Araneola: A Scalable Reliable Multicast System for Dynamic Environments

Roie Melamed
CS Department, Technion
mroi@cs.technion.ac.il

Idit Keidar
EE Department, Technion

Abstract

We present Araneola, a scalable reliable application-level multicast system for highly dynamic wide-area environments. Araneola supports multi-point to multi-point reliable communication in a fully distributed manner while incurring constant load on each node. For a tunable parameter $k \geq 3$, Araneola constructs and dynamically maintains an overlay structure in which each node's degree is either k or $k + 1$, and roughly 90% of the nodes have degree k. Empirical evaluation shows that Araneola's overlay structure achieves three important mathematical properties of k-regular random graphs (i.e., random graphs in which each node has exactly k neighbors) with N nodes: (i) its diameter grows logarithmically with N; (ii) it is generally k-connected; and (iii) it remains highly connected following random removal of linear-size subsets of edges or nodes. The overlay is constructed at a very low cost: each join, leave, or failure is handled locally, and entails the sending of only about $3k$ messages in total.*

Given this overlay, Araneola disseminates multicast messages by gossiping over the overlay's links. We show that compared to a standard gossip-based multicast protocol, Araneola achieves substantial improvements in load, reliability, and latency. Finally, we present an extension to Araneola in which the basic overlay is enhanced with additional links chosen according to geographic proximity and available bandwidth. We show that this approach reduces the number of physical hops messages traverse without hurting the overlay's robustness.

1. Introduction

Our goal is to provide a scalable multi-point to multi-point reliable multicast service for very large groups in wide-area networks. A protocol deployed in such a setting must be able to withstand frequent node failures as well as non-negligible message loss rates [20]. Moreover, studies have shown that users typically frequently join and leave multicast sessions [1]; such behavior is called *churn*. A major design goal for our work is therefore coping efficiently with churn. Specifically, we address the following challenges: (i) providing high reliability despite considerable message loss and failure rates while incurring constant load on each node; (ii) incorporating joining nodes and removing leaving (or failing) ones with a low *constant* overhead; and (iii) providing an undisrupted service to nodes that are up despite high churn rates.

We present Araneola, a scalable reliable application level multicast (ALM) system for dynamic wide-area environments. Reliability is achieved by constructing a richly-connected overlay and disseminating pertinent information on multiple paths in this overlay. The number of paths in the overlay can be tuned according to the expected failure and loss rates. Araneola is designed to incur small constant load on each node. To this end, it builds an overlay in which each node's degree is bounded by a small constant. This approach has three advantages: (i) all nodes, including low bandwidth ones, are capable of participating in the overlay; (ii) the load on all nodes is similar, so no user is required to contribute more bandwidth than its fair share; and (iii) nodes have ample remaining bandwidth for connecting to nearby nodes, as we explain below.

Our search for a constant degree overlay leads us to consider k-regular random graphs. In a k-regular graph, each node's degree is k. A *k-regular random graph* with N nodes is a graph chosen uniformly at random from the set of k-regular graphs with N nodes. For $k \geq 3$, a k-regular random graph is almost always a good expander [8], which implies that (i) its diameter grows logarithmically with N [25]; and (ii) it remains connected after random failures of a linear subset of its nodes and/or edges [9]. In addition, such a graph is generally k-connected, i.e., at least k nodes need to be removed in order to cause a partition [25][2]. We strive to construct and maintain an overlay that resembles a k-regular random graph: Araneola's basic overlay converges to a graph in which each node has a degree of either k or

*Araneola means "little spider" in Latin.

[2]The probability that a k-regular random graph is not k-connected is bounded by $O(N^{2-k})$.

$k+1$ and no two neighboring nodes have a degree of $k+1$. Empirically, we show that Araneola's overlay achieves the desired properties above, namely logarithmic diameter, k-connectivity, and high robustness.

We construct and maintain the overlay at a very low constant overhead; each join or leave (or failure) incurs sending roughly about $3k$ messages in a k-degree overlay, regardless of the number of nodes. Remarkably, in dynamic settings, the cost of handling a single join or leave operation *decreases* as the churn rate increases. This is in contrast to virtually all existing structured peer-to-peer overlays, with which the overhead for handling joins grows logarithmically with N.

The low maintenance cost is achieved due to the facts that: (i) each join, leave, or failure is handled locally; and (ii) the selection of random neighbors uses partial membership views maintained by a distributed low cost membership service similar to the ones in [7, 23]. The overhead of the membership service is independent of the number of nodes and of the churn rate.

Having built a basic k/k+1-degree overlay, we next extend it by adding links between geographically-close nodes. The low degree of Araneola's basic structure allows for allocating plenty of bandwidth for communication with proximate nodes. We show that with this approach, the links in Araneola's overlay traverse substantially fewer physical hops on average. Moreover, the overlay's robustness does not deteriorate.

Given Araneola's overlay, multicast messages are disseminated through gossip between each pair of neighbors. Gossiping in Araneola differs from a standard gossip protocol (e.g., [7, 16]) in that with a standard gossip protocol, each node chooses *different* random nodes to gossip with in each round, whereas in Araneola, each node always gossips with its neighbors in the overlay. We show that this difference leads to substantial improvements in load, reliability, and latency.

This paper proceeds as follows: Section 2 discusses related work. Section 3 presents Araneola's design and pseudo code, and Section 4 empirically evaluates Araneola. Section 5 presents and evaluates the extension that exploits network proximity. Section 6 concludes the paper.

2. Related Work

In recent years, two leading approaches for supporting scalable ALM in dynamic failure-prone networks have emerged: gossip-based (or epidemic) multicast protocols (e.g., [3, 7]) and dynamic overlay networks (e.g., [10, 5]). With gossip-based protocols, each node periodically chooses other random nodes to propagate the information to. Gossip-based multicast generally achieves good load balancing, high reliability, and undisrupted service in the presence of message loss and node joins and leaves, but it also induces a high load, as many duplicate messages are sent [7].

Overlay-based ALM systems usually disseminate messages on a tree structure [10, 5, 23]. With tree-based multicast, no duplicate messages are sent. However, in the presence of churn, the tree structure will frequently become partitioned, causing a significant portion of the multicast messages to be lost. Therefore, in order to achieve reliability, such protocols need to detect message loss and recover from it. This can cause recovered messages to be significantly delayed; can induce substantial overhead, especially if failures are frequent; and can inhibit scalability. A second problem with tree-based multicast is uneven load distribution: as recently argued in [4], inner nodes in the tree carry the burden for the multicast, whereas leaf nodes do not share the load. Two recent projects, SplitStream [4] and Bullet [14], address this issue and build a balanced multicast infrastructure; however these two systems are intended for single-source multimedia transfer and do not strive to provide multi-point to multi-point communication or full reliability of message delivery as we do.

Recently, several peer-to-peer overlays with logarithmic diameters and a bounded node degrees have been suggested, e.g., emulating the Butterfly [17], de Bruijn graphs [11], Small Worlds graphs [13], or random expander graphs with degrees≥ 8 [15]. However, none of these systems can guarantee, with high probability, a lower cost than $O(\log N)$ messages and time for handling joins, since a joining node must search and locate its (random or hashed) joining location prior to joining the system. Chawathe et al. [6] have argued that this logarithmic cost inhibits the scalability of such systems assuming the churn rates measured in Gnutella and Napster [22].

Like our extension of Araneola, several overlay structures, e.g., [18, 23], reduce message delivery latency and communication costs by incorporating links between nearby nodes in addition to the random links required for achieving a good overlay. In comparison with [18, 23], Araneola achieves a smaller average degree than [18, 23] and better load balancing than [23].

3. Araneola's Design

Araneola builds an overlay structure for each multicast group. Since each group is handled independently, we present the protocol for a single group, and omit the group's name. All Araneola nodes run the same code. The code has two main components: one constructs and maintains the overlay, and the second implements the multicast service. The multicast service is very simple: once the overlay is constructed, each Araneola node gossips about recent message identifiers with its neighbors, and requests missing messages from them. Due to space limitations, we do not describe this service in this paper; the details are described in the full paper [19].

When joining the overlay, a node randomly selects several other nodes to connect to. This requires each node to know some other nodes' identities. To this end, we implement a scalable randomized membership protocol similar to [7, 23], where membership information is gossiped over the overlay's links. Each node maintains a small set of node identities, called a *membership view*, which evolves over time. Periodically, the membership protocol piggybacks a small amount of information on gossip messages. Such views have been shown to become uniformly distributed over time [7, 23]. Experimentally, we observe that it suffices to piggyback membership information infrequently, e.g., once a minute. When a new node joins Araneola for the first time, it can ask another node for its membership view, and use that as its initial view. We do not detail the membership protocol in this paper; the interested reader is referred to [7].

Araneola's data structures are presented in Fig. 1. The set *neighbors* holds the node's current neighbors in the overlay, with their respective degrees. The degree of a node is the size of its neighbors set, i.e., $|neighbors|$. *last_connect_to_time* records the latest time a CONNECT_TO message was handled, as explained below. The parameter L determines the graph's target degree (k), and H defines the maximum allowed node degree. A number of timeout values are defined in order to control the frequency at which different events occur.

Data structures:
id – this node's identifier.
neighbors – set of triples $\langle$id,degree$\rangle$, initially $\emptyset$.
last_connect_to_time – a time.
Constants:
L – target number of neighbors.
H – upper bound on the number of neighbors.
Timeouts: connect_timeout, disconnect_timeout, connect_to_timeout.

Figure 1. Araneola's data structures and constant definitions.

Three tasks participate in the construction and maintenance of the overlay: (i) the *connect task* (see Fig. 2) adds new connections when a node's degree is below L; (ii) the *disconnect task* (see Fig. 3) tries to reduce the node's degree if it is above L, without causing any node's degree to drop below L; and (iii) the *failure detector* detects failures and recovers from them.

When a node's degree is below L, the connect task periodically attempts to set up as many new connections as it is missing to randomly chosen nodes (lines 1–6). The target nodes are chosen at random from the local membership view. For each attempted connection, the node sends a CONNECT request (line 5). At bootstrap time, the node issues CONNECT requests to L nodes, and then sleeps for *connect_timeout*. It is expected that during this period enough new connections will be formed, although since some of the chosen nodes may be faulty or overloaded, there may be a need to attempt more connections after the timer expires. The connect task can be awaken by other tasks before the timer expires (line 27).

A node that receives a CONNECT request (line 7) *accepts* it, by calling add_connection, provided that its degree is smaller than H, and otherwise it *redirects* the request, as will be explained shortly. The procedure add_connection adds the sender to neighbors (line 21) and responds with a CONNECT_OK. Upon receiving the CONNECT_OK (line 14), the requester registers the new connection, unless its degree has already reached H, in which case it sends a LEAVE message (line 18). A LEAVE message causes its receiver to remove its connection with the sender (lines 19–20) . Redirecting is done by sending a REDIRECT message to the requester, naming the sender's lowest degree neighbor (line 11). This causes the requester to send a new CONNECT request to the named neighbor (line 13). CONNECT and CONNECT_OK messages carry the sender's current degree for initializing the *degree* in the *neighbors* data structure. In addition, every node periodically sends its degree to its neighbors, in order to keep the neighbors data structure up-to-date (this is not shown in the code).

A node that voluntarily leaves the system sends a LEAVE message to all its neighbors. Unexpected node failures are detected using a simple heartbeat failure detector (not shown in the code). When a node detects a neighbor as faulty, it removes the connection with this neighbor.

There are two rules for removing connections: *Rule 1* and *Rule 2* (See Fig. 3). Rule 1 removes the connection between a pair of nodes that both have degrees higher than L. Specifically, if a node n's degree is $L + i$, then n attempts to remove i of its neighbors. The i neighbors with the highest degrees are candidates for removal; they are inserted into the set *cands* (line 5). Nodes with degrees $\leq$L are then deleted from *cands* (line 8). If n has a higher id than a node c in *cands*, then n sends a DISCONNECT message to c (line 10). Upon receiving this message (line 16), if c's degree is still higher than L, it removes the connection with n, and sends a DISCONNECT_OK message. Upon receipt of a DISCONNECT_OK (line 20), n removes the connection with c. Note that Rule 1 never reduces a node's degree to be below L.

With Rule 1, it is still possible for a node to have degree H while all of its neighbors have degree L. Rule 2 is only invoked at a node n when all of n's neighbors' degrees are $\leq$L. With Rule 2, node n chooses its two neighbors with the highest and lowest degrees, h and l, resp. (lines 12–13). If n's degree is at least $l.degree + 2$, then n tries to cause h to shift one of its connections from n to l. But before removing h's connection with n, we ensure that l is

Connect task:

1. loop forever
2. $gap \leftarrow$ L $- |neighbors|$
3. **for** $(i = 0; i < gap; i++)$
4. $n \leftarrow$ *random node*
5. send $\langle$CONNECT, $|neighbors|\rangle$ to n
6. sleep (connect_timeout)

Event handles:
7. **upon** receive $\langle$CONNECT, $d\rangle$ from n **do**
8. **if** $(|neighbors| <$ H$)$ **then**
9. add_connection (n, d,true)
10. **else**
11. send $\langle$REDIRECT,lowest degree neighbor$\rangle$ to n

12. **upon** receive $\langle$REDIRECT, $n'\rangle$ from n **do**
13. send $\langle$CONNECT, $|neighbors|\rangle$ to n'

14. **upon** receive $\langle$CONNECT_OK, $d\rangle$ from n **do**
15. **if** $(|neighbors| <$ H$)$ **then**
16. add_connection (n, d,false)
17. **else**
18. send $\langle$LEAVE$\rangle$ to n

19. **upon** receive $\langle$LEAVE$\rangle$ from n **do**
20. remove_connection(n)

Procedures:
Procedure add_connection (node_id n, int d, boolean ack)
21. $neighbors \leftarrow neighbors \bigcup \{n, d\}$
22. **if** $(ack = true)$ **then**
23. send $\langle$CONNECT_OK, $|neighbors|\rangle$ to n

Procedure remove_connection (node n)
24. remove n from *neighbors*
25. remove n from all *heard_from* lists
26. **if** $(|neighbors| <$ L$)$ **then**
27. wake up connect task

Figure 2. Overlay construction: the connect task.

Disconnect task:

1. loop forever
2. sleep (disconnect_timeout)
3. $i \leftarrow |neighbors| -$ L
4. **if** $(i > 0)$ **then**
/* *Rule 1* */
5. $cands \leftarrow$ set of i neighbors with highest degrees
6. **foreach** $c \in cands$
7. **if** $(c.degree \leq$ L$)$ **then**
8. $cands \leftarrow cands \setminus \{c\}$
9. **else if** $(c.id < id)$ **then**
10. send $\langle$DISCONNECT$\rangle$ to c
/* *Rule 2* */
11. **if** $(cands = \emptyset)$ **then**
12. $h \leftarrow$ neighbor with highest degree
13. $l \leftarrow$ neighbor with lowest degree
14. **if** $(|neighbors| \geq l.degree + 2)$ **then**
15. send $\langle$CONNECT_TO,$h\rangle$ to l

Event handlers:
16. **upon** receive $\langle$DISCONNECT$\rangle$ from n **do**
17. **if** $(|neighbors| >$ L$)$ **then**
18. send $\langle$DISCONNECT_OK$\rangle$ to n
19. remove_connection(n)

20. **upon** receive $\langle$DISCONNECT_OK$\rangle$ from n **do**
21. remove_connection(n)

22. **upon** receive $\langle$CONNECT_TO, $n'\rangle$ from n **do**
23. **if** $(|neighbors| \leq$ L $\wedge$
24. *clock* $-$ *last_connect_to_time* $>$ connect_to_timeout) **then**
25. send $\langle$CHANGE_CONNECTION,$|neighbors|, n\rangle$ to n'
26. *last_connect_to_time*$\leftarrow$*clock*

27. **upon** receive $\langle$CHANGE_CONNECTION,$d, n'\rangle$ from n **do**
28. **if** $(|neighbors| <$ H$)$ **then**
29. add_connection (n, d,true)
30. send $\langle$DISCONNECT$\rangle$ to n'

Figure 3. Reducing node degrees.

willing to accept h's connection. Therefore, n contacts l (rather than h) and asks it to try to connect to h, and to ask h to remove its connection with n. To this end, n sends a $\langle$CONNECT_TO,$h\rangle$ message to l. If upon receiving this message l's degree is still $\leq$L, and l has not handled another CONNECT_TO request in the last connect_to_timeout (lines 23–24), then l sends a CHANGE_CONNECTION message to h. The recipient, h, connects to l (line 29) and sends a DISCONNECT message to n (line 30). This can increase l's degree, but not to become higher than L+1, since l handles at most one CONNECT_TO request at a time, and only if its degree$\leq$L. Moreover, note that if l's degree will become higher than L, and n's degree will remain above L, then Rule 1 will eventually reduce l's degree back to L.

In the full paper [19], we prove the following:

Proposition 1. *If from some point onward no nodes join, leave, or are detected as faulty, then each node's degree is eventually either L or L+1, and at most 50% of the nodes have degree L+1.*

4. Evaluation

We have implemented Araneola in Java using UDP/IP. In this section, we evaluate Araneola on a single LAN in Netbed [24]. In the next section, where we consider network proximity, we evaluate Araneola also on a WAN. We run multiple Araneola nodes per machine, and therefore need to space the gossip rounds sufficiently so as to allow all the nodes running on the same machine to complete their gossip operation during a round. Thus, we chose a fairly large round duration of 5 seconds. When there is only one node per machine, the round duration can be an order of magnitude smaller. The *disconnect_timeout* is set to 30 seconds, and the *connect_timeout* is 20 seconds. We begin our study, in Section 4.1, by evaluating Araneola's scalability and performance in a static setting. In Section 4.2 we study Araneola's fault-tolerance. In Section 4.3, we consider high churn.

4.1. Static Evaluation

In our static evaluation, all the nodes are created simultaneously, and remain up throughout the experiment. In each round, a single data message is injected into the system, each time from a different machine. A total of 200 data messages are sent in each experiment. Each experiment (with a given number of nodes and choice of parameter settings) was run at least 4 times, for a total of several dozens.

4.1.1 The impact of L

Araneola's parameter L is very significant: it affects Araneola's load, latency, and robustness. Fig. 4 shows the impact of this parameter on message propagation rates in runs with 8000 nodes (on 100 Netbed machines) and values of L ranging from 3 to 10. Each curve in the figure depicts the CDF of the average number of nodes that receive a message by each hop count for a given value of L. As expected, the message latency decreases as L increases.

In most of the experiments below, we set L to 5. We chose this value because it provides a good balance between the desired properties: each node sends a given message identifier only 4 times, and the latency to reach all nodes is reasonable: 7 rounds with 1000 nodes and 9 with $10,000$. Moreover, as we shall see below, it yields a highly robust overlay and achieves 100% reliability at churn rates exceeding those measured on the MBone [1]. We choose H to be L+5 because we have observed that this choice achieves low overhead.

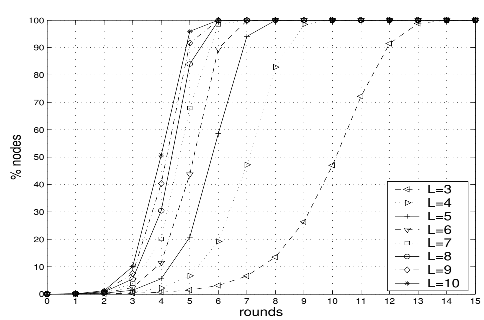


Figure 4. CDF: average % of nodes that receive a message for different degree Araneola overlays, 8000 nodes.

4.1.2 Overlay properties and scalability

In order to understand Araneola's scalability, we vary N, the group size, from 500 nodes (on 10 Netbed machines) to $10,000$ nodes (on 125 Netbed machines). L and H are set to 5 and 10, resp. At the end of each experiment, we take a snapshot of the overlay structure, and then analyze its properties offline. We measure node degrees as well as the overlay's diameter, average distance, and connectivity. The results are summarized in Table 5[3]. The first column shows the percentage of nodes whose degree is L (i.e., 5). The remaining nodes' degrees are L$+1$. In all of our experiments, Araneola converges to a state in which over 90% of the nodes have degree L. The exact percentage of nodes with degree L does not seem related to N. The next column presents the smallest and largest measured diameters for every value of N. In cases where the same diameter was measured in all experiments, we present only one value. The diameter gives a measure for the *worst case* latency (in the absence of failures and message loss), whereas the average latency depends on the average distance between two nodes in the overlay. This average is presented in the following column, and it increases gradually with N. Finally, we measure the overlay's connectivity. In over 90% of our experiments, the overlay is 5-connected, i.e., there are at least 5 disjoint paths between every pair of nodes. In the few cases where the connectivity was less than 5, there were at most 4 nodes with a connectivity of 4, whereas the rest of the nodes had a connectivity of 5. The average number of node-disjoint paths between every pair of nodes is presented in the last column. It does not vary with N.

N	% nodes degree= 5	diameter	avg distance	avg # paths
500	91.8	6–7	4.18	5.01
1000	91.4	7	4.69	5.01
2000	92	7–8	5.16	5.01
4000	91.45	8	5.63	5.01
6000	90.42	8–9	5.93	5.01
8000	90.33	9	6.12	5.01
10000	90.36	9	—	—

Figure 5. Scalability of Araneola's overlay.

Fig. 6 depicts the message propagation rates measured for values of N ranging from 500 to $10,000$. As N increases, messages take longer to propagate, but the slowdown is gradual. The average hop-count in each experiment is very close to the average distance in that experiment's overlay (see Table 5, 3rd column).

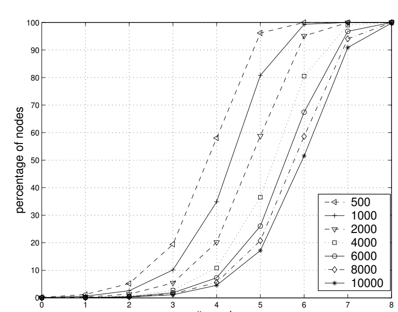


Figure 6. Message propagation rates for different group sizes, L=5.

[3] We did not analyze the average distance and connectivity for the experiments with $N = 10,000$.

4.1.3 Comparison with gossip protocol

We now compare Araneola to a standard gossip protocol [16] implemented using Araneola's gossip-based multicast module. The gossip protocol takes a parameter F, its fan-out. Where an Araneola node sends gossip messages to its neighbors, the gossip protocol sends gossip messages to F randomly selected nodes from its membership view. Whereas Araneola sends each message identifier downstream only, in the gossip protocol each node sends each message identifier to all the chosen targets. Thus, gossip protocol instantiated with a fan-out of F sends information as many times as Araneola with L= $F + 1$.

We experiment with 1000 nodes on 20 Netbed machines. In each experiment, 400 messages are sent. Fig. 7 compares the average message propagation rates of Araneola with L= 5 and 6 to those of the gossip protocol with the corresponding fan-outs $F = 4$ and 5. Evidently, Araneola propagates information much more effectively than the gossip protocol. Initially, the propagation rates are similar, but after about 6 rounds, Araneola continues to effectively propagate the message, while the gossip protocol tapers off. Araneola succeeds in disseminating all the messages to 100% of the nodes in 7 rounds with L= 5, and in 6 rounds with L= 6. In contrast, the gossip protocol only reaches 95.91% of the nodes on average with $F = 4$, and 97.69% with $F = 5$. Indeed, according to previous studies [12], a fan-out of 14 is required for a gossip protocol with 1000 nodes. This is due to the fact that with a gossip protocol only the out-degree (fan-out) is balanced, while the in-degree (fan-in) may be unbalanced. In contrast, Araneola's in-degrees and out-degrees are balanced as all links in the overlay are bi-directional. As more nodes have a given message, the gossip protocol is more likely to "waste" its gossip on nodes that already have the message than Araneola, and therefore is less effective at spreading the information to additional nodes.

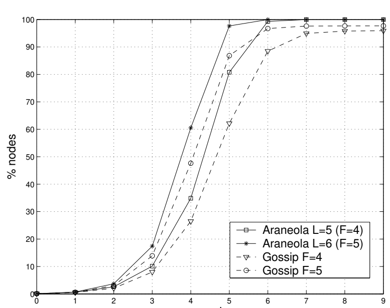


Figure 7. Araneola versus gossip, 1000 nodes.

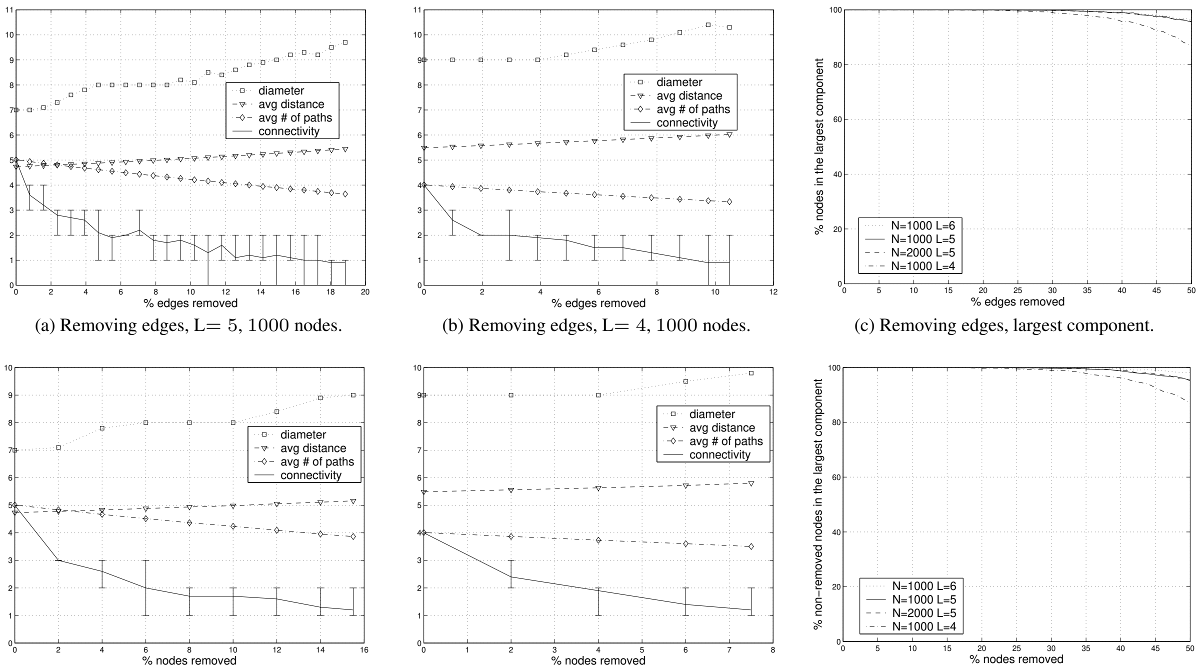
4.2. Fault-tolerance and graceful degradation

We now study the fault-tolerance and robustness of the Araneola overlay. We consider two kinds of failures: communication link failures and node failures. We study the overlay's robustness with an offline analysis of the overlay snapshot obtained at the end of static experiments with 1000 and 2000 nodes. To study communication failures, we remove random subsets of edges from the overlay graph and analyze the resulting graphs. This allows us to predict Araneola's reliability and latency in the presence of message loss. Similarly, we study Araneola's resistance to node failures by removing random subsets of nodes.

We model node and edge failures as *independent and identically distributed (IID)*. For node failures the IID assumption has no significance since the overlay structure is random. Moreover, Bhagwan et al. have found that host failures are indeed independent [2]. For edge failures, the IID assumption fails to capture a situation in which some nodes have poorer links than others. The analysis of non-IID edge failure patterns is an interesting subject for future work.

We first analyze the impact of edge removals on the overlay with L= 5 and $N = 1000$. This overlay has 2547 edges. For each percentage $p \leq 50$ of the edges, we remove 10 different random subsets consisting of p% of the edges from the overlay graph. The overlay becomes partitioned for the first time in one of the ten experiments removing 11% (280) of the edges, and then in one of the experiments removing 15% (380). In both cases, a single node became disconnected from the rest. Fig. 8(a) shows how the removal of up to 19% of the edges affects the overlay's characteristics. For each p in this range, the overlay is partitioned in at most one out of ten experiments in which p% of the edges are removed. We observe that the average diameter increases from 7 to about 8 when 5–10% of the edges are removed, and to 9 when 15% of the edges are removed. The average distance increases more gradually, suggesting that message loss has a very moderate effect on the average latency. The average number of disjoint paths also decreases gradually with the failure rate. The bottom curve illustrates the average connectivity. The bars around each data point show the maximum and minimum connectivity observed in experiments with this p; when the minimum goes does to 0, there was a partition in one of the 10 experiments. We next experiment with L= 4 and $N = 1000$. The overlay is less robust in this case— it partitions in more than 10% of the cases whenever $p > 11\%$. Fig. 8(b) shows the overlay's degradation when up to 11% of the edges are removed.

We next examine how many of the nodes are still connected to each other, i.e., what is the size of the largest connected component in the graph. Fig. 8(c) depicts the average size of the largest connected component after random edge removals for L= $4, 5, 6$ with $N = 1000$ and for L= 5 with $N = 2000$. We can clearly see that the overlay's resilience to the removal of a given percentage of its edges is *completely independent of* N, as is expected in k-regular random graphs [9]: the curves for $N = 2000$ and $N = 1000$



(a) Removing edges, L= 5, 1000 nodes. (b) Removing edges, L= 4, 1000 nodes. (c) Removing edges, largest component.

(d) Removing nodes, L= 5, 1000 nodes. (e) Removing nodes, L= 4, 1000 nodes. (f) Removing nodes, largest component.

Figure 8. Resilience and graceful degradation of Araneola's overlay.

(both with L= 5) are barely distinguishable. As expected, the value of L does impact the overlay's robustness, but the difference between L=5 and L=6 is negligible. Remarkably, for L= 5, after the removal of up to 38% of the edges, 99% of the nodes are still connected to each other, and only 1% of the nodes are partitioned from the rest.

We now turn our attention to node failures. Fig. 8(d) shows how node removals affect the properties of an overlay with 1000 nodes and L= 5 when up to 15% of the nodes are removed. None of the experiments with up to 15% removed nodes resulted in partitions. The overlay becomes partitioned in two of the ten experiments in which 16% (160) of the nodes are removed. This suggests that even if 15% of the nodes running Araneola fail during the brief time interval that it takes to detect and recover from failures (e.g., one minute), Araneola can continue to deliver messages reliably to surviving nodes. As with edge removals, the overlay exhibits graceful degradation: the diameter and average path length increase very moderately, while the average number of disjoint paths moderately decreases. When L= 4, the overlay is half as robust to node failures as with L= 5. It becomes partitioned in two of the ten runs with 8% of the nodes removed. Fig. 8(e) shows the overlay's degradation when up to 7.5% nodes are removed.

In fig. 8(f), we examine the size of the largest connected component that survives following node failures, for L= $4, 5, 6$ with $N = 1000$ and for L= 5 with $N = 2000$. Again, the overlay's resilience shows exactly the same trend with $N = 1000$ as it does with $N = 2000$. This suggests that Araneola's resilience to simultaneous failures of a certain percentage of its nodes is also independent of N. When L= 5, the largest component still includes 99% of the nodes following the failure of up to 38% of the nodes. When L= 4, 99% of the nodes are still connected following the failure of 28% of the nodes. When 50% of the nodes fail, the largest component with L= 5 still includes over 95% of the nodes, and with L= 4, it includes 87%. As with edge removals, increasing L from 5 to 6 achieves only slightly better robustness to node removals when there is an unrealistically high failure percentage.

4.3. Dynamic Evaluation

4.3.1 Methodology

Our model for this evaluation is based on studies of user behavior in multicast groups on the MBone [1], and in file sharing applications [22]. Both of these studies model the join and leave rates of most of the nodes using an exponential distribution. Moreover, both studies observe that a small portion of the nodes have substantially longer life times than others. However, these studies greatly differ in the mean life times they measure: the mean life time measured on the MBone is generally very short, e.g., 7 minutes in a typical multicast session, whereas the average measured life time in a file sharing application is roughly one hour.

We designate a small subset (roughly 7%) of the nodes as *perseverant*. Perseverant nodes are created at the beginning of the experiment and remain active throughout the experiment. Subsequently, every minute, 50 additional (non-perseverant) nodes are awaken, until all nodes (1000 or 2000) are up. Each non-perseverant awaken node join the multicast group (becomes *active*) with probability 0.5. Otherwise, the node remains *inactive*. This gradual joining is modeled after the Berkeley session in [1]. Throughout the experiment, each non-perseverant node once a minute flips a coin with probability λ in order to decide whether to change its state from active to inactive and vice versa. We experiment with values of λ ranging from 0.01 (yielding a mean life time of 100 minutes) to 0.15 (giving a mean life time of 6.7 minutes). As a baseline, we also experiment with $\lambda = 0$, in which case nodes do not change their states. There are roughly 1000 nodes alive at the end of each experiment with $N = 2000$, (and resp., 500 when $N = 1000$), regardless of λ, since the join rate is equal to the leave rate.

4.3.2 Join/Leave overhead

We now measure the cost of constructing and maintaining the overlay in terms of the average number of control messages received by each node, where a control message is any message other than DATA or GOSSIP. In the full paper [19], we also analyze the expected number of control messages incurred by a single join or leave operation when the system is stable. Fig. 9 shows the overhead measured for different values of λ with $N = 1000$ and $N = 2000$. Remarkably, the overhead *decreases* as the rate of such events increases, the only exception occurring when λ increases from 0 to 0.01. Note that when $\lambda = 0$, no leave events occur. The measured average cost per join operation in this case is 15.6, which is very similar to the expected overhead calculated in [19]. The overhead decreases as the churn rate rises because when many join and leave events occur concurrently, their costs can be amortized. E.g., a join event may increase a node's degree while a leave event is reducing it, eliminating the need for correcting the overlay. Furthermore, we observe that the overhead does not increase with N. This is especially impressive given that the overhead for handling joins in structured overlays based on DHTs increases logarithmically with the number of nodes.

4.3.3 Undisrupted service

The second challenge we address is providing an undisrupted service in the presence of churn. For each message m, we define *nodes that are up during m's transmission* to be nodes that have joined at least 12 rounds before m's transmission, and did not leave at least 12 rounds after the transmission. We chose 12 as a very gross over-estimate. In fact, nodes can normally begin to receive messages reliably immediately upon requesting to join. In all of our dynamic experiments, each message is received by 100% of the nodes that were up during its transmission. Moreover, messages are delivered with *the same latency as in static runs*. We illustrate this in Fig. 10 for $N = 1000$; similar results were obtained with $N = 2000$.

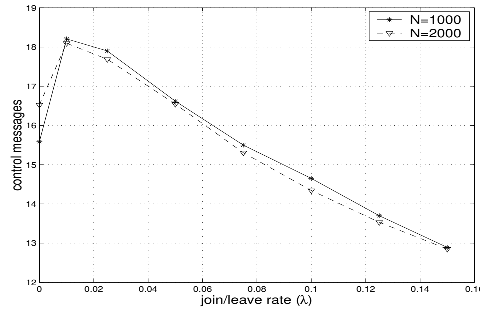


Figure 9. Average cost per join/leave with increasing churn rates for different group sizes, L=5.

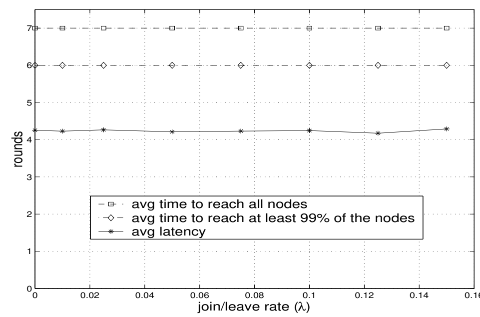


Figure 10. Average latencies in the presence of increasing churn rates, 1000 nodes, L=5.

5 Exploiting Network Proximity

We now preset an extension to Araneola that exploits network proximity by incorporating additional links between nearby nodes. This extension runs in parallel with and independently of the basic overlay construction and maintenance code presented in Section 3. The extension code has two components: (i) a mechanism for locating nearby nodes; and (ii) a connect_nearby task. The first component discovers nearby nodes and stores them in a set named $nearby\_cand$. The second component uses this set.

Generally speaking, Araneola can use a variety of mechanisms for locating nearby nodes. Our implementation does this as follows: at bootstrap time, each node n measures the network-level hop-count distances to the nodes in its local view using the UNIX tracepath utility, and inserts them to the $nearby\_cand$ set in an ascending order of their network-level hop-count distances from n.

The connect_nearby task closely resembles the connect task presented in Section 3, except that no reduction rules are applied and no REDIRECT messages are sent. Specifically, there are three control messages: CONNECT_NEARBY, CONNECT_OK_NEARBY, and LEAVE_NEARBY, which correspond to CONNECT, CONNECT_OK, and LEAVE. In addition, both L and H are replaced by the parameter NB, which is the maximum number of nearby neighbors the node is willing to be connected to, and the $neighbors$ set is replaced by the $nearby\_cand$ set, which holds the node's current nearby neighbors. Note that every node can set its own NB parameter to reflect its available bandwidth. Each CONNECT_NEARBY request is issued to the closest node in $nearby\_cand$, rather than to a random node from the local view.

We evaluate this mechanism over the Internet, running 500 nodes over 25 Planet Lab [21] physical machines, with no two machines at the same site. In all the experiments presented in this section, all the nodes are created simultaneously, and remain up throughout the experiment. Although in principal, each node can choose its own NB parameter, in our experiments, we use the same value of NB for all nodes. We denote an experiment in which each node chooses L random neighbors and NB nearby neighbors as <L,NB>.

It is known that in order to achieve the good properties of k-regular graphs, each node should choose at least three random neighbor [25]. Thus, we run experiments in which each node chooses three random neighbors and three nearby neighbors (<3,3>). We contrast these experiments against experiments in which each node chooses six random neighbors (<6,0>), and against experiments in which each node chooses six nearby neighbors (<0,6>). In addition, we run experiments in which the each node's degree is roughly eight (<3,5>, and <5,3>). Note that all the overlays we experiment with have a low degree compared to those used in previous systems [4, 14, 23]. For each selection of <L,NB>, we run three experiments. In all our experiments, more than 97% of the nodes end up with NB nearby neighbors, and more than 90% of the nodes have exactly L random neighbors; the overall average node degrees in experiments with <3,3>, <6,0>, and <0,6> are almost identical as are those of experiments with <3,5> and <5,3>.

We quantify the effectiveness of our approach by measuring the average number of physical hops that links in the extended overlay traverse. This metric is significant because a smaller hop-count distance implies reduced communication latencies as well as less stress on physical links. The results are summarized in Table 11. The first column shows the percentage of links between two nodes running on the same machine. The second column shows the percentage of short links with a hop-count distance of 3. These are Internet2 links between machines deployed at different sites belonging to the same enterprise. Finally, the third column shows the average hop-count in the overlay. Clearly, as NB is increased at the expense of L, there are more local and short links and the average number of physical hops that each link traverses is reduced.

<L,NB>	% of links on the same machine	% of short links	avg hop count
<3,3>	**34.43**	**15.27**	**5.21**
<6,0>	4.97	6.93	8.69
<0,6>	74.23	3.4	1.88
<3,5>	**51.18**	**12.25**	**3.82**
<5,3>	35.6	10.46	5.54

Figure 11. Hop-count statistics with different selections of <L,NB>.

Having verified that the mechanism achieves its goal, we next check its impact on the overlay's robustness. We repeat the experiments of Section 4.2, i.e., we remove random subsets of edges and nodes from the overlay graphs and measure the sizes of the largest remaining components. The top two curves in Fig. 12 and Fig. 13 are for experiments with <5,3> and <3,5>. These curves are indistinguishable. Slightly below these are the curves for experiments with <6,0> and <3,3>, which are also conjoined. The bottom curve in both figures is for experiments with <0,6>. Remarkably, the robustness of an overlay with <5,3> is almost identical to that with <3,5>, and the robustness of an overlay with <6,0> is virtually identical to that with <3,3>. We believe that this stems from the fact that there is sufficient randomness in the choice of links since: (i) the nodes in $nearb\_cand$ are chosen from the randomized local view; and (ii) each node is connected to at least 3 random neighbors.

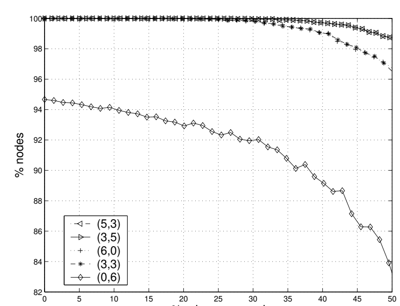


Figure 12. Removing edges, largest component, 500 nodes.

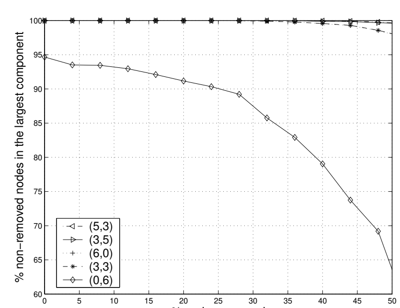


Figure 13. Removing nodes, largest component, 500 nodes.

The curves for experiments with <0,6> show why it is important to choose random nodes as neighbors: in all these experiments, the overlay is partitioned even before we remove any edge or node. Moreover, as the percentage of removed edges or nodes increases, the robustness of the overlay deteriorates much quicker than when random edges are used.

We conclude from the experiments in this section that it is preferable for each node to have three random neighbors, and to allocate the rest of its available bandwidth for communication with nearby nodes.

6 Conclusions

We have presented Araneola, a scalable reliable multi-point to multi-point application-level multicast system for dynamic environments. We have evaluated Araneola over both a LAN and a WAN, and have shown that Araneola is highly scalable. The only aspect of Araneola that varies with the number of nodes is message latency, which increases logarithmically with the group size, whereas Araneola's load, reliability, resilience to message loss, resilience to simultaneous node failures, and overhead for handling join and leave events are all independent of the group size. Araneola can deliver messages with high reliability and bounded latency in the presence of sizable message loss rates, simultaneous failures of a certain percentage of the nodes, and high churn. The failure rates that Araneola can withstand depend on a tunable parameter. As the failure rate increases beyond its expectation, Araneola's reliability degrades gracefully. We have also shown how to extend Araneola to exploit available bandwidth for communication with nearby nodes. Such an approach substantially reduces the communication costs and message latency without hurting the overlay's robustness.

Acknowledgements We thank Ophir Ovadia for his assistance with implementing part of the code. We thank Dahlia Malkhi for helpful comments. We are grateful to the Flux research group at the University of Utah, and especially Leigh Stoller and Jay Lepreau, for allowing us to use their network emulation testbed [24] and assisting us with our experiments.

References

[1] K. C. Almeroth and M. H. Ammar. Collecting and modeling the join/leave behavior of multicast group members in the mbone. In *HPDC*, 1996.

[2] R. Bhagwan, S. Savagen, and G. Voelker. Understanding availability. In *IPTPS*, 2003.

[3] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: a large-scale and decentralized application-level multicast infrastructure. *JSAC*, 2002.

[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM*, 2003.

[7] P. T. Eugster, R. Guerraoui, S. B. Handurukande, A. M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *DSN*, 2001.

[8] J. Friedman. On the second eigenvalue and random walks in random d-regular graphs. Combinatorica, vol. 11, 1991.

[9] A. Goerdt. The giant component threshold for random regular graphs with edge faults. *Theoretical Comput. Sci.*, 259(1-2):307–321, 2001.

[10] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and H. W. O. Jr. Overcast: reliable multicasting with an overlay network. In *OSDI*, 2000.

[11] F. Kaashoek and D. Karger. Koorde: A simple degree-optimal hash table. In *IPTPS*, 2003.

[12] A.-M. Kermarrec, L. Massouli, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[13] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *STOC*, 2000.

[14] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[15] C. Law and K. Siu. Distributed construction of random expander networks. In *Infocom*, 2003.

[16] M. J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *DISC*, 2000.

[17] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC*, 2002.

[18] L. Massoulie, A.-M. Kermarrec, and A. J. Ganesh. Network awareness and failure resilience in self-organising overlay networks. In *SRDS*, 2003.

[19] R. Melamed and I. Keidar. Araneola: A scalable reliable multicast system for dynamic environments. TR CCIT 474, dept. of EE, Technion, March 2004. URL `http://www.ee.technion.ac.il/~idish/Xchange/araneola.ps`.

[20] V. Paxson. End-to-end Internet packet dynamics. In *SIGCOMM*, 1997.

[21] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In Proceedings of HotNets-I, October 2002.

[22] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *MCN*, 2002.

[23] K. Shen. Structure management for scalable overlay service construction. In *NSDI*, 2004.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

[25] N. Wormald. Models of random regular graphs. *Surveys in Combinatorics*, 276:239–298, 1999.