Epidemic Broadcast Trees *

João Leitão
University of Lisbon
jleitao@lasige.di.fc.ul.pt

José Pereira
University of Minho
jop@di.uminho.pt

Luís Rodrigues
University of Lisbon
ler@di.fc.ul.pt

Abstract

There is an inherent trade-off between epidemic and deterministic tree-based broadcast primitives. Tree-based approaches have a small message complexity in steady-state but are very fragile in the presence of faults. Gossip, or epidemic, protocols have a higher message complexity but also offer much higher resilience.

This paper proposes an integrated broadcast scheme that combines both approaches. We use a low cost scheme to build and maintain broadcast trees embedded on a gossip-based overlay. The protocol sends the message payload preferably via tree branches but uses the remaining links of the gossip overlay for fast recovery and expedite tree healing. Experimental evaluation presented in the paper shows that our new strategy has a low overhead and that is able to support large number of faults while maintaining a high reliability.

1. Introduction

Many systems require highly scalable and reliable broadcast primitives. These primitives aim at ensuring that all correct participants receive all broadcast messages, even in the presence of network omissions or node failures. Gossip protocols [8] have emerged as a highly scalable and resilient approach to implement reliable broadcast. Unfortunately, in steady-state, gossip protocols exhibit an excessive message overhead in order to ensure reliability with high probability. On the other hand, tree-based broadcast primitives have a small message complexity in steady-state, but they are very fragile in the presence of failures, lacking the natural resilience of epidemic protocols.

*This work was partially supported by project "P-SON: Probabilistically Structured Overlay Networks" (POS_C/EIA/60941/2004).

Bimodal multicast [2] was one of the first pioneer works to combine tree-based and gossip based primitives. The approach works as follows: in a first phase, the broadcast messages is disseminated using the unreliable IP-Multicast [7] primitive; in a second phase, participants engage in gossip exchanges in order to mask omissions that may occur during the first phase. This approach has two major drawbacks. One is that it depends on the availability of IP-multicast, which is not widely deployed in large-scale [10]. To overcome this limitation one could envision to replace IP-multicast by some application-level multicast protocol. Still, a second drawback persists: the approach requires the use of two distinct protocols and therefore, it may present undesired complexity. Due to these drawbacks, more recent work has favored the use of pure gossip approaches [11, 23].

In this paper, we propose a novel approach to combine tree-based and gossip protocols in order to achieve both low message complexity and high reliability. Contrary to previous work, our approach creates a broadcast tree embedded in a gossip-based overlay. The broadcast tree in created and maintained using a low cost algorithm, described in the paper. Broadcast is achieved mainly by using push gossip on the tree branches. However, the remaining links of the gossip-based overlay are also used to propagate the message using a lazy-push approach. The lazy-push steps ensure high reliability (by guaranteeing that, in face of failures, the protocol falls back to a pure gossip approach) and, additionally, also provide a way to quickly heal the broadcast tree. We have named our protocol push-lazy-push multicast tree, or simply *Plumtree*. Plumtree has low overhead, given that it only requires a gossip-based random overlay for its complete operation, avoiding dependencies from other broadcast schemes (such as IP-based or application level multicast). Simulation results show that Plumtree is scalable and able to quickly recover from failures as large as 80% of total nodes of the system.

The rest of this paper is organized as follows. In section 2 we present a brief survey on pure gossip protocols which are the basis for our protocol. Section 3 introduces Plumtree, explaining the rationale behind its design and describing its algorithms in detail. Section 4 presents experimental results obtained through simulation and discuss these results in some depth. Plumtree is compared with the related work in Section 5 and, finally, Section 6 presents future work and concludes the paper.

2. Gossip Protocols

2.1. Rationale

The basic idea inspiring gossip protocols consists in having all nodes contribute with an equal share to the message dissemination. To reach this purpose, when a node wishes to broadcast a message, it selects t nodes at

random to whom it sends the message (t is a typical configuration parameter called *fanout*). Upon receiving a message for the first time, a node repeats this process (selecting t gossip targets and forwarding the message to them). If a node receives the same message twice - which is possible, as each node selects its gossip targets in an independent way (without being aware of gossip targets selected by other nodes) - it simply discards the message. This assumes that each node keeps track of which messages it has already seen and delivered. The problem of purging message histories is out of the scope of this paper and has been addressed previously [14]. The simple operation model of gossip protocols not only provides high scalability but also, a high level of fault tolerance, as its intrinsic redundancy is able to mask network omissions and also node failures.

In order to operate exactly as described above, gossip protocols require each node to maintain information concerning the entire system membership, in order to select the target nodes in each gossip step. Clearly, such solution is not scalable, not only due to the large number of nodes that may constitute the view, but also due to the cost of maintaining the complete membership up-to-date. To overcome this problem, several gossip protocols rely on *partial views* instead of the complete membership information [20, 9, 11, 23]. A partial view is a small subset of the entire system membership that nodes use when selecting gossip peers. These partial views establish *neighboring* associations between nodes, which can be described as a gossip-based overlay network.

2.2. Gossip Strategies

The following strategies can be used to implement a gossip protocol:

Eager push approach: Nodes send the message payload to random selected peers as soon as they receive it for the first time.

Pull approach: Periodically, nodes query random selected peers for information about recently received messages. If the nodes have new information, they forward it to the querying node. This is a strategy that works best when combined with some unreliable broadcast mechanism (*i.e.* IP Multicast [7]).

Lazy push approach: When a node receives a message for the first time, it sends the message identifier (but not the payload) to random selected peers. If the peers have not received the message, they make an explicit pull request.

There is also a trade-off between push and pull strategies. Eager push strategies produce more redundant traffic but they also achieve lower latency than lazy push or pull strategies, as they require at least an extra round trip

time to produce a delivery. Naturally, these strategies can be combined in different hybrid approaches. Protocols that combine push and pull approaches have been described in [1, 19]. A protocol that combines eager push and lazy push is described in [3]. As its name implies, Plumtree also combines eager and lazy push.

2.3. Metrics

In this section we define three metrics to evaluate a gossip protocol.

Reliability Gossip reliability is defined as the percentage of active nodes that deliver a gossip broadcast. A reliability of 100% means that the protocol was able to deliver a given message to all active nodes or, in other words, that the message resulted in an atomic broadcast as defined in [13].

Relative Message Redundancy (RMR) This metric measures the messages overhead in a gossip protocol. It is defined as:

$$\left(\frac{m}{n-1}\right) - 1$$

where m is the total number of payload messages exchanged during the broadcast procedure and n is the total number of nodes that received that broadcast. This metric is only applicable when at least 2 nodes receive the message.

A RMR value of zero means that there is exactly one payload message exchange for each node in the system, which is clearly the optimal value. By opposition, high values of RMR are indicative of a broadcast strategy that promotes a poor network usage. Note that it is possible to achieve a very low RMR by failing to be reliable. Thus the aim is to combine low RMR values with high reliability. Furthermore, RMR values are only comparable for protocols that exhibit similar reliability. Finally, note that in pure gossip approaches, RMR is closely related with the protocol fanout, as it tends to $\mathit{fanout} - 1$.

Control messages are not considered by this metric, as they are typically much smaller than payload messages hence, they are not the main source of contribution to the exhaustion of network resources. Moreover, these messages can be sent using piggyback strategies providing a better usage of the network.

Last Delivery Hop (LDH) The Last Delivery Hop measures the number of hops required to deliver a broadcast message to all recipients. When the message is gossip for the first time, its hop count is set to 1 and, each time it is relayed, the hop count is increased. The last delivery hop is the hop count of the last message that is delivered

by a gossip protocol or, in other words, is the maximum number of hops that a message must be forwarded in the overlay before it is delivered. This metric is closely related with the diameter of the gossip overlay used to disseminate messages, and it also gives some insight on the latency of a gossip protocol. Note that, if links would exhibit a constant delay, the latency of the gossip protocol would be the LDH multiplied by this constant.

3. Epidemic Broadcast Trees

3.1. Architecture

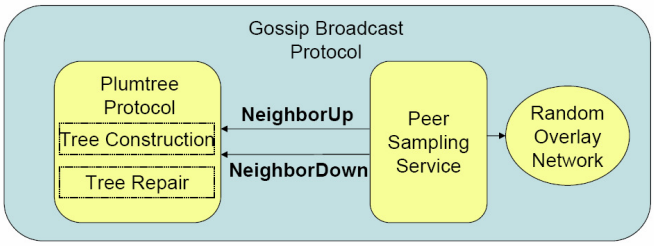


Figure 1. Plumtree architecture

Figure 1 depicts the Plumtree architecture and the interface used by its components to exchange information. The *Random Overlay Network* is the component that maintains a partial view of the system. This is achieved by a *Peer Sampling Service* [12], which is implemented through a gossip based membership service that relies in partial views [15]. The *Plumtree Protocol* is the component that materializes our gossip scheme; it has the following two main functions:

Tree construction This component is in charge of selecting which links of the random overlay network will be used to forward the message payload using an eager push strategy. We aim at a tree construction mechanisms that is as simple as possible, with minimal overhead in terms of control messages.

Tree repair This component is in charge of repairing the tree when failures occur. The process should ensure that, despite failures, all nodes remain covered by the spanning tree. therefore, it should be able to detect and heal partitions of the tree. The overhead imposed by this operation should also be as low as possible.

We assume that the peer sampling service exports a *GetPeers()* primitive in its interface, as suggested in [12], that is used by the gossip broadcast protocol to get information about neighbors to whom it should send messages. In addition, the Plumtree protocol exports the following two primitives: *NeighborUp()* and *NeighborDown()*.

These primitives are used to notify the gossip protocol whenever a change happens on the partial view maintained by the peer sampling service. These primitives are used to support quick healing of the broadcast tree.

3.2. Overview

The main design principles of Plumtree are the following. The protocol operates as any pure gossip protocol, in the sense that, in order to broadcast a message, each node gossips with f nodes provided by a peer sampling service (where f is the protocol fanout). However, each node uses a combination of eager push and lazy push gossip. Eager push is used just for a subset of the f nodes, while lazy push is used for the remaining nodes. The links used for eager push are selected in such a way that their closure effectively builds a broadcast tree embedded in the random overlay network. Lazy push links are used to ensure gossip reliability when nodes fail and also to quickly heal the broadcast tree. Furthermore, contrary to other gossip protocols, the set of (random) peers is not changed at each gossip round. Instead, the same peers are used until failures are detected. Given that the peer connections used to support gossip exchanges are stable, we use TCP connections to support the message exchange, as they offer extra reliability and an additional source of failure detection.

3.3. Peer Sampling Service and Initialization

Plumtree depends on a random overlay network which is maintained by a peer sampling service. The overlay network should present some essential properties that must be ensured by the peer sampling service. We now describe those properties.

Connectivity: The overlay should be connected, despite failures that might occur. This has two implications. Firstly, all nodes should have in their partial views, at least, another correct node. Secondly, all nodes should be in the partial view of, at least, a correct node.

Scalable: Our protocol is aimed toward the support of large distributed applications. Therefore, the peer sampling service should be able to operate correctly in such large systems (*e.g.* with more than 10.000 nodes).

Reactive membership: The stability of the spanning tree structure depends on the stability of the partial views maintained by the peer sampling service. When a node is added or removed to the partial view of a given node, it might produce changes in the links used for the spanning tree. These changes may not be desirable. Hence, the peer sampling service should employ a reactive strategy that maintains the same elements in partial views when operating in steady-state (*e.g.* [11, 16]).

Algorithm 1: Spanning Tree Construction Algorithm

```
1  procedure dispatch do
2      announcements ⟵ policy (lazyQueue) //set of IHAVE messages
3      trigger Send(announcements)
4      lazyQueue ⟵ lazyQueue \announcements

5  procedure EagerPush (m, mID, round, sender) do
6      foreach p ∈ eagerPushPeers: p ≠sender do
7          trigger Send(GOSSIP, p, m, mID, round, myself)

8  procedure LazyPush (m, mID, round, sender) do
9      foreach p ∈ lazyPushPeers: p ≠sender do
10         lazyQueue ⟵ (textscIhave(p, m, mID, round, myself)
11     call dispatch()

12 upon event Init do
13     eagerPushPeers ⟵ getPeers(f)
14     lazyPushPeerss ⟵ ∅
15     lazyQueues ⟵ ∅
16     missing ⟵ ∅
17     receivedMsgs ⟵ ∅

18 upon event Broadcast(m) do
19     mID ⟵ hash(m+myself)
20     call EagerPush (m, mID, 0, myself)
21     call lazyPush (m, mID, 0, myself)
22     trigger Deliver(m)
23     receivedMsgs ⟵ receivedMsgs ∪ {mID}

24 upon event Receive(GOSSIP, m, mID, round, sender) do
25     if mID ∉ receivedMsgs then
26         trigger Deliver(m)
27         receivedMsgs ⟵ receivedMsgs ∪ {mID}
28         if ∃ (id,node,r) ∈ missing :id=mID then
29             cancel Timer(mID)
30         call EagerPush (m, mID, round+1, myself)
31         call lazyPush (m, mID, round+1, myself)
32         eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
33         lazyPushPeers ⟵ lazyPushPeers \ {sender}
34         call Optimize (m, mID, round, sender) // optional
35     else
36         eagerPushPeers ⟵ eagerPushPeers \ {sender}
37         lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
38         trigger Send(PRUNE, sender, myself)

39 upon event Receive(PRUNE, sender) do
40         eagerPushPeers ⟵ eagerPushPeers \ {sender}
41         lazyPushPeers ⟵ lazyPushPeers ∪ {sender}
```

In addition to these properties, that are fundamental to the correct operation of Plumtree the peer sampling service may also exhibit a set of other desirable properties, in the sense that they improve the operation of the protocol. One such property is Symmetric partial views. If the links that form the spanning tree are symmetric, then the tree may be shared by multiple sources. Symmetric partial views render the task of creating bi-directional trees easier, and reduce the amount of peers that each node has to maintain.

For the correct operation of the algorithm, it maintains two sets of peers: the *eagerPushPeers*, with which the node uses eager push gossip and *lazyPushPeers*, with which it uses lazy push gossip. Initially, *eagerPushPeers* contains f random peers, that are obtained through the peer sampling service, and *lazyPushPeers* is empty (Algorithm 1, lines: 12–17). Therefore, in the first rounds, the protocol operates as a pure push gossip protocol. The fanout value f must be selected such that the overlay defined by the *eagerPushPeers* of all nodes is connected and covers all nodes. This task is easier when a peer sampling service such as HyParView [16] is used.

3.4. Gossip and Tree Construction

After the initialization of the *eagerPushPeers* set described above, nodes construct the spanning tree by moving neighbors from *eagerPushPeers* to *lazyPushPeers*, in such a way that, after the protocol evolves, the overlay defined by the first set becomes a tree. When a node receives a message from the first time it includes the sender in the set of *eagerPushPeers* (Algorithm 1, lines: 24–33). This ensures that the link from the sender to the node is

bidirectional and belongs to the broadcast tree. When a duplicate is received, its sender is moved to the *lazyPushPeers* (Algorithm 1, lines: 34–37). Furthermore, a PRUNE message is sent to that sender such that, in response, it also moves the link to the *lazyPushPeers* (Algorithm 1, lines: 38–40). This procedure ensures that, when the first broadcast is terminated, a tree has been created.

One interesting aspect of this process is that, assuming a stable network (*i.e.* with constant load), it will tend to generate a spanning tree that minimizes the message latency (as it only keeps the path that generates the first message reception at each node).

As soon as nodes are added to the *lazyPushPeers* set, messages start being propagated using both eager and lazy push. Lazy push is implemented by sending IHAVE messages, that only contain the broadcast ID, to all *lazyPushPeers* (Algorithm 1, lines: 5–7). Note however that, to reduce the amount of control traffic, IHAVE messages do not need to be sent immediately. A scheduling policy is used to piggyback multiple IHAVE announcements in a single control message. The only requirement for the scheduling policy for IHAVE messages is that every IHAVE message is eventually scheduled for transmission.

3.5. Fault Tolerance and Tree Repair

When a failure occurs, at least one tree branch is affected. Therefore, eager push is not enough to ensure message delivered in face of failures. The lazy push messages exchanged through the remaining nodes of the gossip overlay are used both to recover missing messages but also to provide a quick mechanisms to heal the multicast tree.

When a node receives a IHAVE message, it simply marks the corresponding message as missing (Algorithm 2, lines: 1–15). It then starts a timer, with a predefined timeout value, and waits for the missing message to be received via eager push before the timer expires. The timeout value is a protocol parameter that should be configured considering the diameter of the overlay and a target maximum recovery latency, defined by the application requirements. This is a parameter that should be statically configured at deployment time.

When the timer expires at a given node, that node selects the first IHAVE announcement it has received for the missing message. It then sends a GRAFT message to the source of that IHAVE announcement (Algorithm 2, lines: 6–11). The GRAFT message has a dual purpose. In first place, it triggers the transmission of the missing message payload. In second place, it adds the corresponding link to the broadcast tree, healing it (Algorithm 2, lines: 12–16). The reader should notice that when a GRAFT message is sent, another timer is started to expire after a certain timeout, to ensure that the message will be requested to another neighbor if it is not received

Algorithm 2: Spanning Tree Repair Algorithm

```
1  upon event Receive(IHAVE, mID, round, sender) do
2      if mID ∉ receivedMsgs do
3          if ∄ Timer(id): id=mID do
4              setup Timer(mID, timeout_1)
5          missing ⟵ missing ∪ {(mID,sender,round)}

6  upon event Timer(mID) do
7      setup Timer(mID, timeout_2)
8      (mID,node,round) ⟵ removeFirstAnnouncement(missing, mID)
9      eagerPushPeers ⟵ eagerPushPeers ∪ {node}
10     lazyPushPeers ⟵ lazyPushPeers \ {node}
11     trigger Send(GRAFT,node,mID,round,myself)

12 upon event Receive(GRAFT, mID, round, sender) do
13     eagerPushPeers ⟵ eagerPushPeers ∪ {sender}
14     lazyPushPeers ⟵ lazyPushPeers \ {sender}
15     if mID ∈ receivedMsgs do
16         trigger Send(GOSSIP, sender, m, mID, round, myself)
```

meanwhile. This second timeout value should be smaller that the first, in the order of an average round trip time to a neighbor.

Note that several nodes may become disconnected due to a single failure, hence it is possible that several nodes will try to heal the spanning tree degenerating into a structure that has cycles. This is not a problem however, as the natural process to build the tree will remove any redundant branches produced during this process by sending PRUNE messages (*i.e.*, when a message is received by a node more than once).

3.6. Dynamic Membership

We now describe how Plumtree reacts to changes in the gossip overlay. These changes are notified by the peer sampling service using the *NeighborDown* and *NeighborUp* primitives. When a neighbor is detected to leave the overlay, it is simple removed from the membership. Furthermore, the record of IHAVE messages sent from failed members is deleted from the missing history (Algorithm 3, lines: 1–5). When a new member is detected, it is simply added to the set of eagerPushPeers, i.e., it is considered as a candidate to become part of the tree (Algorithm 3, lines: 6–7).

An interesting aspect of the repair process is that, when "sub-trees" are generated, due to changes on the global membership, it is only required that one of the disconnected nodes receive an IHAVE message, to reconnect all those nodes to the root node (repairing the whole spanning tree). This is enough to heal the spanning tree as long as only a reduced number of nodes fail, generating disconnected "sub-trees". When larger numbers of nodes fail it is more probable to have single nodes isolated from the tree. In such scenarios the time required to repair the tree might be too large. To speedup the healing process, we take benefit of the healing properties of the peer sampling service. As soon has the peer sampling service integrates a disconnected node in the partial view of another member, it generates a *NeighborUp* notification. This notification immediately puts back the disconnected member in the broadcast tree.

Algorithm 3: Overlay Network Change Handlers

```
1  upon event NeighborDown(node) do
2      eagerPushPeers ⟵ eagerPushPeers \ {node}
3      lazyPushPeers ⟵ lazyPushPeers \ {node}
4      foreach (i,n,r) ∈missing:n=node do
5          missing ⟵ missing \ {(i,n,r)}
6  upon event NeighborUp(node) do
7      eagerPushPeers ⟵ eagerPushPeers ∪ {node}
```

3.7. Sender-Based vs Shared Trees

The tree built by Plumtree is optimized for a specific sender: the source of the first broadcast that is used to move nodes from the eagerPushPeers set to the lazyPushPeers set. In a network with multiple senders, Plumtree can be used in two distinct manners.

- For optimal latency, a distinct instance of Plumtree may be used for each different sender. This however, requires an instance of the Plumtree state to be maintained for each sender-based tree, with the associated memory and signaling overhead.
- Alternatively, a single shared Plumtree may be used for multiple senders. Clearly, the LDH value may be sub-optimal for all senders except the one whose original broadcast created the tree. On the other hand, a single instance of the Plumtree protocols needs to be executed.

Later, in the evaluation section, we will depict results of the Plumtree performance for a single sender and for multiple senders using a shared tree, such that the reader can assess the trade-offs involved.

3.8. Optimization

The spanning tree produced by our algorithm is mainly defined by the path followed by the first broadcast message exchanged in the system. Therefore, the protocol does not take advantage of eventual new, and best, paths that can appear in the overlay, as a result of the addition of new nodes/links. Moreover, the repair process is influenced by the policy used to scheduled IHAVE messages. This two factors may have a negative impact in the LDH value exhibit by the algorithm as the system evolves.

To overcome this limitation, we propose here an optimization to the Plumtree algorithm. The rationale for this optimization is as follows. If the tree is optimized, the hop count of messages received by eager push should be smaller or equal to the hop count of the announcements for that message received by lazy push. If this is not the case, this suggest that the tree may be optimized, by replacing the non-optimal eager link by the (better) lazy link.

Algorithm 4: Optimization

```
1  procedure Optimization(mID, round, sender) do
2      if ∃ (id,node,r) ∈ missing: id=mID then
3          if r < round ∧ round−r >= threshold then
4              trigger Send(GRAFT, node, null, r, myself)
5              trigger Send(PRUNE, sender, myself)
```

To promote the stability in the tree, this optimization is only performed if the difference in the hop count is greater than a certain *threshold* value. The *threshold* value will affect the overall stability of the spanning tree. The lower the value, more easily the protocol will try to optimize the tree by exchanging the links that belong to the tree. The reader should notice that the *threshold* value may be lower when the protocol operates in single sender mode, as the distance to the source node to each receiver is relatively constant. On the other hand, with multiple senders, the value should be higher, and close to the diameter of the overlay to avoid constant changes in the links. Notice that, with multiple senders, the distance between the source and receivers change with each broadcast message.

4. Evaluation

We evaluated our broadcast schema using extensive simulations. We conducted all simulation in the PeerSim simulator [18] using its cycle based simulation engine. To that end, we implemented the Plumtree protocol and its optimization for this simulator. In order to obtain comparative figures we used a simple push gossip strategy that presents the particularity of using all links in the same random overlay used to embed the tree. We named this protocol *eager* and we presented the rational for it in [16].

All simulations started with a *join* period in which all nodes join the overlay by using the join mechanism of the peer sampling protocol. All simulations executed for 250 simulation cycles, where the first 50 where used to ensure stabilization of the protocols. All cycles begin with a *failure step* where, in pre-defined cycles of the simulation, nodes are marked as failed, which is followed by a *broadcast step* where one node is selected to send a broadcast message. The broadcast step is only considered as terminated when no more messages are traveling in the overlay. Then a membership step is performed, where the peer sampling protocol executes its periodic operation and detects failed nodes.

As stated in Section 3.7, we experimental the Plumtree protocol and its optimization with a single sender *(s-s)* and with multiple senders *(m-s)* in which the spanning tree built by the protocol is shared by all nodes to disseminate messages.

4.1. Experimental Setting

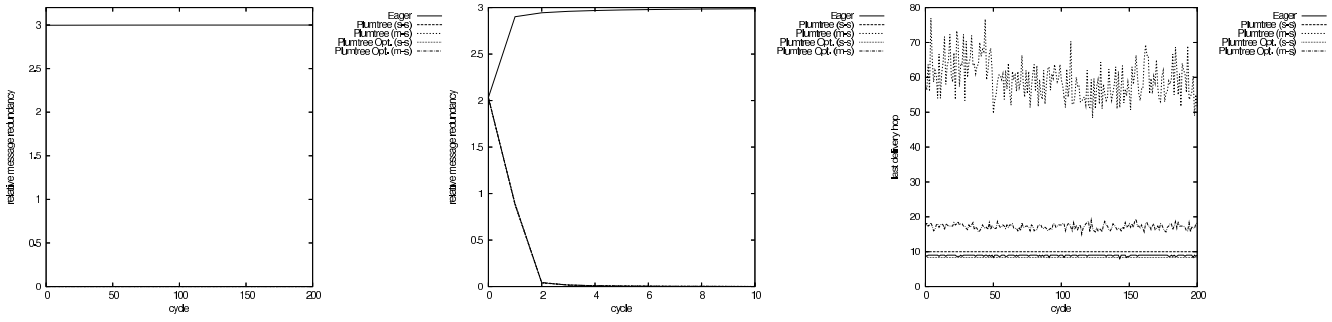
All experiments were conducted in a network composed of 10.000 nodes and results show an aggregation from several runs of each experiment. We used the same peer sampling service in all experiments. We selected the HyParView peer sampling protocol [16, 15] as the service it provides has all the desirable properties stated in Section 3.3. HyParView uses small active views to disseminate messages, which makes the eager strategy efficient. HyParView was configured with the same settings we employed in [16]. Manny of the settings are specific to the internal operation of the protocol and are not relevant in the context of these experiments. However, it is important to state that the active membership size of the protocol was configured to a size of 5. Therefore, the configuration of the fanout value for both the eager and the Plumtree protocols were set to 4.

Given that the properties of HyParView allows us to use a very small fanout value and still get a connected overlay, the message complexity numbers for the eager gossip are somehow conservative.

For the simulations, we did not use any piggybacking policy for the IHAVE messages, *i.e.*, an IHAVE message is sent immediately in each lazy link. Finally, the *threshold* parameter of the Plumtree optimized version was configured with a value of 3 for the single sender mode and a value of 7 for the multiple senders mode. This value was selected by taking in consideration the diameter of the overlay which was typically between 8 and 9 and also because it presented itself as a good choice in earlier simulations.

4.2. Stable environment

We start by presenting evaluation of the Plumtree protocol in a stable environment, where no node failure was induced for the whole duration of the simulations.



(a) RMR (b) RMR in the bootstrap (c) LDH

Figure 2. Behavior in stable environment

Figure 2a depicts the relative message redundancy (as defined in Section 2) for the last 200 cycles of simulation.

The important aspect to retain from this figure is that, as expected, the eager strategy produces a constant value of 3, while both the Plumtree and its optimization generate a value of 0 (for most messages). This is why the lines for Plumtree are not visible.

It is interesting to observe in detail, the behavior of the protocol during the construction of the tree. One can expect that during the construction of the spanning tree, some overhead of messages is present. To quantify this we depict in Figure 2b the relative message redundancy for the first 10 cycles of simulation. Notice that for the 2 first cycles of simulation the Plumtree protocols generate more redundant messages. However, the amount of redundant messages is always below the number of redundant messages produced by the eager strategy.

Figure 2c presents the values for LDH for all protocols. The eager protocol and Plumtree with a single sender present the best performance. Notice that the eager protocol uses all available links to disseminate messages, which ensures that all shortest paths of the overlay are used. This shows that with a single sender Plumtree is able to select links that provide faster delivery. With multiple senders the original Plumtree protocol values are very high. This happens because the spanning tree is optimized to the node who broadcasts the first message therefore, when messages are sent by nodes located at leaf positions of the tree they require to execute more hops in the overlay to reach all other nodes. On the other hand, the optimization of the protocol is able to lower significantly the value of LDH. The reader should notice that because the optimization triggers for different senders, in fact it will better distribute links that form the spanning tree through the overlay, effectively removing the bias of the tree to the sender of the first message.

	Payload	Control	Total
Eager	39984.00	0.00	39984.00
Plumtree (s-s)	9999.00	29987.33	39986.33
Plumtree (m-s)	9999.00	29990.00	39989.00
Plumtree Opt. (s-s)	9999.00	29989.33	39988.33
Plumtree Opt. (m-s)	9999.00	38976.00	48975.00

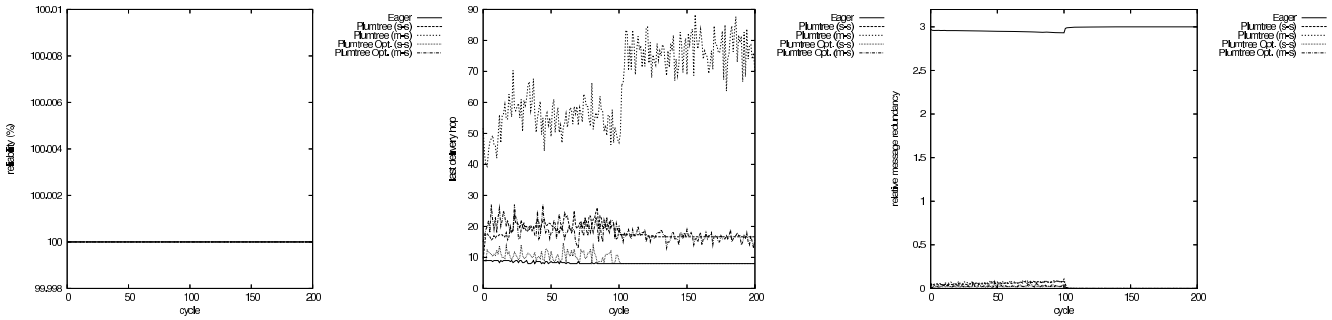
Table 1. Number of messages received

Table 1 shows the number of messages received by each strategy in the 100° cycle. The extra control messages received when using the Plumtree are essentially due to IHAVE messages however, the reader should consider that: *i)* usually IHAVE messages are smaller than payload message, hence these messages will contribute less to the exhaustion of network resources and *ii)* IHAVE messages can be aggregated, by delaying the transmission of these messages and sending several payload message identifiers in a single IHAVE message. Notice that this will not have an impact in reliability, as messages are sent anyway only with a delay. Therefore, this would only affect the time required to repair the spanning tree after failures and the overall latency of the system.

The reader should also notice that the Plumtree protocol optimization with multiple senders generates close to 10.000 extra control messages that the same protocol with a single sender or the original Plumtree protocol with multiple senders. This represents a 22.5% increase in signaling cost. This happens due to the following phenomena. Because each message is sent by a different node, the protocol will trigger many times its optimization routine. This requires the transmission of 2 extra messages to neighbors, which explains the higher amount of control messages in this case.

4.3. Multiple failures

In this section, we present the behavior of the protocols when a constant failure rate is induced in the system. To that end, after the period of stabilization, we failed 50 nodes in each of the first 100 cycles. The results are depicted in Figure 3.



(a) Reliability (b) LDH (c) RMR

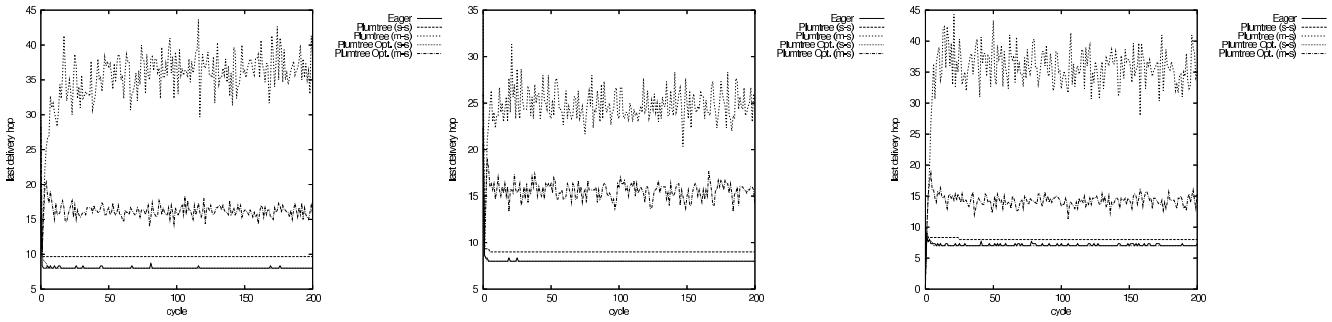
Figure 3. Behavior for a failure rate of 50 nodes per cycle

In terms of reliability, all protocols are able to sustain the constant rate of failure without any impact on their reliability. Figure 3a shows a constant reliability of 100%. The LDH value of all protocols is slightly affected by failures. Figure 3b show that LDH values become unstable for all protocols in cycles where failures are induced. This is expected, as these failures may remove links that were part of the optimal paths between nodes. Results also show that when failures are no longer induced, all protocol are able to regain a more constant value. Also the original Plumtree protocol with multiple senders is the case where the impact of failures is more visible, as the LDH value is significantly lower during this period. This happens because of the following phenomena. The addition of new links as a reaction to failures will produce some redundant paths in the spanning tree. This allows the protocol to optimize the tree by selecting links which reduce the distance to the sender of the first message broadcasted hence, reducing the LDH for those messages. Finally, Figure 3c shows the RMR values for all protocols. The eager protocol presents values slightly lower than 3 in the first 100 cycles while the Plumtree

protocol (in all cases) presents values slightly above 0. Notice that failures will remove some links from the overlay therefore, there will be less payload messages being received by nodes with the eager protocol, on the other hand, the membership protocol will add new links to replace the lost ones. This will trigger *NeighborUp* events in some nodes, which will add new neighbors to their *LazyPushPeers* set hence, more payload messages will be sent and therefore, the number of redundant payload messages received by nodes will increase.

4.4. Massive failures

Finally, we present the effect of massive failures in the behavior of the protocols. A massive failure is when a large percentage of nodes fail simultaneously. In order to do this, we induced several percentages of node failure after the stabilization period. We experimented with failure percentages ranging from 10% to 95% of all nodes in the system.



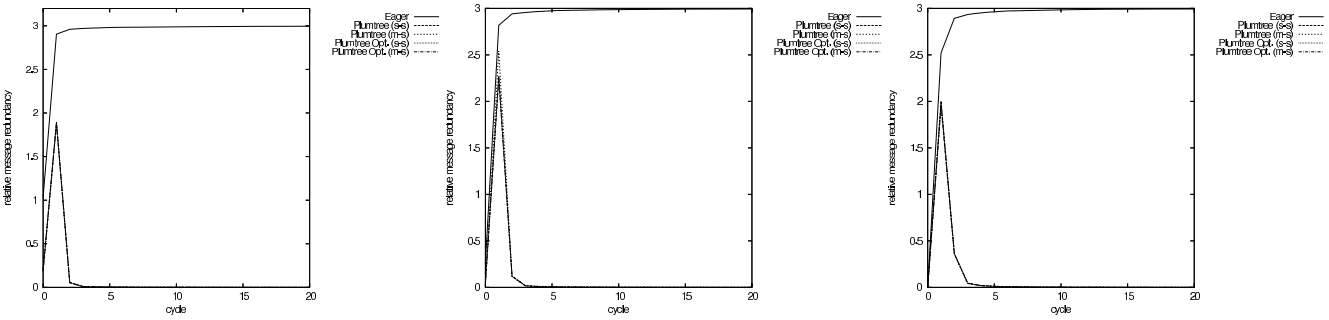
(a) 40% (b) 60% (c) 80%

Figure 4. Last delivery hop after massive failures

Figure 4 shows the LDH for all protocols after failures for 3 different failure percentages. As expected, it confirms the results presented above. All protocols are able to maintain a, somewhat, constant value for LDH. Whereas the eager protocol and Plumtree (and its optimization) with a single sender have the best performance, followed by the optimized Plumtree with multiple senders and finally the original Plumtree with multiple senders.

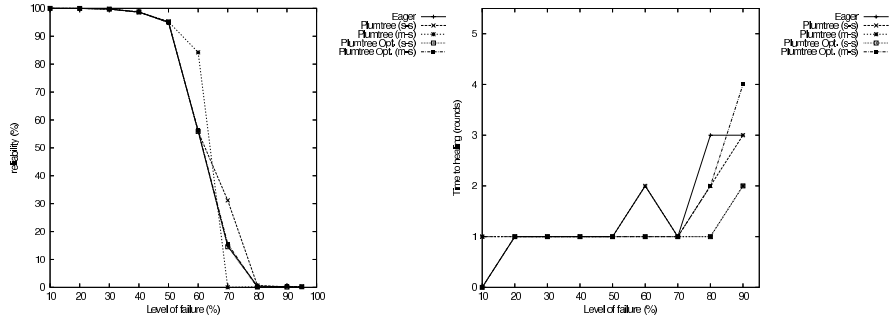
Figure 5 depicts the RMR after failures for the same failure percentages depicted before. The important aspect to retain from these figures is that all protocols are able to regain their RMR levels before failures in only a couple of cycles. After failures all protocols exhibit a low level of redundancy. For all failure percentages, all versions of the Plumtree protocol show an increase of redundant messages after failures. This is due to the transmission of extra payload messages as a result of the healing process of the overlay, which adds new links to compensate those lost due to failures. Eager protocol redundancy of payload messages increases with the healing of the overlay.

We finally present the summary of some effects of these massive failures for each failure percentage. Figure 6a



(a) 40% (b) 60% (c) 80%

Figure 5. Relative message redundancy after massive failures



(a) Reliability (b) Healing time

Figure 6. Impact of massive failures

depicts the reliability of each protocol immediately after failures. The reader should notice that for failure percentages above 70%, reliability drops to values close to 0 for all protocols. This happens because the overlay becomes disconnected after these failures. However, protocols are able to regain their reliability. This is depicted in Figure 6b where the number of cycles required by all protocols to regain their reliability is counted. Again the time to regain realiability, in number of simulation cycles, is not significantly different. This clearly shows that Plumtree retains the reliability of eager gossip protocols, given that the tree embedding is only used to select which links are used for eager/lazy push.

4.5. Discussion

Our results show that, an embedded spanning tree, on top of a low cost random overlay network allows to disseminate messages in a reliable manner with considerably less traffic on the overlay than a simple gossip protocol. Moreover, by exploiting links of the random overlay that are not part of the spanning tree, one can efficiently detect partitions and repair the tree. One interesting aspect of our approach, is that it can be easily used to provide optimized results for a small number of source nodes, by maintaining state for one spanning tree for

each source. This is feasible as our approach does not require the maintenance of complex state.

In terms of latency, our scheme is more effective for building sender based trees but, with the optimizations proposed, it can also be used to support shared trees, with a penalty in terms of overall system latency which presents twice that value. This can be achieved by relaxing the constraints on the stability of the spanning tree. In such a way, one can improve the latency of the system and provide better results when the spanning tree is shared by several nodes to disseminate messages. We showed that the same strategy used to detect and repair the spanning tree can easily be extended to optimize the tree for these conditions. We defend that this is of paramount importance in order to avoid the negative impact in terms of latency when sharing the tree.

5. Related Work

Several application level multicast protocols rely on some sort of spanning tree to efficiently disseminate messages. The Narada protocol [5], is used to support efficient application-level multicast, relying in dissemination trees that are produced in two distinct steps. A first step is used to generate a rich connected overlay which is self-organizing and self-improving. In the second step several multicast trees rooted at each source node are built on top of the overlay created in the first step. This is achieved by relying in a distance vector algorithm. Nodes that wish to join a multicast group explicitly select their parents among their neighbors using information from the routing algorithm. Narada also employs a random overlay network that is used to build the spanning tree, but unlike our approach, it requires a routing algorithm to be executed on top of that overlay, which makes their approach more complex. Unfortunately, Narada is targeted toward medium sized groups; all nodes maintain full membership list and some additional control information for all other nodes consequently, it can't scale to very large systems. Also the normal dynamics of the algorithm may partition the overlay. Affecting the global reliability of the system, until the protocol is able to repair the overlay.

Bayeux [25], is a source-specific, application-level multicast system that leverages in Tapestry [24], a wide-area location and routing architecture that also maintains an overlay network. Bayeux model of operation is also based on messages that are used to explicitly create branches across nodes, these messages are delivered to nodes by using Tapestry to route messages. Nodes that are used by Tapestry to route these messages change their internal state to reflect the structure of the spanning tree. Unlike our approach, Bayeux employs a complex location and routing architecture therefore, it has a larger complexity when compared with our gossip-based random overlay. Although Bayeux is fault-tolerant, it requires that root nodes maintain information concerning all receivers. Also root nodes are single point of failure and a bottleneck for message dissemination. The authors propose a replication

scheme to compensate for this, never the less this implies that Bayeux will not scale properly in very large systems with several thousands of receivers.

Scribe [22, 4] is a scalable application-level multicast infrastructure built on top of Pastry [21]. Scribe supports multicast groups with multiple senders. To that end each group has a node that serves as *rendez-vous point* and that is selected by taking advantage of Pastry resource location mechanism. This node will serve as a root for the spanning tree. State concerning the spanning tree is maintained through a *soft state* approach hence, nodes have to periodically rejoin the spanning tree. A mechanism based in HEARTBEAT messages is used to detect failures. When a node stops receiving HEARTBEATS from its parent, it rejoins the spanning tree. Although Scribe is fault-tolerant and it provides a mechanism to handle root failures, it only provides best-effort guarantees. The authors argue that strong reliability and also order guarantees are only required for some applications, and that those properties can be easily provided on top of Scribe.

MON [17], which stands for Management Overlay Network, is a system designed to facilitate the management of large distributed applications and is currently deployed in the PlanetLab testbed[1] [6] . MON is the most similar approach to our own. It builds on-demand overlay structures that are used to issue a set of *instant management commands* or distribute software across a large set of nodes. To that end it also uses a random overlay network based in *partial views* that is maintained by a *cyclic* approach. It supports the construction of both tree structures and directed acyclic graphs structures. Spanning trees in MON are always rooted in external elements. The construction of the tree is done by broadcasting a special message (SESSION message) through the overlay in an epidemic manner. Specific messages are used to reply to SESSION messages to indicate if the link used to transmit it will be used in the spanning tree (SESSIONOK) or if it is redundant (PRUNE). However, because MON is aimed at supporting short-lived interactions, it does not require to maintain the spanning tree for prolonged time, hence it does not have any repair mechanism to cope with failures.

6. Conclusions and Future Work

Epidemic protocols are an attractive approach to build highly reliable and scalable broadcast primitives as they have a natural resilience to message loss and node failures. On the other hand tree-based broadcast primitives have smaller message complexity in steady-state but are very fragile in the presence of faults.

In this paper we proposed an integrated broadcast scheme that combines both approaches. We used a low cost scheme to build and maintain broadcast trees embedded on a low cost gossip-based random overlay network. The

[1]For further information the reader should check: http://planet-lab.org/

protocol disseminate payload messages preferably via tree branches but uses the remaining links of the random overlay for fast recovery and expedite tree healing. Experimental evaluation presented in the paper showed that our new strategy presents a low overhead and that it is able to support large number of faults while maintaining a high reliability.

As future work we aspire to develop new strategies and membership services that are able to exhibit a self-adaptive behavior concerning the underlying network conditions. Our final goal is to develop a system based on self-adaptive overlay networks that is able to directly compete with low level multicast primitives, like IP-Multicast.

References

[1] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 201–210, June – July 2004.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[3] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proceedings of the Internacional Conference on Dependable Systems and Networks (DSN)*, Edinburgh, UK, June 2007.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[5] Y.-H. Chu, S. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, Oct 2002.

[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[7] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[9] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. Crew: A gossip-based flash-dissemination system. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 45, Washington, DC, USA, 2006. IEEE Computer Society.

[10] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, / 2000.

[11] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*, pages 44–55, 2001.

[12] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[13] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):248–258, 2003.

[14] B. Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, pages 76–87, Florence,Italy, Oct. 2003.

[15] J. Leitão. Gossip-based broadcast protocols. Master's thesis, University of Lisbon, 2007.

[16] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: a membership protocol for reliable gossip-based broadcast. In *Proceedings of the Internacional Conference on Dependable Systems and Networks (DSN)*, Edinburgh, UK, June 2007.

[17] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. MON: On-demand overlays for distributed system management. In *2nd USENIX Workshop on Real, Large Distributed Systems (WORLDS'05)*, 2005.

[18] Peersim p2p simulator. http://peersim.sourceforge.net/.

[19] J. Pereira, L. Rodrigues, M. J. Monteiro, R. Oliveira, and A.-M. Kermarrec. Neem: Network-friendly epidemic multicast. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, pages 15–24, Florence,Italy, Oct. 2003.

[20] J. Pereira, L. Rodrigues, A. Pinto, and R. Oliveira. Low-latency probabilistic broadcast in wide area networks. In *Proceedings of the 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, pages 299–308, Florianopolis, Brazil, Oct. 2004.

[21] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[22] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[23] S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[24] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

[25] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*, June 2001.