An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer Overlays

Miguel Castro*, Michael B. Jones†, Anne-Marie Kermarrec*, Antony Rowstron*,
Marvin Theimer†, Helen Wang† and Alec Wolman†
* Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK
† Microsoft Research, Redmond, WA, USA.

Abstract— **Structured 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 and flooding. This paper evaluates these two approaches using two different types of structured overlay: 1) overlays which use a form of generalized hypercube routing, e.g., Chord, Pastry and Tapestry, and 2) overlays which use a numerical distance metric to route through a Cartesian hyper-space, e.g., CAN. Pastry and CAN are chosen as the representatives of each type of overlay.**

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. The two approaches to multicast are independent of overlay network choice, and we provide a comparison of flooding versus tree-based multicast on both overlays. Results show that the tree-based approach consistently outperforms the flooding approach. Finally, for tree-based multicast, we show that Pastry provides better performance than CAN.

I. INTRODUCTION

The lack of deployment of IP multicast has led to an interest in application-level multicast, e.g., [1], [2], [3], [4], [5]. A number of application-level multicast systems have been proposed that are built using structured peer-to-peer (p2p) overlays and that claim to support large numbers of members in a highly scalable manner, e.g., Bayeux [3], CAN-Multicast [4] and Scribe [5], [6].

Each uses a different p2p overlay and implements application-level multicast using either flooding (CAN-Multicast) or tree-building (Bayeux and Scribe). The p2p overlays provide similar high-level functionality, but use different underlying algorithms. They fall into two categories: one using a generalized hypercube routing algorithm (e.g., Chord [7], Pastry [8] and Tapestry [9]), the other using a numerical distance metric to guide routing through a Cartesian hyper-space (CAN [10]).

The flooding approach to providing multicast creates a separate overlay network per multicast group and leverages the routing information already maintained by a group's overlay to broadcast messages within the overlay. The tree approach uses a single overlay and builds a spanning tree for each group, on which the multicast messages for the group are propagated.

There has been no attempt to compare the performance of these approaches using different p2p overlays. In this paper we examine the performance of both approaches, on both types of p2p overlay. The same network simulator and workloads have been used consistently for all experiments to enable a fair comparison.

We chose Pastry and CAN as representatives for each category of structured p2p overlay. We selected CAN-Multicast and Scribe as the representatives for the flooding and the tree-building approaches to providing application-level multicast. The Scribe tree-building approach, based on reverse path forwarding [11], provides better scalability than Bayeux, which requires the root of the tree to handle group membership.

Our results show that the tree-building approach to multicast achieves lower delay and overhead than flooding over per-group overlays, regardless of the underlying p2p overlay. The biggest disadvantage of per-group overlays is the cost of constructing an overlay for each group. We also show that multicast trees built using Pastry provide higher performance than ones built using CAN.

Section II provides an overview of CAN and Pastry. Flooding and tree-building are described in Section III. Section IV explores configurations and results of detailed simulation experiments to compare the performance of both approaches using both CAN and Pastry. This comparison is summarized in Section V. Related work is reported in Section VI. Section VII concludes.

II. PEER-TO-PEER OVERLAY NETWORKS

Structured peer-to-peer overlays (e.g. CAN [10], Chord [7], Pastry [8] and Tapestry [9]) provide efficient routing over an abstract namespace. They assign a portion of the namespace to each node and provide a primitive to send messages to keys, which are points in the namespace. The overlay routes a message to the node responsible for the portion of the namespace that contains the destination key. There are two main classes of p2p routing algorithms: Chord, Pastry, and Tapestry use a divide-and-conquer approach to route in a ring; and CAN routes in a Cartesian hyper-space by choosing a neighboring node closer to the destination at each hop. The different algorithms exploit network locality for efficiency with varying degrees of success but they are all scalable, fault resilient, and self-organizing.

In this section, we provide an overview of the p2p overlays we have chosen for this study as representative of the two main classes, namely CAN and Pastry.

A. CAN Overlay Network

The Content Addressable Network (CAN) [10] organizes overlay nodes into a d-dimensional hypercube. Each node takes ownership of a specific hyper-rectangle in the space, such that the entire space is covered, and it maintains a routing table with its immediately adjacent neighbors. 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. A node routes a message by forwarding it to a neighbor closer to the destination in the CAN hyper-space. This process is repeated until the message reaches the node whose region contains the destination key. For a d-dimensional space partitioned into n equal zones, the average routing path length is $(d/4)\sqrt[d]{n}$ hops and individual nodes maintain $2d$ neighbors, meaning that, with constant per-node state, routing scales as $O(\sqrt[d]{n})$.

Beyond the basic CAN algorithm described above, CAN has a number of tunable parameters that can be used to improve its routing performance. While these were described in [10], we summarize them here:

Dimensions: Dimensionality of the CAN hypercube.

Network-aware Routing: Ordinary CAN routing chooses the neighbor closest to the destination in CAN space. Ratio-based routing chooses the neighbor with the best ratio of progress in CAN distance to network delay cost. We developed a new greedy routing strategy, network-delay routing (NDR), that chooses the neighbor with least network delay cost, subject to the constraint that the message moves closer to the destination on each hop.

Multiple Nodes per Zone: This parameter 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.

Uniform Partitioning: When a node joins the CAN network, the node responsible for the destination key compares the volume of its region with the volumes of neighboring regions rather than immediately splitting its region in two. If any neighboring zone is larger than the current zone, the join message is forwarded to the neighbor. This test is applied repeatedly, until a local maximum size is reached, at which point 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 address at which to join, the CAN space is divided into evenly sized bins, and the CAN join address is randomly 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 was to produce an independent implementation of CAN for our simulator. To validate this implementation we then set about reproducing the results presented in the original CAN paper. In nearly all cases we were able to reproduce these results within a few percent of the original values.

B. Pastry Overlay Network

Pastry [8] uses a circular 128-bit namespace. It assigns to each overlay node a nodeId chosen randomly with uniform probability from the namespace. Given a message and a destination key, Pastry routes the message to the node whose nodeId is numerically closest to the key. The expected number of hops is $log_{2^b}N$ (where N is the number of nodes in the overlay and b is a configuration parameter with typical value 4). Eventual delivery is guaranteed unless $\lfloor l/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously (l is a configuration parameter with value 16 in this paper).

For the purpose of routing, nodeIds and keys are interpreted as a sequence of digits in base 2^b. A node's routing table is organized into $128/b$ levels with 2^b entries in each level. The entry in column m at level n of a node p's routing table points to a node whose nodeId shares the first n digits with p's nodeId and whose $n+1$th digit is m. The routing table entry is left empty if no such node is known. Additionally, 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.

The uniform distribution of nodeIds ensures an even population of the nodeId space; only $\lceil log_{2^b}N \rceil$ levels in the routing table are populated on average. Additionally, only 2^b-1 entries per level are populated because one corresponds to the local node. Thus, each node maintains only $(2^b-1)\times\lceil log_{2^b}N\rceil+2l$ entries on average. Moreover, after a node failure or the arrival of a new node, the tables can be repaired by exchanging $O(log_{2^b}N)$ messages among the affected nodes.

At each routing step, a node normally forwards the message to a node whose nodeId shares with the destination key a prefix that is at least one digit (or b bits) longer than the prefix that the key 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 key as long as the current node but is numerically closer to the key than the present node's id. Such a node must be in the leaf set unless the message has already arrived at the node responsible for the key or its immediate neighbor. One of these nodes must be live unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously.

Pastry exploits network locality to reduce routing delays. It measures the delay (RTT) to a small number of nodes when building routing tables. For each routing table entry, it chooses one of the closest nodes in the network topology whose nodeId satisfies the constraints for that entry. Since the constraints are stronger for the lower levels of the routing table, the average IP delay of each Pastry hop increases exponentially until it reaches the average delay between two nodes in the network.

III. Overlay-Based Application-Level Multicast

Two approaches have been taken to implementing application-level multicast on peer-to-peer overlays: flooding and tree building. Flooding leverages the information that nodes already maintain for overlay routing 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 utilized 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 [3], [5].

A. Overlay-Per-Group 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.

1) CAN Flooding: The broadcast algorithm we implemented for CAN is based on the flooding algorithm described in [4], with some significant modifications. The naive approach to implement flooding on a CAN overlay network is for each node that receives a message to forward that message to all its neighbors. Nodes cache the sequence number of received messages to filter duplicates. The CAN Multicast [4] study presents a more efficient flooding algorithm that exploits the structure of the CAN coordinate space to limit the directions in which each node will forward the messages it receives. This approach vastly reduces the number of duplicate messages.

We discovered and fixed two flaws in the published CAN efficient flooding algorithm. The first flaw is a race condition that can lead to certain nodes in the CAN overlay that never receive the broadcast message. The second is an ambiguity in the algorithm that leads to a larger number of duplicate messages than specified. The details of our fixes to the efficient flooding algorithm are beyond the scope of this paper and are thoroughly described in [12].

2) Pastry Flooding: The Pastry broadcast algorithm uses the entries in each node's routing table to flood messages. A node wishing to broadcast a message forwards copies to all the nodes in its routing table and tags each copy with the level l of the destination node in the routing table. When a node receives a copy of the message tagged with l, it forwards copies to all nodes in levels greater than l in its routing table. As before, each copy is tagged with the level l' of the destination node. This is repeated until the nodes receiving copies of the message have no other nodes to forward the message to.

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, this may lie within the leafset of the node. If so, nothing else is done; otherwise the message is routed using Pastry to the midpoint of domain d at level l. The message is tagged with level l and marked as a midpoint request. The node that receives the midpoint request is the one with the numerically closest nodeId. If the nodeId is within the requested domain, it forwards the message to all entries in its routing table at levels greater than l. If the nodeId is not within the domain but the nodeId of a leafset node is, the message is forwarded to that node. Otherwise, the message is a duplicate and is discarded.

B. Tree-Per-Group Implementations

As a representative example of tree-based multicast, we use Scribe's approach [5], [6]. Scribe is a generic application-level multicast infrastructure originally implemented using Pastry. Scribe uses reverse path forwarding [11] to build a multicast tree per group, formed by the union of the overlay routes from the group members to the root of the tree.

Each group is identified by a key called the *groupId*, for example, a hash of the group's textual name concatenated with its creator name. The node responsible for the namespace segment containing the groupId is the root of the tree.

To join a group, a node routes a message through the overlay to the groupId. This message is routed towards the root of the tree. Each node along the route checks whether it is already in the tree. If it is, it registers the source node as a child and does not forward the message any further. Otherwise, it creates a children table for the group, adds the source node as a child, and sends a join message for itself to the groupId. Nodes with children in the tree are called forwarders, and may not be members of the group.

Scribe scales well because of its decentralized algorithm. In particular, randomization of overlay addresses ensures that the tree is well balanced. The load to forward multicast messages and handle membership changes is well distributed among the group members. Additionally, Scribe provides a mechanism to remove bottlenecks: 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. This *bottleneck remover* is described in [6].

Scribe may create trees that are deeper than necessary for small groups. It is possible to collapse long paths in the tree by removing nodes that are not members of the group and have only one entry in the group's children table. This mechanism is described and evaluated in [6].

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 [6].

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 single node that is a cohabitant of the zone that the *groupId* is part of.

IV. Evaluation

A. Experimental Setup

We used a simple packet-level, discrete event simulator to evaluate the different approaches. 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 queuing delay or packet losses because modeling these would prevent simulation of large networks.

The simulations ran on five network topologies with 5050 routers, which were generated using the Georgia Tech [13] random graph generator according to the transit-stub model (as described in [6]). The 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 of each topology with uniform probability. Each end system was directly attached by a LAN link to its assigned router (as in [2]). We used different random number generator seeds for each topology.

We used the routing policy weights generated by the Georgia Tech random graph generator [13] to perform IP unicast routing. IP multicast routing used a tree formed by the union of the unicast routes from the source to each recipient. This is similar to what could be obtained in our experimental setting using protocols like Distance Vector Multicast Routing Protocol (DVMRP) [14]. To provide a conservative comparison, we ignored messages required by the IP multicast protocols to maintain the trees. The delay of each LAN link was 1ms and the average delay of core links (computed by the graph generator) was approximately 40ms.

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 $\lfloor Nr^{-1.25} + 0.5 \rfloor$, 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. We also ran each experiment five times using a different topology and different random number generator seeds for each run. We present the average of the results obtained in the five runs.

B. 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 associated with the method used to compute the delay penalty in [2].

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 [2]. 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 phases when members join the group and 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 table 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.

C. CAN Results

1) Parameters: CAN has a 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

State	Dimensions (d)	Nodes Per Zone (z)	Uniform Partitioning
18	10	1	enabled
29	9	2	enabled
38	12	3	enabled
59	10	5	enabled
111	8	10	enabled

TABLE I
REPRESENTATIVE GOOD CONFIGURATIONS, AS A FUNCTION OF NEIGHBOR STATE, FOR AN 80,000 NODE CAN.

following parameters during our exploration: the number of dimensions; the number of nodes per zone; turning on and off uniform partitioning; choosing either random or landmark-based node assignment; and choosing a routing policy from CAN distance, CAN ratio, or NDR. 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 exploration of the CAN parameter space and resulting conclusions are presented in greater detail in [12].

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. Nonetheless, we can provide some general guidelines. Enabling landmark-based topological assignment 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 landmark-based assignment where nodes often end up clustered close together in the CAN space. Furthermore, uniform partitioning never causes RAD to become significantly worse. The NDR routing metric appears to perform consistently better than the other routing metrics. Increasing the number of nodes per zone provides predictable improvements in RAD and the neighbor state overhead increases linearly. As the number of nodes in the system varies, the CAN parameters that perform well also vary - especially the number of dimensions. Table I lists a set of representative good configurations for an 80,000 node CAN at a variety of different state overheads.

For the CAN multicast experiments that follow, we use the configurations listed in Table I. We enable uniform partitioning in all cases, and we vary the routing metric (CAN distance, NDR, or ratio-based) and the node assignment policy (RAND or TOP). For the topological assignment (TOP) policy, we use a set of 32 landmarks. This results in 30 different experimental configurations of our CAN overlay network. We run each of our application-level multicast implementations on each of these variations of CAN.

2) Flooding-Based Results: In this section we explore the behavior of flooding on CAN. We start by describing the impact of the various parameters on delays. Figures 1 and 2 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.

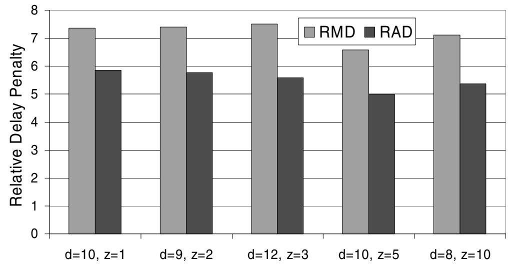

Fig. 1. Relative delay penalty for CAN flooding with different values of d and z.

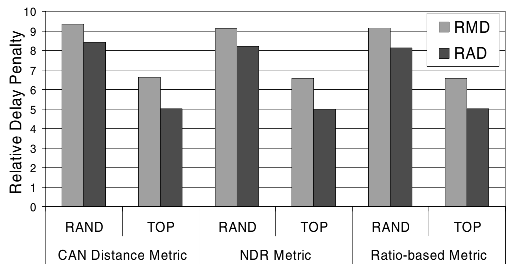

Fig. 2. Relative delay penalty for CAN flooding with and without topology-aware optimizations using a d=10,z=5 CAN configuration.

Figure 1 plots the delay penalty of the "best" representative CAN configurations at each routing table state size. The best representative was considered to be one using landmark-based assignment and the NDR routing metric.

Figure 2 shows the effect on delays of varying the routing metric and the node assignment policy with the best CAN configuration (d=10, z=5) from Figure 1. For flooding-based multicast, all other CAN configurations showed the same effects as are evident for this one. Several interesting things are noticeable: we confirmed that the benefit of landmark-based assignment translates over from unicast communications to flooding-based communications. In all cases, landmark-based assignment proved to dominate random address assignment. Improvements of up to 40% were observed. In contrast, the choice of routing metric did *not* have much effect. All variation was under 10%.

For flooding-based multicast, the benefit to be gained from increased amounts of routing table state is uneven. Increasing the state from 59 to 111 actually yielded slightly poorer delay penalty values, whereas increases up to 59 yielded better delay penalty values. The difference between best and worst values was just under 20%. The independence of delay penalty to routing table state size is unsurprising when one considers that the CAN flooding algorithm cannot take advantage of most of the optimizations that CAN employs.

We also evaluated link stress results for both the subscription and publication phases of multicasting. As with delay penalty, landmark-based assignment yielded uniformly better link stress numbers and the choice of routing metric did not make a significant difference. Table II 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

Configuration	d=10 z=1	d=9 z=2	d=12 z=3	d=10 z=5	d=8 z=10
State size	18	29	38	59	111
Joining phase					
Max	91615	149341	197977	309212	416361
Average	154	183	219	281	431
Flooding phase					
Max	1958	1595	1333	985	631
Average	3.49	3.27	2.93	2.73	2.69

TABLE II
LINK STRESS FOR FLOODING IN CAN.

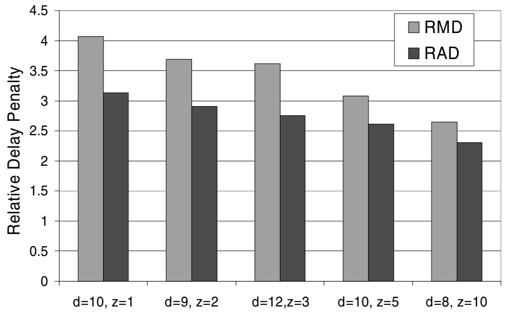

Fig. 3. Relative delay penalty for CAN tree-based multicast with different values of d and z.

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 *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 neighbor traffic that must occur when each group member joins the CAN. The equivalent fall in values for the publication phase can be understood as follows: maintenance of more neighbor state implies that there are a greater number of routing paths out of each node. This increases the chances that a given node will employ all possible network links emanating from it when forwarding a broadcast message.

Finally, we counted the number of duplicate messages received by overlay nodes. With our improved flooding algorithm we never saw more than a few duplicate messages in any of our experimental runs. Many runs encountered no duplicate messages at all.

3) Tree-Based Results: In this section we explore the behavior of tree-based multicast over CAN. We start by describing the impact of the various parameters on delays in Figures 3 and 4.

Figure 3 shows the delay penalty for the "best" CAN configurations varying the amount of routing table state. Although we observe some benefit from increasing amounts of routing table state, the benefit is moderate. The improvement from the smallest to the largest configuration was 36%.

Figure 4 shows the effect on delays of using topology-aware routing and topology-aware address assignment for the best CAN configuration (d=8, z=10) from Figure 3. All other CAN

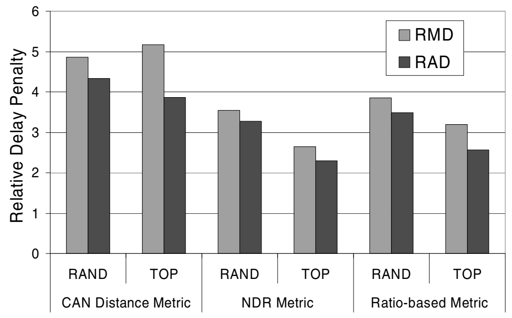

Fig. 4. Relative delay penalty for CAN tree-based multicast with and without topology-aware optimizations using a d=8,z=10 CAN configuration.

Configuration	d=10 z=1	d=9 z=2	d=12 z=3	d=10 z=5	d=8 z=10
State size	18	29	38	59	111
Max	323	220	198	184	225
Average	1.69	1.49	1.42	1.37	1.36

TABLE III
LINK STRESS FOR CAN TREE-BASED MULTICAST.

configurations showed the same effects as this one.

As with the flooding-based results, we observed that landmark-based assignment dominated random assignment. Improvements of up to 30% 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 the NDR metric or the ratio-based routing metric. Of the latter two routing metrics the NDR metric consistently performed slightly better than the ratio-based metric, as was the case for unicast.

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 III shows both maximum and average link stress values for the best representative CAN configurations at each routing table state size. 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.

As with delay penalty, landmark-based assignment yielded uniformly better link stress numbers than did random assignment. However, the improvement in link stress is far more dramatic than the improvement for delay penalties. Typical improvement for maximum link stress was about a factor of 6, while typical improvement for average link stress was about a factor of 2 to 3. In contrast, choice of routing metric made a small difference, with the NDR metric doing the best.

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 multicast messages. Table IV shows the maximum number of entries seen on any node as a function of routing table state size, and the number of nodes having to act as forwarders as a function of routing

Configuration	d=10 z=1	d=9 z=2	d=12 z=3	d=10 z=5	d=8 z=10
State size	18	29	38	59	111
Max number of forwarding entries on any node	19	26	32	49	87
# of forwarder nodes	43726	39384	34318	28101	22548

TABLE IV
CAN TREE-BASED MULTICAST FORWARDING STATISTICS.

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 table entries they need to support as well. At the same time, the number of nodes that have to perform forwarding duties goes down. Thus, going to larger CAN state configurations seems to concentrate more load onto fewer nodes.

A similar node "concentration" effect was observed with respect to the use of landmark-based address assignment. Use of this feature results in an increase in the maximum number of forwarding entries per node of about 20% compared to random address assignment. But, as mentioned earlier, employing landmark-based assignment dramatically reduces link stress in the system. Thus employing landmark-based assignment seems to trade a substantial decrease in link stress on the system for a minor increase in node load across the system. Furthermore, Scribe's bottleneck remover can be used to bound a node's forwarding load. For example, we reran all the experiments with the bottleneck remover and a bound of 32 forwarding entries. The bound was achieved with negligible impact on link stress and delay.

D. Pastry Results

1) Parameters: 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 but it builds the routing tables such that they point to nearby nodes in the network topology whereas NDR optimizes the *choice* of neighbor at each routing hop but does not control the set of neighbors.

Pastry uses topology-aware routing table construction as described in Section II-B: 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

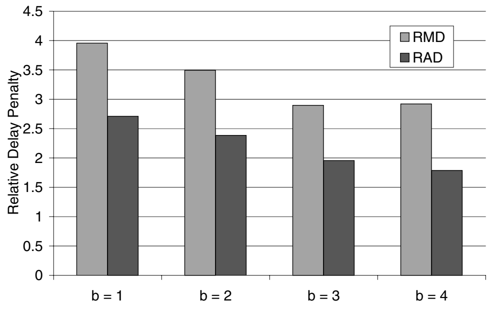

Fig. 5. Relative delay penalty for Pastry flooding with different values of b.

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 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.

2) Flooding-Based Results: 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 5 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. For example, RAD with $b = 4$ is 50% lower than with $b = 1$. This happens because increasing the value of b decreases the number of Pastry hops, which is approximately equal to $log_{2^b}(N)$.

Figure 6 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 effective at reducing the delay penalty relative to the version of pastry without topology-aware optimizations (Pastry with RAND and without TART): they both reduce the RAD by a factor of about 2 and combining them reduces the RAD by a

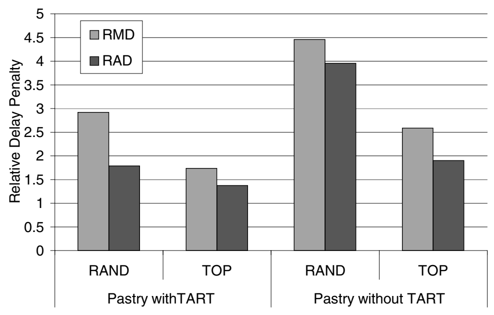

Fig. 6. Relative delay penalty for Pastry flooding with and without topology-aware optimizations for $b = 4$.

	with TART		without TART	
	RAND	TOP	RAND	TOP
Max	6801.4	65.4	2119.0	61.0
Average	4.3	1.4	4.6	1.4

TABLE V
LINK STRESS FOR PASTRY FLOODING WITH AND WITHOUT TOPOLOGY-AWARE OPTIMIZATIONS FOR $b = 4$.

factor of almost 3.

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 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 V 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. The Pastry hops used by flooding form a spanning tree rooted at the sender. At the top level, messages travel a long IP (and namespace) distance but there is only a small number of them. The number of messages increases exponentially towards the leaves of the tree and these messages travel increasingly shorter IP (and namespace) 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 IP distance traversed by messages increases as one moves down the spanning tree. Therefore, the link stress reduction is not as significant as with TOP because most messages traverse a large number of physical links. 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.

Table VI shows the routing state size, i.e., the average

b	1	2	3	4
State size	20.7	29.2	42.7	64.6
Duplicates	13.2	224.6	2201.6	12579.6

TABLE VI
NUMBER OF UNIQUE ENTRIES IN ROUTING TABLE AND LEAFSET, AND NUMBER OF DUPLICATE MESSAGES.

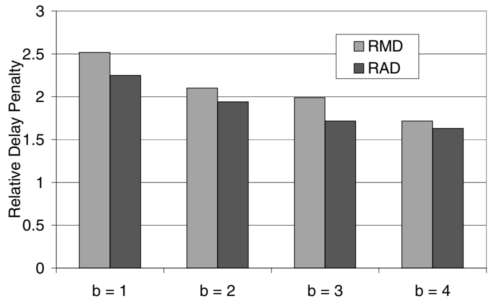

Fig. 7. Relative delay penalty for Pastry tree-based multicast for different values of b.

number of unique nodes in a Pastry node's routing table and leaf set for different values of b. This number increases with b as expected; it is approximately equal to $(2^b - 1) log_{2^b}(N) + L'$, 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 is only 79 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 for different values of b. These results appear in Table VI. 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 16%) but we can repair the routing tables at low cost to prevent almost all duplicates in subsequent multicasts. The topology-aware optimizations have no effect on the number of duplicates.

3) Tree-Based Results: 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 7 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 8.

Table VII shows the impact of the TART and TOP optimizations on link stress during multicasts. 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

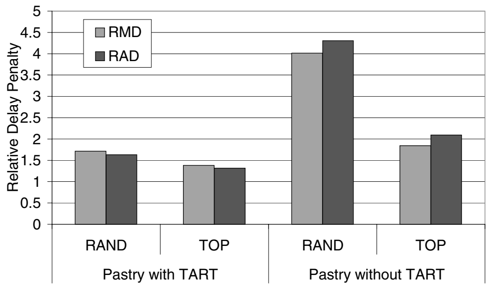

Fig. 8. Relative delay penalty for Pastry tree-based multicast with and without topology-aware optimizations for $b = 4$.

	with TART		without TART	
	RAND	TOP	RAND	TOP
Max	286.2	22,073.8	1,910.6	23,999.4
Average	1.17	3.34	3.87	3.90

TABLE VII
LINK STRESS FOR PASTRY TREE-BASED MULTICAST WITH AND WITHOUT TOPOLOGY-AWARE OPTIMIZATIONS FOR $b = 4$.

follow the reverse of the Pastry routes from each group member to the root. With TART and RAND, the longest hops will be at the top of the tree. Therefore, most messages will travel over a small number of physical links 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 large number of physical links 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 226.8 and the average is 1.07.

The tree-based multicast scheme adds a forwarding table per node. Table VIII shows the maximum number of forwarding table entries per-node for different values of b in Pastry. The maximum number of forwarding table entries is 346 with $b = 3$. This is relatively high but Scribe's bottleneck remover can be used to bound the number of forwarding table entries per node. We reran the experiment using TART and RAND with $b = 4$ and an upper bound of 32 forwarding table entries. The results show that the bound is achieved with a negligible impact on delay; the RMD does not change and the RAD increases by 2%. The maximum link stress during joining increases from 286.2 to 580.75 and the average from 1.17 to 1.79. This is reasonable because joins are less frequent than multicasts in most applications and the link stress during multicasts decreases; the maximum decreases from 286.2 to 72 and the average from 1.17 to 1.11.

b	1	2	3	4
Maximum	215.4	268.8	346	286.2

TABLE VIII
MAXIMUM NUMBER OF FORWARDING TABLE ENTRIES PER NODE FOR DIFFERENT VALUES OF b

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.

E. CAN/Pastry Results for Multiple Multicast Groups

This section describes the set of experiments we ran with 1500 multicast groups instead of just one. The group sizes varied according to a Zipf-like distribution, as described earlier. We explored two cases: group members uniformly distributed over the network, and localized members. 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 the NDR routing metric.

Figures 9 and 10 show the cumulative distribution function for RMD for the non-localized and localized group members, respectively. Figure 11 shows the corresponding cumulative distribution function for RAD for localized group members. We omit the RAD results for non-localized group members because they are almost identical. 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

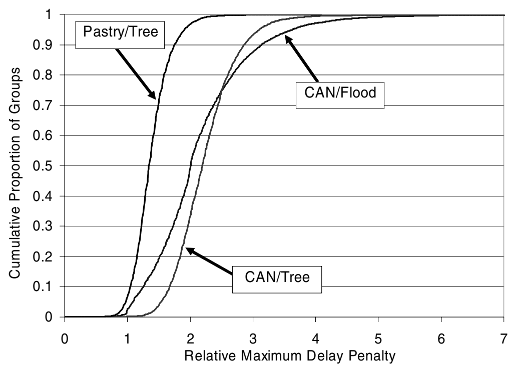

Fig. 9. CDF for RMD for 1500 concurrent multicast groups with localized group members.

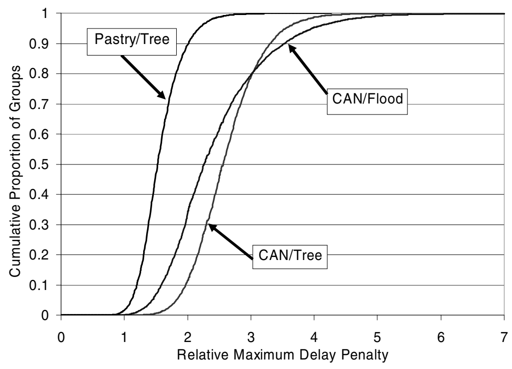

Fig. 10. CDF for RMD for 1500 concurrent multicast groups with globally distributed group members.

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.

V. 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, there is no major difference. 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 the neighbor state and in Pastry create the routing table and leafsets. This may involve finding out about and potentially contacting dozens of other nodes. In contrast, multicast trees built using a single overlay network can take advantage of the routing state already established in the existing overlay network. This makes joining the group lightweight in terms of both space and time.

The advantage of using a separate overlay per multicast group is that traffic for the group is carried only by members of the group. If this property is important for administrative or other reasons, then the overlay-per-group approach should be considered. But otherwise our results argue against it.

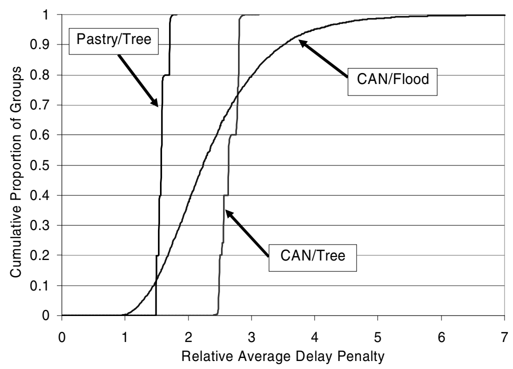

Fig. 11. CDF for RAD for 1500 concurrent multicast groups with localized group members

CAN versus Pastry with Per-Group Multicast Trees: Our results indicate that, with equivalent per-node state, with representative tuned settings, and with identical workloads, the relative delay penalty values obtained using Pastry are typically on the order of 20% to 50% better than those obtained using CAN. When looking at link stress, Pastry exhibited an average link stress that was roughly 15% lower than that seen with CAN, while CAN exhibited a maximum link stress that was roughly 25% lower than that seen with Pastry. Similarly, the maximum number of forwarding entries that had to be maintained under CAN was about a third of that seen with Pastry but the average was similar. The bottleneck remover allows Pastry to retain its delay advantage while lowering the maximum link stress and the maximum number of forwarding table entries to values as low as CAN's. Overall, we conclude that multicast trees built using Pastry can provide higher performance than ones built using CAN.

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.

VI. Related Work

Tapestry [9] and Chord [7] are similar to Pastry; they also use a divide-and-conquer approach to route within a ring. 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; Pastry without locality optimizations provides a good emulation of Chord, and Pastry with locality provides a good emulation of Tapestry.

Another proposal for doing multicast using general-purpose overlay networks is the Bayeux [3] system for Tapestry. Bayeux builds a multicast tree per group differently from the approach examined in this paper. Each request to join a group is to the root, which 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.

Another scalable overlay multicast system is Overcast [1]. Like Bayeux, Overcast requires that joining nodes coordinate with a central root node.

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) [15], End System Multicast [2], and ISIS/Horus-style Virtual Synchrony [16], but which provide other benefits.

Of course, all the work for constructing multicast distribution trees builds upon the techniques originally developed for IP Multicast [14], [17].

VII. 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.

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 provide higher performance than ones built using CAN.

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.

Acknowledgements

We thank Sylvia Ratnasamy for answering our many questions concerning CAN, and for the topology files used in their experiments. Without her help we could not have validated our CAN implementation.

References

[1] J. Jannotti, D. K. Gifford, K. L. Johnson, F. Kaashoek, and J. W. O'Toole, "Overcast: Reliable Multicasting with an Overlay Network," in *Proc. of OSDI*, October 2000, pp. 197–212.

[2] Y.-H. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *Proc. of ACM Sigmetrics*, June 2000, pp. 1–12.

[3] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination," in *Proc. of NOSSDAV*, June 2001.

[4] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proc of NGC*, Nov. 2001.

[5] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *Proc of NGC*, Nov. 2001.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE JSAC*, vol. 20, no. 8, October 2002.

[7] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc of ACM SIGCOMM*, San Diego, California, August 2001.

[8] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc of Middleware*, Nov. 2001.

[9] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing," U. C. Berkeley, Tech. Rep. UCB//CSD-01-1141, April 2001.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Proc. of ACM SIGCOMM*, Aug. 2001.

[11] Y. Dalal and R. Metcalfe, "Reverse path forwarding of broadcast packets," *CACM*, vol. 21, no. 12, pp. 1040–1048, 1978.

[12] M. B. Jones, M. Theimer, H. Wang, and A. Wolman, "Unexpected complexity: Experiences tuning and extending CAN," Microsoft Research, Tech. Rep. MSR-TR-2002-118, December 2002.

[13] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proc of IEEE Infocom*, April 1996.

[14] S. Deering and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, May 1990.

[15] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient overlay networks," in *18th ACM SOSP*, Oct. 2001.

[16] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proc of the ACM SOSP*, November 1987, pp. 123–138.

[17] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei, "The PIM Architecture for Wide-Area Multicast Routing," *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, April 1996.