Abstract

Currently, there is increasing interest for building large-scale overlays to efficiently deliver data to a large number of simultaneous receivers. Example applications include multimedia distribution, event notification, and update propagation among wide-area replicas. In this paper, we discuss techniques for building scalable degree-constrained, low-cost overlays that meet target performance characteristics. We assume that performance and cost are independent and dynamically changing metrics associated with network links. Building the lowest cost tree that satisfies end-to-end performance specifications is an NP-complete problem. Thus, our challenge is to build a distributed and scalable system that closely approximates the global optimum under a variety of conditions. We discuss our experience with achieving this goal through the implementation and evaluation of an ACDC (Adaptive low-Cost, Delay Constrained) overlay.

1 Introduction

System support for efficient point-to-multipoint communication has long been recognized as fundamental for a broad range of applications, including multimedia distribution, event notification, and update propagation among wide-area replicas. However, fundamental difficulties involving scale, congestion control, reliability, and management have limited the deployment of IP multicast. Thus, a number of industrial and academic efforts are investigating the utility of application-layer multicast [1, 3, 5, 6, 7] to efficiently support multicast-style communication. In this model, an application recruits client nodes and/or intermediaries spread across the network to forward data from a source to clients, in effect forming an overlay on top of the underlying IP network. Existing techniques for application-layer multicast seek to limit their overhead relative to native IP multicast by ensuring that the overlay topology closely conforms to the topology of the underlying network through the repeated probing of network conditions. These approaches typically require maintaining global knowledge about nodes participating in the overlay, performing global probing of all potential neighbors, or both, limiting overall system scalability to a few tens of nodes.

In this paper, we show that it is possible to build and maintain scalable overlays that dynamically adapt to the characteristics of the underlying network to deliver end-to-end performance targets. Specifically, we wish to build a delay- and degree-constrained, low-cost tree that scales with increasing numbers of participants and dynamically adapts to changing network conditions. Solving this problem using global knowledge is NP-complete [15, 17]. Our challenge is to approximate the global solution in a decentralized manner using partial information. In this context, this paper makes the following contributions:

- We present the design and evaluation of an overlay algorithm, ACDC (for Adaptive low-Cost, Delay Constrained), to build large-scale overlays. Each node tracks the characteristics of a configurable number of remote nodes \(O(\lg n)\) of total by default) to determine if better connectivity is available elsewhere in the overlay. Membership in this subset probabilistically changes periodically to ensure that, over time, each node is able to consider all remote nodes for better connectivity.

- We show how to simultaneously optimize for a number of conflicting targets for the overlay. Each participating node specifies a degree-constraint, the number of remote nodes that it can peer with. The...
application as a whole specifies an end-to-end delay target for transmission of data from the source to all receivers. ACDC then attempts to build the lowest cost overlay that meets both the degree constraint and the delay target. In our model, overlay link cost and delay are independent metrics. Thus, this work is the first to demonstrate an effective distributed approximation to the NP-complete bi-criteria network optimization problem [15, 17]. Note that while our results focus on delay versus cost tradeoffs in overlay construction, we believe our approach extends to a variety of performance metrics (bandwidth, loss rate, jitter, etc.).

- We demonstrate how to establish a total ordering among overlay nodes in a distributed and scalable manner (e.g., with no locking). This total ordering allows individual nodes to simultaneously modify overlay connectivity and topology without introducing loops into the application-layer multicast distribution tree.

- Finally, we carry out a performance evaluation of a prototype running under a variety of network conditions. Our results indicate that ACDC is able to converge quickly to performance targets with low control overhead. Further, we evaluate the system’s ability to react dynamically to changing network conditions and node failure. We also explore the tradeoff between control overhead and convergence time.

Receivers join and leave the overlay arbitrarily. A key goal is to ensure that each receiver finds a parent in the overlay such that application-specified targets for delay and per-node degree are satisfied. When a particular subtree achieves its target performance, it continues to reorganize in an effort to converge toward the lowest-cost tree able to deliver the target performance. However, the problem of building a degree- and delay-constrained minimum-cost spanning tree is NP-complete even with centralized information. Because we construct ACDC overlays in a decentralized, degree-bounded manner us-
ing partial and potentially stale information, it is clearly not possible to achieve the optimal overlay in general. However, our evaluation in Section 4 shows that we are able to come sufficiently close to the optimum (as embodied by a minimum cost spanning tree or a shortest path tree) for a variety of network topologies even under variable network conditions.

It is not sufficient for ACDC overlays to converge to a single static structure. Because we assume that both link performance and link cost change over time, ACDC must continuously probe the network to ensure that delay targets are met and that the lowest cost links are employed to achieve these targets. Of course, it may not always be possible to achieve target performance for a given set of network conditions. While not explored in this paper, we envision applications adapting their behavior to changing network conditions based on feedback from ACDC (each node in ACDC is aware of the worst-case and average delay to all nodes beneath it). Similarly, an application may be unwilling to “pay” (in terms of overlay cost) more than a given amount to achieve a target level of performance. ACDC overlays are free to relax their performance targets; in return, ACDC will take the opportunity to relax to a lower-cost overlay where possible.

Building scalable ACDC overlays presents a number of challenges. We cannot require global knowledge, global network probing, or global locking. First, each node should only maintain knowledge about a subset of global participants and this subset should grow sub-linearly with increasing numbers of participants. Similarly, the process of locating a “better” parent in the tree would naively require that each of \( n \) participants perform network probes to all \( n - 1 \) other nodes; this operation imposes \( O(n^2) \) network overhead (where \( n \) is the number of overlay participants). Worse, because the overlay must adapt to changing network conditions, these \( O(n^2) \) probes must be performed regularly to maintain target overlay characteristics. Such probing cannot scale beyond a few tens of participants. Assuming the presence of a scalable mechanism for locating a better parent, we must avoid global locking for modifying the overlay (as described in Section 3.2.2). Consider a node \( A \) that decides to move underneath a remote node \( B \). The system would introduce a loop if some ancestor of \( B \) simultaneously decides to move under some descendant of \( A \). The naive approach to avoiding loops requires locking a number of nodes across the wide area to avoid such simultaneous overlay transformations. While this may be appropriate for a small number of nodes or for LAN settings, this process will not scale to large-scale overlays.

3 Distributed ACDC Algorithm

3.1 Overview

Given the system model and goals described in the previous section, this section describes our approach to building and maintaining overlay distribution trees with the following characteristics: i) scalable, ii) degree-constrained, iii) delay-constrained, iv) low-cost, v) adaptive, and vi) self-organizing. To achieve scalability in ACDC, we enforce the following rules:

1. No node should keep track of more than \( O(\lg n) \) remote nodes.

2. No node should perform more than \( O(\lg n) \) network probes during any time period (epoch). The application can configure the number of probed nodes to effect a tradeoff between network overhead and the adaptivity (or agility [8]) of the overlay.

3. No global locking should be required to transform the overlay.

We achieve the above goals through the probabilistic distribution of probe sets to each node once per epoch that achieves a total ordering among all participating overlay nodes. An epoch consists of two phases: one distribute phase to transmit data from the root to all overlay participants (data is distributed down the tree) and a second collect phase where each participant successively updates its parent’s status information (data is aggregated up the tree). During the distribute phase, each node sends to its children a probe set of remote nodes (\( O(\lg n) \) in size by default). The contents of the probe set are constructed from candidate sets gathered during the previous collect phase. Each node \( A \) performs probes to members of its probe set to determine if a remote node \( B \) exists that would deliver better cost, delay or both to \( A \) and its descendants. When a distribute message reaches a leaf, it triggers the beginning of the next collect phase where each node informs its parent of the identity of a subset of its descendants (the candidate set) along with other
metadata. This information continues to propagate up to the root of the tree. The collect phase is complete once the root has received descendant lists from all of its children. The root signals the beginning of a new epoch by distributing a new probe set to each of its children, at which point the entire process begins over again. A total ordering among nodes distributed in the probe set ensures that two nodes cannot perform simultaneous overlay transformation that would introduce a loop in the tree (described in Section 3.2.2).

While each node only tracks and probes $O(\log n)$ remote nodes during any one epoch, the makeup of the probe set changes probabilistically such that, over time, each node probes all potential parents. The size of the probe set effects a tradeoff between scalability—measured by per-node state and network probing overhead—and convergence time—the amount of time it takes to build a low-cost tree that achieves delay targets in response to changing network conditions. We present the details of the algorithm in the following subsections. We provide full details of the correctness of our approach (e.g., with respect to subtle timing issues) in section B. For simplicity, we begin our discussion assuming a static set of nodes attempting to converge to global targets. We then describe the process for the dynamic addition and failure recovery of nodes.

3.2 ACDC Communication and State

Each node participating in an ACDC overlay maintains the following state: parent address, a list of children, the sequence number of the current epoch, a candidate set, a probe set, the root delay, the tree height, the distance to its furthest descendant, the total number of participants in its subtree, the total number of descendants, and the current direction bit. Below, we describe how ACDC uses this information and how it maintains it in a decentralized manner.

3.2.1 Collect Phase

Overall, the goal of the collect message is for each node to: i) compose the candidate sets for constructing the probe set during the subsequent distribute message processing, ii) determine the total number of participants in its local subtree, and iii) learn the delay to its furthest descendant. Individual nodes use this last value to guide future relaxations. For example, if one of a node’s descendants exceeds the overall delay target, it should look for a new parent with a shorter delay to the root.

Figure 1: Collect message propagation.

The collect phase begins at the leaves of the tree in response to the reception of a distribute message. Table 1 describes all the fields in collect messages (in the left half of the table). The collect message has the same sequence number as the triggering distribute message. At the leaves, the number of descendants is set to one and the candidate set contains only the leaf node itself. Once a parent receives all collect messages from its children, it further propagates a collect message to its own parent. The nodes in the candidate set are selected randomly from the candidate sets received from its children to form a set of size $O(\log n)$ by default. A node is selected from a child’s candidate set according to the number of nodes in the subtree rooted at the child. This ensures that the candidate set is uniformly representative of the total population below the node. Each node stores this candidate set to aid in the construction of probe sets distributed to its children during the subsequent distribute phase.

3.2.2 Distribute Phase

An epoch begins when the root receives a collect message from all of its children. To allow time for potential transformations to complete, it then waits an amount of time equal to twice the delay target of the tree before transmitting the first distribute message of the new epoch to all of its children. Thus, an epoch lasts approximately four times the delay target (though this value is config-
<table>
<thead>
<tr>
<th>Collect</th>
<th>Distribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence #</strong></td>
<td>Sequence number of current epoch</td>
</tr>
<tr>
<td><strong>Candidate Set</strong></td>
<td>Random subset of nodes in sender's subtree</td>
</tr>
<tr>
<td><strong>Descendants</strong></td>
<td>Estimate of number of nodes in sender’s subtree</td>
</tr>
<tr>
<td><strong>Delay</strong></td>
<td>Delay estimate for furthest descendant</td>
</tr>
<tr>
<td><strong>Cost gain</strong></td>
<td>Estimate of cost gain from moving to a best alternative parent</td>
</tr>
<tr>
<td><strong>Delay gain</strong></td>
<td>Estimate of delay gain from moving to a best alternative parent</td>
</tr>
</tbody>
</table>

Table 1: Contents of collect and distribute messages.

A parent constructs probe sets for each child in the following manner. Recall that each node stores the candidate set received from each child during the previous collect phase. Thus, for each of $k$ children, a particular node maintains $CS_1, CS_2, \ldots, CS_k$. Also recall that each candidate set, $CS_i$, consists of nodes selected randomly from the subtree rooted at node $i$. A parent node $A$ constructs a probe set for each child from this information saved during the preceding collect phase. This information includes the candidate set for each child, the node $A$ itself, as well as the probe set received from $A$’s parent. For each child $x$ (numbered $1, \ldots, k$), the parent node $A$ constructs $PS_x$ in the following manner:

$$PS_x = \text{Compact}(CS_{x+1}, \ldots, CS_k, PS_A, \{A\})$$ (1)

The Compact operation ensures that members of the probe set are selected randomly, with each set weighted according to the population it represents. Thus, each child will receive a probe set chosen with uniform probability from all potential parents. Figure 2 depicts a potential scenario.

One immediate question with this approach is: Why does the input to the Compact operation need to be different for each of the children? The rationale behind this operation is to establish a total ordering among all probe sets. This total ordering makes it impossible for two nodes to simultaneously pick new parents in an epoch that would introduce a loop into the overlay (ACDC allows each node to make only a single transformation per epoch). For example, consider two nodes $A$ and $B$ both of whom are children of $C$. If $PS_A$ contains $B$ and all $B$’s descendants and $PS_B$ contains $A$ and all its descendants, then it becomes possible for $A$ to determine that it should relocate underneath $B$ in the same epoch that $B$ determines that it should become a descendant of $A$, introducing a loop. Recall that the use of locking to prevent loops does not scale to large-scale overlays. Thus, our approach to constructing probe sets ensures that in any given epoch either $A$ learns of $B$ and its descendants or $B$ learns of $A$ and its descendants, but never both simultaneously.

A limitation of the above approach is that the $k$th child...
of a particular node will always have a smaller set of potential parents to choose from than the first. In fact, the $k$th child’s probe set would always be restricted to a relatively small subset of global nodes. For this reason, the root of the overlay periodically (every configurable $r$ epochs) flips a direction bit. This bit simply determines the order in which probe sets are constructed. At the same time the ordering of the children is also randomly reshuffled to ensure that children in the middle of the list are not unnecessarily restricted. In the discussion above, the direction bit was set such that probe sets were constructed for children from $1,\ldots, k$. If the direction bit is flipped, probe sets are constructed starting with the $k$th child.

### 3.3 Overlay Transformations

A key to ACDC’s ability to converge to target low-cost and delay-constrained structures lies in localized tree transformations. During each epoch, nodes measure the delay and cost between itself and all members of its probe set. The goal is to locate a new parent that will deliver better delay, better cost, or both to itself and all of its descendants. If a better parent is located, the child moves under it. The migrating node issues the add request to the potential parent and waits for the response. In the case the request for the move is accepted, it notifies its old parent, communicates its new delay from root to all its children, and notifies the new parent of its furthest descendant (updating the parent’s state for the next epoch).

We use Figure 3 to help visualize the four ACDC overlay transformations, DELAY\_ONLY, COST\_ONLY, SAFE\_COST, and DELAY\_AND\_COST. Nodes $B$ and $F$ are root’s children. Node $C$ has node $B$ as its parent, and is probing four other nodes, $A$, $E$, $D$, and $F$. We use $x$ to denote Node $C$’s delay to $B$. Ordered pairs show result of the probe in the form $(\text{cost}, \text{delay})$. The number beside the node name denotes its distance from the root. The height of the subtree rooted at each node represents the furthest descendant of that particular node. Root’s furthest descendant (equal to the tree height) is $y$ seconds away. In our example, the delay bound is $1.0$ seconds.

The DELAY\_ONLY transformation is most straightforward: from the probe set, each node picks a potential parent that most reduces its delay from root. The goal of this transformation is to bring the tree within the delay bound. Consider $x > 0.2$, $y > 1.0$: node $C$ has no choice but to perform a DELAY\_ONLY transformation. It will choose node $A$, because doing so offers the largest delay reduction. However, the overall cost of the overlay will increase by $10$.

The COST\_ONLY transformation is designed to reduce overall tree cost. The algorithm considers this transformation only when the overlay satisfies the delay bound. This transformation searches for nodes that have a lower cost than the link to its current parent. In addition, the root delay to the probing node’s furthest descendant must not exceed the delay target when it attaches to this new parent. Although a relocating child sends delay updates to its parent (and, therefore, to all of its ancestors) and its descendants, there is the possibility for this transformation to temporarily violate the overlay’s delay target. Consider the following example. A node $A$ performs a transformation to a new parent that improves cost but somewhat increases delay. This increased delay does not violate the constraints of any of $A$’s current descendants. However, a node $B$ may simultaneously (in the same epoch) perform a COST\_ONLY relaxation to node $C$, node $A$’s descendant, increasing the delay to $B$ and $B$’s descendants, but still within bounds based on $C$’s previous position in the overlay. The cumulative effect of simultaneous delay increases to $A$ and to $B$ may result in delay target violations for $B$ or some subset of its descendants. While such temporary delay violations are acceptable given that we do not pro-

---

**Figure 3:** Sample network illustrating possible transformation types. Nodes $B$ and $F$ are some of root’s children.
vide hard performance guarantees, we mitigate this effect by probabilistically limiting the number of simultaneous COST_ONLY relaxations in any one epoch. To achieve this effect, each node performs COST_ONLY transformations probability \( \frac{1}{\log n} \) during each epoch. The rationale is that \( \log n \) is roughly the height of the tree and we wish to avoid multiple transformations within the same path. To illustrate, consider the case in Figure 3 where \( x < 0.2, y < 1.0 \). Node \( C \) will choose node \( D \) as its potential parent because doing so provides it with the biggest cost gain.

A SAFE_COST transformation similarly improves cost, but has the additional constraint that it cannot increase its new parent’s furthest delay. That is, the worst-case delay of the sub-tree rooted at the new parent will not increase. Thus, any number of nodes may simultaneously attempt SAFE_COST relaxations, subject to the total ordering. Going back to our example, if \( x < 0.2 \) and \( y < 1.0 \), node \( C \) cannot choose node \( D \) for a SAFE_COST transformation because its furthest descendant is 0.4 seconds away, which is less than node \( C \)’s 0.5 and the delay between \( C \) and \( D \). Of the three remaining nodes, node \( E \) results in the highest cost gain (and still maintains the appropriate delay). We note that the overall reduction cost for SAFE_COST is not as large as that achieved by the COST_ONLY transformation.

In order to reduce cost in the case when the delay bound is violated somewhere in the tree, nodes use the DELAY_AND_COST transformation. It is very similar to SAFE_COST, except that it requires the node performing this transformation to also reduce the delay (and thereby help the overall effort of bringing the tree height below the delay target). In our example, if \( x < 0.2 \), but \( y > 1.0 \) (a node other than \( C \) is violating the delay bound somewhere in the tree), node \( C \) cannot choose node \( D \) as its parent. It cannot pick \( E \) because it would actually increase it’s own delay to root. Of the two remaining nodes, node \( F \) results in the highest cost gain. In this case, the overall cost reduction is the smallest among all cost-aware transformations.

### 3.4 Dynamic Node Addition and Failure Recovery

To join an ACDC overlay, a node simply needs to contact any existing member of the overlay. Any number of techniques are appropriate for this operation, for example, i) using locally available IP anycast or multicast with scoped TTL to find a nearby member, ii) consulting a DNS directory dynamically updated to contain a subset of participants, participating in any resource discovery, but in the self-organization of the system once at least one existing member has been found. For our experiments, we pessimistically assume that each new node attempts to join at a random point in the overlay.

Handling node failure is simplified by our periodic distribution of collect and distribute messages, which implicitly act as heartbeat messages. Each parent waits for collect messages from all of its children. If a message is not received within some multiple of the delay target (note that the current root-to-leaf delay of the overlay serves as a convenient baseline for the complex process of determining appropriate timeouts), the parent assumes that one of its children has failed and excludes it from participating in the next epoch by not sending a distribute message to that child. A node \( A \) similarly detects the potential failure of its parent when it does not receive a distribute message within a multiple of the delay target. In this case, \( A \) will send a dummy distribute message (with an empty probe set) to all of its children. However, rejoining the overlay is simplified because \( A \) already has information about a number of group members in its previous probe set. In addition, the random composition of the probe set increases the ability of the node to rejoin the overlay when compared to contacting only its grandparent or ancestors. The empty distribute message enables the entire subtree rooted at \( A \) to rejoin with a single operation, rather than forcing all nodes below \( A \) to rejoin separately.
3.5 Weans

Under certain circumstances, the greedy nature of the ACDC algorithm can lead to sub-optimal overlays. Consider the following situation. A node $A$ with a particular degree bound has a full complement of children. However, a node $B$ somewhere else in the overlay can only achieve its delay bound by becoming a child of $A$. Finally, one of $A$’s children, $C$, is best served by $A$ but it would still be able to achieve its delay bound as a child of some fourth node $D$. As described thus far, ACDC will become stuck in a “local minimum” in this situation. An analogous situation can arise with cost rather than delay.

We address this situation by adding wean operations. Even with this technique, we are not able to achieve the optimum in the general case. Weans allow for increased ability in approximating the SPT and the MST. A parent determines the wean target based on the child estimated to be able to lose the least in delay while still achieving its delay target (similar reasoning applies for cost weans). During each epoch, all nodes maintain information on the best alternate parent with respect to both delay and cost. This information is propagated to parents in the collect phase (see Table 1) and it is used by parents to determine which child to wean. A child that is being weaned operates normally in subsequent epochs with the exception that it will attempt a transformation even if the target is not strictly better than its current parent. Note that a wean may or may not succeed (an appropriate alternate parent may not exist). The wean operation expires after a configurable number of epochs.

3.6 Discussion

Having described our baseline algorithm, we now discuss a number of issues associated with our approach. Our greedy approach to finding superior parents means that ACDC is unable to achieve either the smallest delay delivered by a shortest path tree or the lowest cost achieved by a minimum cost spanning tree. However, our performance evaluation in Section 4 indicates that our approach is able to come within 10-20% of the optimum cost and within 5% of delay under a variety of conditions.

Further, we do not provide hard guarantees on the application-specified delay target. Because we must allow overlay participants to simultaneously execute transformations, we may temporarily exceed the delay target. In addition, because we rely on approximate information (periodic measurements of network conditions that are constantly changing), the delay target may be exceeded before ACDC can react. However, we believe that the Internet has demonstrated that “best effort” approaches are sufficient for a broad range of applications. For our target workload, temporarily exceeding delay is acceptable.

Finally, one interesting question is how convergence time interacts with the number of participants given ACDC’s probabilistic probing techniques. On the one hand, if each node randomly probes $O(\log n)$ remote nodes every epoch and if the “proper” parent is uniformly distributed among the set of remote nodes, then the expected number of epochs to locate one’s best parent is $\frac{n}{\log n}$. While reasonable for moderate-sized overlays with, for example, 1000 participants, the number of such rounds quickly grows to unacceptably large numbers for larger-scale systems with, e.g., a million or more participants. Fortunately, we are not interested in locating the optimal parent in each case. Thus, we make the following argument regarding the scalability of our approach. If we assume that the relative “goodness” of all remote parents can be ranked from $1, \ldots, n - 1$ and that all remote nodes are uniformly distributed across this space, then we can discuss how many expected probes are required to find the “90th-percentile” parent, that is, the one that is better than 90% of all other parents. With $k$ probes, the probability that we do not find such a parent is $0.9^k$. Finding the 90th-percentile parent 50% of the time then requires an expected 7 probes. In general, the expected number of probes, $k$, to find the $P_{\text{target}}$-percentile parent $P_{\text{hit}}$ percent of the time reduces to:

$$k = \frac{\ln(1 - P_{\text{hit}})}{\ln P_{\text{target}}} \quad (2)$$

Hence, the median for finding the 99th-percentile parent 50% of the time is 69 probes while the median for finding 90th-percentile parent 90% of the time is 22 probes. A million node ACDC overlay probing 20 (i.e., $\log n$) nodes every epoch would achieve these probabilistic guarantees in 2-4 epochs under idealized conditions. We offer a more detailed analysis in section C.
4 Evaluating ACDC

This section presents the performance evaluation of our ACDC prototype running under ns network simulator [9]. We design our experiments to answer the following questions: Can ACDC achieve performance (delay) targets while building a low-cost overlay? How do achieved delay and cost of the overlay compare to the shortest path tree and minimum spanning tree respectively? Can applications flexibly trade delay for cost in their overlay? What effect does the probe set size have on the overlay convergence time and on network overhead? How adaptive is ACDC to changing network conditions? Finally, how does ACDC tolerate node failures?

For most of our experiments, we use 600-node transit-stub topologies with the GT-ITM [18] topology generator. We set link bandwidth to 155 Mb/s for transit-transit links, 45 Mb/s for transit-stub links and 10 Mb/s for stub-stub links. GT-ITM determines delay values by calculating propagation delay given the relative positioning of nodes in a plane. Note that ns accurately simulates the additional effects of queueing delays. Finally, we set the cost of physical links to be uniformly distributed between 20-40 for transit-transit links, 10-20 for transit-stub links, and 1-5 for stub-stub links. We also ran our experiments (omitted for brevity) with random cost assignments with results similar to those reported below. Note that a contribution of our work is not an accurate model for assigning link cost. Rather, we wish to demonstrate that ACDC can optimize for overlay cost independent of the technique used to assign costs. By default, we randomly choose 120 nodes out of a 600-node topology to participate in the ACDC overlay. Specific experiments explore the behavior of the system with larger networks and more participants.

4.1 Overlay Convergence

Figure 4 shows the convergence time and cost of two ACDC overlays for a representative 600-node topology and 120 randomly selected participants. The 120 nodes join the overlay sequentially, each picking a random initial parent. In effect, at the beginning of the experiment we pessimistically create a random overlay. Figure 4 shows the cost of the overlay relative to the minimum cost spanning tree (calculated offline) on the left-hand y-axis as a function of time progressing on the x-axis.

A “Cost vs. MST” of 1.0 corresponds to the degree-unbounded optimal. We run these experiments with a probe set of 10 and a maximum degree of 10. For this topology, chosen root, and set of participants, a shortest path tree (constructed from the overlay participants) achieves a 443 ms worst-case root-to-child delay with a maximum degree of 49. We observe the behavior of the system for two different delay targets, 531 ms (corresponding to 1.2 * SPT) and 620 ms (corresponding to 1.4 * SPT). The right-hand y-axis shows the achieved delay relative to the delay target as a function of time progressing on the x axis. ACDC quickly converges to the specified delay target in both cases, despite starting with random overlay connectivity. Once the delay bound is achieved, the cost continues to decrease to within 20% of the MST optimum (whose degree is 7 and delay is 1318 ms).

While omitted for brevity, we also measure network stress [5] for our overlays. Stress measures the average number of times a packet crosses each IP link that makes up the overlay. Idealized IP multicast delivers a stress of 1.0. We find that our overlays typically converge to within 30% of optimal. Overall, this overhead compares favorably with other application-layer multicast schemes and should be acceptable given the difficulties associated with deploying native IP multicast.

4.2 Trading Delay for Cost

ACDC applications can trade delay for cost by allowing senders to specify and vary a target delay. A particular
delay target will correspond to a particular level of cost for an overlay. By relaxing or tightening the delay target, applications running on ACDC can control the cost incurred from the overlay. Figure 5 quantifies such cost versus delay tradeoffs. We average our results over six different randomly generated 600 node topologies and use three different random seeds to select the 120 participating nodes, for a total of 18 different runs for each data point. The error bars in the figure represent the standard error across these 18 runs. The x-axis of this figure varies the application-specified delay bound as a multiple of the idealized delay that could be delivered by a shortest path tree in that topology. The left-hand y-axis is the cost of the overlay relative to a minimum spanning tree while the right hand y-axis depicts the achieved delay as a multiple of the SPT delay. Ideally, the achieved delay on the y-axis would always match the delay target on the x-axis.

Overall, Figure 5 shows that ACDC achieves the target delay as long as it is at least 5% larger than the delay of a shortest path tree. Because ACDC trees are degree-bounded and make local greedy decisions, the overlay typically gets stuck in a local minimum at this point. We believe that coming within 5% of the delay of an SPT (that has no degree constraint) in our target topologies is sufficient for a broad range of applications. Similarly, the cost of the overlay comes within 13% of the MST cost when the delay bound is relaxed sufficiently. More interestingly, the knee of the curve for cost comes at approximately 20% of the optimal SPT. Relaxing delay beyond this point does not result in significantly lower steady-state cost (though it could take longer to achieve such cost).

### 4.3 Effects of Probe Set Size

One important tunable parameter of ACDC is the probe set size. This parameter indicates how many nodes are included in collect and distribute messages, the bandwidth committed to probing overhead, and the overall adaptivity of the overlay. Figure 6 plots the cost convergence time of the overlay, where convergence time is defined as the time it takes to come within 5% of the best achieved cost (determined by running with probe set size of 50 for 8,000 seconds), as a function of the probe set size. The results are averaged over 4 different combinations of topologies and participants with a delay target of 1.8 times the SPT delay (which varied from 396-796 ms for these topologies). In addition to 600-node topologies with 120 participants, we used 1640-node and 2440-node transit stub topologies for these experiments, with 320 and 488 randomly-selected participants respectively to further demonstrate the scalability of our approach. The cost convergence plotted in this figure is necessarily much slower than delay convergence (recall that the rate of \text{COST}\_\text{ONLY} transformations are limited to prevent temporary violations of the delay target). The graph shows that the convergence time drops rapidly with increasing probe set size, leveling once probe set size reaches 20. In this case, even the largest overlay achieves the desired cost target within 600 seconds, reasonable for a system of this size especially when considering that the overlay starts with random interconnectivity. Figure 6 also shows that per-node probing overhead is small, rising to less than 900 bytes/sec/node for a probe set size of 20. Perhaps more importantly, this overhead is largely independent of the number of overlay participants.
To evaluate this ability, we subject a steady-state ACDC overlay (120 participants in a 600-node topology) to wide-spread and sustained change in network characteristics, where 500 randomly selected (out of 1,200 total) links increase their delays by between 0 and 25% of the original link delay every 25 seconds for a total of 1000 seconds. The idea behind this experiment is to determine what happens to the overlay as the network continuously degrades under conditions much worse than those typically found on the Internet. Figure 8 shows the results of this experiment, with the two y-axes plotting cost (versus MST) and the achieved delay as a function of time progressing on the x-axis. Instead of plotting only the maximum ACDC delay, we also show 90th and 95th percentile ACDC delay (x-th percentile delay requires x percent of nodes to have delay less than the value shown). Once again, we run with 120 random participants out of a 600 node graph with a degree bound of 10. For simplicity, we present the results for a single representative topology. Runs for other topologies are similar.

Initially, the overlay quickly converges to the delay target (1550 ms for this experiment). The graph also shows the value for the SPT (whose degree is 49), initially at 443 ms. The overlay also quickly converges to a cost within 15% of the MST (whose degree is 7) 500 seconds into the experiment. At time $t = 1500$ and continuing through time $t = 2500$, we change network characteristics in the manner described above. At this point, the target delay is violated so ACDC performs a number of DELAY_ONLY and DELAY_AND_COST transformations to bring the delay back within bounds. As network conditions continue to change, the delay for the SPT connecting the participating nodes grows from its baseline value of 443 ms to 1285 ms by the time the change in network conditions subsides at $t = 2500$. However, ACDC is able to maintain the delay target, sometimes sacrificing cost to do so. Perhaps more importantly, 95 percent of the nodes stay within the delay bound most of the time. Once network changes subside, ACDC first brings the maximum delay within the delay bound. It then focuses on cost bringing it to within 20 percent of the optimal MST. Overall, we believe that these results show the resilience of ACDC to highly dynamic network conditions.
4.5 Node Failures

This section explores the ability of ACDC to recover from node failures. We run this experiment on the same representative 600-node topology (with 120 overlay participants), as in section 4.1, with the delay target of 1.8 SPT. To capture the notion that nodes that have been up for a given period of time are likely to be more stable (some evidence of this is available in a recent study [16]), we model node uptimes using a Weibull distribution with $\alpha = 0.05, \lambda = 0.5$. These parameter settings result in an average uptime of 800 seconds. We allow the overlay to converge and after 600 seconds we begin to fail nodes. Upon failure, a node is eligible to rejoin the overlay after 25 seconds. Once every second, we randomly choose one node among the set of down nodes to bring back up. We fail nodes for a total of 2400 seconds, and then allow the overlay to converge again.

![Figure 9: Cost and percentage of nodes with delay bound met as a function of time.](image)

Figure 9 shows the cost of the overlay relative to the minimum cost spanning tree (calculated offline among active nodes) on the left-hand y-axis and the percentage of active nodes whose delay bound is satisfied on the right-hand y-axis over time. After we start failing nodes at $t = 600$, there is a 40% decrease in the percentage of connected nodes. Although the AC/DC overlay quickly recovers, after more nodes fail (between 600 and 1,000 seconds), the tree experiences consistently low connected ratio. This is because nodes failures occur independent of their position in the tree, and failures of nodes that are close to the root disconnects significant portions of the tree.

We note that as time progresses (after 1500 seconds until the end of the experiments), the overlay experiences spikes in the ratio of connected nodes that are significantly shorter (never more than 20 percent). This is because longer-lived nodes have percolated up toward the root, and repeated failures of short-lived nodes have a relatively smaller impact on the tree, as they have joined as leaves. The impact of node failures on the cost is similar to the trend observed with the percentage of nodes who meet their delay targets. Initially there are large spikes in cost as nodes high up in the tree may fail. As more stable nodes percolate up the tree, variation in cost is reduced.

While we do not focus on the difficult problem of determining the proper timeout to detect parent failure, we note that the very nature of the ACDC overlay provides a convenient baseline for timeouts. If a node were to rely solely on ACDC distribute messages to timeout a parent, detection time of a few epochs (effectively several delay target multiples) is sufficient. Tighter integration between the application and ACDC could shorten this time. For example, an application that periodically receives data at a higher rate could inform ACDC of a gap in the application stream. Further, individual nodes within the overlay may buffer a certain amount of traffic to aid disconnected nodes in application-specific recovery mechanisms once they rejoin the overlay.

5 Related Work

Our work builds upon a number of recent efforts into “application-layer” multicast, where nodes spread across the Internet cooperate to deliver content to end hosts. Edges in this overlay are TCP connections, ensuring congestion control and reliability in a hop-by-hop manner. Perhaps most closely related to our effort in this space is Narada [5, 6], which builds a mesh interconnecting all participating nodes and then runs a standard routing protocol on top of the overlay mesh. While Narada also optimizes for two metrics, delay and bandwidth, the two-metric network optimization problem is not NP-complete [17] in the case where bandwidth is one of the metrics. This is true because end-to-end bandwidth is determined by the minimum of all link bandwidths between two hosts rather than as a function of all of the links. Further, Narada nodes maintain global knowledge about all group participants. While feasible for their target of video conferencing among a few dozen nodes, such a requirement will not scale to the
large-scale systems we consider. Finally, Narada does not attempt to constrain either bandwidth or delay in their overlays. Rather, the work shows that Narada performs well relative to the best performance that could be delivered by the underlying network. ACDC on the other hand exports a knob to applications enabling them to flexibly trade performance for cost. However, when cost is not a constraint, ACDC similarly delivers delay closely matching the best available from the underlying network.

ACDC bears some similarity to the Banana Tree Protocol (BTP) [4], and Host Multicast Tree Protocol (HMTP) [19]. However, neither of these approaches attempt to provide delay guarantees and neither considers a two-metric network design. All three protocols use the idea of tree transformations based on local knowledge (obtained through limited network probing) to improve overall tree quality. However, BTP implements a more restrictive policy for choosing a potential parent, called “switch-one-hop” [4], which considers only grandparents and immediate siblings. HMTP can introduce loops and thus requires loop detection that requires knowledge of all of one’s ancestors. Since HMTP offers no bounds on tree height, message size and required state are also unbounded ($O(n^2)$), rendering this approach potentially unscalable. Finally, HMTP is not evaluated under changing network conditions.

Yoid [3] shares with ACDC the design philosophy that a tree can be built directly among participating nodes without the need to first build an underlying mesh. Yoid does not describe any scalable mechanism for conforming to the topology of the underlying network, has not been subjected to a detailed performance evaluation, and contains loop detection code as opposed to our approach of avoiding loops in the first place.

ALMI [10] uses all-pairs probing at a cost of $O(n^2)$ and transmits this changing connectivity information to a centralized node that calculates an appropriate topology for the overlay. RMX [1] faces similar scalability limitations. In Overcast [7], all nodes join at the root and migrate down to the point in the tree where they are still able to maintain some minimum level of bandwidth. Relative to our effort, Overcast does not focus on scalability or two-metric network design.

Finally, a number of recent efforts [11, 13, 21] propose building application-layer multicast on top of scalable peer-to-peer lookup infrastructures [2, 12, 14, 20]. While these projects demonstrate that it is possible to probabilistically achieve good delay relative to native IP multicast, they are unable to provide any performance bounds because of the probabilistic nature of the underlying peer-to-peer system. Additionally, they do not consider two-metric network optimizations. In general, we achieve similar scalability ($O(\log n)$ state and $O(\log n)$ network probing) but provide additional control over the performance versus cost tradeoff inherent to application-layer multicast. ACDC does introduce additional complexity to enable such control whereas these approaches push the complexity associated with topology matching and robustness to the underlying peer-to-peer system.

6 Conclusions

This paper presents the design, implementation, and evaluation of a distributed, adaptive, and scalable algorithm to build low-cost, degree- and delay-constrained overlay networks. This problem is NP-complete given centralized information and static network conditions. Thus, our challenge is to develop a distributed algorithm that approximates the global optimum under a variety of dynamic network conditions. In this context, we make a number of contributions. We are able to constrain one metric, delay, while optimizing for another, cost. We develop techniques to significantly improve both scalability and overlay performance by enabling each participant to maintain knowledge about a probabilistically changing set of $O(\log n)$ remote nodes, where $n$ is the total number of participants. Further, we establish a total ordering among all participating nodes to enable simultaneous modifications of the overlay topology without the possibility of introducing loops. Finally, a detailed system evaluation shows that ACDC can flexibly trade overlay delay (within 5% of optimum at one extreme) for cost (within 10-20% of optimum at the other extreme) under a variety of dynamically changing network conditions.

References


This appendix provides insight into some of the details of the ACDC algorithm. We include an overview of the potential conflicts that may arise during distribute and collect processing. We include a proof of the correctness of the ACDC algorithm by showing that no transformation can make the tree ill-formed.

### A Potential Conflicts

There are a number of potential interactions between the timing of epochs (distribute and collect processing) and overlay transformations. Below we discuss how ACDC correctly addresses each such interaction to maintain the well-formed property of the tree.

1. A node $C$ moves from $X$ to $Y$ BEFORE receiving distribute sequence $i$ from $X$ and BEFORE $Y$ has received distribute sequence $i$ from its parent. In this case, $C$ will receive the distribute sequence $i$ from $Y$ and normal processing will occur.

2. A node $C$ moves from $X$ to $Y$ BEFORE receiving distribute sequence $i$ from $X$ and AFTER $Y$ has received distribute sequence $i$ from its parent. In this case, node $C$ will never receive distribute sequence $i$ and will not report its subtree to any node for epoch $i$. Its subtree will be excluded from the global ordering.

3. A node $C$ moves from $X$ to $Y$ AFTER receiving distribute sequence $i$ from $X$, AFTER replying to $X$ with collect sequence $i$, and AFTER $Y$ has received distribute sequence $i$ from its parent. In this case, $C$’s subtree was reported to $X$, and no conflict exists.

4. A node $C$ moves from $X$ to $Y$ AFTER receiving distribute sequence $i$ from $X$, AFTER replying to $X$ with collect sequence $i$, and BEFORE $Y$ has received distribute sequence $i$ from its parent. Because $C$ has already responded to sequence $i$ (to $X$), it will ignore the contents of the sequence $i$ distribute from $Y$ and respond with an empty collect $i$ (so that $Y$ does not wait for $C$).

5. A node $C$ moves from $X$ to $Y$ AFTER receiving distribute sequence $i$ from $X$, BEFORE replying to $X$ with collect sequence $i$, and AFTER $Y$ has received distribute sequence $i$ from its parent. In this case, node $C$ will have an outstanding distribute sequence in its subtree. Upon accepting $C$, $Y$ will not expect a collect sequence $i$ from $C$; yet it will receive it anyway because the distribute is outstanding in $C$’s subtree. $Y$ will ignore this collect. $X$ was expecting to receive a collect $i$ from $C$ but will resolve this expectation at the time $C$ is removed from $X$.

6. A node $C$ moves from $X$ to $Y$ AFTER receiving distribute sequence $i$ from $X$, BEFORE replying to $X$ with collect sequence $i$, and BEFORE $Y$ has received distribute sequence $i$ from its parent. This is is a combination of types 4 and 5. Essentially, both the distribute and the collect are ignored.

To avoid moving into a tree with a newer or older epoch sequence, potential parents check that their distribute sequence is the same as the node joining them.

### B Algorithm Correctness

ACDC’s ability to handle potential conflicts during tree transformation allow it to maintain the invariant of a well-formed ACDC tree.

An ACDC overlay is well-formed if it maintains the following characteristics:

1. It is a connected tree.
2. No node has a degree higher than the specified degree constraint, $D$.

The second characteristic is easily maintained by a node’s refusing to accept more than $D$ children; as a result, a wean/adoption sequence does not occur if it means violating $D$.

The remainder of this proof illustrates how ACDC maintains a connected tree without loops in light of potential conflicts. Degree, cost, and probe set sizes are orthogonal problems to tree formation. Hence, we assume unbounded probe set sizes and simply consider tree moves independent of whether they are initiated because of a delay or cost relaxation. Additionally, because no node is allowed to make a move in the tree after a direction
change, a direction change simply starts the algorithm anew with a new initial ordering. As a result, for the sake of this discussion, we assume no direction changes.

**Lemma 1** At the initiation of a epoch \( i + 1 \) (root transmitting a distribute message with sequence \( i + 1 \)), the root has a global ordering collected in the previous epoch of nodes in the overlay containing each node at most once (perhaps not at all). We call this ordering \( i \).

The root will not transmit a distribute message with ordering \( i \) until it has received collect messages from all of its children from which it expected to. These children recursively received collect messages from their children and created the order for their subtree. Because a node may respond with at most one collect message per ordering, it can appear at most once in the ordering. Note that under interaction types 2 and 5, a node may move and its subtree may be excluded from the ordering \( i \).

**Definition 1** If \( B \) comes before \( A \) in the global ordering, then we write \( B \prec A \). Likewise, if \( B \) does not come before \( A \), we write \( B \not\prec A \).

**Lemma 2** Upon an epoch change to sequence \( i + 1 \), a node contains sufficient information to infer the global ordering \( i \). Thus, it makes sense to speak of a node operating under some ordering sequence.

Proof by Induction. Base case: root. A parent can infer based on its local information which portions of the global ordering contain its children subtrees. Iterative step: A node receives a distribute with the global ordering \( i \) and can infer based on locally cached candidate sets the relative position of each of its children in the ordering. Furthermore, these relatives positions allow the node to infer which portions of the global ordering contain its children’s subtrees.

**Lemma 3** If \( A \) is in a subtree that moves under \( B \) using ordering \( i \), then \( A \not\prec B \) in ordering \( i + 1 \).

Call the ancestor of \( A \) that moves directly under \( B, G \). Let \( H \) be the old parent for \( G \). Then for this move, \( B \) and \( G \) are operating under \( i \) which means that \( B \) has not received ordering \( i + 1 \) yet. Three cases exist:

1. \( H \) received collect \( i + 1 \) from \( G \) before the move and \( G \) is reported as a descendant of \( H \). Now since \( B \prec G \) in \( i \), \( B \prec H \) in \( i \). Any move that could push \( H \) in front of \( B \) will result in the squashing of collect \( i + 1 \) with \( G \) in it. Hence \( G \not\prec B \) in ordering \( i + 1 \).

2. \( B \) received collect \( i + 1 \) from \( G \). Clearly, \( G \not\prec B \) in \( i + 1 \).

3. No node processed a valid collect \( i + 1 \) from \( G \). Clearly \( G \not\prec B \) in \( i + 1 \).

**Lemma 4** If node \( A \) is operating with ordering \( i \), its descendant \( B \) is operating with ordering \( j = i - 1 \) or \( j = i \). Likewise, if \( B \) is operating under ordering \( i \), its ancestor \( A \) is operating at either \( j = i \) or \( j = i + 1 \).

All nodes in the tree operate under at most two potential orderings at any given time. Two orderings occur only when an ordering is in flight.

**Lemma 5** If \( A \) is a descendant of \( B \), then \( A \not\prec B \) in \( A \)'s current operating ordering \( i \).

There are two possible cases:

1. \( A \) sent collect with ordering \( i \) after its was a descendant of \( B \), in which case \( A \not\prec B \) is clear.

2. \( A \) sent collect \( i \) via some other node. At that time it was operating under ordering \( i - 1 \). If it moved while operating under \( i - 1 \), then \( A \not\prec B \) in order \( i \). If it moved while operating under \( i \), then \( B \prec A \) in \( i \) and clearly \( A \not\prec B \) in \( i \).

**Lemma 6** If a distribute carrying ordering \( i \) is in flight from ancestor \( A \) to descendant \( B \), either \( B \) receives the ordering \( i \) or moves away from under \( A \).

**Theorem 1** No node can become a child of its own descendant.

Proof by contradiction. Assume that a node \( R \) has made a move using ordering \( i \) and has become a child of one
Figure 10: example of potential loop created by siblings of its descendants, Z. This means that Z ∼ R in ordering i. Therefore, some move occurred that made Z a descendant of R. Namely, some ancestor Y of Z (perhaps Z itself) became the child of some descendant S of R (perhaps R itself). If Z was in a subtree that moved under R using ordering k, then Z ∼ R in k + 1. Hence, k = i − 1, because Z ∼ R in ordering i. (Lemma 3)

Additionally, because at the time of the move R was operating under ordering i − 1 or i, S (and Y) were operating under i − 2 or i (not i − 1, as above). (Lemma 4)

If R was operating under i − 1, then S was operating under i − 2. The only way that could happen is if a distribute delivering ordering i − 1 were in flight from R to S. Because S must either receive this distribute or move away from being a descendant of R, it must have received it. Hence, S received at least ordering i − 1 (k ≠ i − 2). (Lemma 6)

Since k ≠ i − 1 and k ≠ i − 2, then k = i. Nodes S and Y operated under ordering i. Hence S ∼ Y in ordering i. Now Z is a descendant of Y and is operating under ordering i, ordering i must show Y ∼ Z. (Lemma 5)

Likewise, S is a descendant of R and is operating under ordering i, ordering i must show R ∼ S.

Thus, R ∼ S, S ∼ Y, Y ∼ Z in ordering i, implying that R ∼ Z in ordering i, which contradicts our initial assumption that Z ∼ R in ordering i.

C Convergence Time

In this section, we show provide closed-form expression for the expected time of absolute and partial (to within some factor of the optimum) ADC cost convergence. We first find the convergence time of the ADC under no degree or delay limitations and then provide intuition in regards to the convergence time under a constrained degree and delay.

C.1 Absolute Convergence Time

In ACDC, every node probe other nodes in the overlay in the hopes that it may find a parent that is better suited than its current parent. A node, A is a better parent than P for node X if the cost between A and X is smaller than the cost between P and X as given by some measure of cost (such as available bandwidth or real ISP cost).

In the case of unconstrained degree and delay, if a node is only allowed to probe one randomly selected in each epoch, the probability that the node picks its best parent in the xth epoch is given by the pdf:

$$f_X(x) = \frac{1}{n} \left(\frac{n-1}{n}\right)^{x-1}$$

As a result, the CDF is given by the following:

$$F_X(x) = \sum_{k=1}^{x} \frac{1}{n} \left(\frac{n-1}{n}\right)^{k-1}$$

However, since nodes are allowed to probe multiple (say s) nodes within a single epoch, the pdf of the best of these probes is given by the first order statistic of the original one-probe pdf:

$$f_{X(1)}(x) = sf_X(x)(1 - F_X(x))^{s-1}$$

$$= \frac{1}{n} \left(\frac{n-1}{n}\right)^{x-1} \left[1 - \sum_{k=0}^{x-1} \frac{1}{n} \left(\frac{n-1}{n}\right)^{k-1}\right]^{s-1}$$

$$= \frac{1}{n} \left(\frac{n-1}{n}\right)^{x-1} \left[1 - \frac{n}{n} \sum_{k=0}^{x-1} \left(\frac{n-1}{n}\right)^{k}\right]^{s-1}$$

$$= \frac{1}{n} \left(\frac{n-1}{n}\right)^{x-1} \left[1 - \frac{1 - \left(\frac{n-1}{n}\right)^{x}}{1 - \left(\frac{n-1}{n}\right)}\right]^{s-1}$$

$$= \frac{1}{n} \left(\frac{n-1}{n}\right)^{x-1} \left[\frac{n-1}{n}\right]^{s-1}$$

$$= \frac{1}{n} \left(\frac{n-1}{n}\right)^{xs-1}$$
\[ f_X(x) = \frac{1}{n-1} \left( \frac{n-1}{n} \right)^{xs} \]  

(5)

Now the expectation of \( f_X(x) \) will give us the expected amount of epochs that each node will take to find its best parent assuming that every node performs a cost relaxation every epoch:

\[
E[f_X(x)] = \sum [xf_X(x)] \\
= \sum \left[ \frac{s}{n-1} \left( \frac{n-1}{n} \right)^{xs} \right] \\
= \frac{s}{n-1} \sum \left[ \left( \frac{n-1}{n} \right)^{xs} \right] \\
= \frac{s}{n-1} \left[ \left( \frac{n-1}{n} \right) \right]^{s-1} \left[ n^s - (n-1)^s \right]^2 \\
= s(n-1)^{s-1} \frac{n^s}{[n^s - (n-1)^s]^2} 
\]

Now because a node performs a cost relaxation with at worst probability \( 1/\lg n \), the expected time in epochs that all nodes will find their best parent is given by:

\[ E(x) = s \lg n (n-1)^{s-1} \frac{n^s}{[n^s - (n-1)^s]^2} \]

In the case \( s = 1 \) (each node only probes one other node in each epoch), this complexity reduces to \( O(n \lg n) \).

### C.2 Partial Convergence Time

Though absolute convergence in the general case takes longer than what we would hope, we note that partial convergence is much faster. That is, when the overlay is far from its optimal value, it makes great advances toward the optimal value. However, as it gets closer and closer to this value, improvements are few in number and small in size. So, it makes sense to speak of partial convergence time as the amount of time it takes to be within some percentage of the optimal value. Let \( 0 < p < 1 \) represent this goodness factor, where \( p \) values close to one yield an almost optimal overlay. Then, assuming uniform distribution of costs and one probe per epoch per node, the probability that a node picks a parent that is within \( 1-p \) of the best parent in the \( x \)th epoch is given by the pdf:

\[ f_X(x) = (1-p)(p)^{x-1} \]  

(6)

The CDF is given by the following:

\[ F_X(x) = \sum_{k=1}^{x} (1-p)(p)^{k-1} \]  

(7)

However, as before, nodes are allowed to probe \( s \) nodes within a single epoch. Hence, the pdf of the first order statistic is given by:

\[
f_X(x) = sf_X(x)(1-F_X(x))^{s-1} \\
= s(1-p)(p)^{x-1}(1 - \sum_{k=1}^{x} (1-p)(p)^{k-1})^{s-1} \\
= s(1-p)(p)^{x-1}(1 - \sum_{k=0}^{x-1} (1-p)(p)^{k})^{s-1} \\
= s(1-p)(p)^{x-1}(1 - \frac{1-p}{1-px})^{s-1} \\
= s(1-p)(p)^{x-1}(p^x)^{s-1} \\
= s(1-p)(p)^{sx-1} 
\]

\[ f_X(x) = s \frac{1-p}{p} (p)^{sx} \]  

(8)

Now the expectation of \( f_X(x) \) will give us the expected amount of epochs that each node will take to find a parent within \( p \) goodness assuming that every node performs a cost relaxation every epoch:

\[
E[f_X(x)] = \sum [xf_X(x)] \\
= \sum \left[ \frac{s}{p} (1-p)(p)^{sx} \right] \\
= \frac{s}{p} \sum [x(p)^{sx}] \\
= \frac{s}{p} \frac{1-p}{(1-p^s)^2} 
\]

Again, because a node performs a cost relaxation with at worst probability \( 1/\lg n \), the expected time in epochs that all nodes will find their best parent is given by:

\[ E(x) = s \lg n \frac{1-p}{p} \frac{(p)^{s}}{(1-p^s)^2} \]

For a probe set size \( s = 1 \), this reduces to \( \frac{\lg n}{1-p} \) or \( O(\lg n) \). We note a significant reduction in (effectively) message complexity of a factor of \( n \) that should bring the convergence time to acceptable levels for an Internet-scale system. For a probe set size of \( \lg n \), this expectation decreases to \( (\lg n)^2 \frac{1-p}{p} \frac{(p)^{\lg n}}{(1-p^\lg n)^2} \) which is bounded above by \( O(\frac{1}{p}) \) yielding constant partial convergence time. This means that, in absence of delay and degree
constraints, every node would find it’s low-cost parent using a constant number of messages.

We believe that this finding supports our claim that probabilistic guarantees (in this case, in form of partial convergence) are more appropriate for large-scale wide-area distributed systems than their absolute counterparts.

C.3 Constrained Degree and Delay

Constraining the degree slows convergence time because nodes are potentially weaned from parents to make room for children that fit better with the parent. However, if we assume that cost is a uniformly distributed value, any loss in cost incurred by a node moving away from a parent is amortized by the gain achieved by the incoming node, yielding a positive global gain. Since weaned nodes will only search for better parents for a constant number of epochs, the overall convergence times are only affected by a constant amount and their upper bounds still hold. Constrained delay is a more complicated matter, however. A node $X$ may be restricted in moving under node $A$ at certain times because of the delay constraint, but may be able to make such a move at a different time (when the delay is not so constrained). As a result, potential moves may become unrestricted as $X$ moves closer to root, thereby requiring $X$ to probe any nodes that were previously restricted. We note that in the static case, the likelihood of this occurring is quite small, limited by the height of the tree which is, of course, limited by the delay bound. Hence we expect the upper bound on cost convergence time to still hold in this case. Future work includes deriving a closed form expression for cost convergence taking into account constrained degree and delay.