Computer Networks 53 (2009) 2304–2320

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems ☆

Jay A. Patel [a,*], Étienne Rivière [b], Indranil Gupta [a], Anne-Marie Kermarrec [c]

[a] University of Illinois at Urbana-Champaign, Department of Computer Science, 201 N Goodwin Ave, Urbana, IL 61801, United States
[b] NTNU, Department of Computer and Information Science, Sem Sælandsvei 7–9, Trondheim, NO-7491, Norway
[c] INRIA Rennes – Bretagne Atlantique, Campus Universitaire de Beaulieu, 35042 Rennes, France

ARTICLE INFO

Article history:
Available online 8 April 2009

Keywords:
Publish-subscribe
Application-level multicast
Gossip-based overlay construction
Self-organization

ABSTRACT

In this paper, we present the design, implementation and evaluation of Rappel, a peer-to-peer feed-based publish-subscribe service. By using a combination of probabilistic and gossip-like techniques and mechanisms, Rappel provides noiselessness, i.e., updates from any feed are received and relayed only by nodes that are subscribers of that feed. This leads to a fair system: the overhead at each subscriber node scales with the number and nature of its subscriptions. Moreover, Rappel incurs small publisher and client overhead, and its clients receive updates quickly and with low IP stretch. To achieve these goals, Rappel exploits "interest locality" characteristics observed amongst real multi-user multi-feed populations. This is combined with systems design decisions that enable nodes to find other subscribers, and maintain efficient network locality-aware dissemination trees. We evaluate Rappel via both trace-driven simulations and a PlanetLab deployment. The experimental results from the PlanetLab deployment show that Rappel subscribers receive updates within hundreds of milliseconds after posting. Further, results from the trace-driven simulator match our PlanetLab deployment, thus allowing us to extrapolate Rappel's performance at larger scales.

1. Introduction and background

The popularity of syndicated feeds such as RSS and ATOM have enabled topic-based publish-subscribe for web logs, wikis, news sites, and online social networks, e.g., LiveJournal [26], Twitter [43], etc. In such systems, a topic of interest is called a *feed*, e.g., an RSS news source. A feed has a single *publisher*, which is a computer host that is the source of all updates for that feed. There is a set of *subscriber* nodes (hosts) associated with each feed. These subscriber nodes desire to receive all the feed's updates, including those generated when the subscriber was offline. Each node corresponds to a user and may subscribe to multiple feeds.

In this paper, we propose Rappel–Rapid, Adaptive, Push-Pull of Electronic Feeds. Rappel is a peer-to-peer publish-subscribe system that aims to provide: (1) low overhead at publisher and subscriber nodes, (2) fast reception of updates at subscribing nodes (with low stretch compared to direct reception from the publisher), and (3) noiseless update dissemination. It is worth pointing out that this paper is not focused on security goals but rather on meeting the stated performance related goals.

A few notes are due on the third goal above. We say that a node suffers from noise when it has to receive and relay updates for feeds that it is not subscribed to. Noiselessness is one of our goals because of its potential to provide a desirable property: fairness. Fairness means the overhead at each node will grow only as a function of the number and nature of subscriptions at that node, and not due to overall system behavior.

☆ This work is previously unpublished.

* Corresponding author. Tel.: +1 217 265 5517.
E-mail addresses: jaypatel@cs.uiuc.edu (J.A. Patel), etriviere@gmail.com (É. Rivière), indy@cs.uiuc.edu (I. Gupta), anne-marie.kermarrec@inria.fr (A.-M. Kermarrec).

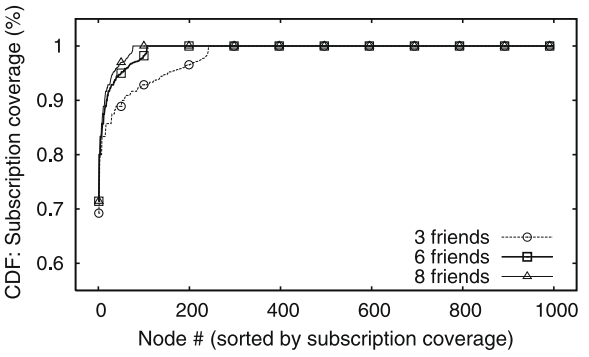
1389-1286/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2009.03.018

Several systems have been proposed to improve the fault-tolerance, scalability and performance of publish-subscribe systems, e.g., [6–9,13,22,24,34,36,38,47]. Yet, we find that none of these simultaneously address all our goals. In brief, centralized techniques impose high load on publishers, a bulk of decentralized techniques use non-subscribers for relaying updates (thus incurring noise), and IP multicast based subscriber-only dissemination trees impose membership management load at the publisher. We elaborate further on related work in Section 7.

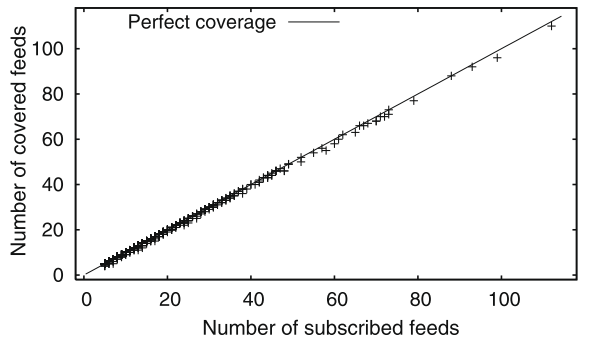
Rappel's approach is to maintain a single collaborative control-plane overlay among all nodes, and use this to build and maintain data dissemination trees for each feed. Within the control-plane overlay, a Rappel node continuously aims to move closer towards its "interest locality".

Interest locality is related to the notion of coverage - two nodes that are subscribed to the same feed are said to cover each other w.r.t. that feed. A system is said to show interest locality if for each node x, a small set of "friend" nodes suffice to cover all of x's subscribed-to feeds. Interest locality arises naturally from the clustering of human interests [14,18,39]. Supposing that each node can greedily select the k best friend nodes to maximize its own coverage, interest locality can be observed amongst the users of LiveJournal [26] – a popular multi-feed subscription platform. Fig. 1 illustrates interest locality for 1000 randomly selected nodes. The first plot shows the CDF of subscription coverage across nodes at various values of k. Even with a low number of $k = 6$ best friends, complete feed coverage is exhibited at 95% of the nodes. Further, the second plot shows that subscription coverage does not degrade with increasing number of subscriptions.

Interest locality has been successfully leveraged in the context of publish and subscribe middleware for reducing nodes' degrees [11] and for creating semantic communities for content-based filtering [10]. Rappel further exploits this interest locality to meet its system performance goals (fairness, decentralized membership management and for reducing structure maintenance costs).



(a) Varying the number of best friends



(b) Coverage scatter plot for 6 best friends

Fig. 1. The subscription traces of LiveJournal users (minimum of five subscriptions) show that if we greedily select six best friends for each user (to maximize that user's coverage), the friends provide complete coverage for over 95% of users.

1.1. System goals

We now elaborate on Rappel's main design goals:

(i) *Low publisher and subscriber overhead:* Rappel's overhead arises mostly from bandwidth, and is of two kinds - control and data. The data bandwidth is used for receiving and relaying the updates themselves, whereas control bandwidth is used for maintaining overlay neighbors. A low and scalable overhead at the publisher translates to bandwidth and resource savings, and thus a higher return on investment. For the system to scale with the number of subscribers, the subscriber overhead needs to be low.

(ii) *Zero noise:* As pointed out previously, the receipt of any feed update at a node that it is not subscribed to is classified as noise. Noise violates fairness since a node's contribution is not commensurate with its subscriptions. Thus, Rappel aims to achieve zero noise. Our experiments in Section 6 will show that Rappel provides better fairness than existing p2p multicast systems.

(iii) *Fast update dissemination:* Simultaneously lowering the publisher overhead and achieving zero noise might result in higher latencies to disseminate updates. Thus, another goal of Rappel is to provide soft real-time behavior, whereby each update is disseminated to all interested subscribers as rapidly as possible. Fast update dissemination is necessary to support dissemination of live sports scores, stock trackers, live blogging [43], etc. More concretely, we desire the update dissemination latency to be only a small factor greater than the direct IP route from the publisher. A low stretch factor is especially useful in end-user satisfaction for hosts that are "far" from the publisher.

1.2. Contributions of Rappel

Rappel solves the above goals by adopting several unique design decisions. First, Rappel maintains a decentralized control-plane overlay (*friends overlay*) among subscribers by leveraging both interest locality as well as network locality among these nodes. Second, the friends overlay is used for executing fast decentralized joins of

nodes into the subscription group for each feed. Third, Rappel creates, among the nodes of each feed's subscription group, a spanning tree for dissemination of updates. The tree is maintained to provide not only zero noise but also low latency and stretch factor relative to the native IP latency (for receiving updates at each node).

We evaluate Rappel via both trace-driven simulations and a PlanetLab [33] deployment. Our experiments make extensive use of real workloads: RSS subscription traces from LiveJournal, churn traces from a p2p network, real Internet topologies, and traces of network latency fluctuations. More importantly, by matching experimental results from simulation and deployment, we are able to show that our simulator accurately predicts the performance of Rappel at a larger scale (in a PlanetLab type environment). Our experiments reveal that in networks consisting of several thousands of subscribers and hundreds of publishers, Rappel hosts spend a median bandwidth of 100 Bps. They are able to obtain updates within a few hundreds of milliseconds to a few seconds after the publisher posts the update. Rappel also achieves very low stretch factors. Finally, we compare Rappel's performance against Scribe [8], a popular p2p multicast system.

2. Design overview

In this section, we present an overview of Rappel. An architectural overview is provided by Fig. 2. Sections 3–5 will elaborate on the details.

In this paper, for simplicity of exposition, we assume that there is a single publisher per feed. Our model generalizes to multiple publishers per feed in a straightforward manner. In a generalization, each feed has a "master publisher", which acts as the root node. All other authorized publishers (i.e., secondary publishers) send their updates to the master publisher, which disseminates the updates to the feed subscribers.

2.1. Rappel components

The design of Rappel is based on two major components. Firstly, Rappel constructs a dissemination tree for each feed, wherein only the subscribers of a particular feed join the tree. While a node could have joined the dissemination overlay in a top-down manner by contacting the publisher, this would lead to high join traffic at the publisher. Moreover, this also puts disproportionally high load at subscribers closer to the root, especially in popular feeds. The join traffic also increases with network churn. Churn has been observed to be as high as 25% per hour in contemporary p2p systems [1].

To improve the reception latency of feed updates, the dissemination trees are maintained to continually reduce the stretch factor. In the face of network churn, a node utilizes a periodic rejoin process to locate a new parent that improves its stretch factor. This results in the compaction of the tree and improved dissemination latency for all its descendants (Section 4).

Secondly, to mitigate excessive and disproportional traffic due to joins, Rappel builds a proximity-aware "friendship" overlay. Each node seeks to find a set of nodes ("friends") that are both close in the network and provide good subscription coverage. Subscription coverage refers to the percentage of node n_i's subscribed feeds that are in common with at least one of n_i's friends. High subscription coverage allows nodes to rapidly join the dissemination trees for a newly subscribed feed via friends. Having these friends in close network proximity allows the joining node to integrate into the dissemination tree without a drastic increase in the stretch factor. An added bonus of the friendship overlay is that it allows a node with numerous subscriptions to join the dissemination trees by contacting only a small set of highly effective friends. Rappel relies on gossip to discover better friends. Using an utility-based approach, Rappel stays converged to a good set of friends (Section 3).

2.2. Building blocks

Rappel leverages two basic building blocks: (a) a *network coordinate* system that enables estimation of the network proximity without repeated empirical measurements; and (b) *Bloom filters* that aid in quick computation of the subscription overlap between nodes, capturing interest locality.

Firstly, we use Vivaldi network coordinates [12] to estimate the network distance between nodes. Vivaldi maps nodes onto an n-dimensional Euclidean space so that inter-node latencies can be estimated directly via the Euclidean distance between the nodes' coordinates. Vivaldi nodes compute and maintain their coordinates based on differences between actual and predicted latencies.

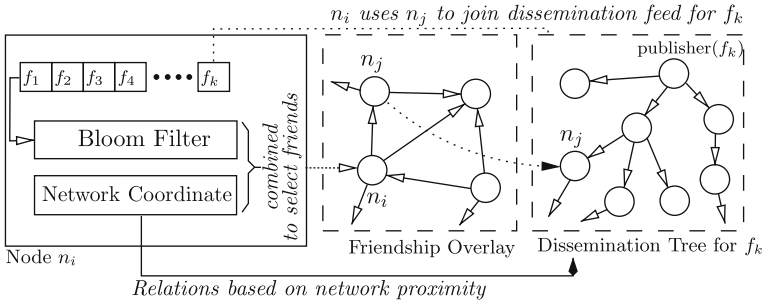


Fig. 2. Architectural overview of Rappel: the building blocks, the friendship overlay, and a per-feed dissemination tree.

Secondly, to represent each node's subscription set, we use a Bloom filter [5]. A Bloom filter *compactly* represents a large set of data using a bitmap in $O(n)$ time, where n is the number of keys. In Rappel, the Bloom filter for each node is created by first initializing the bitmap to zeros, then using multiple hash functions to map the URL of each subscribed feed to bits in the map (by setting the mapped bits to '1'). An inclusion test for a key (i.e., feed URL) can be performed in $O(1)$ time by checking if the hash function mapped bits are all '1'. As a result, Bloom filters are subject to false positives in the inclusion test for a key. Rappel's design takes this into consideration and directly verifies the presence of the key from the source node when necessary.

The size of the Bloom filter determines a trade-off between bandwidth and rate of false positives. However, the rate of false positives is independent from the number of publishers in the system. For a node's Bloom filter, the false positive rate depends only on the number of subscribed feeds. Given that, we use Bloom filters with 1,024 bits and three different uniform hash functions – this gives a false positive probability of 0.25% for a node subscribed to 50 feeds, 1.6% for 100 feeds, and 8.7% for 200 feeds. Since RSS subscription set sizes appear to follow a Zipf-like distribution [25], we believe the above setting is reasonable. The loss of accuracy for the few peers with large subscription sets (incurring a few more messages when key verification fails) is largely compensated by the bandwidth saved at most peers[1].

3. The Rappel friendship overlay

In this section, we describe the algorithms used to build and maintain the interest and network proximity-aware friendship overlay in Rappel. The friendship overlay will be leveraged to let a node quickly join dissemination trees of subscribed feeds. Rappel utilizes two techniques to thread the building blocks (see Section 2.2) together. These techniques are: (1) a *utility function* to calculate the proximity between any node pair, as a function of both network distance and interest overlap, and (2) a *gossip-based voting and audit mechanism* that enables a node to discover new friends. Our experiments find these methods are highly effective in locating both interest and network locality in practice (Section 6).

Below, we describe the soft state stored at each node (Section 3.1), utility calculation of the friends set (Section 3.2), the gossip protocol used to discover candidates for friendship (Section 3.3), and improvement of the friends set via periodic audits (Section 3.4).

3.1. The soft state

Each Rappel node maintains soft state in the form of three data structures: a *friends set*, a *candidates set*, and a *fans set*. While these data structures bear some similarities to those in existing systems, we define them below for completeness. The methods used to compose and maintain the soft state are described in Sections 3.3 and 3.4.

3.1.1. Friends set

The primary goal of the friends set is to provide maximum subscription coverage for each node. A node n_i maintains a set of friends $\texttt{FRIENDS}(n_i)$ containing nodes with close proximity to itself. Each entry for a node in $\texttt{FRIENDS}(n_i)$ is stored as a four-tuple. The four-tuple pointing to a friend n_j consists of the IP address of n_j ($n_j.address$), its network coordinate ($n_j.coord$), its subscription Bloom filter ($n_j.Bloom$), and the last time n_i heard from n_j ($n_j.last\_refresh$).

To maintain a low and fair control overhead due to the friendship overlay, we place an upper bound α on the friends set size at any Rappel node, i.e., we require $|\texttt{FRIENDS}(n_i)| \leqslant \alpha$. Our experiments (in Section 6.3) reveal that in a network with up to 10,000 nodes, a value of $\alpha = 6$ suffices. Due to interest locality (see Section 1), we believe that a low value of α may work with larger networks as well.

3.1.2. Fans set

To allow a node the unilateral flexibility to improve its friends set, friend relationships are asymmetric. For example, a node n_i subscribing to a large number of feeds may be desired as a friend by node n_j subscribing to a small subset of those feeds. While the friendship benefits n_j, it may not benefit n_i. Hence a separate fans set is needed to track and limit inverse friends relations. The fans set of node n_i consists of all nodes n_j that have n_i in their own friends set. We bound the "fanship" load at nodes such as n_i by limiting the number of fans, i.e., $|\texttt{FANS}(n_i)| \leqslant 2 \cdot \alpha$. Having a fans set that is twice as large as the friends set provides the flexibility needed to construct the friendship overlay, while preventing overload.

3.1.3. Candidates set

Each node n_i also maintains a candidates set, denoted as $\texttt{CANDIDATES}(n_i)$. The candidates set contains the nodes that may be audited for inclusion in the friends set. Each entry therein pointing to a node n_j is composed of a six-tuple. The first four entries of this tuple are akin to a friends set entry - the IP address, the network coordinate, the subscription Bloom filter, and the time last heard from. The last two entries help rank the best candidates - they are number of votes for n_j ($n_j.votes$), and whether or not the candidate has been audited ($n_j.audited$ – a Boolean value). The last two values are used for periodically auditing candidate nodes for inclusion in the friends set (Sections 3.3 and 3.4).

3.2. The utility of a friends set

The friends of a node should provide the node with good subscription coverage while being in close network proximity. In this section, we devise a utility function that attempts to capture both interest and network proximity derived from the friends set.

Given two nodes n_i and n_j, the utility of n_j to n_i should be derived from two components: (i) the network distance;

[1] Due to their encoding, Bloom filters have the added advantage of increasing privacy of a subscription set.

and (ii) the subscription overlap. The first can be computed using the Euclidean distance between n_i and n_j in the network coordinate space, i.e., $\|n_i, n_j\|$. The later can be derived using the intersection of '1' bits between the two nodes' Bloom filters, i.e., $|n_i.Bloom \cap n_j.Bloom|$. However, this may impose a "fanship overload" at nodes subscribing to numerous feeds. Hence, we normalize the metric using the *Jaccard index* [42], that is, by dividing it with the union of '1' bits between the two nodes' Bloom filters, i.e., $\frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}$.

More concretely, a prospective friend n_j that is nearby in the network (low $\|n_i, n_j\|$) should have a high utility as long as there is some subscription overlap. On the other hand, a high subscription overlap $\left(high \frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}\right)$ should also have a high utility as long as it is not too far in the network. As such, we use the following to calculate the utility of n_j to n_i:

$$Utility(n_i, n_j) = \frac{1}{\|n_i, n_j\|} \times \frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}$$

To illustrate how the utility calculation is performed, we present a simple example. Let there be two nodes: A and B. Node A subscribes to feeds f_1, f_2, f_3, f_4, and f_5, whereas node B subscribes to feeds: f_1, f_3, f_5, and f_7. Between the two nodes, there are three shared feed subscriptions out a total of 6 feeds. As such, the overlap in the subscription interest of the two nodes is 0.5. This number is multiplied by the inverse of their network distance, measured as the latency to send a message from node A to node B. For instance, if node A and node B are on the same LAN, with a latency of 1 ms, their utility value would be 500. However, if node A and node B were on different continents, with a latency of 100 ms, their utility value would only be 5. Now, suppose there is another node C, which subscribes to feeds f_2, f_4, and f_6. Between node A and node C, they share 2 feed subscriptions out of a total 6 feeds. Hence, the interest overlap between node A and node C is only 0.33. If both node B and node C were equidistant from node A, then node B would provide higher utility to node A than node C, given the higher interest overlap.

To generalize the utility function to the entire friend sets at node n_i, a special *bonus* is given to friends that uniquely share subscription with a node. This may help find dissemination trees for less popular feeds. Concretely, a higher utility is given to a friend that *uniquely* matches at least one bit in the Bloom filter of n_i. In the previous example, if both nodes B and C were in the friends set of node A, node B uniquely covers feeds f_3 and f_5, whereas node C uniquely covers feed f_4.

The comprehensive utility function is shown in Fig. 3. In order to better understand the benefits of the utility function, Section 6.3 will separately evaluate the interest locality and network distance components.

3.3. Maintenance of candidates set via gossip

A node n_i continually monitors the liveness of its friend n_j via periodic ping requests. A liveness check is a lightweight operation and hence nodes perform it often: once every 30 s. To indicate liveness, a pinged node sends back an acknowledgment. As friends and fans are inverse relations, node n_j implicitly uses the ping request from n_i to confirm the liveness of n_i.

A friend failing to reply within a timeout period is deemed as failed. We use a timeout value of 15 s, as median end-to-end node latencies on the Internet are two orders of magnitude smaller (see Section 6 for further details). Similarly, if a node does not receive a new ping request from a fan within the keep-alive period plus the timeout interval, the fan is deemed as failed. A failed friend causes n_i to seek a replacement friend, as described in Section 3.4. On the other hand, a failed fan is merely removed from the fans set (i.e., no need to seek a replacement).

We observe that the ping-ack mechanism can also be leveraged as a gossip protocol in order to evolve the candidates set. By piggybacking the friends set with every ping-ack response, nodes that are up to two-hops away in friendship can be discovered and added to the candidates set. These nodes include friends of friends, as well as friends of fans: these represent the collection of nodes in close network and interest proximity. Further, updates in the two-hop neighborhood are captured by the very next ping-ack.

Whenever a new remote node n_j is encountered, an entry for n_j is created in the candidates set with $n_j.audited$ set to `false`, $n_j.votes$ set to 1, and $n_j.last\_refresh$ set to the current time. To reduce bandwidth overhead, the Bloom filter and network coordinates for this node are not fetched at this time. Whenever n_j is heard from again, $n_j.votes$ is incremented by 1 and $n_j.last\_refresh$ is updated. Stated differently, a vote is implicitly cast for friends two-hops away with each ping-ack message. This gives higher weight (based on vote count) to candidates that are present on more than one "nearby" friend sets, i.e., such candidates are likely to have better network and interest proximity.

Note that Rappel does not ping candidate nodes - the candidate set is kept up to date by evicting inactive candidates. Firstly, the candidate set is restricted to a fixed size.

$$Utility(n_i, \mathtt{F}(n_i)) = \sum_{n_j \in \mathtt{F}(n_i)} \frac{1}{||n_i, n_j||} \cdot \left(\overbrace{\frac{|n_i.Blm \cap n_j.Blm|}{|n_i.Blm \cup n_j.Blm|}}^{\text{base}} + \overbrace{\frac{|\{b | b \in n_i.Blm \cap n_j.Blm \text{ and } \forall_{n_k \in \mathtt{F}(n_i) - \{n_j\}} b \notin n_k.Blm\}|}{|n_i.Blm|}}^{\text{bonus}} \right)$$

Fig. 3. The utility of a friends set depends on the network – and interest – proximity, with a bonus for nodes that uniquely share common feed subscriptions with n_i. For brevity, `FRIENDS`(n_i) and $n_i.Bloom$ are denoted by $\mathtt{F}(i)$ and $n_i.Blm$ respectively.

Once this limit is saturated, we use an eviction policy that eliminates the least recently heard from node (akin to LRU eviction). The maximum size of the candidate set is $(3 \cdot \alpha^2 + 2 \cdot \alpha)$, in order to capture all friends two hops away even if there is no overlap amongst them. This includes friends of friends (α^2), friends of fans $(2 \cdot \alpha^2)$, and fans themselves $(2 \cdot \alpha)$.

Algorithm 1. Periodic auditing of a candidate node. For brevity, we omit the friendship request sent to n_c.

```
n_i::Improve-Friend-Set (Candidate n_c)
begin
  base ← highest ← Utility(n_i, FRIENDS(n_i));
  foreach n_j ∈ CANDIDATES(n_i) do
    current ← Utility(n_i, FRIENDS(n_i) − n_j ∪ n_c);
    if current > highest then
      // Evicting n_j increases utility
      highest ← current;
      victim ← n_j;
  if highest > (1 + δ) × base then
    // Pending positive ACK of friendship request from n_j
    FRIENDS(n_i) ← FRIENDS(n_i) − victim ∪ n_c;
    n_i::Reset-Audit-Flags
end
```

3.4. Improving the friends set via audits

A node n_i periodically attempts to improve its friends. The audit operation builds atop the background voting mechanism already described in Section 3.3. Each node instantiates an audit periodically, i.e., once every 30 s. Audit operations are asynchronous at each node and do not require any global changes or synchronization. Algorithm 1 describes the audit operation, and we explain below in words.

First, the unaudited candidate n_j with the maximum number of votes is selected as a prospective friend. At this point, the n_j.*audited* flag is set to `true`. Further, node n_i fetches the Bloom filter and network coordinate of n_j directly from n_j during this process if either was previously unknown.

Now, if the friends set is not full at the time of an audit operation, i.e., $|\texttt{FRIENDS}(n_i)| < \alpha$, n_j is automatically deemed to a viable friend. However if the friends set is full, a prospective friend can only be included in the friends set if coupled with eviction of a incumbent friend. Further, this should only be done if the swap increases the utility of the friends set. Amongst all the friends sets formed with *each* possible eviction of an incumbent node (coupled with inclusion of n_j), we find the friends set that yields the highest utility. If this set has a higher utility than the current friends set, node n_j is deemed to be a viable friend. To prevent hysteresis, a new friends set must increase the utility by at least $\delta\%$ (=1% in our experiments). If no such case exists, the friends set is left unchanged.

Once the node n_j has been deemed a viable friend, a friendship request is sent to it. Node n_j approves friendship requests on a first-come first-serve basis until its fans set is full. Node n_j piggybacks its friends set to the friendship request response, so that node n_i can continue to expand its candidate set. If n_j denies the friendship request (it does so only if its fans set is full) from n_i, n_i repeats the audit operation if n_i's friends set is not full. Finally, on any change to the friends set at n_i, all $n_j \in \texttt{CANDIDATES}(n_i)$ have their n_j.*audited* and n_j.*votes* flags reset to `false` and 0 respectively, so that they are once again open to periodic auditing.

Bloom filters and network coordinates change only infrequently. This is because feed subscriptions and unsubscriptions at a node occur at much larger time scales than audits, while network coordinates are not changed for the duration of a Rappel session, i.e., an online period (more details in Section 6.1). As a result, we version both the Bloom filters and network coordinates. To save bandwidth, Bloom filter and network coordinates need only be fetched during the first audit. However, the Bloom filter and network coordinates for the friends set need to be kept up to date as they are used for audit operations. Hence, a node piggybacks the latest version numbers of its Bloom filter and network coordinates with each message. Whenever a node learns about a newer version of a Bloom filter or network coordinate of a friend, e.g., via a ping-ack message, it fetches the latest version directly from that friend.

4. Per-feed dissemination trees

With the goal of avoiding noise in update dissemination, Rappel constructs one spanning tree for each feed's subscriber group. The structure and the function of the dissemination trees is detailed in Section 4.1. Recall from Section 1 that it is our goal to have a: (i) low overhead at publishers and subscribers, and (ii) low latency and stretch factor (w.r.t. the direct IP route from publisher to the subscriber) for updates. In Section 4.2, we present a bottom-up process that aids a node in locating a "better" parent in the tree. As opposed to a centralized top-down join at the publisher, a bottom-up approach reduces the traffic load incurred at the top levels of the tree close to the root and instead balances the load out evenly. The traffic load due to centralized joins increases if the system exhibits high churn. In order to keep update latencies low in face of network churn, the node periodically rejoins the tree (Section 4.3). Lastly, in Section 4.4, we present the approaches that help maintain continuity of service to descendants of a properly departing node.

4.1. Structure and function of dissemination trees

Before delving into the details of our protocols, we briefly comment on the structure and function of the dissemination tree. A given node maintains one parent and a few children per tree. The node also maintains the coordinates of the feed publisher and the list of its ancestors in the tree, starting from its parent all the way up to the root (publisher), both of which a node learns from its parent. The ancestor chain is kept up-to-date by piggybacking it atop ping-ack messages sent from the parent node to each of its child nodes.

Each node continually monitors the liveness of its parent via periodic ping requests akin to the ping-ack

mechanism described in Section 3.3. As the parent–child relation is reciprocal, a parent node implicitly uses the ping request from a child node to confirm the child node's liveness. If a node's parent is deemed as failed, the node attempts to find a new parent via a tree rejoin. If a child is deemed as failed, the parent node merely deletes the child entry.

For a given tree, the maximum number of children at any node is parameterized by β. This allows us to limit the data overhead at each node in the system, i.e., each node is limited to forwarding an update to up to β children for each of its feeds.

Too low a value for β leads to deep trees with high latencies, while too high a value overloads nodes. In Section 6.2, we show that a value of $\beta = 5$ works well in practice.

Given a dissemination tree for a given feed f_k, it can be used to both push and pull updates. Since fast dissemination is one of our goals, Rappel publishers push updates down the dissemination tree. While a push is used to send an update to all online nodes, a pull is used by a node to obtain missing updates from a new parent (immediately after a join or a rejoin). Thus, if node n_j's parent fails during the push-based dissemination of an update, n_j will pull the update from its new parent. In turn, n_j will push the update to its children. To facilitate pulls, each node maintains a cache of recently received updates – our implementation uses a cache size of 10 updates.

A single failure causes an additional delay at all the failed node's descendants. The expected additional delay at the descendants is 22.5 s since the pinging interval and timeout takes 45 s. Latency degrades linearly with the number of concurrent failures in the ancestry chain. However, one can expect the number of concurrent failures in the ancestry chain to be relatively low in a deployed system. For example, using an hourly network churn rate of 25% observed in the Overnet p2p network [1], the probability of having three concurrent failures (we pessimistically define concurrent to be within 1 min) in an ancestry chain of 20 nodes is only $p = 0.0125$.

While security issues are not a focus of this paper, we would like to point out a few things. Zero noise can be ensured even in non-collaborative networks if updates are signed by the publisher. Signed updates allows subscriber nodes to refuse forwarding for spurious publishers. Further, the signature can include a sequence number. If there is a lapse in sequence numbers, the missing updates can be pulled from another ancestor. To further safeguard this, a publisher can send a void update (i.e., an update with only the latest sequence number) periodically.

4.2. Locating a new parent: a bottom-up approach

One purpose of the friendship overlay is for a newly joining node n_j to locate an active node n_i for any feed f_k. Initially, the active node also acts as the parent node of n_i. However, this can lead to poor stretch ratio ("zig zag" paths) if the tree is not reorganized periodically. Therefore, in this section, we present an algorithm that selects a new parent in a bottom-up manner. The iterative process leads to the compaction of the tree and hence better stretch ratios.

Algorithm 2. Reception of a join request at node n_i from node n_j for feed f_k

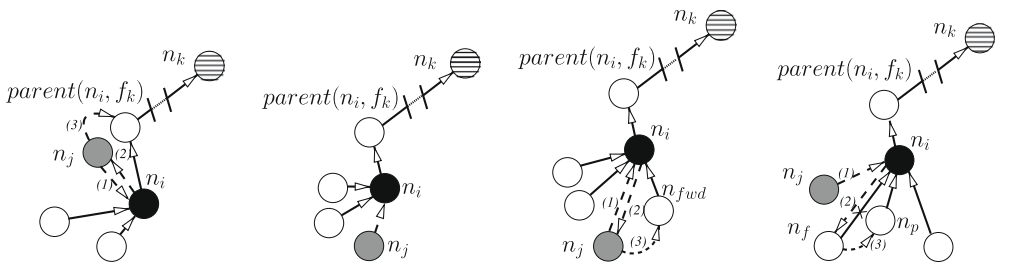
```
n_i::Receive-Join (n_j, f_k)
begin
    n_k ← publisher(f_k);
    if n_i does not subscribe to feed f_k then
        // False positive due to Bloom filter
        Send JoinDeny to n_j;
    else if ||n_i, n_k|| > ||n_j, n_k|| then
        // Figure 4(a): Requesting node is closer to publisher
        Send JoinForward (parent(n_i, f_k)) to n_j;
    else if |CHILDREN(n_i, f_k)| < β then
        // Figure 4(b): There is room for more children
        Send JoinOK to n_j;
    else
        CLOSER ← {n_c | n_c ∈ CHILDREN(n_i, f_k) and
                                ||n_c, n_k|| < ||n_j, n_k||};
        if |CLOSER| = β then
            // Figure 4(c): Every child is closer to publisher
            Find node n_fwd ∈ CLOSER closest to n_j;
            Send JoinForward (n_fwd) to n_j;
        else
            // Figure 4(d): Evict the child farthest-away
            Find n_f ∈ CHILDREN(n_i, f_k) farthest to n_k;
            CHILDREN(n_i, f_k) ← CHILDREN(n_i, f_k) − n_f ∪ n_j;
            Send JoinOK to n_j;
            Find n_p ∈ CHILDREN(n_i, f_k) closest to n_f;
            Send ChangeParent (n_p) to n_f;
end
```

Starting with the current parent, a join request is routed amongst the subscribers of f_k until a parent node for n_j is found. This procedure is described by the pseudo-code in Algorithm 2, illustrated in Fig. 4, and described below.

In selecting a new parent, Rappel always maintains the following invariant: *the parent of a node n_j must be closer to the publisher than n_j itself (in the network coordinate space)*. In other words, all descendants of a node are farther from the publisher than itself[2].

The main goal in this protocol is for the node n_j to find a prospective parent that is both closer than itself to the publisher of feed f_k (in network coordinate space), as well as has spare capacity to add an extra child (i.e., it has fewer than β children for feed f_k's tree). Suppose the current contacted node is n_i (initially, this is the active node). Using network coordinates, node n_i determines whether n_j is closer to the publisher than itself. If so, then n_j is redirected to n_is parent (Fig. 4a). Otherwise (n_i is closer to the publisher), if n_i has spare capacity to add a child, n_j becomes a child of n_i (Fig. 4b). Otherwise (n_i has no spare capacity), if all children of n_i are closer to the publisher than n_j, then it redirects n_j to the child closest to n_j (Fig. 4c). Otherwise (if at least one child of n_i is farther from the publisher than node n_j), then n_j becomes a child of n_i. In turn, n_i evicts the current child that is farthest from the publisher. The victim child is directed to rejoin the tree at the child of n_i that is closest to the victim child (Fig. 4d). The evicted child then repeats the joining protocol – this is not an encumbrance since nodes attempt to seek new parents periodically anyway (as the next section describes). As an optimization, the evicted child skips the next scheduled periodic rejoin.

[2] With the exception of rare ties, which are broken by lexicographical ordering of IP addresses.



(a) Redirect to parent (b) Accept (c) Redirect to child (d) Accept; evict a child

Fig. 4. The actions of node n_i on receiving a join request from node n_j for feed f_k. Note that n_k is the publisher node. For this example, we use a 2-D coordinate space and limit the number of children per node to 3.

4.3. Periodic rejoin operations

In order to maintain low stretch factors, especially under network churn (due to node joins and leaves), it is imperative that each subscriber node continually attempts to minimize its distance to the publisher. To achieve this, we use the convenience of the triangle inequality afforded by a (network) coordinate system.

Although it is well known that the triangle inequality does not hold within the Internet, it does however hold in an Euclidean coordinate space. Note that Dabek et al. [12] find that the number of major triangle inequality violations on the Internet is rare (around 5%), and hence, embedding network latency information into network coordinates remains effective. With this in mind, we observe that the *distance from a subscriber to a publisher, in the Rappel tree, is always minimized if the subscriber attempts to find a parent that is higher up the tree.* This is true because of the Rappel invariant (beginning of Section 4.2), whereby a node is closer to the publisher than any of its children.

Thus, each node n_j periodically attempts a rejoin at a non-parent ancestor. For this, the algorithmic steps represented by Fig. 4c and d move subscriber nodes to a place in the tree that reduces their stretch factor, irrespectively of the order in which nodes joined the tree. When a node moves up the tree, so do all its descendants. As a result, periodic rejoining has the added benefit of continually *compacting* the tree in a distributed fashion. Note that these rejoins are performed asynchronously by subscriber nodes and are *not* a global overhauling of the tree.

Two issues remain to be discussed: (i) selection of ancestors for the rejoin, and (ii) the frequency of rejoins.

If the rejoining ancestor was chosen with each ancestor having equal probability of being selected, nodes closer to the publisher (i.e., having low tree heights) would be overloaded with rejoin messages. To address this, we exponentially decrease the probability of an ancestor being selected as a function of its distance from n_j. Concretely, consider a tree with height H and fan-out β. The height of the publisher is $h = 0$. Let h_i and S_i denote the height of n_i in the tree, and the number of descendants in the sub-tree rooted at n_i respectively. A node n_j will attempt a rejoin at a non-parent ancestor n_a, i.e., the difference of heights of n_j and n_a is at least 2, i.e., $h_j - h_a \geqslant 2$. Node n_a is chosen with probability $\Pr[n_a] = \frac{\beta^{-(h_j - h_a)}}{\sum_{p=2}^{h_j} \beta^{-p}}$. This ensures that each non-leaf node n_i in a tree receives an expected $\sum_{p=h_i+2}^{H} (\sum_{q=2}^{p} \beta^{-q})^{-1} = \Theta(\log_\beta S_i)$ overhead of incoming rejoin messages per period. This is far more preferable than centralized joins which would overload the root node and those below it. Too low a rejoin frequency might cause tree degradation while a high frequency will incur a greater cost. In practice, we found that a rejoin period of 10 min at each Rappel node works best – this is true even for scenarios with heavy network churn.

4.4. Proper leave operations

A node that leaves a tree (i.e., due to an unsubscription or due to a user-requested disconnect) attempts to provide *continuity of service* to its children nodes. Stated differently, (i) during a rejoin, while a node seeks a new parent, it must continue to receive updates from its current parent and disseminate them to its descendants; while (ii) during departure, a node must continue providing service to its children while they seek new parents. These two operations are labeled as *proper rejoin* and *proper leave* procedures.

The proper rejoin procedure ensures that no updates are missed by a node and its descendants while it switches parents. Let us assume that during a periodic rejoin, node n_j switches from its current parent node n_i to a new parent node n_k. The proper rejoin protocol simply requires node n_j to discover a potential new parent in the background. When, and if, a better parent n_k is found, n_j first connects to n_k before leaving n_i. Duplicate updates (i.e., updates received both from n_i and n_k) are simply dropped.

The proper leave procedure aims to maintain the continuity of service to children nodes of the departing node n_j. To this end, n_j continues to forward messages to its children until they are able to find new parents. The proper leave operation is as follows: node n_j first notifies its actual parent node n_i to accept a specified node n_c as an additional child. This node n_c is the child node of n_j that is closest to the publisher. Node n_i will accept n_c as a child even if it means having more than β children momentarily. All other children of n_j are instructed to rejoin the tree at n_c. Upon finding a new parent, the children nodes notify and leave n_j. Once n_j receives notifications from all its children (or after waiting for 30 s), it leaves the tree by notifying its parent n_i. Note that any failures during the leave protocol can be detected and recovered from using the aforementioned ping-ack mechanism.

5. Bootstrapping

In this section, we describe the three bootstrap techniques required in Rappel. Firstly, a node may need to join a dissemination tree for a newly subscribed feed in an ongoing session. Our solution leverages the existing friendship overlay. Secondly, a node may reenter the Rappel system at the start of a new Rappel session, i.e., after an offline period. Our approach uses the stale friends set to quickly create an effective friends set and join numerous dissemination trees via only a few friends. Thirdly, there is a special case of a virgin Rappel session. In this case, we bootstrap both the friendship overlay and join the different per-feed dissemination trees.

5.1. Joining a feed

We first consider the case of a node already in Rappel, attempting to join the dissemination tree of a newly subscribed feed. To join the dissemination tree of a feed f_k, a node examines whether f_k is (could be) encoded in any of the Bloom filters of its current friends set. If there are matches, the closest friend (as measured by network proximity) is requested to be the parent. Note that the request sent to the friend serves as an implicit step to verify the Bloom filter's correctness. If none of the friends provide coverage for the feed, the node contacts the publisher of that feed directly. This ensures that the publisher is contacted only in the rare case when even one friend fails to provide subscription coverage for a feed.

5.2. A virgin Rappel session

A node joining Rappel for the first time ever (its virgin Rappel session) has an empty friends set. Instead of having the node join at each feed's publisher directly, we use a *staggered* join strategy to reduce the load on the publishers and simultaneously construct a friends set.

During the staggered join, the virgin node initially joins dissemination trees for a few of its subscribed feeds (ordered randomly) directly at the respective publishers. The direct node joins help discover several nodes via the iterative tree join process described in Section 4.2. The first few of these nodes help *seed* the friends set. Further, for the duration of the staggered join process, the auditing process is performed continually (see Section 3.4). To let the friends set evolve, we enforce an interval of 20 s before each successive join at the publisher. Ideally the friends set will gain high utility and provide feed coverage for the remaining feeds. In reality, we found that this did indeed happen: a high utility friends set is reached after as few as 4–12 tree joins (the number depends on the node's feed subscription set). Hence, any unfulfilled joins are performed directly at the publisher after performing the 12th staggered join.

Note that an effective friends set – one that provides high subscription coverage – allows a virgin node to join numerous dissemination feeds via only a few friends. This greatly reduces the join load (and time) on nodes that subscribe to tens or hundreds of feeds. Note that a node performs the periodic rejoins (see Section 4.3) only after it has already joined all the required dissemination trees.

5.3. A reentrant Rappel session

If a node is rejoining the Rappel system, it probes its stale friends set to bootstrap a new friends set. We find that, in most cases, even if only one stale friend is alive, a highly effective friends set can be quickly achieved. Using the new friends set, a node iteratively joins as many of the subscribed feeds as possible. This is the common use case: based on its stale friend set, a reentrant node with join numerous dissemination trees via a handful of friends. If unable to locate a stale friend, the node performs a staggered join process while letting its friends set evolve.

6. Experiments

We implemented Rappel in C++ using a common codebase for both network simulation and PlanetLab [33] deployment. Our simulator is driven using traces of Internet topology [48], latency measurements [23], churn [1], and RSS subscriptions [26]. Simulation allows us to study Rappel's behavior at larger scales than PlanetLab, yet gives believable results due to an extensive usage of traces. To support this argument, we show that the experimental data from the simulation results closely match the PlanetLab results. Lastly, we compare Rappel with Scribe in a simulated setting: we show that Rappel performs minimally worse in terms of absolutely update dissemination latency while being fairer due to its noiseless design.

6.1. Experimental methodology

We describe below the methodology via which our simulation makes use of the known Internet topology, latency traces, churn traces, and RSS subscription traces. Then we briefly touch on our PlanetLab deployment.

6.1.1. Simulation network topology

In order to accurately model the underlying network, we used the AS network topology collected by Zhang et al. [48]. This data set consists of 20,062 stub networks, 175 transit networks, and 8,279 transit-and-stub networks. Simulated end-hosts within a randomly selected stub network. However, the Zhang information does not include inter-AS latency measurements. To augment this lack of information, we use a 4-h data set of latency measurements from PlanetLab nodes (collected by Ledlie et al. [23]) to extrapolate inter-AS link latencies. Via trial and error, we find a inter-AS latency distribution that results in a match between the end-to-end latencies calculated by the simulator and the end-to-end latencies measured in the Ledlie trace. The inter-AS latency distribution is as follows: 10% of inter-AS links have a latency between 0 ms and 4 ms (selected uniformly at random), the next 30% of inter-AS links have a latency between 4 ms and 30 ms, and the final 60% of inter-AS links have a latency between 30 ms and 115 ms. Fig. 5 shows that the resulting end-to-end latencies closely model the observed median latencies between PlanetLab nodes. Further, and more importantly, by associating each simulated node pair to a PlanetLab node pair from the trace (based on median latency), we inject realistic latency fluctuations in the simulator. Using this method, our simulator

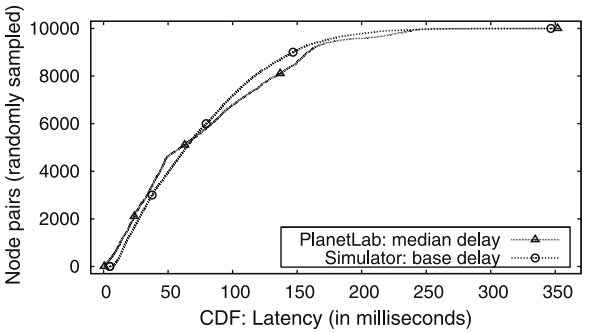


Fig. 5. The latencies modeled by our network simulator closely matches the latencies experienced within PlanetLab.

supports an arbitrary number of end-hosts with realistic and fluctuating end-to-end latencies.

Note that due to the memory overhead required to implement realistic latency fluctuations over Internet-scale topology, our simulation runs are limited to a few thousand nodes. Further, our simulator does not model bandwidth. As our simulation results match closely with the results from the PlanetLab deployment, we verify that control messages are sufficiently small that bandwidth does not become a constraint.

6.1.2. Simulation subscription workload

We used a real workload of RSS subscriptions obtained from the LiveJournal web service [26]. LiveJournal has a large community of users, and averages over 300,000 public posts per day by over 180,000 unique users. Each LiveJournal user maintains a "journal", which is an RSS feed that any other users can subscribe to. Our experiments map journals to publishers and users to subscribers.

Over six months, we obtained via LiveJournal's RPC services information about 1.8 million users[3]. This included: (i) a list of users subscribing to their journals; and also (ii) a list of journals subscribed by these users. In order to obtain a self-contained non-biased universe of subscriptions, we *randomly* selected a small *seed set* of journals from the trace. Next, we gathered the list of all users subscribed to at least one journal in the seed set. These users (and their respective journals) form the universe of nodes in our simulation. As an example, a seed set of 10,000 journals gave us a universe of 304,814 users. Next, based on the experiment, the X most subscribed-to journals in this universe were selected to be our publishers (the value of X depends on the experiment). Subscriptions of users outside the universe's publishers were pruned. Note that using the most popular publishers does not bias correlation, as our seed set is unbiased. The trace refinement procedure leads to a subtrace that exhibits similar characteristics to the smaller-scale RSS subscriptions trace presented by Liu et al. [25]. For brevity, we do not provide further trace analysis in this paper.

6.1.3. Simulation churn trace

In order to study Rappel's behavior under churn, we injected traces of node availability collected by Bhagwan et al. from the Overnet p2p network [1]. It should be noted that Overnet's hourly churn rate is as high as 25% of the total node population. The traces were collected by probing 2400 Overnet nodes at 20-min intervals. At each probing period, nodes were recorded as either being online or offline. To support more realistic churn events, we uniformly distribute the recorded churn events, i.e., node joins and leaves, over the given 20-min probing interval. Further, to support an arbitrary number of end-hosts, the original traces are replicated as necessary.

6.1.4. Deployment atop PlanetLab

We deployed Rappel on nearly 400 nodes within PlanetLab. The Rappel binary uses TCP connections between Rappel peers, and UDP datagrams for Vivaldi messages and other primitives described next. In order to accurately measure sub-second update latencies in spite of hardware clock skews and drift, our measurement code runs periodic clock synchronization between each node and a reference server. Further, to calculate the direct IP latency between a subscriber and a publisher, each subscriber sent a periodic `PING` message to the publisher.

As Rappel builds dissemination trees using network coordinates as a first-class primitive, unnecessary fluctuations of network coordinates may hamper performance. We use heuristic improvements to Vivaldi suggested by Ledlie et al. [23] which provide a reasonable trade-off between accuracy of coordinates and stability in their values over time. Further, we affix the network coordinates of a node for the duration of a session, i.e., an online period. When a node joins the Rappel system, it quickly calculates its network coordinates using 18 geographically diverse landmark servers. This process is completed in a matter of seconds.

6.1.5. Experimental settings

All our experiments use the value of $\alpha = 6$ (the number of friends), $\beta = 5$ (the fan-out of dissemination trees) unless mentioned otherwise. The publishers disseminate an update of size 1 KB once each minute for experiments in Section 6.2. For the simulation, user subscription sets were mapped randomly to end-nodes, whereas real users are likely to be correlated with location. For example, subscriptions to New York Times RSS feed are likely to be most heavily concentrated around New York City. As a result, the random mapping gives more pessimistic results for Rappel.

6.2. Per-feed dissemination trees

This section studies the characteristics of the per-feed dissemination trees formed by Rappel.

6.2.1. Rappel tree snapshot

Fig. 6 illustrates an actual dissemination tree formed using 25 PlanetLab subscriber nodes. The publisher node was located at `UIUC` (the largest square). The outgoing arrows connect a node to its children (limited to $\beta = 3$ for this experiment). This experiment demonstrates the strengths of both the underlying network coordinate system and the bottom-up tree construction algorithm.

[3] An anonymized subset of this trace is available upon request.

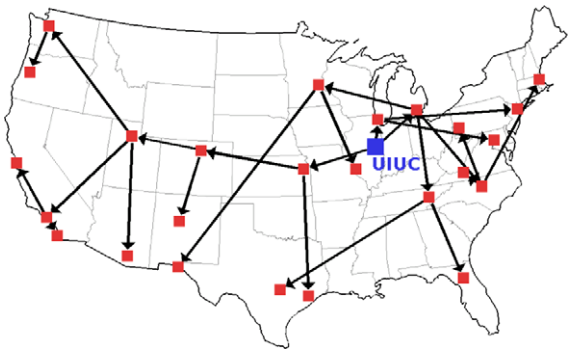


Fig. 6. The geographic projection of a per-feed dissemination tree constructed using PlanetLab nodes.

6.2.2. Update latency and stretch factor

To observe the performance of per-feed dissemination trees in a larger system, we studied, under both simulation and within PlanetLab, a group of 250 subscriber nodes with 1 publisher. Furthermore, we cause 50% of the nodes (selected randomly) to fail simultaneously at time $t = 2$ h, and then rejoin at $t = 3$ h.

We study (1) the update latency, defined as time between publisher creating an update and a subscriber receiving it, and (2) the stretch factor of update latency. Fig. 7 shows the median and 95th percentile (across subscriber nodes) data for both simulation and PlanetLab setups. Note that the same legend is shared across all three figures.

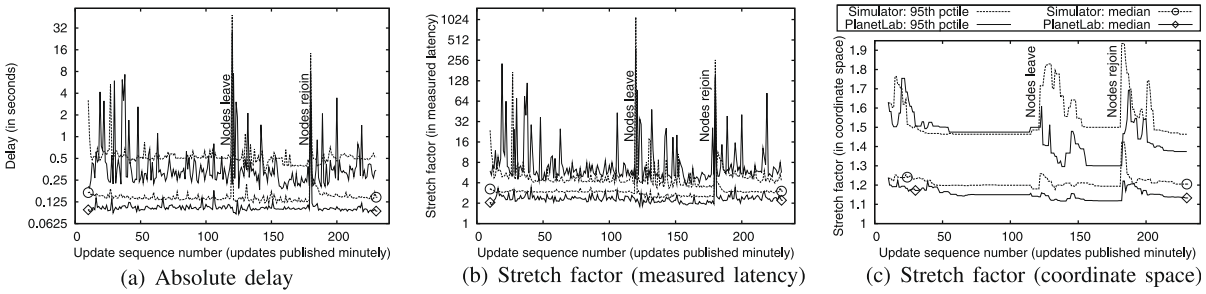
First, we observe a close match between simulation and PlanetLab results on all these plots (both median and 95th percentile). Second, Fig. 7 shows that 50% subscribers receive the update within 100 ms and 95% of nodes receive it within 500 ms. Large spikes in the 95 percentile dissemination latency are noticeable during initialization and right after the churn events – which fade rapidly. The median update latency fluctuates only moderately in spite of 50% instantaneous churn. Smaller spikes in the 95 percentile data are due to periodic rejoin operations (Section 4.3) – this is because some nodes are evicted during the process (Fig. 4). Updates to these nodes and their descendants are delayed until the node pulls the missed update(s) from its new parent.

Third, Fig. 7b and c plots the stretch factor for updates in two different ways. Fig. 7b depicts the stretch factor w.r.t. direct network latency from subscriber to publisher (measured periodically and continuously). The median stretch factor stays between 2 and 4. In Fig. 7c, we plot the stretch factor w.r.t. the subscriber–publisher network distance in the *underlying coordinate system*. The median stretch factor in this plot stays around 1.15, and 95% of the nodes have a stretch factor below 1.25, which are both satisfactorily low. Since Rappel relies solely on the underlying network coordinate system for its network proximity, we can conclude that *the per-feed dissemination trees effectively exploit network proximity to the extent that the underlying coordinate system is accurate*. A reason for the increased 95th percentile stretch factors in Fig. 7b (measured stretch ratio) vs. Fig. 7c (network coordinate stretch ratio) is due to tree rejoins. For an ongoing update (only), a tree rejoin causes the coordinates stretch factor to improve, while the update latency degrades due to an update pull. A rejoin also requires the establishment of a new TCP connection to the parent.

For the next experiment, we simulate a network with 1 publisher and 5000 subscribers. Fig. 8a shows the scatter plot of stretch factor (w.r.t. network coordinates) for each subscriber during dissemination of the final update (at $t = 4$ h). We observe from the plot that the nodes farthest from the publisher receive the update with low stretch factors. A low stretch factor implies that the dissemination path does not "zigzag" beyond a minimal extent. Thus, nodes farthest away from the publisher are successful in finding good dissemination paths. The high stretch factors present in nodes closer to the publisher are less of a concern since updates to these nodes are disseminated within a short absolute latency.

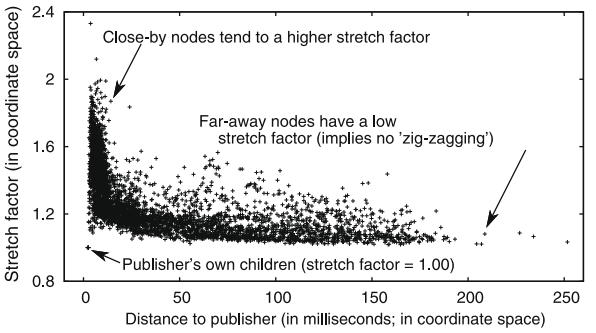
6.2.3. Performance under continuous churn

We used the Overnet traces to simulate a network of 5000 continuously churned subscribers and 1 online publisher; the average churn rate was approximately 30 joins and 30 leaves per minute. For a series of 220 updates, we measured the update latency at a small group of 10 "observer" nodes, which were prevented from being churned. These observer nodes were used to compare the performance of a network under churn against a static network. While the observer nodes were not churned; their parents, children, and friends change continuously due to churn. Fig. 8b shows the CDF of the update latencies across each of the 220 updates at the 10 observers (using circle points on line). For comparison, we also plot data for a static

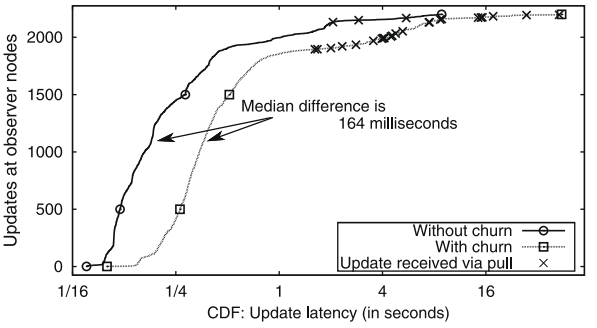


(a) Absolute delay (b) Stretch factor (measured latency) (c) Stretch factor (coordinate space)

Fig. 7. Running the experiment with 1 publisher and 250 subscribers on PlanetLab and our network simulator yields approximately the same results validating the "empirical correctness" of our simulation results.



(a) Stretch factor vs. network distance



(b) The effect of continuous churn

Fig. 8. The per-feed dissemination trees achieve low stretch factor, while quickly delivering updates to subscribers even under continuous churn.

network with 5000 subscribers (square points on line). Also, on both lines we mark the updates that were pulled by their respective nodes (cross points).

This plot shows that continuous and rapid churn worsens the update latency only moderately – the median difference is only 164 ms. Further, 85% of updates are received within 1 s. Higher latencies were caused due to pulls (i.e., after a node rejoins the tree), resulting in higher latencies at its descendants as well. The highest delays due to churn are around 45 s – likely due to the failure of a single ancestor in the dissemination path right after the update was published.

Lastly, we explore the value of β, i.e., maximum number of children, under the same churn conditions in another experiment (not shown). We observe that the performance of the tree improves with increasing β. However there is only marginal improvement, especially at the tail-end, after $\beta = 5$. The favorable load imposed on interior nodes justifies the choice of $\beta = 5$ for our implementation.

6.3. Locality-awareness of Rappel

We evaluated Rappel's ability to exploit interest locality based on subscription traces from LiveJournal. Starting with a seed set of 250 feeds, we obtain a network of 5582 subscribers using $X = 100$ publishers. Each user subscribed to only a subset of the 100 publishers. All results below are via simulations.

Fig. 9a is the scatter plot of the feeds covered by a node's friends set vs. the number of feeds subscribed by the node. As there are numerous coincident points, the plot also shows the median value for each data set. 91% of points lie on the perfect coverage line. Other points just below the perfect coverage line are nodes that have a good majority of their feeds covered. Observe that each node that subscribes to 9 or more feeds has a minimum of 6 feeds covered, i.e., number of feeds covered is at least the same as the number of friends ($\alpha = 6$). However, several nodes subscribing to 6 or fewer feeds are unable to exploit interest locality due to scarcity of locality information.

Each Rappel node has many neighbors (i.e., peers). However, few neighbors are used in multiple roles, e.g., a neighbor may be a child in one dissemination tree and the parent in another tree. We define neighbor overlap ratio as the total number of roles played by neighbors divided by the number of distinct neighbors. A neighbor overlap ratio greater than 1 signifies a bandwidth reduction due to reduced ping-ack traffic. Note that we use only friends, parents, and children to calculate this ratio, i.e., to prevent any fans and candidates from artificially inflating the ratio.

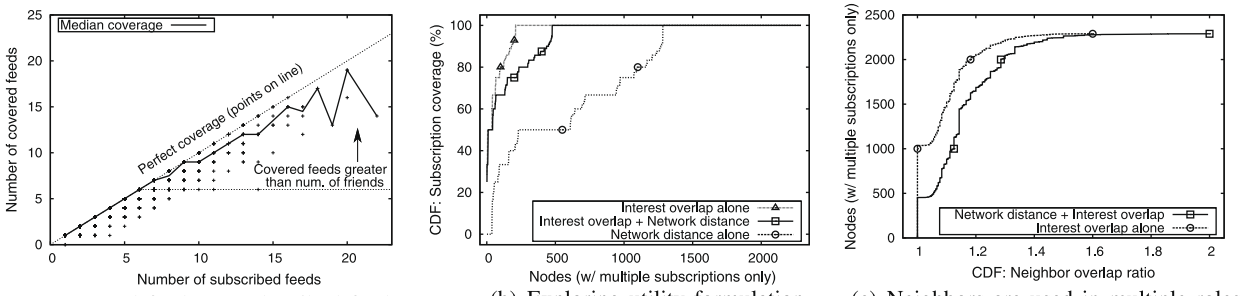
Fig. 9b evaluates different components of Rappel's friend selection heuristic (Section 3.2). The metric plotted is the CDF of the subscription coverage. The subscription coverage of a node is the percentage of subscribed feeds covered by at least one of its friends. Only multi-feed subscribers were used in this plot to eliminate high bias from single-feed subscribers. A CDF line that is farther to the left is more desirable. The plot shows that considering both network distance and interest locality provides comparable coverage to the "greedier" approach of considering only interest locality. On the other hand, Fig. 9c shows that 80+% of nodes are able to exploit some form of neighbor overlap if the friends set utility is calculated using both interest locality and network distance (data shown via square points on line). In comparison, the neighbor overlap ratio achieved by calculating utility simply based on interest overlap alone is much worse (data plotted with circle points on line). This shows that without the network proximity as a component to determine the friends set, a node is unable exploit overlap between its tree neighbors and its friends. We conclude that the Rappel utility function strikes a balance between network proximity and interest locality.

Lastly, we explore the effective size of α in another experiment (not shown). $\alpha = 4$ provides 100% coverage for 73% of multi-feed subscribers, with only marginal improvement for higher α values. Hence, we selected $\alpha = 6$ to limit the gossiping overhead at nodes, while providing high subscription coverage and friendship redundancy.

6.4. Comparison with Scribe

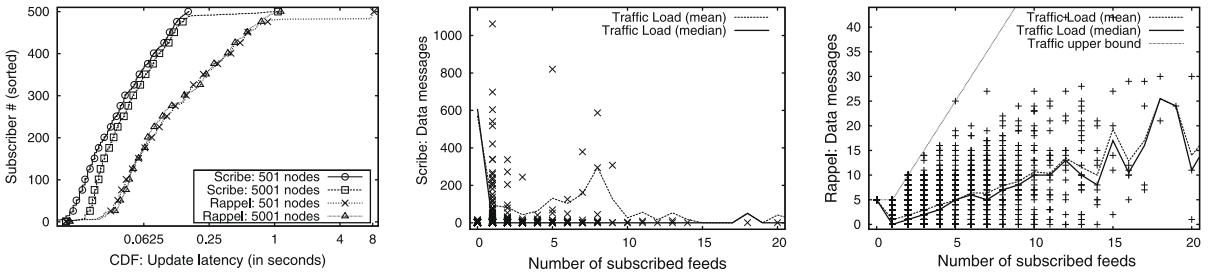
In this section we compare the performance of Rappel with Scribe [8]. Scribe is the underlying publish-subscribe system used by FeedTree [38]. Rappel trades off update dissemination latency (while still keeping it low) in order to achieve fairness.

We use the Scribe implementation available within FreePastry [15]. As the simulators for both systems use a different code-base, we provide the same static latency



(a) Covered feeds vs. subscribed feeds (b) Exploring utility formulation (c) Neighbors are used in multiple roles

Fig. 9. The friendship overlay provides high subscription coverage for most nodes. The locality improves with the number of subscribed feeds.



(a) Update latencies (b) Data traffic at Scribe nodes (c) Data traffic at Rappel nodes

Fig. 10. Rappel and Scribe both achieve low absolute update dissemination latency, however, only Rappel imposes traffic load at a node that is proportional to the number of its subscriptions.

matrix to both the simulators to effectively generate one-on-one node mapping. Note that we use the optimization that enforces that the feed's dissemination tree is rooted at the publisher node. We observe the data traffic in both systems[4]. The data traffic gives us an insight regarding the performance of the systems when the publisher disseminates updates at a high rate.

In the first experiment, we compare the dissemination latency of a single update from a publisher to 500 subscribers under two different scenarios. In the first scenario, the network consists of only 501 nodes, whereas in the second scenario, there are a total of 5001 nodes. Stated differently, there are an additional 4500 nodes (90%) that do not subscribe to this publisher. Fig. 10 shows that Rappel achieves low absolute latency, however it does worse than Scribe. One reason is because Rappel leverages (inaccurate) network coordinates instead of explicit pinging to select tree parents. Another reason is that our implementation limits the publisher to $\beta = 5$ children. On the other hand, the data traffic load imposed by Rappel is better balanced than Scribe. For instance, as Scribe is not noiseless, it uses 40 non-participating intermediate nodes to disseminate updates to 464 subscribers in the larger system.

We perform another experiment using a trace of 100 publishers and 5582 multi-feed subscribers. Each publisher disseminates a single update. Fig. 10b shows that Scribe nodes are imposed with highly variable amounts of data traffic. There is no correlation between the traffic imposed on a node and its number of subscribed feeds. On the other hand, Fig. 10c shows that the data traffic at each Rappel node scales with the number of subscriptions it has. Note that most Scribe nodes forward no messages exhibiting a large imbalance in data traffic.

6.5. Overhead due to control bandwidth

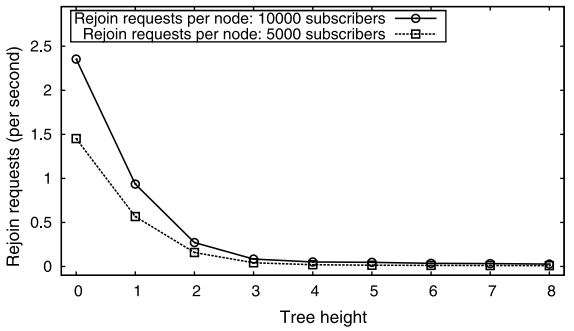
In this section, we show the bandwidth overhead of Rappel due to control operations. Note that the data traffic at a node due to a single update is bounded by β (=5), and the net dissemination traffic depends on the rate at which the publisher posts updates.

We simulate two different settings: systems with 5000 and 10,000 nodes. with each node subscribing to 1 publisher. Fig. 11a shows that the number of periodic rejoin requests received per node at each height-level of the tree decreases exponentially. Further, the number of rejoin requests received by nodes does not increase substantially even with the doubling of subscriber population. This demonstrates scalability.

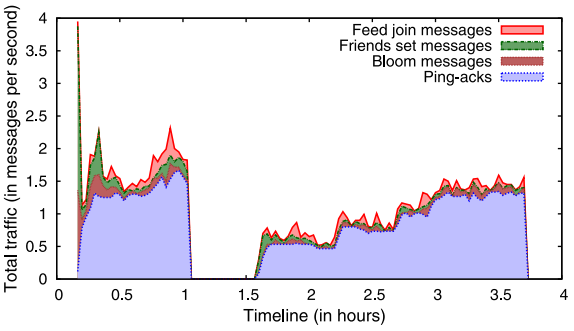
Next, we measure the bandwidth consumption in a system with 5000 subscribers and 1 publisher. The system is injected with churn using Overnet traces [1]. In spite of having only 1 feed, Rappel nodes still maintain the friendship overlay described in Section 3. To measure bandwidth, we count individual messages, i.e., a request and a reply are separate messages.

Fig. 11b shows traffic at a subscriber that ended the simulation with height = 1. A tree height = 1 represents the worst-case load amongst subscriber nodes. Note that

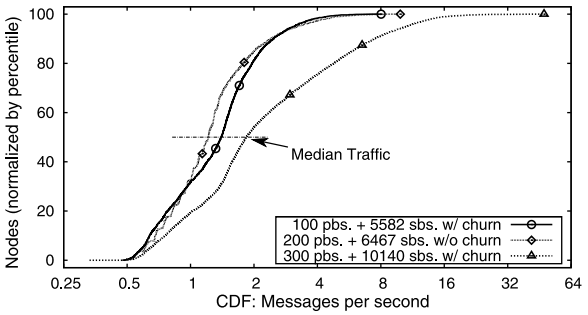
[4] We do not compare control traffic across both systems as it is not obvious how to correctly compare the two.



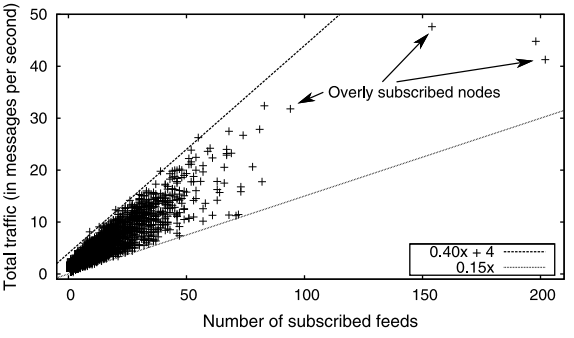
(a) Periodic rejoin traffic per tree height



(b) Traffic at subscriber with height=1 (w/ churn)



(c) Traffic scales well with network size



(d) Traffic load depends on number of subscriptions

Fig. 11. Rappel's bandwidth overhead is low: it scales logarithmically with the number of subscribers at the publisher and linearly with the number of feed subscriptions at a subscriber.

the node does not initially start out with height = 1, i.e., when the system is still bootstrapping. However, it eventually moves up the tree due to its proximity to the publisher. Further, the subscriber is offline from $t = 1$ h to just after $t = 1.5$ h. The data shows a breakdown of traffic by the different types of messages with the top-most line representing total traffic. The subscriber's bandwidth is moderate, staying mostly under 2 messages a second. Since the amortized Rappel message size is 50 bytes[5], the stable traffic load at the subscriber is about 100 Bps. The initial spike in traffic is due to network warm-up, as the nodes initially join at a rate of 10 per minute.

Fig. 11c shows that Rappel's bandwidth usage is not affected drastically by an increase in the numbers of publishers and subscribers. The median bandwidth is less than 2 messages per second in all cases, which translates to approximately 100 Bps. One might notice that the tail-end of the largest network degrades poorly. However, Fig. 11d explains the reasoning behind the degradation. The plot shows fairness of Rappel – traffic is high only at nodes with large number of feed subscriptions. Further, the plot shows that Rappel nodes entail an additional control bandwidth overhead of between 0.15 and 0.4 messages a second (up to 20 Bps) per extra subscription.

7. Related work

We discuss related work in the areas of multicast, publish-subscribe, locality and distributed membership.

7.1. Application-level multicast

Tree-based notification systems relying on structured p2p overlay networks include Scribe [8] and Splitstream [7]. These approaches leverage the underlying Pastry DHT [37], achieving low latency and stretch. Incrementally modifying either Scribe or Splitstream to ensure zero noise significantly disrupts network proximity properties provided by the DHT and is therefore not desirable. Multicast trees such as Narada, SRM, RMTP, etc. [47] focus their attention on network proximity at the expense of interest locality.

Gossip-based application-level multicast systems such as Bimodal Multicast [3], Lpbcast [13], and BAR Gossip [27] achieve good reliability at the expense of increased bandwidth, although the latter can be lowered by considering network proximity [17,30]. However, the involvement of non-interested subscribers in the dissemination leads to noise. Overlapping-group multicast has been addressed in traditional group-communication systems (see [4]), as well as gossip-based systems, e.g., [21], but without looking at interest locality.

7.2. Publish-subscribe

Generic content-based pub-sub systems such as Gryphon [2] or Sienna [6] rely on a backbone of brokers. While these systems are able to support expressive subscriptions and some achieve zero noise at subscribers, the brokers

[5] All Rappel messages except a Bloom filter reply and an ACK from "deep" parents (due to piggybacked ancestry chain) are much smaller. This is a pessimistic estimate.

may be subjected to significant noise even when no interested subscribers are connected to them.

Net-X [35] is a proposed system that uses polynomial signatures to discover interest locality among user interests and data. Sub-2-Sub [45] is a collaborative content-based p2p publish-subscribe system that, like Rappel, exploits interest locality but does not address network locality and may incur high stretch factors. Alternatively, the authors of [40] propose to build content-based filtering atop Scribe [8]. Their approach is to use automatic schema detection to map content-based subscriptions onto a set of topics. However, this approach suffers from false positives. Another approach for supporting content-based publish-subscribe atop structured peer-to-peer networks [31] is based on the division of a content-based publication space into recursively split publication domains. Due to the underlying DHT substrate, nodes are often in charge of operations for publications that they do not even subscribe to.

Using the inherent interest correlation between users' interest to build efficient dissemination systems was previously used by Chand et al. [10] for creating unstructured content-based publish and subscribe networks. The approach is to link peers with similar interests, according to some proximity function. The constructed overlay allows probabilistic broadcast within some semantic interest group. This broadcast may fail if the event semantic domain does not correspond to a linked set of peers in the overlay. This restricts the system usage to popular content and coarse filtering. More, the system incur a significant noise ratio as nodes that lie on the boundaries of some semantic domain receive unexpected content, and within its boundaries nodes can receive an item multiple times.

The SpiderCast topic-based publish and subscribe system [11] uses interest correlation to form sets of connected random graphs for each topic, with the primary goal of aggregating links for multiple such graphs between peers by leveraging the interest proximity of peers (i.e., to reduce nodes' degrees). The authors however do not present how to create and maintain the dissemination structures. While built using a similar idea to Rappel, SpiderCast does not take into account physical proximity, but ensures noiselessness and fairness. Moreover, the system relies on each node knowing either the entire network or a large portion of it, raising scalability issues.

7.3. Decentralized RSS dissemination

A few recent systems have been specifically targeting RSS feeds such as Corona [34], Cobra [36], FeedTree [38] and LagOver [31]. Aggregators such as Corona reduce the load on publishers by proxying on behalf of the subscribers. As such, the publisher load simply shifts to the aggregator (albeit, nodes only issue a single `POLL` request for all their subscriptions). However, the latency of update dissemination depends on the polling frequency. Our rapid dissemination goals are somewhat in common with that of cooperative polling approaches taken by Cobra. However, these proxy systems rely on intermediate infrastructures (third parties) and thus are not completely p2p in nature. FeedTree and LagOver are the only other true p2p system for RSS dissemination. However, as FeedTree is based on Scribe [8], it is not noiseless. Like Rappel, one of the goals of LagOver is soft real-time dissemination of updates. However, LagOver does not leverage the correlation between feed subscriptions, requiring nodes to contact the publisher directly to join a feed.

7.4. Exploiting locality

Temporal and spatial locality of data access by processes has been a motivation for designing caches in OSes [41]. Locality of web access at each user is exploited by local caches, and correlations in interest on web content has led to the rise of cooperative web caches, e.g., [19]. Ideas from social networks have been used to improve the performance of p2p systems, e.g., [29,32]. Correlation in user interest has also been used to improve performance of p2p resource discovery systems [18], as well as for content delivery [46].

7.5. Distributed membership protocols

Rappel's friends set maintenance is motivated by membership discovery protocols. This includes SCAMP [16], Cyclon [44] and T-MAN [20], which construct overlay graphs either randomly or according to a distance function. Finally, Rappel's use of a friends set and a candidates set bear some similarities to the use of the inner and outer rings in the LOCKSS system [28], which however did not discover interest and network locality.

8. Conclusion

This paper showed that interest locality and network proximity can both be leveraged to build a collaborative p2p dissemination system for feed-based publish-subscribe systems. The proposed Rappel system discovers locality using probabilistic techniques such as gossip-like voting and auditing. By periodically auditing the friends set, Rappel stays converged to a highly effective friends set. These friends are used to join dissemination trees for subscribed feeds.

We studied the benefits of Rappel under both: (i) a PlanetLab deployment, as well as (ii) simulation driven by traces of multi-feed user subscriptions, Internet round-trip-times, and churn from a p2p system. Rappel disseminates updates with zero noise within fractions of a second in PlanetLab and within a few seconds in simulation with thousands of nodes. Because of its unstructured nature, Rappel is highly resilient to churn. Nodes spend a median bandwidth of around 100 Bps with the traffic overhead being heavily correlated to the number of subscriptions.

References

[1] R. Bhagwan, S. Savage, G.M. Voelker, Understanding availability, in: Proceedings of IPTPS, 2003.
[2] S. Bhola, Y. Zhao, J. Auerbach, Scalably supporting durable subscriptions in a publish/subscribe system, in: Proceedings of DSN, 2003.

[3] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky, Bimodal multicast, ACM Trans. Comp. Syst. 17 (2) (1999) 41–88.
[4] K.P. Birman, Building Secure and Reliable Network Applications, Manning Publications and Prentice Hall, 1996.
[5] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communication of the ACM 13 (7) (1970).
[6] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and evaluation of a wide-area event notification service, ACM Trans. Comp. Syst. 19 (3) (2001) 332–383.
[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, Splitstream: high-bandwidth content distribution in a cooperative environment, in: Proceedings of IPTPS, 2003.
[8] M. Castro, P. Druschel, A.-M. Kermarrec, A. Rowstron, SCRIBE: a large-scale and decentralized application-level multicast infrastructure, IEEE JSAC 20 (8) (2002) 1489–1499.
[9] M. Castro, M.B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, A. Wolman, An evaluation of scalable application-level multicast built using peer-to-peer overlays, in: Proceedings of INFOCOM, 2003.
[10] R. Chand, P. Felber, Semantic peer-to-peer overlays for publish/subscribe networks, in: Proceedings of EuroPar, 2005, pp. 1194–1204.
[11] G. Chockler, R. Melamed, Y. Tock, R. Vitenberg, Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication, in: Proceedings of DEBS, 2007, pp. 14–25.
[12] F. Dabek, R. Cox, F. Kaashoek, R. Morris, Vivaldi: a decentralized network coordinate system, in: Proceedings of SIGCOMM, 2004.
[13] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, P. Kouznetsov, Lightweight probabilistic broadcast, ACM Trans. Comp. Syst. 21 (4) (2003) 341–374.
[14] P. Fraigniaud, P. Gauron, M. Latapy, Combining the use of clustering and scale-free nature of user exchanges into a simple and efficient p2p system., in: Proceedings of EuroPar, 2005.
[15] Freepastry. <http://freepastry.org/FreePastry/>.
[16] A. Ganesh, A.-M. Kermarrec, L. Massoulié, Peer-to-peer membership management for gossip-based protocols, IEEE Trans. Comp. 2 (52) (2003) 139–149.
[17] I. Gupta, A.-M. Kermarrec, A.J. Ganesh, Efficient epidemic-style protocols for reliable and scalable multicast, IEEE Trans. Par. Dist. Syst. 17 (7) (2006) 593–605.
[18] S. Handurukande, A.-M. Kermarrec, F.L. Fessant, L. Massouli, S. Patarin, Peer sharing behaviour in the edonkey network and its implications for the design of serverless file sharing systems, in: Proceedings of EuroSys, 2006.
[19] S. Iyer, A. Rowstron, P. Druschel, Squirrel: a decentralized, peer-to-peer web cache, in: Proceedings of PODC, 2002.
[20] M. Jelasity, O. Babaoglu, T-Man: gossip-based overlay toplogy management, in: Self-Organising Systems, LNCS, vol. 3910, 2005, pp. 1–15.
[21] K. Jenkins, K. Hopkinson, K.P. Birman, A gossip protocol for subgroup multicast, in: Proceedings of ICDCS Workshops, 2001.
[22] D. Kostic, A. Rodriguez, J. Albrecht, A. Vahdat, Bullet: high bandwidth data dissemination using an overlay mesh, in: Proceedings of SOSP, 2003.
[23] J. Ledlie, P. Pietzuch, M. Seltzer, Stable and accurate network coordinates, in: Proceedings of ICDCS, 2006.
[24] X. Liao, H. Jin, Y. Liu, L.M. Ni, D. Deng, Anysee: peer-to-peer live streaming, in: Proceedings of INFOCOM, 2006.
[25] H. Liu, V. Ramasubramanian, E.G. Sirer, Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews, in: Proceedings of IMC, 2005.
[26] Livejournal. <http://www.livejournal.com/>.
[27] H. Li, A. Clement, E. Wong, J. Napper, L. Alvisi, M. Dahlin, Bar gossip, in: Proceedings of OSDI, 2006.
[28] P. Maniatis, M. Roussopoulos, T.J. Giuli, D.S.H. Rosenthal, M. Baker, The LOCKSS peer-to-peer digital preservation system, ACM Trans. Comp. Syst. 23 (1) (2005) 2–50.
[29] S. Marti, P. Ganesan, H. Garcia-Molina, SPROUT: p2p routing with social networks, in: EDBT Workshops, 2004.
[30] L. Massoulié, A.-M. Kermarrec, A. Ganesh, Network awareness and failure resilience in self-organizing overlay networks, in: Proceedings of SRDS, 2003.
[31] V. Muthusamy, H.-A. Jacobsen, Infrastructure-less content-based publish/subscribe, Tech. rep., Middleware Systems Research Group, University of Toronto, March 2007.
[32] J.A. Patel, I. Gupta, N. Contractor, JetStream: achieving predictable gossip dissemination by leveraging social network principles, in: Proceedings of IEEE NCA, 2006.
[33] Planetlab. <http://www.planet-lab.org/>.
[34] V. Ramasubramanian, R. Peterson, E.G. Sirer, Corona: a high performance publish-subscribe system for the world wide web, in: Proceedings of NSDI, 2006.
[35] P. Rao, J. Cappos, V. Khare, B. Moon, B. Zhang, Net-x: unified data-centric internet services, in: Proceedings of NetDB, 2007.
[36] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Welsh., Cobra: content based filtering and aggregation of blogs and RSS feeds, in: Proceedings of NSDI, 2007.
[37] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: Proceedings of Middleware, 2001.
[38] D. Sandler, A. Mislove, A. Post, P. Druschel, Feedtree: sharing web micronews with peer-to-peer event notification, in: Proceedings of IPTPS, 2005.
[39] S. Saroiu, P.K. Gummadi, S.D. Gribble, A measurement study of peer-to-peer file sharing systems, in: Proceedings of MMCN, 2002.
[40] D. Tam, R. Azimi, H.-A. Jacobsen, Building content-based publish/subscribe systems with distributed hash tables, in: 1st International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P), 2003.
[41] A. Tanenbaum, Modern Operating Systems, second ed., Prentice Hall, 2001.
[42] P.-N. Tan, M. Steinbach, V. Kumar, Introduction to Data Mining, first ed., Addison-Wesley Longman Publishing Co., Inc., 2005.
[43] Twitter. <http://www.twitter.com/>.
[44] S. Voulgaris, D. Gavidia, M. van Steen, CYCLON: inexpensive membership management for unstructured P2P overlays, J. Net. Syst. Mmgt. 13 (2) (2005) 197–217.
[45] S. Voulgaris, E. Rivière, A.-M. Kermarrec, M. van Steen, Sub-2-sub: self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks, in: Proceedings of IPTPS, 2006.
[46] T. Wu, S. Ahuja, S. Dixit, Efficient mobile content delivery by exploiting user interest correlation, in: Proceedings of WWW (Poster), 2003.
[47] C.K. Yeo, B.S. Leea, M.H. Er, A survey of application level multicast techniques, Comp. Commun. 27 (15) (2004) 1547–1568.
[48] B. Zhang, R. Liu, D. Massey, L. Zhang, Collecting the internet AS-level topology, SIGCOMM Comput. Commun. Rev. 35 (1) (2005).

Jay A. Patel is a PhD student at the University of Illinois at Urbana-Champaign (UIUC). He previously received a BSCS from The University of Texas - Pan American (UTPA). His primary research interests include designing distributed systems inspired by the properties of social networks, with secondary interests in areas of wireless networks and operating systems. Jay is a student member of both the ACM and the IEEE. On separate occasions, he has served as both the Chair and Treasurer of the UTPA student chapter of the ACM.

Étienne Rivière is an ERCIM research fellow at NTNU Trondheim in Norway. He received his PhD in Computer Science from the University of Rennes, France in November 2007. Before coming to NTNU, Etienne was with University of Neuchâtel, Switzerland. His research interests lie in the design, analysis, implementation and evaluation of large-scale distributed systems. He is more specifically interested in search mechanisms, publish-subscribe, content dissemination networks, decentralized management, and self-organizing systems. He also has a strong interest in epidemics- and gossip-based protocols. He is a member of the ACM and the IEEE.

Indranil Gupta leads the Distributed Protocols Research Group in the CS Department at UIUC. He is interested in research on distributed protocols, large-scale distributed systems, monitoring and management for distributed systems, and sensor networks. Indranil is recipient of the NSF CAREER award in 2005.

Anne-Marie Kermarrec Anne-Marie Kermarrec is a Senior Researcher at INRIA (France) since 2004, where she is leading the ASAP (As Scalable As Possible) research group, focusing on large-scale dynamic distributed systems. Her research interests are peer-to-peer networks, large-scale information management and epidemic protocols. Before that, Anne-Marie was with Microsoft Research in Cambridge (UK). She obtained her Ph.D. from the University of Rennes (France) in October 1996. Anne-Marie has been awarded a European Research Council Starting Grant in 2008 for her five year GOSSPLE project.