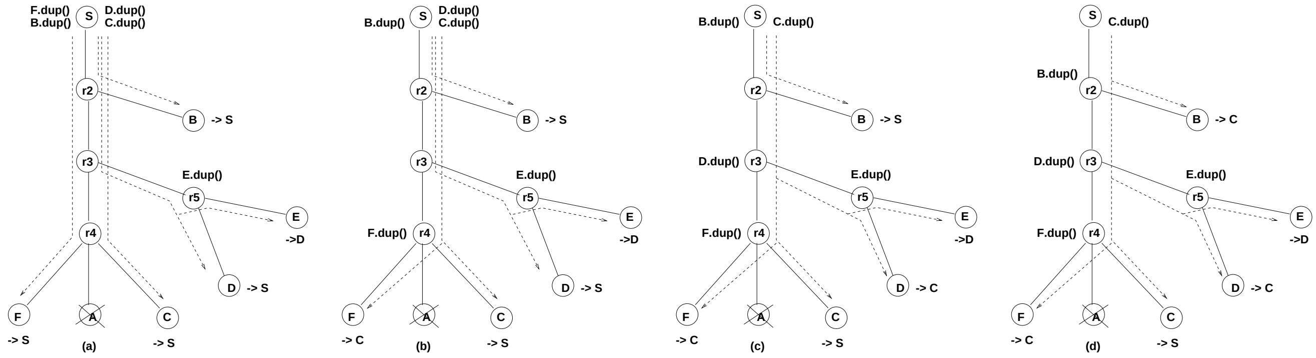

Fig. 11. Steps to reorganize the tree after A leaves.

We simulated a transit-stub topology generated by the GT-ITM topology generator [19]. The graph consisted of 600 nodes consisting of 72 stub domains connected via 3 transit domains. Each transit domain consisted of approximately 8 nodes, each connected to 3 stub domains, which averaged 8 nodes each. The receiver set consisted of 300 nodes randomly selected from the stub domains. The sender was also randomly selected from a stub domain. For the purposes of analyzing the convergence rate of our algorithm, all links in the graph were lossless.

To test the scalability of the system, we varied the rate at which the 300 receivers joined the tree. The fastest join rate has all 300 receivers joining simultaneously at the beginning of the simulation. We also simulated slower join rates with 10 to 290 receivers joining during each tree optimization round. When few members join per round, it takes longer for all receivers to join the group. However, the link stress imposed on the system is less than when all members join at once.

Fig. 12 shows the stress level and convergence time for join rates of 10, 20, 50, 100, and 200 receivers joining per tree optimization round. As expected, the faster join rates imply greater initial stress levels. For all join rates, the average stress level fell below 1.5 after the 4th tree optimization round. In other words, the tree becomes near optimal quickly, regardless of the number of receivers that join, and then remains near optimal. When the join rate is 10, it takes 30 rounds for all receivers to join, and only one more round for the tree to reach optimal configuration. In addition, from the 3rd round onward the tree is always in a near optimal state Therefore, in terms of putting minimum stress on the links at any given time, the algorithm works best when fewer receivers join during each round.

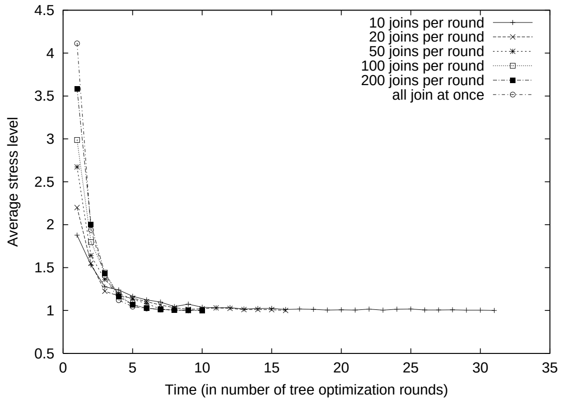

Fig. 12. Average stress level during each round of optimization.

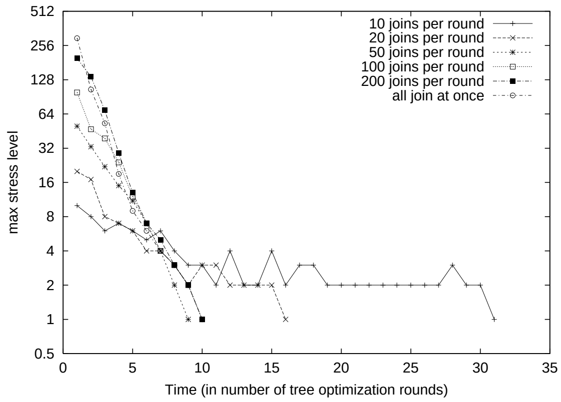

Fig. 13. The maximum stress level for various join rates (in log scale).

Fig. 13 shows the result for the same experiment with the *maximum stress level* (in log scale) as opposed to the average stress level. The increased stress at higher join rates is much more obvious when looking at the maximum stress values. Even so, like the behavior of average stress level, the maximum stress level decreases rapidly (exponentially) with time.

Fig. 14 shows the average control cost (number of move messages each round) for the same experiment, again plotted in log scale. The overhead costs rise quickly as the join rate increases.

The results shown so far did not incorporate any of the overhead reduction techniques described in section V-E. Without any optimization, the overhead cost increases linearly with the join rate. To control the overhead at high join rates, we implemented the optimization in which an upper limit is imposed

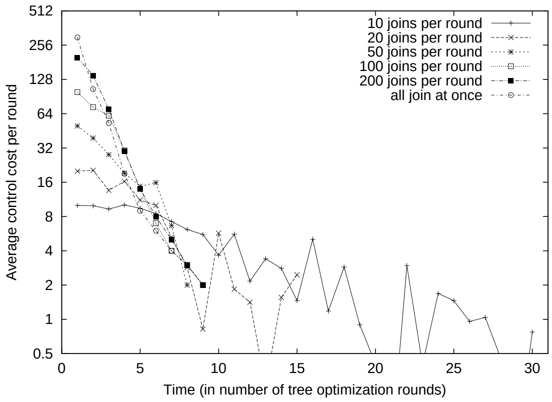

Fig. 14. The average control costs (# move msgs) for various join rates (in log scale).

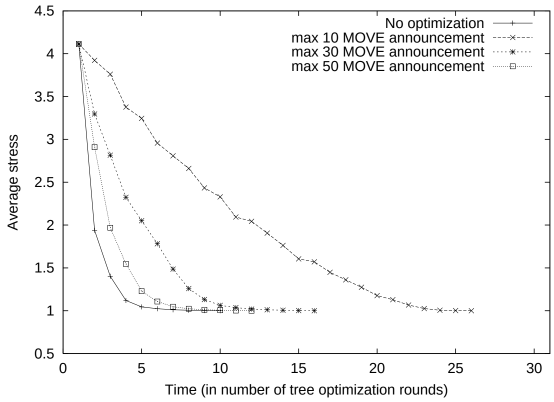

Fig. 16. The average stress level for a join rate of 300 but limiting the number of receivers that can move.

on the number of move messages that can be sent (see *volume limitation* in Section V-E). We tested the effect of this optimization using the worst-case join rate of 300 (all nodes joined at once). Fig. 15 shows the average control cost using the optimization, with 10, 30, and 50 receivers trying to move during any given tree optimization interval. The graph shows that the optimization dramatically reduces control traffic without significantly prolonging the tree convergence time; the tree has converged when the cost reaches zero.

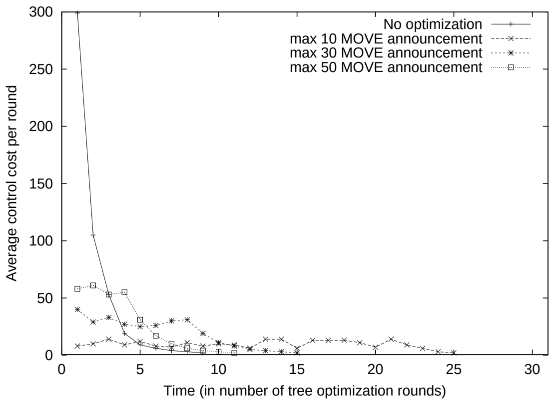

Fig. 15. The average control costs at a join rate of 300 but limits on the number of receivers that can move.

The corresponding change in stress level created by the optimization is illustrated in Fig. 16. Clearly, there is a tradeoff between control overhead and stress level. The optimization reduces control overhead but does so at the expense of slower stress level reduction rate (and the convergence rate). Because move messages are small, they typically consume less bandwidth than the redundant data packets caused by high stress levels. However, implosion can still be a problem with high control overhead, and may be the limiting factor that determines the appropriate tradeoff.

VII. Related Work

Perhaps the most closely related work is the *REUNITE* [16] protocol developed at CMU. REUNITE uses unicast addresses and forwarding to construct single-source distribution trees similar to our receiver-oriented protocol. Unlike our application-level approach, REUNITE is a network-level multicast implementation hardwired into routers, i.e., the tree construction algorithm is embedded in the network. Multicast data is sent in a unicast packet with a multicast group ID consisting of a (unicast IP address, port number) pair. Every router along the path to an existing receiver must maintain either a *multicast forwarding table* (if it is a branch point) or a *multicast control table* (if it is a potential branch point). Routers along the existing delivery tree intercept JOIN messages from new receiver and add the receiver's unicast address to their table for packet duplication. Consequently, *every* router on the distribution tree needs to maintain multicast state information for the group. Our proposed algorithm only places multicast state information (i.e. dup() functions) at the branch points of the distribution tree. Other routers along the path simply forward unicast packets as normal. Moreover, in REUNITE, the multicast path from the sender to a receiver may not be the shortest path. In our approach the multicast delivery tree is on the forward shortest path from the sender to each receiver.

Endsystem Multicast [4] is an application-level approach that attempts to construct an overlay network by establishing logical links between closely connected neighbors using a tree construction protocol called NARADA. A new group member must first learn about the others in the group through an out-of-band bootstrap mechanism. Each host forms logical connections to at most n of its neighbors. Neighbors are selected with the constraint that the delay between any two group member must be at most K times the unicast delay between them. Each host periodically exchanges group membership and routing information with its neighbors to create a mesh topology of all group members. After the mesh is established, the multicast data is routed using the DVMRP routing algorithms at the application level.

To maintain the best mesh quality in face of dynamic group membership change, each host also probes random group members periodically to get their routing tables. Because all group members need to know all other group members in order to form the mesh, the algorithm does not scale well to large group sizes.

Another application-level multicast architecture called *Yoid* [10] focuses on providing large-scale, albeit non-optimal, multicast connectivity. Yoid assumes some number of rendezvous hosts in the system where new members can go to discover other group members. Each rendezvous host only knows nearby receivers. New group members learn about other group members from a nearby rendezvous host and choose one to be their parent in the multicast tree. The Yoid protocol scales to large groups because each group member only needs to know a small section of the total group membership. However, because group members have very limited topology information, there is no guarantee that the resulting tree is optimized. It simply guarantees all group members are connected.

Overcast from Cisco [12] builds a single source tree for reliable content distribution. Overcast contains an overlay topology constructed from special purpose network nodes (servers). The special purpose nodes are placed at strategic (fixed and well-known) locations of the network. A multicast sender delivers data to the servers and redirects requests for content to the closest server. The multicast tree that connects these servers cannot be dynamically configured.

VIII. Conclusions

We presented a simple, programmable network framework based on ephemeral state processing, lightweight processing modules, and standard unicast routing and forwarding. The ephemeral state mechanism provides a time-bounded associated memory store at network nodes that, when combined with small fixed-length straight-line programs, allows end-systems to compute specific information about the network and its topology. Combining knowledge about the network with the ability to inject lightweight processing modules into specific nodes creates the opportunity for users to implement new application-specific network services from the edge of the network.

To demonstrate the utility and flexibility of the framework, we described two different ways to build multicast services at the application level. The first approach illustrated how applications can construct a sender-controlled multicast distribution tree that employs hop-by-hop processing to deliver data to receivers. The sender creates and maintains the multicast distribution tree by sending ESP messages to determine topology information and then install lightweight duplication functions (dup() functions) at strategic nodes in the network. The second example showed how receiver-controlled multicast trees could be built in a scalable fashion. In this case, the receivers conspire to create the multicast tree by installing dup() functions that "snoop" packets destined for other members of the group. As opposed to some multicast routing protocols that build reverse-path forwarding trees, both our algorithms transmit data along the shortest forward path from the sender to the receiver. Moreover, the only network routers that maintain any multicast state are the routers at the branch points. All other routers along the multicast distribution tree are completely unaware of the multicast flow, unlike conventional IP multicast protocols.

References

[1] Nidhi Bhaskar. Source-Specific Protocol Independent Multicast. *Internet Draft: draft-bhaskar-pim-ss-00.txt*, March 2000.

[2] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active Networks. *IEEE Communications Magazine*, 36(10):72–78, October 1998.

[3] K. Calvert, J. Griffioen, and A. Garg. Computing in the Network with Ephemeral State. Technical Report 307-00, Department of Computer Science, University of Kentucky, 2000.

[4] Yanghua Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *the Proceedings of ACM Sigmetrics*, pages 1–12, June 2000.

[5] D. Clark. The Design Philosophy of the DARPA Internet Protocols, 1988.

[6] S. Deering, D. Estrin, D. Farinacci, and V. Jacobson. An Architecture for Wide-Area Multicast Routing. In *the Proceedings of the SIGCOMM'94 Conference*, September 1994.

[7] Stephen Deering, Deborah Estrin, Dion Farinacci, Van Jacobson, Ahmed Helmy, David Meyer, and Liming Wei. Potocol Iependent Mlticast Version 2 Dense Mode Specification. *Internet Draft: draft-ietf-pim-v2-dm-01.txt*, November 1998.

[8] Stephen E. Deering. Host Extensions for IP Multicasting, August 1989. RFC 1112.

[9] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode: Protocol Sepcification. *RFC 2364*, June 1998.

[10] Paul Francis. Yoid: Extending the Internet Multicast Architecture. *http://www.aciri.org/yoid/docs/index.html*, April 2000.

[11] Hugh W. Holbrook and David R. Cheriton. IP Multicast Channels: EXPRESS Support for Large-Scale Single Source Applications. In *Proceedings of SIGCOMM'99*, 1999.

[12] John Jannotii, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of OSDI*, 2000.

[13] R. Perlman. Simple Multicast: A Design for Simple, Low-Overhead Multicast. *Internet Draft: draft-perlman-simple-multicast-03.txt*, October 1999.

[14] J. Postel. Internet Control Message Protocol, September 1981. RFC 792.

[15] T. Pusateri. Distance Vector Multicast Routing Protocol. *Internet Draft: draft-ietf-idmr-dvmrp-v3-08.txt*, Feburary 1999.

[16] Ion Stoica, T.S. Eugene Ng, and Hui Zhang. REUNITE: A Recursive Unicast Approach to Multicast. In *Proceedings of INFOCOM 2000*, 2000.

[17] D. Thaler, D. Estrin, and D. Meyer. Border Gateway Multicast Protocol (BGMP): Protocol Specification, November 2000. Internet Draft: draft-ietf-bgmp-spec-02.txt.

[18] Su Wen. Building Efficient Group Communication Services on a Programmable Network Framework. Technical Report 311-01, Department of Computer Science, University of Kentucky, January 2001.

[19] E. Zegura and K. Calvert. Georgia Tech Internet Topology Models. http://www.cc.gatech.edu/projects/gtitm.