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 by 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, administrative isolation for security and availability, 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 administrative isolation through simple extensions to current DHT algorithms, and (4) achieves robustness 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 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 research 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 [15, 31, 37, 42], service placement [14, 43], data sharing and caching [18, 29, 32, 35, 46], sensor monitoring and control [20, 21], multicast tree formation [8, 9, 33, 36, 38], and naming and request routing [10, 11]. 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 [21, 38] 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 be scalable: it 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 over time. 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 consume high bandwidth when applied to 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 mechanisms to handle different types of attributes and leave the policy decision of tuning replication to the applications.

Third, a SDIMS should provide *administrative isolation*. In a large system, it is natural to arrange nodes in an organizational or an administrative hierarchy (e.g.,

*This is an extended version of SIGCOMM 2004 paper. Please cite the original SIGCOMM paper.

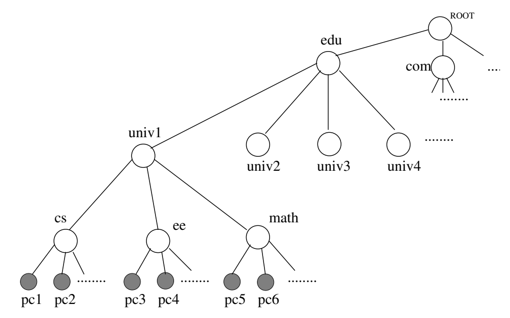

Figure 1: Administrative hierarchy

Figure 1). A SDIMS should support administrative isolation in which queries about an administrative domain's information can be satisfied within the domain so that the system can operate during disconnections from other domains, so that an external observer cannot monitor or affect intra-domain queries, and to support domain-scoped queries efficiently.

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 tradeoff the cost of adaptation with the consistency level in the aggregated results when reconfigurations occur.

We draw inspiration from two previous works: *Astrolabe* [38] and *Distributed Hash Tables (DHTs)*.

Astrolabe [38] is a robust information management system. Astrolabe provides the abstraction of a single logical aggregation tree that mirrors a system's administrative hierarchy. It provides a general interface for installing new aggregation functions and provides eventual consistency on its data. Astrolabe is 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) [18, 28, 29, 32, 35, 46]—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 [28] 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 a basic system building block alongside the DHTs.

At a 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 in 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 attain administrative isolation property? and (4) How to provide 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 and that associates an aggregation function with a particular attribute type. This abstraction paves the way for utilizing the DHT system's internal trees for aggregation and for achieving *scalability* with 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 *administrative 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 using tunable spatial replication of data aggregates or by performing fast on-demand reaggregation to augment the underlying lazy reaggregation or by doing both.

We have built a prototype of SDIMS. Through simulations and micro-benchmark experiments on a number of department machines and PlanetLab [27] 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 lower maximum node stress than unstructured gossiping schemes, achieves isolation properties at a cost of modestly increased read latency compared to flat DHTs, and gracefully handles node failures.

This initial study discusses key aspects of an ongoing 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* [22] and other ongoing efforts to improve DHTs [30] 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 [16, 19] 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 for achieving the flexibility, scalability, and administrative 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.

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 [38].

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 CS department). An internal non-leaf node, which we call *virtual node*, is simulated by one or more physical nodes at the leaves of the subtree for which the virtual 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, f_{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 SDIMS allows arbitrary aggregation functions, it is often desirable that these functions satisfy the *hierarchical computation* property [21]: $f(v_1,...,v_n)$=$f(f(v_1,...,v_{s_1}), \quad f(v_{s_1+1},...,v_{s_2}), \quad ..., f(v_{s_k+1},...,v_n))$, 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 stores values as tuples $(sum, count)$, the attribute satisfies the hierarchical computation property while still allowing the applications 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 [38].

3 Flexibility

A major innovation of our work is enabling flexible aggregate computation and propagation. The definition of the aggregation abstraction allows considerable flexibility in how, when, and where aggregate values are computed and propagated. While previous systems [15, 29, 38, 32, 35, 46] implement a single static strategy, we argue that a SDIMS should provide *flexible computation and propagation* 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.

This definition of the aggregation abstraction allows our system to provide a continuous spectrum of strategies ranging from lazy aggregate computation and propagation on reads to 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 (or write) only affects local state. Then, a probe (or 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

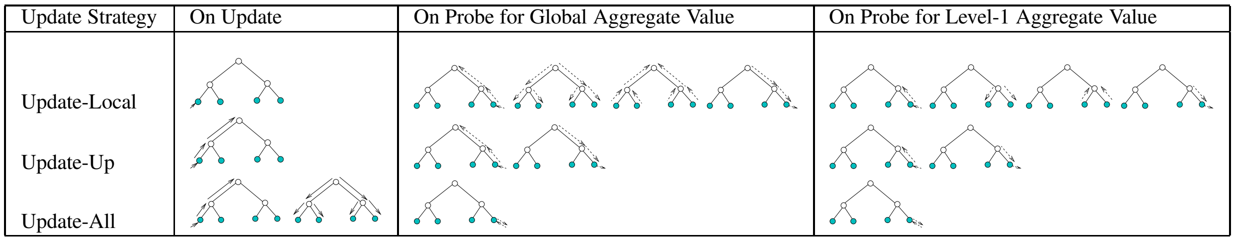
Update Strategy
On Update
On Probe for Global Aggregate Value
On Probe for Level-1 Aggregate Value
Update-Local
Update-Up
Update-All

Figure 2: Flexible API

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 writes [38], when an update occurs, changes are aggregated up the tree, and each new aggregate value is flooded to all of a node's descendants. In this case, 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. 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 exist. For example, an *Update-UpRoot-Down1* strategy (not shown) would aggregate updates up to the root of a subtree and send a subtree's aggregate values to only the children of the root of the subtree. 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. $l \leq \mathtt{k}$) downward 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 changes 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.

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

Table 1: Arguments for the install operation

SDIMS also allows non-uniform computation and propagation across the aggregation tree with different up and down parameters in different subtrees so that applications can adapt with the spatial and temporal heterogeneity of read and write operations. With respect to spatial heterogeneity, access patterns may differ for different parts of the tree, requiring different propagation strategies for different parts of the tree. Similarly with respect to temporal heterogeneity, access patterns may change over time requiring different strategies 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. The install interface allows applications to specify the `k` and `j` parameters of the Update-Up`k`-Down`j` strategy along with the aggregation function. The update interface invokes the aggregation of an attribute on the tree according to corresponding aggregation function's aggregation strategy. The probe interface not only allows applications to obtain the aggregated value for a specified tree but also allows a probing node to *continuously* fetch the values for a specified time, thus enabling an application 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

parameter	description	optional
attrType	Attribute Type	
attrName	Attribute Name	
val	Value	

Table 2: Arguments for the update operation

parameter	description	optional
attrType	Attribute Type	
attrName	Attribute Name	
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 3: Arguments for the probe operation

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*.

The arguments *up* and *down* specify the aggregate computation and propagation strategy *Update-Up*k-*Down*j. The *domain* argument, if present, indicates that the aggregation function should be installed on all nodes in the specified domain; otherwise 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 achieve scalability with respect to nodes and attributes as described in Section 4.

3.1.3 Probe

The *Probe* operation returns the value of an attribute to an application. The complete argument set for the probe operation is shown in Table 3. Along with the *attrName* and the *attrType* arguments, a *level* argument specifies the level at which the answers are required for an attribute. In our implementation we choose to return results at all levels $k < l$ for a level-l probe because (i) it is inexpensive as the nodes traversed for level-l probe also contain level k aggregates for $k < l$ and as we expect the network cost of transmitting the additional information to be small for the small aggregates which we focus and (ii) it is useful as applications can efficiently get several aggregates with a single probe (e.g., for domain-scoped queries as explained in Section 4.2).

Probes with *mode* set to *continuous* and with finite *expTime* enable applications to handle spatial and temporal heterogeneity. When node A issues a continuous probe at level l for an attribute, then regardless of the *up* and *down* parameters, updates for the attribute at any node in A's level-l ancestor's subtree are aggregated up to level l and the aggregated value is propagated down along the path from the ancestor to A. Note that continuous mode enables SDIMS to support a distributed sensor-actuator mechanism where a sensor monitors a level-i aggregate with a continuous mode probe and triggers an actuator upon receiving new values for the probe.

The *up* and *down* arguments enable applications to perform on-demand fast re-aggregation during reconfigurations, where a forced re-aggregation is done for the corresponding levels even if the aggregated value is available, as we discuss in Section 6. When present, the *up* and *down* arguments are interpreted as described in the install operation.

3.1.4 Dynamic Adaptation

At the API level, the *up* and *down* arguments in install API can be regarded as hints, since they suggest a computation strategy but do not affect the semantics of an aggregation function. A SDIMS implementation can dynamically adjust its up/down strategies for an attribute based on its measured read/write frequency. But a virtual intermediate node needs to know the current *up* and *down* propagation values to decide if the local aggregate is fresh in order to answer a probe. This is the key reason why *up* and *down* need to be statically defined at the install time and can not be specified in the update operation. In dynamic adaptation, we implement a lease-based mechanism where a node issues a lease to a parent or a child denoting that it will keep propagating the updates to that parent or child. We are currently evaluating different policies to decide when to issue a lease and when to revoke a lease.

4 Scalability

Our design achieves scalability with respect to both nodes and attributes through two key ideas. First, it carefully defines the aggregation abstraction to mesh well with its underlying scalable DHT system. Second, it refines the basic

DHT abstraction to form an Autonomous DHT (ADHT) to achieve the administrative isolation properties that are crucial to scaling for large real-world systems. In this section, we describe these two ideas in detail.

4.1 Leveraging DHTs

In contrast to previous systems [4, 15, 38, 39, 45], SDIMS's aggregation abstraction specifies both an attribute type and attribute name and associates an aggregation function with a type rather than just specifying and associating a function with a name. Installing a single function that can operate on many different named attributes matching a type improves scalability for "sparse attribute types" with large, sparsely-filled name spaces. 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. A subtree's aggregate value for (FILELOC, name) would be the ID of a 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 [6]. Supporting 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 needs to be updated and propagated to other nodes. Second, splitting values associated with different names into different aggregation values allows our system to leverage Distributed Hash Tables (DHTs) to map different names to different trees and thereby spread the function's logical root node's load and state across multiple physical nodes.

Given this abstraction, scalably mapping attributes to DHTs is straightforward. DHT systems assign a long, random ID to each node and define an algorithm to route a request for key k to a node $root_k$ such that the union of paths from all nodes forms a tree $DHTtree_k$ rooted at the node $root_k$. Now, as illustrated in Figure 3, 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, aggregating 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 leaf nodes, in a DHT-based multi-tree, each node acts as an intermediate aggregation point for some attributes and as a leaf node for other attributes. Hence, this approach distributes the onus of aggregation across all nodes.

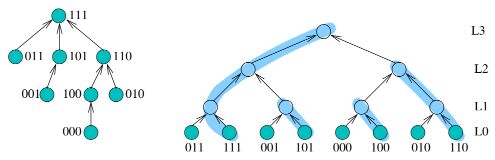

Figure 3: The DHT tree corresponding to key 111 ($DHTtree_{111}$) and the corresponding aggregation tree.

4.2 Administrative Isolation

Aggregation trees should provide administrative isolation by ensuring that for each domain, the virtual node at the root of the smallest aggregation subtree containing all nodes of that domain is hosted by a node in that domain. Administrative isolation is important for three reasons: (i) for security – so that updates and probes flowing in a domain are not accessible outside the domain, (ii) for availability – so that queries for values in a domain are not affected by failures of nodes in other domains, and (iii) for efficiency – so that domain-scoped queries can be simple and efficient.

To provide administrative isolation to aggregation trees, 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 different nodes in a domain should converge at a node in that domain.

Existing DHTs support path locality [18] or can easily support it by using the domain nearness as the distance metric [7, 17], but they do not guarantee path convergence as those systems try to optimize the search path to the root to reduce response latency. For example, Pastry [32] uses *prefix routing* in which each node's routing table contains one row per hexadecimal digit in the nodeId space where 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 common policy is to choose a nearby node according to a *proximity metric* [28] 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 a packet 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 current node uses an additional data structure, the *leaf set* containing L immediate higher and lower neighbors in the nodeId space, and 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 the message is routed directly. Pastry's expected number of routing steps is $\log n$, where n is the number of nodes, but as Figure 4 illustrates, this algorithm does not guarantee path convergence: if two nodes in a domain have nodeIds that match a key in the same number of bits, both of them can route to a third node outside the domain when routing for that key.

Simple modifications to Pastry's route table construction and key-routing protocols yield an Autonomous DHT (ADHT) that satisfies the path locality and path convergence properties. As Figure 5 illustrates, whenever two nodes in a domain share the same prefix with respect to a key and no other node in the domain has a longer prefix, our algorithm introduces a virtual node at the boundary of the domain corresponding to that prefix plus the next digit of the key; such a virtual node is simulated by the existing node whose id is numerically closest to the virtual node's id. Our ADHT's routing table differs from Pastry's in two ways. First, each node maintains a separate leaf set for each domain of which it is a part. Second, nodes use two proximity metrics when populating the routing tables – hierarchical domain proximity is the primary metric and network distance is secondary. Then, to route a packet to a global root for a key, ADHT routing algorithm uses the routing table and the leaf set entries to route to each successive enclosing domain's root (the virtual or real node in the domain matching the key in the maximum number of digits).

The routing algorithm we use in routing for a key at node with *nodeId* is shown in the Algorithm 4.2. 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. The routing algorithm guarantees that as long as the leafset membership is correct, Path Convergence property is satisfied. We use the Pastry leaf set maintenance algorithm for maintaining leafsets at all levels; after reconfigurations, once the repairs are done on a particular domain level's leafset, the autonomy properties are met at that domain level. Note that the modifications proposed for the Pastry still preserves the fault-tolerance properties of the original algorithm; rather, they enhance the fault-tolerance of the algorithm but at the cost of extra maintenance overhead.

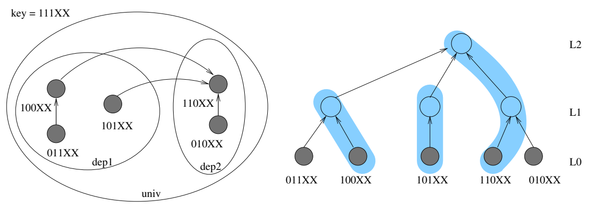

Figure 4: Example shows how isolation property is violated with original Pastry. We also show the corresponding aggregation tree.

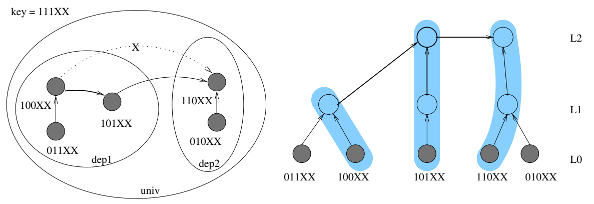

Figure 5: Autonomous DHT satisfying the isolation property. Also the corresponding aggregation tree is shown.

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:    $l \leftarrow l - 1$;
13: end while
14: this node is the root for this key
```

Properties. Maintaining a different leaf set for each administrative hierarchy 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 leaf set size, and l is the number of domain levels. Routing requires $O(lg_b n + l)$ steps compared to $O(lg_b n)$ steps in Pastry; also, each routing hop may be longer than in Pastry because the modified algorithm's routing table prefers same-domain nodes over nearby nodes. We experimentally quantify the additional routing costs in Section 7.

In a large system, the ADHT topology allows domains to improve security for sensitive attribute types by installing them only within a specified domain. Then, ag-

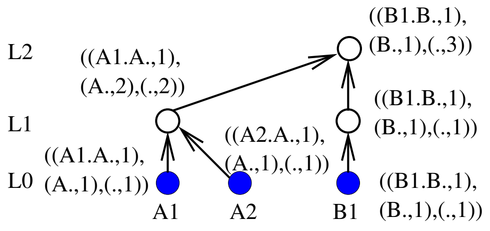

Figure 6: Example for domain-scoped queries

gregation occurs entirely within the domain and a node external to the domain can neither observe nor affect the updates and aggregation computations of the attribute type. Furthermore, though we have not implemented this feature in the prototype, the ADHT topology would also support domain-restricted probes that could ensure that no one outside of a domain can observe a probe for data stored within the domain.

The ADHT topology also enhances availability by allowing the common case of probes for data within a domain to depend only on a domain's nodes. This, for example, allows a domain that becomes disconnected from the rest of the Internet to continue to answer queries for local data.

Aggregation trees that provide administrative isolation also enable the definition of simple and efficient domain-scoped aggregation functions to support queries like "what is the average load on machines in domain X?" For example, consider an aggregation function to count the number of machines in an example system with three machines illustrated in Figure 6. Each leaf node l updates attribute *NumMachines* with a value v_l containing a set of tuples of form (Domain, Count) for each domain of which the node is a part. In the example, the node A1 with name A1.A. performs an update with the value ((A1.A.,1),(A.,1),(.,1)). An aggregation function at an internal virtual node hosted on node N with child set C computes the aggregate as a set of tuples: for each domain D that N is part of, form a tuple $(D, \sum_{c \in C}(count | (D, count) \in v_c))$. This computation is illustrated in the Figure 6. Now a query for *NumMachines* with *level* set to MAX will return the aggregate values at each intermediate virtual node on the path to the root as a set of tuples (tree level, aggregated value) from which it is easy to extract the count of machines at each enclosing domain. For example, A1 would receive ((2, ((B1.B.,1),(B.,1),(.,3))), (1, ((A1.A.,1),(A.,2),(.,2))), (0, ((A1.A.,1),(A.,1),(.,1)))). Note that supporting domain-scoped queries would be less convenient and less efficient if aggregation trees did not conform to the system's administrative structure. It would be less efficient because each intermediate virtual node will have to maintain a list of all values at the leaves in its subtree along with their names and it would be less convenient as applications that need an aggregate for a domain will have to pick values of nodes in that domain from the list returned by a probe and perform computation.

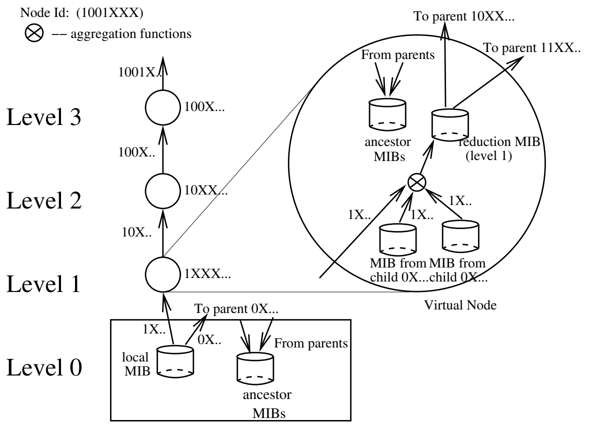

Figure 7: Example illustrating the data structures and the organization of them at a node.

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 refer to a store of (attribute type, attribute name, value) tuples as a Management Information Base or MIB, following the terminology from Astrolabe [38] and SNMP [34]. We refer an (attribute type, attribute name) tuple as an *attribute key*.

As Figure 7 illustrates, each physical node in the system acts as several virtual nodes in the AML: 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 the initial $i * b$ bits, and as the system's global root 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 [46]).

To support hierarchical aggregation, each virtual node at the root of a level-i subtree 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) an *ancestor MIB* containing aggregate values scattered *down* from ancestors. This basic strategy of maintaining child, reduction, and ancestor MIBs is based on Astrolabe [38], 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 ($b = 1$). Our system, built upon Pastry, handles multi-bit correction ($b = 4$) 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 input changes. Nodes maintain their child MIBs in stable storage and use a simplified version of the Bayou log exchange protocol (*sans* conflict detection and resolution) for synchronization after disconnections [26].

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 (see Figure 5). So, the aggregate values are tagged not only with level information, but are also tagged with ID of the node that performed the aggregation.

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. We envision various "sensor" programs and applications insert data into local MIB. For example, one program might monitor local configuration and perform updates with information such as total memory, free memory, etc., A distributed file system might perform update for each file stored on the local node.

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. 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 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 operation for 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 it reaches 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 When a level i virtual node receives an update for an attribute from a child below: it first recomputes the level-i aggregate value for the specified key, stores that value in its reduction MIB and then, subject to the function's *up* and *domain* parameters, passes the updated value to the appropriate parent based on the attribute key. Also, the level-i ($i \geq 1$) virtual node sends the updated level-i aggregate to all its children if the function's *down* parameter exceeds zero. Upon receipt of a level-i aggregate from a parent, a level k virtual node stores the value in its ancestor MIB and, if $k \geq i - down$, forwards this aggregate to its children.

Probe A *Probe* 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'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. In phase 3, the *probe aggregation phase*, when a node receives values for the specified key from each of its children, it executes the aggregate function on these values and either (a) forwards the result to its parent (if its level is less than i) or (b) initiates phase 4 (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 how to decide when a node should stop waiting for its children to respond and send up its current aggregate value. A node 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 mark 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.

At a virtual node, continuous probes are handled similarly as one-shot probes except that such probes are stored in the outstanding probe table for a time period of *expTime* specified in the probe. Thus each update for an attribute triggers re-evaluation of continuous probes for that attribute.

We implement a lease-based mechanism for dynamic adaptation. A level-l virtual node for an attribute can issue the lease for level-l aggregate to a parent or a child only if *up* is greater than l or it has leases from all its children. A virtual node at level l can issue the lease for level-k aggregate for $k > l$ to a child only if $down \geq k - l$ or if it has the lease for that aggregate from its parent. Now a probe for level-k aggregate can be answered by level-l virtual node if it has a valid lease, irrespective of the *up* and *down* values. We are currently designing different policies to decide when to issue a lease and when to revoke a lease and are also evaluating them with the above mechanism.

Our current prototype does not implement access control on install, update, and probe operations but we plan to implement Astrolabe's [38] certificate-based restrictions. Also our current prototype does not restrict the resource consumption in executing the aggregation functions; but, ‘techniques from research on resource management in server systems and operating systems [2, 3] can be applied here.

6 Robustness

In large scale systems, reconfigurations are common. Our two main principles for robustness are to guarantee (i) read availability – probes complete in finite time, and (ii) eventual consistency – updates by a live node will be visible to probes by connected nodes in finite time. During reconfigurations, a probe might return a stale value for 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 that applications can use for improved end-to-end robustness 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 [32] — the leaf sets are repaired as soon as a reconfiguration is detected and the routing table is repaired lazily. Note that maintaining extra leaf sets does not degrade the fault-tolerance property of the original Pastry; indeed, it enhances the resilience of ADHTs to failures by providing additional routing links. Due to redundancy in the leaf sets and the routing table, updates can be routed towards their root nodes successfully even during failures. Also note that the administrative isolation property satisfied by our ADHT algorithm ensures that the reconfigurations in a level i domain do not affect the probes for level i in a sibling domain.

6.2 AML Adaptation

Broadly, we use two types of strategies for AML adaptation in the face of reconfigurations: (1) Replication in time as a fundamental baseline strategy, and (2) Replication in space as an additional performance optimization that falls back on replication in time when the system runs out of replicas. We provide two mechanisms for replication in time. First, lazy re-aggregation propagates already received updates to 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* parameter.

Lazy Re-aggregation: The DHT layer informs the AML layer about reconfigurations in the network using the following three function calls – *newParent, failedChild,* and *newChild*. On *newParent(parent, prefix)*, all probes in the outstanding-probes table corresponding to *prefix* are re-evaluated. If parent is not null, then aggregation functions and already existing data are lazily transferred in the background. Any new updates, installs, and probes for this prefix are sent to the parent immediately. On *failedChild(child, prefix)*, the AML layer marks the child as inactive and any outstanding probes 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 8 shows the time line for the default lazy re-aggregation upon reconfiguration. Probes initiated between points 1 and 2 and that are affected by reconfigurations are reevaluated by AML upon detecting the reconfiguration. Probes that complete or start between points 2 and 8 may return stale answers.

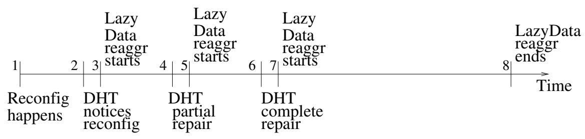

Figure 8: Default lazy data re-aggregation time line

On-demand Re-aggregation: The default lazy aggregation scheme lazily propagates the old updates in the system. Additionally, using *up* and *down* knobs in the Probe API, applications can force on-demand fast re-aggregation of updates to avoid staleness in the face of reconfigurations. In particular, if an application detects or suspects an answer as stale, then it can re-issue the probe increasing the up and down parameters to force the refreshing of the cached data. Note that this strategy will be useful only after the DHT adaptation is completed (Point 6 on the time line in Figure 8).

Replication in Space: Replication in space is more challenging in our system than in 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 control replication in space using *up* and *down* knobs in the Install API; with large *up* and *down* values, aggregates at the intermediate virtual nodes are propagated to more nodes in the system. By reducing the number of nodes that have to be accessed to answer a probe, applications can reduce the probability of incorrect results occurring due to the failure of nodes that do not contribute to the aggregate. For example, in a file location application, using a non-zero positive *down* parameter ensures that a file's global aggregate is replicated on nodes other than the root. Probes for the file location can then be answered without accessing the root; hence they are not affected by the failure of the root. However, note that this technique is not appropriate in some cases. An aggregated value in file location system is valid as long as 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 may receive incorrect results irrespective of the replication. 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 the FreePastry framework [32] and performed large-scale simulation experiments and micro-benchmark experiments on two real networks: 187 machines in the department and 69 machines on the PlanetLab [27] testbed.

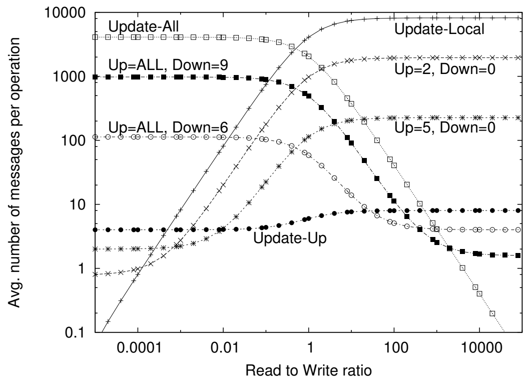

Figure 9: Flexibility of our approach. With different UP and DOWN values in a network of 4096 nodes for different read-write ratios.

In all experiments, we use static up and down values and turn off dynamic adaptation. Our evaluation supports four main conclusions. First, flexible API provides different propagation strategies that minimize communication resources at different read-to-write ratios. For example, in our simulation we observe Update-Local to be efficient for read-to-write ratios below 0.0001, Update-Up around 1, and Update-All above 50000. Second, our system is scalable with respect to both nodes and attributes. In particular, we find that the maximum node stress in our system is an order lower than observed with an Update-All, gossiping approach. Third, in contrast to unmodified Pastry which violates path convergence property in upto 14% cases, our system conforms to the property. Fourth, the system is robust to reconfigurations and adapts to failures with in a few seconds.

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 9, 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 16 and install several attributes with different *up* and *down* parameters. We plot the average number of messages per operation incurred for a wide range of read-to-write ratios of the operations for different attributes. Simulations with other sizes of networks with different branching factors reveal similar results. This graph clearly demonstrates the benefit of supporting a wide range of computation and propagation strategies. Although having a small UP value is efficient for attributes with low read-to-write ratios (write dominated applications), the probe latency, when reads do occur, may be high since the probe needs to aggregate the data from all the nodes that did not send their aggregate up. Conversely, applications that wish to

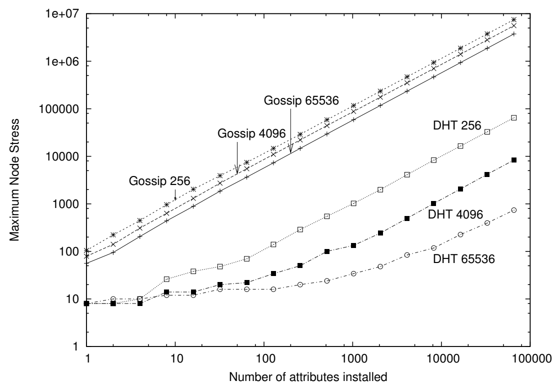

Figure 10: Max node stress for a gossiping approach vs. ADHT based approach for different number of nodes with increasing number of *sparse* attributes.

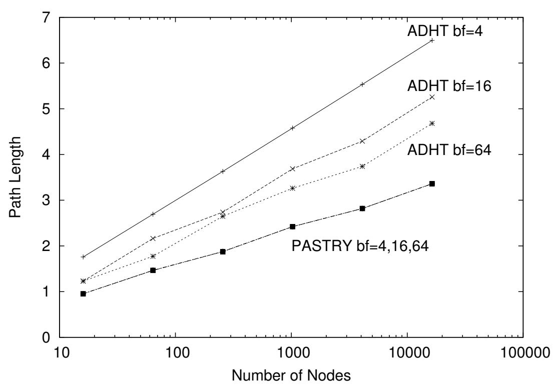

Figure 11: Average path length to root in Pastry versus ADHT for different branching factors. Note that all lines corresponding to Pastry overlap.

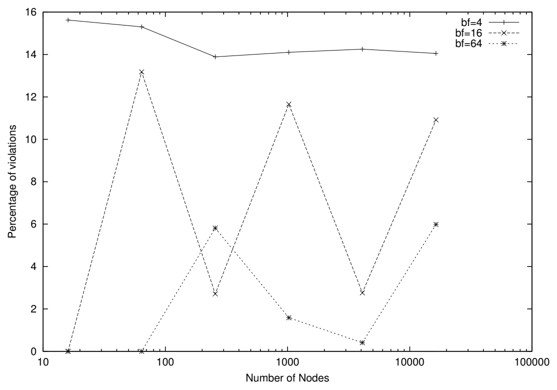

Figure 12: Percentage of probe pairs whose paths to the root did not conform to the path convergence property with Pastry.

improve probe overheads or latencies can increase their UP and DOWN propagation at a potential cost of increase in write overheads.

Compared to an existing Update-all single aggregation tree approach [38], 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. Figure 10 demonstrates the SDIMS's scalability with nodes and attributes. For this experiment, we build a simulator to simulate both Astrolabe [38] (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, 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 at some constant number of attributes, 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. Simulations with other session sizes (4 and 16) yield similar results.

Administrative Hierarchy and Robustness: Although 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 administrative isolation property. In Figure 11, 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 12, we plot the percentage of probe pairs for unmodified pastry that do not conform to the path convergence property. When the branching factor is low, the domain hierarchy tree is deeper resulting in 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.

7.2 Testbed experiments

We run our prototype on 180 department machines (some machines ran multiple node instances, so this configuration has a total of 283 SDIMS nodes) and also on 69 machines of the PlanetLab [27] 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-

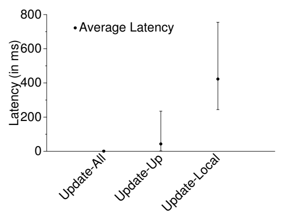 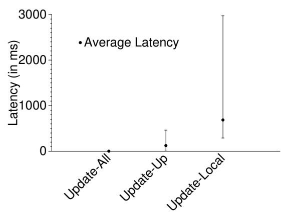

Figure 13: Latency of probes for aggregate at global root level with three different modes of aggregate propagation on (a) department machines, and (b) PlanetLab machines

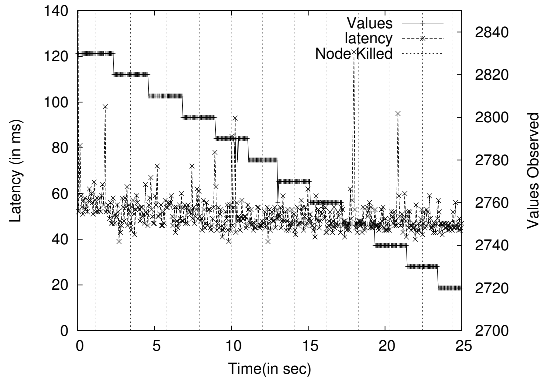

Figure 14: Micro-benchmark on department network showing the behavior of the probes from a single node when failures are happening at some other nodes. All 283 nodes assign a value of 10 to the attribute.

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 13 shows the observed latencies for both testbeds. Notice that the latency in Update-Local is high compared 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 network connection.

In the second benchmark, we examine robustness. We install one aggregation function of type Update-Up that performs sum operation on an integer valued attribute. Each node updates the attribute with the value 10. 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 14 shows the results on the departmental testbed. Due to the nature of the testbed (machines in a department), there is little change in the latencies even in the face of reconfigurations. In Figure 15, we present the results of the experiment on PlanetLab 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

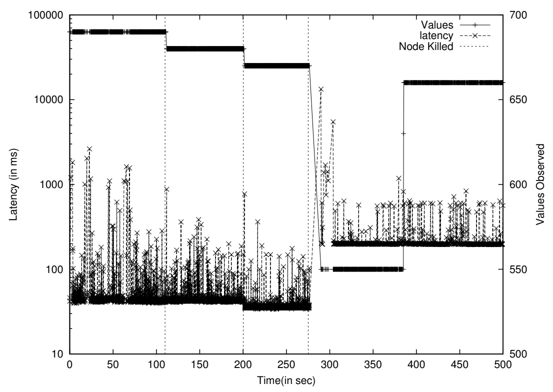

Figure 15: Probe performance during failures on 69 machines of PlanetLab testbed

probes start reflecting the correct situation within a short 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 (Figure 9), read latency, and sensitivity to slow machines (Figure 13), (2) a good default aggregation strategy is *Update-Up* which has moderate overheads on both reads and writes (Figure 9), has moderate read latencies (Figure 13), and is scalable with respect to both nodes and attributes (Figure 10), and (3) small domain sizes are the cases where DHT algorithms fail to provide path convergence more often and SDIMS ensures path convergence with only a moderate increase in path lengths (Figure 12).

7.3 Applications

SDIMS is designed as a general distributed monitoring and control infrastructure for a broad range of applications. Above, we discuss some simple microbenchmarks including a multicast membership service and a calculate-sum function. Van Renesse et al. [38] provide detailed examples of how such a service can be used for a peer-to-peer caching directory, a data-diffusion service, a publish-subscribe system, barrier synchronization, and voting. Additionally, we have initial experience using SDIMS to construct two significant applications: the control plane for a large-scale distributed file system [12] and a network monitor for identifying "heavy hitters" that consume excess resources.

Distributed file system control: The PRACTI (Partial Replication, Arbitrary Consistency, Topology Independence) replication system provides a set of mechanisms for data replication over which arbitrary control policies can be layered. We use SDIMS to provide several key functions in order to create a file system over the low-level PRACTI mechanisms.

First, nodes use SDIMS as a directory to handle read misses. When a node n receives an object o, it updates the *(ReadDir, o)* attribute with the value n; when n discards o from its local store, it resets *(ReadDir, o)* to NULL. At each virtual node, the *ReadDir* aggregation function simply selects a random non-null child value (if any) and we use the Update-Up policy for propagating updates. Finally, to locate a nearby copy of an object o, a node n_1 issues a series of probe requests for the *(ReadDir, o)* attribute, starting with $level = 1$ and increasing the $level$ value with each repeated probe request until a non-null node ID n_2 is returned. n_1 then sends a demand read request to n_2, and n_2 sends the data if it has it. Conversely, if n_2 does not have a copy of o, it sends a nack to n_1, and n_1 issues a retry probe with the *down* parameter set to a value larger than used in the previous probe in order to force on-demand re-aggregation, which will yield a fresher value for the retry.

Second, nodes subscribe to invalidations and updates to *interest sets* of files, and nodes use SDIMS to set up and maintain per-interest-set network-topology-sensitive spanning trees for propagating this information. To subscribe to invalidations for interest set i, a node n_1 first updates the *(Inval, i)* attribute with its identity n_1, and the aggregation function at each virtual node selects one non-null child value. Finally, n_1 probes increasing levels of the the *(Inval, i)* attribute until it finds the first node $n_2 \neq n_1$; n_1 then uses n_2 as its parent in the spanning tree. n_1 also issues a continuous probe for this attribute at this level so that it is notified of any change to its spanning tree parent. Spanning trees for streams of pushed updates are maintained in a similar manner.

In the future, we plan to use SDIMS for at least two additional services within this replication system. First, we plan to use SDIMS to track the read and write rates to different objects; prefetch algorithms will use this information to prioritize replication [40, 41]. Second, we plan to track the ranges of invalidation sequence numbers seen by each node for each interest set in order to augment the spanning trees described above with additional "hole filling" to allow nodes to locate specific invalidations they have missed.

Overall, our initial experience with using SDIMS for the PRACTII replication system suggests that (1) the general aggregation interface provided by SDIMS simplifies the construction of distributed applications—given the low-level PRACTI mechanisms, we were able to construct a basic file system that uses SDIMS for several distinct control tasks in under two weeks and (2) the weak consistency guarantees provided by SDIMS meet the requirements of this application—each node's controller effectively treats information from SDIMS as hints, and if a contacted node does not have the needed data, the controller retries, using SDIMS on-demand re-aggregation to obtain a fresher hint.

Distributed heavy hitter problem: The goal of the heavy hitter problem is to identify network sources, destinations, or protocols that account for significant or unusual amounts of traffic. As noted by Estan et al. [13], this information is useful for a variety of applications such as intrusion detection (e.g., port scanning), denial of service detection, worm detection and tracking, fair network allocation, and network maintenance. Significant work has been done on developing high-performance stream-processing algorithms for identifying heavy hitters at one router, but this is just a first step; ideally these applications would like not just one router's views of the heavy hitters but an aggregate view.

We use SDIMS to allow local information about heavy hitters to be pooled into a view of global heavy hitters. For each destination IP address IP_x, a node updates the attribute $(DestBW, IP_x)$ with the number of bytes sent to IP_x in the last time window. The aggregation function for attribute type $DestBW$ is installed with the Update-UP strategy and simply adds the values from child nodes. Nodes perform continuous probe for global aggregate of the attribute and raise an alarm when the global aggregate value goes above a specified limit. Note that only nodes sending data to a particular IP address perform probes for the corresponding attribute. Also note that techniques from [25] can be extended to hierarchical case to tradeoff precision for communication bandwidth.

8 Related Work

The aggregation abstraction we use in our work is heavily influenced by the Astrolabe [38] project. Astrolabe adopts a Propagate-All and unstructured gossiping techniques to attain robustness [5]. 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 a flexible API for applications to set propagation rules according to their read-to-write ratios. Other closely related projects include Willow [39], Cone [4], DASIS [1], and SOMO [45]. Willow, DASIS and SOMO build a single tree for aggregation. Cone builds a tree per attribute and requires a total order on the attribute values.

Several academic [15, 21, 42] and commercial [37] 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 central host. Ganglia [15, 23] 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 [42] is a distributed monitoring system designed with a declarative logic programming model where the location of query execution is both explicit in the language and can be calculated during evaluation. This research is complementary to our work. TAG [21] 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 [28] 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 [32, 46] and upon other strategies [24, 29, 35]—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 [32], 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.

Although object location is a predominant target application for DHTs, several other applications like multicast [8, 9, 33, 36] and DNS [11] are also built using DHTs. All these systems 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. [7] and Gummadi et al. [17] point out the importance of path convergence from the perspective of achieving efficiency and investigate the performance of Pastry and other DHT algorithms, respectively. SkipNet [18] 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 moving up to the next domain when the search reaches the root in the current domain. Although this strategy guarantees path convergence, it loses the aggregation tree abstraction property of DHTs as the domain constrained routing might touch a node more than once (as it searches forward and then backward to stay within a domain).

There are some ongoing efforts to provide the relational database abstraction on DHTs: PIER [19] and Gribble et al. [16]. 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 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) administrative isolation 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 distributed 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 ($O(m.d.\log_2(N))$ where m is the number of attributes, N is the number of nodes in the system, and d is the fraction of attributes that each node is interested in [44]). Malkhi et al. [22] 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, Amin Vahdat, Jay Lepreau, and the anonymous reviewers for their helpful comments on this work.

References

[1] K. Albrecht, R. Arnold, M. Gahwiler, and R. Wattenhofer. Join and Leave in Peer-to-Peer Systems: The DASIS approach. Technical report, CS, ETH Zurich, 2003.

[2] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. OSDI*, Oct 2000.

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI99*, Feb. 1999.

[4] R. Bhagwan, P. Mahadevan, G. Varghese, and G. M. Voelker. Cone: A Distributed Heap-Based Approach to Resource Selection. Technical Report CS2004-0784, UCSD, 2004.

[5] K. P. Birman. The Surprising Power of Epidemic Communication. In *Proceedings of FuDiCo*, 2003.

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

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

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

[9] 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.

[10] 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.

[11] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, 2002.

[12] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems. Technical Report TR-04-28, The University of Texas at Austin, 2004.

[13] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Internet Measurement Conference 2003*, 2003.

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

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

[16] 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 WebDB*, 2001.

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

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

[19] 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.

[20] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

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

[22] D. Malkhi. Dynamic Lookup Networks. In *FuDiCo*, 2002.

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

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

[25] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, pages 144–155, Sept. 2000.

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

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

[28] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM SPAA*, 1997.

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

[30] S. Ratnasamy, S. Shenker, and I. Stoica. Routing Algorithms for DHTs: Some Open Questions. In *IPTPS*, March 2002.

[31] 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.

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

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

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

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

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

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

[38] R. VanRenesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *TOCS*, 2003.

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

[40] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a wan. In *PODC*, Aug. 2001.

[41] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Elsevier Computer Communications*, 25(4):367–375, Mar. 2002.

[42] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *HotNets-II*, 2003.

[43] 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.

[44] P. Yalagandula. SDIMS: A Scalable Distributed Information Management System, Feb. 2004. Ph.D. Proposal.

[45] Z. Zhang, S.-M. Shi, and J. Zhu. SOMO: Self-Organized Metadata Overlay for Resource Management in P2P DHT. In *IPTPS*, 2003.

[46] 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.