MIT Libraries | DSpace@MIT

MIT Open Access Articles

Census: Location-Aware Membership Management for Large-Scale Distributed Systems

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Cowling, J., et al. "Census: Location-Aware Membership Management for Large-Scale Distributed Systems." Proceedings of the 2009 USENIX Annual Technical Conference (San Diego: June 14-19, 2009).

As Published: http://www.usenix.org/event/usenix09/tech/

Publisher: USENIX Association

Persistent URL: http://hdl.handle.net/1721.1/61401

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0

Massachusetts Institute of Technology

Census: Location-Aware Membership Management for Large-Scale Distributed Systems

James Cowling* Dan R. K. Ports* Barbara Liskov*
Raluca Ada Popa* Abhijeet Gaikwad†
*MIT CSAIL** *École Centrale Paris†*

Abstract

We present *Census*, a platform for building large-scale distributed applications. Census provides a membership service and a multicast mechanism. The membership service provides every node with a consistent view of the system membership, which may be global or partitioned into location-based regions. Census distributes membership updates with low overhead, propagates changes promptly, and is resilient to both crashes and Byzantine failures. We believe that Census is the first system to provide a consistent membership abstraction at very large scale, greatly simplifying the design of applications built atop large deployments such as multi-site data centers.

Census builds on a novel multicast mechanism that is closely integrated with the membership service. It organizes nodes into a reliable overlay composed of multiple distribution trees, using network coordinates to minimize latency. Unlike other multicast systems, it avoids the cost of using distributed algorithms to construct and maintain trees. Instead, each node independently produces the same trees from the consistent membership view. Census uses this multicast mechanism to distribute membership updates, along with application-provided messages.

We evaluate the platform under simulation and on a real-world deployment on PlanetLab. We find that it imposes minimal bandwidth overhead, is able to react quickly to node failures and changes in the system membership, and can scale to substantial size.

1 Introduction

Today's increasingly large-scale distributed systems must adapt to dynamic membership, providing efficient and reliable service despite churn and failures. Such systems typically incorporate or rely on some sort of *membership service*, which provides the application with information about the nodes in the system. The current shift toward cloud computing and large multi-site data centers provides further motivation for a system designed to manage node membership at this large scale.

Many membership services exist, with varying semantics. Some, such as those based on virtual synchrony, provide strict semantics, ensuring that each node sees a consistent view of the system membership, but operate only at small scales [3, 14, 37]. Because maintaining consistency and global membership knowledge is often perceived as prohibitively expensive, many recently proposed systems provide weaker semantics. These systems provide greater scalability, but make no guarantees about members having consistent views [16, 17], and some provide only partial views of system membership [36, 23, 32].

We argue that it is both feasible and useful to maintain consistent views of system membership even in large-scale distributed systems. We present *Census*, a new platform for constructing such applications, consisting of a membership management system and a novel multicast mechanism. The membership management system follows the virtual synchrony paradigm: the system divides time into epochs, and all nodes in the same epoch have identical views of system membership. This globally consistent membership view represents a powerful abstraction that simplifies the design of applications built atop our platform. In addition to eliminating the need for applications to build their own system for detecting and tracking membership changes, globally consistent views can simplify application protocol design.

Census is designed to work at large scale, even with highly-dynamic membership, and to tolerate both crashes and Byzantine failures. It uses three main techniques to achieve these goals.

First, Census uses a locality-based hierarchical organization. Nodes are grouped into *regions* according to their network coordinates. Even in small systems, this hierarchical structure is used to reduce the costs of aggregating reports of membership changes. For systems so large that it is infeasible to maintain a global membership view, we provide a *partial knowledge* deployment option, where nodes know the full membership of their own region but only a few representative nodes from each other region.

Second, Census uses a novel multicast mechanism that is closely intertwined with the membership management service. The membership service relies on the multicast mechanism to distribute update notifications, and the multicast system constructs its distribution trees using node and location information from the membership service. The overlay topology, made up of redundant interior-disjoint trees, is similar to other systems [7, 40]. However, the trees are constructed in a very different way: each node independently carries out a deterministic tree construction algorithm when it receives a membership update. This eliminates the need for complex and potentially expensive distributed tree-building protocols, yet it produces efficient tree structures and allows the trees to change frequently to improve fault-tolerance. We also take advantage of global membership and location information to keep bandwidth overhead to a minimum by ensuring that each node receives no redundant data, while keeping latency low even if there are failures.

Finally, Census provides fault-tolerance. Unlike systems that require running an agreement protocol among all nodes in the system [30, 20], Census uses only a small subset of randomly-chosen nodes, greatly reducing the costs of membership management while still providing correctness with high probability. In most cases, we use lightweight quorum protocols to avoid the overhead of full state machine replication. We also discuss several new issues that arise in a Byzantine environment.

Census exposes the region abstraction and multicast mechanism to applications as additional services. Regions can be a useful organizing technique for applications. For example, a cooperative caching system might use regions to determine which nodes share their caches. The multicast system provides essential functionality for many applications that require membership knowledge, since a membership change may trigger a system reconfiguration (*e.g.* changing responsible nodes in a distributed storage system) that must be announced to all nodes.

Our evaluation of Census, under simulation and in a real-world deployment on PlanetLab, indicates that it imposes low bandwidth overhead per node (typically less than 1 KB/s even in very large systems), reacts quickly to node failures and system membership changes, and can scale to substantial size (over 100,000 nodes even in a high-churn environment).

The remainder of this paper is organized as follows. We define our assumptions in Section 2. Sections 3–5 describe Census's architecture, multicast mechanism, and fault-tolerance strategy in detail. Section 6 presents performance results based on both theoretical analysis and a deployment on PlanetLab. We sketch some ways applications can use the platform in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Model and Assumptions

Census is intended to be used in an asynchronous network like the Internet, in which messages may be corrupted, lost or reordered. We assume that messages sent repeatedly will eventually be delivered. We also assume nodes have loosely synchronized clock rates, such that they can approximately detect the receipt of messages at regular intervals. Loosely synchronized clock rates are easy to guarantee in practice, unlike loosely synchronized clocks.

Every node in our platform has an IP address, a unique random ID, and network coordinates. Tolerating Byzantine failures adds a few more requirements. Each node must have a public key, and its unique ID is assigned by taking a collision-resistant hash of the public key. Furthermore, we require admission control to prevent Sybil attacks [13], so each joining node must present a certificate signed by a trusted authority vouching for its identity.

All nodes have coordinates provided by a network coordinate system such as Vivaldi [11]. We describe the system in terms of a two-dimensional coordinate system plus *height*, analogous to the last-hop network delay. This follows the model used in Vivaldi, but our system could easily be modified to use a different coordinate space. We assume coordinates reflect network latency, but their accuracy affects only performance, not correctness.

Traditional network coordinate systems do not function well in a Byzantine environment since malicious nodes can influence the coordinates of honest nodes [38]. We have developed a protocol [39] that ensures that honest nodes' coordinates accurately reflect their locations by using a group of landmark nodes, some of which are permitted to be faulty. Another approach is described in [34]. These techniques do not provide any guarantees about the accuracy of a Byzantine node's coordinates, and we do not assume any such guarantees.

3 Platform Architecture

Our system moves through a sequence of *epochs*, numbered sequentially. Each epoch has a particular membership view. One of the members acts as the *leader*. Nodes inform the leader of membership events (nodes joining or leaving) and the leader collects this information for the duration of the epoch. The epoch length is a parameter whose setting depends on application needs and assumptions about the environment; for example, our experiments use 30s epochs. Users may opt to place the leader on a fixed node, or select a new leader each epoch based on the system membership and epoch number.

At the end of an epoch, the leader creates an *item* containing the membership changes and next epoch number, and multicasts this information as described in Section 4. The item can also include data provided by the application. The leader makes an upcall to the application code at its node to obtain this data and includes it in the item.

In addition, the system can perform additional multicasts within an epoch to propagate application data if desired.

When a node receives an item, it updates its view of the membership to reflect the latest joins and departures, then *enters* the next epoch. It can only process the item if it knows the system state of the previous epoch; nodes keep a few recent items in a log to enable nodes that are slightly behind to obtain missing information.

Our system ensures *consistency*: all nodes in the same epoch have identical views of the membership. The multicast mechanism delivers items quickly and reliably, so that nodes are likely to be in the same epoch at the same time. Messages include the epoch number at the point they were sent, to ensure they are routed and processed with respect to the correct membership view. Applications that require consistency also include the current epoch number in application messages, only processing messages when the sender and receiver agree on the epoch.

In this section, we describe how the system is organized. We begin in Section 3.1 with a simplified version with only a simple region. In Section 3.2, we introduce the multi-region structure, which improves scalability even though all nodes still know the membership of the entire system. Finally, in Section 3.3, we describe an optional extension to the system for extremely large or dynamic environments, where each node has full membership information only for its own region.

3.1 Single-Region Deployment

In a one-region system, all membership events are processed directly by the leader. The leader gathers notifications of node joins and departures throughout the epoch, then aggregates them into an item and multicasts the item to the rest of the system, starting the next epoch.

To join the system, a node sends a message identifying itself to the leader, providing its network coordinates and identity certificate (if tolerating Byzantine faults). The leader verifies the certificate, adds the node to a list of new joiners, and informs the new node of the epoch number and a few current members. The new node obtains the current membership from one of these nodes, reducing the load on the leader.

To remove a node, a departure request is sent to the leader identifying the node to be removed. A node can leave the system gracefully by requesting its own removal (in a Byzantine environment, this request must be signed). Nodes that do not fail gracefully are reported by other nodes; Section 5 describes this process. If the request is valid, the leader adds the node to a list of departers.

Nodes include their coordinates in the join request, ensuring that all nodes see a consistent view of each other's coordinates. Node locations can change over time, however, and coordinates should continue to reflect network proximity. Each node monitors its coordinates and reports changes, which are propagated in the next item. To avoid instability, nodes report only major location changes, using a threshold.

3.2 Multi-Region Deployment

Even at relatively high churn, with low available bandwidth and CPU resources, our analysis indicates that the single-region structure scales to beyond 10,000 nodes. As the system grows, however, the request load on the leader, and the overhead in computing distribution trees, increases. To accommodate larger systems, we provide a structure in which the membership is divided into regions based on proximity. Each region has a region ID and every node belongs to exactly one region. Even in relatively small systems, the multi-region structure is useful to reduce load on the leader, and to provide the application with locality-based regions.

In a multi-region system each region has its own local leader, which can change each epoch. This *region leader* collects joins and departures for nodes in its region. Towards the end of the epoch, it sends a *report* listing these membership events to the global leader, and the leader propagates this information in the next item. Any membership events that are received too late to be included in the report are forwarded to the next epoch's leader.

Even though all nodes still know the entire system membership, this architecture is more scalable. It offloads work from the global leader in two ways. First, the leader processes fewer messages, since it only handles aggregate information about joins and departures. Second, it can offload some cryptographic verification tasks, such as checking a joining node's certificate, to the region leaders. Moreover, using regions also reduces the CPU costs of our multicast algorithm, as Section 4 describes: nodes need not compute full distribution trees for other regions.

To join the system, a node contacts any member of the system (discovered out-of-band) and sends its coordinates. The member redirects the joining node to the leader of the region whose centroid is closest to the joining node. When a node's location changes, it may find that a different region is a better fit for it. When this happens, the node uses a *move* request to inform the new region's leader that it is leaving another region. This request is sent to the global leader and propagated in the next item.

3.2.1 Region Dynamics: Splitting and Merging

Initially, the system has only one region. New regions are formed by splitting existing regions when they grow too large. Similarly, regions that grow too small can be removed by merging them into other regions.

The global leader tracks the sizes of regions and when one of them exceeds a *split threshold*, it tells that region to split by including a *split request* in the next item. This request identifies the region that should split, and provides

the ID to used be for the newly formed region. When a region's size falls below a *merge threshold*, the leader selects a neighboring region for it to merge into, and inserts a *merge request* containing the two region IDs in the next item. The merge threshold is substantially smaller than the split threshold, to avoid oscillation.

Whenever a node processes an item containing a split or merge request, it carries out the split or merge computation. For a split, it computes the centroid and the widest axis, then splits the region into two parts. The part to the north or west retains the region ID, and the other part is assigned the new ID. For a merge, nodes from one region are added to the membership of the second region. As soon as this item is received, nodes consider themselves members of their new region.

3.3 Partial Knowledge

Even with the multi-region structure, scalability is ultimately limited by the need for every membership event in an epoch to be broadcast in the next item. The bandwidth costs of doing so are proportional to the number of nodes and the churn rate. For most systems, this cost is reasonable; our analysis in Section 6.1 shows, for example, that for systems with 100,000 nodes, even with a very short average node lifetime (30 minutes), average bandwidth overhead remains under 5 KB/s. However, for extremely large, dynamic, and/or bandwidth-constrained environments, the updates may grow too large.

For such systems, we provide a *partial knowledge* deployment option. Here, nodes have complete knowledge of the members of their own region, but know only *summary* information about other regions. We still provide consistency, however: in a particular epoch, every node in a region has the same view of the region, and every node in the system has the same view of all region summaries.

In this system, region leaders send the global leader only a summary of the membership changes in the last epoch, rather than the full report of all joins and departures. The summary identifies the region leader for the next epoch, provides the size and centroid of the region, and identifies some region members that act as its global *representatives*. The global leader includes this message in the next item, propagating it to all nodes in the system.

As we will discuss in Section 4, the representatives are used to build distribution trees. In addition, the representatives take care of propagating the full report, containing the joins and leaves, to nodes in their region; this way nodes in the region can compute the region membership. The region leader sends the report to the representatives at the same time it sends the summary to the global leader.

3.3.1 Splitting and Merging with Partial Knowledge

Splits and merges are operations involving the membership of multiple regions, so they are more complex in a partial knowledge deployment where nodes do not know the membership of other regions. We extend the protocols to transfer the necessary membership information between the regions involved.

When a region s is merged into neighboring region t, members of both regions need to learn the membership of the other. The representatives of s and t communicate to exchange this information, then propagate it on the tree for their region. The leader for t sends the global leader a summary for the combined region, and nodes in s consider themselves members of t as soon as they receive the item containing this summary.

A split cannot take place immediately because nodes outside the region need to know the summary information (centroid, representatives, etc.) for the newly-formed regions and cannot compute it themselves. When the region's leader receives a split request, it processes joins and leaves normally for the remainder of the epoch. At the end of the epoch, it carries out the split computation, and produces two summaries, one for each new region. These summaries are distributed in the next item, and the split takes effect in the next epoch.

4 Multicast

This section describes our multicast mechanism, which is used to disseminate membership updates and application data. The goals of the design are ensuring reliable delivery despite node failures and minimizing bandwidth overhead. Achieving low latency and a fair distribution of forwarding load are also design considerations.

Census's multicast mechanism uses multiple distribution trees, like many other multicast systems. However, our trees are constructed in a different way, taking advantage of the fact that membership information is available at all nodes. Trees are constructed on-the-fly using a deterministic algorithm on the system membership: as soon as a node receives the membership information for an epoch from one of its parents, it can construct the distribution tree, and thereby determine which nodes are its children. Because the algorithm is deterministic, each node computes exactly the same trees.

This use of global membership state stands in contrast to most multicast systems, which instead try to minimize the amount of state kept by each node. Having global membership information allows us to run what is essentially a centralized tree construction algorithm at each node, rather than a more complex distributed algorithm.

Our trees are constructed anew each epoch, ignoring their structure from the previous epoch. This may seem surprising, in light of the conventional wisdom that "stability of the routing trees is very important to achieve workable, reliable routing" [2]. However, this statement applies to multicast protocols that require executing costly protocols to change the tree structure, and may experi-

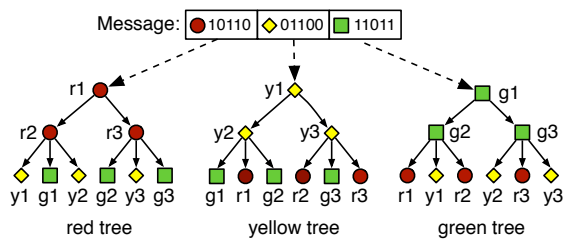


Figure 1: Distribution trees for a 3-color deployment. All nodes are members of each tree, but an internal node in only one. For example, the red nodes $r1$–$r3$ are interior nodes in the red tree, and leaves in the other two.

ence oscillatory or transitory behavior during significant adjustments. Our approach allows the trees to be recomputed with no costs other than those of maintaining the membership information. Changing the trees can also improve fault-tolerance and load distribution because different nodes are located at or near the root of the tree [24].

4.1 Multiple-Tree Overlay

A multicast overlay consisting of a single tree is insufficient to ensure reliability and fair load distribution: the small group of interior nodes bears all the forwarding load while the leaves bear none, and an interior node failure leads to loss of data at all its descendants. Instead, Census uses multiple trees, as in SplitStream [7], to spread forwarding load more evenly and enhance reliability.

Our overlay consists of a set of trees, typically between 4 and 16. The trees are *interior-node-disjoint*: each node is an interior node in at most one tree. We refer to each tree by a *color*, with each node in the system also assigned a color. The interior of the *red* tree is composed completely of red nodes, ensuring our disjointness constraint. The nodes of other colors form the leaves of the red tree, and the red nodes are leaf nodes in all other trees. Using an even distribution of node colors and a fan-out equal to the number of trees provides load-balancing: each node forwards only as many messages as it receives. Figure 1 illustrates the distribution trees in a 3-color system.

The membership update in each item must be sent in full along each tree, since it is used to construct the trees. The application data in an item, however, is split into a number of erasure-coded fragments, each of which is forwarded across a different tree. This provides redundancy while imposing minimal bandwidth overhead on the system. With n trees, we use *m of n* erasure coding, so that all nodes are able to reconstruct the original data even with failures in $n - m$ of the trees. This leads to a bandwidth overhead for application data of close to n/m, with overhead of n for the replicated membership update. However, as Section 4.4 describes, we are able to eliminate nearly all of this overhead under normal circumstances by suppressing redundant information.

We employ a simple *reconstruction* optimization that provides a substantial improvement in reliability. If a node does not receive the fragment it is supposed to forward, it can regenerate and forward the fragment once it has received m other fragments. This localizes a failure in a given tree to nodes where an ancestor in the current tree failed, and where each parent along the path to the root has experienced a failure in at least $n - m$ trees.

In the case of more than $n - m$ failures, a node may request missing fragments from nodes chosen randomly from its membership view. Section 6.2.1 shows such requests are unnecessary with up to 20% failed nodes.

4.2 Building Trees within a Region

In this section, we describe the algorithm Census uses to build trees. The algorithm must be a deterministic function of the system membership. We use a relatively straightforward algorithm that our experiments show is both computationally efficient and effective at offering low-latency paths, but more sophisticated algorithms are possible at the cost of additional complexity. We first describe how the tree is built in a one-region system; Section 4.3 extends this to multiple regions.

The first step is to color each node, *i.e.* assign it to a tree. This is accomplished by sorting nodes in the region by their ID, then coloring them round-robin, giving an even distribution of colors. Each node then computes all trees, but sends data only on its own tree.

The algorithm uses a hierarchical decomposition of the network coordinate space to exploit node locality. We describe how we build the red tree; other trees are built similarly. The tree is built by recursively subdividing the coordinate space into F sub-regions (where F is the fan-out, typically equal to the number of trees). This is performed by repeatedly splitting sub-regions through the centroid, across their widest axis. One red node from each sub-region is chosen to be a child of the root, and the process continues within each sub-region for the subtree rooted at each child, fewer than F red nodes remain in each sub-region. Figure 2 illustrates the hierarchical decomposition of regions into trees for a fan-out of 4.

Once all red nodes are in the tree, we add the nodes of other colors as leaves. We iterate over the other-colored nodes in ID order, adding them to the red node with free capacity that minimizes the distance to the root via that parent. Our implementation allows nodes to have a fan-out of up to $2F$ when joining leaf nodes to the internal trees, allowing us to better place nodes that are in a subregion where there is a concentration of a particular color.

As mentioned in Section 2, we use coordinates consisting of two dimensions plus a *height vector* [11]. Height is

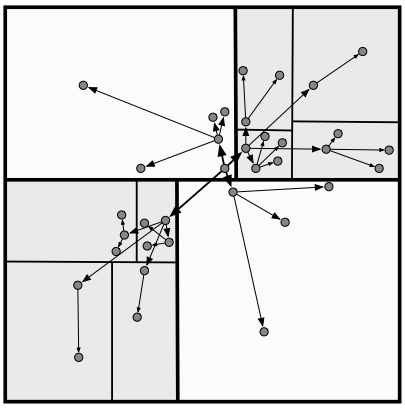


Figure 2: Hierarchical subdivision used to build interior tree with fan-out 4. Each region with more than 4 members is recursively split into smaller sub-regions.

ignored when splitting regions, since it does not reflect the geographic locality of nodes. However, it is used when computing the distance between two nodes, such as when picking the root of a sub-region.

4.3 Building Multi-Region Trees

In the multi-region system, we build an inter-region tree of each color. The nodes in the inter-region tree then serve as roots of the intra-region trees of that color.

The inter-region tree of a particular color is composed of one representative of that color from each region. The representatives are computed by the leader in a global knowledge system, and specified in the summaries in a partial knowledge system. The representatives can be thought of as forming their own "super" region, and we build the tree for that region using recursive subdivision as within a region. The only difference is that we use a smaller fan-out parameter for the inter-region tree, because each node in that tree also acts as a root for a tree within its region, and therefore has descendants in that region as well as descendants in the inter-region tree.

As mentioned in Section 3.3, representatives in the partial knowledge deployment are responsible for propagating the full report that underlies the summary to the members of the region. The extra information is added to the item by the root node of each tree for the region, and thus reaches all the nodes in the region.

4.4 Reducing Bandwidth Consumption

Using erasure coded fragments allows Census to provide high reliability with reasonably low overhead, but is not without bandwidth overhead altogether. In a configuration where 8 out of 16 fragments are required to reconstruct multicast data, each node sends twice as many fragments as strictly required in the non-failure case. Furthermore, membership updates are transmitted in full on *every* tree, giving even greater overhead.

We minimize bandwidth overhead by observing that redundant fragments and updates are necessary only if there is a failure. Instead of having each parent always send both a membership update and fragment, we designate only one parent per child to send the update and m parents per child to send the fragment. The other parents instead send a short "ping" message to indicate to their child that they have the update and fragment. A child who fails to receive the update or sufficient fragments after a timeout requests data from the parents who sent a ping.

This optimization has the potential to increase latency. Latency increases when there are failures, because a node must request additional fragments from its parents after a timeout. Even without failures, a node must wait to hear from the m parents that are designated to send a fragment, rather than just the first m parents that it hears from.

Fortunately, we are able to exploit membership knowledge to optimize latency. Each parent uses network coordinates to estimate, for each child, the total latency for a message to travel from the root of each tree to that child. Then, it sends a fragment only if it is on one of the m fastest paths to that child. The estimated latencies are also used to set the timeouts for requesting missing fragments. This optimization is possible because Census provides a globally consistent view of network coordinates. Section 6.2.3 shows that it eliminates nearly all redundant bandwidth overhead without greatly increasing latency.

5 Fault Tolerance

In this section we discuss how node and network failures are handled. We consider both crash failures and Byzantine failures, where nodes may behave arbitrarily.

5.1 Crash Failures

Census masks failures of the global leader using replication. A group of $2f_{GL} + 1$ nodes is designated as the *global leader group*, with one member acting as the global leader. Here, f_{GL} is not the maximum number of faulty nodes in the entire system, but rather the number of faulty nodes in the particular leader group; thus the group is relatively small. The members of the leader group use a consensus protocol [25, 21] to agree on the contents of each item: the leader forwards each item to the members of the global leader group, and waits for f_{GL} acknowledgments before distributing it on the multicast trees. The members of the global leader group monitor the leader and select a new one if it appears to have failed.

Tolerating crashes or unexpected node departures is relatively straightforward. Each parent monitors the liveness of its children. If the parent does not receive an acknowledgment after several attempts to forward an item,

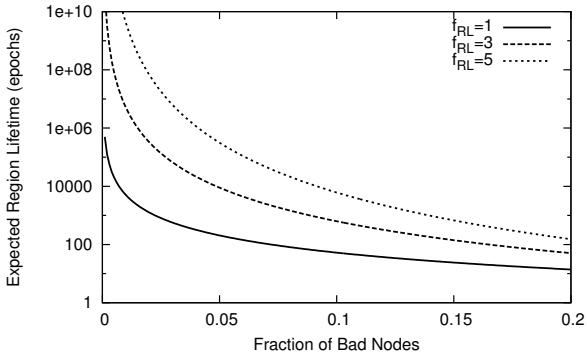


Figure 3: Expected time until a bad leader group (leader and f leader group members faulty) is chosen, in epochs

it reports the absence of the child to the region leader. The child will be removed from the system in the subsequent membership update; if it was only temporarily partitioned from the network, it can rejoin the region. In each epoch, the parent of a particular color is designated as monitor, to prevent multiple parents from reporting the same failure.

Region leaders are not replicated in the global knowledge system; a new region leader is chosen in each epoch, so a failed leader only causes updates to be delayed until it is replaced. In the partial knowledge system, however, we must also ensure that if a region leader sends a summary to the global leader, the corresponding report survives, even if the region leader fails; otherwise, the system would be in an inconsistent state. Census uses a region leader group of $2f_{RL} + 1$ nodes to solve this problem. The region leader sends the report to members of its leader group and waits for f_{RL} acknowledgments before sending the summary to the global leader. Thus, if the representatives receive an item containing a summary for their region, they are guaranteed to be able to retrieve the corresponding report from the leader group, even if the region leader failed, provided that no more than f_{RL} members of the leader group have failed.

Census can select leader groups at random from the system membership, using a deterministic function of the epoch number and region membership. If this approach is used, each summary in the partial knowledge system announces the region's next leader group, and the global leader group is chosen deterministically from the nodes in the region leader groups.

The size of the leader groups (*i.e.* the values of f_{RL} and f_{GL}) depends on the fraction of nodes expected to be failed *concurrently*, since faulty nodes are removed from the system. Figure 3 shows the expected number of leader groups that can be chosen before choosing a bad group. Because Census detects and removes crashed nodes within a couple of epochs, we can expect the fraction of failed nodes to remain small (*e.g.* under 1%), and therefore a small value for f will be sufficient even in a very long lived system.

Many applications have some infrastructure nodes that are expected to be very reliable. If so, using these as replicas in leader groups, especially for the global leader, can provide even better reliability. Using infrastructure nodes is particularly well-suited for applications that send multicast data, since they may benefit from having the global leader co-located with the source of multicast data.

5.2 Byzantine Failures

Our solution for Byzantine fault tolerance builds on the approaches used for crash failures, with the obvious extensions. For example, we require signed reports from $f_M + 1$ parents monitoring a failed node to remove it. If this exceeds the number of trees, the node's predecessors in the region ID space provide additional reports.

We use region leader groups in both the global knowledge and partial knowledge deployments. Since bad nodes may misbehave in ways that cannot be proven, and thus may not be removed from the system, all architectures such as ours must assume the fraction of Byzantine nodes is small. Figure 3 shows that this requirement is fairly constraining if we want the system to be long-lived. For example, with $f = 5$, we must assume no more than 3% faulty nodes to achieve an expected system lifetime of 10 years (with 30-second epochs). Therefore, it would be wise to choose leader groups from infrastructure nodes.

The size of a region leader group is still only $2f_{RL} + 1$, since the group does not run agreement. Instead, a region leader obtains signatures from $f_{RL}+1$ leader group members, including itself, before sending a summary or report to the global leader. These signatures certify that the group members have seen the updates underlying the report or summary. If the leader is faulty, it may not send the report or summary, but this absence will be rectified in subsequent epochs when a different leader is chosen.

To ensure that a faulty region leader cannot increase the probability that a region leader group contains more than f_{RL} faulty nodes, we choose leader group members based on their IDs, using a common technique from peer-to-peer systems [18, 36]: the first $2f_{RL}+1$ nodes with IDs greater than the hash of the epoch number (wrapping around if necessary) make up the leader group. A Byzantine region leader cannot invent fictitious joins or departures, because these are signed, and therefore it has no way to control node IDs. It might selectively process join and departure requests in an attempt to control the membership of the next leader group, but this technique is ineffective.

We increase the size of the global leader group to $3f_{GL} + 1$ nodes. The global group runs a Byzantine agreement protocol [9] once per epoch to agree on which summaries will be included in the next item. The next item includes $f_{GL} + 1$ signatures, ensuring that the pro-

tocol ran and the item is valid. The group members also monitor the leader and carry out a view change if it fails. We have developed a protocol that avoids running agreement but requires $2f_{GL} + 1$ signatures, but omit it due to lack of space. Because the failure of the global leader group can stop the entire system, and the tolerated failure level is lower, it is especially important to use trusted infrastructure nodes or other nodes known to be reliable.

5.2.1 Ganging-Up and Duplicates

Two new issues arise because of Census's multi-region structure. The first is a *ganging-up attack*, where a disproportionate number of Byzantine nodes is concentrated in one region. If so, the fraction of bad nodes in the region may be too high to ensure that region reports are accurate for any reasonable value of f_{RL}. This may occur if an attacker controls many nodes in a particular location, or if Byzantine nodes manipulate their network coordinates to join the region of their choice.

The second problem is that bad nodes might join many regions simultaneously, allowing a small fraction of bad nodes to amplify their population. Such *duplicates* are a problem only in the partial knowledge deployment, where nodes do not know the membership of other regions.

To handle these problems, we exploit the fact that faulty nodes cannot control their node ID. Instead of selecting a region's leader group from the region's membership, we select it from a subset of the global membership: we identify a portion of the ID space, and choose leaders from nodes with IDs in this partition. IDs are not under the control of the attacker, so it is safe to assume only a small fraction of nodes in this partition are corrupt, and thus at most f_{RL} failures will occur in a leader group.

Nodes in the leader partition are globally known, even in the partial knowledge system: when a node with an ID in the leader partition joins the system, it is reported to the global leader and announced globally in the next item. These nodes are members of their own region (based on their location), but may also be assigned to the leader group for a different region, and thus need to track that region membership state as well. Nodes in the leader partition are assigned to the leader groups of regions, using consistent hashing, in the same way values are assigned to nodes in distributed hash tables [36]. This keeps assignments relatively stable, minimizing the number of state transfers. When the leader group changes, new members need to fetch matching state from $f_{RL} + 1$ old members.

To detect duplicates in the partial knowledge system, we partition the ID space, and assign each partition to a region, again using consistent hashing. Each region tracks the membership of its assigned partition of the ID space. Every epoch, every region leader reports new joins and departures to the regions responsible for the monitoring the appropriate part of the ID space. These communications must contain $f_{RL} + 1$ signatures to prevent bad nodes from erroneously flagging others as duplicates. The leader of the monitoring region reports possible duplicates to the regions that contain them; they confirm that the node exists in both regions, then remove and blacklist the node.

5.3 Widespread Failures and Partitions

Since regions are based on proximity, a network partition or power failure may affect a substantial fraction of nodes within a particular region. Short disruptions are already handled by our protocol. When a node recovers and starts receiving items again, it will know from the epoch numbers that it missed some items, and can recover by requesting the items in question from other nodes.

Nodes can survive longer partitions by joining a different region. All nodes know the epoch duration, so they can use their local clock to estimate whether they have gone too many epochs without receiving an item. The global leader can eliminate an entire unresponsive region if it receives no summary or report for many epochs.

6 Evaluation

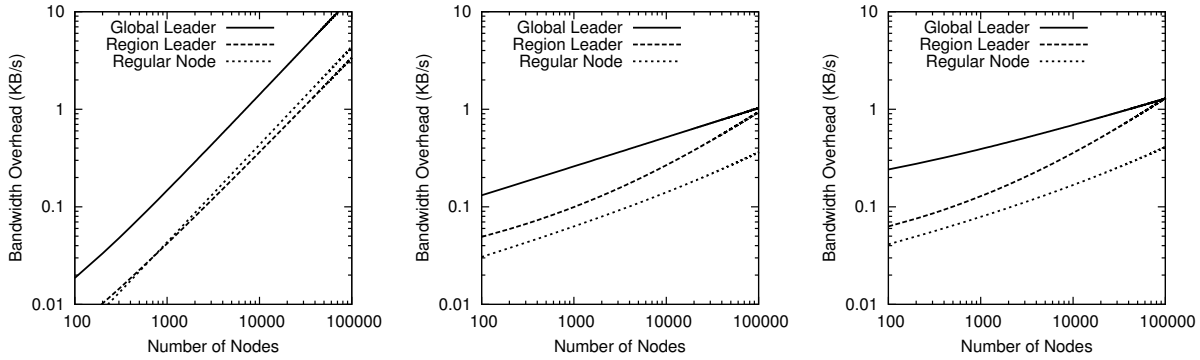
This section evaluates the performance of our system. We implemented a prototype of Census and deployed it on PlanetLab to evaluate its behavior under real-world conditions. Because PlanetLab is much smaller than the large-scale environments our system was designed for, we also examine the reliability, latency, and bandwidth overhead of Census using simulation and theoretical analysis.

6.1 Analytic Results

Figure 4 presents a theoretical analysis of bandwidth overhead per node for a multi-region system supporting both fail-stop and Byzantine failures. The analysis used 8 trees and an epoch interval of 30 seconds. Our figures take all protocol messages into consideration, including UDP overhead, though Figure 4(c) does not include support for preventing ganging-up or for duplicate detection.

Bandwidth utilization in Census is a function of both system membership and churn. These results represent a median node lifetime of 30 minutes, considered a high level of churn with respect to measurement studies of the Gnutella peer-to-peer network [33]. This serves as a "worst-case" figure; in practice, we expect most Census deployments (*e.g.* those in data center environments) would see far lower churn.

The results show that overhead is low for all configurations except when operating with global knowledge on very large system sizes (note the logarithmic axes). Here the global leader needs to process all membership updates, as well as forward these updates to all 8 distribution trees. The other nodes in the system have lower overhead because they forward updates on only one tree. The overhead at the global leader is an order of magni-



(a) Crash Fault Tolerance, Global Knowledge (b) Crash Fault Tolerance, Partial Knowledge (c) Byz. Fault Tolerance, Partial Knowledge

Figure 4: Mean bandwidth overhead with high churn. 30 minute session lengths, 30 second epochs, 8 trees and $f = 3$.

tude lower in the partial knowledge case, where it only receives and distributes the compact region summaries.

In the partial knowledge cases (Figures 4(b) and 4(c)) the region leader incurs more overhead than the regular nodes, primarily due to forwarding each report to the leader group and representatives before sending a summary to the global leader. Supporting Byzantine fault tolerance imposes little additional overhead for the region leaders and global leader, because the cost of the additional signatures and agreement messages are dominated by the costs of forwarding summaries and reports.

These results are sensitive to region size, particularly in large deployments, as this affects the trade-off between load on the region leaders and on the global leader. For the purpose of our analysis we set the number of regions to $\sqrt[3]{nodes}$, mimicking the proportions of a large-scale deployment of 100 regions each containing 10,000 nodes.

6.2 Simulation Results

We used a discrete-event simulator written for this project to evaluate reliability, latency, and the effectiveness of our selective fragment transmission optimization. The simulator models propagation delay between hosts, but does not model queuing delay or network loss; loss due to bad links is represented by overlay node failures.

Two topologies were used in our simulations: the King topology, and a random synthetic network topology. The King topology is derived from the latency matrix of 1740 Internet DNS servers used in the evaluation of Vivaldi [11], collected using the King method [15]. This topology represents a typical distribution of nodes in the Internet, including geographical clustering. We also generate a number of synthetic topologies, with nodes uniformly distributed within the coordinate space. These random topologies allow us to examine the performance of the algorithm when nodes are not tightly clustered into regions, and to freely experiment with network sizes without affecting the distribution of nodes.

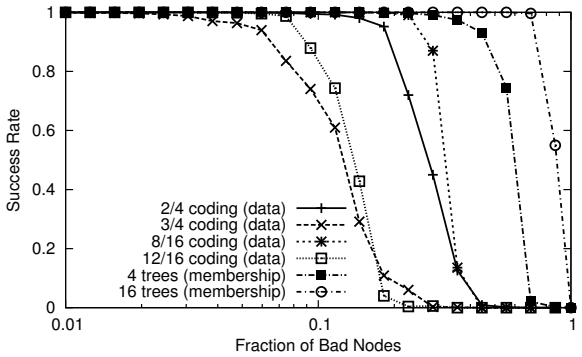


Figure 5: Fraction of nodes that receive tree data in the presence of faulty nodes, with 10 regions of 1,000 nodes.

While our simulator measures network delays using latencies, our algorithms operate solely in the coordinate domain. We generated coordinates from the King data using a centralized version of the Vivaldi algorithm [11]. Coordinates consist of two dimensions and a height vector, as was found to effectively model latencies in Vivaldi. These coordinates do not perfectly model actual network delays, as discussed in Section 6.2.2.

6.2.1 Fault Tolerance

Figure 5 examines the reliability of our distribution trees for disseminating membership updates and application data under simulation. In each experiment we operate with 10 regions of 1,000 nodes each; single-region deployments see equivalent results. The reliability is a function only of the tree fan-out and our disjointness constraint, and does not depend on network topology.

The leftmost four lines show the fraction of nodes that are able to reconstruct application data under various erasure coding configurations. Census achieves very high

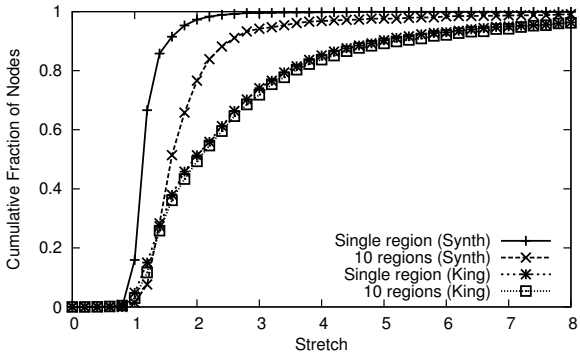


Figure 6: CDF of multicast transmission stretch in King and 10,000 node synthetic topologies. 4/8 erasure coding.

success rates with up to 30% failed nodes in the 16-tree configuration, with 2× redundancy (8/16 coding). This high failure rate is unlikely, since our membership management protocol removes failed nodes promptly. The primary factor influencing reliability is the redundancy rate. Increasing the number of trees also improves reliability, even with the same level of redundancy: using 16 trees instead of 4 tolerates nearly 10% more failures. Additional trees improve reliability by reducing the probability of $n - m$ parents failing, but comes with the cost of more messages and more representatives to maintain.

The rightmost two lines show the fraction of nodes that receive membership update information, which is sent in full on each tree. In a 16-tree deployment, we find that every non-faulty node receives membership information on at least one tree, even if as many as 70% of the region members have failed. Even on a 4-tree deployment, all nodes receive membership information in the presence of upwards of 20% failed nodes.

The high reliability exhibited in Figure 5 is partially due to our reconstruction optimization, discussed in Section 4.1. An 8/16 deployment using reconstruction allows all non-faulty nodes to receive application data with as many as 22% failed nodes, but tolerates only 7.5% failures without reconstruction. Reconstruction mitigates the effects of a failure by allowing a tree to heal below a faulty node, using fragments from other trees.

6.2.2 Latency

Census's multicast mechanism must not impose excessive communication delay. We evaluate this delay in terms of *stretch*, defined as the total time taken for a node to receive enough fragments to reconstruct the data, divided by the unicast latency between the server and the node.

Figure 6 shows stretch on both the King and synthetic topologies, assuming no failures and using 8 trees; results for 16 trees are similar. The figure shows that stretch is close to 1 on the synthetic topology, indicating that our tree-building mechanism produces highly efficient trees. Stretch is still low on the King topology, at an average of 2, but higher than in the synthetic topology. This reflects the fact that the coordinates generated for the King topology are not perfect predictors of network latency, while the network coordinates in the synthetic topology are assumed perfect. The small fraction of nodes with stretch below 1 are instances where node latencies violate the triangle inequality, and the multicast overlay achieves lower latency than unicast transmission.

Stretch is slightly higher in the multi-region deployment with the synthetic topology because the inter-region tree must be constructed only of representatives. In the synthetic topology, which has no geographic locality, stretch increases because the representatives may not be optimally placed within each region. However, this effect is negligible using the King topology, because nodes within a region are clustered together and therefore the choice of representatives has little effect on latency.

Our stretch compares favorably with existing multicast systems, such as SplitStream [7], which also has a stretch of approximately 2. In all cases, transmission delay overhead is very low compared to typical epoch times.

6.2.3 Selective Fragment Transmission

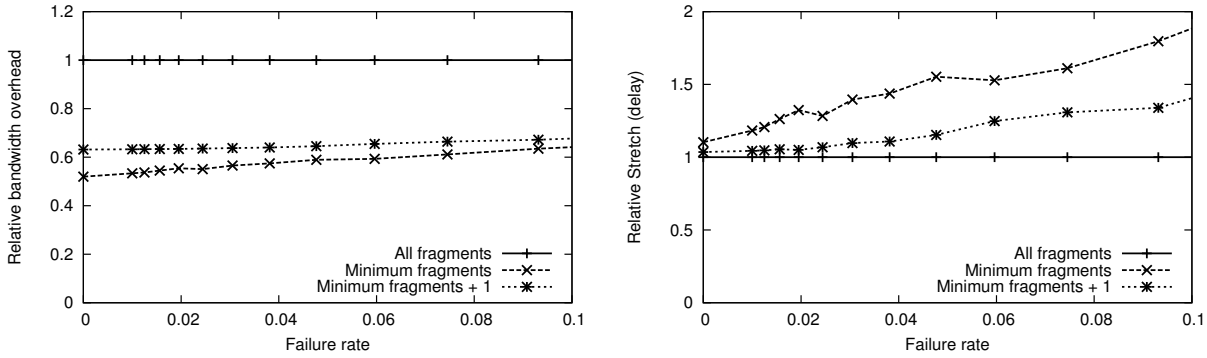
Figure 7(a) illustrates the bandwidth savings of our optimization to avoid sending redundant fragments (described in Section 4.4), using 2× redundancy and 8 trees on the King topology. In the figure, bandwidth is measured relative to the baseline approach of sending all fragments. We see a 50% reduction in bandwidth with this optimization; overhead increases slightly at higher failure rates, as children request additional data after timeouts.

Figure 7(b) shows this optimization's impact on latency. It adds negligible additional stretch at low failure rates, because Census chooses which fragments to distribute based on accurate predictions of tree latencies. At higher failure rates, latency increases as clients are forced to request additional fragments from other parents, introducing delays throughout the distribution trees.

The figures indicate that the optimization is very effective in the expected deployments where the failure rate is low. If the expected failure rate is higher, sending one extra fragment reduces latency with little impact on bandwidth utilization.

6.3 PlanetLab Experiments

We verify our results using an implementation of our system, deployed on 614 nodes on PlanetLab. While this does not approach the large system sizes for which our protocol was designed, the experiment provides a proof of concept for a real widespread deployment, and allows



(a) Relative Bandwidth

(b) Relative Stretch

Figure 7: Performance impact of avoiding redundant fragments in multicast trees. (4 of 8 coding)

us to observe the system under realistic conditions such as non-uniform node failures.

Our implementation supports multiple regions, with dynamic splits and joins, but we found that a single region was sufficient for the number of nodes in our PlanetLab deployment, and more representative of region sizes that would be seen in larger deployments. The implementation currently supports fail-stop failures, moving the leader each epoch, but does not tolerate Byzantine failures.

We configured the system to use 6 distribution trees, with an epoch time of 30 seconds. In addition to membership information, we distributed a 1 KB application message each epoch using 3-of-6 erasure coding, to test the reliability and overhead of this part of our system.

We ran Census for 140 epochs of 30 seconds each. As indicated in Figure 8(a), during the experiment, we failed 10% of the nodes simultaneously, then restarted them; we then did the same with 25% of the nodes. The graph shows the number of nodes reported in our system's membership view. Census reacts quickly to the sudden membership changes; the slight delay reflects the time needed for parents to decide that their children are faulty.

Figure 8(b) shows the average total bandwidth usage (both upstream and downstream) experienced by nodes in our system. Each node uses about 0.1 KB/s at steady-state, much of which is due to the size of the multicast data; the shaded region of the graph represents the theoretical minimum cost of disseminating a 1 KB message each epoch. Bandwidth usage increases for a brief time after our sudden membership changes, peaking at 0.9 KB/s immediately after 25% of the nodes rejoin at once. Node rejoins are more costly than node failures, because more information needs to be announced globally for a newly-joined node and the new node needs to obtain the system membership. We have also run the system for much longer periods, with similar steady-state bandwidth usage.

7 Applications

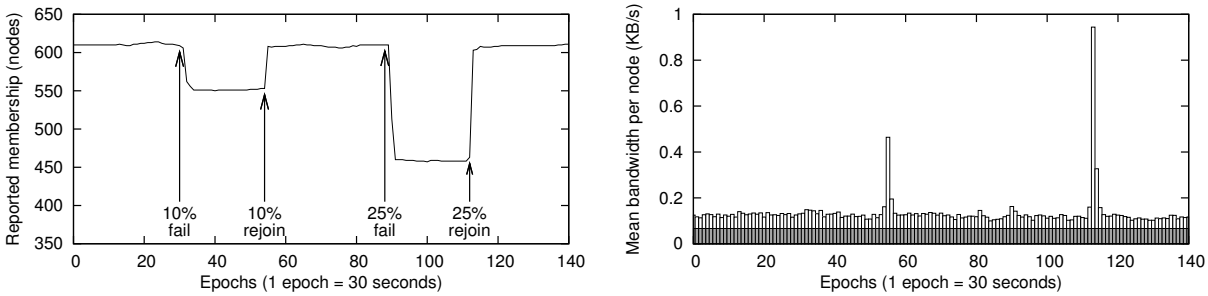
Knowledge of system membership is a powerful tool that can simplify the design of many distributed systems. An obvious application of Census is to support administration of large multi-site data centers, where Byzantine failures are rare (but do occur), and locality is captured by our region abstraction. Census is also useful as an infrastructure for developing applications in such large distributed systems. In this section, we describe a few representative systems whose design can be simplified with Census.

7.1 One-Hop Distributed Hash Tables

A distributed hash table is a storage system that uses a distributed algorithm, usually based on consistent hashing [18], to map item keys to the nodes responsible for their storage. This abstraction has proven useful for organizing systems at scales ranging from thousands of nodes in data centers [12] to millions of nodes in peer-to-peer networks [23]. The complexity in such systems lies primarily in maintaining membership information to route requests to the correct node, a straightforward task with the full membership information that Census provides.

Most DHTs do not maintain full membership knowledge at each host, so multiple (*e.g.* $O(\log N)$) routing steps are required to locate the node responsible for an object. Full global knowledge allows a message to be routed in one step. In larger systems that require partial knowledge, messages can be routed in two steps. The key now identifies both the responsible region and a node within that region. A node first routes a message to any member of the correct region, which then forwards it to the responsible node, much like the two-hop routing scheme of Gupta et al. [16]. Although our membership management overhead does not scale asymptotically as well as many DHT designs, our analysis in Section 6.1 shows that the costs are reasonable in most deployments.

From a fault-tolerance perspective, Census's member-



(a) System Membership

(b) Bandwidth Usage

Figure 8: Results of a 614-node, 70-minute PlanetLab deployment with 10% and 25% correlated failures.

ship views provide several advantages. Single-hop routing eliminates the possibility of a malicious intermediate node redirecting a request [35]. The fault-tolerance of our protocol prevents Eclipse attacks, where malicious nodes influence an honest node's routing table [6]. Applications that use replication techniques to ensure consistency across replicas benefit from Census's consistent membership views, since all nodes in the system agree on the identity of the replicas for each epoch.

Finally, our use of full membership information can enable more sophisticated placement algorithms than the standard consistent hashing approach. Any deterministic function of the membership view, including location information, suffices. For example, we might choose one replica for a data item based on its key, and the others based on location: either nearby nodes (for improved performance) or distant ones (for failure-independence).

7.2 Application-Layer Multicast

Census allows applications to disseminate information on multicast trees by piggybacking it on items. The application can do this as needed: occasionally, on every item, or more frequently. An example of where more frequent multicast is needed is to broadcast video. For high-bandwidth multicast like video streaming, the costs of maintaining membership become less significant.

Compared to other scalable multicast systems, Census's multicast trees can provide higher reliability, using optimizations like reconstruction and selective fragment transmission, and can tolerate Byzantine behavior. The availability of consistent membership views keeps the multicast protocol relatively simple, while still providing strong performance and reliability guarantees.

Census can also be used to construct a publish-subscribe system where only certain nodes are interested in receiving each message. One node is designated as responsible for each interest group, and other nodes contact it to publish or subscribe. When this node receives a message to distribute, it constructs multicast trees over just the subscribers, using them to disseminate both the message and changes in subscriber membership. This multicast is independent of the one we use to distribute membership information, but the trees can be constructed using the same algorithm.

7.3 Cooperative Caching

We are currently developing a wide-scale storage application where a small set of nodes act as servers, storing the definitive copy of system data. The other nodes in the system are clients. To perform operations, they fetch pages of data from the server into local caches and execute operations locally; they write back modified pages to the server when the computation is finished.

To reduce load on the storage servers, clients share their caches, fetching missing pages from nearby clients. A partial knowledge Census deployment makes it easy for clients to identify other nearby clients. We are investigating two approaches to finding pages. In one, nodes announce pages they are caching on the multicast tree within a region, so each node in the region always knows which pages are cached nearby. The other uses an approach similar to peer-to-peer indexing systems (*e.g.* [10]): we use consistent hashing [18] to designate for each page one node per region that keeps track of which region members have that page cached. Members register with this node once they have fetched a page, and check with it when they are looking for a page.

Cached information inevitably becomes stale, rendering it useless for computations that require consistency. To keep caches up to date, storage servers in this system use Census's multicast service to distribute an *invalidation stream*. This consists of periodic notices listing the set of recently modified pages; when a node receives such a notice, it discards the invalid pages from its cache.

8 Related Work

There is a long history of research in group communication systems, which provide a multicast abstraction along

with a membership management service [14, 37, 19, 2, 26, 29]. Many of these systems provide support for group communication while maintaining *virtual synchrony* [3], a model similar to our use of epochs to establish consistent views of system information. Such systems are typically not designed to scale to large system populations, and often require dedicated membership servers, which do not fit well with our decentralized model.

Spread [2] and ISIS [4] use an abstraction of many lightweight membership groups mapping onto a smaller set of core groups, allowing the system to scale to large numbers of multicast groups, but not large membership sizes. We take a different approach in using regions to group physical nodes, and scale to large system memberships, without providing a multiple-group abstraction. Quicksilver [26] aims to scale in both the number of groups and the number of nodes, but does not exploit our physical hierarchy to minimize latency and communication overhead in large system deployments.

Prior group communication systems have also aimed to tolerate Byzantine faults, in protocols such as Rampart [30] and SecureRing [20]. Updating the membership view in these systems requires executing a three-phase commit protocol across all nodes, which is impractical with more than a few nodes. By restricting our protocol to require Byzantine agreement across a small subset of nodes, we achieve greater scalability. Rodrigues proposed a membership service using similar techniques [31], but it does not provide locality-based regions or partial knowledge, and assumes an existing multicast mechanism.

Many large-scale distributed systems employ ad-hoc solutions to track dynamic membership. A common approach is to use a centralized server to maintain the list of active nodes, as in Google's Chubby lock service [5]. Such an approach requires all clients to communicate directly with a replicated server, which may be undesirable from a scalability perspective. An alternative, decentralized approach seen in Amazon's Dynamo system [12] is to track system membership using a gossip protocol. This approach provides only eventual consistency, which is inadequate for many applications, and can be slow to converge. These systems also typically do not tolerate Byzantine faults, as evidenced by a highly-publicized outage of Amazon's S3 service [1]

Distributed lookup services, such as Chord [36] and Pastry [32], provide a scalable approach to distributed systems management, but none of these systems provides a consistent view of membership. They are also vulnerable to attacks in which Byzantine nodes cause requests to be misdirected; solving this problem involves trading-off performance for probabilistic guarantees of correctness [6].

Fireflies [17] provides each node with a view of system membership, using gossip techniques that tolerate Byzantine failures. However, it does not guarantee a *consistent* global membership view, instead giving a probabilistic agreement. Also, our location-aware distribution trees offer faster message delivery and reaction to changes.

Our system's multicast protocol for disseminating membership updates builds on the multitude of recent application-level multicast systems. Most (but not all) of these systems organize the overlay as a tree to minimize latency; the tree can be constructed either by a centralized authority [28, 27] or by a distributed algorithm [8, 7, 22]. We use a different approach: relying on the availability of global, consistent membership views, we run what is essentially a centralized tree-building algorithm independently at each node, producing identical, optimized trees without a central authority.

SplitStream [7] distributes erasure-coded fragments across multiple interior-node-disjoint multicast trees in order to improve resilience and better distribute load among the nodes. Our overlay has the same topology, but it is constructed in a different manner. We also employ new optimizations, such as selective fragment distribution and fragment reconstruction, which provide higher levels of reliability with lower bandwidth overhead.

9 Conclusions

Scalable Internet services are often built as distributed systems that reconfigure themselves automatically as new nodes become available and old nodes fail. Such systems must track their membership. Although many membership services exist, all current systems are either impractical at large scale, or provide weak semantics that complicate application design.

Census is a membership management platform for building distributed applications that provides both strong semantics and scalability. It provides consistent membership views, following the virtual synchrony model, simplifying the design of applications that use it. The protocol scales to large system sizes by automatically partitioning nodes into proximity-based regions, which constrains the volume of membership information a node needs to track. Using lightweight quorum protocols and agreement across small groups of nodes, Census can maintain scalability while tolerating crash failures and a small fraction of Byzantine-faulty nodes.

Census distributes membership updates and application data using a unconventional multicast protocol that takes advantage of the availability of membership data. The key idea is that the distribution tree structure is determined entirely by the system membership state, allowing nodes to independently compute identical trees. This approach allows the tree to be reconstructed without any overhead other than that required for tracking membership. As our experiments show, using network coordinates produces trees that distribute data with low latency, and the multiple-tree overlay structure provides reliable data dissemination

even in the presence of large correlated failures.

We deployed Census on PlanetLab and hope to make the deployment available as a public service. We are currently using it as the platform for a large-scale storage system we are designing, and expect that it will be similarly useful for other reconfigurable distributed systems.

Acknowledgments

We thank our shepherd Marvin Theimer, Austin Clements, and the anonymous reviewers for their valuable feedback. This research was supported by NSF ITR grant CNS-0428107.

References

[1] Amazon Web Services. Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[2] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, 1998.

[3] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. SOSP '87*, Nov. 1987.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *Transactions on Computer Systems*, volume 9, Aug. 1991.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI '06*, Seattle, WA, Nov. 2006.

[6] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. OSDI '02*, Boston, MA, Nov. 2002.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proc. SOSP '03*, Bolton Landing, NY, 2003.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, Oct. 2002.

[9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[10] A. T. Clements, D. R. K. Ports, and D. R. Karger. Arpeggio: Metadata searching and content sharing with Chord. In *Proc. IPTPS '05*, Ithaca, NY, Feb. 2005.

[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. ACM SIGCOMM 2004*, Portland, OR, Aug. 2004.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP '07*, Stevenson, WA, Oct. 2007.

[13] J. R. Douceur. The Sybil attack. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.

[14] B. B. Glade, K. P. Birman, R. C. B. Cooper, and R. van Renesse. Lightweight process groups in the ISIS system. *Distributed Systems Engineering*, 1:29–36, 1993.

[15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proc. IMW '02*, Marseille, France, Nov. 2002.

[16] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proc. NSDI '04*, San Francisco, CA, Mar. 2004.

[17] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. EuroSys '06*, Leuven, Belgium, Apr. 2006.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC '97*, El Paso, TX, May 1998.

[19] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20:191–238, 2002.

[20] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. HICSS '98*, Hawaii, Jan. 1998.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[22] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. MON: On-demand overlays for distributed system management. In *Proc. WORLDS '05*, San Francisco, CA, Dec. 2005.

[23] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.

[24] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proc. P2PEcon '04*, Cambridge, MA, June 2004.

[25] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. PODC '88*, 1988.

[26] K. Ostrowski, K. Birman, and D. Dolev. QuickSilver scalable multicast. In *Proc. NCA '08*, Cambridge, MA, July 2008.

[27] V. N. Padmanabhan, H. J. Wang, and P. A. Chou. Resilient peer-to-peer streaming. In *Proc. ICNP '03*, Atlanta, GA, Nov. 2003.

[28] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proc. USITS '01*, San Francisco, CA, Mar. 2001.

[29] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynck. The Delta-4 approach to dependability in open distributed computing systems. In *Proc. FTCS '88*, June 1988.

[30] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan. 1996.

[31] R. Rodrigues. *Robust Services in Dynamic Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2005.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, Heidelberg, Nov. 2001.

[33] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN '02*, San Jose, CA, June 2002.

[34] M. Sherr, B. T. Loo, and M. Blaze. Veracity: A fully decentralized service for securing network coordinate systems. In *Proc. IPTPS '08*, Feb. 2008.

[35] E. Sit and R. Morris. Security concerns for peer-to-peer distributed hash tables. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.

[36] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Networking*, 11(1):149–160, Feb. 2003.

[37] R. van Renesse, K. P. Birman, and S. M. Eis. Horus: A flexible group communication system. *Communications of the ACM*, 39:76–83, 1996.

[38] D. Zage and C. Nita-Rotaru. On the accuracy of decentralized network coordinate systems in adversarial networks. In *Proc. CCS '07*, Alexandria, VA, Oct. 2007.

[39] Y. Zhou. Computing network coordinates in the presence of Byzantine faults. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2008. Available as technical report MIT-CSAIL-TR-2009-015.

[40] Y. Zhu, B. Li, and J. Guo. Multicast with network coding in application-layer overlay networks. *IEEE Journal on Selected Areas in Communications*, 22(1), Jan. 2004.