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 [13].

[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.

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.

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

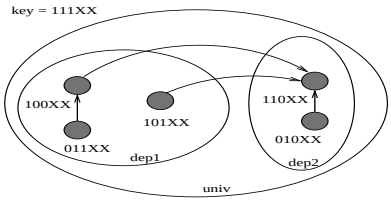


Figure 1: Administrative autonomy violation in a bit-correcting DHT.

support path locality [6] or can support easily by setting the domain nearness as the distance metric [2, 5]. 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.

In this paper, wqe present a novel Autonomous DHT (ADHT) that guarantees path locality and path convergence in routing. ADHT builds on a bit-correcting DHT, Pastry [13], 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.

Our current ADHT design focuses on enhancing a bit-correcting DHT, Pastry [13], to satisfy administrative autonomy. While we choose Pastry for convenience—the availability of a public domain implementation, we also believe that our techniques could be applied to many existing DHT implementations to support path locality and path convergence properties.

In the following section, we presen the design of our system. In Section 3, we discuss the correctness and performance properties of ADHT. We present some experimental results quantifying the usefulness of ADHT algorithm in Section 4. In Section 5, we detail the related work. Finally, we summarize the paper and present open issues in Section 6.

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.

We build a novel DHT called Autonomous DHT (ADHT) that builds upon the Pastry algorithm in the following ways.

1. Data structures: Instead of one leafset in the Pastry algorithm, nodes in ADHT maintain a separate leafset for each administrative hierarchy domain to which a node belongs. We specify a node's position in the administrative hierarchy using similar notation as the Domain Name Service (DNS) [12].

2. Routing algorithm: ADHT uses a novel routing algorithm that ensures that the routing path for a key reaches the root node in a domain before it jumps out of the domain; thus, achieving path convergence and path locality properties. ADHT also 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. We also introduce a two level locality model that incorporates both administrative membership of nodes and network distances between nodes.

3. Join algorithm: To correctly fill multiple leafsets at a joining node, ADHT uses a join algorithm similar to Pastry's join algorithm but uses an appropriate bootstrap node — a node already in the system that is closest to the joining node in terms of domain-nearness.

4. *Zippering*[2] to maintain leafsets: ADHT employs a zippering mechanism to maintain consistent leafsets at all domain levels at all nodes.

In the following sections, we describe our ADHT algorithm in detail, mainly focusing on the four points mentioned above.

[2]Terminology from [14]

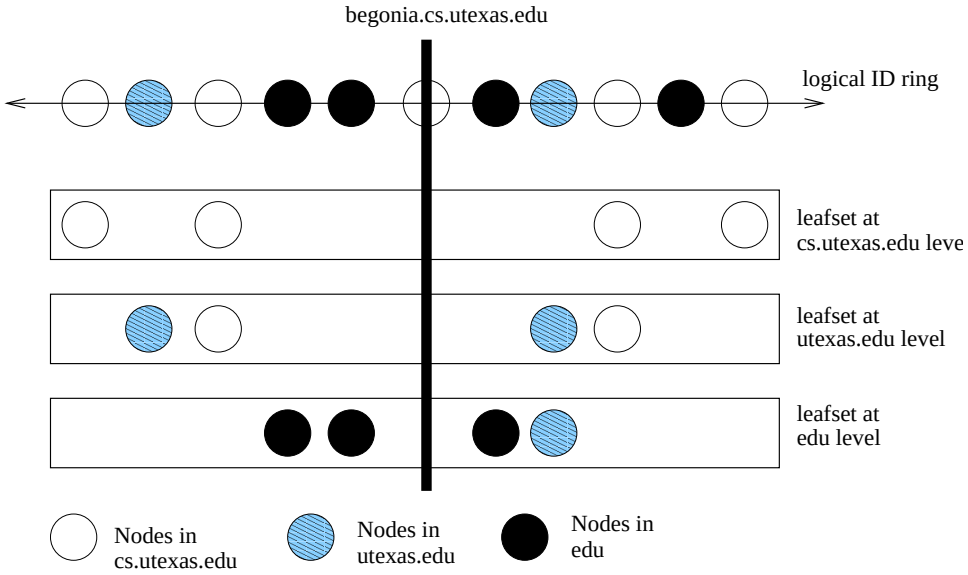


Figure 2: Multiple leafsets maintained by the node `begonia.cs.utexas.edu` (leafset size=4)

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.

2.2 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).

2.2.1 Key space assignment to nodes

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.

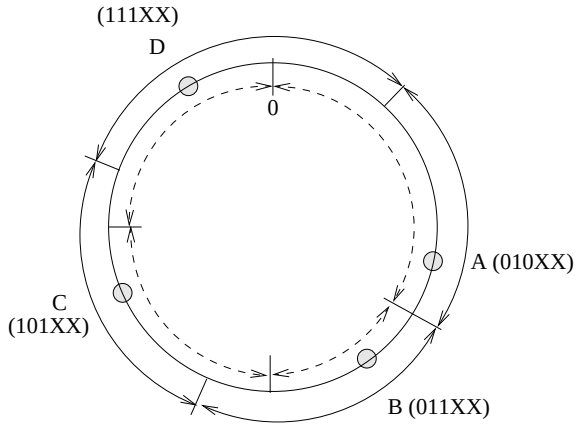


Figure 3: Key space assignment to nodes in Pastry (outer split) and ADHT (inner split).

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 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.

Algorithm 1 ADHTroute(key)

```
1:  flipNeigh ← checkRoutingTable(key) ;
2:  l ← numDomainLevels ; /* number of levels
    in this node's hierarchical name. For exam-
    ple, node begonia.cs.utexas.edu is 3 levels
    down in the domain hierarchy */
3:  while (l >= 0) do
4:    /* commonLevels returns number of com-
      mon levels between flipNeighbor and this
      node; if flipNeighbor is null, it returns -1 */
5:    if (commonLevels(flipNeigh, nodeName)
      == l) then
6:      send the key to flipNeigh ;
7:      return
8:    else
9:      leafNeigh ← an entry in leafset[l] closer
        to key than nodeId ;
10:     if (leafNeigh ! = null) then
11:       send the key to leafNeigh ;
12:       return
13:     else
14:       /* this node is the root for this key in this
          domain */
15:     end if
16:   end if
17:   l ← l − 1; /* move to next higher domain */
18: end while
19: /* this node is the global root for this key */
```

2.3 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 [15] 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.

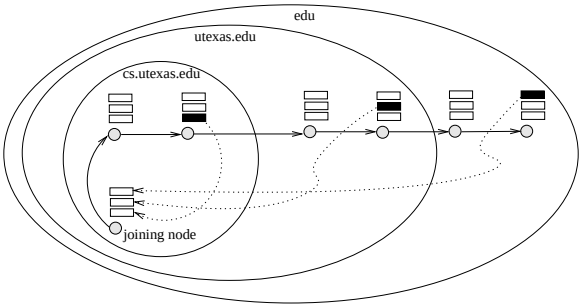


Figure 4: Leafsets that a joining node receives in response to its join request. Dark arrows denote the join request routing path.

2.4 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.

2.4.1 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 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.

A solution is to provide such a bootstrap node manually. An administrator installing ADHT on her set of machines can manually specify the bootstrap nodes. Though this approach is often reasonable in an enterprise type setting where system administration is done through an IT department, it might be infeasible in a university type setting where individual departments have their own system administrators. In the latter case, different department administrators need to coordinate when setting up the first machine in their respective domains. Otherwise, if the first machines in all departments simultaneously join the system and use a node outside the university, then partitions can occur among the machines in the university. 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 per-

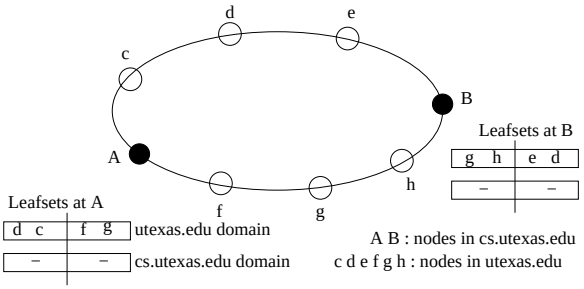


Figure 5: Concurrent joins leading to path convergence property violations. Nodes A and B in `cs.utexas.edu` join concurrently using nodes in `utexas.edu` domain as bootstrap nodes. Observe the incorrect leafset tables at node A and B corresponding to `cs.utexas.edu` domain.

form 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, by default, a DHT `put(key, value)` operation appends the `value` to other existing values stored for that key. This approach might be unscalable as the node responsible for the key corresponding to a large domain has to store a large number of values. 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.

2.5 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 lead to an 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 [7] and Willow [14], 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. This mechanism will ensure the rendezvous between nodes in different partitions after network failures lasting shorter than $T_{deadNodePurge}$. For longer network failures, we provide an interface for an administrator to initiate the rendezvous. This interface also allows administrators to merge two separately formed DHTs into a single DHT.

We describe the zippering mechanism with an example illustrated in Figure 6. We show the partitions as two different circles corresponding to the logical

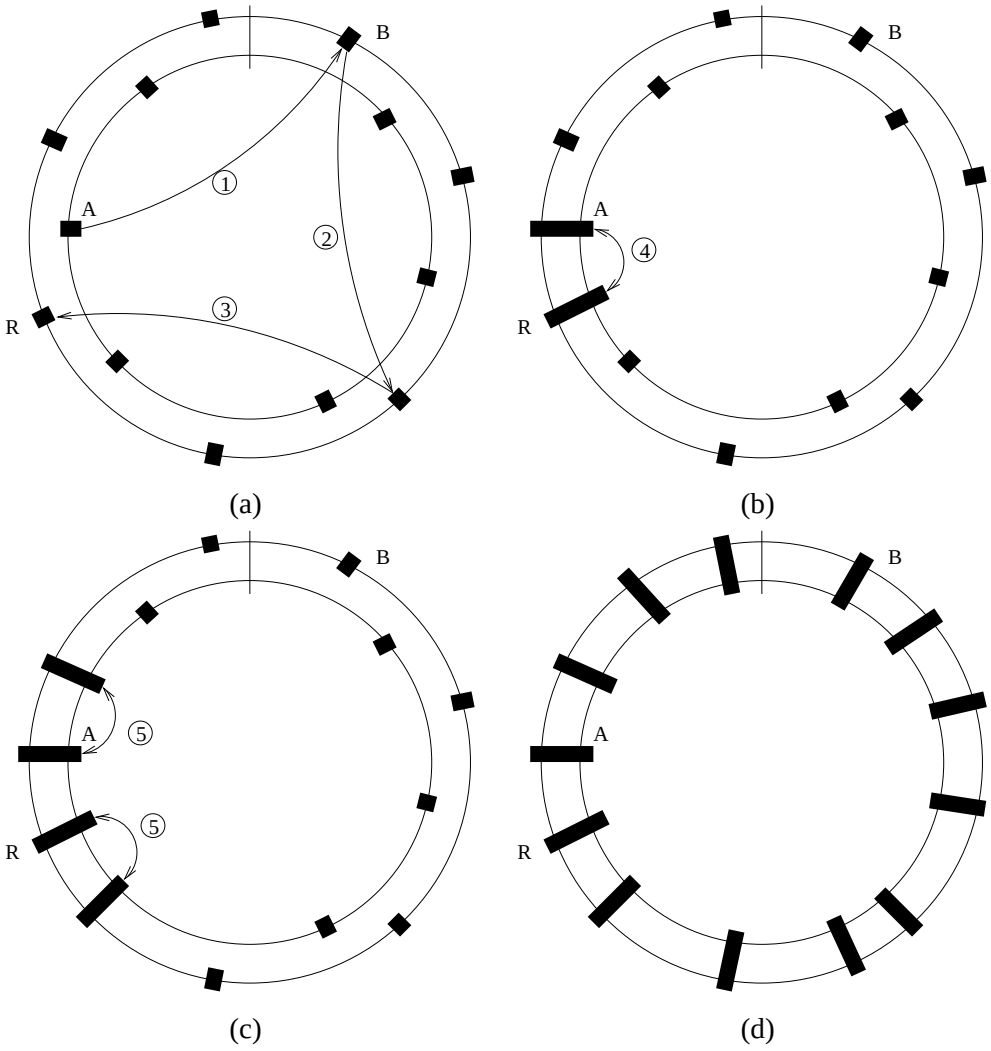


Figure 6: Zippering steps: Node A in a partition discovers node B and starts the zippering procedure. (a) Node A with ID ID_A starts a join procedure using node B as the bootstrap node. Node B routes that request towards node D which is the current root for ID_A in B's partition. (b) Node A and Node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

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 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 [9] and Symphony [10]: 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

In this section, we discuss the correctness and performance properties of the ADHT algorithm.

3.1 Correctness

We discuss 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.

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

Note that the key assignment we use in ADHT always ensures that each key is assigned to either the nearest node on the left or the nearest node on the right side on the logical ID ring. This implies that to achieve consistent routing we just need to ensure that each node correctly knows its current left and right neighbor on the logical ID ring corresponding to the global domain. Hence, consistent leafsets at all nodes guarantee the consistent routing property.

The ADHT routing procedure shown in Algorithm 1 works from the lowest domain level of a node and gives preference to a node found in the leafset at a lower domain level that is closer to the key over a node found in the routing table in a higher domain. Effectively, the routing table entries are used as shortcuts but the leafset entries are used to ensure that a node closest to the key in a domain is reached before the route jump outs of a domain. Hence, with correct leafsets at all nodes, path convergence and path locality properties are satisfied.

In the context of the aggregation framework, we mainly care about eventual path convergence guarantees in the DHT routing layer. Note that disconnected components do not violate administrative isolation — domain restriction option in the install and the probe API allows users and applications to restrict the propagation of queries and updates to desired domains. But during the time when path convergence guarantee is not met, nodes in a domain

may not be able to aggregate information about *all* nodes in that domain. But once the property is met in a domain, the aggregate will reflect the values at all nodes in the domain.

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.*

Even when global leafset is consistent, other domain level leafsets can be inconsistent due to partitions in the domain (example in Figure 5. But in ADHT, each node looks for other nodes in the same domain periodically and performs a zippering step. When the global leafset is consistent and the system is stable, if we have a partition in nodes of a domain, then at least one node in one of the partitions will discover a node in another partition. The zippering step following such discovery attaches those two partitions into one and the periodic leafset exchange procedure ensures that the leafset corresponding to this domain on all nodes in those both partitions become consistent.

Lemma 3 *If all network partitions and node failures are of duration less than $T_{deadNodePurge}$ (the timeout period after which a dead node is removed from the list of dead nodes at a node in ADHT) and after the system becomes stable, the global leafset becomes consistent eventually.*

In ADHT, nodes keep track of recently failed nodes and ping them periodically to check their liveness. If found to be alive, they perform zippering step to include those nodes in the DHT. Hence, when all network partitions and node failures are of a finite duration less than the timeout period used for purging a node from the dead node list on each node, $T_{deadNodePurge}$, and after the system becomes stable, we will have a correct global leafset table in a finite time.

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.*

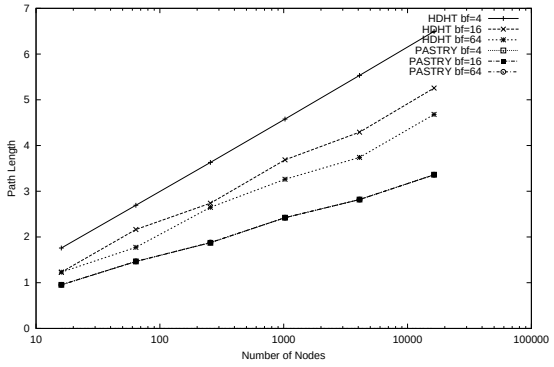


Figure 7: Average path length to root in Pastry versus ADHT for different branching factors. All Pastry lines overlap as the branching factor does not effect the Pastry routing procedure.

3.2 Performance

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

4 Experimental Evaluation

Though the routing protocol of ADHT might lead to an increased number of hops to reach the root for a key as compared to the original Pastry, the algorithm conforms to the path convergence and locality properties and thus provides administrative isolation. In Figure 7, we quantify the increased path length by comparisons with unmodified Pastry for different sized networks with different *branching factors* of the administrative hierarchy. The branching factor

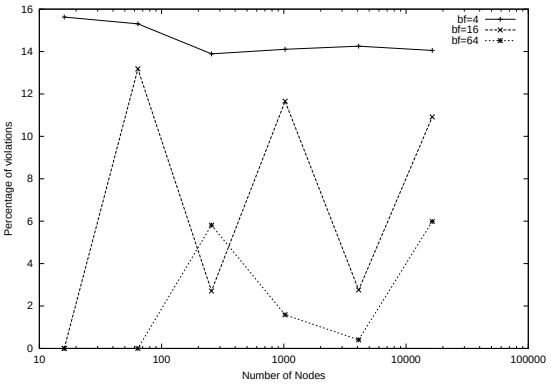


Figure 8: Percentage of probe pairs whose paths to the root did not conform to the path convergence property in Pastry. We do not show lines for ADHT as paths of all probe pairs conform to the path convergence property in ADHT.

denote the maximum degree of each internal node in the hierarchy. Note that the number of levels in the administrative hierarchy for an N node system with a branching factor b is $log_b N$. All lines corresponding to the Pastry overlap as the branching factor does not affect the performance of the original Pastry. Observe that the difference between average path length in ADHT compared to the path length in Pastry increases with decreasing branching factor because the depth of the administrative hierarchy increases with decreasing branching factor.

To quantify the performance of Pastry and ADHT with respect to the path convergence property, we perform simulations with a large number of probe pairs — each pair probing for a random key starting from two randomly chosen nodes. In Figure 8, we plot the percentage of probe pairs that did not conform to the path convergence property. When the branching factor is low, the domain hierarchy tree is deeper and hence a large difference between Pastry and ADHT in the average path length; but it is at these small domain sizes that the path convergence fails more often with the original Pastry.

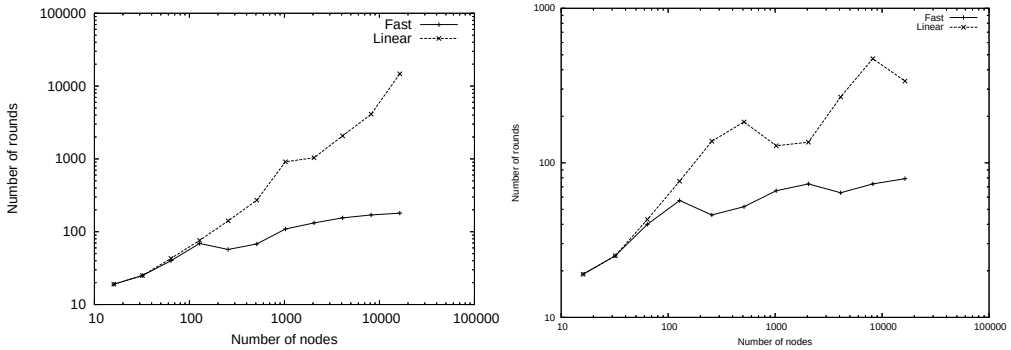
4.1 Zippering

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 some nodes of a partition about a node in the second partition (simulating the node discovery mechanism mentioned in Section 2.5), which starts the zippering activity. In Figure 9, we plot performance results for fast zippering described in Section 2.5 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. We plot for two cases where in one case only one node discovers a node in the other partition and in another case where randomly chosen 1% nodes in a partition discover a node in the other partition. As described earlier, fast zippering mends partitions in $O(\log^2 N)$ steps in contrast to slow zippering which can take $O(N)$ steps; in terms of the number of messages, fast zippering incurs $O(N \log N)$ messages in contrast to $O(N)$ messages in slow zippering. Note that in ADHT, all nodes actively search for nodes in other partitions. Hence, more than one node might discover the partition and start the mending process. In the same figure, we also plot the metrics comparing fast and slow zippering when the mending is initiated by one percent of the nodes in the system. Note that when an f fraction of nodes start the zippering process, then mending of partitions takes $O(1/f)$ simulation steps in slow zippering and $O(\log^2(1/f))$ in fast zippering.

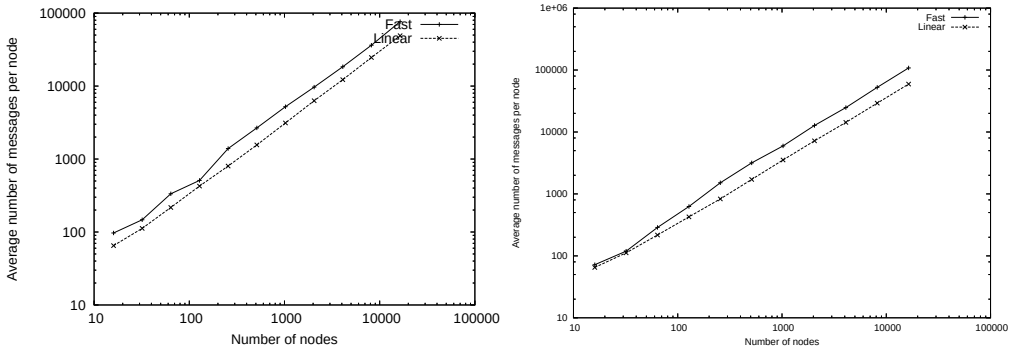
5 Related Work

Internal DHT trees typically do not satisfy domain locality properties required in our system. Castro et al. [2] and Gummadi et al. [5] point out the importance of path convergence from the perspective of achieving efficiency and investigate the performance of Pastry and other DHT algorithms, respectively. In the later study, domains of size 256 or more nodes are considered and their studies show that path convergence is satisfied with high probability. In SDIMS, we expect the size of administrative domains at lower levels to be much less than 256 and it is at these small sizes that the path convergence fails more often (Refer to Graph 8 — smaller branching factors incur higher percentage of violations).

SkipNet [6] provides domain restricted routing where a key search can be limited to a specified



(a)



(b)

Figure 9: Performance of ADHT in merging two equal sized partitions in two cases — when only one node of a partition discovers a node in another partition and when 1% of nodes in a partition discover a node in another partition. We compare performance in case of both fast and slow zippering mechanisms described in Section 2.5. (a) Time taken (in terms of number of simulation time steps) to achieve leafset consistency. (b) Communication cost incurred for leafset consistency.

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, it loses the aggregation tree abstraction property of DHTs as the domain constrained routing might touch a node more than once (as it searches forward and then backward to stay within a domain). Also the search can be quite inefficient as it searches linearly through the nodes in a domain once the search reaches a node in the required domain.

An alternative solution to achieve administrative autonomy is through splitting id space among domains [16], 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. build multiple rings [11] to provide administrative control and autonomy in structured peer-to-peer networks. Nodes in a administrative domain form a ring and few nodes of a domain act as gateways for that domain and participate in a ring corresponding to the next higher domain. Each ring is assigned an ID and lookups involve both a key and a ring ID. To enable locating a gateway responsible for a ring ID, gateways of a ring advertise the corresponding ring ID in higher level rings using standard DHT `put` interface. In contrast to ADHT, this requires that a node, say acting as gateway at all levels, participate in $O(l)$ separate DHTs implying a maintenance overhead of $O(l.\log N)$, where l is the number of levels in the administrative hierarchy and N is the number of nodes in the system.

An extension of Chord considers multiple virtual rings [8] focusing on efficiently supporting multiple subgroups using an existing Chord ring. Their ideas might be applicable to achieve administrative isolation for a two-level administrative hierarchy but it might be inefficient for multi-level hierarchy.

Coral [3, 4] is a peer-to-peer content distribution network which uses a decentralized hierarchical clustering algorithm by which nodes can find each other and form clusters of varying network diameters. Coral then builds different DHTs at each level in the hierarchical clustering tree. In contrast to ADHT's focus on supporting administrative isolation, Coral focuses on finding a nearby node in terms of network locality. They consider shallow hierarchies (three-level hierarchy) and mainly focus on automatically building such hierarchies based on observed round-trip times between nodes.

Similar mechanisms to Zippering are proposed for handling organizational disconnects in SkipNet [7] and for merging two separately formed DHTs in Willow [14].

6 Summary and Open Issues

Administrative isolation is an important requirement to satisfy in an information management system for security, availability, efficiency. We present two properties — Path Locality and Path Convergence — that a DHT should satisfy so that the aggregation trees extracted from such DHT satisfy the administrative isolation property. Current DHT algorithms can achieve path locality but do not guarantee path convergence. In this chapter, we have described a novel Autonomous DHT (ADHT) that satisfies both path convergence and path locality.

ADHT builds upon and augments an existing DHT, Pastry, to achieve administrative isolation through the following four key ideas: (i) Each node in ADHT has multiple leafsets corresponding to levels of administrative hierarchy in which that node participates, (ii) ADHT employs a novel key space assignment and a novel routing algorithm that ensure search paths from nodes in a domain for key converge at a node within that domain, (iii) A node joining ADHT locates a nearest node in terms of domain nearness and uses that node as the bootstrap node to join ADHT, and (iv) Each node in ADHT periodically tests for partitions in each domain it participates and uses a *Zippering* mechanism to mend partitions.

We evaluate the performance of ADHT through simulation experiments. We observe that whereas ADHT satisfies path convergence property, Pastry can incur up to 16% violations in probe pairs. In terms of path length to the root, note that ADHT path lengths are $O(\log N + l)$ compared to Pastry's $O(\log N)$ where N is the number of nodes in the network and l is the number of levels in the administrative hierarchy. In simulations we observe that the path lengths in ADHT are modestly higher than in Pastry and they are higher when the depth of administrative hierarchy is deeper and this is precisely the case where Pastry incurs more path convergence property violations.

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] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing Content Publication with Coral. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[4] M. J. Freedman and D. Mazires. Sloppy Hashing and Self-Organizing Clusters. In *2nd Intl. Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[5] 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.

[6] 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.

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

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

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

[10] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *Proceedings of the USITS conference*, 2003.

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

[12] P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Computer Communications Review*, 18(4):123–133, Aug. 1988.

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

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

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

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