An Evaluation of Scalable Application-Level Multicast Built Using Peer-To-Peer Overlay Networks

Abstract

Peer-to-peer overlay networks such as CAN, Chord, Pastry, and Tapestry can be used to implement Internet-scale application-level multicast. There are two general approaches to accomplishing this: tree building (used by Bayeux and Scribe); and flooding (used by CAN Multicast). This paper presents an evaluation of these two approaches using CAN and Pastry as the underlying overlay networks. We chose CAN and Pastry as representatives of the two major kinds of peer-to-peer overlays. In particular, Chord, Pastry and Tapestry all use a form of generalized hypercube routing with longest-prefix matching, whereas CAN uses a numerical distance metric to route through a Cartesian hyper-space.

To the best of our knowledge, this paper reports the first head-to-head comparison of CAN-style versus Pastry-style overlay networks, using multicast communication workloads running on an identical simulation infrastructure. Furthermore, we observe that the approach to multicast is independent of overlay network choice, and we then provide a head-to-head comparison of flooding versus tree-based multicast on both styles of overlays.

We find that the tree-based multicast approach consistently outperforms the flooding approach. We show that the biggest disadvantage of the flooding approach is the cost of constructing a new overlay for each multicast group. Finally, for the tree-based multicast, we show that Pastry can optionally provide higher performance than CAN but at a higher cost.

1 Introduction

Scalable multicast has long been considered a desirable facility to have. Unfortunately IP multicast is not yet widely deployed and there are questions concerning whether it ever will be. This has led in recent years to an interest in application-level multicast [8, 7, 6, 13]. Recently, a number of application-level multicast systems have been proposed that are built using generic peer-to-peer routing infrastructures and that claim to support large numbers of members in a highly scalable manner. Examples include Bayeux [17], CAN-Multicast [10] and Scribe [12].

Each of these examples employs a different routing infrastructure underneath and implements application-level multicast using either a flooding (CAN-Multicast) or a tree-building (Bayeux and Scribe) approach. Each of the routing infrastructures—also referred to as overlay networks—provides similar functionality, but does so in a different manner. Roughly speaking, they fall into two categories: One category, which includes Chord [14], Pastry [11] and Tapestry [16]), uses a generalized hypercube routing algorithm with longest-prefix address matching, while the other category, which currently only includes CAN [9], uses a numerical distance metric to guide routing through a Cartesian hyper-space.

The flooding, or overlay-per-group, approach to providing multicast creates a separate overlay network for each multicast group and leverages the routing information that a node must already know about other nodes in the overlay network to provide broadcast functionality. The tree-per-group approach builds a distribution tree for each group along which the multicast messages related to that group are propagated.

To date there has been little attempt to compare the performance of these different application-level multicast systems and the different generic peer-to-peer infrastructures they have been built upon. In this paper we examine the performance aspects of both the flooding and tree-based approaches to application-level multicast, on both types of peer-to-peer infrastructure. The same simulator infrastructure and workloads have been used consistently for all experiments to enable fair comparisons.

We chose to use a Pastry implementation as the representative for longest-prefix matching systems and a CAN implementation as the representative for the d-dimensional Cartesian space approach. Pastry was chosen because we had access to its implementation and because considerable effort has already gone into optimizing its design from a performance perspective. We selected CAN-Multicast and Scribe as the representatives for the flooding and the tree-building approaches to providing application-level multicast. The base mechanism for building trees in Bayeux requires the root of the tree to handle group membership, which leads to a far less scalable system than the approach used in Scribe, which is based on reverse path forwarding.

The results of our explorations demonstrate several things. Principal among these is that a tree-based approach to multicast dominates the flooding over per-group overlays approach regardless of the peer-to-peer overlay network employed. This is true both in terms of delivery performance and in general overhead. The biggest disadvantage of per-group overlays—and therefore of flooding—is the cost of overlay construction required for each group. We also showed that multicast trees built using Pastry can optionally provide higher performance than ones built using CAN but at a higher cost.

In the rest of this paper we provide a basic description of CAN and Pastry in Section 2. The overlay-per group and tree-per-group approaches to application-level multicast implementations are presented in Section 3. Section 5 explores a wide spectrum of configurations and present the results of detailed simulation experiments to compare the performance of overlay-per-group and tree-per-group for respectively CAN and Pastry. We compare the strength and weaknesses of various approaches in Section 6. Related works are reported in Section 7. Section 8 presents our conclusions.
2 Peer-to-Peer Overlay Networks

Generic peer-to-peer routing infrastructures (e.g., CAN [9], Chord [14], Pastry [11] and Tapestry [16]) provide efficient routing over an abstract namespace. Each node represents a point or portion of the namespace. Given a message with a key, a node routes that message to the destination node which is numerically closest to the key. Two main kinds of peer-to-peer infrastructures exist and differ by the routing algorithm they are using. The routing algorithms are either based on longest-prefix matching as in Chord, Pastry, and Tapestry or on a distance metric to progress in a Cartesian hyper-space as in CAN. The different approaches to creating and managing the namespace also lead to different ways in which network locality can be exploited.

In this section, we provide an overview of the peer-to-peer overlays we have chosen for this study as representative of the two main classes, namely CAN and Pastry.

2.1 CAN Overlay Network

The Content Addressable Network (CAN) [9] overlay network design organizes nodes of an overlay into a $d$-dimensional hypercube. Each node takes ownership of a specific hyper-rectangle in the space, such that the collection of hyper-rectangles covers the entire space. Each node tracks who its immediately adjacent neighbors are and routes messages to them. Nodes join the hypercube by routing a join message to a randomly chosen point in the space, causing the node owning that region of space to split its region into two, giving half to the new node and retaining half for itself. Message routing consists of choosing one of the current node's neighbors closer in CAN space to the destination than the current node is, and forwarding the message to that neighbor, repeating this process until the message reaches the node whose region contains the destination address.

Beyond the basic CAN algorithm, described above, CAN adds a number of "knobs" that can be used to improve its routing performance. While these were described in [9], we summarize them here:

**Dimensions:** The number of dimensions of the CAN hypercube.

**Ratio-Based Routing:** Vanilla CAN routes to the neighbor closest to the destination in CAN space. Ratio-based routing examines the ratio between the network delay to each neighbor and the progress made in CAN space by routing to that neighbor, choosing to route to the neighbor with the best ratio of CAN distance progress to network cost.

**Multiple Nodes per Zone:** This knob allows more than one node to inhabit the same hyper-rectangle. CAN delivers messages to any one of the zone inhabitants in an anycast manner.

**Multiple Realities:** This knob allows multiple CAN hypercubes to co-exist at once, with the same nodes occupying each, but with completely different assignments of hyper-rectangles to nodes in each. Messages can switch between realities at each hop. Messages are delivered to a zone containing the destination CAN address in any one of the realities.

**Uniform Partitioning:** If enabled, when a node joins the CAN network, once its join message reaches a node containing its target CAN address, rather than immediately splitting the region in two, all the node's neighbors are examined. If a neighbor's zone is larger than the current zone, the join message is forwarded to the neighbor, which will then apply the same test. Once a local maximum neighbor size is reached that zone is split in two, with the new node obtaining half the split zone.

**Landmark-Based Placement:** Landmark-based placement causes nodes, at join time, to probe a set of well known "landmark hosts", estimating each of their network distances. Each node measures its round-trip-time to the landmark machines, and orders the landmarks from the nearest to the most distant in the underlying network. Nodes with the same landmark ordering are clustered into a bin. Rather than choosing a random CAN addresses at which to join, the CAN space is divided into evenly sized bins, and the CAN join address is then chosen from within the bin area. The effect is that nodes with the same landmark ordering end up closer to each other in CAN space.

One of our earliest steps taken in this work was to produce an independent implementation of CAN that could run in our simulator, which was provided by the Pastry authors. To validate our implementation we then set about reproducing the results presented in the original CAN paper. In nearly all cases we were able to reproduce the same graphs within one or two percent of the values obtained by the CAN authors.

2.1.1 New CAN Features

In the course of our investigations we developed several additional knobs beyond those previously published in an attempt to further improve the achievable CAN performance along several dimensions. We summarize them here:

**Network-Based Routing Metric:** Network-based routing chooses to route a message to the neighbor with least network cost, subject to the message still being closer to the destination. This is like the previously-published Ratio-Based Routing except that only network cost is factored into the routing decisions. As shown in Section 5.3.1, we found the greedy Network-Based Routing to be uniformly better than either Ratio-Based Routing or ordinary CAN routing.

**Corner Neighbors:** We extended the CAN implementation to support routing through corner neighbors. Our motivation for adding corner neighbors was to increase the number of available routing choices. In a traditional CAN, a node is only be considered a neighbor if the coordinate spans overlap along d-1 dimensions and are adjacent along 1 dimension. With our extension, a node will be considered a corner neighbor if the coordinate spans are adjacent along 2 or more dimensions, and overlap along all remaining dimensions. In CAN's with a large number of dimensions, the number of corner neighbors has the potential to grow extremely large. To offset this effect, we implement an option where nodes can randomly select a fixed number of corner neighbors from the set of possible corner neighbors.

**Transit-stub Topological Node Placement:** The basic CAN construction is ignorant of the underlying network topology, namely, two adjacent nodes in the CAN coordinate space may be far from each other in terms of the IP network distance. One of our key goals is to understand how much performance benefit there is to constructing overlays using network topol-
ology information. In addition to the landmark-based placement described above, we constructed a placement technique based on the Georgia-Tech transit-stub Internet topology model supported by our simulator [15]. Here, we divide the CAN space into $T$ equal sized bins, where $T$ is the total number of transit networks in the topology. Within a given bin, we randomly choose CAN addresses for each stub network that attaches to the corresponding transit network. This ensures that all stubs that connect to the same transit network will be relatively near each other in CAN space. Furthermore, nodes connected to the same stub network will also end up close to each other in CAN space.

2.2 Pastry Overlay Network

Pastry is an overlay network that provides a request routing and object location infrastructure. Pastry is fully described and evaluated in [11]; a brief overview is provided here.

Pastry is a peer-to-peer routing substrate that is efficient, scalable, fault resilient, and self-organizing. Pastry assigns random, uniformly distributed nodeIds to participating nodes from a circular 128-bit namespace. Given an objectId, Pastry routes an associated message towards the node whose nodeId is numerically closest to the objectId. Assuming a network consisting of $N$ nodes, the expected number of hops is $[\log_2 N]$ ($b$ is a configuration parameter with typical value 4). Despite concurrent node failures, eventual delivery is guaranteed unless $[l/2]$ nodes with adjacent nodeIds fail simultaneously ($l$ is a configuration parameter with typical value 16 or 32).

The tables required in each node have only $(2^b - 1)$ $[\log_2 N] + 2l$ entries on average, where each entry maps a nodeId to the associated node’s IP address. Moreover, after a node failure or the arrival of a new node, the invariants can be restored by exchanging $O(\log_2 N)$ messages among the affected nodes.

For the purpose of routing, nodeIds and objectIds are thought of as a sequence of digits in base $2^b$. A node’s routing table is organized into $[\log_2 N]$ levels with $2^b - 1$ entries in each level. The $2^b - 1$ entries at level $n$ of the routing table refer to nodes whose nodeIds share the present node’s nodeId in the first $n$ digits, and differ in the $n + 1$th digit. The routing table entry is left empty if no such node is known. The uniform distribution of nodeIds ensures an even population of the nodeId space; only $[\log_2 N]$ levels should be populated in the routing table.

In addition to the routing table, each node maintains IP addresses for the nodes in its leaf set: the set of $l$ nodes that are numerically closest to the current node, with $l/2$ being larger and $l/2$ being smaller than the current node’s id.

At each routing step, a node normally forwards the message to a node whose nodeId shares with the object Id a prefix that is at least one digit (or $b$ bits) longer than the prefix that the object Id shares with the present node’s id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the object Id as long as the current node, but is numerically closer to the object Id than the present node’s id. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId or its immediate neighbor. And, unless $[l/2]$ adjacent nodes in the leaf set have failed simultaneously; at least one of those nodes must be live.

Locality Although Chord, Pastry and Tapestry rely on a similar prefix matching based routing algorithm, they differ in the extent they exploit underlying network locality. Pastry attempts to minimize the distance traveled with respect to the proximity metric. The proximity metric is a scalar value that reflects the “distance” between any pair of nodes, such as the number of IP routing hops, geographic distance, delay, or a combination thereof. It is assumed that a function exists that allows each Pastry node to determine the “distance” between itself and a node with a given IP address.

The way locality properties are achieved in Pastry means that the distance traversed at each Pastry hop exponentially increases, until the average distance between two nodes in the network is reached.

2.3 Commonalities and Differences

Although CAN and Pastry employ different routing algorithms to provide somewhat different functionality they do share several features in common. An important distinction to understand between CAN and Pastry is that Pastry delivers a message tagged with a given key to exactly that single node whose node Id is numerically closest to the key in the namespace. In contrast, CAN delivers such a message to any node in the overlay that belongs to the same zone that the key is contained by. Thus, CAN provides anycast semantics whereas Pastry provides unicast semantics. The provision of anycast instead of unicast semantics has significant implications for the applications that use the overlay network since they are now responsible for maintaining consistency between state maintained on all nodes belonging to the same anycast equivalence class.

Another difference is that Pastry has far fewer optimization “knobs” to manipulate than CAN does. Whereas both CAN and Pastry employ network locality-aware routing metrics to select from a set of otherwise equivalent routing table entries, CAN also introduces the notions of multiple nodes-per-zone, multiple realities, uniform partitioning, corner neighbor selection, landmark-based placement, and topology-based placement. Although Pastry does not employ topology-aware node ID assignment, it is a concept that can be considered for use in the same manner that CAN employs topology-aware address assignment.

3 Overlay-Based Application-Level Multicast

Two approaches have been taken to implementing application-level multicast on peer-to-peer overlays: flooding and tree creation. Flooding leverages the information that a node must already know about other nodes in the overlay to provide broadcast functionality. Therefore, if most nodes participating in an existing overlay network are interested in receiving a broadcast message this potentially provides a cheap way of propagating it. However, for groups consisting of a small subset of the overlay network’s membership, it is not efficient to broadcast the message to the entire overlay network and mini-overlay networks have to be separately constructed and
utilised instead. One advantage the flooding approach has is that the only nodes participating in the dissemination of a broadcast message are group members.

The alternative tree-based approach builds a tree for each group instead of a separate overlay network. Multicast messages related to a group are propagated through its associated forwarding tree. This form of application-level multicast is leveraging the object location and routing properties of the overlay network to create groups and join groups. The application then creates and manages the tree and uses it to propagate messages. There are several possible ways to build such trees [17, 12].

3.1 Overlay-Per-Group Application-Level Multicast Implementations

Multicasting by means of flooding to separate overlay networks for each group has certain features that are independent of the choice of overlay network employed. In particular, clients wishing to join a group must first find the overlay that represents the group. To implement this lookup function in a scalable manner requires a distributed name service. For our experiments we implemented this functionality by means of a separate global overlay network that is used to implement a distributed hash table. Both CAN and Pastry support this capability very naturally.

3.1.1 CAN Flooding

The flooding algorithm we implemented for CAN is based on the algorithm described in [10], with a few modifications. We begin by summarizing the published algorithm, and then we present our modifications.

The simplest flooding strategy for a CAN network has each node forward an incoming message to all of its neighbors, and all nodes cache the message-ids of previously received messages to filter out duplicates. The problem with this strategy is that it can lead to a large number of duplicate messages. To reduce the number of duplicates, CAN uses a directed flooding algorithm that is based on the structure of the CAN coordinate space. Each node uses the following five rules to decide whether to forward a message, and to decide to which neighbors to forward the message.

1. A node caches the message-ids of all received messages. When a node receives a duplicate, it does not forward the message.
2. The source node forwards the message to all of its neighbors.
3. A node receives a message from a neighboring node adjacent along dimension i. The node forwards the message to all neighbors adjacent along dimensions 1 through i-1. The node also forwards the message to those neighbors adjacent along dimension i in the opposite direction from the node it received the message from.
4. A node does not forward a message along a particular dimension if that message has already traveled at least half-way across the space from the source coordinate in that dimension.
5. Along dimension 1, a node N only forwards to a neighbor A if a specific corner C of A is in contact with N. C is defined to be the corner of A that is adjacent to N along dimension 1 and has the lowest coordinates along all other dimensions. Note that this rule eliminates certain messages that would otherwise be sent according to rules 2 and 3.

Improvements To CAN Flooding We made two changes to the published CAN flooding algorithm. The first adds an additional constraint to rule 4 to further reduce the number of duplicate messages. When deciding whether to forward to a neighbor A, if A contains the point that is half-way across the space from the source coordinate in that dimension, then it only forwards to the neighbor from the positive direction.

The second change we made is needed for correctness. As described above, the algorithm is vulnerable to race conditions that can lead to certain nodes never receiving a multicast message. These race conditions arise because nodes take into account both the dimension and direction along which they receive a message when deciding to which neighbors to forward that message. Due to space constraints, we omit a detailed example of one race and provide it in a separate technical report.

The idea behind our change to the flooding algorithm is to make static forwarding decisions based on the relative position of a node to the multicast source, rather than dynamic decisions based on the dimension and direction of incoming messages. The new algorithm breaks up the forwarding process into two stages. In first stage, a node decides which dimensions and directions to the forward message along. In the second stage, a node applies a second set of rules to filter the subset of neighbors that satisfy the first stage rules. The stage one rules are:

1. If a node’s region overlaps the source along all dimensions less than or equal to i, then forward the message in both the positive and the negative directions along dimension i.
2. If a node’s region overlaps the source along all dimensions less than i, then forward the message in one direction along i. The direction to forwarding the message should be away from the source coordinate, towards the halfway point.
3. For dimension 1, always forward in one direction. As before, the forwarding direction should be away from the source coordinate, towards the halfway point.

The stage two rules are:

1. For dimensions 2 and greater, only forward to a neighbor if that neighbor’s region overlaps the source coordinates for all dimensions less than i.
2. The modified rule 4 described above.
3. Rule 5 from the original algorithm.

Although the rules for this algorithm look rather different from the original, the way that messages flow through the CAN is quite similar to the original algorithm. An important side effect of the modified flooding algorithm is a significant reduction in the number of duplicate messages, due to the first rule in stage two of the new algorithm.
3.1.2 Pastry Flooding

The Pastry broadcast algorithm uses the entries in the each nodes routing table to control propagation of messages. A node wishing to broadcast a message forwards it to all the nodes present in its routing table, tagging the message with the level \( l \) of the destination node in the routing table. Each node receiving the message forwards it to all nodes in its routing table starting from level \( l + 1 \), replacing the tag associated with it with the level \( l \) of the destination node in the routing table. This is repeated until the nodes receiving the message have no entries at level \( l + 1 \).

At any stage a node may have missing entries in its routing table. If a missing entry is detected at level \( l \) in domain \( d \) then this may lie within the leafset of the node. If so, nothing else is done, otherwise the message is routed using Pastry to a key representing the midpoint of level \( l \) with domain \( d \), with the tag set to level \( l \) and marked as a midpoint request.

A node receiving a midpoint request will be the numerically closest. If the nodeId does not have a longer prefix match than \( l \) and no node in its leafset matches the key with a longer prefix than \( l \) then it discarded the message (it will be a duplicate). If it does have a longer match then it forwards the message to all entries in its routing table which are at level \( l + 1 \) or deeper. If it has no longer match, but a member of the leafset does, the node forwards the request to that node. It may have already received the message but with a larger value of \( l \), in which case the node must forward it to the levels between \( l \) and the level it has already forwarded the message to.

4 Tree-Per-Group Over Single Overlay Implementations

As a representative example of tree-based multicast we use the approach used in Scribe, and full details can be found in [12]. Scribe is a generic multicast application originally implemented using Pastry and uses reverse path forwarding [3] to build a multicast tree per group. The tree is formed by the union of the overlay routes from the group members to the root of the multicast tree.

Each group has a key called the groupId. This could be, for example, a hash of the group’s textual name concatenated with its creator name. The node responsible for or owning the overlay address closest to the groupId is the root of the tree.

To join a group, a node routes a message through overlay using the group’s groupId as the key. This message is routed towards the root of the tree. Each node along the route checks whether it is already a forwarder for that group. If it is, it registers the source node as a child in the multicast tree and stops routing the message any further. Otherwise, it creates an entry for the group, adds the source node as a child and this node attempts to join the group. Nodes that act as nodes in the tree are called forwarders, and may or may not be members of the group.

Scalability is ensured by the fully decentralized design. In particular, use of randomization of overlay addresses ensures that the tree is well balanced. The load of disseminating messages within a group is distributed among the forwarders of the multicast tree. Should a node decide that the load on it is too high, some of the children of this node can be made grandchildren, by the overloaded node passing some of its children to its other children. This is done in such a way to minimize the impact on latency and link stress and is described in [12].

Scribe also tolerates both forwarder and root failures. However, a study of reliability properties is out of the scope of this paper and more details about this and Scribe in general are available in [12].

Whereas Scribe maps onto Pastry in a straight-forward manner, it requires the addition of replicated state coordination when implemented on top of CAN. This is because messages routed with CAN will be directed to any node that is a co-habitant of the zone that the groupId is part of. Furthermore, a message can end up at any (single) node that owns the relevant zone in any of the realities that the CAN is maintaining.

5 Evaluation

5.1 Experimental Setup

We used a simple packet-level, discrete event simulator to evaluate the different multicast implementations. The simulator counts the number of packets sent over each physical link and assigns a constant delay to each link. It does not model either queueing delay or packet losses because modeling these would prevent simulation of large networks.

The simulations ran on a network topology with 5050 routers, which was generated using the Georgia Tech [15] random graph generator according to the transit-stub model. These routers did not run the code to maintain the overlays and implement application-level multicast. Instead, this code ran on 80000 end nodes that were randomly assigned to routers in the core with uniform probability. Each end system was directly attached by a LAN link to its assigned router (as was done in [6]).

The transit-stub model is hierarchical. There are 10 transit domains at the top level with an average of 5 routers each. Each transit router has an average of 10 stub domains attached, and each stub has an average of 10 routers.

We used the routing policy weights generated by the Georgia Tech random graph generator [15] to perform IP unicast routing. The delay of each LAN link was set to 1ms and the average delay of core links (computed by the graph generator) was 40.5ms.

We ran two sets of experiments. The first set ran with a single multicast group and all the overlay nodes were members of the group. These experiments provide a simple setting to evaluate different ways to implement the overlays and application-level multicast on top of them. In particular, we evaluated tradeoffs between the amount of state maintained by each overlay node and routing efficiency and between different ways of taking advantage of network locality to improve routing performance.

The second set of experiments ran with a large number of groups (1500) and with a wide range of membership sizes. Since there are no obvious sources of real-world trace data to drive these experiments, we adopted a Zipf-like distribution for the number of members to each group. The number of members of a group is defined by \( [N \cdot r^{-1.25} + 0.5] \), where \( r \) is the rank of the group and \( N \) is the number of nodes. The
actual group members were selected using both a uniform distribution and a distribution where group members were likely to be close in the network topology. These allow us to evaluate the ability of the different implementations to concurrently support multiple applications with varying requirements.

Both sets of experiments were divided in two phases: first all group members subscribed to their groups, and then a message was multicast to each group. To provide a fair comparison, we were careful to ensure that each group contained the same set of end nodes and that the sender was the same end node in the experiments ran on different implementations.

5.2 Evaluation Criteria

We used several metrics to evaluate the different application-level multicast implementations. These metrics evaluate the delay to deliver multicast messages, the load on the network, and the load imposed on end nodes. The metrics are described in more detail below.

Relative Delay Penalty. Using application-level multicast increases the delay to deliver messages relative to IP multicast. To evaluate this penalty, we measured the distribution of delays to deliver a message to each member of a group using both application-level and IP multicast. We compute two metrics of delay penalty using these distributions: $RMD$ is the ratio between the maximum delay using application-level multicast and the maximum delay using IP multicast, and $RAD$ is the ratio between the average delay using application-level multicast and the average delay using IP multicast. These metrics avoid the anomalies [9, 12] associated with the method used to compute the delay penalty in [6].

Link Stress. Application-level multicast also increases the load on the network relative to IP multicast. We evaluated the load on the network using the link stress metric described in [6]. We measured the stress of each directed link in the network topology by counting the number of packets sent over the link. The stress was measured both during the phase when members join the group and during the phase when messages are multicast.

Node Stress. In application-level multicast, end nodes are responsible for maintaining routing information and for forwarding and duplicating packets whereas routers perform these tasks in IP multicast. To evaluate the stress imposed by application-level multicast on each node, we measured the number of nodes in each node’s routing tables and the number of messages received by each node when members join the groups. The first metric is both a proxy for the amount of routing information maintained by each node, the cost of maintaining that information, and the number of messages sent by the node.

Duplicates. Some of the application-level multicast implementations that we evaluated generate duplicate messages that both waste network resources and increase load on end nodes. We measured the number of duplicates received by end nodes.

5.3 CAN Results

5.3.1 Parameters

CAN has a very large number of parameters that can be used to tune its performance. We performed an extensive exploration of this parameter space attempting to understand which combinations of parameters lead to the best RAD values for unicast communication. We varied the following parameters during our exploration: the number of dimensions; the number of nodes per zone; the number of realities; the number of node instances; turning on and off uniform partitioning; turning on and off routing through corners; and choosing a node assignment policy from random, topological, or landmark based; and choosing a routing policy from CAN distance, CAN ratio, or network distance. To allow direct comparisons between different CAN configurations, we measured the average amount of neighbor state that each node maintains and only compared instances of CAN that used similar amounts of neighbor state. The importance of neighbor state is not the actual memory overhead of neighbor lists, but the communication overhead that it represents.

Our primary conclusion is that the CAN parameter space is difficult to navigate. Configurations that work well with one state budget do not scale up or down in linear fashion. Therefore, discovering the best configuration with 50 neighbors does not provide immediate insight into which parameters will give the best configuration with 100 neighbors. To illustrate the difficulties encountered in exploring the CAN parameter space, Figure 1 shows how varying a single parameter, the number of landmarks, can lead to interesting nonlinear behavior in terms of RAD. Nonetheless, we can provide some general guidelines. Enabling some form of topological assignment either landmark-based or transit-stub-based provides the largest improvement in RAD out of all the CAN parameters. Enabling uniform partitioning often provides a significant reduction in terms of the neighbor state overhead, especially for the two topological assignment schemes where nodes often end up clustered close together in the CAN space. Furthermore, uniform partitioning never causes RAD to become significantly worse. The network distance routing met-
Table 1: Representative good configurations, as a function of neighbor state, for an 80,000 node CAN.

<table>
<thead>
<tr>
<th>State</th>
<th>Dimensions (d)</th>
<th>Nodes Per Zone (z)</th>
<th>Realities</th>
<th>Uniform Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>enabled</td>
</tr>
<tr>
<td>29</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>enabled</td>
</tr>
<tr>
<td>38</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>enabled</td>
</tr>
<tr>
<td>59</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>enabled</td>
</tr>
<tr>
<td>111</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>enabled</td>
</tr>
</tbody>
</table>

Figure 2 shows the effect on delays of using topology-aware routing tables and topology-aware address assignment for the d=12, z=3 CAN configuration. All other CAN configurations that we ran our flooding-based multicast experiments on showed the same effects as are evident for this one.

Several interesting things are noticeable: We confirmed that the benefit of topology-aware address assignment translates over from unicast communications to flooding-based communications. In all cases, topology-aware address assignment proved to dominate random address assignment. Improvements of up to 30% were observed. In contrast, the choice of routing metric did not have much effect. All variation was under 10%.

Figure 3 plots the delay penalty of “best” representative CAN configurations at each routing table state size. The best representative was considered to be one in which topologically-aware address assignment takes place and the routing metric used is the network distance of a neighbor.

For flooding-based multicast, the benefit to be gained from increased amounts of routing table state is uneven. Increasing the size up to 60 actually yielded poorer delay penalty values, whereas going beyond that up to 110 then yielded somewhat better delay penalty values. The difference between best and worst values was about 20%. The independence of delay penalty to routing table state size is unsurprising when one considers that the flooding algorithm that CAN employs cannot take advantage of most of the optimizations that CAN employs.

We also evaluated link stress results for both the subscription and the publication phases of multicasting. As with delay penalty, topological address assignment yielded uniformly better link stress numbers and the choice of routing metric did not make a significant difference. Thus, Table 2 shows both maximum and average link stress values only for the best representative CAN configurations at each routing table state size.

The most noticeable feature in the table is that the link stress caused by having 80,000 members join a multicast group is huge and grows significantly as a function of the amount of routing table state the CAN maintains. In contrast, the link stress caused when a multicast message is sent to 80,000 group members is considerably smaller and mostly drops as a function of routing table state size.

The rise in link stress values as a function of routing table state size during subscription is due to the increased neigh-

5.3.2 Flooding-Based

In this section we explore the behavior of flooding on CAN. We start by describing the impact of the various parameters on delays. Figures 2 and 3 plot the delay penalty of flooding relative to IP. Both the ratio between the maximum delays (RMD) and the ratio between the average delays (RAD) are shown.

Figure 3: Relative delay penalty for flooding an event to 80,000 nodes.
Table 2: Link stress for flooding in CAN.

<table>
<thead>
<tr>
<th>State size</th>
<th>19</th>
<th>29</th>
<th>38</th>
<th>59</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joining phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>107646</td>
<td>134378</td>
<td>143454</td>
<td>398174</td>
<td>480926</td>
</tr>
<tr>
<td>Average</td>
<td>151</td>
<td>182</td>
<td>217</td>
<td>276</td>
<td>424</td>
</tr>
<tr>
<td>Flooding phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>1737</td>
<td>1897</td>
<td>1343</td>
<td>958</td>
<td>652</td>
</tr>
<tr>
<td>Average</td>
<td>3.42</td>
<td>4.13</td>
<td>3.35</td>
<td>3.06</td>
<td>2.94</td>
</tr>
</tbody>
</table>

Table 3: Link stress for tree-based multicast for representative “best” CAN configurations at each routing table size.

<table>
<thead>
<tr>
<th>state size</th>
<th>30</th>
<th>40</th>
<th>60</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>219</td>
<td>178</td>
<td>188</td>
<td>202</td>
</tr>
<tr>
<td>Average</td>
<td>1.52</td>
<td>1.42</td>
<td>1.35</td>
<td>1.31</td>
</tr>
</tbody>
</table>

As with the flooding-based results, we observed that topologically-aware address assignment dominated random address assignment. Improvements of up to 80% were observed. However, unlike in the flooding-based results, the choice of routing metric does matter: routing based purely on CAN hyper-space distance does noticeably worse than routing based on network distance or routing based on the ratio of CAN distance to network distance. Of the latter two routing metrics the purely network distance-based metric consistently performed slightly better than the ratio-based metric, as was the case for unicast.

Although there is benefit to be gained from moving to increasing amounts of routing table state, the benefit is not major. The largest improvement observed was 40%.

Overall, we observe that tree-based multicast seems to outperform flooding-based multicast on CAN by a factor of two to three with respect to delay penalties.

Table 3 shows both maximum and average link stress values for the best representative CAN configurations at each routing table state size. As with delay penalty, topological address assignment yielded uniformly better link stress numbers than did random address assignment. However, the improvement in link stress numbers is far more dramatic than the improvement seen for delay penalties. The typical improvement for maximum link stress was about a factor of 6, while the typical improvement for average link stress was about a factor of 2 to 3. In contrast, the choice of routing metric made a small difference, with purely network distance-based routing doing the best.

Unlike with flooding-based multicast, the link stresses during the subscription and publishing phases are essentially the same and we show only the latter. Furthermore, whereas there is a noticeable difference in link stress values for flooding-based multicast as the amount of routing table state is increased, there is only a relatively minor change in values for tree-based multicast.

Finally, we examined the size of the forwarding tables that tree-based multicast must maintain on each node and how many nodes must act as forwarders for broadcast messages. Figure 6 shows the maximum number of entries seen on any node as a function of routing table state size, while Figure 7 shows the number of nodes having to act as forwarders as a function of routing table state size. As the size of the routing tables increases, we see that individual nodes risk suffering a concomitant increase in the maximum number of forwarding
We studied the impact of several Pastry parameters on the performance of both implementations of application-level multicast.

We varied the value $b$ from 1 to 4. Recall that $b$ is the number of bits of the destination key that Pastry attempts to match at each hop. A small value of $b$ reduces the amount of space used in routing tables at the expense of an increase on the expected number of Pastry hops to reach a destination.

We also evaluated two orthogonal optimizations that take advantage of topology information to improve routing performance in Pastry: topology-aware routing table construction (TART), and topology-aware nodeId assignment (TOP). TART is similar to NDR in CAN, however, TART controls the actual entries in a nodes routing table (so they have good network locality), whereas, NDR optimizes the choice of node in CAN, but does not control which nodes a particular CAN node is aware of.

Pastry uses topology-aware routing table construction as described in Section 2.2: nodes probe each other to estimate the delay between them and these topological distance estimates are used to optimize routing tables to achieve a low delay penalty. We ran experiments with and without this optimization. For each slot in the routing table, Pastry normally chooses a topologically close node whose nodeId satisfies the constraints for that slot. Without the optimization, it chooses a random node with uniform probability from the set that satisfies the constraints.

The current version of Pastry does not use topology-aware nodeId assignment. NodeIds are assigned randomly with uniform probability from a $2^{128}$-bit name space. This is important for reliability because it ensures that the nodes in each leaf set are randomly scattered over the network. Therefore, they are more likely to fail independently. However, topology-aware nodeId assignment could potentially improve performance. To evaluate its benefits, we ran a version of Pastry where nodes with numerically close nodeIds are topologically close. We did this by assigning each node a name obtained by concatenating the identifiers of its transit domain, the transit node its stub attaches to, the stub its LAN attaches to, and the actual stub node the LAN is attached to. Then, we sorted the overlay nodes using their name and assigned random nodeIds to each one such that the ordering of nodeIds matched the ordering of names. We should note that this is not a practical implementation because it uses global knowledge and assumes a fixed population of overlay nodes. We use it to show that even this near-perfect, topology-aware nodeId assignment has significant problems that outweigh its benefits.

When we refer to Pastry without qualification, we mean the version of Pastry with random nodeId assignment (RAND) and with TART.

### 5.4.2 Flooding-Based

We evaluated the impact of varying $b$ and using topology-aware optimizations on the performance of flooding on Pastry.

We start by describing the impact of the various parameters on delays. Figure 8 plots the delay penalty of flooding relative to IP for different values of $b$. It shows that both the ratio between the maximum delays (RMD) and the ratio between the average delays (RAD) decrease when $b$ increases: RMD is 70% lower with $b = 4$ than with $b = 1$, and RAD is 50% lower. This happens because increasing the value of $b$ decreases the number of Pastry hops, which is approximately equal to $\log_{2^3}(N)$.

Figure 5.4.2 shows the effect on delays of using topology-aware routing tables (TART) and topology-aware nodeId assignment (TOP) with $b = 4$. It shows that both optimizations are very effective at reducing the delay penalty relative to the version of Pastry without topology-aware optimizations (Pastry with RAND and without TART): TOP reduces the RAD
Relative Delay Penalty

Figure 8: Relative delay penalty for flooding in Pastry for different values of $b$.

Relative Delay Penalty

Figure 9: Relative delay penalty for flooding in Pastry with and without topology-aware optimizations for $b = 4$.

by a factor of 2.2 while TART is more effective and reduces RAD by a factor of 2.4. Combining both optimizations reduces RAD by a factor of 3.5.

We also evaluated the effect of the various parameters on the link stress induced by flooding. Our results show that increasing the value of $b$ increases both the maximum and average link stress but by a relatively small amount: the average link stress with $b = 4$ is 16% higher than with $b = 1$. This increase is expected because increasing $b$ increases the number of entries in each level of the routing table and, therefore, the number of messages sent by each forwarding node.

Table 4 shows the impact of the topology aware optimizations on link stress. TOP reduces the average link stress by more than a factor of 3 and the maximum link stress by more than a factor of 30. TOP is very effective at reducing the link stress because the assignment of nodeIds matches the network hierarchy. At the top level forwarding sends a few messages to a small number of nodes in each top level transit domain. These messages travel a long IP distance but there is only a small number of them. The number of messages increases exponentially as one moves down the routing tables during flooding but these messages travel increasingly shorter IP distances. Therefore, most messages travel over a small number of links and the resulting link stress is low.

On the other hand, TART reduces the average link stress slightly but it increases the maximum link stress. The reduction in link stress is a result of the reduced number of links traversed on average by each message. However, the distance traversed by messages increases as one moves down the forwarding tree. Therefore, the link stress reduction is not as significant as with TOP because most messages traverse relatively large distances. The maximum link stress increases because the nodes that flood from the levels at the top of the routing table, which are the ones that send the most messages, are likely to be in the stub of the sender using TART. This causes the link from the sender’s stub to its transit to have a high link stress.

Figure 10 shows the average number of unique nodes in a Pastry node’s routing table and leaf set. This number increases with $b$ as expected; it is approximately equal to $(2^b - 1) \log_2(N) + L'$, where $L'$ is the small number of nodes that are in Pastry’s leaf set but not in the routing table. The average number of nodes is less than 65 and the maximum number of nodes in only 78 even when $b = 4$. The topology-aware optimizations have no measurable effect on these numbers.

Finally, we counted the number of duplicate messages received by overlay nodes. These results appear in Table 5. The duplicates are due to missing entries in Pastry routing tables. Since the probability of missing entries increases with $b$, the number of duplicates also increases with $b$. The number of duplicates with $b = 4$ is large (approximately 10%) but we can repair the routing tables at low cost to prevent almost all duplicates in subsequent multicasts. The topology-aware optimizations have no measurable effect on the number of duplicates.

<table>
<thead>
<tr>
<th>$b$</th>
<th>#duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>1305</td>
</tr>
<tr>
<td>4</td>
<td>7228</td>
</tr>
</tbody>
</table>

Table 5: Number of duplicate messages received.
5.4.3 Tree-Based

Next we present results of experiments to evaluate the impact of varying $b$ and using topology-aware optimizations on the performance of tree-based multicast on Pastry.

Figure 5.4.3 shows that the delay penalty decreases when $b$ increases. This is similar to what we observed with flooding and it is also explained by a reduction in the average number of hops in Pastry routes.

The effect of the TART and TOP optimizations on the delays with tree-based multicast is also similar to what we observed for flooding. These results are shown in Figure 5.4.3.

Table 6 shows the impact of the TART and TOP optimizations on link stress during multicast sends. The link stress during group joins is almost identical for the tree-based approach. These results are quite different from what we observed with flooding. TOP reduces average link stress slightly but it increases the maximum link stress. Whereas, TART reduces both the maximum and average link stress significantly.

The reason is that in tree-based multicast the messages follow the reverse of the Pastry routes from the root of the tree to its subscribers. With TART and RAND, the longest hops will be at the top of the tree. Therefore, most messages will travel over a short number of IP hops and the link stress will be low. With TOP, the longest hops are at the bottom of the tree. Therefore, most messages will travel over a long number of IP hops and link stress will be high. Combining TART and TOP reduces the average link stress but it increases the maximum link stress significantly. This is because of a bad interaction between TOP and the node joining algorithm used in TART. This interaction causes information about new nodes that join the network to be propagated only among the nodes with numerically close nodeIds. This results in a large number of nodes with pointers to the same representative in a domain.

Decreasing $b$ reduces both the average and maximum link stress, for example, with $b = 1$, TART, and RAND, the maximum link stress is 243 and the average is 1.06.

The tree-based multicast scheme adds a forwarding table per node. Figure 5.4.3 shows the maximum number of forwarding table entries per-node for different values of $b$ in Pastry.

5.4.4 Discussion

It is clear from the results that flooding only works well in Pastry when using topology-aware nodeId assignment. On the other hand, topology-aware nodeId assignment performs poorly when using tree-based multicast. Tree-based multicast performs better with topology-aware routing tables and random nodeId assignment.

Flooding cannot support multicast efficiently on Pastry because it requires creation of a separate overlay per group. Creating a separate Pastry overlay is significantly more expensive than creating a tree and it induces a large load on the node that is responsible for the group in the base overlay. A tree-based approach can reuse the same overlay for many groups and amortize the cost of creating an overlay with good locality properties using topology-aware optimizations. Therefore, the choice for Pastry is clear: use tree-based multicast with random nodeId assignment and topology-aware routing tables.

We chose a value $b = 4$ because it provides a good balance between low delay penalty and low link stress while requiring only a small amount of space per node. The remaining experiments ran with these choices.
5.5 CAN/Pastry Results for Multiple Multicast Groups

This section describes the set of experiments we ran with 1500 multicast groups instead of just one. Because of memory limitations in our simulator we could only run simulations for systems containing 50,000 nodes instead of 80,000. The group sizes varied according to a Zipf-like distribution, as described earlier. We explored two cases: uniformly distributed group members and members, for a given group, that are mostly localized within the network. The degree of locality of group members was determined by a Zipf-like distribution as well.

We ran these experiments against three different application-level multicast implementations: tree-based multicast on top of Pastry and on top of CAN, and flooding on top of CAN. For each multicast implementation we picked the “best” configuration for the overlay network used. In particular, for Pastry we set \( b \) to 4 and used TART and RAND. For CAN we used configurations with topological address assignment and a network distance-based routing metric.

Figures 14 and 15 show the cumulative distribution function for RMD for, respectively, the non-localized and localized group members cases as a function of the proportion of groups. Figures 16 and 17 show the corresponding cumulative distribution function for RADP. The y-value of a point represents the proportion of groups with a delay penalty less than or equal to the point’s x-value.

In all cases tree-based multicast on top of Pastry yielded the best delay penalty values, with the differences being roughly comparable to those observed in the single-group experiments. Aside from this, the most notable feature is the noticeably shallower shape of the CDF curve for the CAN-based flooding implementation. Thus, whereas tree-based users can expect fairly tightly bounded delay penalties, users of the flooding approach must be prepared to deal with substantially greater variances in multicast delivery times.

Interestingly, while flooding did considerably worse than tree-based in the single-group experiments, it compares much more favorably in the 1500 group experiments. Although it still does worse than tree-based on CAN for RMD with non-localized group members, it does better for the other three cases depicted. An initial hypothesis that flooding may be favored more than tree-based by small size groups has not been born out by experiments we performed with small size single groups.

6 Comparisons and Tradeoffs

Per-Group Overlays versus Per-Group Multicast Trees

Our results show that per-group multicast trees have several advantages over flooding using a mini-overlay network per group. If using CAN, per-group multicast trees are noticeably more efficient in their network usage, with relative delay penalties typically better by factors of 2-3. If using Pastry, the benefit is less pronounced: RMD is about 60% better while RAD is only about 10% better. Probably more significantly, the creation of individual overlays per group incurs significant overheads. When a node joins an overlay network it needs to discover other nodes in the network and, in CAN create forwarding entries that had to be maintained under CAN was

![Figure 14: CDF for RMD for 1500 concurrent multicast groups with localized group members.](image)

![Figure 15: CDF for RMD for 1500 concurrent multicast groups with globally distributed group members.](image)
about a third of that seen with Pastry. However, if Pastry were to use a $b$ value of 1 instead of 4 then it can achieve comparable costs to CAN, albeit with a comparable increase in delay penalties incurred. Overall, we conclude that multicast trees built using Pastry can optionally provide higher performance than ones built using CAN but at a higher cost.

An interesting difference between using tree-based multicast on top of CAN or Pastry is that topologically-aware address assignment turns out to have a diametrically opposite effect on link stress in each case. Topologically-aware address assignment drastically improves link stress values in CAN while doing exactly the opposite in Pastry.

**Complexity of CAN Parameters and Features** CAN has a large number of parameters and features which require tuning to deliver the "best" performance. The tuning of these was found to be time-consuming and the results often unpredictable. The space of obtained values versus CAN parameter settings is by no means smooth or predictable. Whereas for any particular application usage pattern it should be possible to determine a optimal set of settings, we expect such settings will likely only be optimal for that usage pattern.

**Topologically Aware Assignment** Topologically-aware node placement, whether based on direct knowledge of the underlying network topology, or based on a Landmark scheme, provides the greatest impact on CAN’s performance compared to other features. This means that, if possible, when using CAN topologically-aware node placement should be used. However, very little experience is available concerning the real-world costs of implementing this potentially expensive feature. Arguably this represents one of the most important open areas of future research for CAN-based overlay networks.

In Pastry the benefits of topologically-aware nodeId assignment are much more mixed. Its primarily benefit is for flooding over per-group overlays, which we have argued against using. For per-group multicast trees, it does not provide any significant performance increase while incurring substantial link stress penalties.

### 7 Related Work

Two other active research projects are also exploring scalable routing via general-purpose overlay networks: Tapestry [16] and Chord [14]. Both of these systems, like Pastry, use a generalized hypercube routing algorithm with longest-prefix matching. However, although Chord, Pastry and Tapestry rely on a similar routing algorithm, they differ in the extent they exploit underlying network locality. A head-to-head comparison of Chord and Tapestry with Pastry and CAN would be interesting, but was beyond the scope of this work. We do believe that the top-level results from this study are likely to also apply to these systems.

Another proposal for doing multicast using general-purpose overlay networks is the Bayeux [17] system for Tapestry. Bayeux builds a multicast tree per group but this tree is built differently from the approach examined in this paper. Each request to join a group is to the root. Then, the root records the identity of the new member and uses Tapestry to route another message back to the new member. Every Tapestry node along this route records the identity of the new member. Requests to leave the group are handled in a similar way. This introduces two scalability problems when compared to tree based approach used here. Firstly, it requires nodes to maintain more group membership information. Secondly, Bayeux generates more traffic when handling group membership changes. In particular, all group management traffic must go through the root. Bayeux proposes a mechanism to ameliorate these problems, but this only improves scalability by a small constant factor. It should be noted, that if all nodes in a Bayeux network join a single group, then this will produce similar poor results to the flooding approach evaluated on Pastry.

Other systems providing overlay multicast include Overcast [8], Inktomi [7], End System Multicast [6], and the MBONE [13].

A significant amount of work has also gone into overlay networks and application-level multicast systems not designed to scale, such as Resilient Overlay Networks (RONs) [1] and ISIS/Horus-style Virtual Synchrony [2], but which provide other benefits.

Of course, all the work for constructing multicast distribution trees builds upon the techniques originally developed for IP Multicast [4], [5].
8 Conclusion

We have explored some of the possibilities for implementing scalable application-level multicast using peer-to-peer overlay networks. Observing that the style of application-level multicast chosen is largely independent of the style of overlay network selected, we compared four combinations of application-level multicast implementations and peer-to-peer overlay choices. Two approaches to application-level multicast using peer-to-peer overlay networks were considered. One uses reverse path forwarding to build a distribution tree per multicast group. The second builds a separate overlay network per group and uses intelligent flooding algorithms. These approaches were each run on top of two different peer-to-peer overlay networks, each representative of a class of peer-to-peer routing schemes.

All experiment combinations were run on the same simulation infrastructure, using the same placement within the simulated network of nodes participating in the overlay network, and using the same workloads. This enabled us to perform head-to-head comparisons of flooding versus tree-based implementations of application-level multicast, as well as of CAN-style versus Pastry-style overlay routing in the context of multicast communications. To the best of our knowledge, we are the first to have done such a head-to-head comparison.

As part of our exploration we also extended the design of CAN to include several new features, as well as corrected a flaw in the published CAN-multicast algorithm. The new features include a better routing metric based on greedily using estimated network distances to neighbors, the use of some corner-adjacent regions as routing neighbors, and an alternative network topology-aware address assignment scheme that is potentially much cheaper than the current landmark scheme.

The results of our explorations demonstrate several things. Principal among these is that a tree-based approach to multicast dominates the flooding over per-group overlays approach regardless of the peer-to-peer overlay network employed. This is true both in terms of relative delay penalties and in general overhead. The biggest disadvantage of per-group overlays— and therefore of flooding—is the cost of overlay construction required for each group. We also showed that multicast trees built using Pastry can optionally provide higher performance than ones built using CAN but at a higher cost.

A factor that was beyond the scope of this paper is how the various configurations we explored would compare from a fault-tolerance point-of-view. This represents an important area of future work.

Finally, our work enabled us to learn a number of things about CAN-style overlay networks. Most notable is that tuning CAN’s many optimization features is difficult and the effects of changes are unpredictable. In contrast, Pastry employs only a few routing optimizations whose effects can be tuned in a relatively predictable manner.

Perhaps the most interesting thing we discovered about CAN was that dramatic improvements can be obtained by employing topological assignment of addresses instead of random assignment. This is in contrast to Pastry, where topological assignment proved to be, if anything, detrimental when employed underneath tree-based multicast implementations. Since realistic implementations of this feature are an open research problem, this represents an important area for future investigation.

References