Failure-Tolerant Overlay Trees for Large-Scale Dynamic Networks

Davide Frey, Amy Murphy

Failure-Tolerant Overlay Trees for Large-Scale Dynamic Networks

Davide Frey[1] and Amy L. Murphy[2]
[1]INRIA Rennes-Bretagne Atlantique, Rennes, France, davide.frey@irisa.fr
[2]FBK-IRST, Italy, murphy@fbk.eu

Abstract

Trees are fundamental structures for data dissemination in large-scale network scenarios. However, their inherent fragility has led researchers to rely on more redundant mesh topologies in the presence of churn or other highly dynamic settings. In this paper, instead, we outline a novel protocol that directly and efficiently *maintains a tree overlay in the presence of churn. It simultaneously achieves other beneficial properties such as limiting the maximum node degree, minimizing the extent of the tree topology changes resulting from failures, and limiting the number of nodes affected by each topology change. Applicability to a range of distributed applications is discussed and results are evaluated through extensive simulation and a PlanetLab deployment.*

1 Introduction

Tree structures have long been known as very efficient solutions for managing data dissemination in large scale distributed systems such as content-based publish-subscribe, peer-to-peer query services and application level multicast (ALM). In content-based publish-subscribe systems, nodes are arranged in an unrooted tree and each maintains routing information [5]. Subscriptions propagate along the tree to all nodes, establishing reverse routes to recipients, which are later followed by published messages. The tree allows efficient routing and distributed maintenance of subscription tables. A *p2p* that sends search queries to all nodes may also be implemented on a tree topology although without the need of subscription tables. Finally, *Application level multicast* (ALM) is often tree based, generally building a single tree for each multicast topic with a single source that generates all messages.

The achieved efficiency, however, comes at the cost of an inherent fragility resulting from the presence of only a single path between any pair of nodes. For this reason, the need to support large environments with nodes that join and leave at unpredictable moments has lead researchers and designers to topologies that offer a greater degree of redundancy, e.g., mesh overlays. Nevertheless, for efficient data dissemination, such systems often revert to building temporary tree-based distribution structures on top of these mesh overlays. Such structures remain fragile, and if one tree link breaks, a new structure must be generated from scratch. Another option is to directly build and maintain a tree overlay, as pursued by research in ALM [10, 3]. sHowever these mechanisms fall short in dynamic scenarios, for example due to their use of global network knowledge, or serialization of children node re-attachment after parent failure.

In this paper, we instead demonstrate that we can have *both* resiliency in dynamic networks and efficient data distribution without the fragility of a tree built over a mesh and without the global or serialization assumptions of other solutions.

The concrete requirements of our tree maintenance approach arise from the applications outlined earlier: publish-subscribe, *p2p*, and ALM. First, our approach controls *node degree*, properly supporting node processing and bandwidth constraints and preventing the presence of too many nodes with very low degree, a situation that can increase overall latency. Second, these applications benefit from an overlay that does not incur significant topology changes as a result of churn, instead localizing changes around the failure. This allows minimal changes at the higher layers and enables localized recovery mechanisms for messages lost during topology changes.

Based on these requirements, Section 2 presents a novel, efficient application-tunable protocol for maintaining a tree overlay in the presence of churn. The protocol exploits a very simple structure and offers two main contributions. First, it defines a set of repair strategies that allow it to target application requirements such as those outlined above. Second, it exploits the novel concept of real-valued depth to implement a cost-effective cycle avoidance mechanism using only local information. This effectively limits the number of nodes that update their state after a topology change.

Our final contribution, in Section 3, is a comprehensive evaluation using both simulation with Omnet++ and experiments over PlanetLab. The paper ends with a comparison to related efforts in Section 4 and brief concluding remarks.

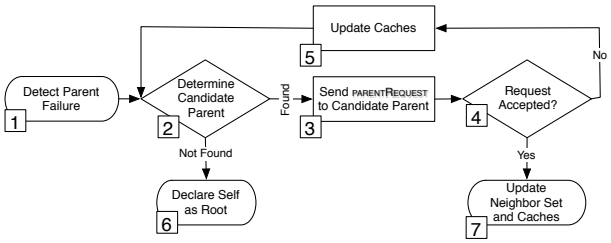


Figure 1: Protocol overview for locating a new parent after the current parent fails.

2 Maintaining a Tree in a Dynamic Setting

Our overlay maintenance protocol arranges nodes in a rooted tree and promptly reacts to node disconnection to maintain tree connectivity. The root is selected during protocol operation and all nodes, including the root, are allowed to join and leave the overlay at any time without explicit departure announcements. The choice of a rooted structure makes our protocol suitable for systems that require rooted or unrooted topologies. We further assume that all nodes are willing to serve as intermediate nodes in the routing tree. A system with heterogeneous nodes would need to make minor modifications to distinguish between routing-enabled nodes and client-only nodes.

The protocol is outlined in Figure 1. Nodes monitor the state of their neighbors by tracking application-level traffic or sending beacons every T_b. As soon as a node detects the failure of its parent in the tree (step 1), it actively identifies a candidate new parent (step 2). When it locates one, a PARENTREQUEST is sent (step 3). The candidate then verifies that it can safely accept the request without violating acyclicity and while satisfying soft requirements such as node degree constraints. If so, the process terminates with the tree reconnected, otherwise the process repeats. In extreme cases, a new parent will not be found, and the process ends with the node declaring itself a root.

2.1 Identifying Candidate Parents

The first action taken when a node detects the failure of its parent is the identification of good candidate parents (step 2 in Figure 1). This selection process must both result in overlay topologies with good tree properties, e.g., balanced node degree, and yield reasonable tree maintenance procedures, e.g., localized changes. We achieve these goals with a combination of multiple, independent strategies, outlined next. The mechanism to combine them into a coherent protocol follows in Section 2.1.2.

2.1.1 Repair Strategies

New candidate parents are identified through a set of tree repair strategies summarized in Table 1, each of which is supported by one or more caches containing potential new parents. The advantages in the third column of the table highlight the desired property each strategy addresses, ranging from localizing tree updates to maintaining soft requirements on node degree. Additional constraints such as bandwidth and physical node location can easily be incorporated into this model of caches and strategies.

The table also outlines the update mechanism for each cache, with most being updated by messages exchanged during the tree repair process itself. Only two caches, Regional and Global, are proactively maintained as described in the following. It is further worth noting that although cached information ages, the protocol is designed such that old data both remains useful and cannot affect correctness.

We begin with the details of each strategy.

Repairing Failures Locally: Regional Strategy. Our first repair strategy aims to limit the impact of overlay maintenance on the higher layers by localizing topology changes to a small region of the overlay near the disconnected node. In publish-subscribe, for example, topology changes cause subscription tables to change. By localizing topology updates, a small fraction of nodes are affected, reducing the reconfiguration cost [12]. With ALM and *p2p*, finding a new parent near the failed node facilitates the implementation of mechanisms to recover application messages lost during topology changes.

Our Regional strategy proposes local candidate parents by maintaining an *Ancestor Chain* and a *Sibling Set*, collectively referred to as the *Regional* cache. The Ancestor Chain contains l_a nodes starting with the parent and continuing toward the root while the Sibling Set contains all nodes that are also children of the parent node. When this information changes as a result of reconfiguration, the parent updates its children either by piggybacking the information on an application message or sending an explicit update.

When invoked, the regional strategy selects either a sibling or an ancestor. If all nodes use only the Ancestor Chain, the topology converges towards a *star* in which a few nodes near the root have very high node degree. Conversely, if all nodes (except one) choose siblings, the resulting topology resembles a line. We avoid both extremes by selecting either option with a given probability p_{star} or $1 - p_{star}$.

Locating Nodes with a Low Degree: Downstream Strategy. Preventing nodes from reaching high degree is one of our target soft requirements. However, repeated selection from the Ancestor Chain tends to identify new parents close to the root causing node degrees to increase. This motivates searching in the opposite direction, namely farther from the root, to find nodes more likely to have smaller degree.

The Downstream strategy achieves this by exploiting a Downstream cache populated with the descendants of candidate parents that have reached too high a degree. To make

Cache	Update Mechanism	Advantage	Disadvantage
Regional	By parent when regional nodes change	Localizes tree topology updates	Requires up to date information
Downstream	With data from candidate refusing for MAXDEGREE	Avoids high degree nodes near root	May create long, thin branches
Upstream	With data from candidate refusing for MINDEGREE	Finds nodes with lower depth higher in tree	May yield high degree nodes near root
BreakMaxDegree	With reference to candidate refusing for MAXDEGREE	Avoids tree split at expense of soft limit	Increases overhead per node
BreakMinDegree	With reference to candidate refusing for MINDEGREE	Avoids tree split at expense of soft limit	May create long, thin branches
Global	Proactively maintained by each node	Useful after cluster of failures	Does not preserve locality

Table 1: Summary of caches, their update policies and properties.

this possible, we assign each node a maximum prescribed node degree, or MAXDEGREE limit. A candidate parent may refuse a connection request if accepting it would cause it to exceed its MAXDEGREE value. When doing so, the candidate parent also selects a subset of l_d node identifiers from its downstream neighbors and sends it to the requesting node, allowing it to continue its parent search downstream to locate a parent with a suitable degree.

Locating Non-Leaf Nodes: Upstream Strategy. In addition to preventing nodes from reaching too high a degree, we also want to avoid the opposite extreme: long chains of nodes connected in a line. This is likely to arise if leaf nodes are continuously selected as candidate parents, possibly as a result of frequent use of the Downstream strategy. To limit these line-like configurations we allow candidate parents to refuse a request, while simultaneously suggesting other candidates *higher* in the tree.

To achieve this, we introduce a MINDEGREE limit. When a candidate parent whose degree is strictly below this limit (e.g., a leaf node) receives a PARENTREQUEST, it may refuse while providing the requesting node with a copy of its Ancestor Chain. The requesting node collects these *upstream* candidates in an Upstream cache to be used in later reconnection attempts.

Softening Degree Constraints: BreakMaxDegree and BreakMinDegree Strategies. Even though maintaining reasonable degree is desirable, it should not be done at the expense reconnecting the tree, or invalidating more important properties such as acyclicity or reconfiguration locality.

The *BreakMaxDegree* and *BreakMinDegree* strategies consider this, making it possible to force a candidate parent to accept a PARENTREQUEST even if the limits are broken. When the corresponding strategies are activated, candidate parents refusing a request because of a degree constraint are added to the BreakMaxDegree or BreakMinDegree caches. Subsequent PARENTREQUEST messages to these candidates are sent with a flag set to require acceptance regardless of the current degree.

Recovering from Catastrophic Failures: Global Strategy. The repair strategies described thus far require bootstrapping by first contacting at least one node in the regional cache to start filling the other caches. However, there may be cases in which all candidates in the regional cache are unreachable. In these cases, we exploit a Global cache that may reference nodes anywhere in the overlay.

Trade-offs exist between the Global cache size, the probability of stale references, and the cache update overhead. Ultimately the choice is middleware and application specific. For this paper, we use a very simple peer-sampling mechanism [16] in which nodes periodically, every T_g, exchange a random subset set of references extracted from their caches in a push fashion. Each update message is sent to a subset of nodes extracted from the same caches. A node receiving an update inserts the references in its Global cache. In addition, nodes periodically ping the nodes in their Global cache to verify their reachability.

It should be noted that, as with the other caches, the Global cache need not contain up-to-date information. Thus, the frequency of updates and ping probes can remain very low, limiting bandwidth consumption.

2.1.2 Combining Recovery Techniques

While each of our strategies targets a particular desirable tree characteristic as shown in Table 1, the protocol behavior as a whole depends on how the strategies are applied. We define a *protocol* instance as a sequence of strategies and whether or not soft constraints like the degree limits are enabled. The candidate parent is selected from the first strategy in the sequence whose cache is non-empty.

In theory, the strategies can be combined in arbitrary ways, however some combinations make more sense. For example, in most applications, nodes over the MAXDEGREE limit are less desirable than low-degree nodes, hence BreakMaxDegree should be applied only after BreakMinDegree has failed. Also, because locality is important, Global should only be applied after all local repair attempts have failed, that is after Regional, Upstream, and Downstream.

In the protocol instances studied in our evaluation and summarized in Table 2, we always include the Regional, Global, and BreakMaxDegree strategies. Further, the minimum degree limit is enabled only in the protocol instances that contain the BreakMinDegree and Upstream strategies, while the maximum limit is always enabled.

2.2 Determining Parent Viability

The previous section outlined motivation to allow a candidate to refuse a PARENTREQUEST due to a constraint on node degrees, however, a parent may also refuse in order to avoid the creation of a cycle. This section outlines how cycles can be avoided using only local information.

Integer Depth One solution to tree maintenance appears in the mobile ad hoc wireless network protocol MAODV [22], which builds a multicast tree among nodes. Movement causes connectivity to change, mimicking some aspects of our target environment. In MAODV, each node knows its *integer hop count*, or depth, from the root. When its parent fails, it searches locally for a node with a *strictly lower* depth. Making this node its parent cannot create a loop.

While this solution is correct, it does not scale to our large target setting. For example, to maintain integer depth, the root node periodically sends a message down the tree, updating all depth values. Additionally, a node changing its depth after finding a new parent propagates this value immediately to all its descendants, updating their depths. While this is acceptable in the wireless network where the number of nodes is small, such global actions to maintain depth values are unacceptable in large-scale networks. Our solution modifies depth values to use real values instead of integers. This simple change eliminates most global operations.

While our goal is to maintain a single tree, it is possible that a node will declare itself to be a root node even though a potential parent exists in the system. In this case, we must recognize when two trees have been formed and merge them with low overhead. MAODV also offers such a service, however it involves several nodes, namely the roots of both trees and the nodes that recognized the existence of two trees. Our protocol, instead, uses a notion of tree identifier to distinguish between disconnected trees and only involves one root node and an arbitrary node in the other tree. As described below, this also yields protocol solutions without far-reaching consequences.

Real-valued Depth The key to our proposal is to remove the *plus-one* restriction on the depths of parent and child, and instead require only that a child's depth must be *greater-than* that of its parent, as shown in Figure 2.

Real-number depths add flexibility over integer hop counts. Consider a case where a node attempts to repair the failure of its parent by connecting to one of its former siblings, e.g., the node with depth 3 tries to connect to the one with depth 2.6. With integer hop count this connection would be forbidden even though no cycles are created because the requesting node is at the same distance from the root as the candidate parent. Instead, with real-number depth, the candidate parent with depth 2.6 may already have suitable depth, allowing the new connection. Moreover, if its depth is too large, it may be able to decrease it, making it less than that of the requesting node but still greater than that of its own parent. The figure illustrates this option applied to a non-sibling node: the node with depth 3.1 can decrease its depth to 2.9, making itself a valid parent for the node with depth 3. Such depth value decreases allow candidate parents to accept connection requests that would otherwise be rejected.

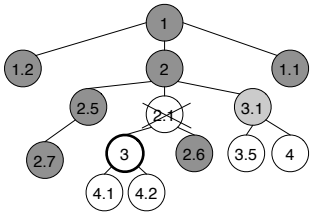


Figure 2: Sample depth levels. All shaded nodes are candidate parents for the node with depth 3, including the one whose depth is 3.1 if it reduces this value.

To guarantee that the tree remains consistent, we allow nodes only to decrease their depth values, but never to increase them.[1] This has two important consequences. First nodes need not coordinate with their children when decreasing their depth. Second, nodes need not have perfect information about the latest depth of their parent, but instead can safely compare their own depth against a previously cached parent depth. If their depth is greater than the cached value, then it is also greater than the current depth. This guarantees correct protocol behavior even if neighboring nodes concurrently modify their depths.

A node learns the depth of its parent when it first connects to it. In addition parent nodes piggyback their depth information on all control messages. Finally, it is worth noting that a connection to a new parent does not trigger any message to reset the depth values of the descendants, as required in the MAODV scheme.

Tree Identifier While depth allows us to guarantee that a loop-free tree is maintained, *tree identifiers* enable nodes to recognize they are members of separate trees (i.e. trees with different roots), and thus trigger tree merging.

Our goals here are to prevent cycles and to avoid requiring coordination among all nodes in the overlay network. We achieve this by defining an ordering relationship between tree ids, denoted $\rightarrow$. The directionality implies that the root of the left tree can connect to any node in the right tree, but not vice versa. Establishing this connection involves only one root and one node in the other tree.

The root establishes the tree identifier, and we guarantee that if the root changes, the id also changes. Tree ids are propagated from parent to child, and must remain consistent with respect to the $\rightarrow$ relation. Satisfying this constraint drives two aspects of our protocol. First candidate parents evaluate connection requests using a combination of id and depth. Second, because id changes propagate from the root node to the rest of the tree and $\rightarrow$ must always hold from a child to a parent with different ids, a node with tree id I_1 can only change its tree id to a new value I_2 such that $I_1 \rightarrow I_2$. The arrow relation can be thought of as a representation of the connectivity from child to ($\rightarrow$) parent.

To allow these parent to child updates, we express a tree identifier as a sequence of node identifiers and define the $\rightarrow$ relationship as follows: $I_1 \rightarrow I_2$ if and only if either (i) I_2 is a longer sequence than I_1; or (ii) I_1 and I_2 are sequences

[1] Depth may be increased when changing tree id as we discuss later.

of the same length and $I_1 > I_2$ according to standard string comparison. We also define a set of rules for the creation and propagation of tree ids. First, the initial tree id of a detached node is an empty sequence, allowing it to become a child of any node with a non-empty tree id. Second, root nodes generate new tree ids by appending their node identifiers to their current tree id. This always happens when a node becomes the new root of a tree, but can also be done spontaneously as we discuss later. Third, each time a node updates its tree id (because it is joining or creating a tree), it must propagate the new value to all descendants, allowing all nodes in the tree to eventually acquire the same tree id. Finally, a node connecting to a new parent with a different tree id accepts and propagates the tree id of that parent.

Exploiting Tree ID and Depth Our protocol exploits both tree identifiers and depth to prevent the creation of cycles when adding a link between two nodes with an extension of $\rightarrow$ to include both components: given two pairs, (I_A, D_A) and (I_B, D_B), $(I_A, D_A) \rightarrow (I_B, D_B)$ holds if and only if $(I_A \rightarrow I_B) \vee (I_A = I_B \wedge D_A > D_B)$.

Further, for every parent p and child c, we enforce the invariant that $(I_c, D_c) \rightarrow (I_p, D_p)$. This allows a node to determine if it can safely accept a parent request. If not, but the tree ids are the same and the candidate can *safely* decrease its depth to satisfy the invariant, it does so and accepts the request. Otherwise the request is rejected. Decreasing depth is safe except when the candidate is itself searching for a new parent. In this case, the candidate replies with a special *busy* refusal message, informing the requesting node that it may later resubmit its request. The process is summarized as follows for a node, self, with parent, parent, which receives a PARENTREQUEST from a node, req:

1. **if** $(I_{\text{req}}, D_{\text{req}}) \rightarrow (I_{\text{self}}, D_{\text{self}}) \wedge softConstraintsMet$
2. accept request
3. **else if** self is not looking for a new parent
4. **if** $(I_{\text{req}}, D_{\text{req}}) \rightarrow (I_{\text{parent}}, D_{\text{parent}}) \wedge softConstraintsMet$
5. let $D_{\text{self}} = D'_{\text{self}}$ s.t. // decrease depth
$(I_{\text{req}}, D_{\text{req}}) \rightarrow (I_{\text{self}}, D'_{\text{self}}) \rightarrow (I_{\text{parent}}, D_{\text{parent}})$
6. accept request
7. **else** refuse request as *invalid*
8. **else** refuse request as *busy*

It is worth noting that this is the only case when a candidate request is rejected due to concurrent activity. In normal operation, simultaneous operations and requests proceed in parallel without conflict.

Decreasing depth values unfortunately causes node depths to converge toward that of the root. To limit this, the root may occasionally select and propagate a new tree id I' such that $I \rightarrow I'$. Upon receipt, each node updates its tree id and sets its depth to that of its parent plus a random number. This is allowed because the $\rightarrow$ definition gives higher priority to tree id. Thus, by properly selecting I', the relation is maintained during propagation. Although this update involves all overlay nodes, depth redistribution is rare.

2.3 Declaring New Roots and Merging Trees

Our recovery strategies provide references to candidate parents. This also allows our protocol to recover from root failures. In detail, when the root fails, its children connect to each other using their Sibling Sets; however, one of them will fail, declaring itself a new root.

New roots may also be created due to a physical partition in the network, or very high failure rates. In these cases, the overlay may temporarily become partitioned in multiple trees. Our protocol effectively merges these trees with the same mechanism it uses to locate new parents. Each root periodically sends a PARENTREQUEST message to the nodes in its Global cache, searching for a parent.

3 Experimental Evaluation

Our evaluation combines a detailed OmNet++ [25] simulation study with a performance analysis in PlanetLab, demonstrating the ability of our protocol to efficiently maintain a connected, overlay tree with controlled node degree.

3.1 Simulation Analysis

To provide realistic results in both simulation and deployment, node connections and disconnections are based on a data trace of node online times measured in a real Gnutella network [23]. We also stress our system with a *Catastrophic* scenario in which a large percentage of the nodes simultaneously fail.

All experiments use the following parameters. The MAXDEGREE limit is $d_{max} = 5$, to yield trees of reasonable depths with the considered number of nodes. MINDEGREE, when used, is $d_{min} = 2$ to discourage the selection of leaf nodes as parents. Nodes may retry connecting to a previously contacted, *busy* candidate at most 3 times. The Ancestor Chain is $l_a = 3$ nodes long and the probability of choosing a regional candidate from the Ancestor Chain or the Sibling Set is $p_{star} = 0.5$. We experimented with other values, however a longer Ancestor Chain or different probabilities did not significantly affect performance. Where not otherwise specified, the Global cache contains $l_g = 10$ entries, while the number of references in refusal messages used to update the Downstream cache is $l_d = 3$. Global cache update messages are exchanged every $T_g = 250s$. Nodes purge failed entries from their Global caches by pinging each node with the same interval. These values strike a balance between cache and messages sizes and the need to reconnect the overlay effectively.

Our analysis evaluates the most representative protocol instances shown in Table 2. Our primary point of comparison is Overcast [15], a system for reliable multicast that

Abbreviation	Sequence of caches
RMG	Regional, BreakMaxDegree, Global
RDGM	Regional, Downstream,Global, BreakMaxDegree
DRGM	Downstream, Regional, Global, BreakMaxDegree
RUmDGM	Regional, Upstream, BreakMinDegree, Downstream, Global, BreakMaxDegree
DUmRGM	Downstream, Upstream, BreakMinDegree, Regional, Global, BreakMaxDegree
Overcast+G	Ancestor Chain, Global— no degree constraints

Table 2: Protocol instances. Those without BreakMinDegree do not enforce MINDEGREE.

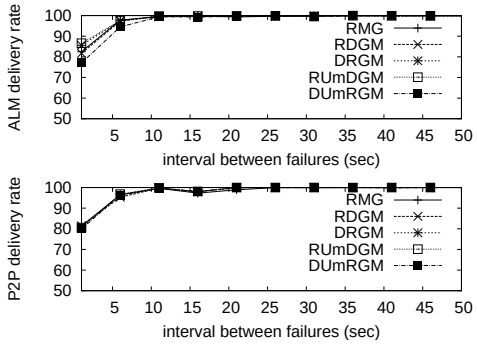


Figure 3: Delivery rate for *ALM* (top) and *p2p* (bottom).

maintains an overlay tree using a strategy similar to our Ancestor Chain. Our simulations show that Overcast alone is unable to maintain a connected overlay in the presence of significant churn; therefore, when required, we compare our solution against a protocol, Overcast+G, that augments the original Overcast with our Global strategy.

3.1.1 Evaluating Applications

The first goal of our evaluation is to determine the impact of our protocol on the applications it supports. In short, our results show that our protocol is able to provide applications with a stable overlay tree with a churn rate up to one failure every few seconds. This stands in sharp contrast to the behavior of Overcast, which, without the use of the Global cache, cannot maintain a connected overlay with even as few as one failure every 180 seconds. For example, an average Overcast run with 2500 nodes created as many as 430 trees at the end of 60 hours of simulated time. Instead, no permanent partitions were recorded by our protocol.

Message Delivery As a first measure of the stability of our tree overlay, we evaluate the impact of our protocol on the delivery rates of our sample applications *p2p* and *ALM*. In the first, each node sends a message reaching all other nodes every 100s, while in the latter only the root send one message every 10s. If the root fails, another node takes its place and begins sending.

Figure 3 shows the average delivery rates with up to one failure per second in a network of 1500 nodes. Tree connectivity is recovered with minimal loss of application messages. With up to one failure every 5 seconds, all instances lose less than 5% of the messages in both applications. In more dynamic scenarios, *ALM* exhibits slightly lower performance due to the presence of a single message source. In *p2p*, instead, messages originate from sources anywhere in the tree thus increasing the percentage of messages that can reach at least some nodes. Confidence intervals are not shown as they are consistently smaller than 5% with more than one failure every 5 seconds and smaller than 1% with lower failure rates.

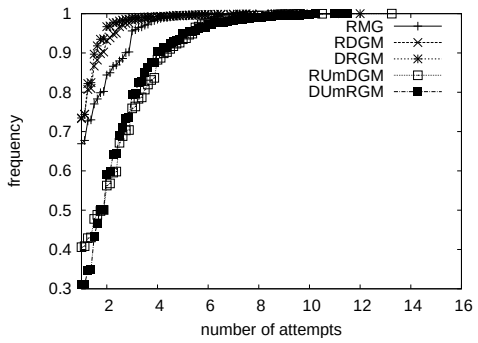


Figure 4: Cumulative distribution of number of candidates contacted for each failure (*Gnutella*)

Recovery Delay. The ability to maintain high delivery rates in the presence of high churn is a direct consequence of our protocol's ability to recover from failures in a timely manner. To evaluate this, we considered a measure of recovery delay expressed as the number of candidates contacted by nodes before locating suitable new parents. Figure 4 shows the cumulative distribution of such delays obtained in *Gnutella*. The protocol promptly recovers failures, with 95% of recoveries completed with 3 to 5 attempts.

Interestingly the best performance in terms of recovery delay is achieved by DRGM. The use of Downstream provides a quick means to react to connection refusals due to MAXDEGREE: repairs complete after contacting only 4 candidates in over 98% of the cases. Only slightly worse is RDMG, which activates the Downstream strategy after Regional, yielding better locality but a longer repair.

The worst performance in terms of recovery delay is by RUmDGM, in which nodes still contact less than 8 candidates in 98% of the cases. This decreased performance is due primarily to the MINDEGREE limit, which together with the Upstream and BreakMinDegree strategies, reduces the number of 2-degree nodes at the expense of a less local and longer reconfiguration. It is worth observing that the cost associated with the MINDEGREE limit depends on the placement of the BreakMinDegree strategy in the priority sequence: the lower the priority, the longer and less local the recovery process. To confirm this, we also experimented with RDUmGM and RDGUmM. These instances have a longer recovery delay, but improve the distribution

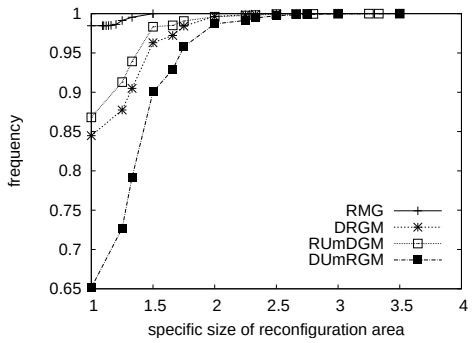


Figure 5: Cumulative distribution of the reconfiguration area (*Gnutella*)

of node degrees with respect to RUmDGM.

Our protocol is also able to improve degree distribution without significantly increasing latency. In the previous discussion, we noted that DUmRGM achieves very good performance in terms of node degree, and Figure 4 shows that its latency is comparable to that of instances that do not enforce MINDEGREE. The reason is that repairs with a high latency caused by the presence of the MINDEGREE limit are balanced by the quick repairs resulting from the high priority given to the Downstream strategy.

Recovery Locality. In addition to quickly restoring connectivity, our protocol maintains *topology stability* across reconfigurations. To measure this, we define *reconfiguration area* as the union of paths in the new tree from all children of a failed node to their former grandparent. Smaller area implies better locality and thus a more stable topology. To remove variability due to the number of children of a failed node, we normalize the size of the reconfiguration area by this value, obtaining a specific reconfiguration-area.

Results for the *Gnutella* scenario, in Figure 5, highlight the ability of our protocol to localize changes. In 4 of 5 protocol instances, over 85% of recoveries have a specific reconfiguration-area size of 1, meaning nodes always find a new parent among the neighbors of a failed node. RMG yields the best performance as nodes are almost always able to reconnect to their grandparents or to their siblings at the cost of increasing their node degrees. On the other hand, RDGM, DRGM, and RUmDGM keep all nodes within the degree limit with only slightly lower locality. The relative worst performance is achieved by DUmRGM, where the non-local effects associated with the Downstream strategy and the MINDEGREE limit combine. Nevertheless, the specific reconfiguration area is still less than 2 in 95% of the cases.

We also tested the performance of instances based on integer depth. Results show the use of real numbers also contributes to good locality: DUmRGM recovers failures with a specific reconfiguration area of 1 in 65% of the cases

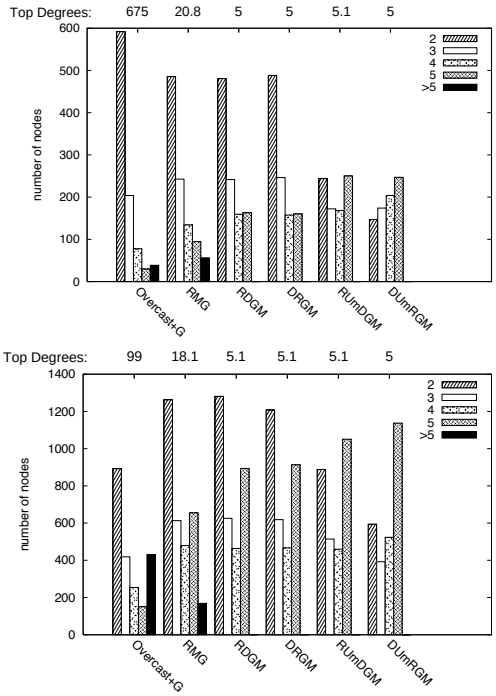


Figure 6: Node degree distributions in *Gnutella* (top) and *Catastrophic* (bottom)

with real numbers but only in 50% of the cases with integer depth. Finally, we also evaluated the size of the reconfiguration area with each of the failure frequencies considered in Figure 3. Results show that the size of the reconfiguration area is not significantly affected by the amount of churn.

Managing Node Degree. A further aspect of our protocol that has a direct impact on the application is its ability to control node degree, and hence the load on any single node. Figure 6 shows the node degree distribution at the end of a sequence of reconfigurations for our reference protocol instances and "Overcast+G". Although, with the addition of our Global strategy, "Overcast+G" reconnects the tree, it does so at the cost of very high node degrees, e.g., up to 675 in *Gnutella* and 99 in *Catastrophic*.

On the other hand, all of our protocol instances effectively manage node degree while keeping the overlay connected. Even RMG, which may break the MAXDEGREE limit early in the recovery process, yields a degree distribution with a small fraction of nodes above the degree limit and a top degree node with an average of 20.8 neighbors in *Gnutella* and 18.1 in *Catastrophic*. Although above the limit, these are reasonable values for most applications.

Degree distribution improves further in protocol instances that postpone the use of BreakMaxDegree. In RDGM and DRGM, the combination of Downstream and Global guarantees connectivity without exceeding the degree limit in all practical cases. The limit is only occasion-

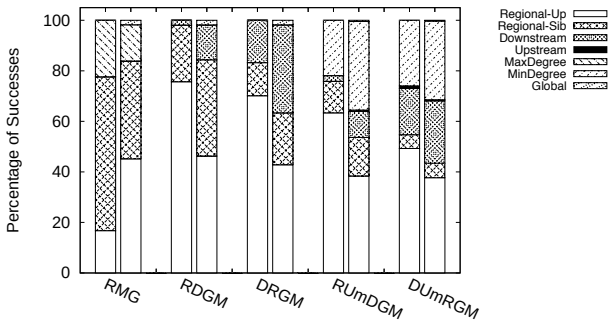


Figure 7: Percentage of reconnections achieved by each strategy in *Gnutella* (left) and *Catastrophic* (right).

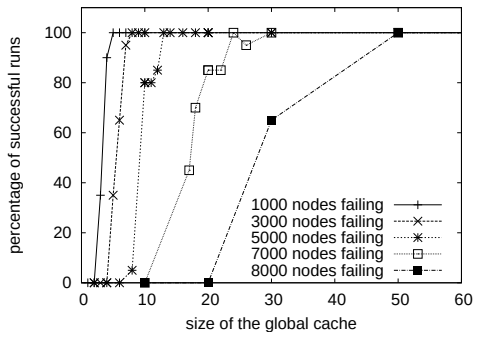


Figure 8: Impact of global-cache size on *Catastrophic* failures

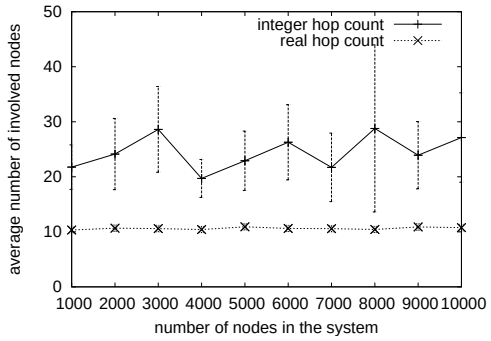


Figure 9: Integer vs real-valued depth: average number of nodes involved in a repair process.

ally broken with very high failure frequencies.

Keeping node degree below the MAXDEGREE limit is, however, only half of the picture. By enforcing the MINDEGREE constraint, our protocol can also shift degree distribution towards MAXDEGREE. Both instances exploiting MINDEGREE, DUmRGM and RUmDGM, significantly reduce the number of two-neighbor nodes. DUmRGM performs better due to its more frequent use of Upstream, as evidenced in Figure 7.

As a further point of comparison, we tested our protocol instances using *integer-valued depth*. In this case, DUmRGM reaches a top degree of 7 (5 with real numbers), while RMG behaves like Overcast+G reaching a top degree of 520 (20.8 with real numbers).

3.1.2 Evaluating the Protocol

So far we have demonstrated our protocol's ability to sustain the correct operation of applications. To gain a better understanding of how this is achieved, we next concentrate on the characteristics of our protocol.

Impact of Recovery Strategies We first evaluate how each of our strategies contributes to reconnection. Figure 7 shows that, for both the *Gnutella* and *Catastrophic* scenarios, all protocol instances repair the majority of failures using the Regional strategy. This is an encouraging result as the Regional strategy represents the best option both in terms of delay to reconnect and reconfiguration locality.

The figure also shows that the Global strategy is almost never used in *Gnutella* and even in *Catastrophic* it is used in less than less than 5% of repairs. However, it would be wrong to conclude that the Global strategy has little importance in the protocol. Its presence is key to keeping the overlay connected in the presence of high churn.

To better understand the impact of the Global strategy on performance, we examined the correlation between the size of the Global cache and the number of successful repairs in the *Catastrophic* scenario. Specifically, in Figure 8, we show the ability of the RMG instance to reconnect the tree in a network of 10000 nodes with cache sizes that vary from 2 to 60 nodes. We also vary the degree of the "catastrophe" by increasing the number of failing nodes. The most interesting result is that very small global caches of 15 nodes are sufficient to reconnect the tree in all runs, even when up to half the nodes fail. This is in line with the current literature on gossip-based peer-to-peer overlays [16].

Benefits of Real-Valued Depth We introduced real-valued depth as an efficient mechanism to prevent the creation of cycles. To demonstrate the gains over integer depths, we compare our RDGM protocol against an MAODV-like hop-count-based protocol. We consider the average number of nodes affected by the repair process, namely those that update their neighbor sets, caches, or tree-id/depth pairs as a result of the repair process.

Figure 9 presents the results of random failures in networks of varying size; each point represents an average over 1200 reconfigurations in 40 runs and is depicted together with its 95% confidence interval. Results confirm the effectiveness of real valued depth in limiting the average number of nodes involved in a reconfiguration. Moreover, the jagged nature of the integer-depth line and the size of the corresponding confidence intervals also illustrate the very

high variance exhibited by the performance of an MAODV-like approach. This is because, with integer depth, the number of contacted nodes strongly depends on the distance of failed nodes from the root. When a node fails, all the nodes in its subtree may need to update their depth values. On the other hand, with real-valued depth changes are always restricted to a very small portion of the overlay. This is confirmed by the maximum values recorded over all reconfigurations: 76 contacted nodes in the case of real-valued depth, and over 2000 when using integer depth.

Cost of Tree Maintenance. We measure cost in terms of messages in the system. For clarity, we discuss the impact of beacons to detect failed parents separately.

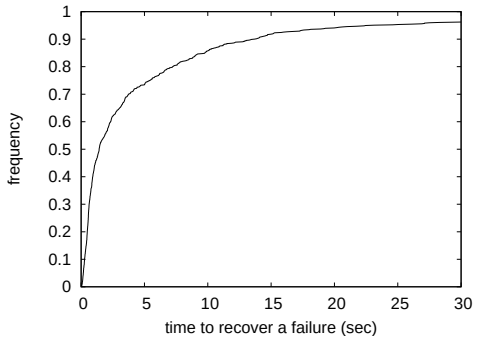
Results show that our protocol has low overhead. In *Gnutella* with an average of one reconfiguration every 5 seconds, our protocol always remains well below 0.2 messages per node per second, with only minor fluctuations, while in *Catastrophic*, the result is more interesting. In a stable topology, the protocol requires less than 0.05 messages per node per second, mostly to refresh the contents of Global caches. When the failure of 2500 nodes occurs, the number of messages spikes. However, even at its peak, traffic remains below 5 messages per node per second, a very good result when compared with the performance of protocols that build trees as subsets of mesh overlays [6].

Finally, in the absence of application traffic, cost is dominated by failure-detection beacons. With a fairly high frequency of 1 beacon per second, the average number messages per node per second is approximately 2. This is consistent with the fact that the average degree of nodes in the tree (including leaves) is 2.

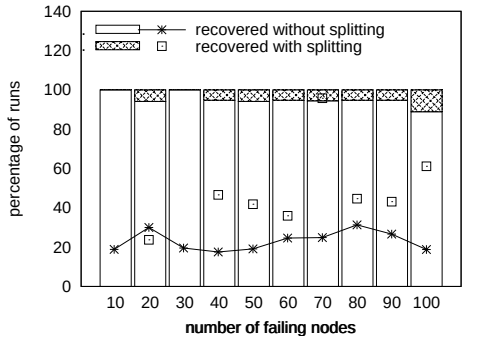
3.2 PlanetLab Deployment

To complement the simulation analysis, we implemented the DUmRGM protocol instance as well as the *ALM* and *p2p* applications and deployed them on approximately 130 PlanetLab [17] hosts. The choice of DUmRGM allows us to test all of our strategies and enables direct comparison with our simulation results. For each experiment, node behavior was determined by the *Gnutella* trace. A connect event in the trace corresponds to the initialization of a new node on an available host; if no host is available, the event is skipped. After its up-time, the application is abruptly terminated, leaving the node available for a new instance. When starting a node, we initialize its global cache with a set of nodes currently in the network.

In general, experimental results confirm those from simulation. When considering how caches contribute to reconfiguration (Figure 7), the main differences observed in PlanetLab are a slight decrease in the success rate of the Regional strategy and a corresponding increase in the success rates of Global and Upstream. The most important aspect



(a) Parent recovery latency, *Gnutella*



(b) Tree repair latency, *Catastrophic*

Figure 10: Parent recovery and tree repair latencies.

of our PlanetLab experiments is the demonstration of the efficiency of our protocol in a real-network environment, measured through the latencies of tree repair and message dissemination.

Repair Latency In simulation, we reported latency in Figure 4 as the number of candidates contacted by a node to find a new parent. Here we report the *time* between when a node detects the failure of its parent and when it is able to reconnect to a new one. Figure 10(a) shows that in the *Gnutella* scenario, over 95% of failures are recovered in under 20 seconds, corresponding to an average of 10 contacted candidates. The high latencies occasionally exhibited are often a result of network congestion or unusually high processing delays on PlanetLab nodes. It is also important to observe that, according to Figure 4, DUmRGM contacts the highest number of candidates among the considered protocol instances. Therefore, we expect other protocol instances to achieve better performance.

Figure 10(b) reports tests with our protocol under a catastrophic failure. Nodes are started with a cache of 30 random nodes that they use to initialize the tree. 25s after the tree has been formed, a random set of 10% to 70% of the nodes is disconnected at the same time. From this instant, we measure the time required to reconnect the remaining, live nodes into a single acyclic tree. Because very

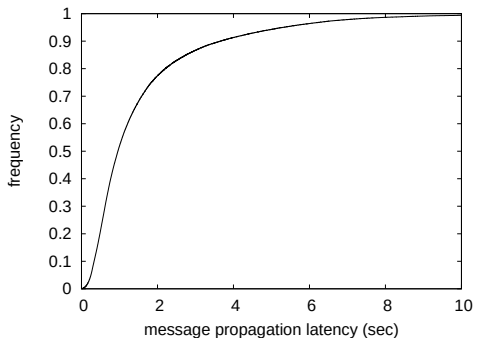


Figure 11: Message Latency

high churn may cause the tree to become temporarily split into multiple subtrees, the bar chart shows the percentage of tree repairs completed with and without splitting. Our data shows that splitting occurs only occasionally, and almost independently of the number of failed nodes. The tree was split only in 11 runs out of a total of 180 (18 per scenario). Moreover, in 9 of these runs, our protocol was able to reconnect the partitions before the end of the experiment (all experiments stopped 250s after the failure), leaving a partitioned tree in only 2 runs. Our analysis of the system state during the runs reveals that repair problems are likely caused by the occasional overload of PlanetLab nodes.

We also show the actual time required to reconnect the tree from the instant when the catastrophic failure occurs. Specifically, the line and the square points on Figure 10(b) show the time required respectively in the runs that do not temporarily split the tree, and in those that do. In the first case the average repair takes on the order of 30s, while it can be one or two minutes when tree partitions occur.

Message Latency Finally, Figure 11 shows the latency associated with message dissemination by the *p2p* application; results for *ALM* were similar. The plot shows that over 95% of the messages arrive within 6 seconds of publication and that over 99% are completed within 10 seconds. Again, these figures are affected by network and processing delays on PlanetLab nodes. Nonetheless, they confirm our protocol's ability to provide a stable overlay, allowing applications to achieve reasonable data distribution performance.

4 Related Work

Tree-based topologies have been studied in several research areas. Those closely related are outlined below.

Application-Level Multicast. One of the first application-level multicast protocols, Narada [10], exploits a two-phase mechanism to build a tree over a mesh structure. However, the protocol is mostly suited for small groups as it assumes that all nodes in the mesh know each other.

Bayeux [27], Scribe [7, 8], SplitStream [6], and I3 [18] construct data distribution trees on top of DHTs [26, 21, 24]. Bayeux requires a rendezvous node (root) to handle all join requests by new group members. Scribe uses a more scalable approach and controls node degree using a mechanism similar to our downstream cache, while SplitStream extends Scribe's behavior to manage the overall degree resulting from a node's participation in multiple trees. Finally, I3 exploits a distributed algorithm to build trees with controlled node degree and low latency. Although DHTs simplify overlay management since trees can be built over their mesh topologies, our work shows that good resilience to failures can be achieved by exploiting simple structures such as the Regional and Global caches without relying on a separate protocol to manage references to other nodes. This results in a reduction of the overall cost of the protocol as shown by our simulations.

In this respect, our approach is similar to that of Yoid [13] and Overcast [15]. However, Yoid reacts to failures by creating separate trees which are then rejoined using a complex distributed cycle-detection mechanism, while Overcast builds an overlay out of a set of dedicated servers and exploits a mechanism similar to our upstream chain. Distributed cycle detection increases the recovery cost, while Overcast causes the topology to reach very high node degrees. Moreover, Overcast is designed for fairly stable conditions, not involving the unexpected failure of the root or a cluster of nodes, whereas our system builds an overlay out of hosts exhibiting very dynamic behaviors.

Some systems, such as CoopNet [19], PeerCast [4], and NICE [3], use a rendezvous node to coordinate the construction of data distribution trees for streaming applications. Both CoopNet and PeerCast use a technique resembling our Downstream strategy, starting at a known coordination node and proceeding down the tree to find a connection point. The need for a well-known node to manage client connection requests makes these protocols unsuitable for systems with a large number of data sources and receivers.

Overlays for Content-Based Publish-Subscribe. One approach [11] maintains an overlay for publish-subscribe on top of a distributed hash table (DHT), focusing on reproducing the characteristics of a reference topology rather than reconfiguration locality. Communication cost is also dominated by underlying DHT maintenance. Similarly, [2] maintains a tree for publish-subscribe, but assumes at most one failure at a time. In contrast, our protocol supports frequent failures and is suitable for supporting publish-subscribe as well as other applications in dynamic environments.

Distributed Tree and Spanning Tree Construction. The problem of distributed tree construction has been widely studied from a theoretical perspective. However, most of this work is not designed for large-scale systems and com-

putes spanning [1, 9, 20] or shortest path [14] trees based on precise information about the underlying network topology. Further in contrast to our approach, changes are not localized nor are temporary routing loops avoided.

5 Conclusions and Future Directions

Overlay trees are among the most common topologies for data distribution in large-scale networks, however they have often been regarded as unable to deal with dynamic environments due to their inherent fragility. In this paper, we confronted this assumption with a novel protocol for maintaining a tree overlay in a highly dynamic environment while at the same time managing node degree and limiting the impact of changes. Our simulation and experimental results demonstrate the effectiveness of our protocol to achieve these goals with a quick and communication-efficient repair process.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *4^{th} Int. Wkshp. on Distributed Algorithms*, pages 15–28. Springer-Verlag New York, Inc., 1991.

[2] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. A self-organizing crash-resilient topology management system for content-based publish/subscribe. In *DEBS*, Edinburgh, Scotland, UK, May 2004.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, 2002.

[4] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media. In *HotNets-I, Princeton, NJ*, pages 107–112, Oct. 2002.

[5] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3):332–383, August 2001.

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

[7] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE J-SAC*, 20(8), October 2002.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communication, 5^{th} Int. COST264 Wkshp.*, September 2003.

[9] N.-S. Chen, H.-P. Yu, and S.-T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39(3):147–151, 1991.

[10] Y.-H. Chu, S.G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[11] P. Costa and D. Frey. Publish-Subscribe Tree Maintenance over a DHT. In *DEBS*, Columbus, Oh, USA, June 2005.

[12] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *SAC*, 2004.

[13] P. Francis. *Yoid Tree Management Protocol (YTMP) Specification*. ACIRI, April 2000. `http://www.icir.org/yoid/docs/ytmp.pdf`.

[14] J. J. Garcia-Luna-Aceves and S.Murthy. A path-finding algorithm for loop-free routing. *IEEE/ACM TON*, 5(1), 1997.

[15] J. Jannotti, D.K. Gifford, K.L. Johnson, M.F. Kaashoek, and J.W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *OSDI*, pages 197–212, Oct. 2000.

[16] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM TOCS*, Aug. 2007.

[17] A. Klingaman, M. Huang, S. Muir, and L. Peterson. PlanetLab Core Specification 4.0. Technical Report PDN–06–032, PlanetLab Consortium, June 2006.

[18] K. Lakshminarayanan, A. Rao, I. Stoica, and S. Shenker. End-host controlled multicast routing. *Elsevier Computer Networks, Special Issue on Overlay Distribution Structures and their Applications*, 2005.

[19] V.N. Padmanabhan, H.J. Wang, P.A. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Wkshp. on Network and operating systems support for digital audio and video*, pages 177–186, New York, NY, USA, 2002.

[20] R. Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *9^{th} Symp. on Data communications*, pages 44–53. ACM Press, 1985.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, November 2001.

[22] E.M. Royer and C.E. Perkins. Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol. In *MobiCom*, pages 207–218, Seattle, WA, USA, August 1999.

[23] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

[24] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2001.

[25] A. Varga. OMNeT++ Discrete Event Simulator. `www.omnetpp.org`.

[26] B.Y. Zhao, L. Huang, S.C. Rhea, J. Stribling, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1), January 2004.

[27] 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 *Wkshp. on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM Press.