End-host Controlled Multicast Routing

Karthik Lakshminarayanan[*] [†] Ananth Rao[*] Ion Stoica[*] Scott Shenker[‡]

University of California, Berkeley

Abstract

The last decade has seen a deluge of proposals for supporting multicast in the Internet. These proposals can be categorized as either infrastructure-based, with the multicast functionality provided by specialized network nodes, or host-based, with the multicast functionality provided by the members of the multicast group itself. In this paper, we present the design and evaluation of a hybrid multicast architecture wherein the infrastructure provides packet forwarding, and the end-hosts implement the control plane. End-hosts build multicast trees by setting up forwarding state in the infrastructure. This division of functionality enables our architecture to combine the efficiency of infrastructure-based solutions and the flexibility and deployability of host-based solutions. We present scalable and efficient algorithms for distributed tree construction and maintenance, and for reliable packet delivery. We have implemented the algorithms using $i3$ as the forwarding infrastructure. We evaluate our techniques using a combination of event-driven packet-level simulations, and our implementation over the PlanetLab testbed.

Keywords: multicast, routing, architecture, end-host control, overlay

[*]EECS Department, UC Berkeley {karthik, ananthar, istoica}@cs.berkeley.edu

[†]Corresponding author. 465 Soda Hall, CS Division, University of California, Berkeley, CA 94720. Phone: +1 510 642 8905. Email: karthik@cs.berkeley.edu

[‡]UC Berkeley and ICSI, shenker@icsi.berkeley.edu

1 Introduction

The original Internet architecture was designed to provide best-effort *unicast* point-to-point communication. This simple abstraction was one of the main reasons behind the success of the Internet as it allowed a scalable and robust implementation. However, efficiently reaching many users simultaneously with high-bandwidth streams (as required for popular video and audio streams) requires *multicast* functionality. The last decade has seen numerous multicast proposals, both in academia and industry, none of which are in widespread use today.[1] These proposed multicast designs can be roughly classified into infrastructure-based and host-based solutions.

Infrastructure-based solutions implement multicast functionality in a set of designated network nodes that are responsible for both constructing the multicast tree and replicating multicast packets at the branch points in that tree. Examples include IP multicast [11, 12] which implements the multicast functionality at the IP layer (and so the specialized multicast nodes are network routers themselves), and application-level solutions such as Overcast [22] and Fastforward [2], which use a set of servers as specialized multicast nodes.

In contrast, host-based multicast designs, such as ESM [10] and NICE [5], require no support from any designated network nodes. Instead, the multicast functionality is implemented entirely by the collection of end-hosts participating in the multicast group.

These approaches have different benefits and performance characteristics. The infrastructure-based approaches can build more efficient and scalable multicast trees as their nodes are typically well-connected and placed at well-designated locations in the network. However, end-host based solutions are limited by the upstream bandwidth constraints of the users of P2P systems [27, 39] and the constant churn that is exhibited by such systems. On the other hand, host-based solutions do not involve any additional infrastructure support and are hence easy to deploy and modify—one can add new functionality (such as construct trees adapted to a different metric) easily. Host-based solutions enjoy these benefits not only

[1] IP multicast is supported by several commercial routers, but inter-domain multicast service is not made available to users by most ISPs.

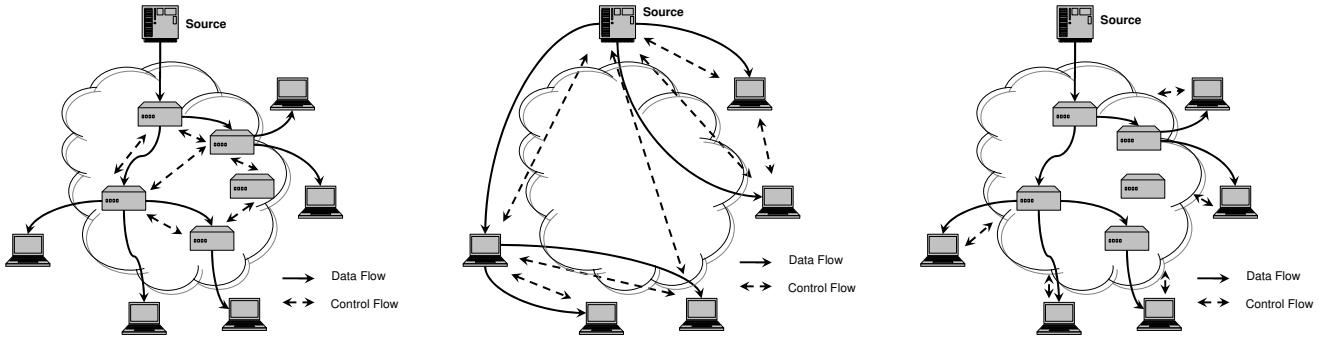


Figure 1: Architectural choices for multicast (left to right): (a) infrastructure implements control plane as well as data forwarding, (b) end-hosts implement both control plane as well as data forwarding, (c) our architecture, where infrastructure nodes implement data forwarding, and end-hosts implement the control plane.

when compared to IP multicast but also when compared to infrastructure-based solutions that implement multicast functionality at the application layer. Since such infrastructures are typically managed by third party entities (*e.g.,* Akamai, RealNetworks), it is difficult for end-hosts to change or adapt the multicast protocol to best meet their needs.

We address the question of whether one can design an architecture which would realize the best of both approaches: the efficiency of infrastructure approach, and the flexibility (the ability to change the tree construction algorithms as per user needs) of the end-host approach. To this end, we propose a hybrid multicast architecture wherein the nodes in the infrastructure export simple primitives, mainly packet forwarding and replication, and the end-hosts use these primitives to construct multicast trees in a distributed fashion. Figure 1 compares our approach with the infrastructure- and host-based alternatives with respect to the entities that implement the control plane protocol and data forwarding. In the rest of the paper, we address the following two issues.

1. *Tree construction and maintenance.* We propose a fully distributed tree construction algorithm, based on a best-first search approach, that can be implemented at the end-hosts using only the primitives that the infrastructure exports. We also present a randomized back-off technique for efficiently maintaining the forwarding state associated with the multicast tree in the infrastructure.

2. *Reliable packet delivery.* We propose a technique that exploits the tree structure and performs local loss recovery efficiently, both in terms of number of messages generated and the recovery time.

Since the control plane is outside the infrastructure in our architecture, a centralized entity could potentially construct and maintain the tree for the entire group as a service. While a centralized approach has the advantages of building more optimal trees, enhanced security and access control, it requires the deployment of a resource-rich centralized service provider. In this paper, we look at the distributed approach only; a description of a centralized routing service can be found in [28].

The remainder of the paper is organized as follows. The next section of the paper introduces the primitives that we consider in the paper, and the following sections address the two aforementioned issues in order. Section 6 presents simulation results that demonstrate the feasibility of our schemes for medium to large scale networks. Section 7 gives implementation details and implementation results. After surveying related work in Section 8, we present a brief critique of some of the design choices that underlie this work in Section 9. We finally conclude the paper in Section 10.

2 Infrastructure Support

The general design philosophy that we advocate in this paper is that the infrastructure should not provide a specific instance of a multicast service, but should only provide support for hosts to build multicast trees tuned to their specific needs—thus, multicast trees based on different metrics can not only co-exist but also evolve over time.

The first issue that we need to address is to design the primitives that the infrastructure should export. It is essential that these primitives, while allowing flexibility, do not compromise the security of the infrastructure. To this end, we leverage Internet Indirection Infrastructure, $i3$ [40], a rendezvous-based infrastructure that supports diverse functionality. We use a subset of $i3$ functionality in this paper: packet forwarding and replication for building the multicast tree, and anycast for packet loss recovery.

While we implement our solutions on top of $i3$, in principle, we can also implement them using a simple label-switching primitive[2] like that used in MPLS. The diverse benefits that $i3$ offers (such as

[2]The anycast primitive used for loss recovery scheme can be implemented by a simple extension to a label-switching protocol. The security extensions to $i3$ directly apply to a label-switching protocol also.

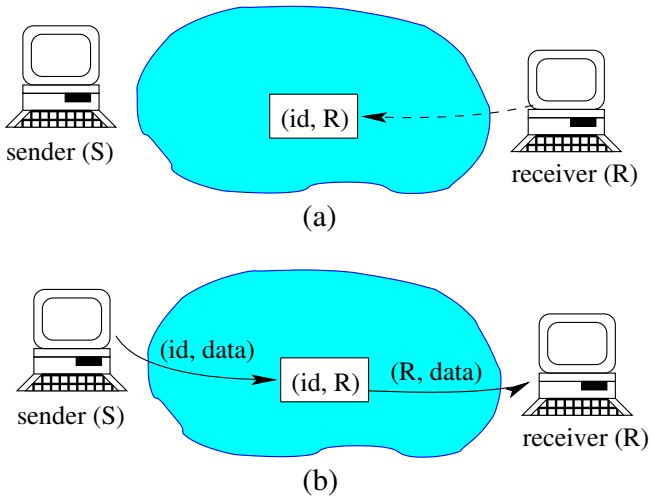


Figure 2: (a) The receiver R inserts trigger (id, R). (b) The sender sends packet $(id, data)$.

mobility, service composition, etc.) make it an attractive option as the multicast algorithms we propose here can be used in conjunction with, say, service composition to yield more useful services. While the $i3$ paper [40] mentions these advantages, it does not explore scalable multicast in detail; in this paper, we restrict the focus to multicast and explore the issues that arise in detail. We describe only the necessary aspects of $i3$ in the remainder of the section; a fuller exposition can be found in [3, 26, 40].

In this paper, we have assumed that the infrastructure is embedded with minimal functionality. Another possibility would be to extend the infrastructure to implement part of the control plane by aiding the end-hosts in measuring various performance metrics or perform error recovery. We discuss the implications briefly in Section 9.

2.1 Overview

For the purpose of this paper, $i3$ can be considered as a overlay label-switched network in which the packets are forwarded over IP paths rather than physical links. In particular, $i3$ [40] implements a rendezvous communication abstraction. Sources send packets to a logical *identifier* (ID) and receivers express interest in packets by inserting a routing entry (also called a *trigger*) into the network corresponding to the ID associated with the packet (Figure 2(a)). Packets are of the form $(id, data)$ and triggers are of the form $(id, addr)$, where $addr$ is either an ID or an IP address. Given a packet $(id, data)$, $i3$ finds the trigger $(id, addr)$ and then forwards $data$ to $addr$. Packets are merely forwarded, not stored, at the infrastructure

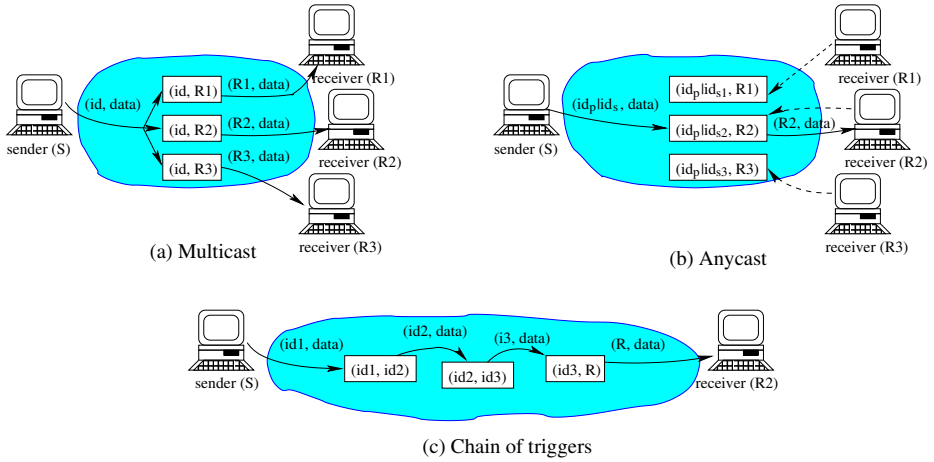


Figure 3: Basic primitives. (a) Packet replication: Every packet $(id, data)$ is forwarded to each receiver R_i that inserts trigger (id, R_i). (b) Anycast: The packet matches the trigger of receiver $R2$. $id_p|id_s$ denotes an ID of size m, where id_p represents the prefix of the k most significant bits, and id_s the suffix of the $m-k$ least significant bits. (c) Trigger chains: Replace the receiver address in the second field of a trigger with another trigger ID.

nodes. Receivers refresh the triggers that they insert as long as they desire to receive packets sent to the ID that the trigger corresponds to.

Identifiers in $i3$ are 256 bits long. IDs in packets are matched with those in the routing entries (or triggers) using longest prefix matching (under the constraint that 192-bits exactly match). $i3$ is implemented as an overlay network of nodes that store triggers and forward packets. Identifiers are mapped to $i3$ nodes using a distributed lookup service such as Chord [41]. However, in practice, caching ensures that in most cases packets from one $i3$ node to another traverse just the IP path.

Figure 2(b) illustrates the communication between two nodes, where receiver R wants to receive packets sent to id. The receiver inserts the trigger (id, R) into the network. When a packet is sent to ID id, the trigger causes it to be forwarded via IP to R.

Next, we give the basic $i3$ communication primitives that we use for multicast.

Packet Forwarding and Replication: If multiple receivers register triggers with the same ID id, packets sent to id would get replicated and a copy sent to each receiver (see Figure 3(a)). As noted in [40], this basic packet replication cannot be used directly for large multicast groups—all triggers of the group would have the same ID and hence be stored at the same $i3$ server, and consequently that server would be responsible for forwarding each packet to all the members of the multicast group. Each $i3$ server places a

limit D on the number of triggers with the same ID it stores. In other words, D represents the maximum number of receivers in a multicast group that are allowed by the basic $i3$ packet replication primitive.

Trigger Chains: $i3$ allows end-hosts to chain multiple triggers by using triggers of the form (id, id'). Figure 3(c) shows an example of a chain of triggers of length three. Replacing a trigger (id_1, R) with a chain of triggers (id_1, id_2), (id_2, id_3), and (id_3, R) is transparent to the end-hosts. Note that routing along a chain in $i3$ is similar to label switching in the case of MPLS.

Anycast: Anycast ensures that a packet is delivered to at most one receiver in a group. $i3$ supports anycast since longest prefix matching is performed only on the last 64 bits of the ID. All hosts in an anycast group maintain triggers which are identical in the $k = 192$ most significant bits. These k bits play the role of the anycast group ID. To send a packet to an anycast group, a sender uses an ID whose k-bit prefix matches the anycast group ID. The packet is then delivered to the member of the group whose trigger ID best matches the packet ID according to the longest prefix matching rule (see Figure 3(b)).

In addition to these primitives, we have also implemented the cryptographic security constructs proposed in [3, 26]. These constructs ensure that adversaries cannot use the flexibility of the route setup mechanisms to launch attacks on the infrastructure or the end-hosts — for instance, one cannot construct cycles in a topology constructed using these primitives.

3 Scalable Tree Construction and Maintenance

In this section, we show how a simple distributed tree construction algorithm can be adapted for building and maintaining a multicast tree using the primitives provided by the infrastructure. Our contribution does not lie in the novelty of the tree join algorithm, but in the techniques we propose to implement the algorithm using only the infrastructure primitives.

We use a two-step methodology for explaining our multicast tree construction protocol. In the first step, we present a pure end-host based multicast algorithm. In the second step, we realize the multicast algorithm using the infrastructure primitives alone. We do this partly for ease of exposition, and partly because the techniques we use in going from the first to the second step are illustrative of what is possible

```
// node n_new joins the multicast tree with sender S
join(S, n_new)
    best_dist = ∞
    n_curr = S
    do
        // return the closest node to n_new from jset(n_curr)
        n = select_node(n_new, jset(n_curr))
        if (best_dist > dist(S, n, n_new))
            best_dist = dist(S, n, n_new)
            n_join = n
        if (fset(n_curr) = ∅)
            break
        n_curr = select_node(n_new, fset(n_curr))
    while (dist(S, n_curr, n_new) < best_dist)
    join_at(n_new, n_join) // n_new joins at n_join
```

Figure 4: Joining algorithm.

with such an architecture. The algorithm we use in the first step can be further optimized based on specific application requirements, possibly leveraging prior research on end-host based approaches.

In our discussion, we consider an algorithm based on the delay metric. In Section 3.2.2, we also discuss how one can build a multicast tree based on other metrics (such as bandwidth) using the same infrastructure primitives. For simplicity, we assume a single source tree; Section 3.2.2 extends this model to multiple sources.

3.1 Basic Tree Building Algorithm

In this section, we present a basic tree building algorithm that abstracts away the interactions between end-hosts and $i3$. The algorithm builds a multicast tree of bounded degree D which exhibits low latency from source to each receiver. Figure 5(a) shows a multicast tree with $D = 3$ consisting of six receivers and one source S, and one new receiver, R_7, that wants to join the multicast group.

We say that a node in the multicast tree is *joinable* if its out-degree is less than D. Otherwise, if the node's out-degree is D, we say that the node is *full*. Let S denote the source of the multicast tree, let n

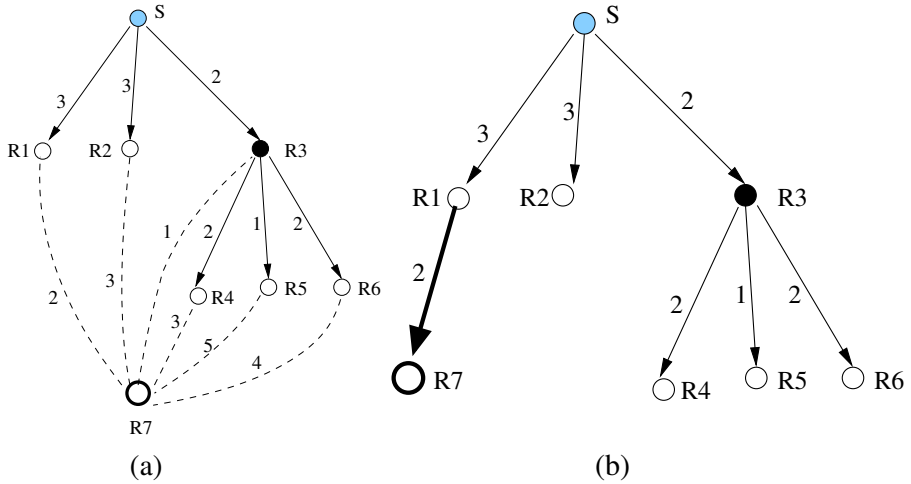


Figure 5: (a) R_7 wants to join an existing multicast tree. Joinable nodes are represented by empty circles; full nodes are represented by black circles. (b) The resulting tree obtained by running the algorithm in Figure 4.

denote a joinable node already in the tree, and let n_{new} denote a new node that wants to join the multicast tree. Assume n_{new} joins the multicast tree at n. Then, let $dist(S, n, n_{new})$ be the latency experienced by a multicast packet from source S to n_{new} via n. In particular, $dist(S, n, n_{new})$ represents the latency from source S to n (via the multicast tree) plus the IP latency from n to n_{new}.

The goal of the joining procedure is to find a joinable node n that provides a short distance path from S to n_{new}. Let $dist(S, n, n_{new})$ denote this distance. To achieve this we use a "branch-and-bound" algorithm that starts from source and goes down the tree until it can no longer improve the distance from S to n_{new} or until it reaches the leaves of the tree. Figure 4 shows the pseudocode of the joining procedure, where $jset(n)$ denote the set of joinable children of n, and $fset(n)$ denote the set of full children of n.

Figure 5 shows a simple example in which a new receiver, R_7, joins an existing multicast tree with $D = 3$ consisting of six receivers. The number along each edge represents the latency associated to that edge. At the first level, there are two joinable nodes $jset(S) = \{R_1, R_2\}$, and one full node $fset(S) = \{R_3\}$. Among the joinable nodes, the algorithm selects R_1 since $dist(S, R_1, R_7) = 5$ is smaller than $dist(S, R_2, R_7) = 6$. Next, the algorithm selects the full node R_3 and iterates. Among the children of R_3, the algorithm selects node R_4, as $dist(S, R_4, R_7) < dist(S, R_5, R_7) = dist(S, R_6, R_7)$. Finally, since $dist(S, R_1, R_7) < dist(S, R_4, R_7)$ the algorithm terminates and R_7 joins at R_1.

```
// join the multicast tree idg
join(idg)
    best_dist = ∞
    id_curr = idg
    do
        // return reply form selected node in jset(id_curr)
        (id, dist) = select_node(jHash(id_curr))
        if (best_dist > dist)
            best_dist = dist
            id_join = id
        (id, dist) = select_node(fHash(id_curr))
        if (id = NULL) break
        id_curr = id
    while (dist < best_dist)
    join_at(id_join) // node joins at id_join

join_at(id_join)
    close_id = pick_close_id()
    insert_trigger(close_id, my_address)
    insert_trigger(id_join, close_id)

select_node(set_id)
    dist = ∞
    p_set = query_distance(set_id)
    foreach (p ∈ p_set)
        d = curr_time − p.time_stamp + p.src_dist
        if (d < dist)
            dist = d
            id = p.id
     return (id, dist)

on_receiving_query_distance(p)
    r.id = my_trigger_id
    r.time_stamp = curr_time
    r.src_dist = src_dist
    send_pkt(p.requester, r)

on_receiving_pkt_from_src(p)
    src_dist = curr_time − p.time_stamp
```

Figure 6: Joining multicast protocol.

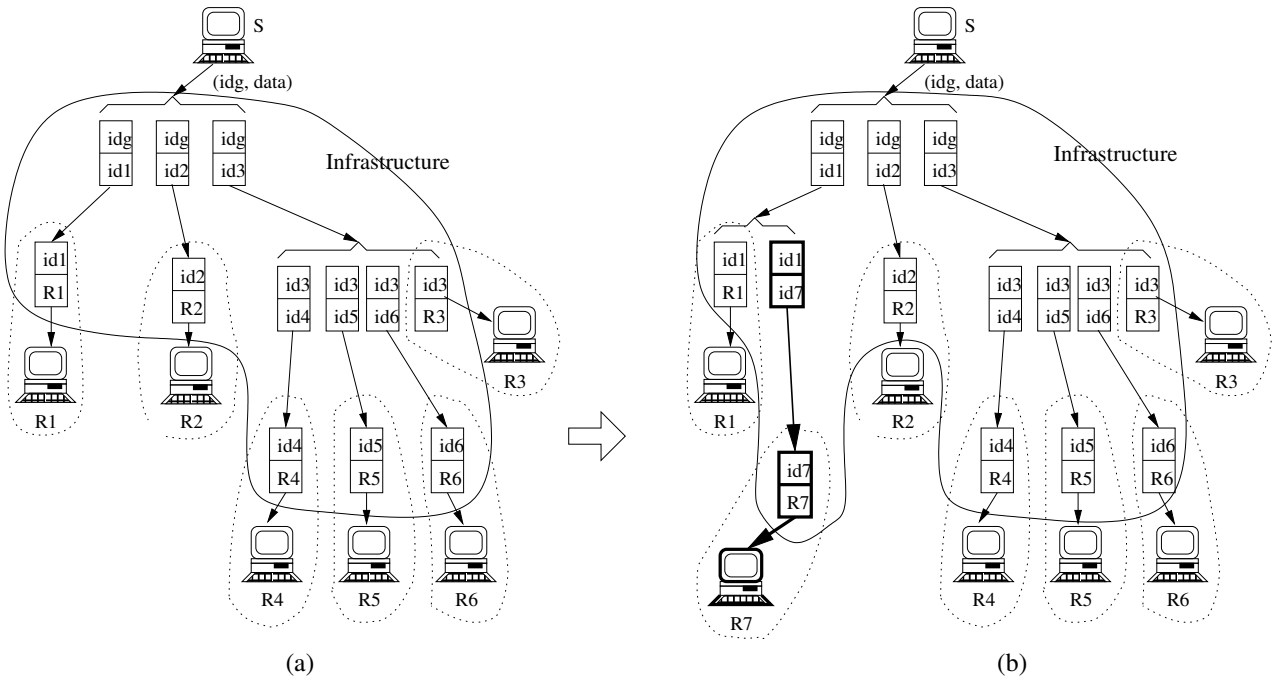


Figure 7: The tree of triggers corresponding to the join operation shown in Figure 5.

3.2 Infrastructure-based Realization of the Tree Construction Algorithm

In this section, we show how the end-host multicast protocol described in the previous section is implemented using the infrastructure primitives alone. Packets will then be replicated in the infrastructure, rather than end-hosts, which will considerably improve the performance and the resilience of the tree.

The main primitive used by the tree construction algorithm is packet replication. To get around the fact that an infrastructure node can generate only a small number of replicas, we build a tree of triggers as proposed in [40]. Figure 7(a) shows a possible tree of triggers that corresponds to the multicast tree in Figure 5(a). Indeed, if we collapse all triggers with the same identifiers, and each receiver R_i with its trigger (id_i, R_i), then the tree in Figure 7(a) reduces to the end-host multicast tree in Figure 5(a).

Each trigger (id_i, R_i) is assumed to be located on a server close to receiver R_i. A new node R_k joins the multicast group at an ID id by inserting two triggers (id, id_k) and (id_k, R_k), where id_k is an ID located at an $i3$ server close to R_k. For example, in Figure 7(b), R_7 joins at ID id_1 by inserting triggers (id_1, id_7) and (id_7, R_7), respectively. To maintain the one-to-one mapping between the multicast trees in Figures 5(a) and 7(a), we slightly change the definition of *joinable* and *full* nodes to take into account the fact that a receiver that joins at an internal node in the multicast tree requires an additional trigger. In particular, we

say that receiver R is *joinable* if there are less than $D+1$ triggers with ID id, and *full* if there are exactly $D+1$ triggers with ID id in the system. Node R_3 in Figure 7(a) is a full node since there are four triggers with ID id_3 in the system. In contrast, all other receivers are joinable.

The main challenge in realizing the pseudocode shown in Figure 4 in $i3$ is to efficiently implement $dist()$ and $select\_node()$ functions. To address these challenges, we use two techniques, (a) $i3$ packet replication to implement $select\_node()$, and (b) a simple scheme based on local time-stamps to compute the relative value of $dist()$, which we describe in Section 3.2.1.

Figure 6 shows the pseudocode of the joining procedure in $i3$. To implement $select\_node()$, all receivers in a set (*jset* or *fset*) will subscribe to a unique multicast group. Consider a receiver R that is connected through triggers (id', id) and (id, R) to the multicast tree. Let $jHash$ and $fHash$ be two well-known hash functions. If R is joinable, R will insert the trigger $(jHash(id'), R)$ in $i3$. Thus, $jHash(id')$ identifies the *jset* to which R belongs. Otherwise, if R is full, it will join the corresponding *fset* by inserting the trigger $(fHash(id'), R)$. For example, receivers R_1 and R_2 will join the *jset* identified by $jHash(id_g)$, while R_3, which is a full node, will join the *fset* identified by $fHash(id_g)$. Since each node goes through each level of the tree at most twice (once each for $jset$ and $fset$), it receives at most $O(D)$ messages at each level, and hence the complexity of the join algorithm in the number of messages is $O(lD)$ where l is the number of levels of the tree. Assuming that $l = O(\log N)$, the join complexity is $O(D \log N)$.

When node R_{new} wants to find the end-host R in a set (identified by, say, id) that minimizes the distance from S to R_{new}, it simply sends a request message with identifier id. In turn, each receiver R that receives this message (*i.e.,* each receiver in the set id) will send a reply to R_{new} along with its estimated distance from sender. Next, we discuss how this distance is estimated and used.

3.2.1 Distance Computation

When computing $dist()$, it is important to note that the pseudocode in Figure 4 uses the results returned by $dist()$ only for comparison proposes. In this section, we show that it is possible to compare the distances from S to R_{new} via different intermediate nodes using local time-stamps alone, and without assuming global clock synchronization. In particular, we compute such distance as a function of local time-stamps

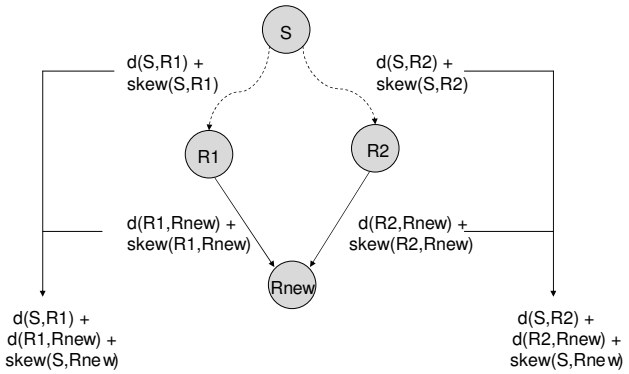


Figure 8: Timing of messages for distance computation.

and the clock skews, and then show that the clock skews cancel out when performing distance comparisons. Figure 8 shows the messages to R_{new} through two receivers R_1 and R_2.

For this description, we assume that, since (id_i, R_i) is located at a server close to receiver R_i, the last hop latency from id_i to R_i can be neglected. For example, in Figure 5(a) we assume that the latency from S to R_7 approximates the latency from S to R_1 plus the latency from R_1 to R_7, and hence we use $dist(S, R_1, R_7)$ to approximate $dist(S, id_1, R_7)$. Note that this assumption might not hold in practice.[3] In such cases, the simplified scheme in the worst case would not consider some low delay paths. However, as long as there is a significant fraction of end-hosts for which the above assumption holds, even the simplified join protocol should still perform well. In practice, to remove this assumption, we can use explicit last hop measurements using mechanisms described in Section 3.2.2.

Let $d(A, B)$ denote the latency between nodes A and B. Then, we have:

$$dist(S, R, R_{new}) = d(S, R) + d(R, R_{new}). \tag{1}$$

Let $p.depart(A)$ denote the *local* time at A when packet p departs from A, and let $p.arrive(B)$ denote the local time at B when packet p arrives at B. Then, we have

$$d(S, R) = p.arrive(R) - p.depart(S) + skew(S, R), \tag{2}$$

where p is a packet that travels from S to R and $skew(S, R)$ denotes the clock skew between S and R. Similarly,

[3]For example, in the case of cable modems the last hop latency can be significant.

$$d(R, R_{new}) \;=\; p'.arrive(R_{new}) - p'.depart(R) + skew(R, R_{new}),$$

where p' is a packet traveling from R to R_{new}. By combining Eqs. (1)-(3), we obtain

$$\begin{aligned} dist(S, R, R_{new}) \;=\; & (p.arrive(R) - p.depart(S)) + \\ & (p'.arrive(R_{new}) - p'.depart(R)) + skew(S, R_{new}), \end{aligned} \tag{3}$$

where we replaced $skew(S, R) + skew(R, R_{new})$ by $skew(S, R_{new})$. Note that the value $(p.arrive(R) - p.depart(S))$ is computed by R upon the arrival of packet p from S, and sent to R_{new} in the reply message as $r.src\_dist$ (see $on\_receiving\_packet\_from\_src()$ and $on\_receiving\_query\_dist()$).

The important point to note in Eq. (3) is that $skew(S, R_{new})$ does *not* depend on R. This allows us to use the sum $(p.arrive(R) - p.depart(S)) + (p'.arrive(R_{new}) - p'.depart(R))$ instead of the absolute value of $dist(S, R, R_{new})$ to *compare* the distances between S and R_{new} via any intermediate node R.

In this mechanism, all joins start at the root of the tree. To reduce the processing overhead at the root of the tree, we can use well-known techniques such as replication at the root level (as used in [7, 22]). In addition, we can use the anycast primitive to spread the load across multiple root servers.

3.2.2 Discussion

We have considered a multicast tree that aims to optimize for end-to-end latency, so far. In order to adapt the same protocol for other metrics such as bandwidth and loss rate, the end-hosts have to report the bandwidth or the loss rate instead of the latency (in the join reply message). The fact that we cannot neglect the last hop bandwidth or loss-rate implies that reports received by end-hosts (while joining) can be used only as hints rather than accurate values.

To address this issue, one could either enhance the infrastructure nodes to perform measurements on behalf of end-hosts or have the joining end-host perform additional measurements. In the latter approach, instead of relying on performance information contained in join replies, a joining node R_{new} can insert a trigger (id, R_{new}) and directly measure the performance metric it is interested in. This approach is similar

to the joining algorithm employed by Overcast [22]. By such an approach we can get more efficient trees at the expense of additional overhead.

We have discussed the case of a single source sending data to the multicast tree. However, this is not a fundamental restriction. As long as a member of the multicast group knows the group identifier id_g, it can send data to the group. The only cost incurred by this simple scheme is that packets originated at sources farther away from the location of id_g, will experience higher latencies.

The join procedure assumes that at each level there is at least a non-empty *jset* or *fset*. However, in time all receivers at a level may leave which will cause these sets to become empty. To alleviate this problem, each receiver periodically probes the parent level and checks if it can join the parent level. We can limit the control traffic by regulating the periodicity of probes.

Finally, each receiver needs to decide whether it is joinable or full and join the corresponding group. We now describe a simple scheme to address this problem. Consider a receiver R connected to the tree through trigger (id, R). R can periodically try to insert a dummy trigger (id, x). A successful insertion means that id is joinable and R joins *jset*. R then immediately removes the trigger (id, x). If unsuccessful, then there are already D triggers with the ID id in the system, and as a result R joins *fset*.

4 Scalable Tree Maintenance

Recall that $i3$ uses a soft-state approach to maintain triggers in the system. If a trigger is not refreshed for a pre-defined period of time T (which in our implementation is 30 sec), the trigger is removed from the system. The issue we address in this section is that of scalably maintaining the triggers in the tree. A naive solution would have each receiver independently refresh all triggers on the path from the source to itself. For example, in Figure 7, R_6 would be responsible for refreshing triggers (id_g, id_3), (id_3, id_6), and (id_6, R_6). The problem with this approach is that the number of refreshes received for a trigger increases exponentially as we move up the tree.

To address this problem, we introduce a control message, denoted by REFRESH_ACK, to suppress redundant refresh messages. When receiver R refreshes trigger (id_i, id_j), it also sends a REFRESH_ACK

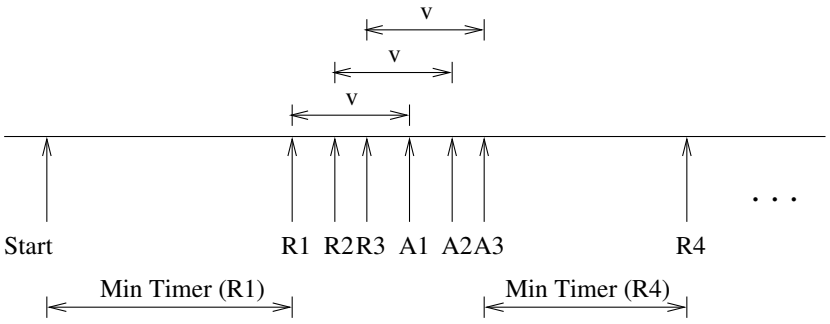


Figure 9: Timing of refresh messages and refresh acks.

message to id_j and hence this message will be multicast to the entire sub-tree rooted at id_j. Upon receiving a REFRESH_ACK message for a given trigger, a receiver simply resets the timer to refresh that trigger.

However, if all receivers refresh a trigger at the same time, REFRESH_ACK will do nothing to suppress any of these refreshes. The classic solution to address this problem is to randomize refresh timers. Let a receiver choose a random timer that is uniformly distributed in the interval $[\alpha T, \beta T)$ and let n be the number of receivers in the sub-tree rooted at trigger t. If all the receivers set the timer at the same instant, then the expected time for the first refresh timer to expire can be computed to be $\alpha T + (\beta - \alpha)T/(n + 1)$. By assuming that the delay in receiving the REFRESH_ACK message is zero, *i.e.,* all receivers are instantaneously notified of the trigger refresh, the expected number of refreshes for a trigger in a trigger refresh period T is $1/(\alpha + (\beta - \alpha)/(n + 1))$.

In order to ensure that the expected number of refreshes per refresh period is constant, we pick α as 0.5. Hence, assuming that receivers are informed of the trigger refreshes instantaneously, the expected number of refreshes for a trigger in a refresh period is at most 2. However, in practice, REFRESH_ACK messages have to be multicast to all the receivers in the sub-tree, and hence more refresh timers might get triggered during that period of propagation.

Let us assume now, for simplicity, that all the receivers get the REFRESH_ACK message for the first refresh after a time v from the time the timer is triggered; v is the vulnerability period during which false timer expirations can happen. Figure 9 shows an example where R_1 sends a refresh first, and in the vulnerability period, two other receivers R_2 and R_3 also send refresh messages. However, all the timers

are reset when the REFRESH_ACK for R_3 arrives. In a vulnerability period, the number of new refreshes that are triggered is roughly $nv/T(\beta - \alpha)$. Since $v \ll T$ in practice, the number of refreshes that are generated in a refresh period is bounded by $1/(\alpha + (\beta - \alpha)/(n + 1)) + nv/T(\beta - \alpha)$.

To reduce the number of refreshes due to false timer expirations, the receivers at levels one and two choose the refresh timer uniformly at random from the interval $(\alpha T, \beta T)$, and all other receivers that are further away choose the timer from the interval $(\beta T, T)$. Thus, a trigger will be almost always refreshed by the receivers at the first and second levels, and the timers at lower levels will be suppressed. In particular, for a trigger with a complete subtree at levels one and two, the number of refreshes is roughly $1/(\alpha + (\beta - \alpha)/(D^2 + D + 1)) + (D^2 + D)v/T(\beta - \alpha)$. Note that since D is a system-wide constant, the number of refreshes does not depend on the size of the tree. To accommodate message losses we decrease the timer intervals by a factor k. In practice, we choose $k = 3$. We compare some of these different strategies by simulation in Section 6.

5 Reliable Packet Delivery

Developing reliable multicast solutions has proven to be a difficult and challenging problem. Even when the assumptions allow changes in the infrastructure, the resulting solutions are complex and exhibit undesirable tradeoffs. For instance, with SRM [16], there is a clear tradeoff between the number of duplicates and the time to recover from failure. This section demonstrates the flexibility of our architecture by presenting a solution for reliable multicast that is both simple and scalable. To reduce the number of duplicate packets, our solution leverages the ability to multicast a packet to a sub-tree in the multicast tree. For instance, in the topology in Figure 7(a) one can multicast a packet only to receivers R_4, R_5, and R_6 by sending the packet to identifier id_3. In addition, to avoid the NACK implosion problem, our solution leverages the anycast capability offered by $i3$.

Figure 10 shows the pseudocode of the recovery procedure. The main idea is to associate a recovery *anycast* group with each internal node in the tree. Each receiver subscribes to exactly one anycast group, that is, the anycast group associated with its parent. In the example in Figure 5(a) there will be two

recovery anycast groups: one associated with the source S consisting of receivers R_1, R_2, and R_3, and another one associated with receiver R_3 consisting of receivers R_4, R_5, and R_6, respectively.

We assume that all data packets have a unique sequence number and that losses are detected when packets arrive out of sequence.[4] When a receiver R detects a packet loss, it sends a repair request for the lost packet to its anycast group. Upon receiving a repair request, a receiver R_a checks whether if it has the requested packet, and if it does sends the packet directly to the requester (R) via IP unicast. If not, R_a assumes that everyone in its anycast group has lost the packet, and it takes the responsibility for recovering the lost packet by sending a repair request to its parent's anycast group. If the recipient of the repair request at the higher level has the packet, it sends the packet to everybody in the subtree rooted at the node corresponding to the requester's anycast group. If the packet is absent at the higher level, the recovery procedure proceeds recursively.

To illustrate the repair procedure consider the multicast tree in Figure 7(a). Assume receiver R_4 loses a packet. As a result it sends a repair request to the anycast group consisting of nodes R_4, R_5, and R_6, respectively. Assume this repair request is delivered to R_6. If R_6 has the packet, it sends it directly to node R_4, and we are done. If not, R_6 forwards the repair request to its parent anycast group, that is, to the anycast group consisting of R_1, R_2, and R_3. Assume the repair request is delivered to R_2 and that R_2 has the requested packet. Then R_2 will send a repair to id_3. As a result, the repair will be multicast to all receivers in the sub-tree routed at id_3 including R_4 and R_6.

We now address the question of how to construct the anycast identifiers. Consider a receiver R which is connected to the multicast tree by triggers (id_{i-1}, id_i) and (id_i, R), respectively. Then R_i will insert a trigger (id_a, R) where $id_a = H_r(id_{i-1})|loss\_rate(R)$. The sign "|" represents the concatenation symbol. $H_r()$ is a well-known hash function that returns 192 bit values, and $loss\_rate(R)$ represents R's loss rate as measured by R. When a receiver R loses a packet, it sends a repair request with identifier $id_{req} = H_r(id_{i-1})|0$. Since $i3$ uses the longest prefix matching scheme to match the packet and the trigger identifiers (see Section 2), the repair request will be delivered to the sibling of R which experiences the

[4] In practice, one may allow a certain number of out-of-sequence packets before concluding that a packet was lost. This would allow us to tolerate packet reordering.

```
// function called on packet loss
on_packet_loss(seq_num)
    // send packet to anycast group at my level to respond
    // to loss of seq_num, reply to be sent to my_addr
    request_repair(repair_group[my_level], seq_num, my_addr, TO)

request_repair(anycast_id, seq_num, req_id, to)
    send_repair_req(anycast_id, seq_num, req_id)
    set_timer(curr_time + to, request_repair, anycast_id, seq_num, req_id, to)

on_receiving_repair_req(q)
    // repair request already received ?
    if (pending_repair_req[q.seq_num] = FALSE)
        // get repair packet
        if (r = get_packet(q.seq_num, packet_queue))
            send_repair(r, q.req_id)
        else
            // doesn't have the repair packet; send request up the tree
            request_repair(repair_group[my_level - 1], q.seq_num, parent_id, TO)
            pending_repair_req[q.seq_num] = TRUE

on_receiving_repair(r)
    pending_repair_req[r.seq_num] = FALSE
    remove_timer(request_repair, seq_num)
```

Figure 10: Pseudocode of the recovery procedure.

lowest loss rate.

A potential problem with this scheme is that when the packet is lost at a higher level of the tree, *all* receivers belonging to the same recovery anycast group at a lower level will send repair request messages. Let R_a be the receiver in the anycast group that experiences the lowest loss rate. Then, in the worst case, R_a receives up to $2 \times D$ repair request messages, where D is the maximum out-degree of the multicast tree: D repair requests from all member of R_a's anycast group (including R_a itself), and D repair requests from the level immediately below. We present two techniques to alleviate this problem.

In the first technique, a receiver that detects a packet loss waits for a random period of time, uniformly distributed in the interval $[0, T)$, before sending a repair request for the missing packet. Assume a packet is lost at a higher level. Let t_0 denote the time when the first repair request for that packet is sent, and let

$t_0 + \Delta$ be the time when the repair arrives at end-hosts which lost the packet. Then all repair requests that were scheduled to be sent after time $t_0 + \Delta$ will be suppressed. As a result, the number of repair requests delivered to the receiver with the lowest loss rate from a recovery anycast group decreases from $2 \times D$ to roughly $2 \times D \times \Delta/T$. This is because the probability that a receiver (which has lost the packet) to send a repair request during a time interval of length Δ is Δ/T. On the downside, this solution increases the time to receive the repair. If k receivers lose a packet, it will take the receiver an additional $T/(k+1)$ time to receive the repair on average.[5]

So far we have assumed that the repair requests are always delivered to the member of the anycast group which experiences the lowest loss rate. The idea of the second technique is to spread the responsibility to answer the repair requests among the members of the repair anycast group. To achieve this we redefine the anycast identifier of receiver R as $id_a = H_r(id_{i-1})|H_r(IP_R)$, where IP_R represents the IP address of receiver R. When a receiver R loses the packet with the sequence number seq_no, it sends a repair request with identifier $id_{req} = H_r(id_{i-1})|H_r(seq\_no)$. Hence, all repair requests for the same packet are delivered to the same receiver, while repair requests for different packets are delivered to different receivers, thus achieving load balancing. The downside of this approach is that the receiver to which the repair requests are delivered may have poor reception characteristics. Section 6.3 uses simulation experiments to compare these two solutions.

6 Simulations

In this section, we evaluate our algorithm using simulations. Our results focus on (1) the efficiency of tree construction, (2) scalability of the trigger refresh mechanism, and (3) the efficiency and scalability of the reliability mechanism. Since the main contribution of this paper is architectural, our experiments are meant to show that our algorithms provide reasonable performance. We do not perform a detailed comparison of the performance of our implementation with different earlier multicast proposals since most of them can be adapted to our architecture.

[5]The additional $T/(k+1)$ represents the expected delay of sending the repair request.

Our simulator is based on the $i3$ protocol described in [40]. In our simulations, we consider a 10,000-node transit stub topology generated using the GT-ITM topology generator [18].[6] Link latencies are chosen uniformly at random between 1 and 3 ms for intra-transit domain links, between 5 and 10 ms for transit-stub links, and between 20 and 50 ms for inter-transit links. $i3$ servers and multicast members are attached to stub nodes chosen uniformly at random. In all experiments, we assume $8,192$ $i3$ servers[7], and that each multicast member knows a set of identifiers that maps on to the closest $i3$ server. These identifiers can be discovered by using offline sampling [40] or through service discovery mechanisms. We run simulations with a group size (the number of receivers in the multicast tree) of up to 65536. To support a group sizes larger than that of the topology, we assume that each stub in the topology is actually a LAN with one $i3$ server on it. We neglect the delay on the LAN while computing path latency.

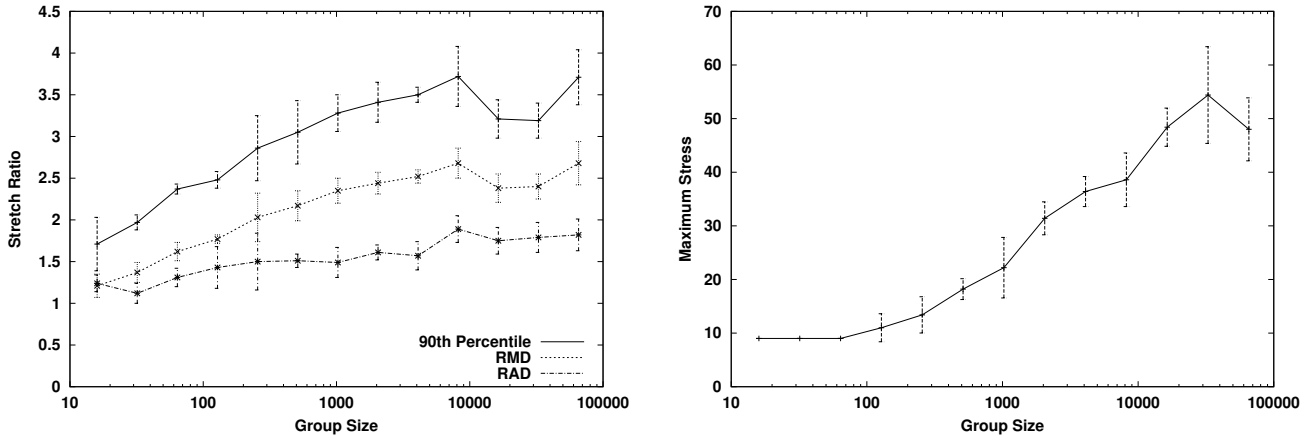
6.1 Tree construction

To test the scalability and the efficiency of the tree construction algorithm described in Section 3, we ran a number of simulations varying the multicast group size from 16 to 65536. In our evaluation, we use four metrics, which were previously introduced in [8, 10]:

- *Latency Stretch,* the ratio of the latency from the source to receiver in $i3$ to the latency along the shortest path (IP latency).
- *Ratio of the Maximum Delay (RMD),* the ratio between the maximum delay using $i3$ multicast and the maximum delay using IP routing.
- *Ratio of the Average Delay (RAD),* the ratio between the average delay using $i3$ multicast and the average delay using IP routing.
- *Link stress,* the number of copies of each multicast packet that travels through a given link.

[6] We have also ran simulations using power-law network topologies and obtained similar results.

[7] The increase in the number of links and the time taken to compute shortest paths made it difficult to scale the simulation to a larger topology.



(a) (b)

Figure 11: (a) The latency stretch of the multicast tree, (b) The maximum stress caused by the overlay multicast tree on the links of the underlying network.

Figure 11(a) plots the 90^{th} percentile latency stretch, the RAD, and the RMD between the source and the receivers. As expected, the stretch increases only logarithmically with the size of the network. In addition, these results indicate that the resulting multicast trees could meet the latency requirements of a large number of applications. Indeed, even for a group of size 65536, the RMD and RAD do not exceed 1.5, and the 90^{th} percentile of the latency stretch does not exceed 3.5. Also, the stretch stabilizes for the very large group sizes. This is because beyond a certain group size almost every $i3$ server is already receiving the multicast packets, and there is very little overhead in adding additional receivers.

Implementing multicast at the application layer results in multiple copies of a packet being sent on the same physical link. Figure 11(b) plots the maximum stress observed on any link. While the maximum stress increases with the group size, we note that this increase is sub-linear and that even for a group of size 65536, it does not exceed 50. The reason that this is larger than the out-degree of the multicast tree is partly because of using an overlay network for multicast, and partly due to the greedy heuristic we use in our join algorithm. We also found that only a very few links have a high stress, for example, even in the largest group size the 99^{th} percentile stress was only 18.6.

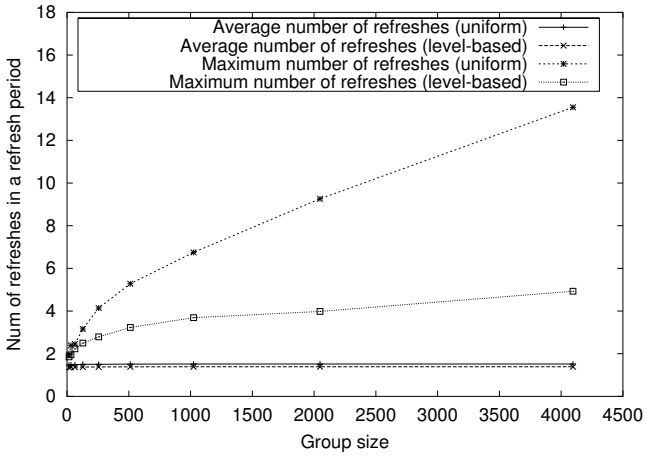


Figure 12: Maximum and average number of refreshes per refresh period

6.2 Scalable trigger refreshing

In this section, we evaluate the refresh schemes described in Section 4. With the first scheme, called *refresh-unif*, each receiver refreshes each trigger in its chain by using timers uniformly distributed in the interval $[T/2, T)$, where T is the refresh period for triggers in $i3$. In contrast, with the second scheme, called *refresh-level*, a receiver uses timers uniformly distributed in the interval $[T/2, 3T/4)$ to refresh the triggers one and two levels above it in the hierarchy, and timers uniformly distributed in the interval $[3T/4, T)$ to refresh all the other triggers in its chain.

Figure 12 plots the maximum and the average number of refreshes per trigger during a time interval of length T, as a function of the multicast group size. As expected, the *refresh-level* scheme reduces the maximum number of refreshes per trigger significantly and the difference increases as the size of the group increases. This is because with *refresh-level* the refreshes of the receivers placed at more than two levels down the hierarchy are suppressed with a very high probability. In contrast, the average number of refreshes are roughly the same for both schemes (i.e., about 1.5), with *refresh-unif* performing slightly better. This is because receivers near a trigger are more aggressive in the case of *refresh-level* (their timers are three times smaller on average), which results in a slightly larger number of refreshes.

6.3 Reliability

We evaluate the recovery procedure described in Section 5. We assign loss rates uniformly at random between 0-4% to $i3$ hops, and between 0-8% to the hops from $i3$ servers to end-hosts. Both data and control packets are dropped with the same probability. On detecting a loss, a receiver sends a repair request after a time randomly chosen between 0 and 300 ms. This optimization decreases the chance of a receiver sending a repair request before the repair data triggered by another request is received. The timeout for re-sending a repair request is 800 ms.

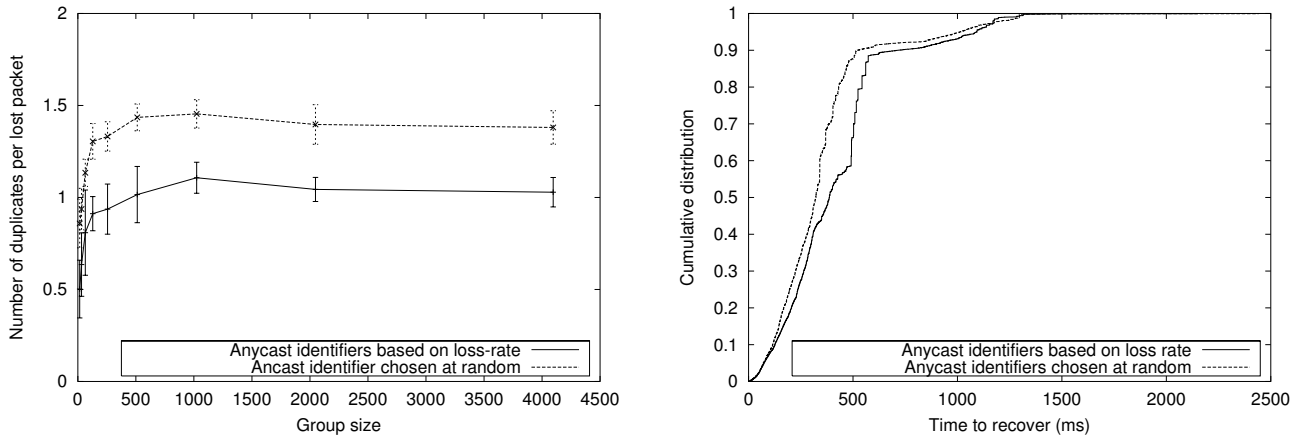
We evaluate the two schemes presented in Section 5: *rel-loss*, where the repair request is delivered to the receiver which experiences the lowest loss rate, and *rel-random*, where the repair request is delivered to a random receiver within the same recovery anycast group. In our evaluation, we consider three metrics: (1) the number of packet duplicates per packet loss, (2) the time it takes to receive the repair, and (3) the number of repair requests received by an end-host.

Figure 13(a) plots the number of packet duplicates per packet loss. Ideally, this value should be zero. However, due to the fact that repairs might be multicast, receivers who haven't lost the packet would receive duplicates. As shown in Figure 13(a) both schemes perform well; in either case the number of duplicates per packet loss is less than 1.5 and practically remains constant at large group sizes[8]. However, as expected, the *rel-loss* perform slightly better. This is because, the receiver that gets the repair requests is the one that experiences lowest loss rate and is hence more likely to have the packet.

Figure 13(b) plots the CDF of the recovery time, that is, the time it takes to receive a repair from the moment the loss was detected. Two points merit mention. First, in 85% of the cases, the repair arrives within 600 ms, thus obviating the need to send a second repair request. Second, in our experiments, all repairs arrive before 2400 ms, so a third repair request is almost never needed.

Finally, Table 1 gives the *average* number of repair requests per packet loss, and the *maximum* number of repair requests per sequence number received by an end-host for both schemes. In terms of the average number of repair requests, both schemes perform similarly, with each of them generating about 1.2 repair

[8]For comparison, in SRM [16], if only one receiver loses a packet, all receivers would receive a duplicate packet.



(a) (b)

Figure 13: (a) Total number of duplicates as a fraction of total number of drops, (b) Cumulative distribution of time to recover for 1024 receivers.

Group size	# repairs per drop (*rel-loss*)	# repairs per drop (*rel-random*)	max repairs per seq # (*rel-loss*)	max repairs per seq # (*rel-random*)
16	1.12 (0.12)	1.15 (0.16)	1.00 (0.07)	0.18 (0.03)
64	1.07 (0.16)	1.11 (0.12)	1.92 (0.28)	0.31 (0.04)
256	1.16 (0.05)	1.18 (0.12)	2.87 (0.25)	0.44 (0.06)
1024	1.18 (0.08)	1.19 (0.07)	3.89 (0.24)	0.64 (0.05)
4096	1.13 (0.04)	1.16 (0.070)	5.09 (0.54)	0.84 (0.08)

Table 1: Number of repairs as a fraction of drop rate, and maximum number of anycast requests per sequence number for rel-random and rel-loss (standard deviation in parentheses).

requests per packet loss. *rel-loss* performs better because the end-hosts receiving the repair requests are more likely to have the lost packet, which reduces the chance of forwarding the request at the higher level. However, in terms of maximum number of repair requests per packet loss, *rel-random* performs better. In particular the maximum number of repair requests per packet loss is about five times larger in the case of *rel-loss* for 4096 receivers. This is because, with *rel-loss*, the repair requests are delivered to the same receiver, while with *rel-random* repair requests are spread among different receivers.

In summary, while *rel-loss* reduces the number of duplicate packets and the total number of repair requests, *rel-random* is more effective in avoiding hot-spots due to repair requests being delivered to the same receiver.

7 Implementation

We have implemented a prototype of the reliable multicast protocol described in this paper on top of $i3$ [40] in about 2000 lines of C/C++ code. To test our protocol, we conduct experiments on two platforms: Millennium, a large cluster of PCs at UC Berkeley [1], and the PlanetLab test-bed [34]. The experiments on Millennium are aimed at testing the implementation for medium size groups in a repeatable manner, and the second set of experiments demonstrate the performance of our protocol in a more realistic scenario.

7.1 Millennium Experiments

Our experiments on the Millennium test-bed involve 32 $i3$ servers and 64 multicast clients. To test our implementation for trees with multiple levels, we use $D = 4$ instead of $D = 8$ (as used in simulations).

Table 2 (a) shows the depth of the resulting multicast tree as function of the group size. As expected, the depth of the tree is roughly logarithmic in the group size. In particular, the depth of the tree for 64 clients is 4.5 which is only 1.5 times as large as the depth of a perfectly balanced tree with 64 nodes.

To evaluate the overhead of the joining procedure, in Table 2 (b), we give the number of control messages (i.e., $query\_distance$ messages in Figure 6) processed by a receiver as a function of its level in the tree. The number of messages increases exponentially with the height of the receiver in the tree. This is because the joining procedure starts always from the root, and, as a result, receivers at the first level will be contacted by every new receiver joining the tree. Since our current implementation takes about 200 μs to process a $query\_distance$ message, we can support a few thousands of new clients joining every second. While this number is large enough for most practical applications, for very large groups the processing of $query\_distance$ messages can still be a bottleneck. This can be alleviated using standard techniques such as replicated root servers (as used in [7, 22]). Alternatively, we can use the anycast primitive to spread the load across multiple root servers.

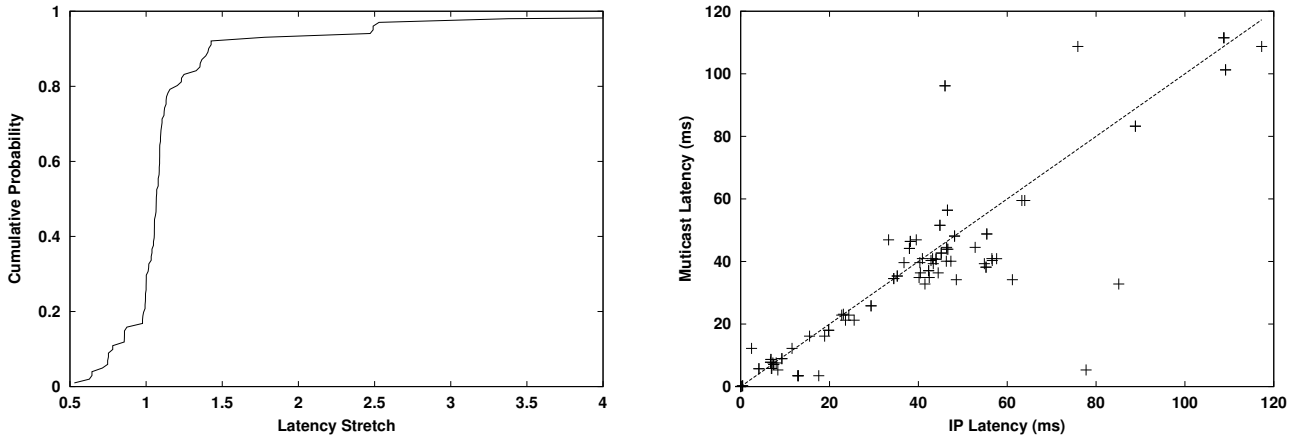
Group size	Depth of the tree
8	2 (0)
16	2.8 (0.13)
32	3.4 (0.22)
48	3.9 (0.21)
64	4.5 (0.22)

(a)

Level	# of messages
1	55.9 (2.19)
2	10.1 (0.62)
3	2.36 (0.68)
4	0.88 (0.40)
5	0.066 (0.03)

(b)

Table 2: Results from experiments on Millennium showing the depth of the tree and the number of $query\_distance$ messages processed (standard deviations in parentheses).



(a) (b)

Figure 14: (a) CDF of the stretch, and (b) scatterplot of multicast latency vs. IP latency for PlanetLab experiments.

7.2 PlanetLab Experiments

To evaluate our protocol in a realistic scenario, we performed experiments over the PlanetLab testbed [34]. In our experiments, we use 50 nodes at different locations in USA, Asia and Europe.[9] Each node runs an $i3$ server and two multicast clients. Since $i3$ performs caching, the latency along each hop in the constructed tree is merely the IP latency. Furthermore, since each end-host runs an $i3$ server, each receiver will pick a trigger located on the same end-host. In each experiment, we have the multicast tree rooted at Berkeley and we report results after the multicast tree is fully constructed.

Figure 14(a) plots the cumulative distribution function (CDF) of the latency stretch. We observe that

[9]The reason for using only 50 nodes despite the apparently larger size of PlanetLab is because most of the PlanetLab machines were unusable for experiments for a variety of reasons at the time of experimentation.

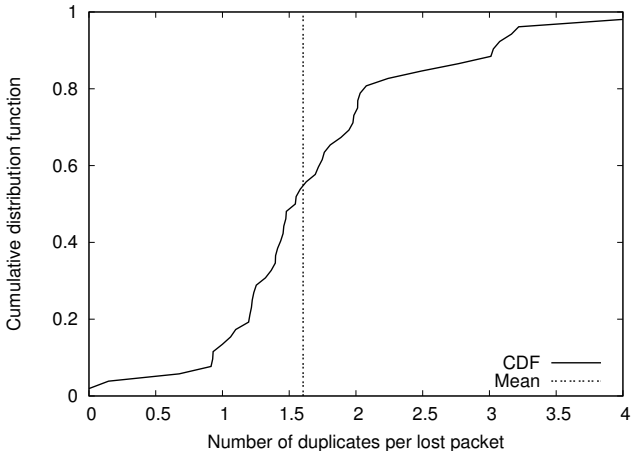


Figure 15: CDF of ratio of number of duplicates to number of losses at a node.

in 85% of cases the stretch is less than two, *i.e.,* the multicast tree latency is less than twice the IP latency.

To provide more insight into the relationship between the multicast latencies and IP latencies, Figure 14(b) plots the latencies in the multicast tree versus the IP latencies for each (sender, receiver) pair. The points above the line correspond to a stretch larger than one, while the points below the line correspond to a stretch smaller than one. We observe several points corresponding to a stretch lower than one mostly due to IP routing inefficiencies, and partly due to the variability of our measurements to estimate network distances taken a few minutes apart.

To evaluate the reliability algorithm we instrumented the $i3$ servers to artificially drop packets since the actual loss we observed was very low. We assigned loss rates exactly from the same distribution as in the simulation section—uniformly at random between $0-4\%$ for $i3$ hops and $0-8\%$ for last hop between $i3$ servers and end-hosts. Figure 15 shows the CDF of the ratio of number of duplicate messages to the number of lost packets at the multicast clients when *rel-loss* algorithm is used. The observed mean, around 1.6, is consistent with our simulation results.

8 Related Work

Architecturally, separating routing control from forwarding has been proposed in a variety of contexts ranging from resolving the problems of BGP (RCP [15]) to ATM/MPLS networks [24]. Particularly, in the context of multicast, Keshav and Paul [23] have proposed providing multicast routing in a centralized

manner. While we share the same philosophy as these works, we differ in two important ways. First, instead of centralizing route computation, we explore how the end-hosts can construct the trees over the infrastructure in a distributed manner. Second, our infrastructure runs as an overlay and exports simple primitives that end-hosts can use. While our solution does not require a centralized entity to perform route computation, our trees are in general less efficient than those computed by a centralized entity since we don't have the view of the entire network.

Jannotti [21] has argued for enhancing the network to provide more support for construction of efficient multicast trees. The two primitives that are proposed are *path reflection*, which allows end-hosts to request redirection and replication at nearby routers, and *path painting*, which allows end-hosts to find the intersection point of their paths to a common destination. In contrast, our infrastructure is simpler but exports primitives that provide hosts greater flexibility in picking replication points in the network.

Broadly, most current multicast proposals fall into one of two categories: infrastructure-based or host-based. Among the infrastructure-based solutions are IP-level multicast solutions such as IP Multicast [12], CBT [4], PIM [14] and EXPRESS [20] and application-level (or overlay-network) solutions such as Overcast [22], Scattercast [9] Yoid [17], and ALMI [33]. Since the multicast tree construction algorithm is typically implemented at the nodes these solutions have a low degree of flexibility. Some of the host-based multicast protocols include ESM [10], Peercast [13] and NICE [5]. While these solutions are highly flexible since they don't have a deployed infrastructure, they often suffer from scalability and robustness concerns.

In the multicast literature, many different approaches have been adopted for tree construction. Some proposals (such as ESM [10] and [45]) construct trees over an overlay mesh. Approaches such as Overcast [22] and NICE [5] use searching techniques over a hierarchical structure; our tree construction algorithms in this paper are very similar in spirit. More recently, peer-to-peer lookup and routing protocols [19, 35, 37, 41] have also led to the development of efficient multicast solutions [8, 29, 36, 38, 46]. Finally, cooperative strategies to improve the throughput of multicast trees have been employed in CoopNet [31] and SplitStream [7]. The techniques used in such approaches are orthogonal and can be leveraged in our architecture.

There have been many attempts to provide reliable multicast functionality. SRM [16] adds support for reliability by multicasting the repair requests and retransmitting the data. While this solution (and others such as TMTP [44]) address the problem of NACK implosion, the number of duplicates can be very high. This is because repairs are multicast to all members of the group. LMS [32] presents a finer grained recovery service by creating a dynamic hierarchy of servers. Similarly, STORM [43] uses a self organizing overlay network to provide recovery. However, these solutions are quite complex since they require the application to build and maintain an overlay network for recovery purposes only. In addition, they are based on IP multicast which limits their deployability. Digital Fountain [6], WEBRC, and RLM [30] provide reliability and/or congestion control through erasure coding. Since these schemes would work on any multicast protocol that does best effort delivery of data, they can be deployed on top of our multicast protocol also. ROMA [25] uses erasure coding with loosely coupled TCP connections to achieve reliable multicast; in contrast, we explore how end-hosts can achieve reliability without infrastructure support.

9 Discussion

In this section, we discuss potential alternatives to our design choices.

Infrastructure support. We have used a minimalist approach in choosing what functionality to embed in the infrastructure. Another possibility would be to extend the infrastructure to implement part of the control plane by aiding the end-hosts in measuring various performance metrics or in performing error recovery. While this is a valid approach, in this paper we demonstrate that it is indeed possible to remove end-hosts completely from the data path by using a few basic primitives.

There are two advantages of assuming a simple infrastructure with a well-defined interface. First, it is easier to implement the functionality of such infrastructure at high speeds. For example, the main operation performed by an $i3$ node on the data path is longest prefix matching, for which there are well-known fast algorithms (e.g. [42]). Compare this with implementing packet recovery, which requires maintaining and managing a buffer for each group, and processing the ACK/NACK packets. Second, a well-defined interface allows the infrastructure and the tree construction algorithms to evolve independently.

Centralized vs. Distributed tree construction. We have assumed that the entire control plane, including tree construction and message recovery, is implemented by end-hosts. An alternative, would be to provide the multicast functionality as a third-party service. A single entity would keep track of all the members of the group as well as construct and maintain the tree on their behalf in a centralized manner. Besides relieving the end-hosts of the tree-management burden, this model could also provide better security. For example, since only the third-party needs to be aware of the IDs used internally in tree, we can enforce fine-grained access control over who can send packets to each sub-tree [3]. We discuss a complete solution based on this approach in [28].

Deployment. In this paper, we implicitly assume that there is a deployed infrastructure that allows end hosts to set up forwarding state and perform packet replication. While it is unclear how or by whom such an infrastructure will be commercially deployed, and what is the business model behind such an infrastructure, we note that our approach is already useful today in the context of the PlanetLab testbed. In particular, our approach allows application developers that do not have PlanetLab accounts to build their own multicast trees to use PlanetLab for forwarding packets. In contrast, with traditional application-level multicast solutions such as SplitStream and Scribe, application developers need to have direct access to PlanetLab nodes to upload their multicast algorithms.

10 Conclusion

In this paper, we propose a new architecture for overlay multicast where the functionality is carefully split between the infrastructure and end-hosts; the infrastructure implements the data plane, while the end hosts implement the control plane. We can thus leverage the cost and performance benefits of a large-scale shared infrastructure, but at the same time allow applications with diverse needs to evolve independently. We illustrate this principle by using a very simple label-switching like primitive in the infrastructure, and propose distributed algorithms for tree construction, maintenance as well as for loss recovery. We believe that the overall design, and to a lesser extent, the details, presented in the paper is an important step in an attempt to building a flexible, yet efficient multicast architecture in the Internet.

References

[1] http://www.millennium.berkeley.edu.

[2] Fastforward networks. http://www.ffnet.com.

[3] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Towards a More Functional and Secure Network Infrastructure. Technical Report UCB/CSD-03-1242, UCB, 2003.

[4] A. J. Ballardie, P. F. Francis, and J. Crowcroft. Core Based Trees. In *Proc. of ACM SIGCOMM*, 1993.

[5] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application level Multicast. In *Proc. of ACM SIGCOMM*, 2002.

[6] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proc. of ACM SIGCOMM*, 1998.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Multicast in a Cooperative Environment. In *Proc. of SOSP*, 2003.

[8] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An Evaluation of Scalable Application-Level Multicast Built Using Peer-to-Peer Overlay Networks. In *IEEE Infocom*, 2003.

[9] Y. Chawathe, S. McCanne, and E. Brewer. An Architecture for Internet Content Distribution as an Infrastructure Service, 2000.

[10] Y. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc of ACM SIGMETRICS*, 2000.

[11] L. H. M. K. Costa, S. Fdida, and O. C. M. B. Duarte. Hop By Hop Multicast Routing Protocol. In *Proc. of ACM SIGCOMM*, 2001.

[12] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM TOCS*, May 1990.

[13] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming Live Media over Peers. Technical Report 2002-21, Stanford University, 2002.

[14] 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 (pim-sm) : Protocol specification, Jun. 1997. RFC-2117.

[15] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The Case for Separating Routing from Routers. In *Proc. FDNA*, 2004.

[16] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[17] P. Francis. Yoid: Extending the Internet Multicast Architecture, 2000.

[18] Georgia Tech Internet Topology Model. `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`.

[19] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proc. of ACM SPAA*, 2002.

[20] H. Holbrook and D. Cheriton. IP Multicast Channels: EXPRESS Support for Large-scale Single-source Applications. In *Proc. of ACM SIGCOMM*, 1999.

[21] J. Jannotti. *Network Layer Support for Overlay Networks*. PhD thesis, MIT, August 2002.

[22] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of OSDI*, 2000.

[23] S. Keshav and S. Paul. Centralized Multicast. In *Proc. of ICNP*, pages 59–68, 1999.

[24] H. Khosravi and T. Anderson. Requirements for Separation of IP Control and Forwarding. RFC 3654, Nov. 2003.

[25] G.-I. Kwon and J. W. Byers. ROMA: Reliable Overlay Multicast with Loosely Coupled TCP Connections. In *Proc. of IEEE Infocom*, 2004.

[26] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Towards a Secure Indirection Infrastructure. In *Proc. ACM PODC (Brief Announcement)*, 2004.

[27] K. Lakshminarayanan and V. N. Padmanabhan. Some Findings on the Network Performance of Broadband Hosts. In *Proc. of IMC*, 2003.

[28] K. Lakshminarayanan, I. Stoica, and S. Shenker. Routing as a Service. Technical Report UCB-CS-04-1327, UC Berkeley, 2004.

[29] J. Liebeherr and M. Nahas. Application-layer Multicast with Delaunay Triangulations. In *IEEE Globecom*, 2001.

[30] S. McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, UC Berkeley, Dec. 1996. UCB/CSD-96-928.

[31] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *Proc. of ACM NOSSDAV*, 2002.

[32] C. Papadopoulos, G. M. Parulkar, and G. Varghese. An Error Control Scheme for Large-Scale Multicast Applications. In *Symposium on Principles of Distributed Computing*, page 310, 1998.

[33] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An Application Level Multicast Infrastructure, 2001.

[34] Planet Lab. `http://www.planet-lab.org`.

[35] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM*, San Diego, 2001.

[36] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-Level Multicast Using Content-Addressable Networks. In *Networked Group Communication*, pages 14–29, 2001.

[37] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proc. of Middleware*, 2001.

[38] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[39] S. Saroiu, K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of MMCN*, 2002.

[40] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. of ACM SIGCOMM*, 2002.

[41] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, 2001.

[42] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High-speed Prefix Matching. *ACM Transactions on Computer Systems*, 19(4):440–482, Nov. 2001.

[43] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient Multicast Support for Continuous-media Applications. In *Proc. of NOSSDAV*, 1997.

[44] R. Yavatkar, J. Griffoen, and M. Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *ACM Multimedia*, pages 333–344, 1995.

[45] A. Young, J. Chen, Z. Ma, A. Krishnamurthy, L. Peterson, and R. Wang. Overlay Mesh Construction Using Interleaved Spanning Trees. In *Proc. of IEEE Infocom*, 2004.

[46] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proc. of ACM NOSSDAV*, 2001.