Using Random Subsets to Build Scalable Network Services

Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat*
Department of Computer Science
Duke University
{*dkostic,razor,albrecht,abhi,vahdat*}*@cs.duke.edu*

Abstract

In this paper, we argue that a broad range of large-scale network services would benefit from a scalable mechanism for delivering state about a random subset of global participants. Key to this approach is ensuring that membership in the subset changes periodically and with uniform representation over all participants. Random subsets could help overcome inherent scaling limitations to services that maintain global state and perform global network probing. It could further improve the routing performance of peer-to-peer distributed hash tables by locating topologically-close nodes. This paper presents the design, implementation, and evaluation of *RanSub*, a scalable protocol for delivering such state.

As a first demonstration of the RanSub utility, we construct SARO, a scalable and adaptive application-layer overlay tree. SARO uses RanSub state information to locate appropriate peers for meeting application-specific delay and bandwidth targets and to dynamically adapt to changing network conditions. A large-scale evaluation of 1000 overlay nodes participating in an emulated 20,000-node wide-area network topology demonstrate both the adaptivity and scalability (in terms of per-node state and network overhead) of both RanSub and SARO. Finally, we use an existing streaming media server to distribute content through SARO running on top of the PlanetLab Internet testbed.

1 Introduction

Many distributed services must track the characteristics of a subset of their peers. This information is used for failure detection, routing, application-layer multicast, resource discovery, or update propagation. Ideally, the size of this subset would equal the number of all global participants to provide each node with the highest quality information. Unfortunately, this approach breaks down beyond a few tens of nodes across the wide-area, encountering scalability limitations both in terms of per-node state and network overhead. Recent work suggests building scalable distributed systems on top of a location infrastructure where each node can quickly (in $O(\lg n)$ steps) locate any remote node while maintaining only $O(\lg n)$ local state [22, 24, 26, 30]. This approach holds promise for scaling to distributed systems consisting of millions of participating nodes.

While existing techniques track the characteristics of a fixed set of $O(\lg n)$ nodes, a hypothesis of this work is that there are significant additional benefits from periodically distributing a different random subset of global participants to each node. By ensuring that the received subsets are uniformly representative of the entire set of participants and are frequently refreshed, nodes will eventually receive information regarding a large fraction of participants. Consider the applicability of such a mechanism to the following application classes:

- *Adaptive overlays:* A number of efforts build overlays that adapt to dynamically changing network conditions by probing peers. For instance, both Narada [14] and RON [2] maintain global group membership and periodically probe all participants to determine appropriate peering arrangements, limiting overall system scalability. The presence of a mechanism to deliver random subsets to each node would allow overlay participants to learn of remote nodes suitable for peering, while at the same time periodically learning enough new information to adapt to dynamically changing network conditions.
- *Parallel downloads:* One recent effort [5] suggests "perpendicular" downloads of popular content across a set of peers receiving erasure-coded content. Here, nodes receive data not only from the source, but also from peers that might have already

*This research is supported in part by the National Science Foundation (EIA-9972879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Albrecht is also supported by an NSF graduate fellowship and Vahdat is also supported by an NSF CAREER award (CCR-9984328).

received the data from the source or some other peer. One unresolved challenge to this approach is locating peers with both available bandwidth and diversity in the set of received data items. Random subsets would provide a convenient mechanism for locating such peers. Related to this approach, a number of efforts into reliable multicast [4] propose the use of peers in the multicast tree for data repairs (to avoid scalability issues at the root). Random subsets would likewise provide a convenient mechanism for locating nearby peers that do not share the same bottleneck link (and hence have a good chance of containing lost data).

- *Peer to peer systems:* For locality, peer to peer systems [22, 24, 26, 30] often desire multiple potential choices at each hop between source and destination. A changing, random subset of participating nodes would enable nodes to insert entries into their routing table with good locality properties and to adapt to dynamically changing network conditions.

- *Content distribution networks:* In CDNs, objects are stored at multiple sites spread across the network. Important challenges from the client perspective include resource discovery (determining which replicas store which objects) and request routing (sending the request to the replica likely to deliver the best performance given current load levels and network conditions). Random subsets would allow CDNs to track the state of a subset of global replicas. A number of earlier studies [9] indicate that making decisions based on a random subset of global information often performs comparably to maintaining global system state.

- *Epidemic algorithms:* A classic application of random subsets is epidemic algorithms [10, 27], where nodes transmit updates to random neighbors. With high probability, n nodes performing "anti-entropy" will converge to see the same set of updates in $O(\lg n)$ communication steps. Random subsets provides a convenient mechanism for locating neighbors and perhaps biasing communication to nearby sites.

Thus, we view a scalable mechanism for delivering uniformly random subsets of global participants as fundamental to a broad range of important network services. This paper presents the design and implementation of *RanSub*, one such protocol. RanSub utilizes an overlay tree to periodically distribute random subsets to overlay participants. We could leverage any number of existing techniques [3, 7, 12, 13, 14, 16, 19, 21, 23, 31] to provide this infrastructure. However, to demonstrate some of the key benefits of RanSub in support of adaptive overlay construction, we present the design and evaluation of SARO (Scalable Adaptive Randomized Overlay). SARO uses random subsets to build overlays that i) meet application-specified targets for delay and bandwidth, ii) match the characteristics of the underlying network and iii) adapt to changing network conditions.

Much like RanSub, a key goal of SARO is scalability: no node tracks the characteristics of more than $O(\lg n)$ remote participants and no node probes more than $O(\lg n)$ peers during any time period (a configurable epoch). Further, SARO requires no global coordination or locking to perform overlay transformations. A special instance of RanSub ensures a total ordering among all participants such that no two simultaneous transformations can introduce loops into the overlay.

We have completed an implementation of both SARO and RanSub and conducted a number of large-scale experiments. We show that a 1,000-node instance of SARO running on an emulated 20,000 node network using ModelNet [28] quickly converges to user-specified performance targets with low overhead from both per-node probing and RanSub operation. We further subject our prototype to live runs over the PlanetLab testbed [20], demonstrating similarly low convergence times, and the ability to stream live media over our overlays using publicly available media servers.

The remainder of this paper is organized as follows. Section 2 presents the RanSub algorithm for distributing random subsets. Section 3 then details SARO, a scalable and adaptive overlay that uses random subsets to conform to the underlying topology and dynamically adapt to changing network conditions. Section 4 evaluates our prototype's behavior under a variety of network conditions. Section 5 places our work in the context of related efforts and Section 6 presents our conclusions.

2 Random Subsets

2.1 Desirable Properties

Before we present the details of our design and implementation, we discuss desirable properties of a random subset "tool". Ideally, the system will offer:

1. *Customization:* Applications should determine the size of random subsets that are delivered. This size will depend on application-specific actions per-

formed by nodes upon receiving the random subset. For example, a parallel download application may wish to initiate data transfer with only a small constant number of peers while a P2P system may wish to probe $O(\lg n)$ nodes.

2. *Scalability:* The system should support large-scale services without posing a burden on the underlying network in terms of control overhead. Additionally, correct system operation should not depend on system size, i.e., the application should be able to request any random subset size. Overall, scalability implies that required per-node state and network communication overhead should grow sub-linearly with the number of participants.

3. *Uniform, changing subsets:* We envision a tool that is repeatedly invoked to retrieve "snapshots" of global participants at different points in time. Each snapshot, or random subset, should consist of nodes uniformly distributed across all global participants, such that each remote node appears in a delivered subset with equal probability. If desired by the appliaction, each invocation of the tool should return to each participant a *different* random subset independently chosen over all participants. Similarly, across invocations, each participant should receive probabilistically different subsets with no correlation across invocations. In this way, over time, each node can be exposed to a wide variety of global participants. Certain applications may desire non-uniform distribution that, for example, favors nearby nodes; this functionality can be layered on top of the baseline system.

4. *Frequent updates:* To support network services that use the system to adapt to changing network conditions, the system should offer frequent distribution of random subsets.

5. *Resilience to failures:* The system will preserve its properties even in the face of failures. Failed nodes should not appear in future random subsets within a short and bounded amount of time.

6. *Resilience to security attacks:* Even when under attack by malicious users, the system should maintain its properties (uniform distribution, etc.), and degrade its performance gracefully when it is unable to defend against a massive attack.

2.2 Overview

Given the goals described above, we now describe RanSub, our scalable approach to distributing random subsets

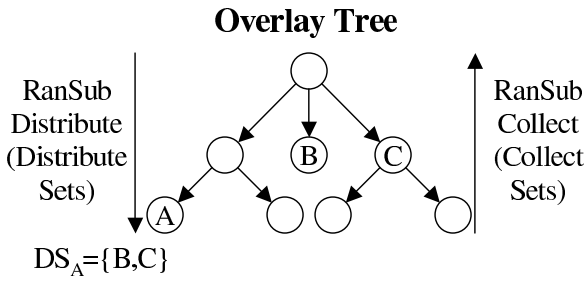

Figure 1: RanSub operation.

containing nodes that are uniformly spread across all participants. For the purposes of this discussion we assume the presence of some scalable mechanism for efficiently building and maintaining an overlay tree. A number of such techniques exist [3, 14, 16]; in the next section, we describe SARO, one technique for building such an overlay that both makes use of RanSub functionality and also provides the necessary overlay infrastructure.

Figure 1 summarizes RanSub operation. RanSub distributes random subsets through *Collect* messages that propagate up the tree and leave state at each node. *Distribute* messages traveling down the tree use soft state from the previous collect round to distribute uniformly random subsets to all participants.

RanSub distributes a subset of participants to each node once per configurable *epoch*. An epoch consists of two phases: one *distribute phase* in which data is transmitted from the root of an overlay tree to all participants (data is distributed down the tree) and a second *collect phase* where each participant successively propagates to its parent a random subset called a *collect set* (CS) containing nodes in the subtree it roots (data is aggregated up the tree). During the distribute phase, each node sends to its children a uniformly random subset called a *distribute set* (DS) of remote nodes. The contents of the distribute set are constructed using collect sets gathered during the previous collect phase.

When a Distribute message reaches a leaf in the RanSub tree, it triggers the beginning of the next collect phase where each node sends its parent a subset of its descendants (the collect set) along with other metadata. This process continues until the root of the tree is reached. The collect phase is complete once the root has received collect sets from all of its children. The root signals the beginning of a new epoch by distributing a new distribute set to each of its children, at which point the entire process begins again. The length of an epoch is configurable

based on the requirements of applications running on top of RanSub. The lower bound on the length of an epoch is determined by the worst-case root-to-leaf and leaf-to-root transmission times of the overlay.

2.3 Collect/Distribute

Each node participating in a RanSub overlay maintains the following state: address of its parent in the overlay, a list of its children, and the sequence number of the current epoch. In addition, it maintains the following soft state: a collect set and number of subtree descendants for each of its children, a distribute set, and the total number of overlay participants. Below, we describe how RanSub uses this information and how it maintains it in a decentralized manner.

2.3.1 Collect Phase

Overall, the goal of the Collect message is for each node to: i) compose the collect sets for constructing the distribute set during the subsequent distribute phase, and ii) determine the total number of participants in its local subtree.

The collect phase begins at the leaves of the tree in response to the reception of a Distribute message. Table 1 describes all the fields in Collect messages (in the left half of the table). The Collect message has the same sequence number as the triggering Distribute message. At the leaves, the number of descendants is set to one and the collect set contains only the leaf node itself. Once a parent receives all Collect messages from its children, it further propagates a Collect message to its own parent. The nodes in the collect set are selected randomly from the collect sets received from its children to form a subset of configurable size ($O(\lg n)$ by default). Each node stores this collect set to aid in the construction of distribute sets distributed to its children during the subsequent distribute phase.

One key challenge is to ensure that membership in the collect set propagated by a node, A, to its parent is both random and uniformly representative of all members of the sub-tree rooted at A. To achieve this, RanSub makes use of a *Compact* operation, which takes as input multiple subsets and the total population represented by each subset. *Compact* outputs a new subset with two properties: i) group membership randomly chosen over the input subsets and ii) a target size constraint. This is achieved by building the output set incrementally. We first randomly choose an input subset based on the population that it represents. We then randomly choose a member of this subset (not already selected) and add it to the output set. Consider the case where *Compact* were performed over two subsets, A and B. A and B each contain 8 members, A represents a population of 30 while B represents a population of 10. *Compact* would choose members of A to add to the output set with probability 0.75. Thus, the output set of 8 members would uniformly represent a population of 40, with an *expected* membership of 6 members from A and two members from B. Note that *Compact* is able to properly weigh each subset because as part of collect/distribute, each node learns the total number of nodes at the subtree rooted at each of its children.

2.3.2 Distribute Phase

A new epoch can begin once the root has received a Collect message from all of its children for the previous epoch. The actual length of an epoch is determined by individual application requirements. The right half of Table 1 describes the fields contained in the Distribute message.

A parent constructs distribute sets for each child in the following manner. Recall that each node stores the collect set received from each child during the previous collect phase. Thus, for each of k children, a particular node maintains $CS_1, CS_2, \ldots, CS_k$. Also recall that each collect set, CS_i, consists of nodes selected uniformly randomly from the subtree rooted at node i. A parent node A constructs a distribute set for each child from this information saved during the preceding collect phase. This information includes the collect set for each child, the node A itself, as well as DS_A, A's own distribute set.

To the application using it, RanSub offers three choices regarding the contents of distribute sets:

- *RanSub-all :* This is suitable when the application requires uniformly random subsets of all nodes in the system. There are two flavors of the *All* option. *All-identical* delivers the *same* distribute set to all nodes in the overlay. This distribute set, DS_{root}, is created by the root using the Compact operation:

  $$DS_{root} = Compact(CS_1, \ldots, CS_j, \{root\})$$

  where CS_i represents the subtree rooted at child i of the root (numbered $1, \ldots, j$). A potentially more useful construct is evident with the *ALL-non-identical* option that delivers *different* distribute sets

Collect		Distribute	
Sequence #	Sequence number of current epoch	*Sequence #*	Sequence number of current epoch
Collect Set	Uniformly random subset of nodes in sender's subtree	*Distribute set*	Uniformly random subset of overlay participants
Descendants	Estimate of number of nodes in sender's subtree	*Participants*	Estimate of total number of nodes in the overlay
		Reshuffle flag (only for ordered RanSub)	Determines if children should be reshuffled so that a new total ordering is created

Table 1: Contents of Collect and Distribute messages.

to each node. In this case, node Z receives a Distribute message from its parent P containing DS'_P. Z constructs DS_Z using DS'_P and the collect sets stored from its children, $CS_1, \ldots, CS_k$, in the following manner:

$$DS_Z = Compact(CS_1, \ldots, CS_k, DS'_P, \{P\})$$

Z then forwards the following to each child x:

$$DS'_Z = Compact(CS_1, \ldots, CS_{x-1}, CS_{x+1}, \ldots, CS_k, DS'_P, \{P\})$$

Note that the root's DS'_P is $\{\}$ since it has no parent.

- *RanSub-nondescendants :* In this case, each node should receive a random subset consisting of all nodes except its descendants. This might be appropriate for an application-layer multicast structure where participants are probing for better bandwidth and latency to the root of the tree. In this case, considering a node's descendants could introduce a cycle in the overlay tree. For each child x (numbered $1, \ldots, k$), the parent node A constructs DS_x in the following manner:

$$DS_x = Compact(CS_1, \ldots, CS_{x-1}, CS_{x+1}, \ldots, CS_k, DS_A, \{A\})$$

- *RanSub-ordered :* This type of distribute set calculation imposes a total ordering among participating nodes. A node receives a distribute set containing random nodes that come before it in the total ordering. For each child x (numbered $1, \ldots, k$), the parent node A constructs DS_x in the following manner:

$$DS_x = Compact(CS_1, \ldots, CS_{x-1}, DS_A, \{A\}) \quad (1)$$

Our sample application, an adaptive application-layer multicast overlay, uses Ransub-ordered to ensure that simultaneous transformations to the tree structure do not introduce loops (as discussed in Section 3). Thus, we assume RanSub-ordered for the remainder of this paper.

2.3.3 Discussion

A limitation of RanSub-ordered is that the first child of a particular node will always have a smaller set of potential nodes to choose from than the kth. In fact, the first child's distribute set would always be restricted to a relatively small subset of global nodes. For RanSub-ordered, this violates our goal of distributing random subsets to *all* nodes that are uniformly chosen across all global participants in a single epoch. We take the following step to ensure that every node still receives a uniformly random subset across multiple invocations of RanSub-ordered. Every configurable r epochs, the root of the overlay periodically sets the reshuffle flag in its Distribute message, signaling overlay participants to randomly reshuffle children lists. This allows children that were at the beginning of the total ordering (and hence received few nodes in distribute sets) a chance to move toward the end of the total ordering and receive information about more nodes.

Figure 2 summarizes the operation of the two phases of the RanSub protocol. For simplicity, we do not include the results of *Compact*, which would appropriately reduce the size of all subsets to the application-specified size constraint. In the collect phase for this example, each node constructs a collect set (CS) composed of the union of itself and all members of the collect sets it received from its children. Thus, A receives a collect set from each of its children, B and C, that are uniformly representative of the subtrees rooted at B and C respectively. Each node determines which of its collect sets should be used to compose the distribute set (DS) for each of its children.

For the distribute phase, node A constructs DS_B by taking the union of itself (A) with CS_C. Node B in turn constructs DS_D by taking the union of itself (B) with its own distribute set ($DS_B = \{A, C, F, G\}$) and $CS_E = \{E\}$ from the previous collect phase. D gets "lucky" in this ordering (it is actually the last node in this total ordering)

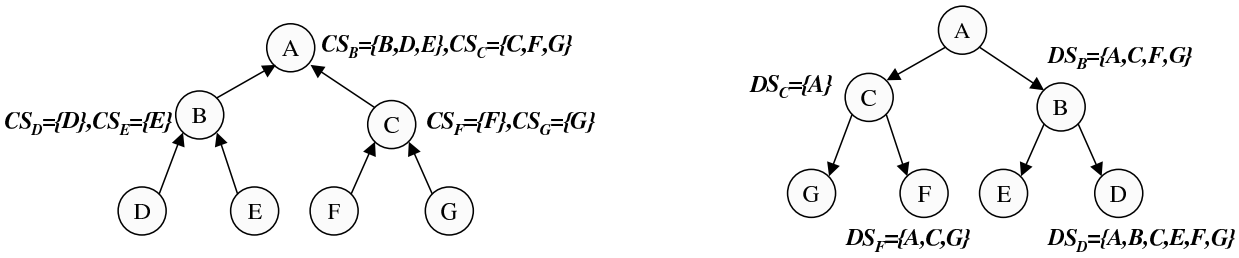

Figure 2: Example scenario depicting the two phases of the RanSub protocol: The collect phase traveling up the overlay in the left panel and the distribute phase traveling down the overlay in the right panel.

and receives a distribute set representative of the entire topology (once again, recall that we are omitting the compact operation that would throw out appropriate set elements to maintain size-constrained sets). Node G would get "unlucky" and only receive a distribute set consisting of itself (it is the first node in this total ordering). However, once the children lists are reshuffled, the resulting total ordering would be a random walk of the tree in which each node is only visited once yielding an entirely different new total ordering. Finally, note that the complexity of reshuffling children is only needed if a total ordering is required.

2.4 Comparison with the Ideal Random Subset Primitive

We believe that RanSub closely approximates the ideal properties outlined at the beginning of this section. Since it uses a tree to propagate sublinear-sized collect and distribute sets between parents and children, it imposes low overhead on the underlying network. Using an efficient overlay structure further ensures that epochs can be short. For instance, we find that for a 1000-node system (in a network with a diameter less than 500 ms), epochs can be as short as five seconds.

In the absence of node failure, RanSub delivers random subsets that are close to uniformly distributed. The key behind achieving uniformity is accounting for nodes that are represented by a random sample at any given time during the protocol execution. We achieve this by running the protocol over a tree, where it is straightforward for a node to have an estimate of the number of its descendants. Future work includes adapting RanSub to function over meshed overlays, not just trees.

RanSub uniformity might suffer as nodes join and leave the system. We do not provide the guarantee that nodes in the random subset will be active when considered by other nodes. RanSub is a mechanism for taking a snapshot of "live" tree participants and then taking uniformly random samples of it. This is a strength of our approach since we do not require a separate group membership mechanism. If the node is alive when it receives a Collect message, it may be included in distribute sets given to other nodes. If that node fails soon after the snapshot, it may be unavailable when considered by other nodes. If a node joins after the collect phase of a previous epoch has completed, it will not be present in the snapshot.

In RanSub, the time for node failure detection can be on the order of a small number of seconds. Nodes below a point of failure rejoin the tree at current participants using previously received distribute sets. The node failure detection interval can be further reduced if the underlying tree is used to support application-layer multicast. In this case, absence of data for a few hundreds of milliseconds might signify disconnection from the parent. In essence, application-layer data may serve as a heartbeat mechanism for failure detection.

RanSub assumes trust between overlay participants, and therefore is not resilient to internal attacks. For example, a malicious node might alter the contents of collect and distribute sets. We are investigating several techniques to address this limitation. For instance, one approach involves sending subsets over multiple tree "cross-links" to allow identification and confinement of damage caused by malicious users. These "cross-links" would also help with the overall reliability of the tree. Multiple paths through the tree means that a single failure may not disconnect nodes in the tree.

3 SARO

3.1 Overview

As discussed earlier, distributing uniformly random subsets of global participants is applicable to a broad range of important services. This section describes SARO, a Scalable Adaptive Randomized Overlay, one such application of RanSub. The use of RanSub for SARO is circular in this example. SARO uses random subsets to probe peers to locate neighbors that meet performance targets and to adapt to dynamically changing network conditions. At the same time, RanSub uses the SARO overlay for efficient distribution of Collect and Distribute messages.

The goal of SARO is to construct overlays that are: i) scalable, ii) degree-constrained, iii) delay- and bandwidth-constrained, iv) adaptive, and v) self-organizing. For scalability, we enforce the following rules:

1. No node should track more than $O(\lg n)$ remote nodes.
2. No node should perform more than $O(\lg n)$ network probes during any time period (epoch). The application can configure the number of probed nodes to effect a tradeoff between network overhead and the adaptivity (or agility [17]) of the overlay.
3. No global locking should be required to transform the overlay.

We achieve the first two goals using the distribute sets that RanSub transmits to each node every epoch. Each SARO node A performs probes to members of this subset to determine if a remote node B exists that would deliver better delay or bandwidth to A and its descendants. If so, A attempts to move under B.

To motivate the third requirement, consider a node A that decides to move underneath a remote node B. The system would introduce a loop if some ancestor of B simultaneously decides to move under some descendant of A. The naive approach to avoiding loops requires locking a number of nodes across the wide area to avoid such simultaneous overlay transformations. While this may be appropriate for a small number of nodes or for LAN settings, this process will not scale to large overlays. Thus, we impose a total ordering among nodes provided by the ordered flavor of RanSub to ensure that no two simultaneous moves in the same epoch can introduce a SARO loop. During any epoch, each node's distribute set contains only remote nodes that come before it in the current total order.

Recall that RanSub periodically changes the total ordering of nodes and random membership in delivered distribute sets. Thus, while each node only tracks and probes $O(\lg n)$ remote nodes during any one epoch, RanSub ensures that the makeup of the distribute set changes probabilistically such that, over time, each node quickly probes all potential parents. The size of the random subset effects a tradeoff between scalability (measured by per-node state and network probing overhead) and convergence time (the amount of time it takes to build an overlay that achieves delay and bandwidth targets even under changing network conditions).

SARO requires additional information beyond that distributed by RanSub (see Table 1). In general, applications may wish to piggyback different state information with the existing Collect and Distribute messages. Our RanSub layer is designed in a manner to make such extensibility straightforward, though a detailed description is beyond the scope of this paper. Table 2 describes the additional information transmitted by SARO in Collect and Distribute messages.

3.2 Probing and Overlay Transformations

Recall that the length of an epoch (random subsets are distributed once per epoch) is determined by application-specific requirements. Shorter epochs provide more information, while longer epochs incur less overhead. For SARO, our goal is to quickly converge to an overlay that matches the underlying topology and adapts to dynamically changing network characteristics. The implementation and evaluation in this paper is pessimistic in that we run with a constant epoch length of 10 seconds in all cases. In the future, we plan to investigate setting the epoch length adaptively based on both overlay characteristics and network conditions. For example, if SARO has already matched the underlying topology and if network characteristics are not changing rapidly (likely the common case in the Internet), then the system can afford a longer epoch length. Thus, we envision reducing overall system overhead by running SARO with a short epoch length during initial self-organization and in response to large changes in network conditions, but running with a long epoch in the common case.

A key to SARO's ability to converge to bandwidth and delay (maximum delay from root to all other overlay participants) targets lies in localized tree *transformations*. Dur-

Collect		Distribute	
Delay	Delay estimate for furthest descendant	*Tree height*	Estimate of the actual highest root-to-leaf latency in the tree
Delay gain	Estimate of delay gain from moving to a best alternative parent	*Root delay*	Estimate of recipient's delay from root

Table 2: Additional fields in Collect and Distribute messages required by SARO.

ing each epoch, nodes measure the delay and bandwidth between themselves and all members of their distribute set. These probes consist of a small number of packets inter-spaced by the target application bandwidth[1]. If the loss rate of the probes is below a specified threshold, the probing node calculates the average round trip time. The goal is to locate a new parent that will deliver better delay, better bandwidth, or both to itself and all of its descendants. If a better parent is located, the child attempts to move under it. The migrating node issues the add request to the potential parent and waits for the response. If the request is accepted, it notifies its old parent, communicates its new delay from root to all its children, and notifies the new parent of its furthest descendant (updating the parent's state for the next epoch).

In general, the goal of SARO is to achieve the lowest delay configuration that still maintains the target bandwidth to the root. Thus, during each epoch, nodes use information from their probes to perform two types of transformations: `BANDWIDTH_ONLY` and `DELAY_AND_BANDWIDTH`. Nodes that have not yet reached their bandwidth target may perform `BANDWIDTH_ONLY` transformation to improve their bandwidth to root, even if it means increasing their delay. `DELAY_AND_BANDWIDTH` transformations allow nodes to rotate under a new parent that improves the nodes' delay to root while maintaining (or improving) bandwidth.

3.3 Dynamic Node Addition and Failure Recovery

To this point, our discussion assumes a static set of nodes dynamically self-configuring to match changing network conditions. In general however, the set of overlay participants will also be changing. A node performing a SARO join simply needs to contact any existing member of the overlay. The initial bootstrapping parent may be sub-optimal from the perspective of bandwidth or delay. However, the node will begin receiving random subsets as part of the collect/distribute process, and it can use the associated random subsets to probe for superior parents using the process described above.

Since the bootstrapping parent node might fail, we make an additional assumption that every node is aware of the root of the tree. Therefore, an incoming node will always have at least one bootstrapping parent (similar to Overcast [16], we can replicate the root to improve root availability). For the sake of scalability, we allow nodes to contact the root only if it is absolutely necessary (i.e., its original bootstrapping parent failed).

If a bootstrapping parent does not have enough slots in its children list to accept a joining node, it redirects the incoming node randomly to either its parent or one of its children. Similarly, if the incoming node would violate the delay bound, the bootstrapping parent redirects it to its parent. If the node is redirected too many times, it joins the tree at the first available point, and tries to improve its position later.

Handling node failure is simplified by our periodic distribution of Collect and Distribute messages, which implicitly act as heartbeat messages. Each parent waits for Collect messages from all of its children. If a message is not received within some multiple of the subtree height[2], the parent assumes that one of its children has failed and excludes it from participating in the next epoch by not sending a Distribute message to that child. However, the node does proceed with a Collect message to its own parent when it detects the failure to ensure that a failure low in the tree does not cascade all the way back up the tree. A node, A, can similarly detect the failure of its parent when it does not receive a Distribute message within a multiple of the delay target. In this case, A will send a dummy Distribute message (with an empty distribute set) to all of its children. This empty Distribute message signals A's descendants that no probing or overlay transformations should be performed during this epoch. A then attempts to locate a new parent using information from

[1] We currently assume that overlay traffic makes up a relatively small portion of overall traffic through the bottleneck. We leave more accurate, more stable, and TCP-friendly probing to future work. In general, higher accuracy probes will inherently incur higher overhead, though this issue is orthogonal to our own work.

[2] Note that the current maximum node-to-leaf delay serves as a convenient baseline for the complex process of determining appropriate timeouts.

previous distribute sets where appropriate. Thus, upon a failure, the entire subtree rooted at A is able to rejoin the overlay with a single transformation, rather than forcing all nodes below A to rejoin separately.

3.4 Weans

Under certain circumstances, the greedy nature of SARO can lead to sub-optimal overlays. Consider the following situation. A node A with a particular degree bound has a full complement of children. However, a node B somewhere else in the overlay can only achieve its bandwidth target by becoming a child of A. Finally, one of A's children, C, is best served by A but would still be able to achieve its bandwidth target as a child of some fourth node D. As described thus far, SARO will become stuck in a "local minimum" in this situation.

We address this situation by introducing *wean* operations. At a high level, the goal is to ensure that each parent leaves a number of slots open whenever possible to address the above condition. Thus, as a node approaches its degree limit, it will send a wean message to one of its children. In subsequent epochs, that child will move to a new parent if it can find a suitable location that still meets its bandwidth requirement (though its delay may be increased). A wean may or may not succeed (an appropriate alternate parent may not exist) and the wean operation expires after a configurable number of epochs.

One difficulty is choosing the child to wean. Ideally, a parent would wean the child that would lose the least in delay and bandwidth while still achieving its targets. We approximate this in the following manner. During each epoch, all nodes maintain information on the *best alternate parent* with respect to both delay and bandwidth. This information is propagated to parents in the collect phase (see Table 2) and is used by parents to determine the wean target.

4 Evaluation

We have completed an implementation of both RanSub and SARO as described in the previous sections. We wrote our code to a compatibility layer that allows us to evaluate our working system in both the *ns* packet simulation environment [18] and across live networks. For brevity, we omit the majority of the results of our detailed simulation evaluation. Instead, we focus on the behavior of our system running live on ModelNet [28], an Internet emulation environment and across the Internet in the PlanetLab testbed [20].

For our ModelNet experiments, SARO runs on 35 1.4Ghz Pentium-III's running Linux 2.4.18 and interconnected with both 100 Mbps and 1 Gbps Ethernet switches. We multiplex 28-29 instances of SARO on each of the Linux nodes, for a total of one thousand nodes self-organizing to form overlays. We validated our ModelNet results with ns experiments using identical topologies and communication patterns. In ModelNet, all packet transmissions are routed through a *core* responsible for emulating the hop-by-hop delay, bandwidth, and congestion of a target network topology. For our experiments, we use a single 1Ghz Pentium III running FreeBSD-4.5 as the core. The core has a 1 Gbps connection to the rest of the cluster. The core was never a bottleneck either in CPU or bandwidth for any of our experiments. Our earlier work [28] shows that, for a given wide-area topology, ModelNet is accurate to within 1 ms of the target end-to-end packet transmission time up to and including 100% CPU utilization, and 120,000 packets/sec (1 Gbps assuming packets are 1000 bytes on average). ModelNet can scale its capacity with additional core nodes, though this was not necessary for our experiments here. ModelNet emulates packet transmission hop-by-hop (including per-hop queue sizes and queuing disciplines) through a target network. Thus, a packet's end-to-end delay accounts for congestion and for per-hop queuing, propagation, and transmission delay.

By default, the core emulates the characteristics of a random 20,000-node INET-generated topology [6]. We randomly assign 1,000 nodes to act as clients connected to one-degree stub nodes in the topology. One of these participants is randomly selected to act as the root of the SARO tree. We classify network links as being Client-Stub, Stub-Stub (both with bandwidth between 1-10 Mbps), Transit-Stub (bandwidth between 10-100 Mbps) and Transit-Transit (100-155 Mbps). We calculate propagation delays among nodes from the relative placement of the nodes in the plane by INET. The baseline diameter of the network is approximately 200 ms. While the presented results are restricted to a single topology, the results of additional experiments and simulations all show qualitatively similar behavior.

4.1 RanSub Uniformity

To verify that RanSub distributes uniformly random subsets to all nodes, we used our complete RanSub prototype to create a SARO overlay of 1000 emulated nodes as described above. After convergence, we let our experiment run for a total of 360 epochs and tracked the cumulative number of unique remote peers that RanSub distributes to each node over time. We configured RanSub to distribute

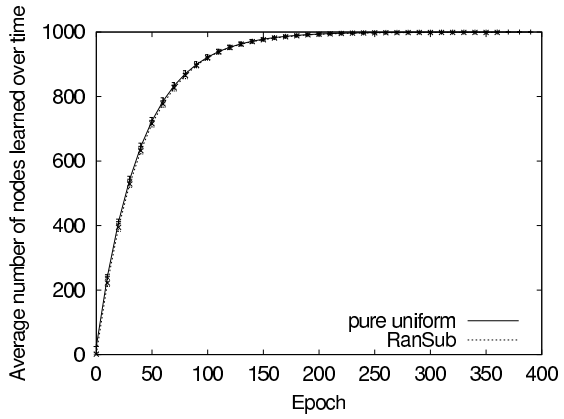

Figure 3: Average number of nodes each of 1,000 nodes learns of as a function of number of epochs for the optimal pure uniform case and for RanSub.

25 random participants each epoch. Figure 3 plots the average number of peers each node is exposed to on the y-axis as a function of time progressing in epochs on the x-axis. The vertical bars represent the standard deviation. We also show the best case in which we simulate pure uniform random subsets (using the same random number generator as the one used by our RanSub implementation). RanSub delivers random subsets with essentially optimal uniformity.

4.2 SARO Overlay Convergence

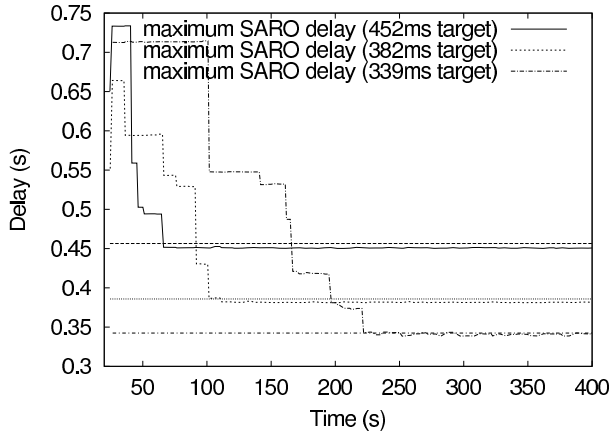

Figure 4: Delay convergence as a function of time for three different delay targets.

Figure 4 shows the convergence time of three SARO overlays running with a maximum degree of 10 and random subsets of size 15. The 1000 nodes join the overlay sequentially at a random point in the network over the first 20 seconds of the experiment (50 nodes/second) and then use random subsets to probe for parents that will deliver the appropriate delay target (bandwidth targets of 64Kbps were easily achieved for this experiment). In effect, at the beginning of the experiment we pessimistically create an overlay with random interconnectivity. We observe the behavior of the system for three different delay targets, 339 ms, 382 ms, and 452 ms. The 339 ms delay target is quite difficult to achieve for our topology and degree bound. The figure plots the achieved worst-case delay relative to the delay target as a function of time progressing on the x axis. SARO uses random subsets to converge to the specified delay target in all cases, with convergence time varying from 60 seconds to 220 seconds in the three cases depending on the tightness of the delay target. We note that our convergence times using random subsets are comparable to a number of smaller-scale overlay construction techniques that maintain global state information and that perform global probing.

4.3 Effects of Random Subset Size

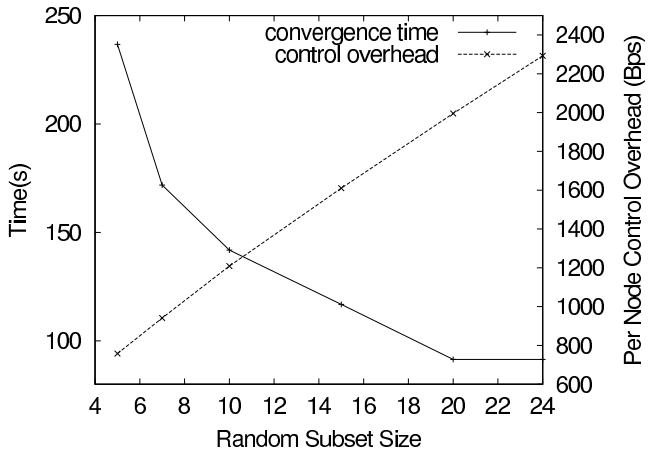

Figure 5: Delay convergence time and resulting per-node probing overhead as a function of the size of the random subset.

We now quantify the effects of the size of the random subsets on SARO convergence time. In general, less information in the random subset increases convergence time as nodes have to spend more time to find an appropriate parent. More information will likely decrease the convergence time, but at the cost of increased network probing overhead. With a maximum per-node degree of 10, we measure the time for the 1,000 node SARO tree to converge to a 382 ms delay target as a function of the size of the random subsets. As shown in Figure 5, we increase the size of the random subset from 5 to 24 on the x-axis and plot the resulting convergence time on the left-hand y-axis. The convergence time decreases from approxi-

mately 240 seconds to 90 seconds. Figure 5 also plots the associated tradeoff with per-node control overhead, accounting for both RanSub Collect/Distribute messages and probing overhead. As expected, probing overhead on the right-hand y-axis grows linearly with the size of the random subset, but only to a manageable 2300 bytes/sec even when the random subset size grows to 24, most of which is probing overhead. Note that the benefits of increasing the random subset size beyond 20 diminishes rapidly. Of course, this point of diminishing returns will vary with the topology and delay target. We have experimented with dynamically increasing or decreasing the random subset size by taking real-time measurements of the overlay's convergence, but we leave this to future work.

4.4 Adaptivity

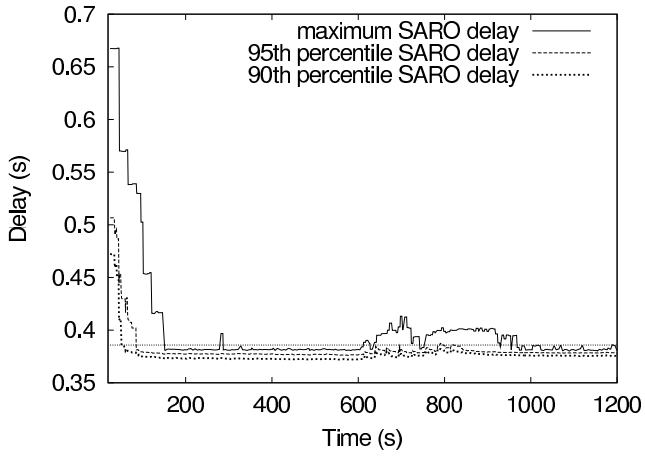

Figure 6: Adaptivity of a SARO tree in response to pronounced change in network delay.

One of the most important aspects of SARO is its ability to dynamically react to changing network conditions. To evaluate this ability, we subject a steady-state 1000-node SARO tree to widespread and sustained change in network characteristics. Every 25 seconds, we increase the propagation delay of a randomly chosen 14% of all network links by between 0-25% of the link's original delay. The idea behind this experiment is to determine what happens to the overlay as the network continuously degrades under conditions much worse than those typically found on the Internet. Figure 6 shows the results of this experiment, plotting delay as a function of time for the 90th percentile, 95th percentile, and worst case node in the overlay as a function of time progressing the x-axis. Note that the 90th percentile indicates that 90% of SARO nodes have delay better than the indicated value. We set the delay target to 382 ms for this experiment, the degree bound to 10 and the size of the random subset to 15.

We intentionally set our target delay to a value that is relatively difficult to achieve for our degree bound in the face of highly variable network conditions. Thus, the overlay initially takes approximately 150 seconds to converge to the delay target. We perturb network conditions in the manner described above beginning at time $t = 600$ and continuing for 200 seconds. While 95% of nodes are able to maintain their delay target during most of the network perturbation, it takes SARO another 180 seconds after the network perturbation subsides (though link delays remain at their elevated levels) to once again bring all nodes within delay bounds. We include the results of this experiment to quantify the level of adaptivity that SARO can deliver. Additional experiments indicate that if network conditions were perturbed for a longer period of time (with same number of links, magnitude of change, and frequency of change), SARO would be unable to once again achieve the delay target. With less perturbation or a more relaxed delay target, SARO typically quickly recovers from changes to network conditions.

4.5 PlanetLab Deployment

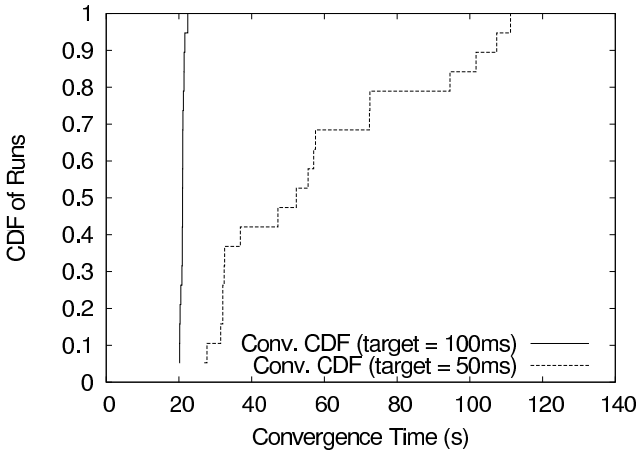

Figure 7: CDF of convergence time for 19 PlanetLab nodes, with each node acting as root in turn for two different delay targets.

To further evaluate the utility of our approach, we evaluated the behavior of SARO running on a subset of 19 PlanetLab nodes [20] during September 2002. We ran SARO over PlanetLab 19 times with each separate run using a different PlanetLab node as the root of the overlay. We then measure the convergence time for each of two different delay targets: 50ms and 100ms. We set the maximum per-node degree to 5 and the random subset size to 5. Figure 7 plots the result of this experiment. We

find that for a relatively relaxed delay target of 100ms, all nodes converge within 20 seconds. Tightening the delay bound to 50 ms increases the typical convergence time to approximately a minute with worst case convergence taking up to two minutes. Note that all reported values are once again for the worst-case convergence time of the last node to meet its delay target.

Next, we streamed live audio over SARO by integrating an HTTP proxy as a separate thread in the SARO address space, as depicted in Figure 8. We use a publicly available SHOUTCast server [25], to stream MP3 encoded over HTTP. We then instantiate a SARO/HTTP process on the same node as the SHOUTCast server to act as the root of the overlay tree. Each SARO node below the root instructs its associated HTTP proxy to establish a connection to its parent to receive streaming data. The node is then able to stream content to local Winamp players and to its own children in the overlay. Each time a SARO node locates a better parent, it also instructs is HTTP proxy to reestablish a connection to this new parent.

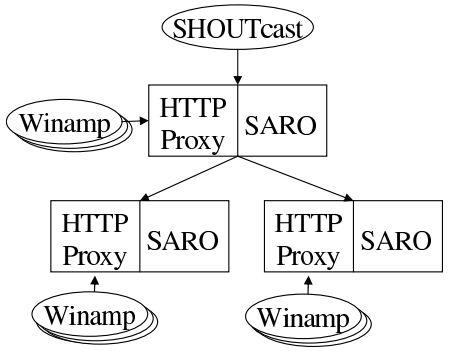

Figure 8: Streaming live media using SARO and SHOUTcast over PlanetLab.

We use this configuration to successfully stream MP3 files at 256Kbps from an existing SHOUTcast servers over 1000 nodes in our emulation environment and the PlanetLab testbed. Figure 9 shows the results of a 5-minute experiment of SHOUTcast streaming over 19 SARO nodes spread across the PlanetLab testbed. We set the delay target to 75ms, the size of the random subsets to 5 and the maximum per-node degree to 5. The figure plots a CDF of the percent of bytes received by each of the 19 nodes. Byte loss rates vary from 0-5%. However, we note that we measure the loss rate while SARO was still self-organizing at the beginning of the experiment. The vast majority of the losses came at this time. We verified the correctness of our experiment by connecting to individual HTTP/SARO nodes using the Winamp media player to playback the stream.

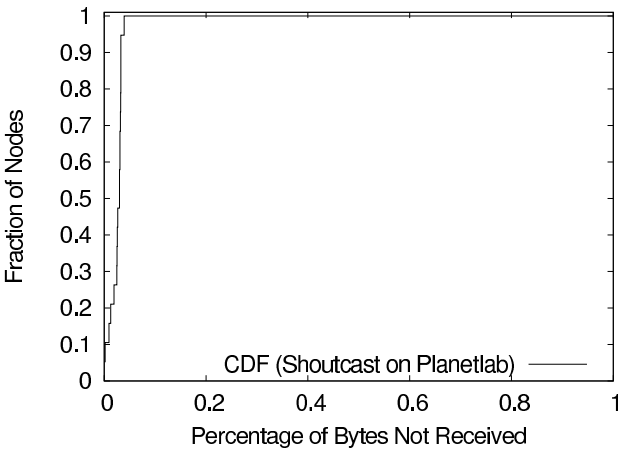

Figure 9: Distribution of percent of packets received for a 5 minute experiment streaming 256Kbps over SARO running on PlanetLab.

5 Related Work

RanSub shares some design goals with gossip-based dissemination protocols such as SCAMP [1] and LBPCAST [11]. However, because it operates over a tree, RanSub offers uniformly random subsets relative to these earlier efforts. RanSub does not require a minimum random subset size for correct operation. Finally, in combination with SARO, RanSub allows more predictable time between updates (one epoch).

Our work on SARO builds upon a number of recent efforts into "application-layer" multicast, where nodes spread across the Internet cooperate to deliver content to end hosts. Edges in this overlay are TCP connections, ensuring congestion control and reliability in a hop-by-hop manner. Perhaps most closely related to our effort in this space is Narada [14, 15], which builds a mesh interconnecting all participating nodes and then runs a standard routing protocol on top of the overlay mesh. Relative to our work, Narada nodes maintain global knowledge about all group participants. In comparison, we use the RanSub layer to maintain information about a probabilistic $O(\lg n)$ subset of global participants, making applications built on top of RanSub, including SARO, more scalable.

SARO bears some similarity to the Banana Tree Protocol (BTP) [13], and Host Multicast Tree Protocol (HMTP) [29]. However, neither of these approaches attempt to provide delay or bandwidth guarantees and neither considers a two-metric network design. All three protocols use the idea of tree transformations based on

local knowledge (obtained through limited network probing) to improve overall tree quality. However, BTP implements a more restrictive policy for choosing a potential parent, called "switch-one-hop", which considers only grandparents and immediate siblings. HMTP can introduce loops and thus requires loop detection that requires knowledge of all of one's ancestors. Since HMTP offers no bounds on tree height, message size and required state are also unbounded ($O(n)$), rendering this approach potentially unscalable. Finally, HMTP is not evaluated under changing network conditions.

Yoid [12] shares the design philosophy that a tree can be built directly among participating nodes without the need to first build an underlying mesh. Yoid does not describe any scalable mechanism for conforming to the topology of the underlying network, has not been subjected to a detailed performance evaluation, and contains loop detection code as opposed to our approach of avoiding loops.

ALMI [19] uses all-pairs probing at a cost of $O(n^2)$ and transmits this changing connectivity information to a centralized node that calculates an appropriate topology for the overlay. RMX [7] faces similar scalability limitations. In Overcast [16], all nodes join at the root and migrate down to the point in the tree where they are still able to maintain some minimum level of bandwidth. Relative to our effort, Overcast does not focus on providing delay guarantees (given its focus on bandwidth-intensive applications). Its convergence time is also limited by probes to immediate siblings and ancestors. NICE [3] uses hierarchical clustering to build overlays that match the underlying network topology. Relative to our approach, NICE focuses on low-bandwidth (i.e., single-metric, delay-optimized) applications and requires loop detection code. We believe that a variety of existing overlay construction techniques, including Yoid, ALMI, RMX, Overcast, and NICE could benefit from the availability of our RanSub layer.

Finally, a number of recent efforts [21, 23, 31] propose building application-layer multicast on top of scalable peer-to-peer lookup infrastructures [8, 22, 24, 30]. While these projects demonstrate that it is possible to probabilistically achieve good delay relative to native IP multicast, they are unable to provide any performance bounds because of the probabilistic nature of the underlying peer-to-peer system. Further, these systems do not focus on two-metric network optimization (e.g., delay and bandwidth). Finally, to the best of our knowledge, these approaches, have largely been evaluated through simulation and have not been subjected to live Internet conditions.

6 Conclusions

This paper argues for a generalized mechanism for periodically distributing state about random subsets of global participants in large-scale network services. Sample applications include epidemic algorithms, reliable multicast, adaptive overlays, content distribution networks, and peer-to-peer systems. This paper makes the following contributions:

- We present the design and implementation of a scalable protocol, RanSub, that distributes state about uniformly random subsets of configurable size once per application-specific epoch.
- We argue for the utility and generality of such an infrastructure through the evaluation of SARO, a scalable and adaptive overlay construction protocol. SARO is able to match underlying networking topology and to adapt to dynamically changing network conditions by sampling members of its random subset once per configurable epoch.
- A large-scale evaluation of 1000 SARO nodes in an emulated 20,000 node network topology confirm the scalability and adaptivity of our approach.
- We use an existing streaming media player to transmit live streaming media using SARO running on top of the PlanetLab Internet testbed.

Acknowledgments

We thank our shepherd, Steve Gribble, for helping us refine the paper and David Becker for his help with ModelNet. We also thank Rebecca Braynard and anonymous reviewers for their helpful comments.

References

[1] A.J.Ganesh, A.-M Kermarrec, and L. Massoulie. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International workshop on Networked Group Communication*, 2001.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[3] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, August 2002.

[4] Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transaction on Computer Systems*, 17(2), May 1999.

[5] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *Proceedings of ACM SIGCOMM*, August 2002.

[6] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[7] Yatin Chawathe, Steven McCanne, and Eric A. Brewer. RMX: Reliable multicast for heterogeneous networks. In *INFOCOM (2)*, pages 795–804, 2000.

[8] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[9] Michael Dahlin. Interpreting Stale Load Information. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1999.

[10] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic Algorithms for Replicated Database Maintenance. *Operating Systems Review*, 22(1):8–32, 1988.

[11] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, 2001.

[12] Paul Francis. Yoid: Extending the Internet Multicast Architecture. Technical report, ICSI Center for Internet Research, April 2000.

[13] David A. Helder and Sugih Jamin. End-host multicast communication using switchtrees protocols. In *Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, 2002.

[14] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[15] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.

[16] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[17] Brian Noble, M. Satyananarayanan, Giao Nguyen, and Randy Katz. Trace-based Mobile Network Emulation. In *Proceedings of SIGCOMM*, September 1997.

[18] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[19] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[20] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[21] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication*, November 2001.

[22] Sylvia Ratnasamy, Paul Francis Mark Handley, Richard Karp, and Scott Shenker. A Content Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[23] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-scale Event Notifi cation Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[25] SHOUTcast. `http://www.shoutcast.com`.

[26] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[27] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflcts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[28] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[29] Beichuan Zhang, Sugih Jamin, and Lixia Zhang. Host Multicast: A Framework for Delivering Multicast To End Users. In *Proceedings of INFOCOM*, 2002.

[30] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[31] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.