Subscriber/Volunteer Trees: Polite, Efficient Overlay Multicast Trees

John Dunagan[*] Nicholas J.A. Harvey[†] Michael B. Jones[*] Marvin Theimer[*]
Alec Wolman[*]

Microsoft Research Technical Report MSR-TR-2004-131

December 2004

Abstract

Application-level multicast trees built using reverse-path forwarding (RPF) on overlay network routing paths are a useful mechanism for scalable information dissemination. One major drawback of this approach is that nodes that are not subscribers to a multicast group can still be required to forward traffic for that group if they happen to lie on an overlay routing path between a subscriber and the group root node. This could serve as a disincentive for nodes to participate in the overlay since they may be required to perform substantial amounts of work for which they receive no immediate benefit. This paper presents Subscriber/Volunteer (SV) trees – a new form of overlay multicast tree that removes this possible obstacle to deployment.

In SV trees, only nodes that are subscribers to a multicast group, or that volunteer to do so, are responsible for forwarding content. SV trees are implemented as RPF trees augmented by adding content forwarding links that route around nodes that are not subscribers or volunteers. SV trees maintain all the benefits of RPF trees, such as scalable delivery and join behavior, while also being polite to non-subscribers. The RPF tree, used for node joins, and the forwarding tree, used for content delivery, are kept consistent even in the face of nodes leaving and joining the multicast tree and/or overlay. Finally, our results show that SV trees deliver content more efficiently than RPF trees, since unnecessary network hops have been grafted out of the delivery trees.

1 Introduction

Scalable application-level multicast systems allow information to be efficiently disseminated to arbitrary numbers of subscribers [6, 20, 2, 15, 13]. These systems typically organize participants into one or more multicast trees, with each participant receiving information from its parent in the tree and forwarding it on to its children. Thus, the amount of work required of each participant to multicast a piece of content is bounded and is proportional to the branching factor of the constructed multicast tree times the size of the content.

Scalable overlay networks [23, 17, 19, 27, 12] provide one means of constructing application-level multicast trees, through the use of standard reverse-path forwarding (RPF) techniques [7, 8]. Because overlay routing tables are often optimized along some metric (e.g., latency), a key benefit of this technique is that the RPF trees are similarly optimized. Systems such as Scribe [20] employ this method and have demonstrated its effectiveness. Reverse-path forwarding has the advantage that join and leave operations are scalable; no central point of coordination for tree membership is employed. This is in contrast to systems like CoopNet [15], where membership operations could become a bottleneck.

To join an RPF overlay multicast tree a node routes a *join* message towards the multicast tree

[*]Microsoft Research, Microsoft Corporation, Redmond, WA 98052, {jdunagan, mbj, theimer, alecw}@microsoft.com

[†]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, nickh@mit.edu

root. Each node along the overlay routing path taken by the *join* message remembers the node from which the message arrived and the ID of the multicast group. When the *join* message reaches a node that is already part of the multicast tree for the group, the join has succeeded. When content is sent from the root it is routed down the tree along routes that are the reverse of the paths taken by the *join* messages.

Despite their usefulness, RPF overlay multicast trees face a potentially significant obstacle to their adoption: overlay nodes that are not interested in the content being sent on a multicast tree are required to bear the costs of forwarding that content if they fall along an overlay route from a subscriber to the root. Figure 1 shows an RPF overlay multicast tree with three interior nodes that are not subscribers, but which nonetheless must forward multicast traffic for the tree. If the amount of traffic is significant the node's owners may very well ask themselves "Why should I be a member of the overlay when it costs me more than I get from it?". Seemingly simple approaches to addressing this problem, such as building a separate overlay for each group, introduce other concerns such as high network load due to the maintenance traffic for multiple overlays [4].

Subscriber/Volunteer trees (SV trees) are a new kind of overlay multicast tree that eliminates this problem. In SV trees, non-subscribers are never required to forward multicast traffic. Unless a non-subscribing node explicitly *volunteers* to forward traffic for a group, the multicast traffic for a group is routed around that node, even if it is on an overlay routing path between a subscriber and the root. Only *subscribers* and *volunteers* forward traffic in SV trees – hence the name.

An SV tree is derived from the RPF tree for a given group root and set of subscribers. Figure 2 shows an RPF tree and an SV tree derived from it. In the SV tree, traffic is routed around the two non-participating nodes $N1$ and $N2$ that would have been required to forward traffic in the RPF tree; they incur no forwarding costs. The non-subscribing volunteer node $V1$ does forward traffic for the SV tree because it indicated its willingness to do so.

Two benefits of SV trees are evident from this example. First, they are ***polite***: nodes not interested in the content of a multicast group bear no costs of forwarding traffic for that group. Second, they provide ***efficient content delivery***: the content forwarding paths for SV trees may be shorter than the paths for the RPF trees from which they are derived. For instance, in the trees in Figure 2, the RPF forwarding route between the root R and subscriber $S1$ takes four hops, whereas the SV forwarding route between these nodes takes only two hops. Both benefits are important to any practical deployment of systems employing application-level multicast groups.

A challenge for a different aspect of efficient content delivery is this: Many overlay networks optimize routes based on some metric (e.g., latency or bandwidth or some combination of the two). Thus, one benefit of RPF trees built using them is that the content forwarding routes follow the reversed overlay routes, and are thus optimized. A key challenge is constructing SV trees to be similarly optimized.

There are several additional important challenges involved in building and maintaining SV trees. One is ***scalable content delivery***: ensuring that the content delivery load on each SV tree node is bounded, just as it is for RPF trees. A second is ***scalable membership operations***: enabling subscribers to efficiently and scalably locate and join the SV tree even though the overlay route between the subscriber and the root may not pass through any other participating SV tree members – plus ensuring that when tree or overlay members leave that these goals are still met.

Note that in SV trees the underlying RPF tree continues to be used to support join and leave operations. Thus, non-subscribers will still incur a slight amount of overlay traffic for SV tree membership operations – an amount that is bounded by the branching factor of the tree times the rate of membership churn. This membership traffic will be substantially less than the content traffic for any active multicast group.

A final key challenge is ***correctness***: ensuring, as overlay nodes and group members come and go, that the SV tree continues working properly. Given the possibility of routing loops and the need to keep the RPF and content forwarding trees in sync, ensuring correctness is non-trivial. Indeed, ensuring correctness was the major design challenge for SV trees. Many of our design choices were driven by

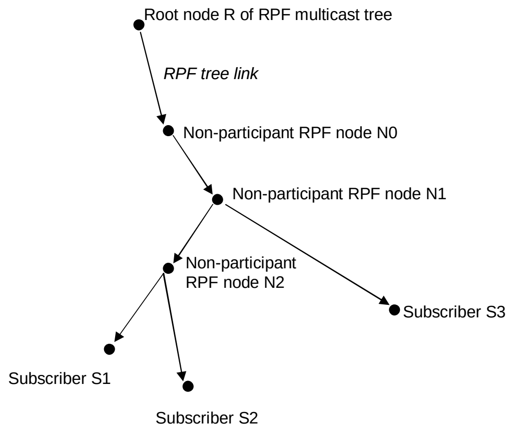

Root node R of RPF multicast tree
RPF tree link
Non-participant RPF node N0
Non-participant RPF node N1
Non-participant RPF node N2
Subscriber S3
Subscriber S1
Subscriber S2

Figure 1: *Reverse-path forwarding (RPF) tree containing three non-participant nodes.*

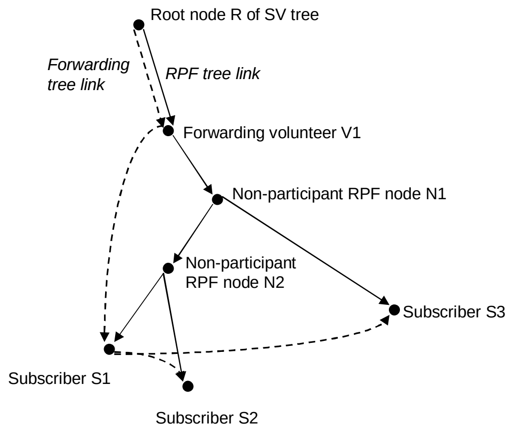


Figure 2: *SV tree and corresponding RPF tree containing two non-participant nodes. Solid lines represent RPF tree links and dashed lines represent SV forwarding tree links.*

the need to be able to reason about whether the system would operate correctly in all circumstances, rather than by other traditional concerns such as absolute efficiency, etc.

We present results from an implementation of SV trees built using the SkipNet [12] overlay network. Our implementation runs both in a network simulator and on a live network of machines. Our results demonstrate that SV trees meet all the design challenges outlined above.

2 Related Work

Reverse Path Forwarding is a well-established technique for building multicast forwarding trees over a network providing routing between nodes. It forms the basis of several variations of Internet multicast systems [7, 8].

Overlay networks [23, 17, 19, 27, 12] provide a means for organizing network nodes into a communication mesh providing content-based routing of messages between nodes. Two classes of mechanisms are used to implement application-level multicast trees for overlay networks: forming a separate overlay for every multicast group, as is done by the CAN multicast system [18], and using Reverse Path Forwarding to build a multicast tree along the overlay routes, as is done by Scribe [20]. Castro et al. [4] showed that the single-overlay approach offers the significant advantages of decreased network load and faster group creation. However, they also point out the lack of politeness in the single overlay approach.

SV trees retain the benefits of Scribe trees, while also making them *polite* – meaning that they impose no forwarding load on overlay nodes not interested in the content of a multicast tree. Like Scribe trees, SV trees are more efficient than simple RPF trees because both systems perform a kind of "path compression". Scribe trees bypass non-subscribing links that only forward to a single node. SV trees bypass all non-subscribing nodes that are not volunteers.

SV tree are designed to scale to arbitrarily large numbers of subscribers. One consequence of this decision is that they have no coordinating node that members must contact when joining because in any sufficiently large system such a node would become overloaded. This is one of the factors that distinguishes SV trees from systems such as End System Multicast [6], including its live Internet deployment to broadcast conferences [13], NICE [1], CoopNet [15], ScatterCast [5], and ALMI [16].

Other systems performing application-level content delivery are organized differently, with different goals. For instance, the XML content-based routing system described in [22], differs from SV trees in at least four significant ways: First, it filters content, rather than delivering the full contents of the multicast group to all subscribers. Second, it distinguishes between clients and servers, whereas SV trees have no clients that are not potentially also content servers. Third, it implements redundant delivery paths as an integral part of the service – something that SV trees do not do. And finally, it relies on a manually constructed mesh of nodes, whereas SV trees use the automatically constructed mesh provided by an overlay network. Many such examples of application-level content distribution systems with different goals and characteristics are possible.

Numerous other systems for application-level multicast have been proposed reflecting these different goals. For instance, SplitStream [2] is scalable, but requires the participation of non-subscriber nodes to forward content. The Overcast [14] system also uses multicast for content distribution, however unlike SV trees and many of the systems above, it relies on having a set of dedicated servers for content distribution. SV trees also allow for the possibility of dedicated servers by supporting volunteer nodes, but do not require them. Yoid [10] defines a distributed tree-building protocol among end-hosts resulting in a source-specific tree per group not based on any overlay.

Finally, it is worth pointing out that SV trees are correct by construction, even in the face of changes to overlay and tree membership. Routing loops can not be constructed and local failures are discovered in bounded time and repaired locally. Other systems, such as Scribe (with the bottleneck remover optimization enabled [3]), have algorithms to detect inconsistencies such as routing loops and use a different algorithm that selects nodes to rejoin the tree using a randomized route in an attempt to repair detected inconsistencies. In contrast, SV trees have no such need for a separate inconsistency repair algorithm since SV trees, by construction, can never enter an inconsistent state.

3 Subscriber/Volunteer Trees

Subscriber/Volunteer trees are designed to simultaneously meet several goals: *politeness*, *efficient content delivery*, *scalable content delivery*, *scalable membership operations*, and *correctness*. This section presents the design of SV trees and describes how each of these goals is met.

Subscriber/Volunteer trees are derived from reverse-path forwarding trees, and in fact, maintain RPF trees as part of their implementation. Figure 2 shows both the RPF tree and the content forwarding tree employed by an SV tree.

In the following description three related data structures will be referenced by the following names:

- **SV Tree**: All the data structures used to implement a Subscriber/Volunteer tree. Includes the RPF tree and the forwarding tree.

- **Forwarding Tree**: The set of links between nodes used for forwarding content from the root to the subscribers.

- **RPF Tree**: The reverse-path forwarding tree used by an SV tree to help implement operations such as subscriber joins.

There are four main kinds of operations that any application-level multicast tree must support. Their implementations in SV trees are outlined below:

Content Forwarding: Content is sent down the forwarding tree links from the root to the leaves.

Node Joins: As in standard RPF trees, nodes route a *join* message towards the multicast tree root. Each node traversed becomes part of the SV tree's RPF tree. After reaching an RPF tree node an additional step is required to make a forwarding tree link from an existing forwarding tree member to the joining node. Either (1) the node reached by the join message was also a forwarding tree member, in which case the link is made directly from that node, or (2) the node reached is a non-participant in the forwarding tree, in which case the link must be made from a node that is part of the forwarding tree. The precise means of finding such a node are described in Section 3.1, but the core idea is this: if a node along the RPF path doesn't wish to forward traffic it instead remembers which nodes were used to route content traffic around it, and has one of them establish the forwarding link to the joining node.

For example, in Figure 2, when node $S3$ joined, it first routed a *join* message towards R. This message reached $N1$, which was already a member of the RPF tree, but which is a non-participant in the forwarding tree. $N1$ remembered that nodes $V1$ and $S1$ are used to forward content traffic around it and contacts one of them to establish a forwarding tree link to $S3$. In this case, $S1$ is contacted and it completes the forwarding tree by making a link to $S3$.

Node Departures: There are several ways that nodes that are participating in an SV tree can leave the tree. They may stop subscribing to the multicast tree but remain in the overlay, they may voluntarily leave the multicast tree and the overlay, or they may crash or become unable to communicate with the overlay. An SV tree handles all of these cases in the same way: It treats it as if the departing node had failed, invoking the tree repair algorithm.

Tree Repairs: An SV tree node remembers all the SV nodes that affect its relationships to other SV nodes. This includes its parent in the RPF tree, its parent in the forwarding tree (if in the forwarding tree), any nodes that forward content around it (if not in the forwarding tree), and which subscribers or volunteers caused it to be a member of the SV tree. Furthermore, we employ liveness checking functionality, which is described in Section 3.2.1, to make sure that these nodes are still functioning correctly. If any of these related nodes fail or withdraw from the SV tree or overlay, the node liveness checking code invokes the SV tree repair algorithms (see Section 3.2) to restore the SV tree invariants.

When nodes fail or withdraw only the local region of the SV tree around them must be repaired. This is accomplished by having the subscribers or volunteers immediately affected by the broken portion of the tree rejoin the SV tree. Note that this is a local operation and doesn't cause all tree nodes below the break to have to rejoin. For instance, nodes below a rejoining node in the forwarding tree will continue to have content forwarded to them after their ancestor rejoins with no work upon their part. Likewise, changes in the middle or near the root of the tree don't cause the subtrees below them to be rebuilt. Local changes in the set of participating nodes cause only local tree reconfigurations. This *locality* is key to achieving our *scalability* goals. The tree repair algorithm is discussed in Section 3.2.

As nodes come and go from the overlay, overlay routes can change. Unless care is taken, this can result in loops in the content forwarding routes (as described in [3]). SV trees employ algorithms that prevent routing loops from forming, such as having each node maintain a partial order among its RPF tree children of who forwards to who, as described in Section 3.2.2. Maintaining these invariants is key to achieving our *correctness* goals.

3.1 Join

A node initiates a subscription to an SV tree by routing a *join* message towards the tree root (just as is done for RPF trees). Depending upon what kinds of nodes are traversed until a member of the SV tree's RPF tree is reached, there are several different cases.

3.1.1 Simple Join Cases

In the simplest case, the message reaches another member of the forwarding tree on its first hop. In this case, a new forwarding tree link is established from the node reached to the subscriber, an RPF tree link is also established over the same hop, and the join is complete (except for establishing a failure handling group for the subscriber, which is described in Section 3.2).

In the next simplest case, the *join* message may take several overlay hops to reach an existing RPF tree member and the node reached is also a forwarding tree member. In this case, all nodes traversed are marked as members of the RPF tree and a forwarding tree link is established directly from the node reached to the subscriber, bypassing the non-subscribing nodes traversed.

Figures 3 and 4 depict a situation where a subscription request is satisfied by a join request that is forwarded up the RPF tree until a forwarding tree node is reached. The new subscriber is attached to the forwarding tree node reached when searching up the tree. Figure 3 shows the join messages sent. Figure 4 shows the SV tree after the join has completed.

As a special case, should a node that is traversed be a volunteer node, then multiple forwarding hops will be established so that it is included in the forwarding tree. This case is handled by having the volunteer node satisfy the initial join request. The volunteer then joins the tree itself using its own join request. For instance, in Figure 2, when $S1$ joined its *join* request was satisfied at the volunteer node $V1$, which itself then joined, connecting to the root node R.

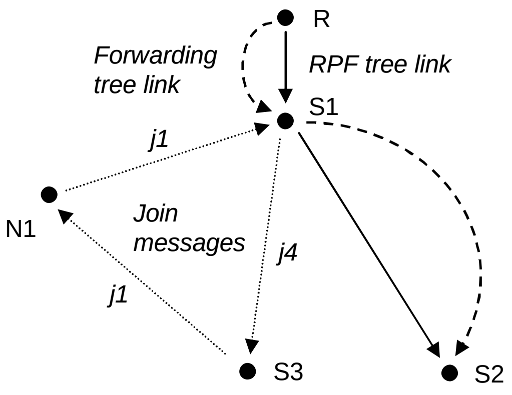


Figure 3: *Subscriber $S3$ joins an SV tree. $S3$ routes join message $j1$ towards root, reaching subscriber $S1$ via non-subscriber $N1$. $S1$ sends join candidate message $j4$ to $S3$. This tells $S3$ that a forwarding tree link can be established from $S1$.*

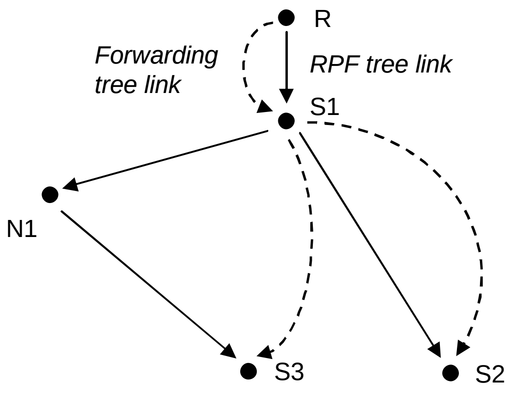


Figure 4: *Subscriber $S3$ has joined the SV tree. A new forwarding tree link has been established from $S1$ to $S3$. New RPF tree links have been established from $S1$ to $N1$ and $N1$ to $S3$.*

3.1.2 Joins that Search the RPF Tree

The next situation is if the *join* message reaches an RPF tree member that is not a forwarding tree member. When this occurs, it is the responsibility of the node reached to locate an appropriate forwarding tree member for the subscriber to attach itself to. It does so by relying on the fact that its membership in the RPF tree means that (1) there is a forwarding tree member somewhere above it in the RPF tree (possibly the tree root node) and (2) there are one or more forwarding tree members below it in every branch of the RPF tree. This means a forwarding tree member could be found either above or below the node reached by the *join* message to attach it to the forwarding tree.

If join requests were always forwarded up the RPF tree until a forwarding tree node was reached, this would violate our scalability goals as it could cause the branching factor for nodes higher up in the tree to grow without bound. For example, all subscribers whose overlay routing paths to the root don't pass through other subscribers (or volunteers) would be joined at the root – clearly a non-scalable situation. Thus, we adopt the rule that each RPF node forwards at most one join request up the RPF tree before forwarding all others down.

Figures 5 and 6 depict the case where a subscription request is satisfied by a join request that initially reaches an RPF tree node that is not in the forwarding tree. This node sends the request down the RPF tree until it reaches one or more forwarding tree nodes, which respond to the subscribing node. Such a join occurs in several phases.

First, the initial *join* message $j1$ is routed from the subscriber $S5$ towards the root until the RPF tree node $N1$ is reached. $N1$ decides to search down the RPF tree for a forwarding tree node to attach $S5$ to.

Second, $N1$ sends *join probe* messages $j2$ to *all* its RPF tree children. This replication allows multiple choices for forwarding tree attachment points to be considered.

Third, each of $N1$'s children forwards the received *join probe* messages $j3$ to *only one* of their RPF tree children, which is randomly chosen among them. Having the children not send the messages to all of their children prevents the message from flooding the entire tree below $N1$. Likewise, any further forwarding, as by $N4$, is to only one child.

Fourth, once the *join probe* messages reach forwarding tree nodes, in this case $S2$ and $S3$, they are forwarded directly to the subscribing node $S5$ as *join candidate* messages $j4$.

Finally, the subscriber $S5$ chooses among the *candidate join* messages $j4$ received, and picks the sender of one of them as its attachment point in the forwarding tree. In our implementation we choose the one that arrives first, helping to build a latency-optimized forwarding tree. Figure 6 depicts the case in which the message from $S2$ to $S5$ arrived first

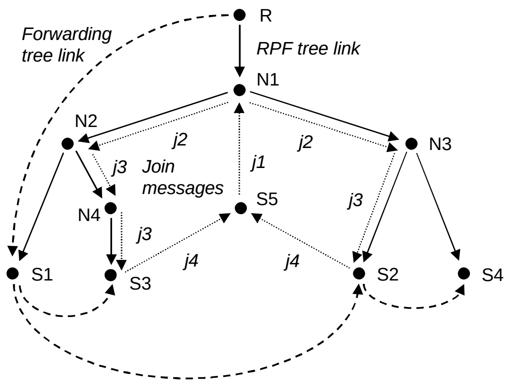


Figure 5: *Subscriber $S5$ joins the SV tree. $S5$ routes join message $j1$ towards root, reaching $N1$. $N1$ sends join probe messages $j2$ to its RPF tree children $N2$ and $N3$. $N2$ and $N3$ forward the join probe messages $j3$ to one of their children, which do likewise, eventually reaching forwarding tree members $S3$ and $S2$. $S3$ and $S2$ send join candidate messages $j4$ to $S5$. $S5$ chooses the first one of these to arrive and establishes a forwarding tree link from its sender.*

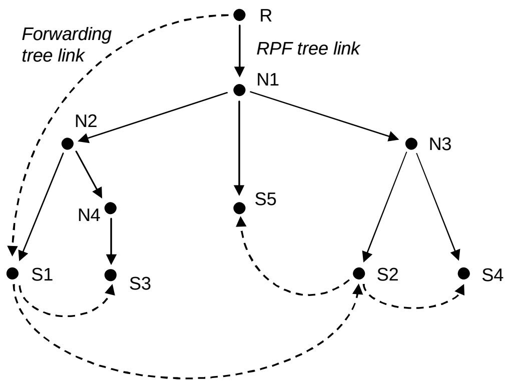


Figure 6: *Subscriber $S5$ has joined the SV tree. A new forwarding tree link has been established from $S2$ to $S5$. A new RPF tree link has been established from $N1$ to $S5$.*

and a forwarding tree link was established from $S2$ to $S5$.

3.2 Failure Handling and Detection

SV trees are designed to operate in highly dynamic environments where nodes may fail or may join or leave the overlay and multicast groups. Thus, it is critical that SV trees are able to dynamically reconfigure themselves to adapt to these changes. SV trees use ***failure handling groups*** as a tool for accomplishing this goal. The idea behind failure handling groups is to notify all nodes related to a forwarding tree link whenever any of these nodes fail or leave the SV tree, triggering a local tree repair operation.

Each of these groups contain exactly those nodes that were involved in the communication path to establish the forwarding tree link. For instance, in figure Figure 6, the failure handling group for the forwarding link from $S2$ to $S5$ would have as its members $S5$, $N1$, $N3$, and $S2$ – the members of the join communication path in Figure 5 that established it. If any of these members fail that causes all the other members to discard their state associated with this forwarding tree link and for $S5$ to rejoin the group (assuming it wasn't the node that failed).

3.2.1 FUSE Failure Detection

Failure handling groups are implemented via a system called FUSE [9], which is a separate body of work only summarized here. The design and implementation of FUSE was motivated by the needs of SV trees to reliably perform coordinated failure handling when any of a set of related nodes fails.

FUSE implements a lightweight failure notification service among groups of nodes organized into failure handling groups called "FUSE groups". FUSE guarantees that failure notifications never fail: Whenever a failure notification is triggered, all live members of the FUSE group will be notified within a bounded period of time, irrespective of any node or communication failures. FUSE implements this guarantee through active, efficient monitoring of the liveness of FUSE group members.

FUSE group notifications can be triggered in two ways: either explicitly by group members, for instance, if one of them is voluntarily leaving the SV tree, or implicitly by the FUSE failure detection service, if one of the nodes has crashed or becomes unable to communicate with another node.

Using FUSE greatly simplifies the implementation of some aspects of SV trees because we are guaranteed that all the nodes that play a role in a forwarding tree link will be informed if any of them die, withdraw, or become unable to communicate with the others. Thus, all nodes will clean up the state associated with any failed portions of the SV tree and initiate recovery actions in a timely fashion. This allows us to use a single failure detection and handling strategy for all kinds of failures, rather than different ones for different kinds. Using FUSE made it significantly easier to ensure the correctness of SV trees.

3.2.2 Preventing Routing Loops

Consider the SV tree depicted in Figure 7. If subscriber $S2$ dies or withdraws from the SV tree the temporary result will be the disconnected SV tree in Figure 8. Node $S3$ will have been in a FUSE group containing itself, $N1$, and $S2$. $S2$'s failure will notify the remaining members of the group, causing $S3$ to initiate a join to reattach itself and its children to the SV tree.

Naïvely, one might think that this would entail sending a *join* to $N1$, $N1$ sending *join probe* messages to its other RPF tree children $S1$ and $S4$, these children sending *join candidate* messages to $S3$, and $S3$ choosing whichever of them arrived first. But suppose that a *join candidate* message was sent from $S4$ to $S3$ and $S3$ selected it as its forwarding tree attachment point. There would be a big problem: a forwarding tree loop where $S3$ forwards to $S4$ and $S4$ forwards to $S3$ and nobody forwards to either of them!

SV trees prevent this by maintaining a ***partial order*** among children of non-forwarding-tree RPF tree nodes, where the order is determined by which children are forwarding to which other children. In this example, $N1$ remembers, among other things, that $S3$ forwards to $S4$. Therefore when $S3$ tries to rejoin, the *join probe* message is not sent down the RPF tree link to $S4$ because it is after $S3$'s link in the partial order. Consequently, in this example, $N1$ sends a *join probe* message only to $S1$, resulting in $S3$ rejoining the forwarding tree with a connection to $S1$.

Other systems, such as Scribe, have handled this problem in more ad-hoc manner by detecting loops when they form, breaking them, and trying another join using a "randomized route" that hopefully will not result in another routing loop [3]. They detect loops by storing the entire overlay path to the root in each Scribe tree node and checking whether the partial path from a joining node contains one of those nodes already. (Note that loops do not form in Scribe unless the bottleneck remover optimization is enabled.)

In contrast, SV trees can not form routing loops and so separate detection and stochastic retry mechanisms are not necessary. They are correct by construction, even in the face of changes.

3.2.3 Cleanup Actions Upon Failures

Recall that each node involved in establishing a forwarding tree link is made a member of a failure handling group associated with that link. When any of these nodes dies (or removes itself from the SV tree) all the members of the group are notified and perform compensating actions. Depending upon the type of node, different actions are taken.

The subscriber node will try to resubscribe by routing a new join message. The source node for the forwarding tree link will delete the link. The RPF nodes involved will clean up state associated with establishing that link. Two kinds of such state are maintained by RPF nodes.

First, RPF nodes remember whether or not they have yet forwarded a join request up the RPF tree for that group, and they remember which request it was. This allows the invariant that each RPF node forwards at most one join request up the RPF tree before forwarding all others down, as explained in Section 3.1.2. When the forwarding tree link established by a request sent up the RPF tree is torn down, one of the RPF tree node's clean-up actions is to clear the state saying it has already forwarded a request up the tree. Thus, it can forward the next join request that it receives up the tree. For instance, in Figure 8, if $S1$ failed then $N1$ would clear its "sent up" state, meaning that the next join request to reach $N1$ would be forwarded up the tree, rather than sent down to its RPF tree children.

Second, RPF nodes remember the partial order among its RPF tree children of which branches are

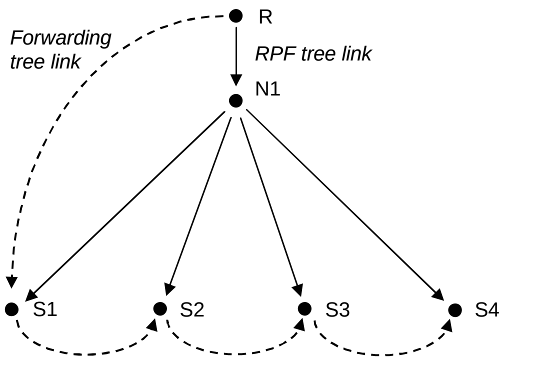


Figure 7: *SV tree before node $S2$ fails.*

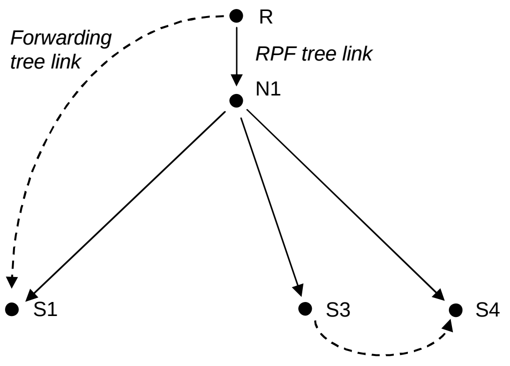


Figure 8: *SV tree after node $S2$ fails. Note that nodes $S3$ and $S4$ are temporarily disconnected from the forwarding tree, causing $S3$ to need to rejoin.*

forwarding to which others, as described in Section 3.2.2. When one of these forwarding relationships is torn down, this portion of the partial order is also removed.

3.2.4 The Role of Retries

Join requests may fail because of temporary inconsistencies in the SV tree state encountered during the join. For instance, if a node traversed by one of the messages dies before the join completes, this will be discovered when the join process attempts to establish a FUSE group containing all the nodes involved in the join. This causes the join to fail.

Because FUSE groups reliably notify all their members of the failure of the group within bounded time such inconsistencies will not persist. Thus, retries of failed joins will eventually be effective, and are employed by the SV tree implementation.

3.3 Scalability

One of the most important properties of RPF trees is that they are scalable – meaning that the load imposed on each tree participant is independent of the size of the tree. In practice, there are two kinds of loads: the load caused by nodes joining the tree and the load caused by content being forwarded through the tree.

SV trees maintain these same scalability properties in two different ways. First, SV trees ensure that the join load is scalable by keeping around their underlying RPF trees and using them to intercept join traffic for new subscribers. Essentially, the SV join operations are scalable because the underlying RPF join operations are scalable, as described in more detail in Section 3.1. To the first approximation, both loads are proportional to the product of the routing "fan-in" at that node – the number of distinct overlay routes that pass through that node – times the rate of overlay churn – the rate at which nodes enter and leave the overlay. As long as the number of overlay routes through a node is bounded, which is true for overlays such as Chord [23], Pastry [19], and SkipNet [12], this effectively means that the join load on a node is proportional only to the rate of overlay churn.

Second, SV trees ensure that the content forwarding load is scalable by ensuring that each node in the SV tree will typically only have to forward to at most as many other nodes as they would in a fully-populated RPF tree, which is bounded by the overlay routing "fan-in" at that node. Since the fan-in is bounded for the overlays of interest, this means that for the same reasons that the content forwarding load is scalable for RPF trees, it is also scalable for SV trees. Each piece of content will typically only need to be forwarded to at most *fan-in* children.

Since, as in Figure 5, *join probe* messages are sent to randomly selected descendents of the RPF tree's children, it is theoretically possible for all these random choices to come out the same. Should this unlikely event occur, it would mean that the number of forwarding tree nodes attached to a leaf node could rise to be on the order of $\log^2 N$, where N is the number of tree members, rather than $\log N$, which is the expected value. Should a particular node's branching factor be too large, load balancing operations could be used to rebalance local regions of the tree. We have not implemented such operations because we have found no need for them in our usage.

4 Implementation and Methodology

We implemented SV trees on top of the SkipNet [12] overlay network and concurrently built the FUSE [9] implementation for SV trees to use. Our experiments were performed in two different environments: a scalable discrete event simulator, and a live implementation with 400 virtual nodes running on a cluster of 40 workstations. For the cluster experiments, our router uses Modelnet [25] to emulate wide-area Internet-like network characteristics. Each virtual node in the live system is a separate process running an instance of our implementation. In order to emulate nodes running on physically separated machines, there is no explicit sharing of state between these processes, and all communication between processes is forced to pass through the Modelnet core. Our SkipNet, FUSE, and SV tree implementations running on the live system and in the simulator use identical code, except for the base messaging layer.

Our SV tree implementation contains approximately 4000 lines of C# code, including comments and whitespace. This compares to approximately 2500 lines for FUSE and 8000 for SkipNet.

4.1 Parameters

We configured the SkipNet overlay to employ a 60 second ping period, a base of size 8, a leaf set of size 16, and R-Table routes. More details on the implications of these parameter choices can be found here [11]. For a 400 node overlay, this yielded an average of 32.3 distinct neighbors per node.

Both our live and simulator experiments were run on a Mercator topology [24] with 102,639 nodes and 142,303 links. Each node is assigned to one of 2,662 Autonomous Systems (ASs). There are 4,851 links between ASs. The Mercator topology does not include latency or bandwidth values, and therefore we were forced to assign such values. To determine link latency and bandwidth, we assigned 97% of links to be OC3 and 3% to be T3. For each OC3 link, we assigned the link latency uniformly between 10 and 40 milliseconds and a bandwidth of 155 Mbps. For each T3 link, we assigned the link latency uniformly between 300 and 500 milliseconds and a bandwidth of 45 Mbps. This led to round-trip latencies with a median value of 130 milliseconds and a significant heavy-tail; in Section 5.1 we present the CDF of latencies in the context of a simple RPC test. Paths traversing one or more T3 links are in the heavy-tail.

We emulated this topology on our cluster with 400 virtual nodes. We also used this topology in our simulator, where we ran experiments with up to 16,000 nodes, to model how our system would scale to a much larger deployment. The simulator used the same latency values, but does not model bandwidth constraints.

5 Results

The two principal design goals of SV trees are that they are *polite* and that they provide *efficient content delivery*. SV Trees are polite by construction – non-subscriber nodes never have to perform content forwarding. The experimental results in Section 5.2 show that SV trees do indeed provide efficient content delivery, and that both latencies and link stresses are better than in the RPF overlay multicast approach.

In implementing SV trees, we found that maintaining correctness in the face of failure and other stresses greatly influenced our design. The experiments we present measure our success in meeting this goal as well: our results in Sections 5.3 and 5.4 show that SV Trees are correctly constructed even during flash crowds, and that persistent disconnections are prevented even during periods of high churn.

Finally, we present results in Section 5.5 from running a flash crowd experiment on our live cluster. Our SV tree implementation running on the cluster shows the same ability to support large numbers of simultaneous joins as the simulator.

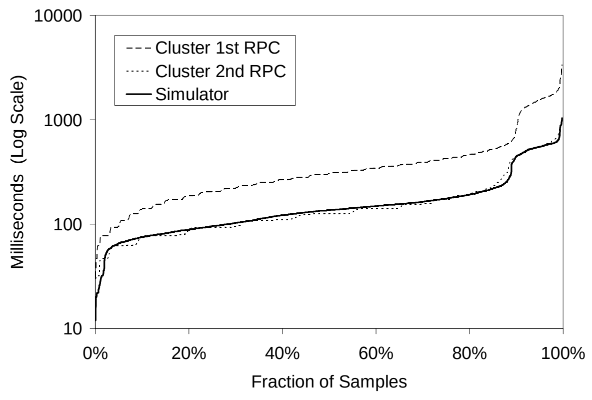


Figure 9: *RPC Latencies on simulator and cluster.*

5.1 Calibration of Simulator and Modelnet

We used an experiment that performed RPC message exchanges between randomly chosen nodes on a 400-node overlay network to calibrate the wide-area network topology model used in our experiments and to make sure that results obtained through simulation were comparable to those obtained through running on the live cluster with Modelnet.

Towards that end, we performed a simple RPC test of 2400 RPCs on both our cluster and our simulator. Figure 9 shows a Cumulative Distribution Function (CDF) of the RPC times measured for three sets of RPCs: those obtained in the simulator, and two kinds of RPC times obtained on the cluster. Because the cluster code caches TCP connections between pairs of nodes, the first communication between a pair of nodes takes longer than subsequent communications, due to the additional time required for connection establishment. Our experiment performs two back-to-back RPCs between pairs of nodes on the cluster and reports the durations of both the first RPC, which is likely to incur connection setup overhead, and the second one, which will not.

As can be seen in Figure 9, the values for the second RPC on the cluster closely track those for the simulator. This gives us confidence that both the simulator and Modelnet are faithfully emulating the chosen Mercator wide-area network topology.

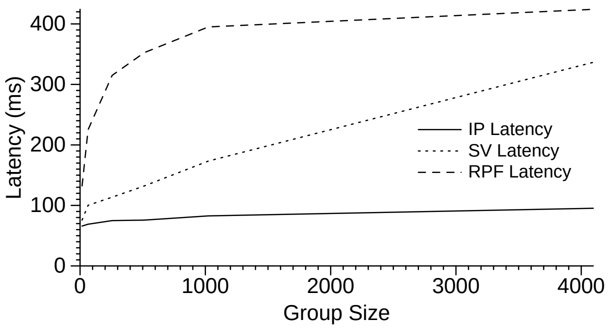


Figure 10: *Mean latency to deliver content over SV, RPF, and IP-Multicast Trees for different group sizes.*

5.2 Efficiency

Figure 10 shows the latency of content delivery for three different distribution architectures: IP multicast, overlay reverse-path forwarding (RPF), and SV trees. Each data point is an average over 5 runs, each with a differently chosen random seed. The number of overlay participants was kept constant at 16,000 nodes, while the number of subscribers varied over {16, 64, 256, 512, 1024, 4096}.

The first observation we make is that SV trees noticeably out-perform RPF trees when there are relatively few subscribers. In particular, RPF trees suffer approximately a factor of five penalty relative to IP multicast for even small group sizes, while the SV tree penalty is quite small for small group sizes. As the number of subscribers grows, this difference diminishes. This is not surprising; the relative performance gain of SV trees comes from routing content directly between subscribers, bypassing intermediate non-subscriber nodes. The number of opportunities to bypass intermediate nodes is greatest when the number of subscribers is a small fraction of all nodes in the overlay.

It is also worth comparing SV trees to an overlay multicast design such as Scribe [3] that is solely focused on efficiency. Published evaluations of Scribe use the Pastry overlay running on the GT-ITM topology [26], whereas our evaluation of SV Trees uses the SkipNet overlay running on the Mercator topology. In spite of these differences, we see that the published performance numbers are comparable: for a 4000 node group, the delay penalty of SV relative to IP multicast is approximately 3.5, while for Scribe the relative delay penalty is 2.5.

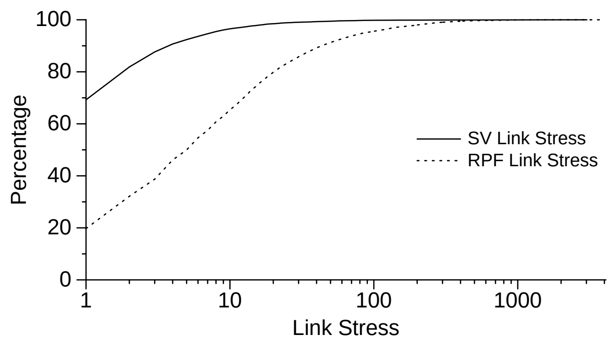


Figure 11: *Cumulative distribution of Link Stress for SV and RPF Trees with 4096 subscribers. Note that the x-axis uses a log scale.*

Figure 11 shows a CDF of link stress for both RPF trees and SV trees. The number of overlay participants is again held constant at 16,000 nodes, and the number of subscribers was set to 4096. We concatenated together the link stresses obtained over 5 runs of content forwarding on different trees, and we used this to construct a single CDF for both RPF and SV trees.

Figure 11 shows that SV trees incur link stresses that are noticeably better than straightforward RPF trees. In particular, SV trees cause 90% of links to incur a link stress of 3 or less, while for RPF Trees the 90% link stress is approximately 50. This is not surprising, as SV trees reduce the number of intermediate forwarding hops.

The absolute maximum link stresses incurred by SV trees are comparable to published results for Scribe [3]. This is good; SV trees meet the goal of politeness without incurring a link stress penalty relative to previous work that has focused on efficiency.

5.3 Resilience to Flash Crowds

Figure 12 shows the subscription latency during flash crowds. We took 5 different runs from our simulator. For each run, we constructed both 1024 and 4096 subscriber trees. The x-axis of the graph is the time between when a node began subscribing and when that same node received content being published on the tree. The y-axis is a CDF over the joint distribution of all 5 runs at the two different sizes. We configured the root of the tree to publish messages with sufficient frequency (once every 10

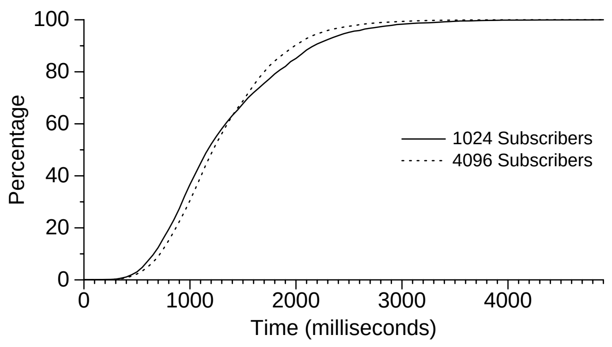


Figure 12: *Cumulative distribution of Subscription Latency for SV trees during a flash crowd, with 1024 and 4096 subscribers.*

milliseconds) that the pipeline was always full of messages.

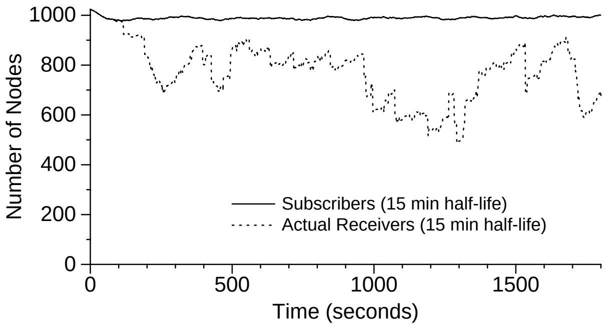
Scalable overlays are good at distributing the message traffic during simultaneous joins [3]. However, this does come at a cost: a subscribing node might have a path to the root involving many hops, and content might not arrive until every intermediate node had finished subscribing. During a flash crowd, a number of additional events might occur that could delay nodes from receiving content. For example, race conditions might cause subscription retries at some fraction of the nodes.

Our experiments show that the SV tree design handles flash crowds well. For both distributions of subscribing nodes, over 3/4 are receiving content within 2 seconds of starting to subscribe, and all subscribers are receiving content by 5 seconds from the beginning of the flash crowd.

Figure 12 also shows that SV trees meet their stated design goal of scalable membership operations; the SV tree performance does not degrade as the number of subscribers increases. The flash crowd of 4096 nodes shows almost no slowdown relative to the flash crowd of 1024 nodes.

5.4 Resilience to Churn

Handling churn is difficult for application-layer multicast systems. Because new nodes are always failing, large portions of the tree are frequently cut off by the failure of nodes near the root. Additionally, SV trees do not implement queueing of undeliverable messages (instead they are dropped). This is a standard approach in application domains



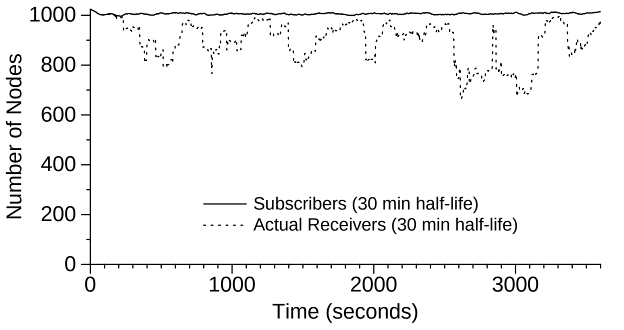


Figure 13: *Number of nodes that receive messages during churn, for a group that would have 1024 subscribers with no failures. The churn rate results in a 15 minute half-life (on the left) or a 30 minute half-life (on the right).*

such as audio and video streaming. For example, CoopNet, an application-layer multicast system for video, employs forward error correction, not retransmission. However, when messages are delivered, they arrive with latencies like those in Figure 10.

The experiment in Figure 13 were obtained by running two churn experiments. In both cases we ran the experiment on a 4000 node overlay; at any given point in time, exactly half the nodes were failed. The first experiment uses a half-life of 15 minutes, and the second experiment uses a half-life of 30 minutes. A half-life of 15 minutes is shorter than has been observed in any peer-to-peer client population in the wild [21], and so it represents a pessimistic worst case. In both cases, slightly less than 1024 nodes were attempting to subscribe to the SV tree at any given time. The exact number of subscribers as a function of time is shown in the graph. The experiment spanned a stretch of time sufficient for every node in the system to have changed in status once, either by joining the overlay or failing.

We choose to make all failures in this experiment be silent failures because this is the most difficult case for our algorithm to handle. Silent failures leave holes in other nodes' routing tables that they may not discover until at least one overlay-layer timeout – one minute in our configurations.

Figure 13 shows that for a 15 minute half life, more than half the nodes are receiving content at any given time. For a 30 minute half life, the results are even stronger. This shows that the SV tree repair is managing to keep up with both churn rates; persistent disconnections are not occurring, as these would have manifested themselves through a steady downward trend in the number of subscribers receiving messages. Of course, the tree is always closer to fully connected in the lower churn regime.

5.5 Live System Results

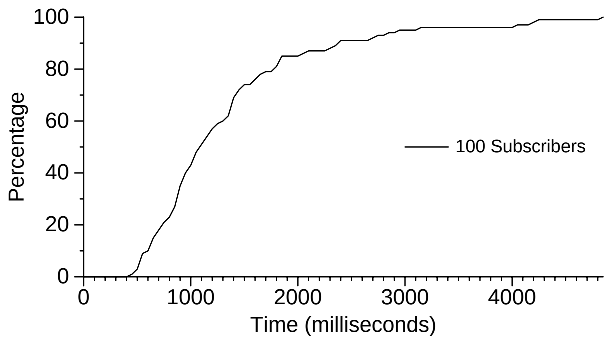


Figure 14: *Cumulative distribution of Subscription Latency for SV trees during a flash crowd, with 100 subscribers running on the cluster with 400 overlay participants on 40 machines.*

Figure 14 depicts the CDF of subscription latencies during a flash crowd on our live system. We used an overlay consisting of 400 virtual nodes on our 40 node cluster. We configured 100 of the nodes to simultaneously subscribe to a topic. Publications are sent only every 400 milliseconds so as not to overload the Modelnet router.

For most nodes, slightly more than a second elapsed between when subscription was initiated and when a content message was first received from the tree. All nodes were receiving content in less than 5 seconds from the initiation of subscription. These results are consistent with the results of our simulation experiments for flash crowds in Figure 12.

6 Conclusions and Future Work

Application-level multicast systems are a promising technology for delivering on the potential of multicast. Prior systems combining reverse path forwarding and scalable overlay networks have lacked politeness: participating nodes might be required to forward significant amounts of content in which they have no interest. This could be a major obstacle to the widespread adoption of application-level multicast. By providing politeness, Subscriber/Volunteer trees remove this potentially significant obstacle.

In addition to achieving politeness, SV trees inherit the scalability properties of RPF trees on an overlay network. SV trees also achieve greater efficiency that standard RPF trees by grafting unnecessary network hops out of the tree. Our simulation experiments validate these claims at large scales.

Our experience building SV trees is that avoiding orphaned sub-trees dictated a number of our design decisions. A key benefit of this design is that SV trees do not require loop detection and repair mechanisms. Our simulation experiments validate that the design decisions we made allowed us to achieve an implementation that worked under stress, including both flash crowds and churn.

Finally, we evaluated our implementation in a live system consisting of 400 virtual nodes running on a 40 node cluster. We presented results from subjecting our live system to a flash crowd, and these results closely match our simulation results. This evaluation makes the case for Subscriber/Volunteer trees as an important technology for enabling application-level multicast.

The focus of our larger research agenda has been using application level multicast for scalable event delivery, e.g., instant messaging. Another compelling use of application level multicast is streaming video. While SV trees clearly offer some benefit in the video context, the high bandwidth requirements may necessitate additional work to provide fine granularity control over the outgoing bandwidth consumed by forwarding. This is an exciting area for future research.

Acknowledgements

The authors wish to thank Patricia Jones for her expert editorial assistance.

References

[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proceedings of the SIGCOMM Conference*. ACM, Aug. 2002.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proceedings of Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[4] 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 Overlays. In *Proceedings of Infocom 2003*, Apr. 2003.

[5] Y. Chawathe. Scattercast: An Adaptable Broadcast Distribution Framework. *Multimedia Systems*, 9(12):104–118, July 2003.

[6] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *ACM SIGMETRICS 2000*, pages 1–12, June 2000.

[7] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems (TOCS)*, 8(2), May 1990.

[8] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for Wide-Area Multicast Routing. *IEEE/ACM Transactions on Networking*, 4(2), Apr. 1996.

[9] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. *Submitted for publication*, 2004.

[10] P. Francis. Yoid: Extending the Multicast Internet Architecture, 1999. White paper: http://www.icir.org/yoid/.

[11] N. J. A. Harvey, J. Dunagan, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. Technical Report MSR-TR-2002-92, Microsoft Research, 2002.

[12] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[13] Y. hua Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early Experience with an Internet Broadcast System Based on Overlay Multicast. In *USENIX Annual Technical Conference*, June 2004.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 197–212, October 2000.

[15] V. N. Padmanabhan and K. Sripanidkulchaiy. The Case for Cooperative Networking. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002.

[16] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *Proceedings of Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, Mar. 2001.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[18] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of the Third International Workshop on Networked Group Communication*, Nov. 2001.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[20] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, Nov. 2001.

[21] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, CA, Jan. 2002.

[22] A. C. Snoren, K. Conley, and D. K. Gifford. Mesh-Based Content Routing using XML. In *18th Symposium on Operating Systems Principles*, Oct. 2001.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[24] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Proceedings of IEEE INFOCOM 2001*, Apr. 2001.

[25] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[26] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, Apr. 1996.

[27] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, UC Berkeley, Apr. 2001.