A Scalable Distributed Information Management System

Praveen Yalagandula and Mike Dahlin
Department of Computer Sciences
The University of Texas at Austin

Abstract

We present a Scalable Distributed Information Management System (SDIMS) that *aggregates* information about large-scale networked systems and that can serve as a basic building block for a broad range of large-scale distributed applications providing detailed views of nearby information and summary views of global information. To serve as a basic building block, a SDIMS should have four properties: scalability to many nodes and attributes, flexibility to accommodate a broad range of applications, support administrative autonomy and isolation, and robustness to node failures and disconnections. We design, implement and evaluate a SDIMS that (1) uses techniques from Distributed Hash Table (DHT) literature to create scalable aggregation trees, (2) provides flexibility through a simple API that lets applications control propagation of reads and writes, (3) provides autonomy and isolation through simple augmentations of current DHT algorithms, and (4) is robust to node and network reconfigurations through lazy reaggregation, on-demand reaggregation, and tunable spatial replication. Through extensive simulations and micro-benchmark experiments, we observe that our system is an order of magnitude more scalable than existing approaches, achieves autonomy and isolation properties at the cost of modestly increased read latency in comparison to flat DHTs, and gracefully handles failures.

1 Introduction

The goal of this paper is to construct a Scalable Distributed Information Management System (SDIMS) that *aggregates* information about large-scale networked systems and that can serve as a basic building block for a broad range of large-scale distributed applications. Monitoring, querying, and reacting to changes in the state of a distributed system are core components of applications such as system management [11, 26, 34, 35], service placement [10, 36], data sharing and caching [19, 23, 27, 31, 37, 33], sensor monitoring and control [16], multicast tree formation [4, 5, 25, 29, 32], and naming and request routing [6, 7]. We therefore speculate that a SDIMS in a networked system would provide a "distributed operating systems backbone" and facilitate the development and deployment of new distributed services.

For a large scale information system, *hierarchical aggregation* is a fundamental abstraction for scalability. Rather than expose all information to all nodes, hierarchical aggregation allows a node to access detailed views of nearby information and summary views of global information. In a SDIMS based on hierarchical aggregation, different nodes can therefore receive different answers to the query "find a [nearby] node with at least 1 GB of free memory" or "find a [nearby] copy of file foo." A hierarchical system that aggregates information through reduction trees [16, 25] allows nodes to access information they care about while maintaining system scalability.

To be used as a basic building block, a SDIMS should have four properties. First, the system should accommodate large numbers of participating nodes, and it should allow applications to install and monitor large numbers of data attributes. Enterprise and global scale systems today might have tens of thousands to millions of nodes and these numbers will increase as desktop machines give way to larger numbers of smaller devices. Similarly, we hope to support many applications and each application may track several attributes (e.g., the load and free memory of a system's machines) or millions of attributes (e.g., which files are stored on which machines).

Second, the system should have *flexibility* to accommodate a broad range of applications and attributes. For example, *read-dominated* attributes like *numCPUs* rarely change in value, while *write-dominated* attributes like *numProcesses* change quite often. An approach tuned for read-dominated attributes will suffer from high bandwidth consumption when applied for write-dominated attributes. Conversely, an approach tuned for write-dominated attributes may suffer from unnecessary query latency or imprecision for read-dominated attributes. Therefore, a SDIMS should provide a flexible mechanism that can efficiently handle different types of attributes, and leave the policy decision of tuning read and write propagation to the application installing an attribute.

Third, an SDIMS should provide *autonomy and isolation*. In a large computing platform, it is natural to arrange nodes in an organizational or an administrative hierarchy (e.g., Figure 1). A SDIMS should support administrative autonomy so that, for example, a system administrator can control what information flows out of her ma-

chines and what queries may be installed on them. And, a SDIMS should provide isolation in which queries about a domain's information can be satisfied within the domain so that the system can operate during disconnections and so that an external observer cannot monitor or affect intra-domain queries.

Fourth, the system must be *robust* to node failures and disconnections. A SDIMS should adapt to reconfigurations in a timely fashion and should also provide mechanisms so that applications can exploit the tradeoff between the cost of adaptation versus the consistency level in the aggregated results when reconfigurations occur.

We draw inspiration from two previous works: *Astrolabe* and *Distributed Hash Tables (DHTs)*.

Astrolabe [25] is a robust information management system. Astrolabe provides the abstraction of a single logical aggregation tree that mirrors a system's administrative hierarchy for autonomy and isolation. It provides a general interface for installing new aggregation functions and provides eventual consistency on its data. Astrolabe is highly robust due to its use of an unstructured gossip protocol for disseminating information and its strategy of replicating all aggregated attribute values for a subtree to all nodes in the subtree. This combination allows any communication pattern to yield eventual consistency and allows any node to answer any query using local information. This high degree of replication, however, may limit the system's ability to accommodate large numbers of attributes. Also, although the approach works well for read-dominated attributes, an update at one node can eventually affect the state at all nodes, which may limit the system's flexibility to support write-dominated attributes.

Recent research in peer-to-peer structured networks resulted in Distributed Hash Tables (DHTs) [4, 5, 7, 19, 22, 23, 27, 29, 31, 32, 37, 33]—a data structure that scales with the number of nodes and that distributes the read-write load for different queries among the participating nodes. It is interesting to note that although these systems export a global hash table abstraction, many of them internally make use of what can be viewed as a scalable system of aggregation trees to, for example, route a request for a given key to the right DHT node. Indeed, rather than export a general DHT interface, Plaxton et al.'s [22] original application makes use of hierarchical aggregation to allow nodes to locate nearby copies of objects. It seems appealing to develop a SDIMS abstraction that exposes this internal functionality in a general way so that scalable trees for aggregation can be considered a basic system building block alongside the distributed hash tables.

At first glance, it might appear obvious that simply combining DHTs with Astrolabe's aggregation abstraction will result in a SDIMS. However, meeting the requirements discussed above requires a design to address four questions: (i) How to scalably map different attributes to different aggregation trees within a DHT mesh? (ii) How to provide flexibility in the aggregation to accommodate different application requirements?, (iii) How to adapt a global, flat DHT mesh to satisfy the required autonomy and isolation properties? and (iv) How to provide good robustness without unstructured gossip and total replication?

Our key ideas for building a SDIMS using ideas from DHTs and Astrolabe are as follows.

1. We expose a DHT system's internal trees as an aggregation abstraction by aggregating an attribute along the tree corresponding to the attribute type and name. This approach gives SDIMS *scalability* with respect to both nodes and attributes.

2. We provide a flexible API that lets applications control the propagation of reads and writes and thus trade off update cost, read latency, replication, and staleness.

3. We augment an existing DHT algorithm to ensure *path convergence* and *path locality* properties in order to achieve *autonomy* and *isolation*.

4. We provide *robustness* to node and network reconfigurations by (a) providing temporal replication through lazy reaggregation that guarantees eventual consistency and (b) ensuring that our flexible API allows demanding applications gain additional robustness by either using tunable spatial replication of data aggregates and/or performing fast on-demand reaggregation to augment the underlying lazy reaggregation.

We have built a prototype of SDIMS. Through simulations and micro-benchmark experiments on a number of department machines and Planet-Lab [21] nodes, we observe that the prototype achieves scalability with respect to the number of nodes and the number of attributes through use of its flexible API, inflicts an order of magnitude less maximum node stress when compared to unstructured gossiping schemes, achieves autonomy and isolation properties at the cost of modestly increased read latency compared to flat DHTs, and gracefully handles node failures.

This initial study discusses key aspects of an ongoing large system building effort, but it does not address all issues with constructing a SDIMS. For example, we believe that our strategies for providing robustness will mesh well with techniques such as *supernodes* [17] for further improving robustness as well as other ongoing efforts to improve DHTs [24]. Also, although splitting aggregation among many trees improves scalability for simple queries, this approach may make complex, and multi-attribute queries more expensive compared to a single tree. Additional work is needed to understand the significance of this limitation for real workloads and, if necessary, to adapt query planning techniques from DHT abstractions [12, 14] to scalable aggregation tree abstractions.

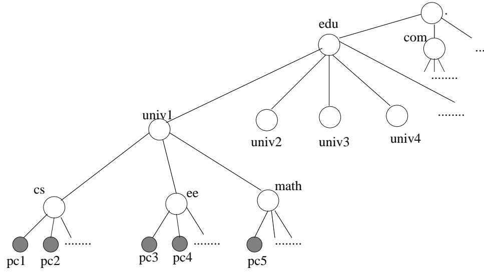

Fig. 1: Administrative hierarchy

In Section 2, we explain the aggregation abstraction and the flexible API exported by our system. In Section 3, we detail the DHT construction that provides autonomy and isolation properties and explain the data structures and the behavior of the node. Section 4 addresses the issue of adaptation to the topological reconfigurations. In Section 5, we present the evaluation of our system through large-scale simulations and microbenchmarks on real networks. Section 6 details the related work and Section 7 summarizes our contribution and points out the future research directions.

2 Aggregation Abstraction and Flexible API

Our system provides a standard aggregation abstraction via a novel flexible API that allows applications to provide hints about whether the system should (a) compute aggregation functions and propagate them to readers on demand in response to reads, (b) recompute aggregation functions whenever updates to their input data occur but propagate them to readers on demand, or (c) recompute aggregation functions and aggressively propagate the results through the system when updates occur.

2.1 Aggregation Abstraction

Aggregation is a natural abstraction for a large-scale distributed information system because aggregation provides scalability by allowing a node to view detailed information about the state near it and progressively coarser-grained summaries about progressively larger subsets of a system's data [25].

Our aggregation abstraction works on the assumption that nodes is the system are arranged in a tree that complies with the administrative boundaries. As Figure 1 illustrates, each physical node in the system is a leaf of the tree, and each subtree represents a logical grouping of nodes. Note that logical groupings can correspond to administrative domains (e.g., "cs.univ1.edu" or "edu") or groupings of nodes within a domain (e.g., 10 workstations on a LAN in the CS department). We describe how to form such trees in Section 3.

Each physical node has *local data* stored as a set of $(attributeType, attributeName, value)$ tuples such as *(configuration, numCPUs, 16)*, *(mcast membership, session foo, yes)*, or *(file stored, foo, myIPaddress)*.

The system associates an *aggregation function* f_{type} with each attribute type, and for each level-i subtree T_i in the system, the system defines an *aggregate value* $V_{i,type,name}$ for each (attributeType, attributeName) pair as follows. For a (physical) leaf node T_0 at level 0, $V_{0,type,name}$ is the locally stored value for the attribute type and name or NULL if no matching tuple exists. Then the aggregate value for a level-i subtree T_i is the aggregation function for the type computed across the aggregate values of each of T_i's k children: $V_{i,type,name} = f_{type}(V^0_{i-1,type,name}, V^1_{i-1,type,name}, \ldots, V^{k-1}_{i-1,type,name})$.

Having aggregation trees that conform with the administrative hierarchy helps SDIMS provide important autonomy, security, and isolation properties [25]. Security and autonomy are important in that a system administrator must be able to control what information flows out of her machines and what queries may be installed on them. The isolation property ensures that a malicious node in one domain cannot observe or affect system behavior in another domain for computations relating only to the second domain.

Although our system allows arbitrary aggregation functions, it is desirable that aggregation functions satisfy the *hierarchical computation* property [16]: $f(v_1, ..., v_n) = f(f(v_1, ..., v_{s_1}), f(v_{s_1+1}, ..., v_{s_2}), ..., f(v_{s_k+1}, ..., v_n))$, where v_i is the value at node i. For example, the average operation, defined as $avg(v_1, ..., v_n) = 1/n. \sum_{i=0}^{n} v_i$, does not satisfy the property. Instead, if values for such an attribute are stored as tuples $(sum, count)$ and the function is defined as $avg(v_1, ..., v_n) = (\sum_{i=0}^{n} v_i.sum, \sum_{i=0}^{n} v_i.count)$, it satisfies the hierarchical computation property. Note that the applications then have to compute the average from the aggregate sum and count values.

Finally, note that for a large-scale system, it is difficult or impossible to insist that the aggregation value returned by a probe corresponds to the function computed over the current values at the leaves at the instant of the probe. Systems, therefore, typically provide only weak consistency guarantees, such as eventual consistency, to improve responsiveness and robustness [25].

2.2 Flexible computation

The definition of aggregate values allows considerable flexibility in how, when, and where they are computed. In particular, instead of gathering all leaf values at one location and recursively evaluating the function to obtain the global aggregate, this definition allows a system to perform aggregation on a tree in a distributed fashion for scalability, and it allows this computation to occur lazily on reads, eagerly on updates, or using hybrid strategies.

As Figure 2 illustrates, under an *Update-Local* option, an update only affects local state. Then, a probe that reads

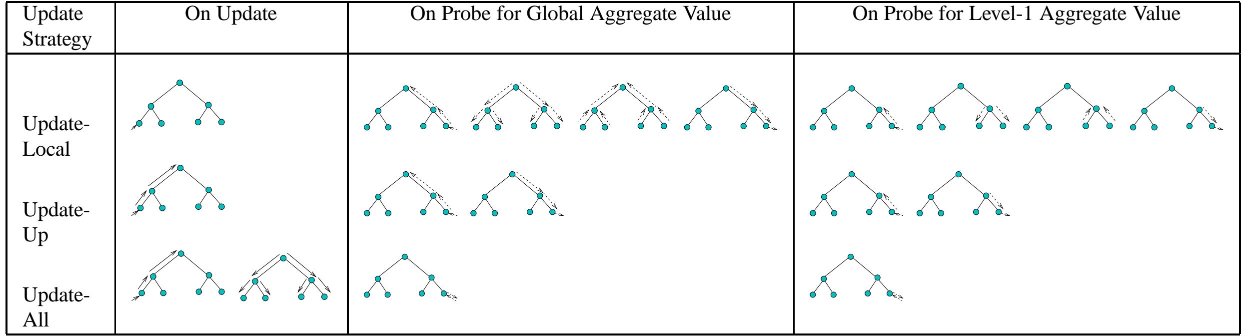

Fig. 2: Flexible API

a level-i aggregate value can be sent up the tree to the issuing node's level-i ancestor and then down the tree to the leaves. The system can then compute the desired aggregate value at each layer up the tree until the level-i ancestor holds the desired value. Finally, the level-i ancestor can send the result down the tree to the issuing node. Alternately, under an *Update-Up* strategy, the root of each subtree maintains the subtree's current aggregate value, and when an update occurs, the leaf node updates its local state and passes the update to its parent, and then each successive enclosing subtree updates its aggregate value and passes the new value to its parent. This strategy satisfies a leaf's probe for a level-i aggregate value by sending the probe up to the level-i ancestor of the leaf and then sending the aggregate value down to the leaf. In an *Update-All* strategy [25] each level-i node not only maintains the aggregate values for the level-i subtree but also receives and locally stores copies of all of its ancestors' level-j ($j > i$) aggregation values. So, when an update occurs, changes are aggregated up the tree, and each new aggregate value is broadcast to all of a node's descendants. Under this strategy, a leaf can satisfy a probe for a level-i aggregate using purely local data. Finally, notice that other strategies also exist. For example, an *Update-UpRoot-Down2* strategy (not shown) would aggregate updates up to the root of a subtree and send a subtree's aggregate values to the children and grandchildren of the subtree's root.

The nature of the attributes that applications install vary extensively. For example, a *read-dominated* attribute like *numCPUs* rarely change in value, while a *write-dominated* attribute like *numProcesses* changes quite often. An aggregation strategy like Update-All could work well for *read-dominated* attributes but suffer high bandwidth consumption when applied for *write-dominated* attributes. Conversely, an approach like Update-Local could work well for for *write-dominated* attributes but suffer from unnecessary query latency or imprecision for *read-dominated* attributes.

2.3 Aggregation API

SDIMS provides a flexible API that allows applications to control the strategy used for computing each attribute type's aggregate values for three purposes:

parameter	description	optional
attrType	Attribute Type	
aggrfunc	Aggregation Function	
attrName	Attribute Name	X
domain	Domain restriction (default: none)	X
up	How far upwards each update is sent (default: all)	X
down	How far downwards each aggregate is sent (default: none)	X
expTime	Expiry Time	

Table 1: Arguments for the install operation

- To minimize the cumulative cost for both updates and probes. For example, an attribute type with a large reads-to-write ratio might be installed as an Update-All type, and one with a reads-to-write ratio of about one might be installed as Update-Up type.
- To tune probe latencies v. update overheads. Probes incur different latencies based on how aggressively updates are distributed. For example, an application requiring a very low latency on probes could install its attribute type as an Update-All type.
- To tune robustness against failures v. update overheads. By propagating aggregated values downwards to more nodes, the applications can mask network and node reconfigurations by providing multiple redundant locations where a given aggregate value is stored and by reducing a probe's dependencies on network connectivity up the aggregation tree. This issue is discussed in depth in Section 4.

The API to applications consists of three functions: *Install()* installs an aggregation function that defines an operation on an attribute type and specifies the update strategy that the function will use, *Update()* inserts or modifies a node's local $(attributeType, attributeName, value)$ tuple (which may trigger aggregation computation and propagation depending on the function's update strategy), and *Probe()* obtains an aggregate value for a specified subtree.

2.3.1 Install

The *Install* operation installs an aggregation function in the system. The arguments for this operation are listed in Table 1. The *attrType* argument denotes the type of

attributes on which this aggregation function is invoked. The optional *attrName* argument denotes that the aggregation function be applied only to the particular attribute with name *attrName*. Installed functions are soft state that must be periodically renewed or they will be garbage collected at *expTime*. Finally note that each domain specifies a security policy that restricts the types of functions that can be installed by different entities based on the attributes they access and their scope in time and space [25].

The optional *domain* argument, if present, indicates that the aggregation function should be installed on all nodes belonging to the specified domain; if this argument is absent, then the function is to be installed on all nodes in the system.

An aggregation function installed with a specific *attrName* takes precedence over the aggregation function with matching *attrType* and with no *attrName* specified for updates whose name and type both match.

The arguments *up* and *down* specify the strategy for propagating updates. When an update occurs at a leaf, the system updates any changed aggregate values for the level-0 (leaf) through level-*up* subtrees enclosing the updated leaf. After the root of a level-i subtree computes a new aggregate value, the system propagates and stores this value to the subtree's level-i to level-$i-down$ roots. At the API level, these arguments can be regarded as hints, since they suggest a computation strategy but do not affect the semantics of an aggregation function. In principle, it would be possible, for example, for a system to dynamically adjust its up/down strategies for a function based on measured read/write frequency. However, our implementation always simply follows these directives.

Finally, note that our aggregation function installation differs from Astrolabe's by specifying both an attribute type and attribute name and associating an aggregation function with a type rather than just specifying an attribute name and associating a function with a name. Installing a single function that can operate on many different named attributes matching a specific type improves scalability for "sparse attribute types" with a large, sparsely-filled namespace. For example, to construct a file location service, our interface allows us to install a single function that compute an aggregate value for any named file (e.g., the aggregate value for the (function, name) pair for a subtree would be the ID of one node in the subtree that stores the named file). Conversely, Astrolabe copes with sparse attributes by having aggregation functions compute sets or lists and suggests that scalability can be improved by representing such sets with Bloom filters [2]. Exposing sparse names within a type provides at least two advantages. First, when the value associated with a name is updated, only the state associated with that name need be updated and (potentially) propagated to other nodes. Second, for the multiple-tree system we describe in Section 3, splitting values associated with different names into different aggregation values allows our system to map different names to different trees and thereby spread the function's logical root node's load and state across multiple physical nodes.

parameter	description	optional
attrType	Attribute Type	
attrName	Attribute Name	
val	Value	
ts	Timestamp	X

Table 2: Arguments for the update operation

parameter	description	optional
attrType	Attribute Type	
attrName	Attribute Name	
origNode	Originating Node	
serNum	Serial Number	
mode	Continuous or One-shot (default: one-shot)	X
level	Level at which aggregate is sought (default: highest level)	X
up	How far up to go and re-fetch the value (default: install up)	X
down	How far down to go and re-aggregate (default: install down)	X
expTime	Expiry Time	

Table 3: Arguments for the probe operation

2.3.2 Update

The update operation creates a new (attributeType, attributeName, value) tuple or updates the value of an old tuple at a leaf node. The arguments for the update operation are shown in Table 2.

As outlined above and described in detail in Section 3.2, after a leaf applies an update locally, the update may trigger re-computation of aggregate values up the tree and may also trigger propagation of changed aggregate values down the tree.

2.3.3 Probe

Whereas update propagates the aggregates in the system according to the specifications of the install operation, a probe operation collects the aggregated values at the application-queried levels. The complete argument set for the probe operation is shown in Table 3. Along with the *attrName* and the *attrType* arguments to denote the aggregate value of interest to this probe, a *level* argument specifies the level at which the answers are required.

When *up* and *down* arguments are specified in a probe, a forced re-aggregation is done for the corresponding levels even if the aggregated value is available. The *up* and *down* arguments are interpreted as described in Section 2.3.1. In Section 4, we explain how applications can exploit these arguments to perform on-demand fast aggregation during reconfigurations.

3 System Design

This section describes the internal design of the SDIMS system. As Figure 3 indicates, the design comprises two

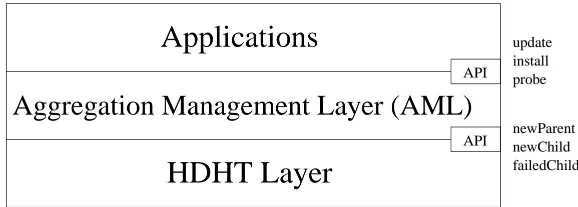

Fig. 3: Two layer SDIMS design and interfaces.

layers: the Aggregation Management Layer (AML) stores attribute tuples and calculates and stores aggregate values and the Hierarchical DHT (HDHT) layer manages the internal topology of the system. In Section 3.1, we discuss how we modify DHTs to support the autonomy and isolation properties required by SDIMS and how we map attributes types and names to this collection of trees. We describe how the HDHT layer constructs a scalable set of trees by exposing the internal aggregation facilities already present in many existing DHTs. In Section 3.2, we discuss in detail the internal operation of each node in our system. We defer to Section 4 the discussion on how SDIMS handles network and node reconfigurations.

3.1 Hierarchical DHT for Aggregation

Existing DHTs (Distributed Hash Table) can be viewed as a mesh formed of several trees. DHT systems assign an identity to each node (a *nodeId*) that is drawn randomly from a large space. Keys are also drawn from the same space and each key is assigned to a live node in the system. Each node maintains a routing table with nodeIds and IP addresses of some other nodes. The DHT protocols use these routing tables to route the packets for a key k towards the node responsible for that key. Suppose the node responsible for a key k is $root_k$. The paths from all nodes for a key k form a tree rooted at the node $root_k$ — say $DHTtree_k$.

It is straightforward to make use of this internal structure for aggregation. [22] By aggregating an attribute along the tree $DHTtree_k$ for k =*hash(attribute type, attribute name)*, different attributes will be aggregated along different trees. In comparison to a scheme where all attributes are aggregated along a single tree, the DHT based aggregation along multiple trees incurs lower maximum node stress: whereas in a single aggregation tree approach, the root and the intermediate nodes pass around more messages than the leaf nodes, in a DHT-based multi-tree, each node acts as intermediate aggregation point for some attributes and as leaf node for other attributes. Hence, this approach distributes the onus of aggregation across all nodes.

As noted in Section 2, aggregation trees in SDIMS should follow the system's administrative hierarchy. To conform to these requirements, a HDHT should satisfy two additional properties:

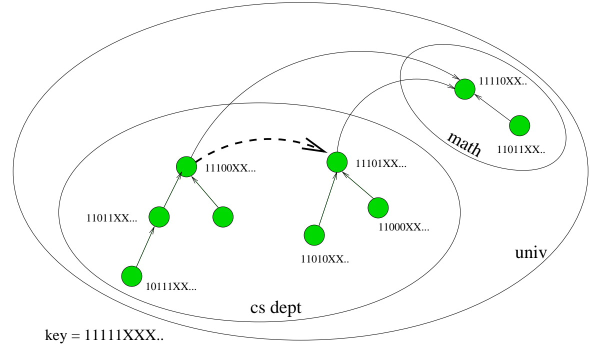

Fig. 4: Example shows why original pastry (solid lines) does not satisfy the isolation properties. Simple augmentations to the links maintained (dashed lines) and routing protocol (take dashed line) make it abiding.

1. Path Locality : Search paths should always be contained in the smallest possible domain.
2. Path Convergence : Search paths for a key from two different nodes in a domain should converge at a node in the same domain.

Existing DHTs do not guarantee path convergence. In the rest of this section we explain how an existing DHT, Pastry [27], does not satisfy path convergence, and then we describe a simple modification to Pastry that supports convergence by introducing a few additional routing links and a two level locality model that incorporates both administrative membership of nodes and network distances between nodes. We choose Pastry for convenience—the availability of a public domain implementation. We believe that similar simple modifications could be applied to many existing DHT implementations to support path convergence.

3.1.1 Pastry

In Pastry [27], each node maintains a leaf set and a routing table. The leaf set contains the L immediate clockwise and counter-clockwise neighboring nodes in a circular nodeId space (*ring*). The routing table supports *prefix* routing: each node's routing table contains one row per hexadecimal digit in the nodeId space and the ith row contains a list of nodes whose nodeIds differ from the current node's nodeId in the ith digit with one entry for each possible digit value. Notice that for a given row and entry (viz. digit and value) a node n can choose the entry from many different alternative destination nodes, especially for small i where a destination node needs to match n's ID in only a few digits to be a candidate for inclusion in n's routing table. A system can choose any policy for selecting among the alternative nodes. A common policy is to choose a nearby node according to a *proximity metric* [22] to minimize the network distance for routing a key. Under this policy, the nodes in a routing table sharing a short prefix will tend to be nearby since there are many such nodes spread roughly evenly throughout the

system due to random nodeId assignment. Pastry is self-organizing—nodes come and go at will. To maintain Pastry's locality properties, a new node must join with one that is nearby according to the proximity metric. Pastry provides a seed discovery protocol that finds such a node given an arbitrary starting point.

Given a routing topology, to route to an arbitrary destination key, a node in Pastry forwards a packet to the node with a nodeId prefix matching the key in at least one more digit than the current node. If such a node is not known, the node forwards the packet to a node with an identical prefix but that is numerically closer to the destination key in the nodeId space. This process continues until the destination node appears in the leaf set, after which it is delivered directly. The expected number of routing steps is $\log N$, where N is the number of nodes.

Unfortunately, as the solid lines in Figure 4 illustrate, when Pastry uses network proximity as the locality metric, it does not satisfy the desired SDIMS properties because (i) if two nodes with nodeIds match a key in same number of bits, both of them can route to a third node outside the domain when routing for that key and (ii) if the network proximity does not match the domain proximity then there is little chance that a tree will satisfy the properties. The second problem can be addressed by simply changing the proximity metric to declare that any two nodes that match in i levels of a hierarchical domain are always considered closer than two nodes that match in fewer than i levels. However, this solution does not eliminate the first problem.

3.1.2 Autonomous Pastry

To provide autonomy properties to an aggregating HDHT, the system's route table construction algorithm must provide a single exit point in each domain for any key and its routing protocol should route keys along intra-domain paths before routing them along inter-domain paths. Simple modifications to Pastry's route table construction and key-routing protocols achieve these goals. In Figure 4, our algorithm routes towards the node with nodeId $11101XXX\ldots$ for key $11111XXX\ldots$ (shown by dashed lines).

In HDHT, each node maintains a separate leaf set for each domain it is part of, unlike Pastry that maintains a single leaf set for all the domains. Maintaining a different leafset for each level increases the number of neighbors that each node tracks to $(2^b) * \lg_b n + c.l$ from $(2^b) * \lg_b n + c$ 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.

Each node in HDHT has a routing table. The algorithm for populating the routing table is similar to Pastry with the following difference: it uses hierarchical domain proximity as the primary proximity metric (two nodes that match in i levels of a hierarchical domain are more proximate than two nodes that match in fewer than i levels of a domain) 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).

Similar to Pastry's join algorithm [27], a node wishing to join HDHT routes a join request with target key set to its *nodeId*. In Pastry, the nodes in the intermediate path respond to the node's request with the pertinent routing table information and the current root node sends its leafset. In our algorithm, to enable the joining node fill its leafsets at all levels, the following two modifications are done to Pastry's join protocol: (1) a joining node chooses a bootstrap node that is closest to it with respect to the hierarchical domain proximity metric and (2) each intermediate node sends its leafsets for all domain levels in which it is the root node. These simple modifications ensure that the joining node's leafsets and route table are properly filled.

The routing algorithm we use in routing for a key at node with *nodeId* is shown in the Algorithm 3.1.2. By routing at the lowest possible domain till the root of that domain is reached, we ensure that the routing paths conform to the Path Convergence property.

Algorithm 1 HDHTroute(key)

```
1:  flipNeigh ← checkRoutingTable(key) ;
2:  l ← numDomainLevels - 1 ;
3:  while (l >= 0) do
4:    if (commLevels(flipNeigh, nodeId) == l) then
5:      send the key to flipNeigh ; return ;
6:    else
7:      leafNeigh ← an entry in leafset[l] closer to key
        than nodeId ;
8:      if (leafNeigh ! = null) then
9:        send the key to leafNeigh ; return ;
10:     end if
11:   end if
12:   l ← l − 1;
13: end while
14: this node is the root for this key
```

3.2 Aggregation Data Structures and Operation

Given the HDHT topology described above, each node implements an Aggregation Management Layer (AML) to support the flexible API described in Section 2.3. This subsection describes the internal state and operation of the AML layer of a node in the system.

We refer to a tuple store of (attribute type, attribute name, value) tuples as a Management Information Base or MIB, following the terminology from Astrolabe [25] (originally used in the context of SNMP [30]). We refer

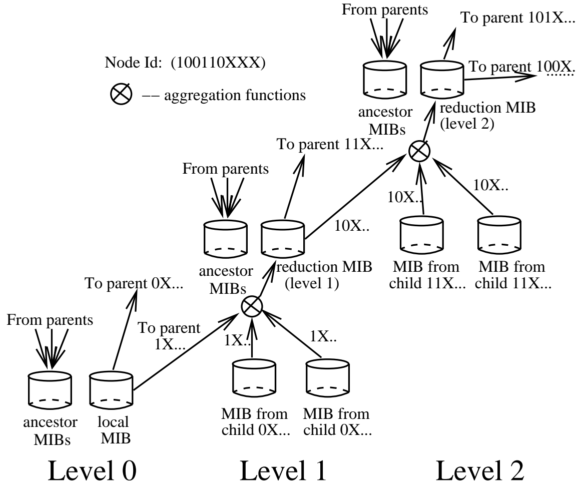

Fig. 5: Example illustrating the datastructures and the organization of them at a node.

to the pair (attribute type, attribute name) as an *attribute key*.

Each physical node in the system acts as several logical nodes in the HDHT: a node acts as root for all attribute keys, as a level-1 subtree root for attribute keys whose hash matches the node's ID in b bits (where b is the number of bits corrected in each step of the HDHT's key routing algorithm), as a level-i subtree root for attribute keys whose has matches the node's ID in b^i bits, and as the system's global root for the attribute key for attribute keys whose hash matches the node in more bits than any other node.

As Figure 5 illustrates, to support hierarchical aggregation, each logical node corresponding to a level-i subtree root for some attribute keys maintains several MIBs that store (1) *child MIBs* containing raw aggregate values gathered from children, (2) a *reduction MIB* containing locally aggregated values across this raw information, and (3) *ancestor MIBs* containing aggregate values scattered *down* from ancestors. This basic strategy of maintaining child, reduction, and ancestor MIBs is based on Astrolabe [25], but our structured propagation strategy channels information that flows up according to its attribute key and our flexible propagation strategy only sends child updates *up* and ancestor aggregate results *down* as far as specified by the attribute key's aggregation function. Note that in the discussion below, for ease of explanation, we assume that the routing protocol is correcting single bit at a time ($b = 1$) in contrast to default Pastry scheme where the routing protocol tries to correct up to four bits in each stem ($b = 4$). Our system, built upon Pastry, does handle multi-bit correcting and is a simple extension to the scheme described here.

For a given virtual node n_i at level i, each *child MIB* contains the subset of a level $i-1$ child's reduction MIB that contains tuples that match n_i's node ID in i bits and whose *up* aggregation function attribute is at least i. These local copies make it easy for a node to recompute a level-i aggregate value when one child's inputs changes. Nodes maintain their child MIBs in stable storage and use a simplified version of the Bayou protocol (*sans* conflict detection and resolution) for synchronization after disconnections [20].

Virtual node n_i at level i maintains a *reduction MIB* of tuples with a tuple for each key present in any child MIB containing the attribute type, attribute name, and output of the attribute type's aggregate functions applied to the children's tuples.

If a reduction tree has s subtree levels including the root, virtual node n_i at level i maintains $s-i-1$ *ancestor MIBs*. Ancestor MIB j ($i < j \leq s$) contains level-j aggregate values computed across enclosing subtree j and propagated *down* to level i.

Note that level-0 differs slightly from other levels. Each level-0 leaf node maintains a *local MIB* rather than maintaining child MIBs and a reduction MIB. This local MIB stores information about the local node's state inserted by local applications via *update()* calls.

Along with these MIBs, a node maintains two other tables—an aggregation function table and an outstanding probes table. An aggregation function table contains the aggregation function and installation arguments (see Table 1) associated with an attribute type or an attribute type and name. Note that a function that matches an attribute key in type and name has precedence over a function that matches an attribute key in type only. Each aggregate function is installed on all nodes in a domain's subtree, so the aggregate function table can be thought of as a special case of the ancestor MIB with domain functions always installed *up* to a root within a specified domain and *down* to all nodes within the domain. The outstanding probes table maintains temporary information regarding information gathered and outstanding requests for in-progress probes.

Given these data structures, it is simple to support the three API functions described in Section 2.3.

The *Install* operation (see Table 1) installs on a domain an aggregation function that acts on a specified attribute type. Execution of an install function *aggrFunc* on attribute type *attrType* and (optionally) attribute name *attrName* proceeds in two phases: first the install request is passed up the HDHT tree with the key *(attrType, attrName)* until reaching the root for that key within the specified domain. Then, the request is flooded down the tree and installed on all intermediate and leaf nodes.

Before installing an aggregation function, a node checks it against its per-domain access control list [25], and after installing an aggregation function, a node sets a timer to uninstall the function when it expires.

The *Update* operation (see Table 2) creates a new (at-

tributeType, attributeName, value) tuple or updates the value of an old tuple at a leaf. Then, subject to the update propagation policy specified in the *up* and *down* parameters of the aggregation function associated with the update's attribute key, the update triggers a two-phase propagation protocol as Figure 2 illustrates. An update operation invoked at a leaf always updates the local MIB. Then, if the update changes the local value and if the aggregate function for the attribute key was installed with $up > 0$ and if the leaf's parent for the attribute key is within the domain to which the installed aggregation function is restricted, the leaf passes the new value up to the appropriate parent based on the attribute key. Level i behaves similarly when it receives a changed attribute from level $i-1$ below: it first recomputes the level-i aggregate value for the specified key, stores that value in the level-i reduction table and then, subject to the function's *up* and *domain* parameters, passes the updated value to the appropriate level-$i+1$ parent based on the attribute key. After a level-i ($i \geq 1$) virtual node has updated its reduction MIB, if the reduction function *down* argument indicates that aggregate values should be sent down to level j ($j < i$), the node sends the updated value down to all of its level $i-1$ children marked as the level-i aggregate for the specified attribute key. Upon receipt of such a level-i aggregate value message from a parent from level j ($j \leq i$), a node stores the value in its level-i ancestor MIB and, subject to the relevant installed function's *down* parameter, forwards this level-i aggregate value to its children.

A *Probe* operation collects and returns the aggregate value for a specified attribute key for a specified level of the tree. As Figure 2 illustrates, the system satisfies a probe for a level-i aggregate value using a four-phase protocol that may be short-circuited when updates have previously propagated results or partial results up or down the tree. In phase 1, the *route probe phase*, the system routes the probe up the attribute key's tree to either the root of the level-i subtree or to a node that stores the requested value in its level-i ancestor MIB. In the former case, the system proceeds to phase 2 and in the latter it skips to phase 4. In phase 2, the *probe scatter phase*, each node that receives a probe request sends it to all of its children unless the node is a leaf or the node's reduction MIB already has a value that matches the probe's attribute key, in which case the node initiates phase 3 on behalf of its subtree by forwarding its local MIB or reduction MIB value up to the appropriate parent for the attribute key. In phase 3, the *probe aggregation phase*, when a node receives input values for the specified key from each of its children, it executes the aggregate function across these values and either (a) forwards the result to its parent (if its level is less than i) or (b) initiates phase 4 by forwarding the result to the child that requested it (if it is at level i). Finally, phase 4, the *aggregate routing phase* the aggregate value is routed down to the node that requested it. Note that in the extreme case of a function installed with $up = down = 0$, a level-i probe can touch all nodes in a level-i subtree while in the opposite extreme case of a function installed with $up = down = ALL$, probe is a completely local operation at a leaf.

For probes that include phases 2 (probe scatter) and 3 (probe aggregation), an issue is determining when a node should stop waiting for its children to respond and send up its current aggregate value. A node at level i stops waiting for its children when one of three conditions occurs: (1) all children have responded, (2) the HDHT layer signals one or more reconfiguration events that marks all children that have not yet responded as unreachable, or (3) a watchdog timer for the request fires. The last case accounts for nodes that participate in the HDHT protocol but that fail at the AML level.

4 Robustness

In large scale systems, reconfigurations are a norm. Our two main principles for robustness are to guarantee (i) read availability – probes complete in a finite time, and (ii) eventual consistency – updates by a live node will be reflected in the answers of the probes in a finite time. During reconfigurations, a probe might return a stale value due to two reasons. First, reconfigurations lead to incorrectness in the previous aggregate values. Second, the nodes needed for aggregation to answer the probe become unreachable. Our system also provides two hooks for end-to-end applications to be robust in the presence of reconfigurations: (1) On-demand re-aggregation, and (2) application controlled replication.

Our system handles reconfigurations at two levels – adaptation at the HDHT layer to ensure connectivity and adaptation at the AML layer to ensure access to the data in SDIMS.

4.1 HDHT Adaptation

Our HDHT layer adaptation algorithm is same as Pastry's adaptation algorithm [27] — the leaf sets are repaired as soon as a reconfiguration is detected and the routing table is repaired lazily. Due to redundancy in the leaf sets and the routing table, the updates can be routed towards their root nodes successfully even during failures. Also note that the autonomy and isolation properties satisfied by our HDHT algorithm ensure that the reconfigurations in a level i domain do not affect the probes for level i in the sibling domains.

4.2 AML Adaptation

Broadly, we use two types of strategies for AML adaptations in the face of reconfigurations: (1) Replication in time, and (2) Replication in space. We first examine replication in time as this is more basic strategy than the latter.

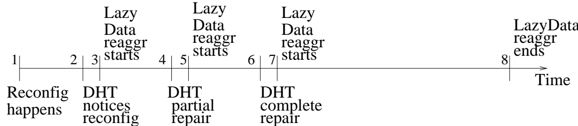

Fig. 6: Default lazy data re-aggregation timeline

Replication in space is a performance optimization strategy and depends on replication in time when the system runs out of replicas. We provide two mechanisms as part of replication in time. First, a lazy re-aggregation is performed where already received updates are propagated to the new children or new parents in a lazy fashion over time. Second, applications can reduce the probability of probe response staleness during such repairs through our flexible API with appropriate setting of the *down* knob.

Lazy Re-aggregation The DHT layer informs the AML layer about the detected reconfigurations in the network using the API shown in Figure 3. Here we explain the behavior of AML layer on the invocation of the API.

On *newParent(parent, prefix)*: If there are any probes in the outstanding-probes table that correspond to this prefix, then send them to this new parent. Then start transferring aggregation functions and already existing data lazily in the background. Any new updates, installs and probes for this prefix are sent to the parent immediately.

Note that it might be possible for a node to get an update or probe message for an attribute key for which it does not yet have any aggregation function installed on it as it might have just joined the system and is still lazily getting the data and functions from its children. Upon receiving such a probe or update, AML returns an error if invoked by a local application. And if the operation is from a child or a parent, then an explicit request is made for the aggregation function from that sender.

On *failedChild(child, prefix)*: The AML layer notes the child as inactive and any probes in the outstanding-probes table that are waiting for data from this child are re-evaluated.

On *newChild(child, prefix)*: The AML layer creates space in its data structures for this child.

Figure 6 shows the timeline for the default lazy re-aggregation upon reconfiguration. The probes that initiate between points 1 and 2 and that got affected by the reconfigurations are rescheduled by AML upon detecting the reconfiguration. Probes that complete or start between points 2 and 8 may return stale answers.

On-demand Re-aggregation The default lazy aggregation scheme lazily propagates the old updates in the system. By using *up* and *down* knobs in the Probe API, applications can force on-demand fast re-aggregation of the updates to avoid staleness in the face of reconfigurations. Note that this strategy will be useful only after the DHT adaptation is completed (Point 6 on the timeline in Figure 6).

Replication in Space Replication in space is more challenging in our system than a DHT file location application because replication in space can be achieved easily in the latter by just replicating the root node's contents. In our system, however, all internal nodes have to be replicated along with the root.

In our system, applications can control replication using the *up* and *down* knobs in the Install API; applications can reduce the latencies and possibly the probability of stale values by replicating the aggregates. The probability of staleness is reduced only if the replicated value is still valid after the reconfiguration. For example, in a file location application, an aggregated value is valid as long the node hosting the file is active, irrespective of the status of other nodes in the system. Whereas, an application that counts the number of machines in a system will suffer from staleness irrespective of replication. However, if reconfigurations are only transient (like a node temporarily not responding due to a burst of load), the replicated aggregate closely or correctly resembles the current state.

4.3 Discussion

Reconfigurations are expensive if many attributes are installed with non-zero up and down values. In the worst case, when all attributes are installed as Update-ALL and any change at one node effects aggregates at all levels, a reconfiguration incurs $O(N.\log N.m)$ communication cost where n is the number of nodes in the system and m is the number of attributes installed.

Fail-stop failure model During reconfigurations, probe latency and staleness might get affected in our system as the nodes that need to be contacted for aggregates to answer a probe are unreachable due to the change in the structure. A gossiping approach like Astrolabe distributes aggregates to all nodes – incurs communication overhead upfront before failures – to avoid increased read latencies and possibly less stale responses during failures. Our system provides different options that incur different overheads on top of basic data reconfiguration cost: (1) Update-All scheme guarantees same performance as a gossipping scheme with similar overheads, (2) on-demand reaggregation incurs increased read latency for a consistent response, and (3) lazy reaggregation incurs no extra communication cost but might return a stale value.

Handling temporary failures Astrolabe [25] can effectively handle temporary failures because of the replication of local MIBs of all nodes and the replication of aggregate values of a subtree on all nodes in that subtree. It is currently not possible to achieve similar reliability in our approach through same mechanisms because: (i) new root node for a subtree might not belong to that subtree, and (ii) subtree might split into multiple subtrees and