Administrative Autonomy in Structured Overlays

Praveen Yalagandula
HP Labs

Mike Dahlin
The University of Texas at Austin

Abstract

Large scale distributed applications spanning multiple domains should satisfy administrative autonomy property — allow users in a domain to control flow of information in to and out of their domains. Though DHTs offer a scalable solution for building distributed applications, they do not guarantee this property. We present a novel Autonomous DHT (ADHT) that guarantees path locality and path convergence in routing to satisfy autonomy property.

1 Introduction

In this paper, we explore administrative autonomy property of distributed applications built on structured overlays like Pastry, Chord, and CAN. For distributed applications deployed in large scale networked systems comprising several administrative domains[1], administrative autonomy property allows users of a domain to control the flow of information coming in and going out of their domain and also ensures availability of the data in the domain irrespective of the behavior of nodes outside the domain. Though DHTs offer solution for scalability with the nodes and the amount of information, most of them do not guarantee the administrative autonomy property.

Administrative autonomy is a key requirement for many distributed applications for security, availability, and efficiency. We further discuss this point with reference to Figure 1 where we present routing for a key 111XX in a bit-correcting DHT like Pastry [9].

Security: Consider a file location system on Pastry in an enterprise network. While payroll files should be locatable by employees of payroll department, they should not be exposed to employees/machines outside the department. Using scalable overlay networks such as Chord, Pastry, etc. do not provide any control over where the data is placed and do not provide any guarantees that the queries for information in a domain are not exposed outside that domain. In the example, queries related to key 111XX in domain dep1 will be exposed outside that domain.

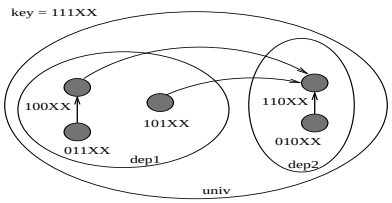


Figure 1: Administrative autonomy violation in a bit-correcting DHT.

Availability: Domain disconnections or organizational partitions are common in the Internet. Also a node in a domain can behave maliciously either by responding lazily for messages from nodes outside the domain or by dropping messages from nodes outside the domain. For example, for nodes in domain dep1, such domain disconnections or malicious behavior of node with key 110XX in another domain dep2 can potentially decrease the availability for operations within the domain.

Efficiency: Application like multicast systems and aggregation systems use overlays for constructing trees. Since domain-nearness also implies network proximity in many cases, administrative autonomy results in efficient trees in contrast to a bit-correcting DHT overlay routing. Consider building multicast application on top of a bit-correcting DHT. For a session corresponding to key 111XX shown in Figure 1, note that two nodes with ids 100XX and 101XX in domain dep1 connect to node 110XX in another domain for receiving multicast data.

To achieve administrative autonomy, a DHT routing algorithm should satisfy two properties: (i) **Path Locality**: Routing paths should always be contained in the smallest possible domain, and (ii) **Path Convergence**: Routing paths for a key from two different nodes in a domain should converge at a node in the same domain. Existing DHTs either already support path locality [4] or can support easily by setting the domain nearness as the distance metric [2, 3]. But they do not *guarantee* path convergence as those systems try to optimize the path length to the root to reduce response latency. As motivated above, we believe that guaranteeing path convergence in overlay networks and satisfying administrative autonomy will enable the industry to embrace the DHT work in real applications.

[1]Domain in our system is defined as a set of machines either administered by a common authority or a logical group with in such sets (e.g., set of machines sharing a switch). Note that these domains does not necessarily correspond to the DNS domain hierarchy even though we use a similar notation.

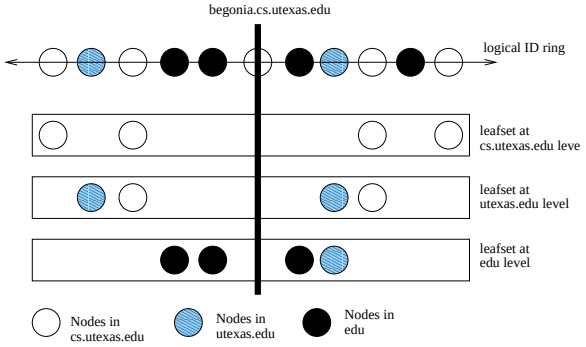


Figure 2: Multiple leafsets maintained by the node `begonia.cs.utexas.edu` (leafset size=4)

In a previous paper, we presented a brief description of initial design of a novel Autonomous DHT (ADHT) that satisfies administrative autonomy. This paper presents more details of the ADHT design that builds on a bit-correcting DHT, Pastry [9], by maintaining multiple leafsets at each node and using novel join and key routing algorithms. We also present a new *Zippering* technique for maintaining consistent routing with multiple leafsets in the face of node joins, node leaves, and network partitions.

2 Our Approach

To ensure path convergence, a DHT routing must provide a single exit point in each domain for a key. And for path locality, a DHT routing protocol should route keys along intra-domain paths before routing them along inter-domain paths. Below, we describe the following aspects of ADHT design – data structures maintained at each node, routing and join algorithms, and finally a novel *Zippering*[2] algorithm employed by ADHT for maintaining consistency in data structures.

2.1 Data Structures

Similar to Pastry and other DHT algorithms, each node in ADHT has a routing table to maintain pointers to $O(\log N)$ other nodes in the system. In contrast to one leafset in Pastry, each node in ADHT maintains a separate leaf set for each domain to which the node belongs. In Figure 2, we illustrate leafsets maintained by a node `begonia.cs.utexas.edu` in the ADHT algorithm. Note that the Pastry algorithm maintains just one leafset – corresponding to the top domain `edu` level. Maintaining a different leafset for each level increases the number of neighbors that each node tracks to $(2^b) * \lg_{2^b} n + c.l$ in ADHT compared to $(2^b) * \lg_{2^b} n + c$ in unmodified Pastry, where b is the number of bits in a digit, n is the number of nodes, c is the leafset size, and l is the number of domain levels. But these extra leafsets ensure path locality and convergence properties during routing.

[2]Terminology from [10]

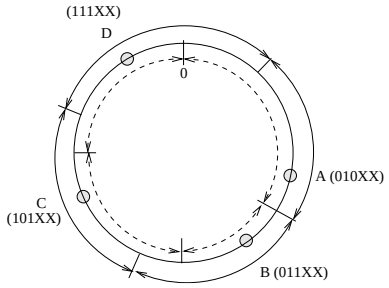


Figure 3: Key space assignment to nodes in Pastry (outer split) and ADHT (inner split).

2.2 Routing in ADHT

The algorithm for populating the routing table is quite similar to Pastry but with the following key difference: ADHT uses hierarchical domain proximity as the primary proximity metric (two nodes that match in i levels of domain hierarchy are more proximate than two nodes that match in fewer than i levels in domain hierarchy) and network distance as the secondary proximity metric (if two pairs of nodes match in the same number of domain levels, then the pair whose separation by network distance is smaller is considered more proximate).

ADHT uses a novel key space assignment to nodes so that routing paths do not visit a node twice during routing for any key — a property required so that we can extract aggregation trees from the routing structure. A key k is assigned to a node A such that ID_A matches more prefix bits of k than any other node's ID. If IDs of multiple nodes match key k in the same number of prefix bits, then we pick a node B from that set such that $dist(k, ID_B) = MIN(|k - ID_B|, 2^b - |k - ID_B|)$ is smaller than difference between key k and any other node's ID. Note that Pastry only uses $dist$ function to decide the key space assignment. The key space split is shown for ADHT and Pastry in Figure 3 for four nodes A, B, C, and D. Below we first explain the routing algorithm and then discuss how this key assignment ensures no loops in ADHT key routing.

Routing Algorithm The routing algorithm we use in routing for a key at node with *nodeId* is shown in Algorithm 1. To route a key k, a node A with ID ID_A first checks its routing table for another node that matches the key in more digits than this node. We call such bit correcting neighbor a *flipNeighbor*. If no such node exists, then we consider leafsets starting from the smallest domain. If a *flipNeighbor* exists and is in the node's lowest domain, then we route the key to the *flipNeighbor*. If a *flipNeighbor* exists and is not in the same domain as the node, then we consider leafsets corresponding to the levels below the common

Algorithm 1 ADHTroute(key)

```
1:  flipNeigh ← checkRoutingTable(key) ;
2:  l ← numDomainLevels ; /* node's lowest level */
3:  while (l >= 0) do
4:    if (commonLevels(flipNeigh, node) == l) then
5:      send the key to flipNeigh ;
6:      return
7:    else
8:      leafNeigh ← an entry in leafset[l] closer to
        key than nodeId ;
9:      if (leafNeigh ! = null) then
10:       send the key to leafNeigh ;
11:       return
12:     end if
13:   end if
14:   l ← l − 1; /* move to next higher domain */
15: end while
16: /* this node is the global root for this key */
```

domain between the *flipNeighbor* and this node, starting from the lowest domain leafset. For example, if a node `begonia.cs.utexas.edu` finds a *flipNeighbor* `linux1.cs.cmu.edu`, then the node considers the leafsets at levels `cs.utexas.edu` and `utexas.edu` in that order. If the node finds another node in its leafset that is closer to the key than the node, then it forwards the key to that node closer to the key. If no such node is found in a leafset at a level, then this node is considered the *root* node for key k in that domain. If a node has no *flipNeighbor* for a key k and has no neighbor in any leafset at any level that is closer to the key k than it is, then such node is the global root for key k. Note that by routing at the lowest possible domain until the root of that domain is reached, we ensure that all routing paths starting in a domain converge within that domain, thus achieving the Path Convergence property.

Discussion The key space assignment in ADHT ensures that no node is visited twice during routing – a key requirement for many applications on DHTs that extract trees from DHT routing such as for aggregation in SDIMS [12] and for spanning trees in multicast. If we use the Pastry key assignment, then the routing paths using the ADHT routing algorithm might touch a node more than once. For example, consider routing for a key k =000XX in a domain with four nodes shown in Figure 3 and suppose the whole system has several other nodes in other. In this domain, node D is considered as the root for this key with Pastry's key assignment. If we start routing for key k from node B, node B will forward the route request to node D as that is the root. But, when routing in the next domain level from node D, the next hop goes back to node B as it is the nearest first bit-correcting node. Thus node B will be visited twice and hence can create loops in the routing. The ADHT key space assignment assigns node B as the root for this domain in this case and hence it is not touched more than once.

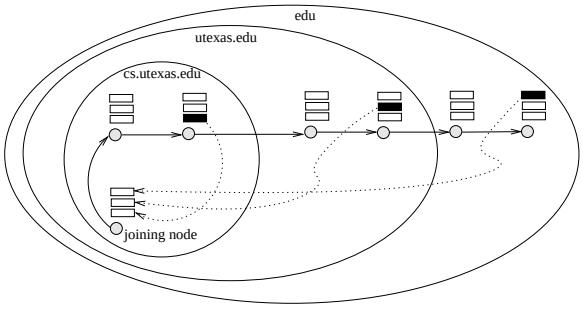


Figure 4: Leafsets that a joining node receives in response to its join request. Dark arrows denote the join request routing path.

2.3 Join Algorithm

Similar to Pastry join algorithm, a node joins an existing ADHT by contacting one or more user supplied nodes. But, instead of bootstrapping from that node, the joining node uses the contact node to search for an appropriate bootstrap node that is closer to the joining node in terms of the domain-nearness metric. This is to ensure that leafsets at different domain levels are filled in a correct manner. In the following paragraphs, we will first describe how a node joins once an appropriate bootstrap node is found and then describe how a joining node finds such an appropriate bootstrap node.

Suppose node A is joining ADHT using a bootstrap node B. Node A asks node B to route a special *join* message with key $k = ID_A$. ADHT then routes the join message to an existing node R that is currently responsible for key k. Each intermediate node C on the routing path from node B to node R sends its leafsets for each common domain between C and A for which node C is the root node in that domain for key k. Figure 4 illustrates the leafsets that a joining node receives in response to its join request in a three level deep domain hierarchy case. Also the intermediate nodes send their routing table to node A so that node A can initialize its routing tables. Finally, node A informs all nodes that need to be aware of its arrival by sending entries in its routing or leafset tables. This procedure ensures that node A initializes its state with appropriate values and that the state in all other affected nodes is updated.

2.3.1 Finding a bootstrap node

A joining node needs to find a node already in the ADHT that is closer to the joining node in terms of domain-nearness. For example, node `begonia.cs.utexas.edu` that wishes to join a ADHT

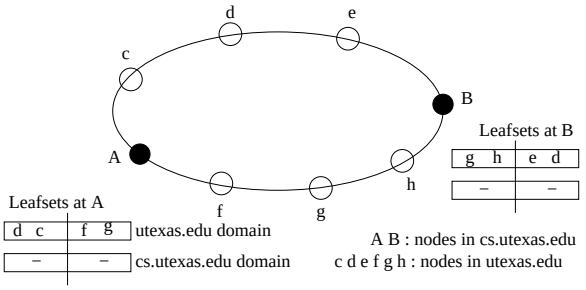


Figure 5: Concurrent joins leading to path convergence property violations. Nodes A and B join concurrently leading to incorrect `cs.utexas.edu` domain leafset tables.

searches for some other node in `cs.utexas.edu` domain. If no such node exists, then it looks for a node in `utexas.edu` domain, and so on. The joining node uses such a near node as its bootstrap node for joining the ADHT. While such appropriate bootstrap node can be provided to a joining node manually in small scale systems, it will be infeasible for large scale systems in practice. Below, we describe how ADHT leverages its *put* and *get* operations to locate bootstrap nodes.

Using DHT `put` and `get` operations: Each node after joining the ADHT will perform a `put` operation for keys corresponding to different domains to which this node belongs. For example, a node `begonia.cs.utexas.edu` will perform three put operations for keys corresponding `cs.utexas.edu`, `utexas.edu`, and `edu` with its own IP address as the value. Now, a new node, say `linux1.ece.utexas.edu` that wishes to join the DHT will use a supplied contact node to perform a sequence of `get` operations starting from the lowest domain `ece.utexas.edu` up to the highest domain `edu` until it finds another node. If no node is found, then it considers itself to be the first node in the `edu` domain and uses a user-specified contact node as the bootstrap node.

Note that storing IP addresses of all nodes in a domain is unnecessary and might be infeasible for large domains. We do not need a `get()` operation to return all IP addresses stored for a domain but *any* or *few* values will suffice. So, instead of storing all values, we store only a few for each domain and use a FIFO policy to purge the list as new IP addresses are inserted.

2.4 Maintaining Consistent Leafsets

Note that although mechanisms described in the previous section provide one way for rendezvous between nodes in same domain, they do not guarantee that a new joining node always finds other nodes in its domain. For example, if nodes in a new domain concurrently join an existing ADHT, they might not find each other during the join phase and might use some node outside their domain as the bootstrap node, which can leaf to incorrect leafset state. We illustrate this in Figure 5(a) where two nodes A and B in domain `cs.utexas.edu` join concurrently using some nodes outside `cs.utexas.edu` as bootstrap nodes. Hence, the leafset tables of these nodes end up in an incorrect state leading to the violation of path convergence. In our system, we ensure path convergence and path locality properties are met by using the following mechanism — each node periodically searches for other nodes in all domains that it is part of using DHT-based method, contacts those nodes, and corrects any incorrect entries in its routing table or leafset tables. In the following, we first describe the *Zippering* mechanism that is useful for any DHT to handle partitions and then describe how we use this to achieve leafset consistency in ADHT.

Zippering for Mending Partitions Several DHT systems, like SkipNet [5] and Willow [10], provide a way to merge partitioned components. In Pastry, nodes periodically perform a leafset maintenance task where each node checks for the liveness of its leafset table entries and broadcasts its current leafset to members of that leafset if it finds any dead entries. Though this maintenance protocol is appropriate to mend machine crash failures, it fails during network partitions, leading to possible partitions in the DHT. Even after the network heals, Pastry does not have a mechanism where these different partitions can merge back together.

To mend partitions in a DHT, we need a way to rendezvous between nodes in those partitions and a way to merge a partition with another after a node in a partition discovers a node in the second partition. In ADHT, nodes keep a log of dead nodes and occasionally ping them to check if they became alive. The entries in the log expire after a certain timeout period, $T_{deadNodePurge}$. Also we provide an interface for manually initiated rendezvous.

We describe the zippering mechanism with an example illustrated in Figure 6. We show the partitions as two different circles corresponding to the logical ID space and show nodes in different partitions on those two different circles. In this figure, node A with say ID_A initially discovers node B and starts a join procedure using node B as the bootstrap node (message 1 in the figure). Node B forwards the message using ADHT routing towards the root for key ID_A, which is routed to node R that is currently responsible for key ID_A in the partition to which node B belongs. If there is no partition, then the join request will be routed to node A and the procedure stops. In case of a partition, node R returns response for join request to node A and node A gathers that R is in a different partition. Then, Node A contacts and exchanges leafsets with node R so that both of them are in the same partition. Thus the logical

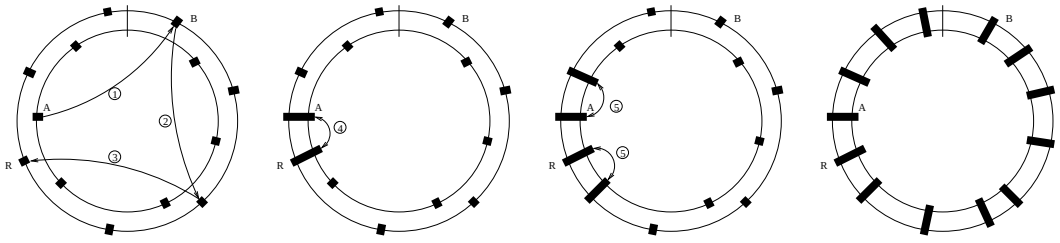


Figure 6: Zippering steps: (a) Node A starts a join procedure using node B as the bootstrap node. Node B routes that request towards node R which is the current root for ID_A in B's partition. (b) Node A and Node R detect the partitions and exchange their leafsets. (c) Node A and Node R propagate the information about partitions during their periodic leafset exchanges with neighbors. (d) Eventually, the partitions are merged together.

ID space near node A and node R is mended (depicted in the figure as node A and node R participating in both rings). As node A and node R perform their periodic leafset exchanges with neighbors in their leafsets, the mending of logical ID space spreads around the ring (shown in Figure 6(c)) and eventually the partitions are zippered together (depicted in the Figure 6(d)).

Fast Zippering The method for zippering in the last few paragraphs can take $O(N)$ time steps before two partitions with N nodes are merged together. Note that the healing of the partitions is spread around the ID ring linearly in time. To hasten the zippering process, we propose the following scheme. Upon detecting partitions and mending leafsets, a node picks a constant number of random nodes in its current partition (from its previous leafsets and routing table) and informs them about the nodes in the other partition (new entries in the leafset). These nodes then start zippering procedure to mend their leafsets. With each node informing a constant number of other nodes after it completes zippering, all nodes will get to know of partitions in $O(\log N)$ such rounds with high probability. Overall, all nodes will complete performing zippering step by $O(\log^2 N)$ time steps. This procedure will incur $O(N \log N)$ messages as each node might perform the zippering step in contrast to $O(N)$ messages in slow zippering procedure described previously.

Leafset Maintenance using Zippering In ADHT, we also use the above zippering mechanism to correctly maintain leafsets at different levels. Periodically each node looks for other nodes in each domain it is part of using the DHT-based method described in the previous section. Once it finds such nodes, it uses the zippering mechanism to mend any possible partitions. An important question is how frequently should ADHT nodes perform this discovery step. If all nodes perform such operation quite frequently, then few chosen representatives for a large domain will be inundated with a large number of join requests. To avoid such hot spots, the frequency for discovery step at a node in a domain is set to be proportional to the estimated number of nodes in the domain. To estimate the size, we leverage the following result from Viceroy [7]: If X_s denote the sum of segment lengths (length of logical ID space) managed by any set of s nodes, then $\frac{s}{X_s}$ is an unbiased estimator for the number of nodes.

3 Properties

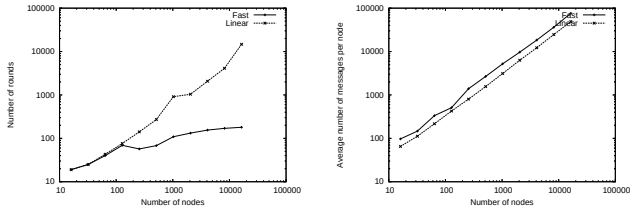
Correctness: We explore the correctness of ADHT from the perspective of three properties — (1) Consistent Routing [1]: A lookup operation for a key always ends at the current root node responsible for the key in the system, (2) Path Convergence, and (3) Path Locality. For brevity, in the following, we only present lemma and theorem statements. Please refer to our extended report [11] for more details.

Lemma 1 *Consistent leafsets at all nodes guarantee consistent routing, path convergence, and path locality.*

Lemma 2 *If the global leafset at all nodes is consistent and after the system becomes stable (no further node and network failures), eventually all leafsets at different domain levels at all nodes become consistent.*

Theorem 1 *If all network partitions and node failures are of duration less than $T_{deadNodePurge}$ and after the system is stable, eventually consistent routing, path convergence, and path locality properties are met.*

Increased Path Length in ADHT: In contrast to Pastry routing, ADHT routes to a domain root node before jumping out of the domain to ensure path convergence. In Pastry routing, when an entry is found in the routing table that is closer to the key than the current node, then the request is forwarded to that node. In ADHT, entries in a domain leafset are given priority over a routing table entry when the routing table entry corresponds to a node outside the domain. This routing procedure leads to an increase in the hop count for reaching a root node for a key from any node in the system. This path length increases to $O(\log N + l)$ from $O(\log N)$ in Pastry, where N is the number of nodes in the system and l is the number of levels in the administrative hierarchy,



(a) (b)

Figure 7: Fast vs slow zippering mechanisms (a) Time taken (number of simulation steps) to achieve leafset consistency and (b) Communication costs.

which in practice will grow no faster than $\log N$.

4 Simulation Results

For demonstrating the performance of ADHT Zippering, we build a DHT ensuring that nodes in the DHT get partitioned into two equal sized partitions. Then we inform the nodes of a partition about a node in the second partition (simulating the node discovery mechanism mentioned in Section 2.4), which starts the zippering activity. In Figure 7, we plot performance results for fast zippering and compare it with slow linear zippering. We measure performance as time taken in terms of simulation steps and number of messages incurred during zippering. As expected, fast zippering mends partitions quickly while incurring a modest increase in the number of messages.

5 Related Work

SkipNet [4] provides domain restricted routing where a key search can be limited to a specified domain. This interface can be used to ensure path convergence by searching in the lowest domain and moving up to the next domain when the search reaches the root in the current domain. Although this strategy guarantees path convergence, this routing might touch a node more than once (as it searches forward and then backward to stay within a domain) and hence renders it unsuitable for applications extracting trees from DHT routing (e.g.,overlay multicast).

An alternative solution to achieve administrative autonomy is through splitting id space among domains [13], but this approach needs an estimate of the size of the domains before the id space can be split which will be hard to obtain accurately in practice. Mislove et al. propose multiple DHT rings [8] for administrative control and autonomy in Pastry. Each domain has a separate DHT ring and a chosen gateway node from each domain form next higher level DHT ring. An extension of Chord considers multiple virtual rings [6] focusing on efficiently supporting multiple subgroups using an existing Chord ring. Similar mechanisms to Zippering are proposed for handling organizational disconnects in SkipNet [5] and for merging two separately formed DHTs in Willow [10].

6 Summary and Open Issues

Administrative autonomy is an important requirement to satisfy in any distributed system spanning multiple administrative domains for security, availability, efficiency. Path Locality and Path Convergence are two properties that a DHT should guarantee for satisfying autonomy. Current DHT algorithms can achieve path locality but do not guarantee path convergence. In this paper, we have described a novel Autonomous DHT that guarantees both properties.

An open question for discussion is how to achieve Administrative Autonomy in other DHT systems. We believe our techniques are applicable in other bit-correcting DHTs like SkipNet and logical ring based DHTs like Chord. For example, we can achieve path convergence in Chord by maintaining multiple leafsets and use a routing algorithm similar to ADHT routing algorithm – route to a node in the lowest domain closer to the key before jumping out of the domain.

References

[1] M. Castro, M. Costa, and A. Rowstron. Performance and Dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, MSR Cambridge, UK, 2003.

[2] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. Technical Report MSR-TR-2002-82, MSR.

[3] K. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *SIGCOMM*, 2003.

[4] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, March 2003.

[5] N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Efficient Recovery From Organizational Disconnect in SkipNet. In *IPTPS*, 2003.

[6] D. R. Karger and M. Ruhl. Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks. In *IPTPS*, 2004.

[7] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *PODC*, 2002.

[8] A. Mislove and P. Druschel. Providing Administrative Control and Autonomy in Peer-to-Peer Overlays. In *IPTPS*, 2004.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Middleware*, 2001.

[10] R. VanRenesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *IPTPS*, 2004.

[11] P. Yalagandula and M. Dahlin. Administrative Autonomy in Structured Overlays. `http://www.cs.utexas.edu/users/dahlin/projects/sdims/`.

[12] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.

[13] S. Zhou, G. Ganger, and P. Steenkiste. Balancing Locality and Randomness in DHTs. Technical Report CMU-CS-03-203, CMU, 2003.