SCRIBE: The design of a large-scale event notification infrastructure

Antony Rowstron[1], Anne-Marie Kermarrec[1], Peter Druschel[2], and Miguel Castro[1]

[1]Microsoft Research Ltd., St. George House,

1 Guildhall Street, Cambridge, CB2 3NH, UK.

[2]Rice University MS-132, 6100 Main Street,

Houston, TX 77005-1892, USA.

PRELIMINARY DRAFT

Abstract

In this paper we outline the design of Scribe, a reliable large-scale event notification infrastructure, built on top of a peer-to-peer object location and routing infrastructure overlayed on the Internet (Pastry). Scribe provides large-scale distributed applications with a highly flexible group communication protocol. Scribe leverages Pastry [1] and benefits from its robustness, reliability, self-organization and locality properties. The peer-to-peer model of Pastry is very well suited to a scalable implementation of group communication. The Pastry routing scheme is used to create a group (a topic) and to build an efficient multicast tree for the dissemination of events within groups composed of a potentially large set of members (subscribers). Scribe is highly scalable and can support potentially billions of topics and subscribers per topic and multiple publishers per topic. Scribe relies on an efficient tree control mechanism, which reconfigures the multicast tree to track changes in group membership and use.

1 Introduction

Scalable event notification is an important component of many Internet-wide distributed applications. Its publish-subscribe paradigm is well-suited to the loosely coupled nature of many such applications. Subscribers register their interest in an event or a pattern of events; once subscribed, they are asynchronously notified of any event matching their interest, regardless of the event's source. Topic-based publish-subscribe is similar to group-based communication; subscribing to a topic is equivalent to becoming a member of a group. Events are associated with a particular topic, and all the subscribers of that topic receive the event.

The publish-subscribe paradigm is very general. Consequently, applications built around the publish-subscribe paradigm have varying characteristics, both in terms of the number of subscribers and the number of publishers. Within the same application, the number of publishers and subscribers can vary over time. For instance, a flash-crowd phenomenon can occur, where the number of subscribers increases suddenly.

Two example applications with very different numbers of subscribers per topic are instant messaging/presence monitoring and topical news (e.g. sports result) dissemination. Instant messaging/presence notification is typically characterised by a small group of subscribers ('buddies') for each topic (an individual), whilst in the topical news dissemination each topic can have millions of subscribers and potentially multiple publishers.

This motivates the need for a general-purpose, highly-flexible and reliable event-notification infrastructure capable of supporting 10^9 topics and subscribers per topic. IP multicast is not widely deployed and lacks reliability guarantees, and this motivates the need for application-level multicast protocols. Appropriate algorithms and systems for scalable subscription management and scalable, reliable propagation of events are still an active research area [2, 3, 4]. A particularly promising approach, exemplified by Scribe, is to employ an application-level multicast protocol built on top of a scalable, self-organizing and secure peer-to-peer overlay network.

In this paper we sketch the design and a preliminary evaluation of Scribe, a large-scale event notification infrastructure. Scribe is designed to be an efficient and highly-flexible application-level multicast protocol able to scale to a large number of subscribers, publishers and topics. Scribe is built as a service on top of *Pastry*, a scalable, secure and self-organizing peer-to-peer overlay network [1]. Scribe is a peer-to-peer system with no centralised components.

A group communication protocol or publish subscribe system is designed to maintain topic or group membership information and disseminate events. For both of these activities, centralization is inappropriate as it limits scalability with respect to both the number of topics and the number of subscribers per topic. Better suited is the decentralised model of a peer-to-peer system, where each participating node has equal responsibilities. Load balancing is achieved in Scribe, because the membership management is fully distributed over the participating nodes.

Scribe uses Pastry to subscribe, unsubscribe and raise events on a particular topic. Scribe uses an efficient tree control mechanism. As the number of subscribers increases for a given topic, Scribe adjusts the configuration of the associated multicast tree. The depth of the multicast tree automatically adapts to the number of subscribers, thus achieving a balance between the load on individual nodes and the latency of event delivery. Moreover, Scribe leverages the robustness, self-organization, security, locality and reliability properties of Pastry.

The rest of the paper is organized as follows. Section 2 gives an overview of the Pastry routing and object location substrate. Section 3 describes the design of Scribe. We present some preliminary performance results in Section 4 and discuss related work in Section 5. Section 6 offers conclusions and outlines future work.

2 Pastry

In this section we briefly describe Pastry [1], a peer-to-peer location and routing substrate that provides the basic infrastructure for Scribe. Pastry forms a secure, 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 existing nodeIds are uniformly distributed. The basic capability Pastry provides is to efficiently and reliably route messages towards the node whose nodeId is numerically closest to a given destination nodeId (destId), among all live nodes. Moreover, Pastry routes have good locality properties. At each routing step, a message is forwarded to a node whose nodeId shares a longer prefix with the destId than the current node, while travelling the least possible distance in the underlying Internet. Distance is defined here according to a scalar proximity metric, such as the number of IP hops.

Pastry is highly efficient, secure, scalable, fault resilient and self-organizing. 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 $\lfloor l/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously (l is a configuration parameter with typical value 16).

The tables required in each Pastry node have only $(2^b - 1) * \lceil log_{2^b} N \rceil + 2l$ 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 performing $O(log_{2^b} N)$ remote procedure calls (RPCs). In the following, we give a brief overview of the Pastry routing scheme. More detailed information about Pastry can be found in [1].

For the purpose of routing, nodeIds are thought of as a sequence of digits with base 2^b. A node's routing table is organized into levels with $2^b - 1$ entries each. The $2^b - 1$ entries at level n of the routing table each refers to a node whose nodeId shares 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. Note that an entry in the routing table points to one of potentially many nodes whose nodeId have the appropriate prefix. Among such nodes, the one closest to the present node (according to the proximity metric) is chosen in practice.

In addition to the routing table, each node maintains pointers to the set of l nodes whose nodeIds are numerically closest to the present node's nodeId, irrespective of prefix. (More precisely, the set contains $l/2$ nodes with larger and $l/2$ with smaller nodeIds). This set is called the *leaf set*. Figure 1 depicts the state of a hypothetical Pastry node with the nodeId 10233102 (base 4), in a system that uses 16 bit nodeIds and a value of $b = 2$.

In each routing step, a node normally forwards the message to a node whose nodeId shares with the destId a prefix that is at least one digit (or b bits) longer than the prefix that the destId shares with the present node's id. If no such node exists, the message is forwarded to a node whose nodeId shares a prefix with the destId as long as the current node, but is numerically closer to the destId than the present node's id. It follows from the definition of the leaf set that such a node exists in the leaf set unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously.

2.1 Locality

Pastry can route messages to any node in $\lceil log_{2^b} N \rceil$ steps in the common case. Another issue is the distance (in terms of the proximity metric) a message is travelling. Recall that the entries in the node routing tables are chosen to refer to the nearest node with

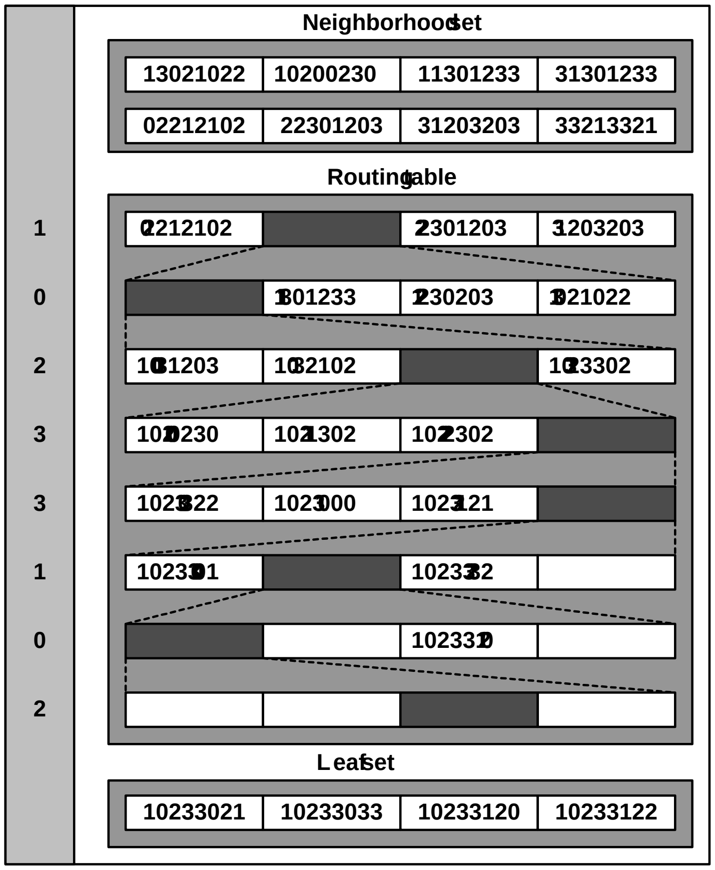

Figure 1: State of a hypothetical Pastry node with nodeId 10233102, $b = 2$. All numbers are in base 4. The top row of the routing table represents level zero. The neighborhood set is not used in routing, but is needed during node addition/recovery.

the appropriate nodeId prefix. As a result, in each step a message is routed to the nearest node with a longer prefix match (by one digit). While this local decision process clearly can't achieve globally shortest routes, simulations have shown the average distance travelled by a message is only 40% higher than the distance between the source and destination in the underlying network [1].

2.2 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, leaf set and neighbourhood sets, in the presence of node failures, node recoveries, and new node arrivals. The protocol is described and evaluated in [1].

Briefly, an arriving node with the new 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 to the existing node Z with nodeId numerically closest to X. X then obtains the leaf set from Z, the neighbourhood set from A, 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 it state and notify nodes that need to know of its arrival, thereby restoring all of Pastry's invariants.

To handle node failures, neighbouring 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 to restore the invariant. 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 [1].

2.3 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:

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. Pastry's security model is discussed in [5].

route(msg,destId) causes Pastry to route the given message to the node with nodeId numerically closest to destId, among all live Pastry nodes.

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

deliver(msg,destId) called by Pastry when a message is received and the local node's nodeId is numerically closest to destId, among all live nodes.

forward(msg,destId,nextId) called by Pastry just before a message 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 [5, 6].

3 Scribe

Scribe is a scalable event notification infrastructure built on top of Pastry. It allows nodes to create event *topics*. Other nodes can then register interest in specific topics thereby becoming *subscribers* to the topic. Scribe disseminates events published to a topic to all the topic's subscribers. Events are delivered reliably (under certain assumptions), and events sent by the same publisher are delivered in the same order they were sent. Nodes can subscribe or publish to many topics, and topics can have many publishers and subscribers. Scribe is scalable because it can support many topics and many subscribers per topic.

Scribe offers a simple API to applications:

Create(credentials, topicId) creates a topic with topicId, where credentials allow a mechanism for checking that that the node creating the topic has the authority to do so.

Subscribe(credentials, topicId, eventHandler) causes the node to subscribe to a topic, and all events that are received for that topic are passed to the event handler specified. The credentials are used to perform access control. When this call returns, all subsequent events published to the topic will be received[1].

Unsubscribe(credentials, topicId) causes the node to unsubscribe to a topic.

[1]Within the design of Scribe it is possible that events that have been published to a topic before the subscription to a topic can be retrieved. However, the control and mechanism for controlling this is not yet fully designed, and is therefore, currently omitted.

Publish(credentials, topicId, event) causes the event to be published to the topic specified provided the credentials allow the caller to do this.

The next section sketches the design of Scribe. We are currently finalizing a security model for Scribe and analysing its properties and, therefore, these are not described in this paper.

3.1 Design

Scribe uses the Pastry overlay network to manage topic creation and subscription lists, and to disseminate events published to the various topics. It uses Pastry to choose and locate a *rendez-vous point* for each topic. Then, it creates a multicast tree to disseminate the events published to the topic. Subscriptions and unsubscriptions to a topic are managed in a decentralized way to enable Scribe to support large subscription lists and fast changes in these lists.

Scribe creates a separate tree for each topic that is rooted at the topic's rendez-vous point. It builds the tree on top of the Pastry network using a scheme similar to reverse path forwarding [7]: the tree is formed by the set containing the reverse of the Pastry routes from each subscriber to the rendez-vous point.

Both Pastry and Scribe are fully decentralised, all decisions are made locally with every node having the same level of responsibility, and every node being symmetric. A node can act as a publisher, a root of a rendez-vous point, a subscriber to a topic, or a forwarder in the multicast tree (or any sensible combination of these). The scalability and reliability of Scribe and Pastry relies on this peer-to-peer model.

For the sake of clarity, we first describe the base mechanism used to build the multicast trees, manage topic creation and subscription. Section 3.3 discusses event publishing and dissemination. Section 3.4 describes an improved mechanism that reduces multicast latency and space overhead. The techniques used to maintain the tree when nodes fail and to provide reliable event delivery are discussed in Section 3.5.

3.2 Base mechanism

To create a topic with identifier *topicId*, a CREATE message is routed using Pastry to the node with the *nodeId* numerically closest to *topicId*. This node becomes the *rendez-vous point* for the topic. The *topicId* is the hash of the topic's textual name concatenated with its creator's name. The hash is computed using a collision resistant hash function (e.g. SHA-1 [8]). This ensures an even distribution of topics across

Pastry nodes because *nodeIds* have a uniform distribution. Additionally, rendez-vous points can be located using Pastry given only the textual names of the topic and its creator without the need for an additional naming service indirection.

We are considering alternative techniques to name topics and locate rendez-vous points. For example, we could choose the rendez-vous point to be the node with *nodeId* closest to the topic creator's *nodeId*. Therefore, the creator would be the rendez-vous point for the topic but when it failed the node with the next closest *nodeId* would take over. We would then use Pastry to map textual names to the *nodeId* of the topic creator. This can be advantageous when the creator publishes events to the topic frequently because it is the root of the multicast tree in the common case.

Scribe creates a separate multicast tree for each topic that is rooted at the topic's rendez-vous point. The nodes in the multicast tree are called *forwarders*. Some forwarders are subscribers but others are not. Each forwarder maintains a *forwarding table* (per topic) with the IP addresses of its children in the multicast tree.

A node can subscribe to a topic by using Pastry to route a JOIN message with the topic's *topicId* as the destination. This message will be routed towards the rendez-vous point for the topic, which is the root of the tree. Each node along the route checks if it is already a forwarder for the topic. If it isn't, it sends the JOIN message to the next node along the route and becomes a forwarder for the topic. If it is already a forwarder, it does not send the message again. In either case, it adds the previous node along the route to its forwarding table for the topic.

Figure 2 illustrates the base subscription 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. In this figure, we assume that there is a topic with *topicId* 1100 whose rendez-vous point is the node with the same identifier. The node with *nodeId* 0111 is subscribing to this topic. In this example, Pastry will route the JOIN message through nodes 1001 and 1101 before it arrives at 1100. This route is represented with the solid lines in Figure 2.

Let us assume that nodes 1001 and 1101 are not forwarders for topic 1100 at the beginning. The subscription of node 0111 causes the other two nodes along the route to become forwarders for the topic, and to add the preceding node in the route to their forwarding tables. Now let us assume that node 0100 decides to subscribe to the same topic. The route of its JOIN message is shown using a dot-dash line. Since node 1001 is already a forwarder, it adds node 0100 to its forwarding table for the topic, and the JOIN message is not routed any further. Unsubscription requests are handled in a similar fashion as described in Section 3.4.2.

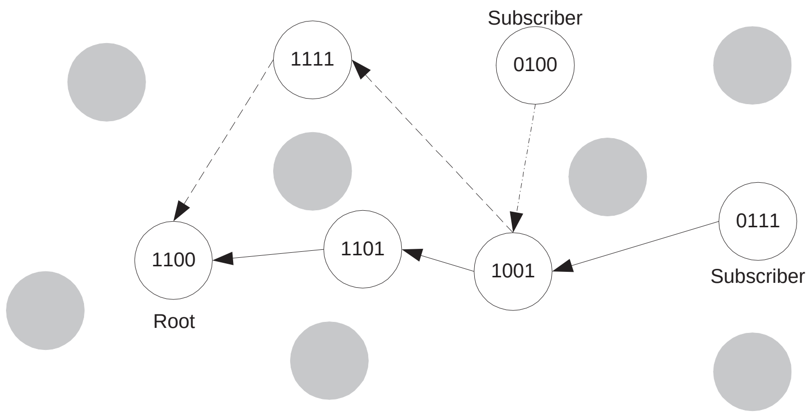

Figure 2: Base Mechanism for Subscription and Multicast Tree Creation.

This mechanism is well-suited to topics with a large number of subscribers. The list of subscribers to the topic is distributed across the nodes in the multicast tree. The randomization properties of Pastry ensure that the tree is well balanced and that the forwarding load is evenly balanced across the nodes. This enables Scribe to support extremely large groups.

Additionally, subscription requests are handled locally in a distributed fashion. In particular, the root does not need to handle all subscription requests, and the locality properties of Pastry ensure that most JOIN messages reach a forwarder that is topologically close (according to Pastry's distance metric). This enables Scribe to provide efficient support for topics whose subscription lists change rapidly.

The other interesting property is that most subscribers are children of a forwarder that is topologically close. This reduces the number of duplicates of the same packet that are sent in the same physical network link, which improves scalability to large groups.

3.3 Event dissemination

Publishers to a topic use Pastry to locate the root of the topic's multicast tree. To raise an event on a given topic for the first time, a publisher sends an appropriate request using the Pastry operation `route(event,topicId)`. Pastry routes the event to the root of the tree. Events are disseminated from the root down to the leaves of the multicast tree in the obvious way.

To improve performance, a publisher caches the IP addresses of the root for each topics it publishes to. This allows it to subsequently send events for dissemination directly to the root without routing through Pastry, unless a failure has occurred.

There is a single multicast tree for each topic and all publishers use the procedure above to publish events to the topic. This has the advantage that it allows the root to perform access control, and it can be used to ensure ordering guarantees across events from different publishers.

An alternative would be for each publisher to the topic to attach to the multicast tree and flood events over the tree from its attachment point. This has the potential of decreasing latency for all subscribers that descend from the same child of the root as the publisher. Unfortunately, these are only $\frac{1}{16}$ (with $b = 4$) of all subscribers on average. For the remaining subscribers, latency increases because the event is routed (less efficiently) to the root before it can be forwarded to them. Additionally, note that all events must be received by the root in both alternatives; the root's bandwidth and processing power limit the rate of event dissemination in both alternatives.

A third alternative would be to build a separate multicast tree rooted at each publisher to the topic. This has the potential to decrease both latency and improve throughput when compared to the previous alternatives. However, it complicates tree maintenance, and increases space overhead and message traffic to maintain the trees. Latency would not improve by more than a factor of two for topics with a large number of subscribers, and bandwidth would likely not improve because the bottleneck in systems like this is usually the link to the slowest subscriber. Therefore, we rejected this alternative.

3.4 Tree maintenance

For topics with few subscribers, the basic subscription mechanism described above produces deep trees that have long paths with no branching. This unnecessarily increases the number of forwarders, and the latency for event dissemination. This section describes a mechanism to adjust the height of a topic's multicast tree as the number of subscribers and the load on nodes varies. The tree grows when the number of subscribers or the load on forwarders increases, and it shrinks when the parent of a node acting as a forwarder in the multicast tree is capable of handling the load.

3.4.1 Growing the tree

When growing a tree, nodes do not automatically become forwarders for a topic when they receive a JOIN message. Instead, the message is forwarded along the Pastry route to the root until it reaches a node that is already a forwarder (or is the root). Then, this node decides whether it can add the subscriber to its forwarding table. This decision is made by examining the local resources of the node, e.g., the total number of entries in the forwarding tables of all the topics it forwards, the available network bandwidth, and its processing power. If the node is not overloaded, it adds the subscriber to its forwarding table for the topic. Otherwise, the preceding node p in the route is added to the forwarding table; p uses the same procedure to decide whether to add the original subscriber to its forwarding table.

To help clarify this, consider Figure 2 again. If the root node (1100) is not overloaded when 0111 subscribes to *topicId* 1100, it will add 0111 to its forwarding table directly, and 1101 and 1001 do not become forwarders. But if the root is overloaded, 1101 will be added to its forwarding table. Then, 1101 may add 0111 to its forwarding table if it is not overloaded.

Of course, we need a mechanism for an overloaded node to shed load by dropping entries from its forwarding table. Otherwise, subscribers that Pastry routes to an overloaded node would be unable to join the multicast tree. This mechanism classifies nodes according to their distance in Pastry hops: *hops*$(t, p, q) = k$ if the Pastry route from p towards the root of topic t has k hops between p and q.

When a node q becomes overloaded, it rejects requests to add any node p with *hops*$(t, p, q) > 1$ to its forwarding table for t. But it always accepts requests to add any node p with *hops*$(t, p, q) = 1$. In the latter case, node q attempts to reduce its load by removing any node o in its forwarding table for some topic t' such that *hops*$(t', o, q) > 1$. This is done by informing o that it should send a new JOIN message for topic t'. This new message may reach q again but q will add the preceding node in the route to its forwarding table rather than o, thereby growing the multicast tree.

We are still experimenting with different heuristics to select which forwarding table entries to drop. Here, we describe the heuristic we use in the experimental results presented in this paper. The overloaded node q chooses topic t' and node o to drop as follows:

1. t' is the topic with the shortest prefix match with q's *nodeId* such that q's forwarding table for t' has an entry with distance greater than 1 hop. If more than one topic satisfies this condition, q chooses one of them randomly with uniform probability.

2. o is the entry in q's forwarding table for t' with the minimum value of *hops*(t', o, q) greater than 1. If there are several entries with the minimum value, q chooses

one of them randomly with uniform probability.

The choice of t' is designed to keep multicast trees small for topics with a small number of subscribers; trees for such topics have forwarding table entries only on nodes that match a large prefix of the topic's identifier. The choice of o is designed to reduce the overhead to grow the tree, and reduce the number of forwarders in the tree that are not subscribers.

There is an additional problem that is important to address. Due to the properties of Pastry routing, the average number of nodes p such that *hops*$(t, p, q) = 1$ increases with the length of the prefix of t's *topicId* that is matched by q's *nodeId*. For example, if q is the root of the multicast tree for topic t it can be chosen as a representative of a domain at any level for Pastry routing. If nodes are uniformly distributed in topological space and all Pastry nodes subscribe to a topic, the root of the topic is expected to have an average of 15 nodes (when b=4), which have no common prefix with the topic's *topicId*, in its forwarding table. These nodes chose the root as their representative in level 0 for the domain where the *topicId* lies but there are $\frac{N}{16} - 1$ other nodes that could be chosen as the representative at level 0.

The problem can be fixed by having an overloaded node q that matches a topic t's *topicId* to level l drop from its forwarding table for t nodes that match the *topicId* to level $m < l - 1$. These nodes are asked to send a new JOIN message for the topic using an alternative representative instead of node q. Pastry nodes maintain redundant information to provide fault tolerance, which makes routing through an alternative possible in most cases.

3.4.2 Shrinking the tree

The multicast tree for a topic should shrink when the number of subscribers to the topic decreases. This improves latency and reduces resource consumption (forwarding table space, processing, and bandwidth) at forwarders that are pruned. This section describes the handling of unsubscriptions and a mechanism to shrink a tree.

When a node wishes to unsubscribe to a topic it proceeds as follows. If it does not have any entry in the forwarding table for that topic, it sends a LEAVE message to its parent in the multicast tree, and the parent removes the node from its forwarding table. Otherwise, the node simply registers that it is no longer a subscriber.

When the number of entries in a node's forwarding table for a topic drops below a threshold, it sends a SHRINK message to its parent asking if it could take these entries. The message contains the entries and indicates if the sender is a subscriber. The parent checks its load, decides whether or not to accept the request, and sends the appropriate reply to the requester. If the request is accepted, the requester clears its forwarding table, and the parent adds the new entries to its forwarding table and removes the requester if it is not a subscriber.

The SHRINK message is sent when the number of entries in the forwarding table drops below a threshold for a certain period of time. It is important to wait for a certain period of time to prevent instability when the tree is growing. A node also waits for the same period of time after a request to shrink the tree is denied. The values for both the threshold and the time period are still under study.

In the example of Figure 2, consider that the node 1001 has only two entries in its forwarding table for topic 1100 and this is below the threshold. Node 1001 will send a shrink request to its parent to take over nodes 0100 and 0111. If the parent node accepts, it adds 0100 and 0111 to its forwarding table and, since node 1001 is not a subscriber, it is removes 1001. As a result, nodes 0100 and 0111 will receive events directly through node 1101, and node 1001 does not consume resources on behalf of the multicast tree for topic 1100.

3.5 Reliability

Scribe relies on Pastry to achieve resilience to node faults and we intend to use TCP to disseminate events reliably in the presence of lossy links.

Periodically, each parent in the multicast tree sends a heartbeat message to its children. Most of these messages can be piggybacked on the events being disseminated, and on the heartbeat messages already exchanged between Pastry nodes. A child suspects that its parent is faulty when it fails to receive heartbeat messages. When this happens, it uses Pastry to route a JOIN message to a different parent.

All forwarders for a topic keep a small buffer containing the last events published to the topic that they received. The new parent uses this buffer to retransmit messages that the new child missed while detecting and recovering from the old parent's fault.

For example, in Figure 2, 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. The message reaches node 1111 who adds 1001 to its forwarding table and, since it is not a forwarder, sends a JOIN message towards the root. This causes node 1100 to add 1111 to its forwarding table. Node 1100 retransmits events to node 1111 that might have been missed. These are then retransmitted to node 1001.

Forwarding table entries are soft-state that is discarded unless it is periodically refreshed. Children periodically send messages to their parent in the multicast tree restating their interest in the topic. The parent discards their entry in the forwarding table if they fail to refresh it for a certain time period.

Scribe can also tolerate faults of multicast tree roots. The state associated with the rendez-vous point, which identifies the 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 will be in the leaf set of the root node. If the root fails, the node whose *nodeId* is now the closest to the *topicId* becomes the new root. Children of the old root will detect that it failed and will be rerouted to the new root when they run the procedure described above. The same will happen with publishers. When a publisher starts publishing to the new root, it uses its event buffer for the topic to retransmit events that might have been lost while recovering from the fault.

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

4 Preliminary experimental results

We are currently evaluating Scribe in a simulated network environment. We present some preliminary results of experiments in this section.

These experiments ran in a Pastry network with $N = 100,000$ nodes, $b = 4$, and $l = 16$. The topological distance metric used by Pastry to optimize route locality was determined as follows: each node was mapped to a point in a two dimensional Cartesian space, and the distance between two nodes was set to the distance between the points they mapped to. The nodes were uniformly distributed over the Cartesian space.

There was a single topic and we varied the number of subscribers to the topic from 1 to N. The subscribers were chosen randomly with uniform probability from the set of all Pastry nodes. When a node had more than c entries in its forwarding tables, it was considered overloaded for the purpose of controlling tree growth (as described in Section 3.4).

Figure 3 shows the average and maximum depth of the multicast tree generated when $c = 0$ and $c = 75$ for different numbers of subscribers. Setting $c = 0$ is equivalent to using the base mechanism for tree construction described in Section 3.2.

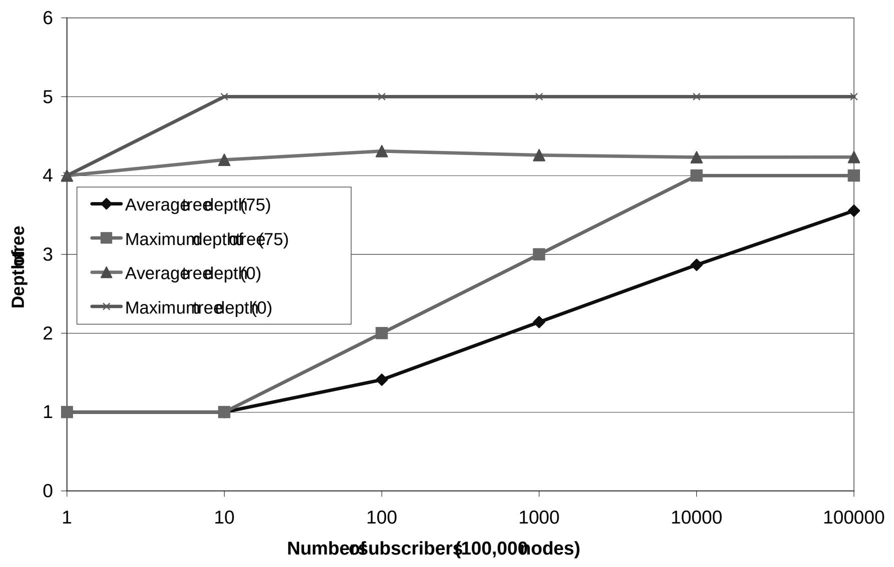

Figure 3: Maximum and average depth of multicast tree when $c = 0$ and $c = 75$ versus number of subscribers, with $b = 4$ and $N = 100,000$.

The results indicate that the multicast tree is well balanced. As expected, the results show that with $c = 0$ the height of the tree is largely independent of the number of subscribers. The height of the tree in this case is determined by the number of Pastry nodes and not by the number of subscribers to the topic. It is $log_{2^b}N = 4.15$ on average and the maximum height is 5, which is the ceiling of this value. With $c = 75$, the height of the tree is determined by the number of subscribers to the topic. It grows with $log_c n$ (where n is the number of subscribers to the topic). This reduces event dissemination latency, and the number of forwarders that are not subscribers.

5 Related work

Scribe implements a form of application-level multicast as proposed by several projects, for example, Narada [9] and Overcast [10]. This approach has the potential to overcome the naming, scalability, security, and deployment problems that have plagued IP multicast [11, 12] at the expense of decreased performance. application-level multicast duplicates packets on physical links and incurs higher latency but the experimental results presented here and in [10, 9] suggest that this performance penalty is acceptable.

Like Scribe, Overcast and Narada 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 bandwidth 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 follows a similar two level approach: it builds a multicast tree on top of a Pastry network and it performs end-to-end measurements to optimize this network according to some metric (e.g., IP hops, latency, or bandwidth). The main difference is that the Pastry network 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 [4] is the most similar to Scribe. Bayeux is built on top of a scalable peer-to-peer object location system called Tapestry [13] (which is similar to Pastry). Like Scribe, it supports multiple groups, and it builds a multicast tree per group on top of Tapestry but 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 (or router) along this route records the identity of the new member. Requests to leave the group from the topic are handled in a similar way.

Bayeux has two scalability problems when compared to Scribe. Firstly, it requires nodes to maintain more group membership information. The root keeps a list of all group members, the routers one hop away from the route keep a list containing on average $\frac{S}{b}$ members (where b is the base used in Tapestry routing), and so on. Secondly, Bayeux generates more traffic when handling group membership changes. In particular, 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 bounded by a constant independent of the number of group members. 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.

The height of multicast trees in Bayeux grows with the logarithm of the number of nodes in the Tapestry network. Whereas in Scribe, the height of the multicast trees grows with the logarithm of the number of group members. This can result in lower latency in Scribe for small group sizes, which are common in instant messaging/presence notification applications.

The mechanisms for fault resilience in Bayeux and Scribe are also very different. All the mechanisms for fault resilience proposed in Bayeux are sender-based whereas Scribe uses a receiver-based mechanism. In Bayeux, routers proactively duplicate outgoing packets across several paths or perform active probes to select alternative paths. Both these schemes have some disadvantages. The mechanisms that perform packet duplication consume additional bandwidth, and the mechanisms that select alternative paths require replication and transfer of group membership information across different paths. Scribe relies on heartbeats sent by parents to their children in the multicast tree to detect faults, and children use Pastry to reroute to a different parent when a fault is detected. Nodes keep a window of recently seen packets to allow local repair when a node attaches to a new parent. Additionally, Bayeux does not provide a mechanism to handle root failures whereas Scribe does.

Herald [3] is a companion project that shares the goals of Scribe.

There are several peer-to-peer object location and routing schemes that are similar to Pastry, for example, Tapestry [13], Chord [14], and Content Addressable Networks (CAN) [15]. We are currently evaluating whether Scribe can be built on top of these systems, and what properties can be achieved using each of them.

6 Conclusions and Future work

We have presented Scribe, a large-scale event notification system built on top of Pastry, a peer-to-peer object location and routing infrastructure. Scribe is designed to scale to billions of subscribers and topics, and supports multiple publishers per topic. Scribe can support many topics with a wide range of subscribers per topic, hence the same infrastructure can concurrently support applications such as instant messaging/presence notification, topical news distribution (e.g. sports results), and stock price broadcasting.

Scribe relies on the Pastry routing mechanism to perform subscription management and to configure multicast trees used to disseminate events to a set subscribers for a topic. The properties of Pastry ensures that the multicast trees are balanced, and tree management techniques are used to ensure that Scribe can rapidly adapt to changes in the number of subscribers to a topic. Resilience to failure in Scribe is based on the Pastry self-organization properties.

Preliminary simulations results are very promising and we are currently performing a detailed analysis of results obtained from the simulations. A security model for Scribe is currently under development, and we are adding a more sophisticated network topology model to the Pastry simulator to allow us to better understand the characteristics of Pastry and Scribe with respect to network locality.

Acknowledgements

We would like to thank Marc Shapiro and Andrew Herbert, and the other members of the Microsoft Research Cambridge Distributed Systems Group for their insightful and helpful comments.

References

[1] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, January 2001. http://www.research.microsoft.com/ antr/PAST/.

[2] 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)*, July 2001.

[3] Luis F. Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In *HotOS VIII*, May 2001.

[4] 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)*, June 2001.

[5] Peter Druschel and Antony Rowstron. PAST: A persistent and anonymous store. In *HotOS VIII*, May 2001.

[6] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, 2001. Accepted for SOSP01. http://www.research.microsoft.com/ antr/PAST/.

[7] Yogen K. Dalal and Robert Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

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

[9] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proc. of ACM Sigmetrics*, pages 1–12, June 2000.

[10] 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)*, pages 197–212, October 2000.

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

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

[13] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report 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. Technical Report TR-819, MIT, March 2001.

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