SCRIBE: A large-scale and decentralized application-level multicast infrastructure

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec and Antony Rowstron

Abstract—

This paper presents Scribe, a scalable application-level multicast infrastructure. Scribe supports large numbers of groups, with a potentially large number of members per group. Scribe is built on top of Pastry, a generic peer-to-peer object location and routing substrate overlayed on the Internet, and leverages Pastry's reliability, self-organization, and locality properties. Pastry is used to create and manage groups and to build efficient multicast trees for the dissemination of messages to each group. Scribe provides best-effort reliability guarantees, and we outline how an application can extend Scribe to provide stronger reliability. Simulation results, based on a realistic network topology model, show that Scribe scales across a wide range of groups and group sizes. Also, it balances the load on the nodes while achieving acceptable delay and link stress when compared to IP multicast.

Keywords— **group communication, application-level multicast, peer-to-peer.**

I. INTRODUCTION

Network-level IP multicast was proposed over a decade ago [1], [2], [3]. Subsequently, multicast protocols such as SRM (Scalable Reliable Multicast Protocol) [4] and RMTP (Reliable Message Transport Protocol) [5] have added reliability. However, the use of multicast in applications has been limited because of the lack of wide scale deployment and the issue of how to track group membership.

As a result, application-level multicast has gained in popularity. Algorithms and systems for scalable group management and scalable, reliable propagation of messages are still active research areas [6], [7], [8], [9], [10], [11]. For such systems, the challenge remains to build an infrastructure that can scale to, and tolerate the failure modes of, the general Internet, while achieving low delay and effective use of network resources.

Recent work on peer-to-peer overlay networks offers a scalable, self-organizing, fault-tolerant substrate for decentralized distributed applications [12], [13], [14], [15]. In this paper we present Scribe, a large-scale, decentralized application-level multicast infrastructure built upon Pastry, a scalable, self-organizing peer-to-peer location and routing substrate with good locality properties [12]. Scribe provides efficient application-level multicast and is capable of scaling to a large number of groups, of multicast sources, and of members per group.

Scribe and Pastry adopt a fully decentralized peer-to-peer model, where each participating node has equal responsibilities. Scribe builds a multicast tree, formed by joining the Pastry routes from each group member to a rendez-vous point associated with a group. Membership maintenance and message dissemination in Scribe leverage the robustness, self-organization, locality and reliability properties of Pastry.

The rest of the paper is organized as follows. Section II gives an overview of the Pastry routing and object location infrastructure. Section III describes the basic design of Scribe. We present performance results in Section IV and discuss related work in Section V.

II. PASTRY

In this section we briefly sketch Pastry [12], a peer-to-peer location and routing substrate upon which Scribe was built. Pastry forms a robust, self-organizing overlay network in the Internet. Any Internet-connected host that runs the Pastry software and has proper credentials can participate in the overlay network.

Each Pastry node has a unique, 128-bit nodeId. The set of existing nodeIds is uniformly distributed; this can be achieved, for instance, by basing the nodeId on a secure hash of the node's public key or IP address. Given a message and a key, Pastry reliably routes the message to the Pastry node with the nodeId that is numerically closest to the key, among all live Pastry nodes. Assuming a Pastry network consisting of N nodes, Pastry can route to any node in less than $\lceil log_{2^b} N \rceil$ steps on average (b is a configuration parameter with typical value 4). With concurrent node failures, eventual delivery is guaranteed unless $l/2$ or more nodes with *adjacent* nodeIds fail simultaneously (l is an even integer parameter with typical value 16).

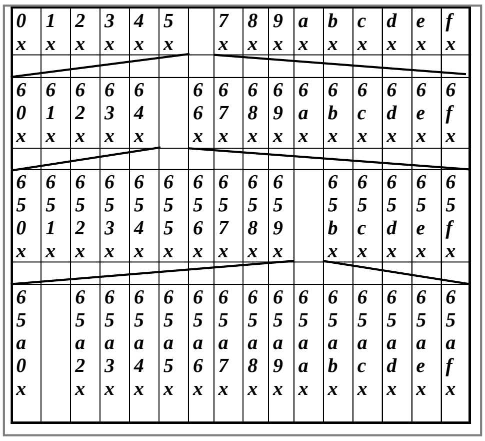
The tables required in each Pastry node have only $(2^b - 1) * \lceil log_{2^b} N \rceil + l$ entries, where each entry maps a nodeId to the associated node's IP address. Moreover, after a node failure or the arrival of a new node, the invariants in all affected routing tables can be restored by exchanging $O(log_{2^b} N)$ messages. In the following, we briefly sketch the Pastry routing scheme. A full description and evaluation of Pastry can be found in [12], [16].

For the purposes of routing, nodeIds and keys are thought of as a sequence of digits with base 2^b. A node's routing table is organized into $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each (see Figure 1). The $2^b - 1$ entries in row n of the routing table each refer to a node whose nodeId matches the present node's nodeId in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n + 1$th digit in the present node's id. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, only $\lceil log_{2^b} N \rceil$ levels are populated in the routing table. Each entry in the routing table refers to one of potentially many nodes whose nodeId have the appropriate prefix. Among such nodes, the one closest to the present node (according to a scalar proximity metric, such as the round trip time) is chosen.

In addition to the routing table, each node maintains IP addresses for the nodes in its *leaf set*, i.e., the set of nodes with the $l/2$ numerically closest larger nodeIds, and the $l/2$ nodes

M. Castro, A-M. Kermarrec and A. Rowstron are with Microsoft Research Ltd., 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK

P. Druschel is with Rice University MS-132, 6100 Main Street, Houston, TX 77005, USA.



0x	*1x*	*2x*	*3x*	*4x*	*5x*		*7x*	*8x*	*9x*	*ax*	*bx*	*cx*	*dx*	*ex*	*fx*
60x	*61x*	*62x*	*63x*	*64x*		*66x*	*67x*	*68x*	*69x*	*6ax*	*6bx*	*6cx*	*6dx*	*6ex*	*6fx*
650x	*651x*	*652x*	*653x*	*654x*	*655x*	*656x*	*657x*	*658x*	*659x*		*65bx*	*65cx*	*65dx*	*65ex*	*65fx*
65a0x		*65a2x*	*65a3x*	*65a4x*	*65a5x*	*65a6x*	*65a7x*	*65a8x*	*65a9x*	*65aax*	*65abx*	*65acx*	*65adx*	*65aex*	*65afx*

Fig. 1. Routing table of a Pastry node with nodeId $65a1x$, $b = 4$. Digits are in base 16, x represents an arbitrary suffix. The IP address associated with each entry is not shown.

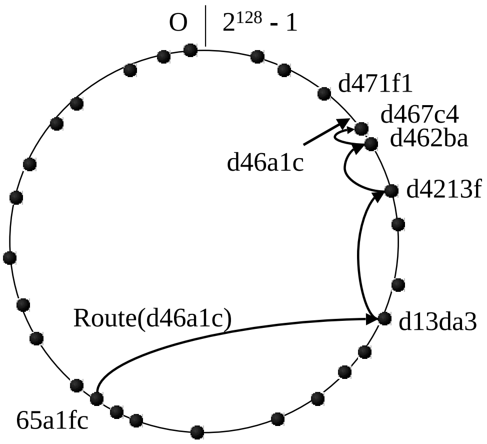


Fig. 2. Routing a message from node $65a1fc$ with key $d46a1c$. The dots depict live nodes in Pastry's circular namespace.

with numerically closest smaller nodeIds, relative to the present node's nodeId.

Figure 2 shows the path of an example message. In each routing step, the current node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the current nodeId. If no such node is found in the routing table, 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 current nodeId. Such a node must exist in the leaf set unless the nodeId of the current node or its immediate neighbour is numerically closest to the key, or $l/2$ adjacent nodes in the leaf set have failed concurrently.

A. Locality

Next, we discuss Pastry's locality properties, i.e., the properties of Pastry's routes 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 round trip time. 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.

We limit our discussion to two of Pastry's locality properties that are relevant to Scribe. The *short routes* property concerns the total distance, in terms of the proximity metric, that messages travel along Pastry routes. Recall that each entry in the node routing tables is chosen to refer to the nearest node, according to the proximity metric, with the appropriate nodeId prefix. As a result, in each step a message is routed to the nearest node with a longer prefix match. Simulations performed on several network topology models show that the average distance traveled by a message is between 1.59 and 2.2 times the distance between the source and destination in the underlying Internet [16].

The *route convergence* property is concerned with the distance traveled by two messages sent to the same key before their routes converge. Simulations show that, given our network topology model, the average distance traveled by each of the two messages before their routes converge is approximately equal to the distance between their respective source nodes. These properties have a strong impact on the locality properties of the Scribe multicast trees, as explained in Section III.

B. Node addition and failure

A key design issue in Pastry is how to efficiently and dynamically maintain the node state, i.e., the routing table and leaf set, in the presence of node failures, node recoveries, and new node arrivals. The protocol is described and evaluated in [12], [16].

Briefly, an arriving node with the newly chosen nodeId X can initialize its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message using X as the key. This message is routed to the existing node Z with nodeId numerically closest to X[1]. X then obtains the leaf set from Z, and the ith row of the routing table from the ith node encountered along the route from A to Z. One can show that using this information, X can correctly initialize its state and notify nodes that need to know of its arrival.

To handle node failures, neighboring nodes in the nodeId space (which are aware of each other by virtue of being in each other's leaf set) periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed. All members of the failed node's leaf set are then notified and they update their leaf sets. Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial. A recovering node contacts the nodes in its last known leaf set, obtains their current leaf sets, updates its own leaf set and then notifies the members of its new leaf set of its presence. Routing table entries that refer to failed nodes are repaired lazily; the details are described in [12], [16].

C. Pastry API

In this section, we briefly describe the application programming interface (API) exported by Pastry to applications such as Scribe. The presented API is slightly simplified for clarity. Pastry exports the following operations:

[1] In the exceedingly unlikely event that X and Z are equal, the new node must obtain a new nodeId.

nodeId = pastryInit(Credentials) causes the local node to join an existing Pastry network (or start a new one) and initialize all relevant state; returns the local node's nodeId. The credentials are provided by the application and contain information needed to authenticate the local node and to securely join the Pastry network. A full discussion of Pastry's security model can be found in [17].
route(msg,key) causes Pastry to route the given message to the node with nodeId numerically closest to key, among all live Pastry nodes.
send(msg,IP-addr) causes Pastry to send the given message to the node with the specified IP address, if that node is live. The message is received by that node through the deliver method.

Applications layered on top of Pastry must export the following operations:

deliver(msg,key) called by Pastry when a message is received and the local node's nodeId is numerically closest to key among all live nodes, or when a message is received that was transmitted via *send*, using the IP address of the local node.
forward(msg,key,nextId) called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL will terminate the message at the local node.
newLeafs(leafSet) called by Pastry whenever there is a change in the leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

In the following section, we will describe how Scribe is layered on top of the Pastry API. Other applications built on top of Pastry include PAST, a persistent, global storage utility [18], [19].

III. SCRIBE

Scribe is a scalable application-level multicast infrastructure built on top of Pastry. Any Scribe node may create a *group*; other nodes can then join the group, or multicast messages to all members of the group (provided they have the appropriate credentials). Scribe provides best-effort delivery of multicast messages, and specifies no particular delivery order. However, stronger reliability guarantees and ordered delivery for a group can be built on top of Scribe, as outlined in Section III-B. Nodes can create, send messages to, and join many groups. Groups may have multiple sources of multicast messages and many members. Scribe can support simultaneously a large numbers of groups with a wide range of group sizes, and a high rate of membership turnover.

Scribe offers a simple API to its applications:

create(credentials, groupId) creates a group with groupId. Throughout, the credentials are used for access control.
join(credentials, groupId, messageHandler) causes the local node to join the group with groupId. All subsequently received multicast messages for that group are passed to the specified message handler.
leave(credentials, groupId) causes the local node to leave the group with groupId.
multicast(credentials, groupId, message) causes the message to be multicast within the group with groupId.

Scribe uses Pastry to manage group creation, group joining and to build a per-group multicast tree used to disseminate the messages multicast in the group. Pastry and Scribe are fully decentralized: all decisions are based on local information, and each node has identical capabilities. Each node can act as a multicast source, a root of a multicast tree, a group member, a node within a multicast tree, and any sensible combination of the above. Much of the scalability and reliability of Scribe and Pastry derives from this peer-to-peer model.

A. *Scribe Implementation*

A Scribe system consists of a network of Pastry nodes, where each node runs the Scribe application software. The Scribe software on each node provides the *forward* and *deliver* methods, which are invoked by Pastry whenever a Scribe message arrives. The pseudo-code for these Scribe methods, simplified for clarity, is shown in Figure 3 and Figure 4, respectively.

Recall that the forward method is called whenever a Scribe message is routed through a node. The deliver method is called when a Scribe message arrives at the node with nodeId numerically closest to the message's key, or when a message was addressed to the local node using the Pastry *send* operation. The possible message types in Scribe are JOIN, CREATE, LEAVE and MULTICAST; the roles of these messages are described in the next sections.

The following variables are used in the pseudocode: *groups* is the set of groups that the local node is aware of, *msg.source* is the nodeId of the message's source node, *msg.group* is the groupId of the group, and *msg.type* is the message type.

A.1 Group Management

Each group has a unique *groupId*. The Scribe node with a nodeId numerically closest to the groupId acts as the *rendez-vous point* for the associated group. The rendez-vous point is the root of the multicast tree created for the group.

To create a group, a Scribe node asks Pastry to route a CREATE message using the groupId as the key (e.g. route(CREATE,groupId)). Pastry delivers this message to the node with the nodeId numerically closest to groupId. The Scribe deliver method adds the group to the list of groups it already knows about (line 3 of Figure 4). It also checks the credentials to ensure that the group can be created, and stores the credentials. This Scribe node becomes the rendez-vous point for the group.

The groupId is the hash of the group's textual name concatenated with its creator's name. The hash is computed using a collision resistant hash function (e.g. SHA-1 [20]), which ensures a uniform distribution of groupIds. Since Pastry nodeIds are also uniformly distributed, this ensures an even distribution of groups across Pastry nodes.

Alternatively, we can make the creator of a group be the rendez-vous point for the group as follows: a Pastry nodeId can be the hash of the textual name of the node, and a groupId can be the concatenation of the nodeId of the creator and the hash of the textual name of the group. This alternative can improve performance with a good choice of creator: link stress and delay will be lower if the creator sends to the group often, or is close in the network to other frequent senders or many group members.

```
(1)  forward(msg, key, nextId)
(2)      switch msg.type is
(3)          JOIN :          if !(msg.group ∈ groups)
(4)                              groups = groups ∪ msg.group
(5)                              route(msg,msg.group)
(6)                          groups[msg.group].children ∪ msg.source
(7)                          nextId = null    // Stop routing the original message
```

Fig. 3. Scribe implementation of forward.

```
(1)  deliver(msg,key)
(2)      switch msg.type is
(3)          CREATE :        groups = groups ∪ msg.group
(4)          JOIN :          groups[msg.group].children ∪ msg.source
(5)          MULTICAST :     ∀ node in groups[msg.group].children
(6)                              send(msg,node)
(7)                          if memberOf(msg.group)
(8)                              invokeMessageHandler(msg.group, msg)
(9)          LEAVE :         groups[msg.group].children = groups[msg.group].children - msg.source
(10)                         if (|groups[msg.group].children| = 0)
(11)                             send(msg,groups[msg.group].parent)
```

Fig. 4. Scribe implementation of deliver.

In both alternatives, a groupId can be generated by any Scribe node using only the textual name of the group and its creator, without the need for an additional naming service. Of course, proper credentials are necessary to join or multicast messages in the associated group.

A.2 Membership management

Scribe creates a multicast tree, rooted at the rendez-vous point, to disseminate the multicast messages in the group. The multicast tree is created using a scheme similar to reverse path forwarding [21]. The tree is formed by joining the Pastry routes from each group member to the rendez-vous point. Group joining operations are managed in a decentralized manner to support large and dynamic membership.

Scribe nodes that are part of a group's multicast tree are called *forwarders* with respect to the group; they may or may not be members of the group. Each forwarder maintains a *children table* for the group containing an entry (IP address and nodeId) for each of its children in the multicast tree.

When a Scribe node wishes to join a group, it asks Pastry to route a JOIN message with the group's groupId as the key (e.g. route (JOIN,groupId)). This message is routed by Pastry towards the group's rendez-vous point. At each node along the route, Pastry invokes Scribe's forward method. Forward (lines 3 to 7 in Figure 3) checks its list of groups to see if it is currently a forwarder; if so, it accepts the node as a child, adding it to the children table. If the node is not already a forwarder, it creates an entry for the group, and adds the source node as a child in the associated children table. It then becomes a forwarder for the group by sending a JOIN message to the next node along the route from the joining node to the rendez-vous point. The original message from the source is terminated; this is achieved by setting nextId = null, in line 7 of Figure 3.

Figure 5 illustrates the group joining mechanism. The circles represent nodes, and some of the nodes have their nodeId shown. For simplicity $b = 1$, so the prefix is matched one bit at a time. We assume that there is a group with groupId 1100 whose rendez-vous point is the node with the same identifier. The node with nodeId 0111 is joining this group. In this example, Pastry routes the JOIN message to node 1001; then the message from 1001 is routed to 1101; finally, the message from 1101 arrives at 1100. This route is indicated by the solid arrows in Figure 5.

Let us assume that nodes 1001 and 1101 are not already forwarders for group 1100. The joining of node 0111 causes the other two nodes along the route to become forwarders for the group, and causes them to add the preceding node in the route to their children tables. Now let us assume that node 0100 decides to join the same group. The route that its JOIN message would take is shown using dot-dash arrows. However, since node 1001 is already a forwarder, it adds node 0100 to its children table for the group, and the JOIN message is terminated.

When a Scribe node wishes to leave a group, it records locally that it left the group. If there are no other entries in the children table, it sends a LEAVE message to its parent in the multicast tree, as shown in lines 9 to 11 in Figure 4. The message proceeds recursively up the multicast tree, until a node is reached that still has entries in the children table after removing the departing child.

The properties of Pastry routes ensure that this mechanism produces a tree. There are no loops because the nodeId of the next node in every hop of a Pastry route matches a longer prefix of the groupId than the previous node, or matches a prefix with

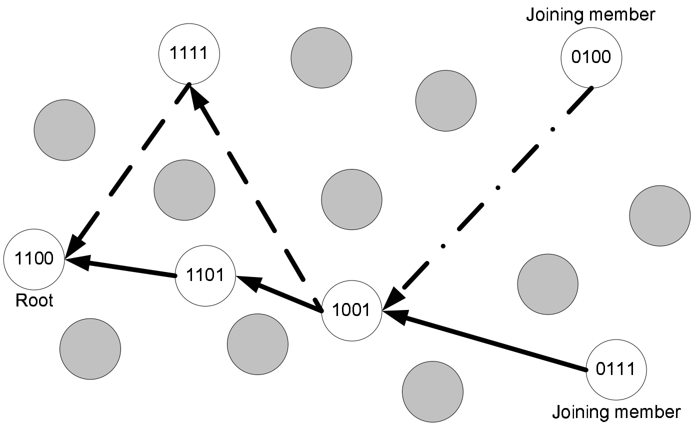


Fig. 5. Membership management and multicast tree creation

the same length and is numerically closer, or is the nodeId of the root.

The membership management mechanism is efficient for groups with a wide range of memberships, varying from one to all Scribe nodes. The list of members of a group is distributed across the nodes in the multicast tree. Pastry's randomization properties ensure that the tree is well balanced and that the forwarding load is evenly balanced across the nodes. This balance enables Scribe to support large numbers of groups and members per group. Joining requests are handled locally in a decentralized fashion. In particular, the rendez-vous point does not handle all joining requests.

The locality properties of Pastry ensure that the multicast tree can be used to disseminate messages efficiently. The delay to forward a message from the rendez-vous point to each group member is small because of the *short routes* property. Second, the *route convergence* property ensures that the load imposed on the physical network is small because most messages are sent by the nodes close to the leaves and the network distance traversed by these messages is short. Simulation results quantifying the locality properties of the Scribe multicast tree will be presented in Section IV.

A.3 Multicast message dissemination

Multicast sources use Pastry to locate the rendez-vous point of a group: they route to the rendez-vous point (e.g. route(MULTICAST, groupId)), and ask it to return its IP address. They cache the rendez-vous point's IP address and use it in subsequent multicasts to the group to avoid repeated routing through Pastry. If the rendez-vous point changes or fails, the source uses Pastry to find the IP address of the new rendez-vous point.

Multicast messages are disseminated from the rendez-vous point along the multicast tree in the obvious way (lines 5 and 6 of Figure 4).

There is a single multicast tree for each group and all multicast sources use the above procedure to multicast messages to the group. This allows the rendez-vous node to perform access control.

B. Reliability

Applications that use group multicast may have diverse reliability requirements. Some groups may require reliable and ordered delivery of messages, whilst others require only best-effort delivery. Therefore, Scribe provides only best-effort delivery of messages but it offers a framework for applications to implement stronger reliability guarantees.

Scribe uses TCP to disseminate messages reliably from parents to their children in the multicast tree and for flow control, and it uses Pastry to repair the multicast tree when a forwarder fails.

B.1 Repairing the multicast tree

Periodically, each non-leaf node in the tree sends a heartbeat message to its children. Multicast messages serve as an implicit heartbeat signal avoiding the need for explicit heartbeat messages in many cases. A child suspects that its parent is faulty when it fails to receive heartbeat messages. Upon detection of the failure of its parent, a node calls Pastry to route a JOIN message to the group's identifier. Pastry will route the message to a new parent, thus repairing the multicast tree.

For example, in Figure 5, consider the failure of node 1101. Node 1001 detects the failure of 1101 and uses Pastry to route a JOIN message towards the root through an alternative route (indicated by the dashed arrows). The message reaches node 1111 who adds 1001 to its children table and, since it is not a forwarder, sends a JOIN message towards the root. This causes node 1100 to add 1111 to its children table.

Scribe can also tolerate the failure of multicast tree roots (rendez-vous points). The state associated with the rendez-vous point, which identifies the group creator and has an access control list, is replicated across the k closest nodes to the root node in the nodeId space (where a typical value of k is 5). It should be noted that these nodes are in the leaf set of the root node. If the root fails, its immediate children detect the failure and join again through Pastry. Pastry routes the join messages to a new root (the live node with the numerically closest nodeId to the groupId), which takes over the role of the rendez-vous point. Multicast senders likewise discover the new rendez-vous point by routing via Pastry.

Children table entries are discarded unless they are periodically refreshed by an explicit message from the child, stating its desire to remain in the group.

This tree repair mechanism scales well: fault detection is done by sending messages to a small number of nodes, and recovery from faults is local; only a small number of nodes ($O(log_{2^b} N)$) is involved.

B.2 Providing additional guarantees.

By default, Scribe provides reliable, ordered delivery of multicast messages only if the TCP connections between the nodes in the multicast tree do not break. For example, if some nodes in the multicast tree fail, Scribe may fail to deliver messages or may deliver them out of order.

Scribe provides a simple mechanism to allow applications to implement stronger reliability guarantees. Applications can define the following upcall methods, which are invoked by Scribe.

forwardHandler(msg) is invoked by Scribe before the node forwards a multicast message, msg, to its children in the multicast tree. The method can modify msg before it is forwarded.
joinHandler(msg) is invoked by Scribe after a new child is added to one of the node's children tables. The argument is the JOIN message.
faultHandler(msg) is invoked by Scribe when a node suspects that its parent is faulty. The argument is the JOIN message that is sent to repair the tree. The method can modify msg to add additional information before it is sent.

For example, an application can implement ordered, reliable delivery of multicast messages by defining the upcalls as follows. The *forwardHandler* is defined such that the root assigns a sequence number to each message and such that recently multicast messages are buffered by the root and by each node in the multicast tree. Messages are retransmitted after the multicast tree is repaired. The *faultHandler* adds the last sequence number, n, delivered by the node to the JOIN message and the *joinHandler* retransmits buffered messages with sequence numbers above n to the new child. To ensure reliable delivery, the messages must be buffered for an amount of time that exceeds the maximal time to repair the multicast tree after a TCP connection breaks.

To tolerate root failures, the root needs to be replicated. For example, one could choose a set of replicas in the leaf set of the root and use an algorithm like Paxos [22] to ensure strong consistency.

IV. EXPERIMENTAL EVALUATION

This section presents results of simulation experiments to evaluate the performance of a prototype Scribe implementation. These experiments compare the performance of Scribe and IP multicast along three metrics: the delay to deliver events to group members, the stress on each node, and the stress on each physical network link. We also ran our implementation in a real distributed system with a small number of nodes.

A. Experimental Setup

We developed a simple packet-level, discrete event simulator to evaluate Scribe. The simulator models the propagation delay on the physical links but it does not model either queuing delay or packet losses because modeling these would prevent simulation of large networks. We did not model any cross traffic during the experiments.

The simulations ran on a network topology with 5050 routers, which were generated by the Georgia Tech [23] random graph generator using the transit-stub model. The routers did not run the code to maintain the overlays and implement Scribe. Instead, this code ran on 100,000 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 [10]).

The transit-stub model is hierarchical. There are 10 transit domains at the top level with an average of 5 routers in each. Each transit router has an average of 10 stub domains attached, and each stub has an average of 10 routers. We generated 10 different topologies using the same parameters but different random seeds. We ran all the experiments in all the topologies. The results we present are the average of the results obtained with each topology.

We used the routing policy weights generated by the Georgia Tech random graph generator to perform IP unicast routing. IP multicast routing used a shortest path tree formed by the merge 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) [1] or PIM [3]. But in order to provide a conservative comparison, we ignored messages required by these protocols to maintain the trees. The delay of each LAN link was set to 1ms and the average delay of core links (computed by the graph generator) was 40.7ms.

Scribe was designed to be a generic infrastructure capable of supporting multiple concurrent applications with varying requirements. Therefore, we ran experiments with a large number of groups and with a wide range of group sizes. Since there are no obvious sources of real-world trace data to drive these experiments, we adopted a Zipf-like distribution for the group sizes.

Groups are ranked by size and the size of the group with rank r is given by $gsize(r) = \lfloor Nr^{-1.25} + 0.5 \rfloor$, where N is the total number of Scribe nodes. The number of groups was fixed at 1,500 and the number of Scribe nodes (N) was fixed at 100,000, which were the maximum numbers that we were able to simulate. The exponent 1.25 was chosen to ensure a minimum group size of eleven; this number appears to be typical of Instant Messaging applications, which is one of the targeted multicast applications. The maximum group size is 100,000 (group rank 1) and the sum of all group sizes 395,247. Figure 6 plots group size versus group rank.

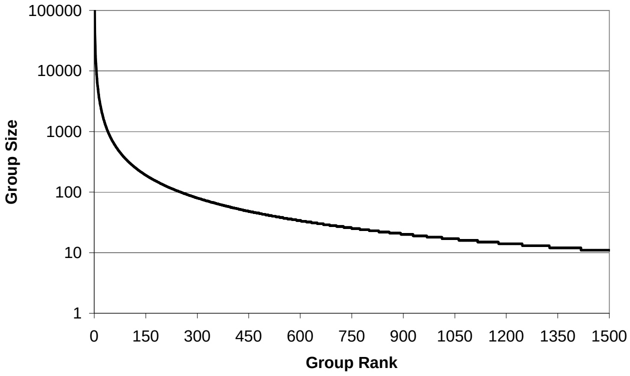


Fig. 6. Distribution of group size versus group rank.

The members of each group were selected randomly with uniform probability from the set of Scribe nodes, and we used different random seeds for each topology. Distributions with better network locality would improve the performance of Scribe.

We also ran experiments to evaluate Scribe on a different topology, which is described in [24]. This is a realistic topology with 102,639 routers that was obtained from Internet measurements. The results of these experiments were comparable with the results presented here.

B. Delay penalty

The first set of experiments compares the delay to multicast messages using Scribe and IP multicast. Scribe 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 Scribe and IP multicast. We compute two metrics of delay penalty using these distributions: *RMD* is the ratio between the maximum delay using Scribe and the maximum delay using IP multicast, and *RAD* is the ratio between the average delay using Scribe and the average delay using IP multicast. This differs from the *relative delay penalty* (RDP) used in [10], where the delay ratio was computed for each *individual* group member. RAD and RMD avoid the anomalies [15] associated with RDP.

Figure 7 shows the cumulative distribution of the RAD and RMD metrics. The y-value of a point represents the number of groups with a RAD or RMD lower than or equal to the relative delay (x-value). Scribe's performance is good because it leverages Pastry's *short routes* property. For 50% of groups, a RAD of at most 1.68 and a RMD of at most 1.69 is observed. In the worst case, the maximum RAD is 2 and the maximum RMD is 4.26.

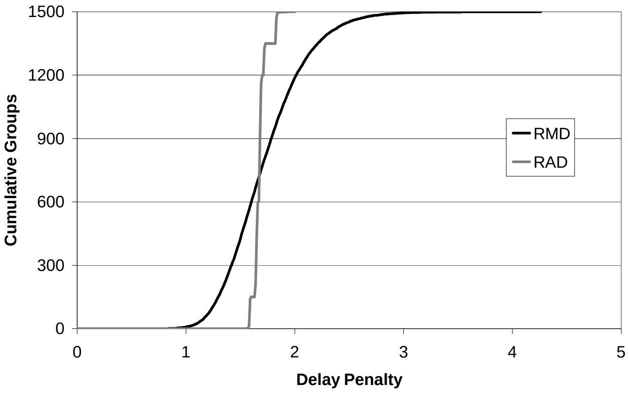


Fig. 7. Cumulative distribution of delay penalty relative to IP multicast per group (average standard deviation was 62 for RAD and 21 for RMD).

We also calculated the RDP for the 100,000 members of the group with rank 1. The results show that Scribe performs well: the mean RDP is 1.81, the median is 1.65, more than 80% of the group members have RDP less than 2.25, and more than 98% have RDP less than 4. IP routing does not always produce minimum delay routes because it is performed using the routing policy weights from the Georgia Tech model [23]. As a result, Scribe was able to find routes with lower delay than IP multicast for 2.2% of the group members.

C. Node stress

In Scribe, end nodes are responsible for maintaining membership information and for forwarding and duplicating packets whereas routers perform these tasks in IP multicast. To evaluate the stress imposed by Scribe on each node, we measured the number of groups with non-empty children tables, and the number of entries in children tables in each Scribe node; the results are in Figures 8 and 9.

Even though there are 1,500 groups, the mean number of non-empty children tables per node is only 2.4 and the median number is only 2. Figure 8 shows that the distribution has a long tail with a maximum number of children tables per node of 40. Similarly, the mean number of entries in all the children tables on any single Scribe node is only 6.2 and the median is only 3. This distribution also has a long thin tail with a maximum of 1059.

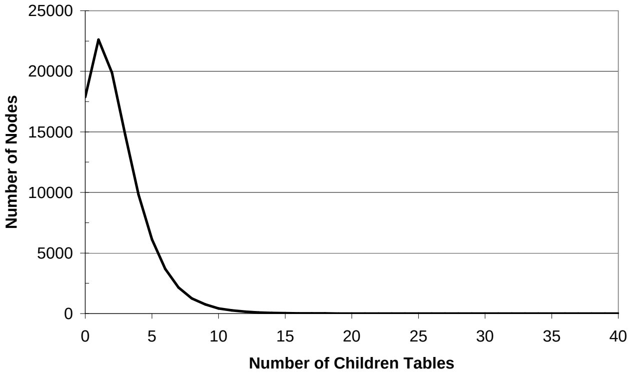


Fig. 8. Number of children tables per Scribe node (average standard deviation was 58).

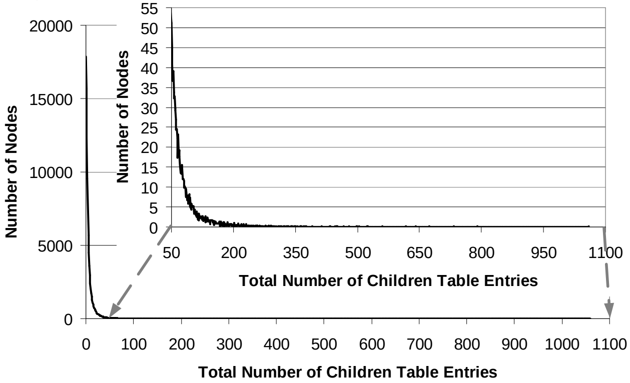


Fig. 9. Number of children table entries per Scribe node (average standard deviation was 3.2).

These results indicate that Scribe does a good job of partitioning and distributing forwarding load over all nodes: each node is responsible for forwarding multicast messages for a small number of groups, and it forwards multicast messages only to a small number of nodes. This is important to achieve scalability with group size and the number of groups.

D. Link stress

The final set of experiments compares the stress imposed by Scribe and IP multicast on each directed link in the network topology. We computed the stress by counting the number of packets that are sent over each link when a message is multicast to each of the 1,500 groups. Figure 10 shows the distribution of link stress for both Scribe and IP multicast with the results for zero link stress omitted.

The total number of links is 1,035,295 and the total number of messages over these links is 2,489,824 for Scribe and 758,853 for IP multicast. The mean number of messages per link in Scribe is 2.4 whilst for IP multicast it is 0.7. The median is 0 for both. The maximum link stress in Scribe is 4031, whilst for IP multicast the maximum link stress is 950. This means that the maximum link stress induced by Scribe is about 4 times that for a IP multicast on this experiment. The results are good be-

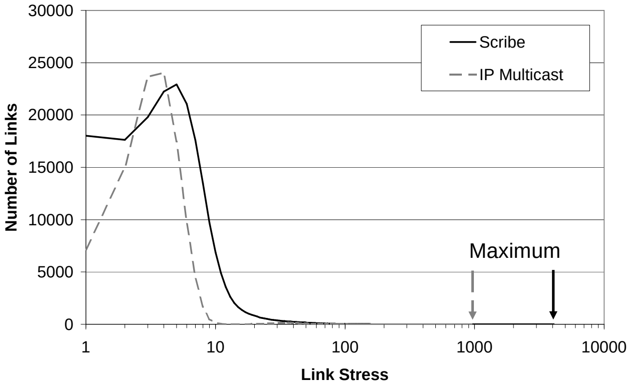


Fig. 10. Link stress for multicasting a message to each of 1,500 groups (average standard deviation was 1.4 for Scribe and for 1.9 for IP multicast).

cause Scribe distributes load across nodes (as shown before) and because it leverages Pastry's *route convergence* property. Group members that are close in the network tend to be children of the same parent in the multicast tree that is also close to them. This reduces link stress because the parent receives a single copy of the multicast message and forwards copies to its children along short routes.

It is interesting to compare Scribe with a naïve multicast that is implemented by performing a unicast transmission from the source to each subscriber. This naïve implementation would have a maximum link stress greater than or equal to 100,000 (which is the maximum group size).

Figure 10 shows the link stress for multicasting a message to each group. The link stress for joining is identical because the process we use to create the multicast tree for each group is the inverse of the process used to disseminate multicast messages

E. Bottleneck remover

The base mechanism for building multicast trees in Scribe assumes that all nodes have equal capacity and strives to distribute load evenly across all nodes. But in several deployment scenarios some nodes may have less computational power or bandwidth available than others. Under high load, these lower capacity nodes may become bottlenecks that slow down message dissemination. Additionally, the distribution of children table entries shown in Figure 9 has a long tail. The nodes at the end of the tail may become bottlenecks under high load even if their capacity is relatively high.

This section describes a simple algorithm to remove bottlenecks when they occur. The algorithm allows nodes to bound the amount of multicast forwarding they do by off-loading children to other nodes.

The *bottleneck remover* algorithm works as follows. When a node detects that it is overloaded, it selects the group that consumes the most resources. Then it chooses the child in this group that is farthest away, according to the proximity metric. The parent drops the chosen child by sending it a message containing the children table for the group along with the delays between each child and the parent. When the child receives the message, it performs the following operations: *(i)* it measures the delay between itself and other nodes in the children table it received from the parent; *(ii)* then for each node, it computes the total delay between itself and the parent via each of the nodes; *(iii)* finally, it sends a join message to the node that provides the smallest combined delay. That way, it minimizes the delay to reach its parent through one of its previous siblings.

Unlike the base mechanism for building multicast trees in Scribe, the bottleneck remover may introduce routing loops. However, this happens only when there are failures and with low probability. In particular, there are no routing loops in the experiments that we describe below. Loops are detected by having each parent p propagate to its children the nodeIds in the path from the root to p. If a node receives a path that contains its nodeId, it uses Pastry to route a JOIN message towards the group identifier using a randomized route. Additionally if a node receives a JOIN message from a node in its path to the root, it rejects the join and informs the joining node that it should join using a randomized route.

We reran all the experiments in the previous sections to evaluate the bottleneck remover. Since we do not model bandwidth or processing at the nodes in our simulator, the cost of forwarding is the same for all children. A node is considered overloaded if the total number of children across all groups is greater than 64, and the group that consumes the most resources is the one with the largest children table.

Figure 11 shows the distribution of the number of children table entries per node. As expected, the bottleneck remover eliminates the long tail that we observed in Figure 9 and bounds the number of children per node to 64.

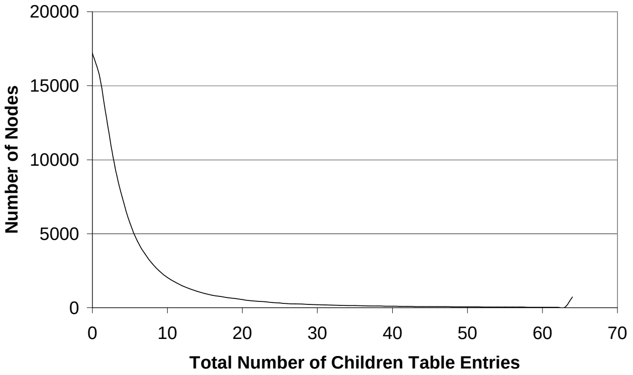


Fig. 11. Number of children table entries per Scribe node with the bottleneck remover (average standard deviation was 57).

The drawback with the bottleneck remover is that it increases the link stress for joining. The average link stress increases from 2.4 to 2.7 and the maximum increases from 4031 to 4728. This does not account for the probes needed to estimate delay to other siblings; there are an average of 3 probes per join. Our experimental setup exacerbates this cost; the bottleneck remover is invoked very often because all nodes impose a low bound on the number of children table entries. We expect this cost to be low in a more realistic setting.

We do not show figures for the other results because they are almost identical to the ones presented without the bottleneck remover. In particular, the bottleneck remover achieves the goal of bounding the amount of forwarding work per node without noticeably increasing latency.

F. Scalability with many small groups

We ran an additional experiment to evaluate Scribe's scalability with a large number of groups. This experiment ran in the same setup as the others except that there were 50,000 Scribe nodes, and 30,000 groups with 11 members each (which was the minimum group size in the distribution used in the previous experiments). This setup is representative of Instant Messaging applications.

Figures 12 and 13 show the distribution of children tables and children table entries per node, respectively. The lines labelled "scribe collapse" will be explained later. The distributions have sharp peaks before 50 and a long thin tail, showing that Scribe scales well because it distributes children tables and children table entries evenly across the nodes.

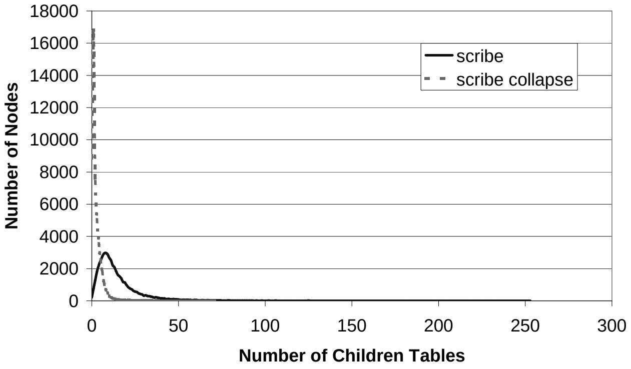


Fig. 12. Number of children tables per Scribe node.

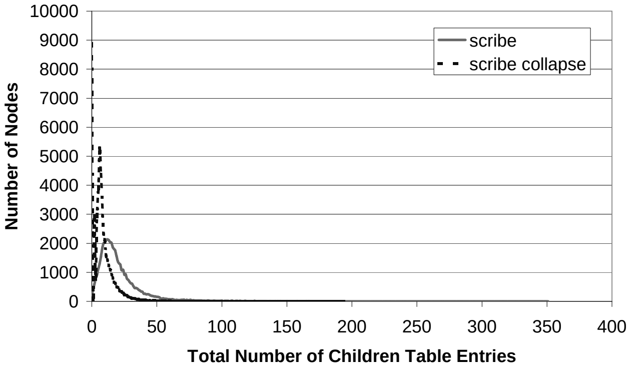


Fig. 13. Number of children table entries per Scribe node.

But the results also show that Scribe multicast trees are not as efficient for small groups. The average number of children table entries per node is 21.2, whereas the naïve multicast would achieve an average of only 6.6. The average is higher for Scribe because it creates trees with long paths with no branching. This problem also causes higher link stress as shown in Figure 14: Scribe's average link stress is 6.1, IP multicast's is 1.6 and naïve multicast's is 2.9. (As before, one message was sent in each of the 30,000 groups).

We implemented a simple algorithm to produce more efficient trees for small groups. Trees are built as before but the algorithm *collapses* long paths in the tree by removing nodes that are not members of a group and have only one entry in the group's children table. We reran the experiment in this section using this algorithm. The new results are shown under the label "scribe collapse" in Figures 12, 13, and 14. The algorithm is effective: it reduced the average link stress from 6.1 to 3.3, and the average number of children per node from 21.2 to 8.5.

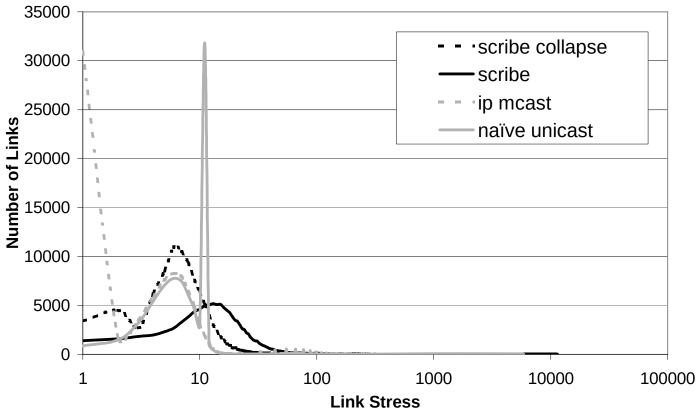


Fig. 14. Link stress for multicasting a message to each of 30,000 groups.

We also considered an alternative algorithm that grows trees less eagerly. As before, a joining node, c, uses Pastry to route a JOIN message towards the root of the tree. But these messages are forwarded to the first node, p, that is already in the tree. If p is not overloaded, it adds c to its children table and the previous nodes along the route do not become forwarders for the tree. Otherwise, p adds the previous node in the route to its children table, and tells this node to take c as its child. This alternative algorithm can generate shallower trees but it has two disadvantages: it can increase link stress relative to the algorithm that collapses the tree; and it reduces Scribe's ability to handle large numbers of concurrent joins when a group suddenly becomes popular.

V. Related work

Like Scribe, Overcast [25] and Narada [10] implement multicast using a self-organizing overlay network, and they assume only unicast support from the underlying network layer. Overcast builds a source-rooted multicast tree using end-to-end measurements to optimize bandwidth between the source and the various group members. Narada uses a two step process to build the multicast tree. First, it builds a mesh per group containing all the group members. Then, it constructs a spanning tree of the mesh for each source to multicast data. The mesh is dynamically optimized by performing end-to-end latency measurements and adding and removing links to reduce multicast latency. The mesh creation and maintenance algorithms assume that all group members know about each other and, therefore, do not scale to large groups.

Scribe builds a multicast tree per group on top of a Pastry overlay, and relies on Pastry to optimize the routes from the root to each group member based on some metric (e.g. latency). The main difference is that the Pastry overlay can scale to an extremely large number of nodes because the algorithms to build and maintain the network have space and time costs of $O(log_{2^b}N)$. This enables support for extremely large groups and sharing of the Pastry network by a large number of groups.

The recent work on Bayeux [9] and Content Addressable Network (CAN) multicast [26] is the most similar to Scribe. Both Bayeux and CAN multicast are built on top of scalable peer-to-peer object location systems similar to Pastry. Bayeux is built on top of Tapestry [13] and CAN multicast is built on top of CAN [15].

Like Scribe, Bayeux supports multiple groups, and it builds a multicast tree per group on top of Tapestry. However, this tree is built quite differently. Each request to join a group is routed by Tapestry all the way to the node acting as 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.

Bayeux has two scalability problems when compared to Scribe: it requires nodes to maintain more group membership information, and it generates more traffic when handling group membership changes. In particular, the root keeps a list of all group members and all group management traffic must go through the root. Bayeux proposes a multicast tree partitioning mechanism to ameliorate these problems by splitting the root into several replicas and partitioning members across them. But this only improves scalability by a small constant factor.

In Scribe, the expected amount of group membership information kept by each node is small because this information is distributed over the nodes. Furthermore, it can be bounded by a constant independent of the number of group members by using the bottleneck removal algorithm. Additionally, group join and leave requests are handled locally. This allows Scribe to scale to extremely large groups and to deal with rapid changes in group membership efficiently.

Finally, whilst Scribe and Bayeux have similar delay characteristics, Bayeux induces a higher link stress than Scribe. Both Pastry and Tapestry (on which Bayeux is built) exploit network locality in a similar manner. With each successive hop taken within the overlay network from the source towards the destination, the message traverses an exponentially increasing distance in the proximity space. In Bayeux, the multicast tree consists of the routes from the root to each destination, while in Scribe the tree consists of the routes from each destination to the root. As a result, messages traverse the many long links near the leaves in Bayeux, while in Scribe, messages traverse few long links near the root.

CAN multicast does not build multicast trees. Instead, it uses the routing tables maintained by CAN to flood messages to all nodes in a CAN overlay network, and it supports multiple groups by creating a separate CAN overlay per group. A node joins a group by looking up a contact node for the group in a global CAN overlay, and by using that node to join the group's overlay. Group leaves are handled by the regular CAN maintenance algorithm.

CAN multicast has two features that may be advantageous in some scenarios: group traffic is not restricted to flow through a single multicast tree, and only group members forward multicast traffic for a group. But it is significantly more expensive to build and maintain separate CAN overlays per group than Scribe multicast trees. Furthermore, the delays and link stresses reported for CAN multicast in [26] are significantly higher than those observed for Scribe. Taking network topology into account when building CAN overlays is likely to reduce delays and link stresses but it will increase the cost of overlay construction and maintenance further. Additionally, the group join mechanism in CAN multicast does not scale well. The node in the CAN overlay that supplies the contact node for the group and the contact node itself handle all join traffic. The authors of [26] suggest replicating the functionality of these nodes to avoid the problem but this only improves scalability by a constant factor.

The mechanisms for fault resilience in CAN multicast, Bayeux and Scribe are also very different. CAN multicast does not require any additional mechanism to handle faults besides what is already provided by the base CAN protocol. Bayeux and Scribe require separate mechanisms to repair multicast trees. All the mechanisms for fault resilience proposed in Bayeux are sender-based whereas Scribe uses a receiver-based mechanism. Bayeux does not provide a mechanism to handle root failures whereas Scribe does.

VI. Conclusions

We have presented Scribe, a large-scale and fully decentralized application-level multicast infrastructure built on top of Pastry, a peer-to-peer object location and routing substrate overlayed on the Internet. Scribe is designed to scale to large numbers of groups, large group size, and supports multiple multicast source per group.

Scribe leverages the scalability, locality, fault-resilience and self-organization properties of Pastry. The Pastry routing substrate is used to maintain groups and group membership, and to build an efficient multicast tree associated with each group. Scribe's randomized placement of groups and multicast roots balances the load among participating nodes. Furthermore, Pastry's properties enable Scribe to exploit locality to build an efficient multicast tree and to handle group join operations in a decentralized manner.

Fault-tolerance in Scribe is based on Pastry's self-organizing properties. The default reliability scheme in Scribe ensures automatic repair of the multicast tree after node and network failures. Multicast message dissemination is performed on a best-effort basis. However, stronger reliability models can be easily layered on top of Scribe.

Our simulation results, based on a realistic network topology model, indicate that Scribe scales well. Scribe is able to efficiently support a large number of nodes, groups, and a wide range of group sizes. Hence Scribe can concurrently support applications with widely different characteristics. Results also show that it balances the load among participating nodes, while achieving acceptable delay and link stress, when compared to network-level (IP) multicast.

References

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

[2] S. E. Deering, *Multicast Routing in a Datagram Internetwork*, Ph.D. thesis, Stanford University, Dec 1991.

[3] 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.

[4] S. Floyd, V. Jacobson, C.G. liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transaction on networking*, vol. 5, no. 4, pp. 784–803, Dec. 1997.

[5] J.C. Lin and S. Paul, "A reliable multicast transport protocol," in *Proc. of IEEE INFOCOM'96*, 1996, pp. 1414–1424.

[6] K.P. Birman, M. Hayden, O.Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, no. 2, pp. 41–88, May 1999.

[7] Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov, "Lightweight probabilistic broadcast," in *Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, July 2001.

[8] Luis F. Cabrera, Michael B. Jones, and Marvin Theimer, "Herald: Achieving a global event notification service," in *HotOS VIII*, Schloss Elmau, Germany, May 2001.

[9] Shelly Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz, "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination," in *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, Port Jefferson, NY, June 2001.

[10] Yang hua Chu, Sanjay G. Rao, and Hui Zhang, "A case for end system multicast," in *Proc. of ACM Sigmetrics*, Santa Clara, CA, June 2000, pp. 1–12.

[11] P.T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," Tech. Rep. DSC ID:2000104, EPFL, January 2001.

[12] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[13] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

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

[16] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks," 2002, Submitted for publication (http://www.research.microsoft.com/∼antr/pastry). Also available as Microsoft Research TR MSR-TR-2002-82.

[17] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach, "Security for structured peer-to-peer overlay networks," in *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[18] Peter Druschel and Antony Rowstron, "PAST: A persistent and anonymous store," in *HotOS VIII*, Schloss Elmau, Germany, May 2001.

[19] Antony Rowstron and Peter Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[20] FIPS 180-1, "Secure hash standard," Tech. Rep. Publication 180-1, Federal Information Processing Standard (FIPS), National Institute of Standards and Technology, US Department of Commerce, Washington D.C., April 1995.

[21] Yogen K. Dalal and Robert Metcalfe, "Reverse path forwarding of broadcast packets," *Communications of the ACM*, vol. 21, no. 12, pp. 1040–1048, Dec. 1978.

[22] L. Lamport, "The Part-Time Parliament," Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[23] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *INFOCOM96*, San Francisco, CA, 1996.

[24] H. Tangmunarunkit, R. Govindan, D. Estrin, and S. Shenker, "The impact of routing policy on internet paths," in *Proc. 20th IEEE INFOCOM*, Alaska, USA, Apr. 2001.

[25] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, "Overcast: Reliable Multicasting with an Overlay Network," in *Proc. of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000, pp. 197–212.

[26] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of the Third International Workshop on Networked Group Communication*, London, UK, Nov. 2001, pp. 14–29.