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 and network failures. We design, implement and evaluate a SDIMS that (1) leverages Distributed Hash Tables (DHT) 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 to 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 design and build 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, 28, 35, 36], service placement [10, 37], data sharing and caching [14, 25, 29, 33, 38], sensor monitoring and control [16, 18], multicast tree formation [4, 5, 27, 31, 34], 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 [18, 27] 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 incur high bandwidth consumption when applied for write-dominated attributes. Conversely, an approach tuned for write-dominated attributes will suffer from unnecessary query latency or imprecision for read-dominated attributes. Therefore, a SDIMS should provide different mechanisms to handle different types of attributes, and leave the policy decision of choosing a mechanism to the application installing the attribute.

Third, a SDIMS should provide *administrative 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

*Draft: Please check http://www.cs.utexas.edu/users/ypraveen for the most current version. A revised version of this paper will appear at SIGCOMM2004.

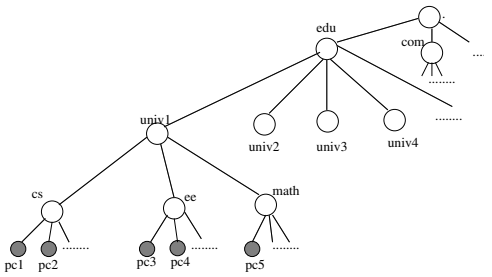

Fig. 1: Administrative hierarchy

system administrator can control what information flows out of her machines 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* [27] and *Distributed Hash Tables (DHTs)*.

Astrolabe [27] 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) [14, 24, 25, 29, 33, 38]—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 [24] 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 to be obvious that simply fusing DHTs with Astrolabe's aggregation abstraction will result in a SDIMS. However, meeting the SDIMS requirements forces a design to address four questions: (1) How to scalably map different attributes to different aggregation trees within a DHT mesh? (2) How to provide flexibility in the aggregation to accommodate different application requirements? (3) How to adapt a global, flat DHT mesh to satisfy the required autonomy and isolation properties? and (4) How to provide good robustness without unstructured gossip and total replication?

The key contributions of this paper that form the foundation of our SDIMS design are as follows.

1. We define a new aggregation abstraction that specifies both attribute type and attribute name for an attribute and associates an aggregation function with a particular attribute type and thus paving a way to utilize the DHT system's internal trees for aggregation and to achieve *scalability* with both nodes and attributes.
2. Unlike previous aggregation systems like Astrolabe [27], Ganglia [11], and DHT based systems, 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 *administrative 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 [23] nodes, we observe that the prototype achieves scalability with respect to both nodes and 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 in building a SDIMS. For example, we believe that our strategies for providing robustness will mesh well with techniques such as *supernodes* [19] and other ongo-

ing efforts to improve DHTs [26] for further improving robustness. 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, 15] to scalable aggregation tree abstractions.

In Section 2, we explain the hierarchical aggregation abstraction that SDIMS provides to applications. In Sections 3 and 4, we describe the design of our system in achieving the flexibility, scalability and administrative autonomy and isolation requirements of a SDIMS. In Section 5, we detail the implementation of our prototype system. Section 6 addresses the issue of adaptation to the topological reconfigurations. In Section 7, we present the evaluation of our system through large-scale simulations and microbenchmarks on real networks. Section 8 details the related work and Section 9 summarizes our contribution and points out the future research directions.

2 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 [27].

Our aggregation abstraction is defined across a tree spanning all nodes in the system. Each physical node in the system is a leaf and each subtree represents a logical group of nodes. Note that logical groups can correspond to administrative domains (e.g., department or university) or groups of nodes within a domain (e.g., 10 workstations on a LAN in the CS department). An internal non-leaf node of the aggregation tree is simulated by one or more physical nodes that belong to the subtree for which the non-leaf node is the root. We describe how to form such trees in a later section.

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: $V_{i,type,name} = f_{type}(V^0_{i-1,type,name}, V^1_{i-1,type,name}, \ldots, V^{k-1}_{i-1,type,name})$.

Although our system allows arbitrary aggregation functions, it is often desirable that aggregation functions satisfy the *hierarchical computation* property [18]: $f(v_1,...,v_n) = f(f(v_1,...,v_{s_1}), f(v_{s_1+1},...,v_{s_2}),..., f(v_{s_k+1},...,v_n))$, where v_i is the value of an attribute 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 an attribute type stores values as tuples $(sum, count)$ and defines the aggregation function as $avg(v_1,...,v_n) = (\sum_{i=0}^{n} v_i.sum, \sum_{i=0}^{n} v_i.count)$, the attribute 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. Therefore our system provides only weak consistency guarantees – specifically eventual consistency as defined in [27].

3 Flexibility

A major innovation of our work is enabling flexible computation and aggregate value propagation. The definition of the aggregation abstraction allows considerable flexibility in how, when, and where aggregate values are computed and propagated. While previous systems [27, 11, 38, 29, 33, 25] choose to implement a single static strategy, we argue that a SDIMS should provide *flexible computation and propagation* to be able to efficiently support wide variety of applications with diverse requirements. In order to provide this flexibility, we develop a simple interface that decomposes the aggregation abstraction into three pieces of functionality: install, update and probe.

The definition of aggregation abstraction allows our system to provide a continuous spectrum of strategies ranging from lazy aggregate computation and propagation on reads to an aggressive immediate computation and propagation on writes. In Figure 2, we illustrate both extreme strategies and an intermediate strategy. Under the lazy *Update-Local* computation and propagation strategy, an update (aka write) only affects local state. Then, a probe (aka read) that reads a level-i aggregate value is sent up the tree to the issuing node's level-i ancestor and then down the tree to the leaves. The system then computes the desired aggregate value at each layer up the tree until the level-i ancestor that holds the desired value. Finally, the level-i ancestor sends the result down the tree to the issuing node. In the other extreme case of the aggressive *Update-All* immediate computation and propagation on reads [27], when an update occurs, changes are aggregated up the tree, and each new aggregate value is broadcast to all of a node's descendants. In this case, each level-i node not only maintains the aggregate values for the

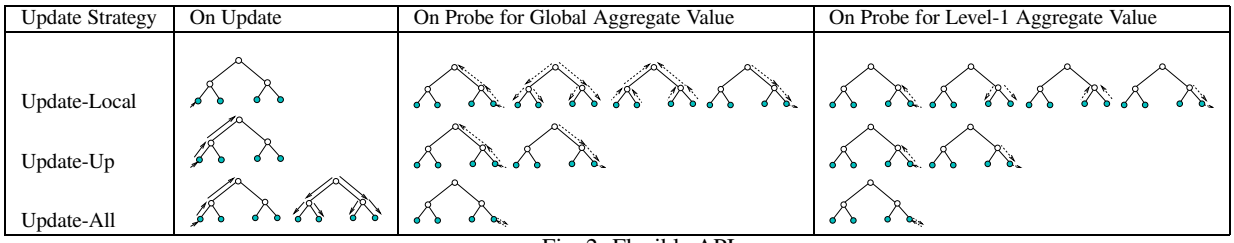

Fig. 2: Flexible API

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

Table 1: Arguments for the install operation

level-i subtree but also receives and locally stores copies of all of its ancestors' level-j ($j > i$) aggregation values. Also, a leaf satisfies a probe for a level-i aggregate using purely local data. In an intermediate *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. Finally, notice that other strategies also exist. In general, an Update-Up`k`-Down`j` strategy aggregates up to the `k`th level and propagates the aggregate values of a node at level l (s.t. $l \leq$ `k`) downwards for `j` levels.

A SDIMS must provide a wide range of flexible computation and propagation strategies to applications for it to be a general abstraction. An application should be able to choose a particular mechanism based on its read-to-write ratio that reduces the bandwidth consumption while attaining the required responsiveness and precision. Note that the read-to-write ratio 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 works well for *read-dominated* attributes but suffers high bandwidth consumption when applied for *write-dominated* attributes. Conversely, an approach like Update-Local works well for *write-dominated* attributes but suffers from unnecessary query latency or imprecision for *read-dominated* attributes.

Our system also allows the flexible computation and propagation to happen non-uniformly across the aggregation tree – different up and down levels in different subtrees – so that the applications can efficiently adapt with the spatial and temporal heterogeneity of the read and write operations. With respect to spatial heterogeneity, access patterns may differ for different parts of the tree; hence, the need for different propagation strategies for different parts of the tree depending on the workload. Similarly with respect to temporal heterogeneity, access patterns may change over time and hence the need for different computation and propagation patterns over time.

3.1 Aggregation API

We provide the flexibility described above by splitting the aggregation API into 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 value for an attribute, and *Probe()* obtains an aggregate value for a specified subtree. Install interface allows applications to specify the `k` and `j` parameters of Update-Up`k`-Down`j` strategy along with the aggregation function. Update interface invokes the aggregation of an attribute on the tree according to corresponding aggregation function's aggregation strategy. Probe interface not only allows the applications to obtain the aggregated value for a specified tree but also allows probing node to *continuously* fetch the values for a specified time; thus enabling the applications to adapt to spatial and temporal heterogeneity. The rest of the section describes these three interfaces in detail.

3.1.1 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. Installed functions are soft state that must be periodically renewed or they will be garbage collected at *expTime*. Also 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 [27].

The arguments *up* and *down* specify the aggregate computation and propagation strategy *Update-Up*`k`-*Down*`j`, where `k` = *up* and `j` = *down*. 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

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: at all levels)	X
up	How far up to go and re-fetch the value (default: none)	X
down	How far down to go and re-aggregate (default: none)	X
expTime	Expiry Time	

Table 2: Arguments for the probe operation

read/write frequency. However, our implementation always simply follows these directives.

The optional *domain* argument, if present, indicates that the aggregation function should be installed on all nodes belonging to the specified domain; if not specified, the function is installed on all nodes in the system.

3.1.2 Update

The update operation takes three arguments *attrType, attrName* and *value* and creates a new (attrType, attrName, value) tuple or updates the value of an old tuple with matching *attrType* and *attrName* at a leaf node.

The update interface meshes with installed aggregate computation and propagation strategy to provide flexibility. In particular, as outlined above and described in detail in Section 5, 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. Notice that our abstraction associates an aggregation function with only an *attrType* but lets updates specify an *attrName* along with the *attrType*. This technique helps us in leveraging DHTs for achieving scalability with respect to nodes and attributes. We elucidate this aspect in Section 4.

3.1.3 Probe

Whereas update propagates the aggregates in the system implementing the install time specified global *Update-Up*k*-Down*j strategy, a probe operation collects the aggregated values at the application-queried levels either continuously for a specified time or just once; and thus provides capability to adapt for spatial and temporal heterogeneity. The complete argument set for the probe operation is shown in Table 2. Along with the *attrName* and the *attrType* arguments, a *level* argument specifies the level at which the answers are required for an attribute.

The probes with *mode* set to *continuous* and with finite *expTime* enable applications to handle spatial and temporal heterogeneity. In the continuous mode for a probe at level l by a node A, on any change in the value at any leaf node B of the subtree rooted at level l ancestor of the node A, aggregation is performed at all ancestors of B till level l and the aggregated value is propagated down at least to the node A irrespective of the install time specified *up* and *down* parameters. Thus the probes in continuous mode from different nodes for aggregate values at different levels handles spatial heterogeneity. By setting appropriate *expTime*, the applications extend the same technique to handle temporal heterogeneity.

The *up* and *down* arguments enable applications to perform on-demand fast re-aggregation during reconfigurations. 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. When used, the *up* and *down* arguments are interpreted as described in Section 3.1.1. In Section 6, we explain how applications can exploit these arguments during reconfigurations.

4 Scalability

Our system accommodates a large number of participating nodes, and it allows applications to install and monitor a large number of data attributes. Our design achieves scalability with respect to both nodes and attributes through two key ideas. First, in contrast to previous systems [27, 11], our aggregation abstraction specifies 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 name space. For example, to construct a file location service, our interface allows us to install a single function that computes 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, splitting values associated with different names into different aggregation values allows our system to leverage Distributed Hash Tables(DHT) to map different names to different trees and thereby spread the function's logical root node's load and state across multiple physical nodes.

Second, our system employs simple modifications to DHTs to ensure the required autonomy and isolation properties. while DHTs offer solution for scalability with the nodes and attributes, they do not guarantee that the administrative autonomy is preserved in the aggregation trees. Having aggregation trees that conform with the administrative hierarchy helps SDIMS provide important autonomy, security, and isolation properties [27]. 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. We present two properties – Path Locality and Path Convergence – that a DHT routing should satisfy to guarantee the conformation to administrative autonomy requirement and then present simple modifications to an existing DHT algorithm that guarantee these properties.

In the following sections, we describe how DHTs are used to form multiple aggregation trees and the details of our Autonomous DHT(ADHT).

4.1 Multiple Aggregation Trees

We exploit the Distributed Hash Tables (DHT) to form multiple aggregation trees. Existing DHTs can be viewed as a mesh 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 [24]. Now, by aggregating an attribute along the aggregation tree corresponding to $DHTtree_k$ 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.

4.2 Administrative Autonomy

To conform to administrative autonomy requirement, a DHT should satisfy two properties:

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 either already support path locality [14] or can support easily by setting the domain nearness as the distance metric [13, 3]. But they do not *guarantee* path convergence as those systems try to optimize the search path to the root to reduce response latency.

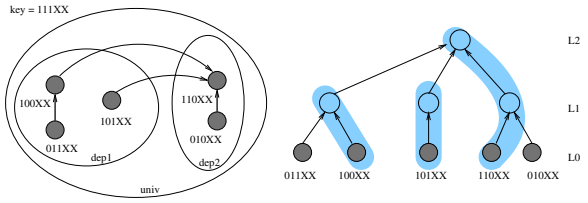

Fig. 3: Example shows how isolation property is violated with original Pastry. We also show the corresponding aggregation tree.

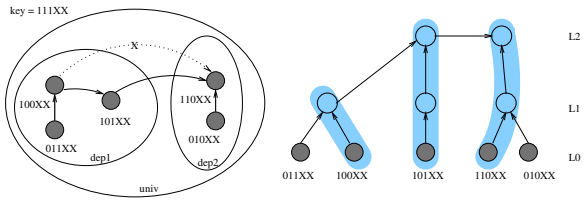

Fig. 4: Autonomous DHT satisfying the isolation property. Also the corresponding aggregation tree is shown.

In the rest of this section we explain how an existing DHT, Pastry [29], 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.

4.2.1 Pastry

In Pastry [29], 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* [24] 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 illustrated in Figure 3, 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 so that 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.

4.2.2 Autonomous DHT

To provide autonomy properties to an aggregating autonomous DHT (ADHT), the system's route table construction algorithm must provide a single exit point in each domain for a 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 $101XX$ for key $111XX$. In Section 5, we explain more about the extra virtual node that is created at node with id $111XX$ to aggregate at level L2 in this special case.

In the ADHT, 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 the ADHT 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 [29], a node wishing to join ADHT 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 4.2.2. 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 ADHTroute(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
```

5 Prototype Implementation

The internal design of our SDIMS prototype comprises of two layers: the Autonomous DHT (ADHT) layer manages the overlay topology of the system and the Aggregation Management Layer (AML) maintains attribute tuples, performs aggregations, stores and propagates aggregate values. Given the ADHT construction described in Section 4.2, each node implements an Aggregation Management Layer (AML) to support the flexible API described in Section 3. In this section, we describe the internal state and operation of the AML layer of a node in the system. We defer the discussion on the interfaces between AML layer and the ADHT layer and how SDIMS handles network and node reconfigurations to Section 6.

We refer to a tuple store of (attribute type, attribute name, value) tuples as a Management Information Base

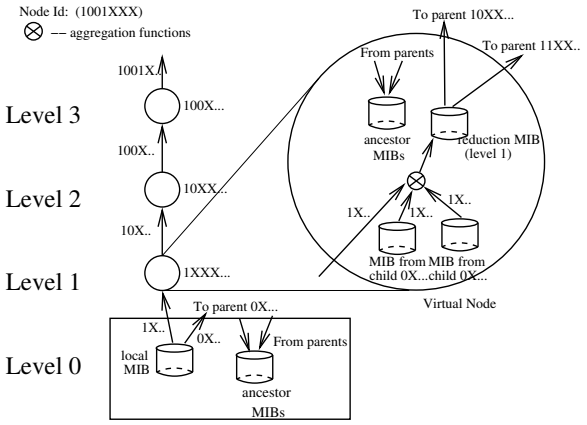

Fig. 5: Example illustrating the data structures and the organization of them at a node.

or MIB, following the terminology from Astrolabe [27] (originally used in the context of SNMP [32]). We refer to the pair (attribute type, attribute name) as an *attribute key*.

Each physical node in the system acts as several virtual nodes in the AML(e.g., Figure 5): a node acts as leaf for all attribute keys, as a level-1 subtree root for keys whose hash matches the node's ID in b prefix bits (where b is the number of bits corrected in each step of the ADHT's routing scheme), as a level-i subtree root for attribute keys whose hash matches the node's ID in initial $i * b$ bits, and as the system's global root for for attribute keys whose hash matches the node's ID in more prefix bits than any other node (in case of a tie, the first non-matching bit is ignored and the comparison is continued [38]). A node might be a level-i subtree root for keys matching the node's ID in only initial $(i-1) * b$ bits in some special cases as illustrated by node 101XX in Figure 4.

As Figure 5 illustrates, to support hierarchical aggregation, each virtual 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 MIB* containing aggregate values scattered *down* from ancestors. This basic strategy of maintaining child, reduction, and ancestor MIBs is based on Astrolabe [27], 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 step ($b = 4$). 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 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 [22].

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.

A virtual node n_i at level i also maintains an *ancestor MIB* to store the tuples containing attribute key and a list of aggregate values at different levels scattered down from ancestors. Note that the list for a key might contain multiple aggregate values for a same level but aggregated at different nodes (refer to Figure 4). So, the aggregate values are tagged not only with the level information, but are also tagged with the information of the node that performed the aggregation.

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 virtual 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 3.1.

Install 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 $attrType$ proceeds in two phases: first the install request is passed up the ADHT tree with the attribute key *(attrType, null)* 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.

Update The *Update* operation creates a new (attributeType, 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 aggregation function *down* argument indicates that the aggregate values be sent down to $j \geq 1$ levels, the node sends the updated value down to all of its 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, a virtual node n_k at level k stores the value in its ancestor MIB and, if $k \geq i-j$, forwards this level-i aggregate value to its children.

Probe 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 either 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 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, in 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 ADHT 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 ADHT protocol but that fail at the AML level.

6 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 ADHT layer to ensure connectivity and adaptation at the AML layer to ensure access to the data in SDIMS.

6.1 ADHT Adaptation

Our ADHT layer adaptation algorithm is same as Pastry's adaptation algorithm [29] — 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 ADHT algorithm ensure that the reconfigurations in a level i domain do not affect the probes for level i in the sibling domains.

6.2 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. 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

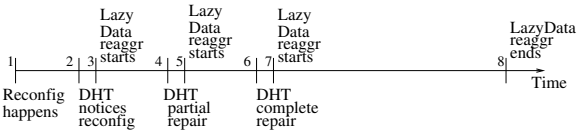

Fig. 6: Default lazy data re-aggregation time line

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 following three API – *newParent, failedChild* and *newChild*. 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 time line 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 time line 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.

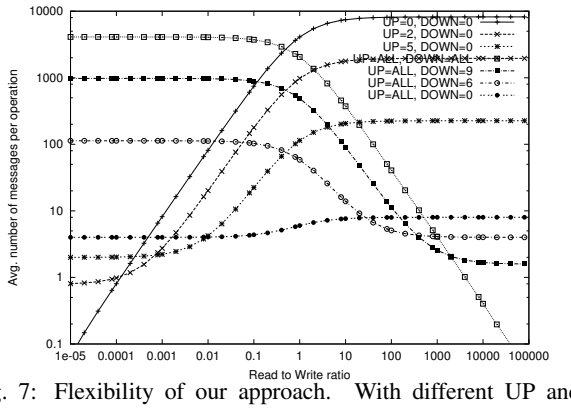

Fig. 7: Flexibility of our approach. With different UP and DOWN values in a network of 4096 nodes for different read-write ratios.

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.

7 Evaluation

We have implemented a prototype of SDIMS in Java using FreePastry framework [29] and performed large-scale simulation experiments and micro-benchmark experiments on two real networks: 187 machines in the department, and 69 machines on the Planet-Lab [23] testbed. The results from the evaluation of our prototype substantiate (i) the flexibility provided by the API of our system, (ii) the scalability with respect to both nodes and attributes achieved due to aggregating along multiple DHT trees and with the power of a flexible API, (iii) the isolation properties provided by the ADHT algorithms, and (iv) the system robustness to reconfigurations.

7.1 Simulation Experiments

Flexibility and Scalability A major innovation of our system is its ability to provide flexible computation and propagation of aggregates. In Figure 7, we demonstrate the flexibility exposed by the aggregation API explained in Section 3. We simulate a system with 4096 nodes arranged in a domain hierarchy with branching factor (bf) of

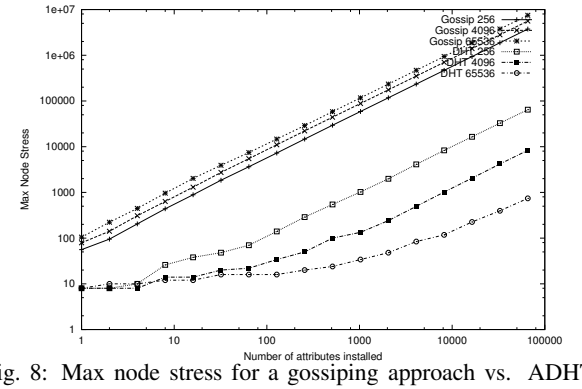

Fig. 8: Max node stress for a gossiping approach vs. ADHT based approach for different number of nodes with increasing number of *sparse* attributes.

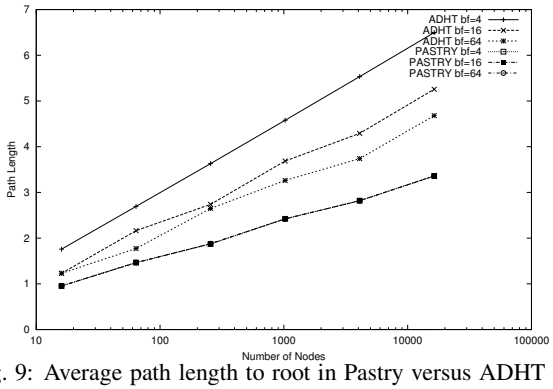

Fig. 9: Average path length to root in Pastry versus ADHT for different branching factors.

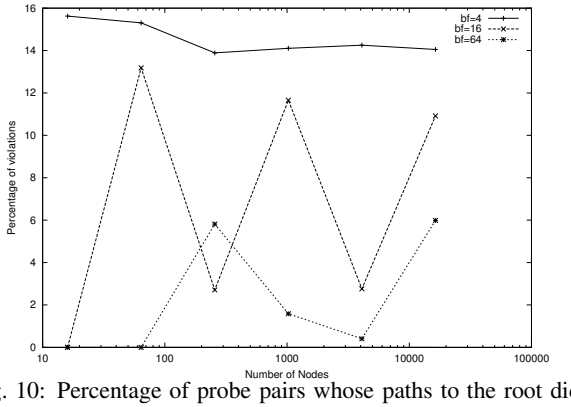

Fig. 10: Percentage of probe pairs whose paths to the root did not conform to the path convergence property.

16 and install several attributes with different *up* and *down* parameters. We plot the average number of messages per operation incurred by different attributes for a wide range of read-to-write ratios of the operations. We simulate with other sizes of networks with different branching factors and observe similar results. This graph clearly demonstrates the need for a wide range of computation and propagation strategies. While having a lower UP value will be efficient for attributes with low read-to-write ratios (write dominated applications), the probe latency, when reads do occur, will be high since the probe needs to aggregate the data from all the nodes that did not send their aggregate upwards. Conversely, applications that need to improve probe latencies increase their UP and DOWN propagation at a potential cost of increase in the overheads of writes.

Compared to existing Update-all single aggregation tree approaches [27], scalability in SDIMS comes from (1) leveraging DHTs to form multiple aggregation trees that split the load across nodes, and (2) flexible propagation that avoids propagation of all updates to all nodes. We demonstrate the SDIMS's scalability with nodes and attributes in Figure 8. For this experiment, we build a simulator to simulate both Astrolabe [27] (a gossiping, Update-All approach) and our system for an increasing number of *sparse* attributes. Each attribute corresponds to the membership in a multicast session with a small number of participants. For this experiment, the session size is set to 8, the branching factor is set to 16 and the propagation mode for SDIMS is Update-Up and the participant nodes perform continuous probes for the global aggregate value. We plot the maximum node stress (in terms of messages) observed in both schemes for different sized networks with increasing number of sessions, when the participant of each session performs an update operation. Clearly, the DHT based scheme is more scalable with respect to attributes than an Update-all gossiping scheme. Observe that, as the number of nodes increase in the system, the maximum node stress increases in the gossiping approach, while it decreases in our approach as the load of aggregation is spread across more nodes.

Administrative Hierarchy and Robustness While, the routing protocol of ADHT might lead to an increased number of hops to reach the root for a key as compared to original Pastry, the algorithm conforms to the path convergence and locality properties and thus provides autonomy and isolation properties. In Figure 9, we quantify the increased path length by comparisons with unmodified Pastry for different sized networks with different branching factors of the domain hierarchy tree. To quantify 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 10, 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.

In Figure 11, we plot the increase in path length to reach the global root node when node failures occur. In this experiment with 4096 nodes, we measure the number of hops to reach the root node before and after repairs with increasing percentage of failed nodes. The plot shows that

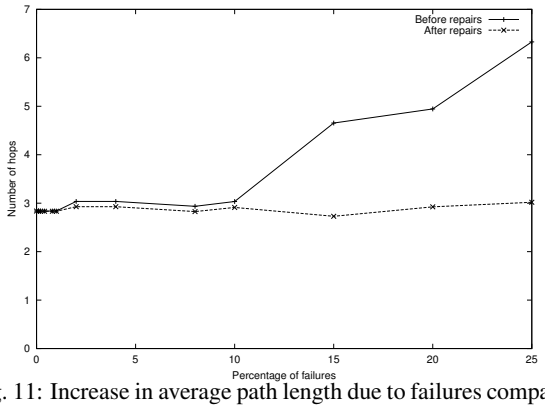

Fig. 11: Increase in average path length due to failures compared to average path length after repair

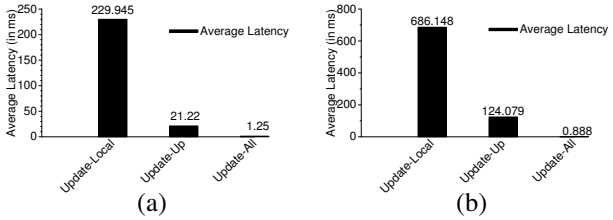

Fig. 12: Average latency of probes for aggregate at global root level with three different modes of aggregate propagation on (a) 187 department machines, and (b) 69 Planet-Lab machines

the average path length increase is minimal even when 10% of nodes fail. On further analysis of the probes, we observe that all probes are able to reach the correct root node even after 25% failures. We present some robustness results of our prototype on a real network testbed in the next section.

7.2 Testbed experiments

We run our prototype on 187 department machines and also on 69 machines of the Planet-Lab [23] testbed. We measure the performance of our system with two micro-benchmarks. In the first micro-benchmark, we install three aggregation functions of types Update-Local, Update-Up, and Update-All, perform update operation on all nodes for all three aggregation functions, and measure the latencies incurred by probes for the global aggregate from all nodes in the system. Figure 12 shows the observed latencies for both testbeds. Notice that the latency in Update-Local is very high in comparison to the Update-UP policy. This is because latency in Update-Local is affected by the presence of even a single slow machine or a single machine with a high latency connectivity in the network.

In the second benchmark, we install one aggregation function of type Update-Up that performs sum operation on an integer valued attribute. Each node is updated with a value of 10 for the attribute. Then we monitor the latencies and results returned on the probe operation for global aggregate on one chosen node, while we kill some nodes after every few probes. Figure 13 shows the results on the

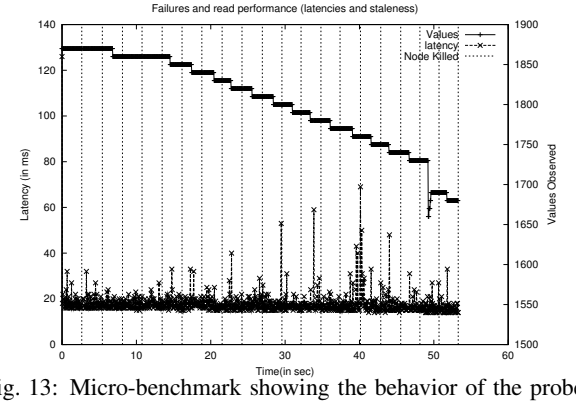

Fig. 13: Micro-benchmark showing the behavior of the probes from a single node when failures are happening at some other nodes. All nodes assign a value of 10 to the attribute. This experiment is done on 187 department machines.

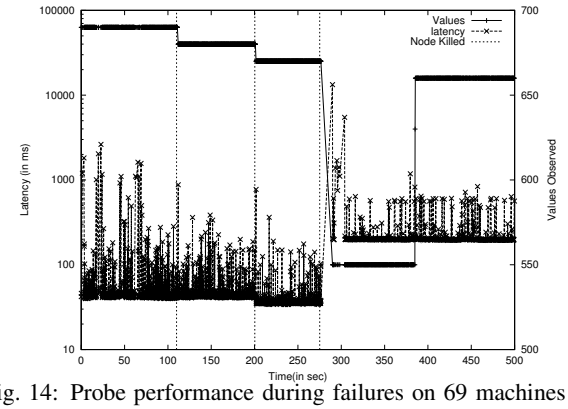

Fig. 14: Probe performance during failures on 69 machines of Planet-Lab testbed

departmental testbed. The dip in the observed values at around 49th second is due to the termination of the root node for the aggregation tree and subsequent reconfigurations. Due to the nature of the testbed (machines in a department), there is not much change in the latencies even in the face of reconfigurations. In Figure 14, we present the results of the experiment on Planet-Lab testbed. The root node of the aggregation tree is terminated after about 275 seconds. There is a 5X increase in the latencies after the death of the initial root node as a more distant node becomes the root node after repairs. In both experiments, the values returned on probes start reflecting the correct situation within a small amount of time after the failures.

From both the testbed benchmark experiments and the simulation experiments on flexibility and scalability, we conclude that (1) the flexibility provided by SDIMS allows applications to tradeoff read-write overheads, staleness, read latency and sensitivity to slow machines, (2) a good default aggregation strategy is *Update-Up* which has moderate overheads on both reads and writes, moderate read latencies and staleness values, and is scalable with respect to both nodes and attributes, and (3) the small domain sizes are the cases where DHT algorithms does not provide path convergence more often and SDIMS ensures

path convergence with only a moderate increase in path lengths.

8 Related Work

The aggregation abstraction we use in our work is heavily influenced by the Astrolabe [27] project. Astrolabe adopts Propagate-All and unstructured gossiping techniques to attain robustness [1]. However, any gossiping scheme requires aggressive replication of the aggregates. While such aggressive replication is efficient for *read-dominated* attributes, it incurs high message cost for attributes with a small read-to-write ratio. Our approach provides flexible API for applications to set propagation rules according to their read-to-write ratios. Such flexibility is attainable in our system because of our design choice to aggregate on the structure rather than through gossiping.

Several academic [11, 18, 36] and commercial [35] distributed monitoring systems have been designed to monitor the status of large networked systems. Some of them are centralized where all the monitoring data is collected and analyzed at a single central host. Ganglia [11, 20] uses a hierarchical system where the attributes are replicated within clusters using multicast and then cluster aggregates are further aggregated along a single tree. Sophia [36] is a distributed monitoring system, currently deployed on Planet-Lab [23], and is designed around a declarative logic programming model where the location of query execution is both explicit in the language and can be calculated in the course of evaluation. This research is complementary to our work; the programming model can be exploited in our system too. TAG [18] collects information from a large number of sensors along a single tree.

The observation that DHTs internally provide a scalable forest of reduction trees is not new. Plaxton et al.'s [24] original paper describes not a DHT, but a system for hierarchically aggregating and querying object location data in order to route requests to nearby copies of objects. Many systems—building upon both Plaxton's bit-correcting strategy [29, 38] and upon other strategies [21, 25, 33]—have chosen to hide this power and export a simple and general distributed hash table abstraction as a useful building block for a broad range of distributed applications. Some of these systems internally make use of the reduction forest not only for routing but also for caching [29], but for simplicity, these systems do not generally export this powerful functionality in their external interface. Our goal is to develop and expose the internal reduction forest of DHTs as a similarly general and useful abstraction and building block. Dabek et al [9] propose common APIs (KBR) for structured peer-to-peer overlays that facilitate the application development to be independent from the underlying overlay. While KBR facilitates the deployment of our abstraction on any DHT implementation that supports KBR API, it does not provide any interface to access the list of children for different prefixes.

While search application is a predominant target application for DHTs, several other applications like multicast [4, 5, 31, 34], file storage [8, 17, 30], and DNS [7] are also built using DHTs. All of these applications implicitly perform aggregation on some attribute, and each one of them must be designed to handle any reconfigurations in the underlying DHT. With the aggregation abstraction provided by our system, designing and building of such applications becomes easier.

Internal DHT trees typically do not satisfy domain locality properties required in our system. Castro et al. [3] and Gummadi et al. [13] 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 is considered. In SDIMS, we expect the size of administrative at lower levels to be much less than 256 and it is at these sizes that the convergence fails more often(Refer to Graph 10. SkipNet [14] provides domain restricted routing where a key search is limited to the specified domain. This interface can be used to ensure path convergence by searching in the lowest domain and move up to the next domain when the search reaches the root in the current domain. While this strategy guarantees path convergence, we loose 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).

There are some ongoing efforts to provide the relational database abstraction on DHTs: PIER [15] and Gribble et al. [12]. This research mainly focuses on supporting "Join" operation for tables stored on the nodes in a network. We consider this research to be complementary to our work; the approaches can be used in our system to handle composite probes – e.g., *find a nearest machine with file "foo" and has more than 2 GB of memory*.

9 Conclusions

This paper presents a Scalable Distributed Information Management System (SDIMS) that aggregates information in large-scale networked systems and that can serve as a basic building block for a broad range of applications. For large scale systems, *hierarchical aggregation* is a fundamental abstraction for scalability. We build our system by picking and extending ideas from Astrolabe and DHTs to achieve (i) scalability with respect to both nodes and attributes through a new aggregation abstraction that helps leverage DHT's internal trees for aggregation, (ii) flexibility through a simple API that lets applications control propagation of reads and writes, (iii) autonomy and isolation properties through simple augmentations of current DHT algorithms, and (iv) robustness to node and network reconfigurations through lazy reaggregation, on-demand reaggregation, and tunable spatial replication.

Our system is still in a nascent state. The initial work does provide evidence that we can achieve scalable dis-

tributed information management by leveraging aggregation abstraction and DHTs. Our work also opens up many research issues in different fronts that need to be solved. Below we enumerate some future research directions.

1. Robustness: In our current system, in spite of our current techniques, reconfigurations are costly. Malkhi et al. [19] propose Supernodes to reduce the number of reconfigurations at the DHT level; this technique can be leveraged to reduce the number of reconfigurations at the Aggregation Management Layer.
2. Self-tuning adaptation: The read-to-write ratios for applications are dynamic. Instead of applications choosing the right strategy, the system should be able to self-tune the aggregation and propagation strategy according to the changing read-to-write ratios.
3. Handling Composite Queries: Queries involving multiple attributes pose an issue in our system as different attributes are aggregated along different trees.
4. Caching: While caching is employed effectively in DHT file location applications, further research is needed to apply this concept in our general framework.

Acknowlegements

We are grateful to J.C. Browne, Robert van Renessee, and Amin Vahdat for their helpful comments on this work.

References

[1] K. P. Birman. The Surprising Power of Epidemic Communication. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2003.

[2] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–425, 1970.

[3] 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, Microsoft Research Center, 2002.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Multicast in a Cooperative Environment. In *SOSP03*, October 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A Large-scale and Decentralised Application-level Multicast Infrastructure. *IEEE JSAC (Special issue on Network Support for Multicast Communications)*, 2002.

[6] J. Challenger, P. Dantzig, and A. Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *In Proceedings of ACM/IEEE, Supercomputing '98 (SC98)*, Nov. 1998.

[7] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *SOSP01*, October 2001.

[9] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceeding of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[10] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM SOSP*, Oct. 2003.

[11] Ganglia: Distributed Monitoring and Execution System. `http://ganglia.sourceforge.net`.

[12] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do for Databases, and Vice Versa? In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB 2001)*, 2001.

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

[14] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th USENIX Symposium on Internet In Technologies and Systems*, March 2003.

[15] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the VLDB Conference*, May 2003.

[16] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the ACM Conference on Mobile Computing and Networking (MobiCom)*, pages 56–67, 2000.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th ASPLOS*, November 2000.

[18] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the OSDI*, December 2002.

[19] D. Malkhi. Dynamic Lookup Networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2002.

[20] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. In submission.

[21] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceesings of the IPTPS*, March 2002.

[22] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM SOSP*, Oct. 1997.

[23] Planetlab. `http://www.planet-lab.org`.

[24] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM*, 2001.

[26] S. Ratnasamy, S. Shenker, and I. Stoica. Routing Algorithms for DHTs: Some Open Questions. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[27] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[28] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems. In *Proceedings of the SIGOPS European Workshop*, 2002.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms(Middleware)*, November 2001.

[30] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP01*, October 2001.

[31] S.Ratnasamy, M.Handley, R.Karp, and S.Shenker. Application-level Multicast using Content-addressable Networks. In *Proceedings of the NGC*, November 2001.

[32] W. Stallings. *SNMP, SNMPv2, and CMIP*. Addison-Wesley, 1993.

[33] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*,

pages 149–160, 2001.

[34] S.Zhuang, B.Zhao, A.Joseph, R.Katz, and J.Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the NOSSDAV*, June 2001.

[35] IBM Tivoli Monitoring. `www.ibm.com/software/tivoli/products/monitor`.

[36] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[37] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, Oct 1999.

[38] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.