DISSERTATION PROPOSAL

SDIMS: A Scalable Distributed Information Management System

Praveen Yalagandula

Laboratory for Advanced Systems Research
Department of Computer Sciences
University of Texas at Austin
Taylor Hall 2.124
Austin, Texas 78712-1188

Email: ypraveen@cs.utexas.edu, Telephone: (512)232-7881, Fax: (512)232-7886
URL: http://www.cs.utexas.edu/users/ypraveen

Supervising Professor: Dr. Mike Dahlin

February 24, 2004

1 Introduction

The goal of this dissertation proposal 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 [4, 13, 35, 41, 44, 45], service placement [12, 46], data sharing and caching [29, 33, 36, 40, 43, 47], sensor monitoring and control [20, 24], multicast tree formation [6, 7, 34, 38, 42], and naming and request routing [8, 9]. We therefore speculate that an 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 an 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 [24, 34] allows nodes to access information they care about while maintaining system scalability.

1.1 Requirements

To be used as a basic building block, an SDIMS should have four properties – (1) *Scalability* with respect to both nodes and attributes, (2) *Flexibility* to accommodate a broad range of applications, (3) *Administrative autonomy and isolation*, and (4) *Robustness* to reconfigurations.

Scalability 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).

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

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). An SDIMS should support administrative autonomy so that, for example, a system administrator can control what information flows out of her machines and what queries may be installed on them. And, an 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.

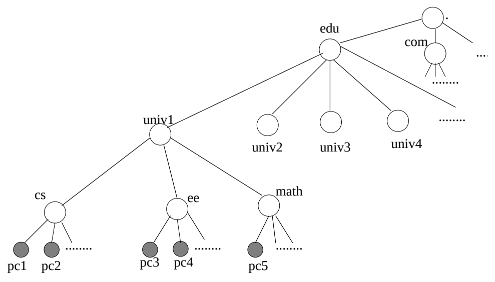


Figure 1: Administrative hierarchy

Robustness The system must be robust to node failures and disconnections. An 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 and the response latency when reconfigurations occur.

1.2 Our Approach

We draw inspiration from two previous works: *Astrolabe* [34] and *Distributed Hash Tables (DHTs)*.

Astrolabe [34] 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) [1, 16, 17, 21, 23, 26, 27, 29, 32, 33, 36, 40, 47]—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 [32] original application makes use of hierarchical aggregation to allow nodes to locate nearby copies of objects. It seems appealing to develop an SDIMS abstraction that exposes this internal functionality in a general way so that scalable trees for aggregation can be considered a basic system building block alongside the distributed hash tables.

At first glance, it might appear obvious that simply combining DHTs with Astrolabe's aggregation abstraction will result in an SDIMS. However, for such combination to meet the requirements of an SDIMS discussed above, we need to address following new challenges.

Flexibility Both Astrolabe and DHT based systems provide only a single aggregate computation and propagation policy – Astrolabe distributes aggregates to all nodes and DHT based systems typically aggregate till root of the aggregation tree. Choosing either one of the strategies will make a combined approach unable to provide flexibility to the applications. Our agenda is to provide a clean framework for flexible computation and propagation that provides several mechanisms and let applications choose a policy based on their requirements. Our current system provides three flexible API: *install*, *update* and *probe* that lets applications trade off update cost, read latency, bandwidth, replication, and staleness. As part of future work, we plan to augment our system to dynamically self-tune the computation and propagation strategy for an attribute based on the read and writes observed in the system for that attribute.

Scalability Instead of using Astrolabe's single aggregation tree conforming with administrative hierarchy and unstructured gossiping for propagation of aggregate values, leveraging DHTs to build multiple aggregation trees and propagating along those trees might improve scalability of the combined approach with respect to the attributes as the computation load is spread across multiple nodes. But the combined approach will still suffer from scalability issues because (i) the Astrolabe's aggregation abstraction does not distinguish between *sparse attributes*, attributes that are of interest to only few nodes, and *dense attributes*, attributes that are of interest to all nodes; and propagates all aggregates to all nodes, (ii) the Astrolabe's aggregation abstraction associates an aggregation function with an attribute name thus for set of attributes that need to be aggregated in same way, either a separate aggregation function for each attribute has to be maintained or a single composite attribute representing the set of all attributes has to maintained; while the former approach is inefficient in terms of communication and memory resources, the later effects scalability as the multiple DHT trees can not be efficiently utilized, and (iii) aggregating attributes along different aggregation trees makes it difficult to satisfy composite queries involving multiple attributes. In our current work, the flexible API addresses the first issue while our new aggregation abstraction, through defining an attribute by a tuple ⟨attribute type, attribute name⟩ and associating an aggregation function with a particular attribute type, addresses the second issue. We plan to tackle the composite queries as part of the future work.

Autonomy and Isolation 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. In our current system, we augment an existing DHT algorithm, Pastry [36], in order to achieve *administrative autonomy* and *isolation*. As part of the future work, we plan to show that similar simple modifications to other DHT algorithms will guarantee these properties.

Robustness The unstructured gossiping in the Astrolabe allows it to handle various types of failures; but, applying the same strategy in a combined approach will affect the scalability and the flexibility. While DHT based applications are shown to be robust, almost all of them depend on the ability to reach the root and does not depend on the structure of the DHT. By exposing the structure of a DHT routing overlay, our system is more prone to reconfigurations than those other previous applications. In our current system, We handle 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. While this solution is sufficient in scenarios where all failures are permanent, it performs poorly in situations with frequent temporary failures. We propose two approaches that we plan to further investigate and adopt in our system as part of this dissertation – K-way hashing and Supernodes [25] that provide fault-tolerance to combat temporary failures.

1.3 Results

We have built a prototype of an SDIMS. Through simulations and micro-benchmark experiments on a number of department machines and Planet-Lab [31] nodes, we observe that the prototype achieves scalability with respect to the number of nodes and the number of attributes through use of its flexible API, inflicts an order of magnitude less maximum node stress when compared to unstructured gossiping schemes, achieves autonomy and isolation properties at the cost of modestly increased read latency compared to flat DHTs, and gracefully handles node failures.

1.4 Proposal Outline

In Section 2, we explain the hierarchical aggregation abstraction that an SDIMS provides to applications. In Sections 3, 4, and 5, we describe the design of our system in achieving the flexibility and scalability while conforming to administrative autonomy and isolation requirements of an SDIMS. In Section 6, we detail the data structures and the behavior of a node in our system. Section 7 addresses the issue of adaptation to the topological reconfigurations. In Section 8, we present the evaluation of our system through large-scale simulations and microbenchmarks on real networks. Section 9 details the related work. In Section 10, we lay out the future research directions and the time line of the dissertation. Section 11 summarizes our contributions.

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 [34].

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 [24]: $f(v_1, ..., v_n) = f(f(v_1, ..., v_{s_1}), f(v_{s_1+1}, ..., v_{s_2}), ..., f(v_{s_k+1}, ..., v_n))$, 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.

Though heavily inspired by the aggregation abstraction of Astrolabe [34], our abstraction differs in specifying both an attribute type and attribute name and associating an aggregation function with a type rather than just specifying an attribute name and associating a function with a name. Installing a single function that can operate on many different named attributes matching a specific type enables our system to handle applications that install a large number of attributes with same aggregation function. For example, to construct a file location service, our interface allows us to install a single function that compute an aggregate value for any named file (e.g., the aggregate value for the (function, name) pair for a subtree would be the ID of one node in the subtree that stores the named file). Conversely, Astrolabe copes with sparse attributes by having aggregation functions compute sets or lists and suggests that scalability can be improved by representing such sets with Bloom filters [3]. Exposing sparse names within a type provides at least two advantages. First, when the value associated with a name is updated, only the state associated with that name need be updated and (potentially) propagated to other nodes. Second, for the multiple-tree system we describe in Section 4, splitting values associated with different names into different aggregation values allows our system to map different names to different trees and thereby spread the function's logical root node's load and state across multiple physical nodes.

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 conistency guarantees – specifically eventual consistency as defined in [34].

3 Flexible Computation and Propagation

A major innovation of our work is enabling flexible computation and propagation. The definition of the aggregation abstraction allows considerable flexibility in how, when, and where aggregate values are computed. While previous systems like Astrolabe [34], Ganglia [13], and DHT based systems [33, 36, 40, 47] choose to implement a single static strategy, we make a requirement that an SDIMS should provide *flexible computation and propagation* to be able to efficiently support wide variety of applications with diverse requirements. We show that providing such flexibility is feasible by presenting three simple yet powerful API – install, update, and probe – that expose flexible computation and propagation to the applications.

3.1 Motivation

The definition of the aggregation abstraction allows a continuous spectrum of computation and propagation 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 these 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 [34], 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 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. $l \leq \texttt{k}$) downwards for `j` levels.

While such large design space exist for the aggregation computation, previous systems like Astrolabe [34], Ganglia [13] and DHT based systems [33, 36, 40, 47] chose to implement a single static propagation mechanism. For example, Astrolabe implements an Update-All strategy for robustness while most DHT based systems chose Update-Up for lower communication costs while maintaining a reasonable read latency.

Why should an SDIMS provide flexibility? An 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

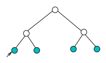
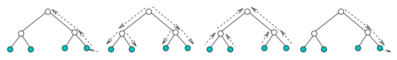
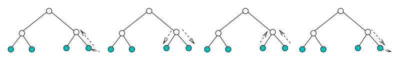
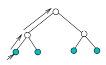
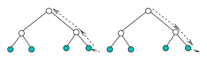
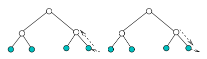
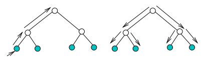
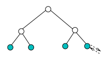
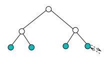
Update Strategy	On Update	On Probe for Global Aggregate Value	On Probe for Level-1 Aggregate Value
Update-Local			
Update-Up			
Update-All			

Figure 2: Flexible API

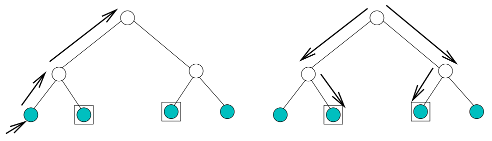


Figure 3: Spatial heterogeneity - On update, send the aggregated value to only interested nodes

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.

Also note that the abstraction 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 (e.g., refer to Figure 3). Similarly with respect to temporal heterogeneity, access patterns may change over time and hence the need for different computation and propagation patterns over time.

3.2 Our Research Agenda

Our research agenda is to build a clean framework for flexible computation that provides several mechanisms with different performance guarantees and enable applications to choose a policy based on their requirements. In the following section, we describe the API through which we expose the flexibility to the applications.

3.2.1 Completed Work: 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.

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

Table 1: Arguments for the install operation

parameter	description	optional
attrType	Attribute Type	
attrName	Attribute Name	
val	Value	

Table 2: Arguments for the update operation

Install

The *Install* operation installs an aggregation function in the system. The arguments for this operation are listed in Table 1. The *attrType* argument denotes the type of attributes on which this aggregation function is invoked. The optional *attrName* argument denotes that the aggregation function be applied only to the particular attribute with name *attrName*. An aggregation function installed with a specific *attrName* takes precedence over the aggregation function with matching *attrType* and with no *attrName* specified for updates whose name and type both match. Installed functions are soft state that must be periodically renewed or they will be garbage collected at *expTime*. Finally note that each domain specifies a security policy that restricts the types of functions that can be installed by different entities based on the attributes they access and their scope in time and space [34].

The optional *domain* argument, if present, indicates that the aggregation function should be installed on all nodes belonging to the specified domain; if this argument is absent, then the function is to be installed on all nodes in the system. This argument allows our system to provide the important administrative autonomy and isolation properties; we will discuss more about this issue in Section 5.

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

Update

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

The update interface meshes with installed aggregate computation and propagation strategy to provide

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 3: Arguments for the probe operation

flexibility. In particular, as outlined above and described in detail in Section 6, 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 allows an application to associate an aggregation function with only an *attrType* but allows updates to 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.

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 3. 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. For example, in Figure 3, the attribute is configured as *Update-Up* and the interested nodes perform a *continuous mode* probe with a large *expTime* thus receiving any updates of the aggregated value at the root node. 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 *Install* API. In Section 7, we explain how applications can exploit these arguments during reconfigurations.

3.2.2 Future Work

Self-tuning adaptation Instead of requiring applications to keep track of the exact read-to-write ratios, which we expect to be dynamic, an SDIMS system should adapt to changing read-to-write ratios by dynamically adapting the computation and propagation modes to conserve computation and communication resources. If applications indeed specify a computation and propagation strategy (for example, to reduce response latency), then an SDIMS should adapt to only those strategies that subsume application specified strategy.

Caching and Consistency While caching is employed effectively in DHT file location applications, further research is needed to apply this concept in our general framework. Caching raises the issue of consistency and both of these topics need to be handled together.

Probe Functions Our current probe API allows applications to specify only levels for fetching aggregated values. While this interface is functionally enough, a more powerful interface would allow the applications to specify *probe functions* instead of level numbers. The probe functions will be predicates defined on aggregated values that are evaluated dynamically on aggregated values at a level to decide if the required answer is found.

4 Scalability

Our system accommodates a large numbers of participating nodes, and it allows applications to install and monitor large numbers of data attributes. Our design achieves such scalability through (1) leveraging Distributed Hash Tables (DHT) to construct multiple aggregation trees for aggregating different attributes, (2) exploiting the fact that not all nodes are interested in all attributes and by providing flexible API that enables applications to control the propagation of attributes, (3) specifying attribute type and name and associating an aggregation function with type instead of just specifying attribute name and associating a function with name, and (4) augmenting existing DHTs to conform to the required administrative properties.

4.1 Motivation and Challenges

One of the key requirements of an SDIMS system is that it should be able to scale with both the number of nodes and the number of 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).

A natural way to build an aggregation tree is based on the fully qualified domain names(FQDN) of the machines [34]. Figure 1 depicts such an hierarchy. Such hierarchy construction scheme suffers from inherent skew present in the domain name allocation at various levels of the DNS system; machines simulating non-leaf nodes with large number of children incur high communication costs. For example, we plot the name distribution within the `utexas.edu` domain in Figure 4. Even though a well balanced aggregation tree might scale with the number of nodes, a single tree might not scale with the number of attributes as the machines simulating the root has to perform a large number of aggregations linear with the number of attributes.

4.2 Our approach

Our design achieves scalability with respect to both nodes and attributes through four key ideas.

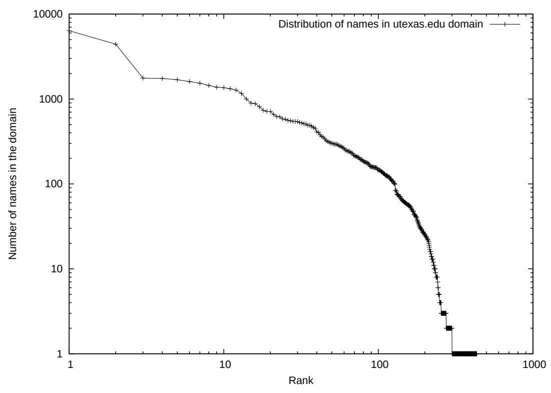


Figure 4: The name distribution in utexas.edu domain

First, while previous distributed information management systems like Astrolabe [34] and Ganglia [13] choose to aggregate on a single aggregation hierarchy achieving scalability with nodes, we leverage Distributed Hash Tables to construct multiple aggregation trees for aggregating different attributes on different trees to achieve scalability with both nodes and attributes. A single tree is unscalable with attributes as the number of aggregations that the root has to perform grows linearly with the number of attributes. By aggregating different attributes along different trees, the load of aggregation is split across multiple nodes and hence the scalability.

Second, our system exploits the fact that not all nodes are interested in all attributes and provides flexible API that allows applications to control propagation of aggregation values to only those few nodes (possibly using some few other nodes). Typically, the attribute set will consist of a small percentage of *dense attributes* that are of interest to all nodes at all times and a large percentage of *sparse attributes* in which a small number of nodes are interested at any time. For example, a file location application with one attribute per file name will create a large number of sparse attributes. The attributes corresponding to system monitoring parameters like *cpuLoad, memoryAvailable, etc.,* that are periodically updated by all nodes in the system are examples of dense attributes. While Astrolabe, with single static *Update-All* strategy, propagates the aggregate values for all attributes to all nodes, our flexible API allows applications to control the propagation of updates to only few nodes by setting appropriate propagation strategy and hence achieves scalability with the number of attributes.

Third, in contrast to previous systems [13, 34], we define a new aggregation abstraction that specifies both an attribute type and attribute name and associates 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 enables our system to efficiently handle applications that install a large number of attributes with same aggregation function. 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 such attributes by congregating them into a single set and having aggregation functions compute on such sets; and also suggests that scalability can be improved by representing such sets with Bloom filters [3]. Exposing 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.

Fourth, 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. We will elucidate this aspect in Section 5.

In the following section, we describe how DHTs are used to build multiple aggregation trees.

4.3 Completed Work: Building 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 [32]. By aggregating an attribute along the tree $DHTtree_k$ for k =*hash(attribute type, attribute name)*, different attributes will be aggregated along different trees. In comparison to a scheme where all attributes are aggregated along a single tree, the DHT based aggregation along multiple trees incurs lower maximum node stress: whereas in a single aggregation tree approach, the root and the intermediate nodes pass around more messages than the leaf nodes, in a DHT-based multi-tree, each node acts as intermediate aggregation point for some attributes and as leaf node for other attributes. Hence, this approach distributes the onus of aggregation across all nodes.

Example The pointers maintained at nodes in a 8-node DHT with 3-bit address space are tabulated in Table 4. The DHT trees for key IDs 111 and 000 are shown in Figures 5 and 6 respectively along with the corresponding aggregation tree they represent. The figures also illustrate which physical nodes simulate the virtual nodes in the aggregation trees. Note how different sets of nodes simulate the virtual nodes in different trees.

Node	Pointers
000	(1XX, 100), (01X, 010), (001, 001)
001	(1XX, 101), (01X, 011), (000, 000)
010	(1XX, 110), (00X, 000), (011, 011)
011	(1XX, 111), (00X, 001), (010, 010)
100	(0XX, 000), (11X, 110), (101, 101)
101	(0XX, 001), (11X, 111), (100, 100)
110	(0XX, 010), (10X, 100), (111, 111)
111	(0XX, 011), (10X, 101), (110, 110)

Table 4: Example pointer table for a DHT comprising of 8 nodes and addresses drawn from a 3-bit ID space

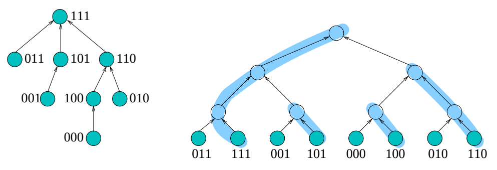


Figure 5: The DHT tree corresponding to ID 111 ($DHTtree_{111}$) and the corresponding aggregation tree.

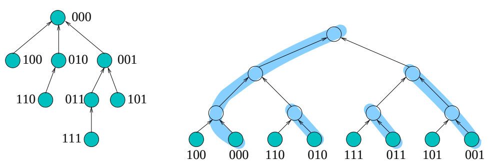


Figure 6: The DHT tree corresponding to ID 000 ($DHTtree_{000}$) and the corresponding aggregation tree.

4.3.1 Scalability with Attributes

In case of sparse attributes, DHTs provide scalability as each node in the system just needs to know about and perform aggregation on only few attributes. Here, we estimate the average number of attributes for which a node has to perform aggregations. In our DHT based aggregation tree building approach, each node is assigned to act as the root for a m/N fraction of attributes where m is the number of attributes in the system of N nodes. But each node acts as intermediate point of aggregation for many other attributes. Here we estimate the fraction of attributes a node has to work on in our design assuming that the correction is done bit-by-bit in DHT routing. Let the density factor of the attributes be d $(0 <= d <= 1)$ – each node is interested in a fraction d of all the attributes in the system. In a well formed DHT, a node will have $O(\log_2(N))$ in-degree – a small constant (about 1) number of children for each prefix. For a node A, let c_i be the number of nodes that route to this node A for the i length prefix of A – the nodes in the subtree rooted at the level i of A. A level i child of the node A sends attributes which matches A's ID in starting i bits. The average fraction of attributes that children at level i sends to node A are dependent on the number of nodes in the subtree rooted at level i and is $\frac{1-(1-d)^{c_i}}{2^i}$. Hence, the average fraction of all attributes f that node A has to perform aggregation on is given by following equation:

$$f = \left[1 - \prod_{i=0}^{i=\#bits} \left(1 - \frac{1-(1-d)^{c_i}}{2^i}\right)\right] \tag{1}$$

For small values of d, $(1-d)^{c_i}$ can be approximated to $1-(c_i * d)$. Hence the fraction f simplifies to

$$f = \left[1 - \prod_{i=0}^{i=\#bits} \left(1 - \frac{c_i d}{2^i}\right)\right] \tag{2}$$

Further approximating the value of c_i to 2^i and the number of bits to $O(\log_2(N))$, the equation reduces to

$$f = \left[1 - \prod_{i=0}^{i=\#bits} (d)\right] = O(d\log_2(N)) \tag{3}$$

We plot the Equation 1 for a million node network (#bits = 20) and with approximation of $c_i = 2^i$ in Figure 7 along with the approximation $O(d\log_2(N))$.

4.4 Future Work: Handling Composite Queries

While aggregating different attributes along different trees provides scalability with respect to attributes, solving composite queries involving two or more attributes becomes hard. For example a probe like *find a nearest machine with load less than 20 percent and has more than 2 GB of memory*. If query compositions are known in advance, then attributes can be grouped and can be aggregated along one tree. For example, load and memory of machines can be aggregated along one tree if queries as in the above example are very common. But by grouping extensively, we lose the property of load balancing. This tradeoff presents a fundamental limitation

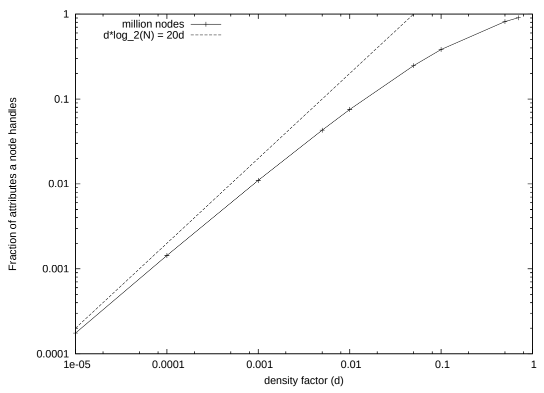


Figure 7: Fraction of attributes dealt by a node in a million node network

of distributing attributes across trees. Ongoing efforts by other researchers to provide the relational database abstraction on DHTs – PIER [19] and Gribble et al. [14] – will ease solving such composite queries.

Handling ad-hoc composite queries, whose compositions are unknown in advance, is more complicated. Here we propose solutions for handling the queries with OR and AND operations: (1) *a* OR *b*: Walk along trees corresponding to both attributes *a* and *b*. (2) *a* AND *b*: Guess the smaller of the trees corresponding to *a* and *b*, and compute the predicate along the tree. Two approaches can be used to determine the size of the trees: (a) Along with the computation of the aggregation function for an attribute, maintain a count of the number of contributing nodes or (b) Use statistical sampling techniques – randomly choose a small percentage of nodes and evaluate the attributes. For handling general logical expressions, convert the logical expressions to their Disjunctive Normal Forms (DNF) and use above AND operation for each conjunctive term. We are currently further investigating these different strategies.

5 Administrative Autonomy

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. We present two properties – Path Locality and Path Convergence – that a DHT routing should satisfy to guarantee the conformation to administrative autonomy requirement. We also present simple modifications to a DHT algorithm, Pastry, that guarantee these properties.

5.1 Motivation and Challenges

Having aggregation trees that conform with the administrative hierarchy helps an SDIMS provide important autonomy, security, and isolation properties [34]. 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.

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.

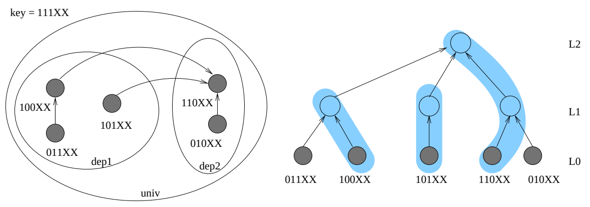


Figure 8: Example shows how isolation property is violated with original Pastry. We also show the corresponding aggregation tree.

Existing DHTs either already support path locality [17] or can support easily by setting the domain nearness as the distance metric [5, 15]. But they do not *guarantee* path convergence as those systems try to optimize the search path to the root to reduce response latency.

In the rest of this section we explain how an existing DHT, Pastry [36], does not satisfy path convergence.

5.1.1 Pastry

In Pastry [36], 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* [32] 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 8, when Pastry uses network proximity as the locality metric, it does not satisfy the desired SDIMS properties because (i) if two nodes with nodeIds match a key in same number of bits, both of them can route to a third node outside the domain when routing for that key and (ii) if the network proximity does not match the domain proximity then there is little chance that a tree will satisfy the properties. The second problem can be addressed by simply changing the proximity metric to declare that any two nodes that match in i levels of a hierarchical domain are always considered closer than two nodes that match in fewer than i levels. However, this solution does not eliminate the first problem.

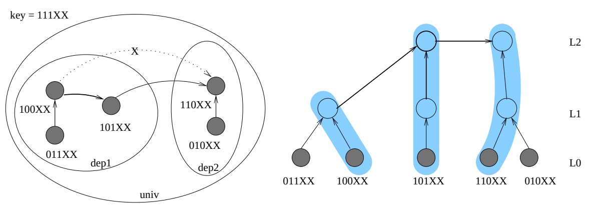


Figure 9: Autonomous DHT satisfying the isolation property. Also the corresponding aggregation tree is shown.

5.2 Our Approach

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.

5.2.1 Completed Work: 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 9, our algorithm routes towards the node with nodeId $101XX$ for key $111XX$. In Section 6, 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 [36], 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 1. 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, we meet path conver-

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
```

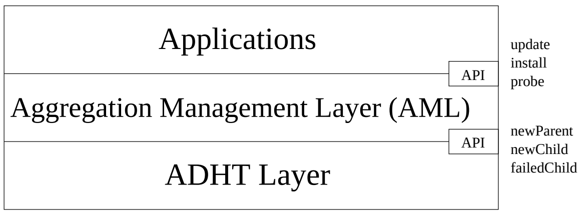


Figure 10: Layered SDIMS prototype design and interfaces.

gence property. We use the Pastry leaf set maintenance algorithm for maintaining leafsets at all levels. Being the crux for the correct operation of Pastry, this leaf set maintenance algorithm repairs the different leafsets in our algorithm correctly after reconfigurations. 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.

6 Prototype Details

This section describes the internal design of our SDIMS prototype. As illustrated in Figure 10, the design comprises two layers: the Aggregation Management Layer (AML) stores attribute tuples and calculates and stores aggregate values and the Autonomous DHT (ADHT) layer manages the internal topology of the system. Given the ADHT topology described in Section 5, 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 how an SDIMS handles network and node reconfigurations to Section 7.

We refer to a tuple store of (attribute type, attribute name, value) tuples as a Management Information Base or MIB, following the terminology from Astrolabe [34] (originally used in the context of SNMP [39]). 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 11): 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

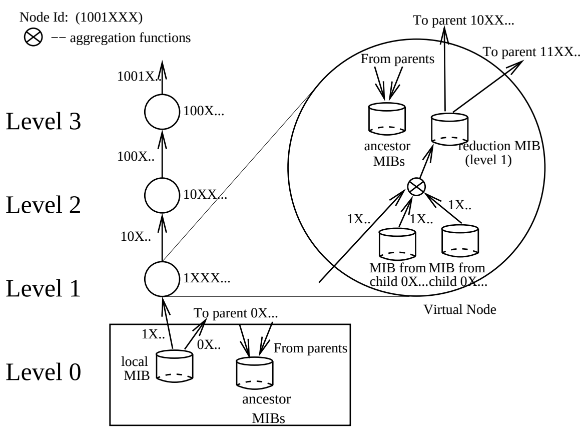


Figure 11: Example illustrating the datastructures and the organization of them at a node.

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 [47]). 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 9.

As Figure 11 illustrates, to support hierarchical aggregation, each logical node corresponding to a level-i subtree root for some attribute keys maintains several MIBs that store (1) *child MIBs* containing raw aggregate values gathered from children, (2) a *reduction MIB* containing locally aggregated values across this raw information, and (3) *ancestor MIBs* containing aggregate values scattered *down* from ancestors. This basic strategy of maintaining child, reduction, and ancestor MIBs is based on Astrolabe [34], 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 [30].

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 9). 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. Note that the virtual nodes hosted on a node can share a single aggregation function table; hence, we optimize memory requirements by maintaining only one single aggregation function table per physical node. 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 3.2.1.

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* and (optionally) attribute name *attrName* proceeds in two phases: first the install request is passed up the ADHT tree with the key *(attrType, attrName)* until reaching the root for that key within the specified domain. Then, the request is flooded down the tree and installed on all intermediate and leaf nodes.

Before installing an aggregation function, a node checks it against its per-domain access control list [34], and after installing an aggregation function, a node sets a timer to uninstall the function when it expires.

The aggregation function is provided with one argument: a vector of tuples of form $\langle child, value \rangle$, one tuple for each child. The *child* field of a tuple contains the child's name and nodeId; the *value* field is the *value* of the corresponding child for the attribute key (*null* if it is not present). The first entry in the vector corresponds to the aggregated value computed at the one level below the current level at the local node. For example, in Figure 11, the first entry of the vector at the virtual node corresponding to 10XX would be the aggregated value computed in the virtual node corresponding to 1XXX. The aggregation function is expected to return a single aggregated *value*, which is stored and then propagated according the installed propagation strategy.

Update The *Update* operation (see Table 2) 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

Algorithm 2 $f_{\texttt{fileLocation}}$(childTupleSet)

```
1: if $\exists c \in$ childTupleSet s.t. $c.value \neq NULL$ then
2:     return $c.value$
3: else
4:     return NULL
5: end if
```

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. Also, in our system, instead of returning only the level-*i* aggregate for a probe from a node requesting level-*i* aggregated value, we choose to return the aggregated values at all ancestors of the node at levels till level-*i* as these values are readily available after collecting the value at level-*i* ancestor.

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.

6.1 System Usage

In this section, we describe the usage of our SDIMS system through some examples.

File Location The aggregation function for the file location application $f_{\texttt{fileLocation}}$ is shown in Algorithm 2. This function is installed with *Update-Up* strategy ($up = ALL$ and $down = 0$). Nodes that have a file, say *foo*, will invoke update with following arguments – *(*`fileLocation`*, foo, IPaddr)* denoting that the file *foo* is available at machine with IP address *IPaddr*. Upon such update at a leaf, the aggregate value for the parent virtual node is computed by invoking function $f_{\texttt{fileLocation}}$ with *foo* and the children's tuple set as arguments. The children's tuple set is a vector of tuples of form (child, value), where value is *NULL* when the corresponding child does not have any tuple corresponding to the attribute type and name. The pseudocode for this aggregation function is shown in Algorithm 2. The aggregation is performed at all ancestor nodes of the leaf that generated the update. Any node wishing to locate file *foo* performs a probe for the aggregated values at the root by specifying $level = MAX\_LEVEL$ and thus gets aggregated value at all ancestors. Then the node can extract the nearest node hosting that file by choosing the lowest level ancestor with non-null value.

Note that the aggregation function $f_{\texttt{fileLocation}}$ can also choose a particular child's value on the basis of different metrics like computing capacity, bandwidth available, etc., The aggregation function that considers

Algorithm 3 $f_{\texttt{fileLocationBW}}$(childTupleSet)

```
 1: A $\Leftarrow \{c : c \in$ childTupleSet $\wedge\ c.value \neq NULL\}$
 2: maxBW $\leftarrow$ 0
 3: maxBWValue $\leftarrow$ NULL
 4: for all $c \in$ A do
 5:   if $c.value.BW >$ maxBW then
 6:     maxBW $\leftarrow c.value.BW$
 7:     maxBWValue $\leftarrow c.value$
 8:   end if
 9: end for
10: return maxBWValue
```

Algorithm 4 $f_{\texttt{multicast}}$(childTupleSet)

```
 1: A $\Leftarrow \{c : c \in$ childTupleSet $\wedge\ c.value \neq NULL\}$
 2: childRepList $\leftarrow \Phi$
 3: for all $c \in$ A do
 4:   childRepList $\leftarrow$ childRepList $\bigcup$ {c.value.rep}
 5: end for
 6: if childRepList $= \Phi$ then
 7:   return NULL
 8: else
 9:   rep $\Leftarrow$ chooseRep(childRepList)
10:   return (rep, childRepList)
11: end if
```

the bandwidth is shown in Algorithm 3. Instead of having only *IPaddr* in the value field of the attribute set tuple, the nodes insert *(IPaddr, BW)* tuple as values, where *BW* represents the bandwidth available at the node. The same technique can be used for building a more general resource or service location application.

For fault-tolerance, the file location aggregation function can also be installed with *Update-UpRoot-Down*`j` with *j* set to a small value greater than zero. The files stored at a node does not change that often and hence such strategy provides robustness to the application.

Multicast Tree Formation A multicast tree forming aggregation function $f_{\texttt{multicast}}$ is shown in Algorithm 4. This function is also installed with *Update-Up* aggregation and propagation strategy. A node A subscribes to a multicast session, say *sessOne*, installs the tuple *(*`multicast`, *sessOne, (IPaddr$_A$, [IPaddr$_A$]))* in its attribute set. The value field is a tuple of two parameters (rep, childRepList) denoting the chosen representative at that level and the list of representatives at the children. The subscribed nodes also perform *continuous* probes for the aggregate values at the root node($level = MAX\_LEVEL$).

A node wishing to send a multicast message invokes the *multicastRoutine* shown in Algorithm 5 locally (with the node's IP address, localIPAddr, passed as *fromIPAddr* argument). The multicast application on any node executes the same routine upon receiving a multicast message. This routine ensures that only nodes subscribed to a multicast session are involved in the forwarding process of a message for the multicast session.

In Algorithm 4, a representative is chosen from the childRepList by invoking the *chooseRep* routine. Similar to enhanced $f_{\texttt{fileLocationBW}}$ shown in the Algorithm 3, nodes can provide the locally available bandwidth in the value field of the tuples and then let *chooseRep* routine pick a representative with the highest bandwidth available from the childRepList.

Algorithm 5 multicastRoutine(sessName, message, fromIPAddr)

```
1: for level = 0 to maxLevel do
2:   (rep, childRepList) ⇐ getAggregateValue(multicast, sessName, level)
3:   if localIPAddr = rep then
4:     send message to all nodes in childRepList
5:   else
6:     if fromNode ≠ rep then
7:       send message to rep
8:     end if
9:     break
10:  end if
11: end for
12: deliver message locally
```

Distributed DoS Detection A small amount of traffic from a large number of machines to a particular target machine hosting a service might results in a denial of service (DDoS). Consider a Planetlab [31] type of environment where a single login can access multiple machines spread over the Internet. A simple aggregation function summating the traffic from each node for an IP address can detect DDoS attacks launched from such environments. The aggregation function will be installed with *Update-UP* strategy and the nodes sending messages to an IP address also install continuous probes to monitor the net bandwidth directed towards that IP address. Notice that DDoS detection requires that the aggregation system be capable of handling potentially a large number of attributes.

Obtaining Domain Based Aggregation Information Notice that the levels in our system does not correspond to the administrative levels in the system. Here we explain different options provided by our system to facilitate the applications, administrators or users in attaining the aggregate values for a domain. First, the *install* API described in Section 3 allows applications to install an aggregation function to be installed on nodes belonging to only a certain domain. For example, an application wishing to obtain to the aggregate value for nodes in the department *dep* of an university *univ* can install with domain argument set to *dep.univ.edu*. Second, the aggregation function and the syntax of the values can be defined such that the aggregated values at all domain levels is maintained in the values. For example, node *node.dep.univ.edu* installs value as a vector of tuples $((node.dep.univ.edu, value), (dep.univ.edu, value), (univ.edu, value), (edu, value))$ representing the aggregated values at different domain levels as known at this level. The aggregation function at a virtual node also outputs the value similar to this vector of tuples; for each domain level, it considers the children who are part of that domain and performs aggregation on them and forms a tuple. Now the applications can get the aggregated values at all levels by probing for the aggregate value at the root. As the SDIMS trees conform to administrative and autonomy properties, an application on a node can easily extract the aggregate value at different domain levels of the node using the list of aggregate values it obtained for the probe.

Probe Functions The file location and multicast examples discussed above could be made more efficient if we provide applications with a capability to define a probe function in the *Probe* API. In file location example, nodes probing for a file just need one non-null aggregate value; hence the probe need not be done for the root level aggregate but can be stopped at a level when a non-null aggregate value is observed. Similarly in the multicast example, a node just needs to obtain the aggregate values only for those levels at which the node is chosen as the representative and the value at next one higher level. While implementation of such powerful probe function capability is obvious for a *Update-UP* strategy, the implementation is not so straightforward in other propagation strategies and for continuous mode probes. We are currently further exploring this issue.

7 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 Aggregation Management Layer (AML) (Refer to Figure 10) to ensure access to the data in an SDIMS.

7.1 ADHT Adaptation

Our ADHT layer adaptation algorithm is same as Pastry's adaptation algorithm [36] — 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 domain D_1 do not affect the probes in a sibling domain D_2 as long as the probes are for such levels whose roots fall within the domain D_2.

7.2 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 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.

7.2.1 Replication in Time

Lazy Re-aggregation The DHT layer informs the AML layer about the detected reconfigurations in the network using three API shown in Figure 10. 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.

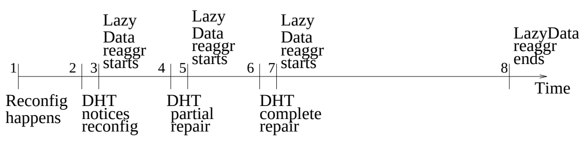


Figure 12: Default lazy data re-aggregation timeline

Figure 12 shows the timeline for the default lazy re-aggregation upon reconfiguration. The probes that initiate between points 1 and 2 and that got affected by the reconfigurations are rescheduled by AML upon detecting the reconfiguration. Probes that complete or start between points 2 and 8 may return stale answers.

On-demand Re-aggregation The default lazy aggregation scheme lazily propagates the old updates in the system. By using *up* and *down* knobs in the Probe API, applications can force on-demand fast re-aggregation of the updates to avoid staleness in the face of reconfigurations. Note that this strategy will be useful only after the DHT adaptation is completed (Point 6 on the timeline in Figure 12).

7.2.2 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 nodes' contents as the information is not location dependent. In our system, however, the information at a node is location dependent – depends on the set of leaves of the subtree for which this node is the root. Hence, spatial replication similar to DHT file location services is not as useful.

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.

7.3 Future Work: Supernodes and K-way Hashing

An SDIMS exposes DHT trees for aggregation; hence, any change in the structure directly effects the aggregated values in the system and initiates the bandwidth expensive reaggregation procedure. Since temporary failures are very common in such a large distributed network due to temporary network and host failures, detecting and reconfiguring DHT on every such failure might result in AML incurring huge bandwidth costs just for reaggregations. A solution is to choose a long timeout in detection procedure to circumvent detecting such temporary failures. But this will effect availability as the response times of probes will increase. Here we propose two approaches – K-way hashing and Supernodes [25] that provide fault-tolerance to combat unavailability in such scenarios.

7.3.1 K-way Hashing

In k-way hashing, each attribute key is aggregated along k different trees corresponding to DHT trees formed from using k different hash functions, that are chosen a priori, on the attribute key. Updates and probes are sent along all of the k trees for an attribute. This approach masks unavailability for probes at all stages of the reconstruction in the face of reconfigurations.

7.3.2 Supernodes

The key idea is that a single nodeId is simulated by multiple nodes. All nodes that are part of a supernode replicate the behavior of a single node in normal DHT. Two approaches are possible in forming supernodes: (1) Shadowing: Each node selects k shadow nodes to form a supernode with nodeId equal to the node's nodeId and (2) Clustering: Few nodes group together to form a supernode.

Shadowing: In Shadowing technique, the number of supernodes in the system are same as the number of nodes. As long as one of the shadows of a node is up, the node's failure and recovery does not effect the system. If a shadow of a node fails, the node creates a new shadow. Based on the information that is shadowed, we have two variants: (A) shadow only DHT connectivity information: Since the structure is maintained, queries can be answered by aggregating the data and (B) shadow both connectivity information and aggregation data: this is full behavioral replication at both DHT layer and the SDIMS layer and hence applications does not perceive any difference during a node failure. While approach A masks DHT repair time, the data still needs to be reaggregated. Approach B masks both unavailability and reduce reconfiguration costs but at the expense of extra communication cost for data shadowing during normal operation.

Clustering In Clustering technique, few nodes cluster together to form a supernode. When the number of nodes increases a *high threshold* mark, the cluster is split into two smaller clusters. Similarly, when the number decreases below a *low threshold* mark, the cluster is merged with some other cluster. The *high threshold* mark should be greater than twice the *low threshold* mark for splitting to be possible and should be much more to avoid frequent splits and merges.

Issues in supernodes (i) How many replicas to choose? A larger k implies stronger robustness properties but at the cost of higher bandwidth requirements. (ii) How to choose the replicas? Choosing nearby nodes reduces the communication costs while making the scheme vulnerable to domain failures. (iii) How to perform reconstruction transparently? During reconstructions both old and new supernodes have to be maintained. (iv) How to maintain consistency between the replicas? One solution is to use gossiping between replicas that guarantees eventual consistency.

7.3.3 Analytic comparison

Assume p be the probability of a node failure and l be the path length from a node to the root for an attribute. The probability of an access failure at the node in case of k-way hashing, P_{hash}, is

$$\begin{aligned} P_{hash} &= (\text{one path fails})^k \\ &= (1-(1-p)^l)^k \end{aligned}$$

and in case of k-way shadowing, P_{shadow}, is

$$\begin{aligned} P_{shadow} &= 1-(\text{atleast one node does not fail})^l \\ &= 1-(1-p^k)^l \end{aligned}$$

For k-way clustering, the probability of an access failure is similar to P_{shadow} with l replaced by $l-\lg k$ in the exponent.

These functions are compared in Figure 13 for $l = 10$ and $k = 4$. Clearly, supernode approach is more robust than K-way hashing.

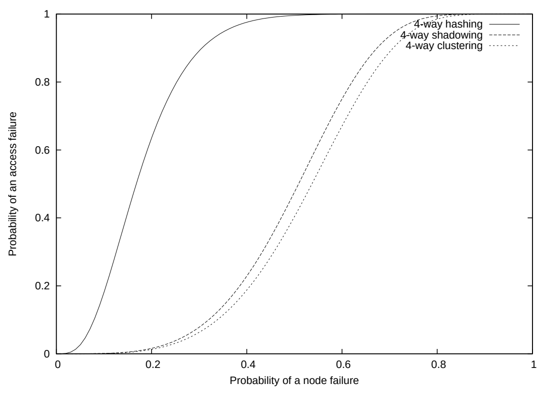


Figure 13: K-way hashing vs supernode approach

7.3.4 Discussion

While K-way hashing is simpler than supernodes, the probability of an access failure is greater than in supernode approach for same values of replication. The DHT maintenance overheads in K-way hashing during normal operation are none but reconstruction of the DHT is needed during reconfigurations. In Supernodes, reconstructions can be delayed and hence no overhead during reconfigurations but requires more overhead for normal DHT maintenance as each node needs to keep track about the connectivity information of k other nodes.

Shadowing v. Clustering: In Shadowing, a node has the flexibility to choose any other node as its shadow. This enables us to handle heterogeneity – by choosing more stable nodes as a shadows, by choosing more capable nodes as a shadows, etc., This is not possible in Clustering where each node can only backup $k-1$ other nodes.

8 Evaluation

We have implemented a prototype of SDIMS in Java using FreePastry framework [36] 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 [31] 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.

8.1 Simulation Experiments

Flexibility and Scalability A major innovation of our system is its ability to provide flexible computation and propagation of aggregates. In Figure 14, 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 by different attributes for a wide range of read-to-write ratios of the operations. We also simulate with other sizes of network 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

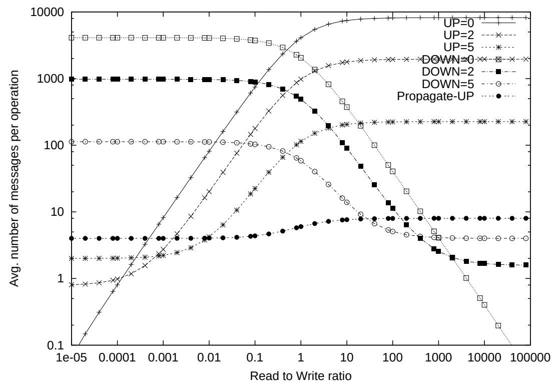


Figure 14: Flexibility of our approach. With different UP and DOWN values in a network of 4096 nodes for different read-write ratios.

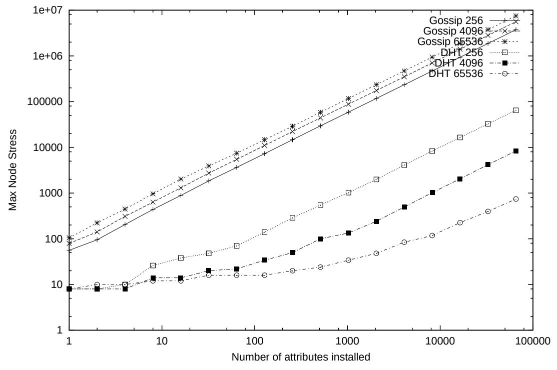


Figure 15: Max node stress for a gossiping approach vs. ADHT based approach for different number of nodes with increasing number of *sparse* attributes.

probe latencies increase their UP and DOWN propagation at a potential cost of increase in the overheads of writes.

Observe that the average number of messages per operation in case of the Update-UP strategy doubles when read-to-write ratio moves from write-dominated operations to read-dominated operations. In this strategy, probes incur twice the message cost in comparison to update cost as the responses for a probe back trace the forward path. While such backtracking aids in caching results at the intermediate nodes, optimizations are possible such that responses can be sent directly from the root node to the probing node.

Compared to existing Update-all single aggregation tree approaches [34], scalability in an 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 15. For this experiment, we build a simulator to simulate both Astrolabe [34] (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,

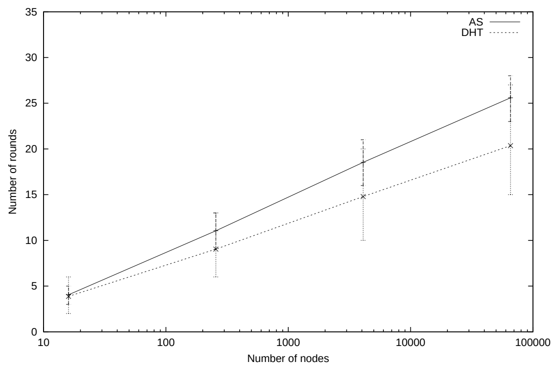


Figure 16: Number of rounds for which updates are not observed in a gossiping scheme(AS) vs ADHT based approach

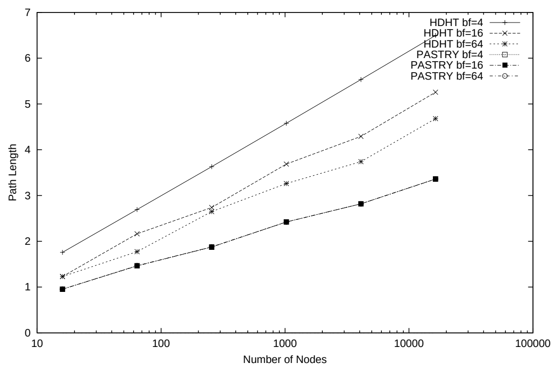


Figure 17: Average path length to root in Pastry versus ADHT for different branching factors.

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.

For the same experimental setup, Figure 16 plots the measured number of *rounds* [34] for which an update by a randomly chosen node is not observed by a probe of another randomly chosen node. In a *round* in ADHT, each node receives at most one message from its children and sends at most one message to its parent. This result shows that our approach performs similarly or better than the gossiping approach in terms of correctness or consistency of responses to the probes.

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 17, 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 18, 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 19, we plot the increase in path length to reach the global root node when node failures occur.

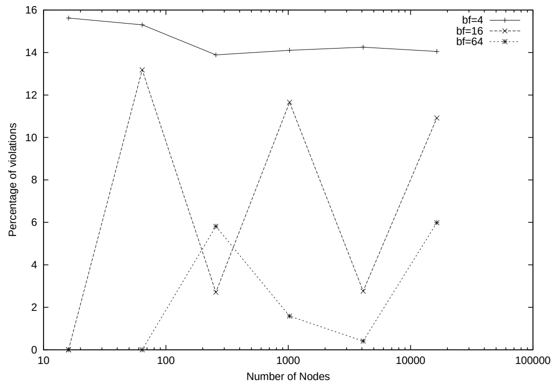


Figure 18: Percentage of probe pairs whose paths to the root did not conform to the path convergence property.

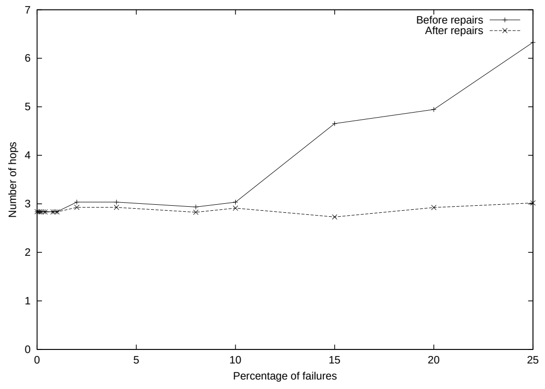


Figure 19: Increase in average path length due to failures compared to average path length after repair

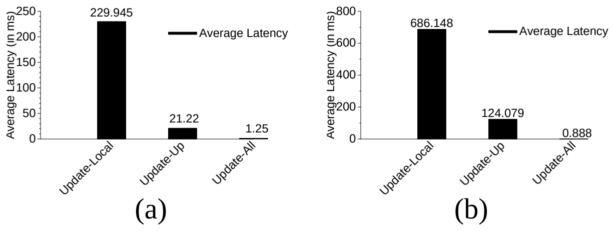


Figure 20: Average latency of probes for aggregate at global root level with three different modes of aggregate propagation on (a) 187 department machines, and (b) 69 Planet-Lab machines

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 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. Further study is required to define workloads appropriate to this system and analyze the impact of failures on the system.

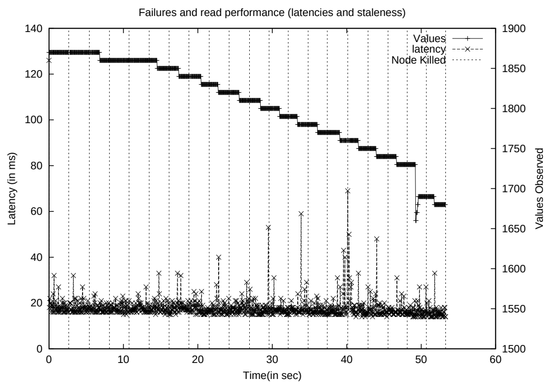


Figure 21: Micro-benchmark showing the behavior of the probes from a single node when failures are happening at some other nodes. All nodes assign a value of 10 to the attribute. This experiment is done on 187 department machines.

8.2 Testbed experiments

We run our prototype on 187 department machines and also on 69 machines of the Planet-Lab [31] 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 20 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 21 shows the results on the 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 22, 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 the 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 an SDIMS ensures path convergence with only a moderate increase in path lengths.

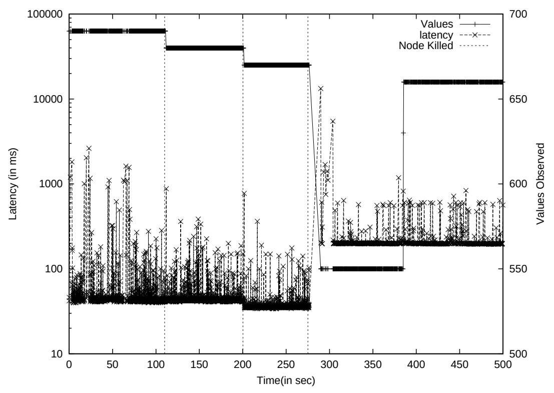


Figure 22: Probe performance during failures on 69 machines of Planet-Lab testbed

9 Related Work

The aggregation abstraction we use in our work is heavily influenced by the Astrolabe [34] project. Astrolabe adopts Propagate-All and unstructured gossiping techniques to attain robustness [2]. 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 [13, 24, 45] and commercial [44] 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 [13, 28] 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 [45] is a distributed monitoring system, currently deployed on Planet-Lab [31], 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 [24] 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 [32] 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 [36, 47] and upon other strategies [29, 33, 40]—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 [36], 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 [11] 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 [6, 7, 38, 42], file storage [10, 22, 37], and DNS [9] 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. [5] and Gummadi et al. [15] 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 an 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 18). SkipNet [17, 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 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 [19] and Gribble et al. [14]. 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*.

10 Dissertation Plan

The goal of this dissertation is to build a scalable distributed information system. Our current system is still in a nascent state but the initial work does provide evidence that we can achieve an SDIMS by leveraging Astrolabe's aggregation abstraction and DHTs. Our work also opens up many research issues in different fronts that need to be solved. Below we present some future research directions.

10.1 Extensions to Robustness

Supernodes In our current system, reconfigurations are costly. Though we can hide costs by lazy reaggregation schemes and enabling applications to control costs or enable them to adapt, reconfigurations are still expensive, particularly with machines fluctuating between reachable and unreachable state (machine ups and downs) being a common reconfiguration than nodes leaving permanently. We are currently exploring the Supernodes approach to alleviate these problems as explained in Section 7.

10.2 Extensions to Flexibility

Self-tuning adaptation The read-to-write ratios for applications are dynamic. An SDIMS system should adapt to changing read-to-write ratios by dynamically modifying the compuatation and propagation modes to conserve bandwidth while still conforming to the application specified propagation strategy.

Caching While caching is employed effectively in DHT file location applications, further research is needed to apply this concept in our general framework. Caching raises the issue of consistency and both of these topics need to be handled together.

Probe Functions Our current probe API allows applications to specify only levels for fetching aggregated values. While this interface is functionally enough, a more powerful interface would allow the applications to specify probe functions that decide the level of the aggregated values sought by evaluating the aggregated values as explained in Section 6.1.

Attribute feeding functions: Instead of a applications creating a new separate attribute for each aggregation function, we should have a way so that applications can specify single attribute and multiple aggregation functions that aggregate on that single attribute. Applications should be able to install some base case functions, which we call *attribute feeding functions*, that feed the reduction MIB from local attribute set.

10.3 Extensions to Scalability

Handling Composite Queries: Queries involving multiple attributes pose an issue in our system as different attributes are aggregated along different trees. We outline some techniques we are pursuing to tackle this issue in Section 4.4.

10.4 Timeline

Spring 2004 The crucial requirement for the success and the wide adoption of our system is to efficiently combat failures and reconfigurations. While current system implements different techniques, as mentioned in previous section and in Section 7 our design is far from efficient in this aspect. Hence, our top priority is to fully investigate the proposed approaches for robustness (Section 7) and build a robust SDIMS.

We also plan to build three applications mentioned in Section 6.1 on top of our SDIMS system and evaluate their performance on two real network testbeds – the department machines and the PlanetLab testbed. We estimate to complete above tasks by the end of this semester (Spring 2004) and submit a paper for the OSDI conference.

Fall 2004 & Spring 2005 We plan to implement dynamic self-tuning adaptation to reduce bandwidth, the capability to support probe functions and the attribute feeding functions in the next academic year 2004-2005. We also aim at investigating and building caching facility and ability to handle the composite queries in our system during the same time. The goal is to defend and submit the final thesis by the end of next Spring semester (May 2005).

11 Conclusions

We present 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 exposure of DHT's internal trees as an aggregation abstraction, (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.

References

[1] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

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

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

[4] R. Buyya. PARMON: a portable and scalable monitoring system for clusters. *Software — Practice and Experience*, 30(7):723–739, 2000.

[5] 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.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Multicast in a Cooperative Environment. In *Proceedings of the 19th ACM SOSP*, October 2003.

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

[8] 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.

[9] 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.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM SOSP*, October 2001.

[11] 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.

[12] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

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

[14] 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.

[15] 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.

[16] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. In *Proceedings of the 2nd International Workshop on Peer To Peer Systems (IPTPS)*, 2003.

[17] 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.

[18] N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Efficient Recovery From Organizational Disconnect in SkipNet. In *Proceeding of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 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 *Proceedings of the ACM Conference on Mobile Computing and Networking (MobiCom)*, pages 56–67, 2000.

[21] F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-Optimal Hash Table. In *Proceedings of the 2nd International Workshop on Peer To Peer Systems (IPTPS)*, 2003.

[22] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[23] X. Li and C. G. Plaxton. On Name Resolution in Peer-to-Peer Networks. In *Proceedings of the POMC*, October 2002.

[24] 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.

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

[26] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

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

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

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

[30] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM SOSP*, October 2001.

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

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

[40] 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.

[41] Supermon: High speed cluster monitoring. `http://www.acl.lanl.gov/supermon/`.

[42] 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.

[43] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1999.

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

[45] 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.

[46] 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.

[47] 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.