The Dissertation Committee for Praveen Yalagandula
certifies that this is the approved version of the following dissertation:

A Scalable Information Management Middleware for Large Distributed Systems

Committee:

Michael Dahlin, Supervisor

Lorenzo Alvisi

James C. Browne

C. Greg Plaxton

Robbert van Renesse

Harrick M. Vin

A Scalable Information Management Middleware for Large Distributed Systems

by

Praveen Yalagandula, B.Tech.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2005

Dedicated to my wife and my parents

Acknowledgments

I deem myself to be very fortunate to have Mike Dahlin as my advisor. I am really impressed by Mike's intuition, thoughtfulness, and quick comprehension capability. I am really obliged to Mike for guiding me in picking a very interesting and fruitful research direction. He was never demanding yet always encouraging natural growth. I greatly cherished his advice on research, coding, writing, reviewing, and time management and I drew a great inspiration and learned a lot from articles that he occasionally distributed to all his students like Turing award lectures, on career and general advice, and on programming standards.

I am thankful to Harrick Vin, Mike Dahlin, and Lorenzo Alvisi for creating a really collaborative and very inspiring environment by founding LASR lab. Their COPE seminar class made it easy for me to switch from hardware verification field to systems area. I also enjoyed several chats with Lorenzo Alvisi and learned a lot from his views on research ethics and discussions on interviewing faculty candidate's and visitor's lectures. I will never forget Lorenzo's Distributed Systems class lectures; they were both very entertaining and informative.

I am thankful to Greg Plaxton, James C. Browne, and Robbert van Renessee for serving on my dissertation committee. I greatly enjoyed several discussions with Greg on exploring the theoretical side of dynamic adaptation. I am impressed by Dr. Browne's extensive research experience in systems area and enjoyed working on range queries problem with him. I am thankful to Robbert for carefully studying

my entire thesis and for his extensive comments.

I really savored all my five years in the LASR lab, thanks to very intelligent and friendly colleagues. I have enjoyed my collaborations with Ravi Kokku, Arun Venkatarami, Sadia Sharif, Amit Garg, Mitul Tiwari, and Navendu Jain. I am also thankful to Suat Jain, Jayaram Mudigonda, Rama Krishna Rao Kotla, JP Martin, Amol Nayate, Taylor Riche, Jasleen Kaur Sahni, Sergey Gorinski, Taroon Mandhana, and Puneet Chopra for discussions on several ideas and for their help on fine tuning my presentations.

I would like to thank Adnan Aziz for arranging a teaching assistantship which enabled me to come to UT Austin for my graduate studies. I have thoroughly enjoyed several discussions with Adnan during my Masters in Computer engineering department. I immensely enjoyed discussions on research, food, and puzzles with Padmini Gopalakrishnan, Malay Ganai, Amit Prakash, Gaurav Rastogi, Tameen Khan, Shashank Gupta, and Rashmi Tripathi.

I am thankful to Palicherla Navin Reddy, Shailja Pathania, Anand Ramachandran (Randy), Vivekananda Vedula (Nandu), Mudanai P Sivakumar (MP), Madhukar Korupolu, Gauri Karve, and Sreekanth Samavedam (Freaky) for including me in a lot of entertaining activities. Colorado rockies trip with them was a very memorable experience for me.

I am thankful to my B.Tech. friends Debasis Mishra, Dhruba Chandra, Shailendra Jha, Vinit Srivastava, Mukul Khandelia, Vivek Gulati (gullu), Ranadeb Chaudhuri (rolly), Sunil Saini, Bibhudatta Sahoo, and Chandrasekhar Puthilathe (lathe) for keeping in touch with me and for their encouragement.

My parents have provided their constant support and encouragement throughout my life.

Finally, without the love and care of Tanjeet Juneja, my wife, this dissertation might not have been completed yet. Her constant encouragement and patience got

me through the low periods of my graduate life.

Praveen Yalagandula

The University of Texas at Austin
August 2005

A Scalable Information Management Middleware for Large Distributed Systems

Publication No. _____

Praveen Yalagandula, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Michael Dahlin

Information management is one of the key tasks of any large-scale distributed application. The goal of this dissertation is to design and build a general and scalable information management middleware for large distributed systems that will facilitate design, development, and deployment of distributed applications and that will enable application developers to explore the tradeoffs between communication cost, response latency, and consistency.

In this dissertation, we present a Scalable Distributed Information Management System (SDIMS) that *aggregates* information about large-scale networked systems and that can serve as a basic building block for a broad range of large-scale

distributed applications by providing detailed views of nearby information and summary views of global information. To serve as a basic building block, an SDIMS should have four properties: scalability to many machines and data items, flexibility to accommodate a broad range of applications, administrative isolation for security and availability, and robustness to node and network failures.

We design, implement, and evaluate an SDIMS that (1) leverages Distributed Hash Tables (DHT) to create scalable aggregation trees, (2) provides flexibility through a simple API that lets applications control propagation of reads and writes and through a self-tuning mechanism that adapts the propagation to observed load in the system, (3) provides administrative isolation through a novel Autonomous DHT algorithm, and (4) achieves robustness to node and network reconfigurations through lazy reaggregation, on-demand reaggregation, and tunable spatial replication.

Through extensive simulations and micro-benchmark experiments on several real testbeds, we observe that our system is an order of magnitude more scalable than existing approaches, provides a wide range of choices for applications to control the propagation of data to tradeoff the bandwidth cost with the response latency, achieves administrative isolation properties at a cost of modestly increased read latency in comparison to flat DHTs, and gracefully handles failures. We implement several applications on top of SDIMS including a file location system and a multicast system. We also use SDIMS in two other research efforts in our lab — as a controller for a distributed file replication system and as an information gathering plane in a distributed network monitoring system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Challenges

The goal of this dissertation 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 [21, 41, 84, 95, 99, 105], resource discovery [8, 30, 36], service placement [40, 108], data sharing and caching [47, 66, 80, 86, 94, 97, 118], sensor monitoring and control [50, 61], multicast tree formation [24, 25, 90, 101, 119], publish-subscribe systems [87, 102], and naming and request routing [26, 29]. 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 informa-

tion 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 [61, 101] allows nodes to access information they care about while maintaining system scalability.

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

Second, the system should have *flexibility* to accommodate a broad range of applications and attributes. For example, *read-dominated* attributes like *numCPUs* rarely change in value, whereas *write-dominated* attributes like *numProcesses* change quite often. An approach tuned for read-dominated attributes will consume high bandwidth when applied to write-dominated attributes. Conversely, an approach tuned for write-dominated attributes will suffer from unnecessary query latency or imprecision for read-dominated attributes. Therefore, an SDIMS should provide mechanisms to handle different types of attributes and either leave the policy decision of tuning replication to the applications or dynamically adapt between aggregation mechanisms by tracking the workload patterns.

Third, an SDIMS should provide *administrative isolation*. In a large system, it is natural to arrange nodes in an organizational or an administrative hierarchy (e.g., Figure 1.1). An SDIMS should support administrative isolation in which

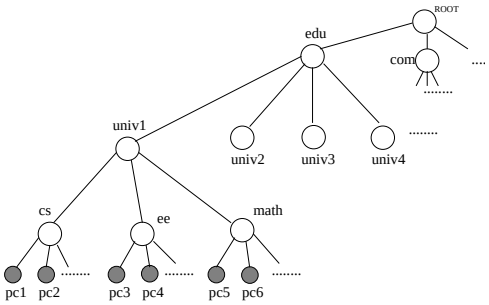


Figure 1.1: Administrative hierarchy

queries about an administrative domain's information can be satisfied within the domain for availability (e.g., so that the system can operate during disconnections from other domains), for security (e.g., so that an external observer cannot monitor or affect intra-domain queries), and for efficiency (e.g., to support domain-scoped queries efficiently).

Fourth, the system must be *robust* to node failures and disconnections. Reconfigurations in large distributed systems is a norm and an SDIMS should adapt to reconfigurations in a timely fashion and should also provide mechanisms so that applications can tradeoff the cost of adaptation with the consistency level in the aggregated results when reconfigurations occur.

We draw inspiration from two previous works: *Astrolabe* [101] and *Distributed Hash Tables (DHTs)*.

Astrolabe [101] is a robust information management system. Astrolabe provides the abstraction of a single logical aggregation tree that mirrors a system's administrative hierarchy. It provides a general interface for installing new aggregation functions and provides eventual consistency on its data. Astrolabe is robust due to its use of an unstructured gossip protocol for disseminating information and its strategy of replicating all aggregated attribute values for a subtree to all nodes in the subtree. This combination allows any communication pattern to yield even-

tual 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 or even balanced read-write attributes.

Recent research in peer-to-peer structured networks resulted in Distributed Hash Tables (DHTs) [6, 45, 47, 52, 59, 63, 64, 77, 80, 83, 86, 94, 118]—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 [77] 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 a basic system building block alongside the DHTs.

At a first glance, it might appear to be obvious that simply fusing DHTs with Astrolabe's aggregation abstraction will result in an SDIMS. However, meeting the SDIMS requirements forces a design to address four questions: (1) How to scalably map different attributes to different aggregation trees in a DHT mesh? (2) How to provide flexibility in the aggregation to accommodate different application requirements? (3) How to adapt a global, flat DHT mesh to attain administrative isolation property? and (4) How to provide robustness without unstructured gossip and total replication?

This dissertation answers the above questions by defining a new aggregation

abstraction that enable SDIMS scalably map different attributes to different aggregation trees, by exposing a flexible API that provide several aggregation mechanisms, by designing a novel Autonomous DHT (ADHT) algorithm that ensures administrative isolation, and by reaggregating data in the face of reconfigurations. We evaluate a prototype of SDIMS with a large number of simulation and testbed microbenchmark experiments. As case studies, we use SDIMS for building a controller for a distributed file replication system and for building a network monitoring system.

1.2 Contributions

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

1. We define a new aggregation abstraction that specifies both attribute type and attribute name and that associates an aggregation function with a particular attribute type. This abstraction paves the way for utilizing the DHT system's internal trees for aggregation and for achieving *scalability* with both nodes and attributes.

2. We provide a flexible API that lets applications control the propagation of reads and writes and thus tradeoff update cost, read latency, replication, and staleness. We also build Shruti, a sub-system in SDIMS, that tracks reads and writes in SDIMS and dynamically adapts the propagation to optimize communication costs.

3. We propose and build a novel DHT, autonomous DHT (ADHT), that augments an existing DHT algorithm to ensure *path convergence* and *path locality* properties in order to achieve *administrative isolation* requirement.

4. We provide *robustness* to node and network reconfigurations by (a) providing temporal replication through lazy reaggregation that guarantees eventual

consistency and (b) ensuring that our flexible API allows demanding applications achieve additional robustness by using tunable spatial replication of data aggregates or by performing fast on-demand reaggregation to augment the underlying lazy reaggregation or by exploiting both approaches.

We have built a prototype of SDIMS in Java using the FreePastry [39] framework. Through simulations and micro-benchmark experiments on a number of department machines, Emulab [107] nodes, and PlanetLab [76] nodes, we observe that the prototype achieves scalability with respect to both nodes and attributes through use of its flexible API, inflicts an order of magnitude lower maximum node stress than unstructured gossiping schemes, achieves administrative isolation properties at a cost of modestly increased read latencies compared to flat DHTs, and gracefully handles node failures.

This dissertation discusses key aspects of an ongoing system building effort, but it does not address all issues in building an SDIMS. For example, we believe that our strategies for providing robustness will mesh well with techniques such as *supernodes* [62] and other ongoing efforts to improve DHTs [81] for further improving robustness. Also, although splitting aggregation among many trees improves scalability for simple queries, this approach may make complex and multi-attribute queries more expensive compared to a single tree. Additional work is needed to understand the significance of this limitation for real workloads and, if necessary, to adapt query planning techniques from DHT abstractions [43, 49] to scalable aggregation tree abstractions.

1.3 Thesis Roadmap

In Chapter 2, we explain the hierarchical aggregation abstraction that SDIMS provides to applications. In Chapters 3, 4, and 5, we describe the design of our system for achieving the flexibility, scalability, and administrative isolation requirements of

an SDIMS. Chapter 6 addresses the issue of adaptation to the topological reconfigurations. In Chapter 7, we describe Shruti, a subsystem of SDIMS, that tracks workload patterns for an attribute and dynamically adapts aggregation strategy for that attribute. In Chapter 8, we detail the implementation of our prototype system and present the evaluation of our system through microbenchmarks on three real testbeds. We present couple of case studies using SDIMS for building the controller for a distributed file system and for a network monitoring system in Chapter 9. Chapter 10 details the related work, and Chapter 11 summarizes this dissertation.

Chapter 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 [101].

Our aggregation abstraction is defined across a tree spanning all nodes in the system. Each physical node in the system is a leaf and each subtree represents a logical group of nodes. Note that logical groups can correspond to administrative domains (e.g., department or university) or groups of nodes within a domain (e.g., 10 workstations on a LAN in CS department). An internal non-leaf node, which we call *virtual node*, is simulated by one or more physical nodes at the leaves of the subtree for which the virtual node is the root. We describe how to form such trees in Chapter 4.

Each physical leaf node has *local data* stored as a set of tuples in the following form:

$$(attributeType, attributeName, value).$$

Examples for such tuples are *(configuration, numCPUs, 16), (mcast membership, session foo, yes)*, and *(file stored, foo, myIPaddress)*. The system associates an *ag-*

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

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

Although SDIMS allows arbitrary aggregation functions, it is often desirable that these functions satisfy the *hierarchical computation* property [61]:

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

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 stores values as tuples of form $(sum, count)$, 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 while allowing applications to compute the average from the aggregate sum and count values.

Finally, note that for a large-scale system, it is difficult or impossible to insist that the aggregation value returned by a probe corresponds to the function computed over the current values at the leaves at the instant of the probe. Therefore our system

provides only weak consistency guarantees — specifically eventual consistency as defined in [101].

Chapter 3

Flexibility

For SDIMS to support a wide range of distributed applications, it should be flexible enough to allow applications to decide how data is aggregated in the system. In this chapter, we show how the aggregation abstraction we have defined in the previous chapter allows our system to provide a wide range of aggregation strategies and present the interfaces that SDIMS exposes to applications that applications can use specify their aggregation policy.

3.1 Motivation

The definition of the aggregation abstraction presented in Chapter 2 allows considerable flexibility in how, when, and where aggregate values are computed and propagated. Whereas previous systems [41, 80, 86, 94, 101, 118] implement a single static strategy, we argue that a SDIMS should provide *flexible computation and propagation* to efficiently support a wide variety of applications with diverse requirements. In order to provide this flexibility, we develop a simple interface that decomposes the aggregation abstraction into three pieces of functionality: install, update, and probe.

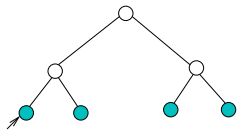
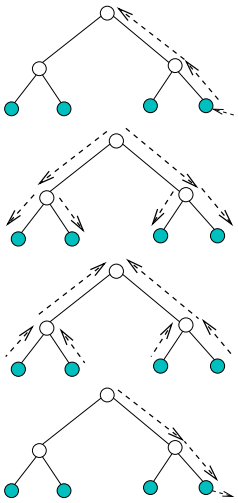
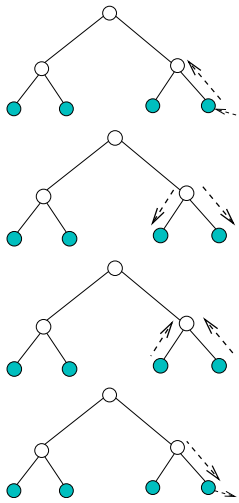
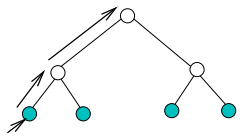
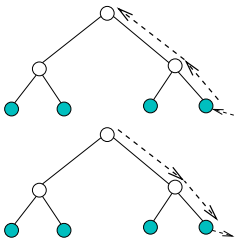
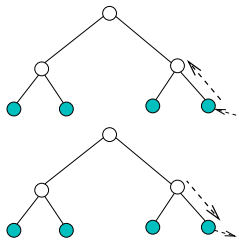
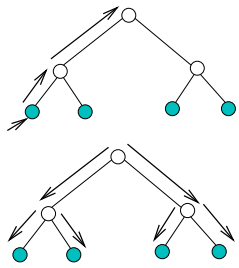
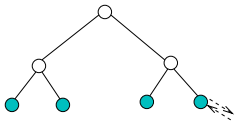
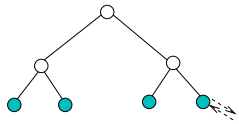
Update Strategy	On Update	On Probe for	
		Global Aggregate	Level-1 Aggregate
Update-Local			
Update-Up			
Update-All			

Figure 3.1: Flexible API

The definition of the aggregation abstraction allows our system to provide a continuous spectrum of strategies ranging from lazy aggregate computation and propagation on reads to aggressive immediate computation and propagation on

writes. In Figure 3.1, we illustrate both extreme strategies and an intermediate strategy. Under the lazy *Update-Local* computation and propagation strategy, an update (or write) only affects local state. Then, a probe (or read) that reads a level-i aggregate value is sent up the tree to the issuing node's level-i ancestor and then down the tree to the leaves. The system then computes the desired aggregate value at each layer up the tree until the 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* strategy [101], when an update occurs, changes are aggregated up the tree, and each new aggregate value is flooded to all of a node's descendants. In this case, each level-i node not only maintains the aggregate values for the level-i subtree but also receives and locally stores copies of all of its ancestors's level-j ($j > i$) aggregation values. Also, a leaf satisfies a probe for a level-i aggregate using purely local data. In an intermediate *Update-Up* strategy, the root of each subtree maintains the subtree's current aggregate value, and when an update occurs, the leaf node updates its local state and passes the update to its parent, and then each successive enclosing subtree updates its aggregate value and passes the new value to its parent. This strategy satisfies a leaf's probe for a level-i aggregate value by sending the probe up to the level-i ancestor of the leaf and then sending the aggregate value down to the leaf. Finally, notice that other strategies exist. For example, an *Update-UpRoot-Down1* strategy (not shown) would aggregate updates up to the root of a subtree and send a subtree's aggregate values to only the children of the root of the subtree. In general, an Update-Up`k`-Down`j` strategy aggregates up to the `k`th level and propagates the aggregate values of a node at level l (s.t. $l \leq$ `k`) downward for `j` levels.

A SDIMS must provide a wide range of flexible computation and propagation strategies to applications for it to be a general abstraction. Depending on its read-to-write ratio, an application should be able to use a particular mechanism

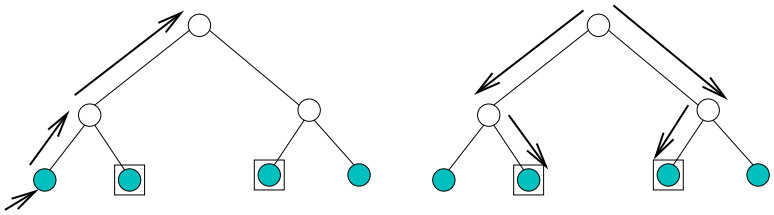


Figure 3.2: Spatial heterogeneity - On update, send the aggregate value to only interested nodes

to reduce the bandwidth consumption while attaining the required responsiveness and precision. Note that the read-to-write ratios of the attributes that applications install vary extensively. For example, a *read-dominated* attribute like *numCPUs* rarely changes in value, whereas 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.

SDIMS should also allow non-uniform propagation across the aggregation tree with different up and down parameters in different subtrees so that applications can adapt with the spatial and temporal heterogeneity of read and write operations. With respect to spatial heterogeneity, access patterns may differ for different parts of the tree, requiring different propagation strategies for different parts of the tree. An example case is shown in Figure 3.2, where only two nodes are interested in an attribute. Upon any updates, the changed aggregate values in this case are propagated to only those interested nodes. Similarly with respect to temporal heterogeneity, access patterns may change over time requiring different strategies over time.

parameter	description	optional
attrType	Attribute Type	No
aggrfunc	Aggregation Function	No
attrName	Attribute Name	Yes
`UP`	How far upward each update is sent (default: all)	Yes
`DOWN`	How far downward each aggregate is sent (default: none)	Yes
domain	Domain restriction (default: none)	Yes
expTime	Expiry Time	No

Table 3.1: Arguments for the install operation

3.2 Aggregation API

A major innovation of our work is enabling flexible aggregate computation and propagation. We provide the flexibility described in the previous section by splitting the aggregation API into three functions: *Install()* installs an aggregation function that defines an operation on an attribute type and specifies the update strategy that the function will use, *Update()* inserts or modifies a node's local value for an attribute, and *Probe()* obtains an aggregate value for a specified subtree. The install interface allows applications to specify the `k` and `j` parameters of the Update-Up`k`-Down`j` strategy along with the aggregation function. The update interface invokes the aggregation of an attribute on the tree according to corresponding aggregation function's aggregation strategy. The probe interface not only allows applications to obtain the aggregated value for a specified tree but also allows a probing node to *continuously* fetch the values for a specified time, thus enabling an application to adapt to spatial and temporal heterogeneity. The rest of the section describes these three interfaces in detail.

parameter	description	optional
attrType	Attribute Type	No
attrName	Attribute Name	No
val	Value	No

Table 3.2: Arguments for the update operation

3.2.1 Install

The *Install* operation installs an aggregation function in the system. The arguments for this operation are listed in Table 3.1. The *attrType* argument denotes the type of attributes on which this aggregation function is invoked. Installed functions are soft state that must be periodically renewed or they will be garbage collected at *expTime*.

The arguments *up* and *down* specify the aggregate computation and propagation strategy *Update-Up*`k`*-Down*`j`. The *domain* argument, if present, indicates that the aggregation function should be installed on all nodes in the specified domain; otherwise the function is installed on all nodes in the system.

3.2.2 Update

The *Update* operation takes three arguments *attrType, attrName,* and *value* and creates a new (attrType, attrName, value) tuple or updates the value of an old tuple with matching *attrType* and *attrName* at a leaf node. The arguments for the update operation are shown in Table 3.2.

The update interface meshes with the installed aggregate computation and propagation strategy to provide flexibility. In particular, as outlined above and described in detail in Chapter 8, after a leaf applies an update locally, the update may trigger re-computation of aggregate values up the tree and may also trigger propagation of changed aggregate values down the tree. Notice that our abstraction associates an installed aggregation function with only an *attrType* but lets updates

parameter	description	optional
attrType	Attribute Type	No
attrName	Attribute Name	No
mode	Continuous or One-shot (default: one-shot)	Yes
level	Level at which aggregate is sought (default: global root level)	Yes
domain	Domain restrictions (default: none)	Yes
`UP`	How far up to go and re-fetch the value (default: none)	Yes
`DOWN`	How far down to go and re-aggregate (default: none)	Yes
expTime	Expiry Time	No

Table 3.3: Arguments for the probe operation

specify an *attrName* along with the *attrType*. This technique helps achieve scalability with respect to nodes and attributes as described in Chapter 4.

3.2.3 Probe

The *Probe* operation returns the value of an attribute to an application. The complete argument set for the probe operation is shown in Table 3.3. Along with the *attrName* and the *attrType* arguments, a *level* argument specifies the level at which the answers are required for an attribute. In our implementation we choose to return results at all levels $k < l$ for a level-l probe because (i) it is inexpensive as the nodes traversed for level-l probe also contain level k aggregates for $k < l$ and as we expect the network cost of transmitting the additional information to be small for the small aggregates on which we focus and (ii) it is useful as applications can efficiently get several aggregates with a single probe (e.g., for domain-scoped queries as explained in Chapter 5).

Probes with *mode* set to *continuous* and with finite *expTime* enable applications to handle spatial and temporal heterogeneity. When node A issues a continuous probe at level l for an attribute, then regardless of the `UP` and `DOWN` parameters

specified in the install, updates for the attribute at any node in A's level-l ancestor's subtree are aggregated up to level l and the aggregated value is propagated down along the path from the ancestor to A. Note that continuous mode enables SDIMS to support a distributed sensor-actuator mechanism where a sensor monitors a level-i aggregate with a continuous mode probe and triggers an actuator upon receiving new values for the probe.

Similar to the *domain* argument in the *install* API, the *domain* argument in probes allows applications to restrict the probe to an administrative boundary in the aggregation tree. This argument facilitates the applications that need aggregates at different levels of administrative domain hierarchy. Without this argument, the applications need to know the exact level numbers that correspond to such domain boundaries in the aggregation trees.

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

3.3 Dynamic Adaptation

At the API level, the `UP` and `DOWN` arguments in install API can be regarded as hints, since they suggest a computation strategy but do not affect the semantics of an aggregation function. An SDIMS implementation can dynamically adjust the up/down strategies for an attribute based on the measured read/write frequency for that attribute. But a virtual intermediate node needs to know the current `UP` and `DOWN` propagation values to decide if the local aggregate is fresh in order to answer a probe. This is the key reason why `UP` and `DOWN` need to be statically defined at the install time and can not be specified in the update operation. This

in turn requires applications to know the read and write access patterns a priori to choose an appropriate strategy. We build Shruti, a system for dynamically adapting aggregation strategy based on the observed read and write rates for an attribute. We describe Shruti in detail in Chapter 7.

Chapter 4

Scalability

Our system accommodates a large number of participating nodes, and it allows applications to install and monitor a large number of data attributes. Our design achieves 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) designing and building novel Autonomous DHTs to conform to the required administrative isolation 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 [101]. Figure 1.1 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 hosting intermediate virtual nodes corresponding to domains with large number of machines will have a large number of children and hence incur high communication costs. For example, we plot the name distribution within the `utexas.edu` domain in Figure 4.1. Observe that there are a significant number of domains with large numbers of participants and a large number of domain with one or very few participants. 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 hosting the root node would incur a storage cost linear with the number of attributes and 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 following four key ideas.

First, whereas previous distributed information management systems like Astrolabe [101] and Ganglia [41] 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

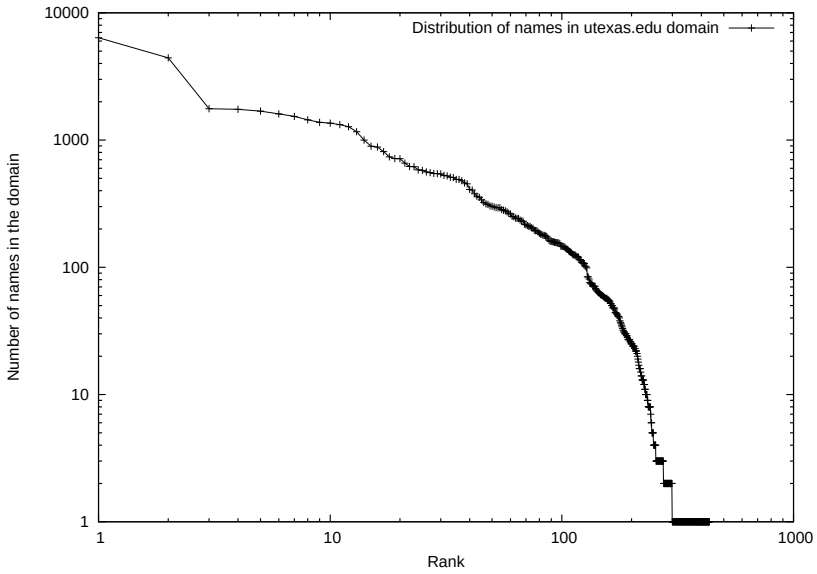


Figure 4.1: The name distribution in utexas.edu domain

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 [12, 41, 101, 102, 116], 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, file)* 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 the scalability can be improved by representing such sets with Bloom filters [17]. 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 a novel Autonomous DHT (ADHT) to ensure the required administrative isolation properties. Though existing DHTs offer solution for scalability with the nodes and attributes, they do not guarantee that the administrative isolation is preserved in the aggregation trees. We will discuss this aspect in detail in Chapter 5.

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

4.3 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 that is *closest* to the key. 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 building aggregation trees. Remember that aggregation trees have physical machines as the leaves of the tree and each intermediate virtual node corresponds to a grouping of machines in the system (refer to Chapter 2). A $DHTtree_k$ can be viewed as an aggregation tree as follows — A physical machine A with ID ID_A hosts m virtual levels where m is the number of common prefix bits between ID_A and the key k. The i-th level virtual node in the aggregation tree has the following children — local $(i-1)$-th level virtual node and all other machines in the DHT with IDs that match k in $(i-1)$ prefix bits and use node A to correct i-th bit. Below, we present a simple example to illustrate this process.

Example The pointers maintained at nodes in a 8-node DHT with 3-bit address space are tabulated in Table 4.1. The DHT trees for key IDs 111 and 000 are shown in Figures 4.2 and 4.3 respectively along with the corresponding aggregation tree they represent. The figures also illustrate which physical nodes host the which 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 4.1: Example pointer table for a DHT comprising of 8 nodes and addresses drawn from a 3-bit ID space

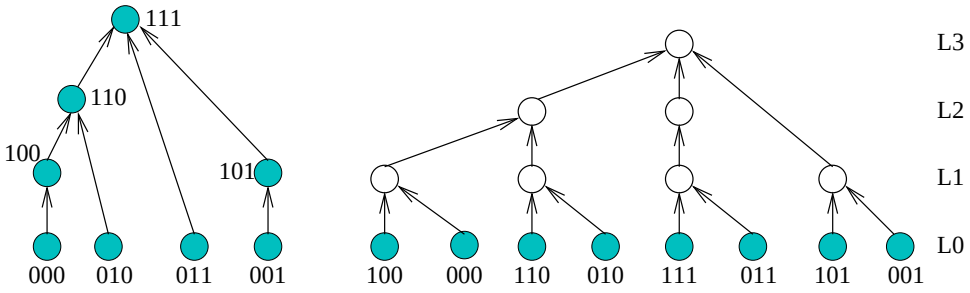


Figure 4.2: The DHT tree corresponding to ID 111 ($DHTtree_{111}$) and the corresponding aggregation tree.

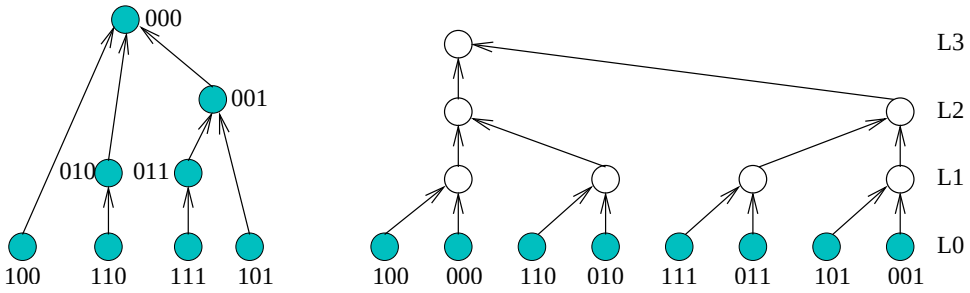


Figure 4.3: The DHT tree corresponding to ID 000 ($DHTtree_{000}$) and the corresponding aggregation tree.

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

Table 4.2: Example pointer table for a DHT comprising of six nodes and addresses drawn from a 3-bit ID space (missing nodes with IDs 111 and 110 compared to the DHT shown in Table 4.1)

In the simple example above, on all paths for a key, each step corrects at least one bit of the key. But in practical scenarios this may not happen. Consider a similar scenario as in the above example but suppose nodes with IDs 110 and 111 are missing. A DHT pointer table for this six node network is shown in table 4.2. The DHTtree for key k=111 is shown in Figure 4.4. Since node with ID 101 is the closest to key k=111, that node is chosen as the root for that ID. But the aggregation tree defined as in the previous paragraphs would yield a disconnected aggregation tree in this case (shown in Figure 4.5). To handle such case, we introduce an extra virtual node at machines that has same level children. The aggregation tree with such an extra virtual node is shown in Figure 4.6. Note that this extra virtual node is considered to be at the same level as the machine's maximum matching bits with key in consideration. We will later explain how installs, updates, and probes handle these extra virtual nodes in Chapter 8 on prototype implementation.

By aggregating an attribute along the aggregation tree corresponding to a $DHTtree_k$ 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: where as in a single aggregation tree approach, the root and the intermediate nodes pass around more messages than the leaf nodes, in

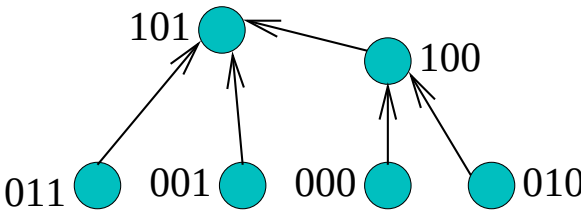


Figure 4.4: The DHT tree corresponding to ID 111 ($DHTtree_{111}$) in a six node network corresponding to the case shown in Table 4.2.

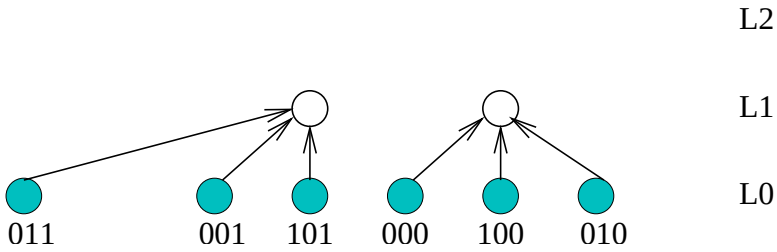


Figure 4.5: The disconnected incorrect aggregation tree corresponding to DHTtree_{111} shown in Figure 4.4.

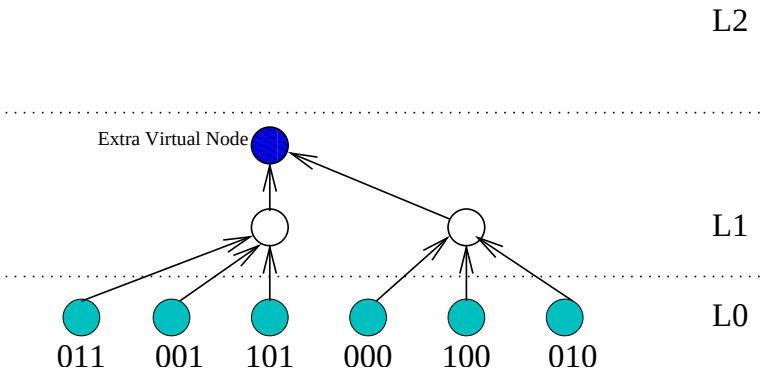


Figure 4.6: The correct aggregation tree corresponding to DHTtree_{111} shown in Figure 4.4 with an extra virtual node.

a DHT-based multi-tree, each node acts as intermediate aggregation point for some attributes and as leaf node for other attributes. Hence, this approach distributes the onus of aggregation across all nodes.

4.3.1 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 an average m/N fraction of attributes, where m is the number of attributes in the system of N nodes. But each node acts as an intermediate point of aggregation for many other attributes. Here we estimate the fraction of attributes for which a node has to perform aggregation assuming that the correction is done bit-by-bit in DHT routing.

Let the density factor of the attributes be d ($0 <= d <= 1$) i.e., 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 other nodes that route to this node for each prefix of its node ID to correct the last bit of the prefix. For a machine A, let c_i be the number of machines from which the routes for keys starting with i length prefix of machine A's ID reach node A after correcting i bits. These will be the set of leafnodes in the subtree rooted at level-i virtual node on node A in the aggregation tree corresponding to a key that matches ID of node A in at least the first i bits. The average fraction of attributes that children at level i sends to node A is dependent on the number of nodes in the subtree rooted at level i, which will be $\frac{1-(1-d)^{c_i}}{2^i}$. Hence, the average fraction of all attributes f for which node A has to perform aggregation is given by the following equation:

$$f = \left[1 - \prod_{i=0}^{i=\#bits} \left(1 - \frac{1-(1-d)^{c_i}}{2^i}\right)\right] \tag{4.1}$$

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

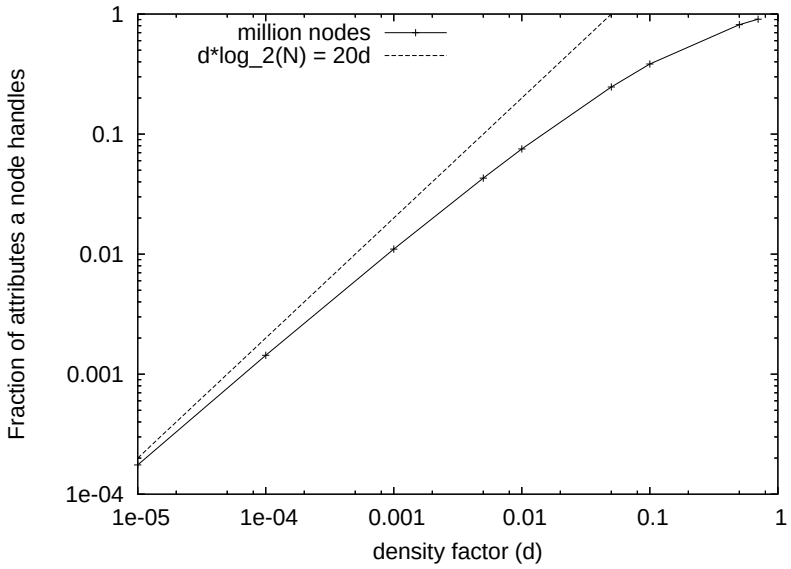


Figure 4.7: Average fraction of attributes which a node stores and on which it performs aggregations in a million node network

$$f = \left[1 - \prod_{i=0}^{i=\#bits} \left(1 - \frac{c_i d}{2^i}\right)\right] \tag{4.2}$$

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

$$f = \left[1 - \prod_{i=0}^{i=\#bits} (d)\right] = O(d \log_2(N)) \tag{4.3}$$

We plot the Equation 4.1 for a million node network (#bits = 20) and with approximation of $c_i = 2^i$ in Figure 4.7 along with the approximation $O(d \log_2(N))$. At low density factors, the fraction of attributes for which a node performs aggregation will be quite small. Hence, the scalability in SDIMS with the number of attributes.

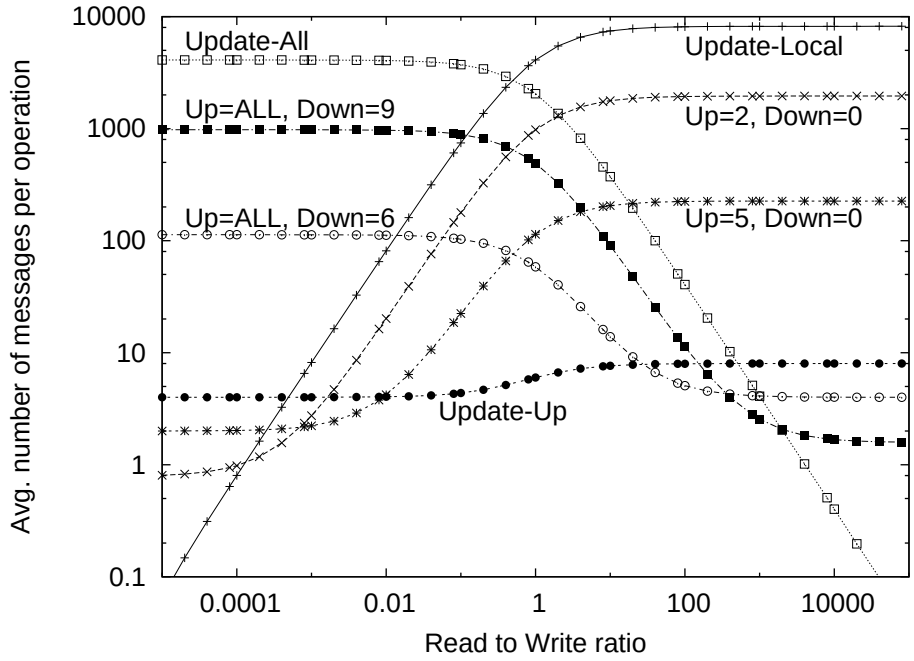


Figure 4.8: Flexibility of our approach. With different `UP` and `DOWN` values in a network of 4096 nodes for different read-write ratios.

4.4 Simulation Experiments

We have implemented an SDIMS system in Java using the FreePastry framework [86] and performed large-scale simulation experiments. Our evaluation here supports two main conclusions. First, our flexible API provides different propagation strategies that minimize communication resources at different read-to-write ratios. For example, in our simulation we observe Update-Local to be efficient for read-to-write ratios below 0.0001, Update-Up around 1, and Update-All above 50000. Second, our system is scalable with respect to both nodes and attributes. In particular, we find that the maximum node stress in our system is an order smaller than observed with an Update-All, gossiping approach.

4.4.1 Flexibility

A major innovation of our system is its ability to provide flexible computation and propagation of aggregates. In Figure 4.8, we demonstrate the flexibility exposed by the aggregation API explained in Section 3. We simulate a system with 4096 nodes and install several attributes with different `UP` and `DOWN` parameters. We plot the average number of messages per operation incurred for a wide range of read-to-write ratios of the operations for different attributes. This graph clearly demonstrates the benefit of supporting a wide range of computation and propagation strategies. Although having a small UP value is efficient for attributes with low read-to-write ratios (write dominated applications), the probe latency, when reads do occur, may be high since the probe needs to aggregate the data from all the nodes that did not send their aggregate up. Conversely, applications that wish to improve probe overheads or latencies can increase their UP and DOWN propagation at a potential cost of increase in write overheads.

4.4.2 Scalability

Compared to an existing Update-all single aggregation tree approach [101], scalability in SDIMS comes from (1) leveraging DHTs to form multiple aggregation trees that split the load across nodes and (2) flexible propagation that avoids propagation of all updates to all nodes. Figure 4.9 demonstrates the SDIMS's scalability with nodes and attributes. For this experiment, we build a simulator to simulate both Astrolabe [101] (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 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

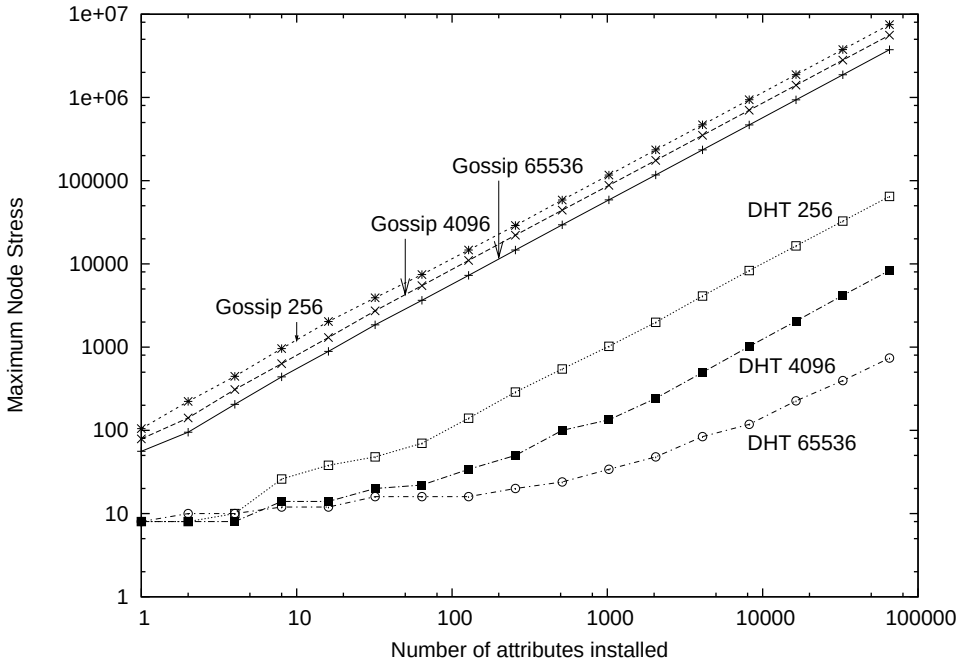


Figure 4.9: Max node stress for a gossiping approach vs. DHT based SDIMS approach for different number of nodes with increasing number of *sparse* attributes.

different sized networks with increasing number of sessions when the participant of each session performs an update operation. Clearly, the DHT based scheme is more scalable with respect to attributes than an Update-all gossiping scheme. Observe that at some constant number of attributes, the maximum node stress increases in the gossiping approach as the number of nodes increase in the system; but, in our approach, it decreases because the load of aggregation is spread across more nodes. Simulations with other session sizes (4 and 16) yield similar results.

4.5 An Open Issue: Handling Composite Queries

Though 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 of distributing attributes across trees. Ongoing efforts by other researchers to provide the relational database abstraction on DHTs — PIER [49] and Gribble et al. [43] — 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. Further investigation is necessary to evaluate the feasibility and performance of these different strategies.

Chapter 5

Administrative Isolation

5.1 Introduction

Though DHTs offer solution for scalability with the nodes and attributes, they do not guarantee that administrative isolation is preserved in the aggregation trees. Having aggregation trees that conform to the administrative hierarchy helps SDIMS provide important autonomy, security, and isolation properties [101]. 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 administrative domain[1] cannot observe or affect system behavior in another domain for computations relating only to the second domain.

We present two properties — Path Locality and Path Convergence — that a DHT routing layer should satisfy to guarantee that the aggregation trees exposed by the DHT routing conforms to the administrative isolation requirement. We present a novel Autonomous DHT (ADHT) that builds upon an existing DHT algorithm,

[1]Domain in our system is defined as a set of machines either administered by a common authority or a logical group with in such sets such set of machines sharing a switch. Note that these domains does not necessarily correspond to the DNS domain hierarchy even though we use a similar notation.

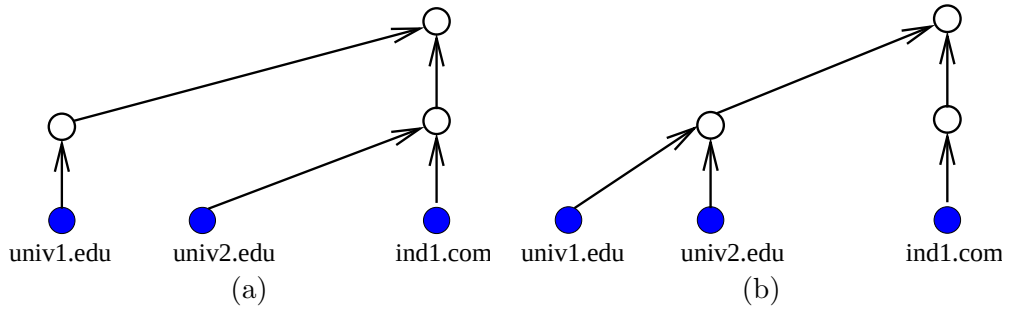


Figure 5.1: Example aggregation trees for a system with three nodes. Machines `univ1.edu` and `univ2.edu` are in one administrative domain and machine `ind1.com` is in a different administrative domain (a) An aggregation tree that does not satisfy the administrative isolation property (b) An aggregation tree that satisfies the isolation property.

Pastry, and guarantees these properties.

5.1.1 Motivation and Challenges

Administrative isolation requirement is important in SDIMS for three reasons: (i) for security — so that updates and probes flowing in a domain are inaccessible outside the domain, (ii) for availability — so that queries for values in a domain are unaffected by failures of nodes in other domains or by network disconnections of a domain from other domains, and (iii) for efficiency — so that domain-scoped queries can be simple and efficient.

To satisfy the administrative isolation requirement, an aggregation tree should satisfy the following property.

Property 1 *For each administrative domain, the virtual node in the aggregation tree at the root of the smallest subtree containing all nodes of the domain is hosted on a node in that domain.*

Consider the two aggregation trees in Figure 5.1. The aggregation tree in Figure 5.1(a) does not satisfy the above property — the virtual node that is at

the root of the smallest subtree corresponding to nodes in domain `edu` is hosted on a machine outside that domain. The second aggregation tree in Figure 5.1(b) satisfies the isolation requirement. In the following paragraphs, we further explain why satisfying the Property 1 in the aggregation tree is important for security, availability, and efficiency.

Security Security is an important issue for an aggregation system particularly in a large distributed system consisting of multiple administrative domains. Consider the example of using an aggregation framework to run a file location system in an enterprise. The file location information from cluster of machines belonging to the board of directors should be invisible to machines outside that cluster. Even though particular file names can be masked by using encoded names instead of plain names, the system should not expose even update and probe patterns outside that cluster as those patterns can reveal information about files on those machines or behavior of individuals. Such security can only be assured if the common aggregation point is within the domain of machines corresponding to the board of directors.

Availability Another key reason to have the common aggregation point for a domain to be hosted on a machine in that same domain is to ensure high availability. Note that a domain in our system corresponds to set of machines under a single administrative authority or corresponds to logical grouping of nodes with in such administrative domains like nodes sharing a switch, etc. Such domain are prone to domain disconnections when the connecting router fails or the subnetwork fails due to ISP failure, etc. Also a node in a domain can behave maliciously either by responding lazily for messages from nodes outside the domain or by dropping messages from nodes outside the domain. For a domain, such domain disconnections or malicious behavior of a node in another domain can potentially decrease the availability for operations within the domain. For example, since node `ind1.com`

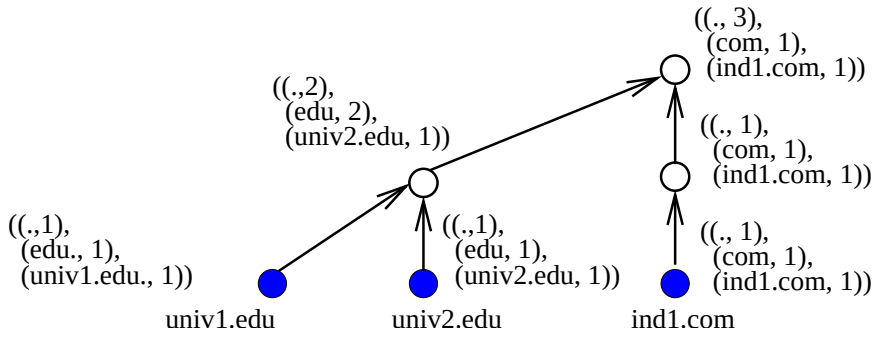


Figure 5.2: Example for the domain-scoped queries

hosts the common aggregation point for nodes in `.edu` domain in Figure 5.1(a), it can decrease the availability for nodes in `.edu` domain even for operations corresponding to aggregate values for `.edu` domain.

Efficiency Aggregation trees that provide administrative isolation also enable the definition of simple and efficient domain-scoped aggregation functions to support queries like "what is the average load on machines in domain X?" For example, consider an aggregation function to count the number of machines in our running example system with three machines illustrated in Figure 5.2. Each leaf node l updates attribute *NumMachines* with a value v_l containing a set of tuples of form (Domain, Count) for each domain of which the node is a part. In the example, the node univ1 with name `univ1.edu` performs an update with the value ((`univ1.edu.`,1),(`edu.`,1),(`.`,1)). An aggregation function at an internal virtual node hosted on node N with child set C computes the aggregate as a set of tuples: for each domain D that N is part of, form a tuple $(D, \sum_{c \in C}(count|(D, count) \in v_c))$. This computation is illustrated in the Figure 5.2. Now a query for *NumMachines* with *level* set to MAX will return the aggregate values at each intermediate virtual node on the path to the root as a set of tuples (tree level, aggregated value) from which it is easy to extract the count of machines at each enclosing

domain. For example, univ1 would receive ((2, ((`ind1.com.`,1),(`com.`,1),(.,3))), (1, ((`univ2.edu.`,1),(`edu.`,2),(.,2))), (0, ((`univ1.edu.`,1),(`edu.`,1),(.,1)))).

Note that it is possible to support domain-scoped queries even if the aggregation trees does not satisfy the Property 1. But supporting such queries would be less efficient and less convenient when aggregation trees do not conform to the property. It would be less efficient because each intermediate virtual node would have to maintain a list of all values at the leaves in its subtree along with their names and it would be less convenient as applications that need an aggregate for a domain would have to pick values of nodes in the domain from the list returned by a probe and invoke aggregation function on those values.

5.1.2 Our Approach

We achieve administrative isolation in SDIMS through the following two ideas. First, we allow applications to specify restrictions on visibility of attributes to a particular administrative domain. In the install and probe API (shown in Tables 3.1 and 3.3 in Chapter 3), we allow applications to specify a domain parameter that SDIMS uses to restrict the propagation of updates and probes to a specified domain. Second, we build aggregation trees ensuring that they satisfy administrative domain boundaries. These two ideas in combination allow applications to perform updates and probes in a domain while not exposing any information about those operations to nodes outside the domain.

Note that we extract aggregation trees from the DHT routing infrastructure. To ensure that such aggregation trees conform to administrative isolation requirement, the DHT routing should satisfy two properties:

1. Path Locality: Routing paths should always be contained in the smallest possible domain.

2. Path Convergence: Routing paths for a key from two different nodes in a

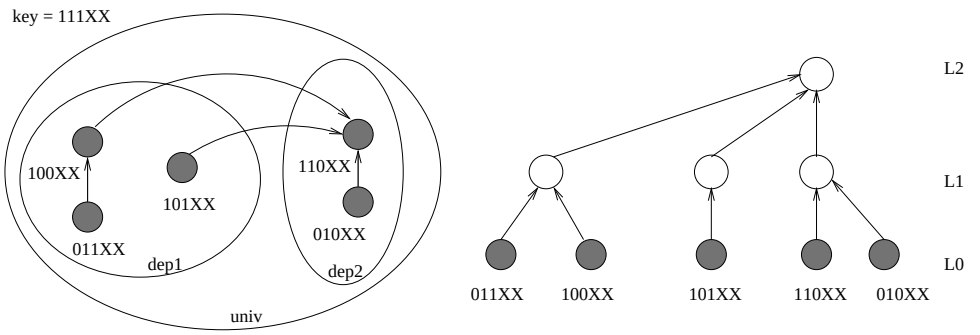


Figure 5.3: Example shows how isolation property is violated in the original Pastry. We also show the corresponding aggregation tree.

domain should converge at a node in the same domain.

Existing DHTs either already support path locality [47] or can support easily by setting the domain nearness as the distance metric [23, 44]. But they do not *guarantee* path convergence as those systems try to optimize the path length to the root to reduce response latency.

In the following section we explain how an existing DHT, Pastry [86], does not satisfy path convergence. Then we describe the design of our ADHT algorithm that builds upon the Pastry algorithm in Section 5.3. 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 locality and path convergence properties. In Section 5.4, we discuss the correctness and performance properties of ADHT. We present some experimental results quantifying the usefulness of ADHT algorithm in Section 5.5. In Section 5.6, we detail the related work and summarize the chapter in Section 5.7.

5.2 Background: Pastry

In Pastry [86], 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* [77] 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 unknown, 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 5.3, Pastry does not satisfy the ad-

ministrative isolation property 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 routing paths might not satisfy the required path locality and convergence properties. The second problem can be addressed by using domain-nearness as the proximity metric — any two nodes that match in i levels of domain hierarchy are considered closer than two nodes that match in fewer than i levels. However, this solution does not eliminate the first problem.

5.3 Our Approach

To ensure isolation properties in aggregation trees, a DHT must provide a single exit point in each domain for a key. Also the DHT routing protocol should route keys along intra-domain paths before routing them along inter-domain paths.

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

1. Data structures: Instead of one leafset in the Pastry algorithm, nodes in ADHT maintain a separate leafset for each administrative hierarchy domain to which a node belongs. We specify a node's position in the administrative hierarchy using similar notation as the Domain Name Service (DNS) [68].

2. Routing algorithm: ADHT uses a novel routing algorithm that ensures that the routing path for a key reaches the root node in a domain before it jumps out of the domain; thus, achieving path convergence and path locality properties. ADHT also uses a novel key space assignment to nodes so that routing paths do not visit a node twice during routing for any key — a property required so that we can extract aggregation trees from the routing structure. We also introduce a two level locality model that incorporates both administrative

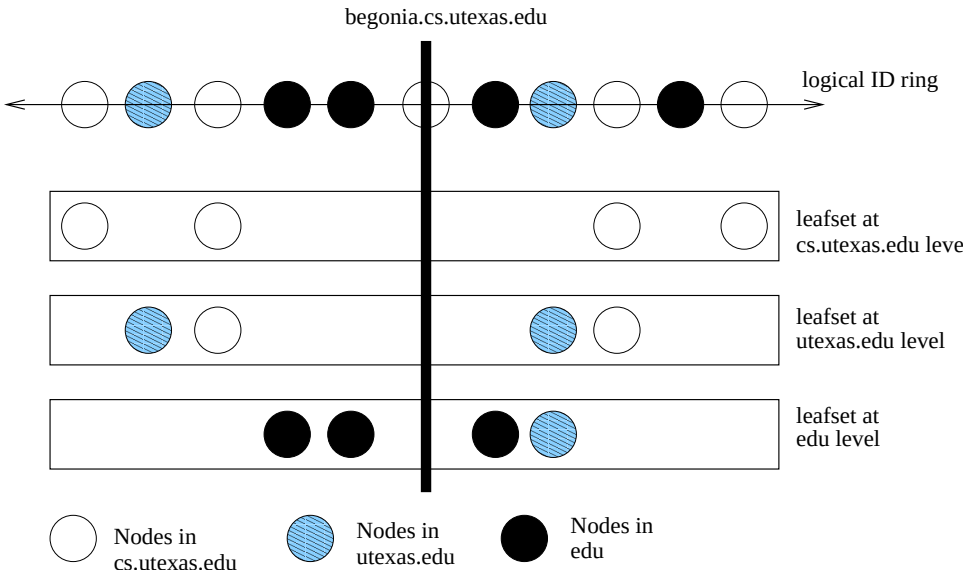


Figure 5.4: Multiple leafsets maintained by the node `begonia.cs.utexas.edu`. We assume a leafset size of four in this example.

membership of nodes and network distances between nodes.

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

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

In the following sections, we describe our ADHT algorithm in detail, mainly focusing on the four points mentioned above. At the end of this section, we present how we extract aggregation trees that satisfy Property 1 from the ADHT routing infrastructure.

[2]Terminology from [102]

5.3.1 Data Structures

Similar to Pastry and other DHT algorithms, each node in ADHT has a routing table to maintain pointers to other nodes that correct prefix bits of that node's nodeId. In contrast to Pastry where each node maintains a single leafset, each node in ADHT maintains a separate leaf set for each domain to which the node belongs. In Figure 5.4, we illustrate leafsets maintained by a node `begonia.cs.utexas.edu` in the ADHT algorithm. Note that the Pastry algorithm maintains just one leafset corresponding to the top domain level. For the example in figure, Pastry maintains only the leafset corresponding to `edu` level. Maintaining a different leafset for each level increases the number of neighbors that each node tracks to $(2^b) * \lg_b n + c.l$ in ADHT compared to $(2^b) * \lg_b n + c$ 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. But these extra leafsets ensure path locality and convergence properties during routing.

5.3.2 Routing in ADHT

The algorithm for populating the routing table is quite similar to Pastry but with the following key difference: ADHT uses hierarchical domain proximity as the primary proximity metric (two nodes that match in i levels of domain hierarchy are more proximate than two nodes that match in fewer than i levels in domain hierarchy) and network distance as the secondary proximity metric (if two pairs of nodes match in the same number of domain levels, then the pair whose separation by network distance is smaller is considered more proximate).

Key space assignment to nodes

In Pastry, a key k is assigned to a node A that is closest to the key on the logical ID ring. The distance on the logical ring between ID_A and the key k is defined as

$$MIN(|k - ID_A|, 2^b - |k - ID_A|),$$

where b is the number of bits used for IDs and keys. For example, the key space split is shown on the outer side of the ring in Figure 5.5 with dark lines for split among four nodes A, B, C and D.

In ADHT, a key k is assigned to a node A such that ID_A matches more prefix bits of k than any other node's ID. If IDs of multiple nodes match key k in the same number of prefix bits, then we pick a node B from that set such that $|k - ID_B|$ is smaller than difference between key k and any other node's ID. The key space split shown on the inner side of the ring in Figure 5.5 with dotted lines is the split among four nodes A, B, C, and D in case of the ADHT.

We will later describe in this section on how this different key space assignment allows us to construct aggregation trees that satisfy Property 1.

Routing Algorithm

The routing algorithm we use in routing for a key at node with *nodeId* is shown in Algorithm 1. To route a key k, a node A with ID ID_A first checks its routing table for another node that matches the key in more digits than this node. We call such bit correcting neighbor a *flipNeighbor*. If no such node exists, then we consider leafsets starting from the smallest domain. If a *flipNeighbor* exists and is in the node's lowest domain, then we route the key to the *flipNeighbor*. If a *flipNeighbor* exists and is not in the same domain as the node, then we consider leafsets corresponding to the levels below the common domain between the *flipNeighbor* and this node, starting from

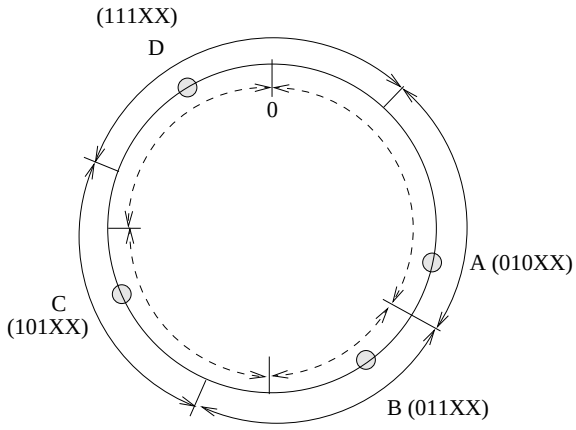


Figure 5.5: Key space assignment to nodes in Pastry and ADHT. The split shown on the outer side of the logical ring correspond to assignment in Pastry and the inner side corresponds to the assignment in ADHT.

the lowest domain leafset. For example, if a node `begonia.cs.utexas.edu` finds a *flipNeighbor* `linux1.cs.cmu.edu`, then the node considers the leafsets at levels `cs.utexas.edu` and `utexas.edu` in that order. If the node finds another node in its leafset that is closer to the key than the node, then it forwards the key to that node closer to the key. If no such node is found in a leafset at a level, then this node is considered the *root* node for key k in that domain. If a node has no *flipNeighbor* for a key k and has no neighbor in any leafset at any level that is closer to the key k than it is, then such node is the global root for key k. Note that by routing at the lowest possible domain until the root of that domain is reached, we ensure that all routing paths starting in a domain converge within that domain, thus achieving the Path Convergence property.

Discussion

The difference in key space assignment between Pastry and ADHT allows us to construct aggregation trees that satisfy administrative isolation requirements. This

Algorithm 1 ADHTroute(key)

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

difference ensures that no node is visited twice during routing. If we use the Pastry key assignment, then the routing paths using the ADHT routing algorithm might touch a node more than once. For example, consider routing for a key 1000XX in a domain comprising only two nodes with IDs 0111XX and 1110XX. In this domain, node with ID 0111XX is considered as the root for that domain. But, when routing in the next domain level from node 0111XX, the next hop goes to node 1110XX as it is the nearest bit-correcting node. Thus node 1110XX will be visited twice. Note that we can not form hierarchical aggregation trees from such an overlay routing algorithm where the path for a key routes to a node, leaves the node and routes back to the node. With ADHT key assignment, node 1110XX is assigned as the root in the lower domain and hence that node is not touched more than once.

5.3.3 Join Algorithm

Like any peer-to-peer algorithm, a node joins an existing ADHT by contacting one or more user supplied nodes. The join algorithm in ADHT differs from Pastry join algorithm primarily in choosing an appropriate bootstrap node and filling leafset table entries. Note that, in contrast to nodes in Pastry that have a single leafset table, nodes in ADHT maintain several leafset tables. In the following section, we first briefly explain the join algorithm used in Pastry [86] and then we present an ADHT join algorithm.

Pastry Join Algorithm Suppose node A with ID ID_A is joining using a bootstrap node B with ID ID_B. Node A asks node B to route a special *join* message with the key equal to ID_A. Like any message, Pastry routes the join message to the existing node R whose id ID_R is numerically closest to ID_A. In response to receiving the *join* request, nodes A, R, and all nodes encountered on the path from A to R send their routing tables to A. Also, node R sends its leafset table to node A which node A uses as its own leafset table. The new node A inspects this information and initializes its own routing tables. Finally, node A informs all nodes that need to be aware of its arrival by sending its routing or leafset table entries. This procedure ensures that node A initializes its state with appropriate values, and that the state in all other affected nodes is updated.

ADHT Join Algorithm Similar to Pastry's join algorithm, a node A wishing to join ADHT routes a special *join* message with target key set to ID_A from a given bootstrap node B. We modify the Pastry's join algorithm in the following ways so that the joining node in ADHT can fill its multiple leafsets. First, given a random contact node, the joining node searches for an appropriate bootstrap node B that is a closer node to the joining node in terms of the domain-nearness metric (explained in detail in the following section) than any other node in the system. Second, each

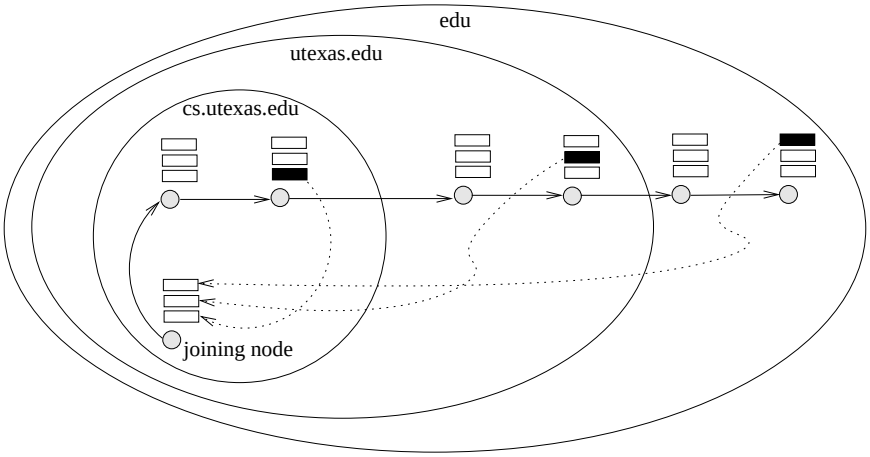


Figure 5.6: An example illustrating the leafsets that a joining node receives in response to its join request in a three level deep domain hierarchy. The dark arrows denote the path taken by the join request.

intermediate node C, on the routing path from node B to the current root node for ID_A, sends its leafsets for each common domain between C and A and for which node C is the root node in that domain for key ID_A (also implies that the next node on the routing path for ID_A from C does not belong to the domain). Figure 5.6 illustrates the leafsets that a joining node receives in response to its join request in a three level deep domain hierarchy case.

Finding a bootstrap node

A node that wishes to join an existing ADHT needs to find a node already in the ADHT that is closer to the joining node in terms of domain-nearness. For example, node `begonia.cs.utexas.edu` that wishes to join a ADHT searches for some other node in `cs.utexas.edu` domain. If no such node exists, then it looks for a node in `utexas.edu` domain, and so on. The joining node uses such a near node as its bootstrap node for joining the ADHT.

In the following, we present different ways in which a joining node searches

for an appropriate bootstrap node.

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

- **Using DHT `put` and `get` operations:** Joining nodes can use the already existing DHT to locate appropriate bootstrap nodes. Each node after joining the DHT will store information about the domains to which it belongs in the DHT. A joining node then uses DHT to search for nodes starting from the lowest domain to which it belongs. Below, we describe this procedure in detail.

  Joining nodes are provided with information about some arbitrary nodes in the system, which we call *contact* nodes. Each node after joining the DHT will perform a `put` operation for keys corresponding to different domains to which this node belongs. For example, a node `begonia.cs.utexas.edu` will perform three put operations for keys corresponding `cs.utexas.edu`, `utexas.edu`, and `edu` with its own IP address as the value. Now, a new node, say `linux1.ece.utexas.edu` that wishes to join the DHT will use a supplied contact node to perform a sequence of `get` operations starting from the lowest domain `ece.utexas.edu` up to the highest domain `edu` until it finds another

node. If no node is found, then it considers itself to be the first node in the `edu` domain and uses a user-specified contact node as the bootstrap node.

Note that, by default, the DHT `put(key, value)` operation appends the `value` to other existing values stored for that key. This approach might be unscalable as the node responsible for the key corresponding to a large domain has to store a large number of values. For this particular case, we do not need the `get(key)` operation to return all values stored for that key but *any* or *few* values will suffice. So, instead of storing all values for a key, we store only a few values for each key and use a FIFO policy to purge the value list as the new values are inserted.

Though the DHT-based technique described above provides a way for locating an appropriate bootstrap node, concurrent joins of many nodes in a domain can become very slow as all nodes might use one or a few bootstrap nodes.

- **Using SDIMS:** Though the DHT-based technique provides a way for locating an appropriate bootstrap node, concurrent joins of many nodes in a domain can incur high load on one or a few bootstrap nodes. Here we propose a technique based on the SDIMS aggregation mechanism that alleviates this problem by forming a tree among concurrently joining nodes in a domain. The tree is rooted at an existing node closer to the joining nodes in terms of domain-nearness. Each node in the tree uses their parent as the bootstrap. Note that a node's parent might not have completed joining the DHT when the node contacts it. The parent in this case sends a *retry* message to the node to inform that it has not yet joined the DHT. The joining node in this case retries after a short timeout period. Below we describe how we use SDIMS to locate bootstrap nodes and build such trees.

  Suppose node A initially knows some contact node B in the system. Assume that node B has already joined the system and is already running

SDIMS. Node A with name `begonia.cs.utexas.edu` performs three updates on Node B for attributes (`DOMAIN-NODES`, `cs.utexas.edu`), (`DOMAIN-NODES`, `utexas.edu`), and (`DOMAIN-NODES`, `edu`) with the same value of (IPADDR$_A$, `WILL_JOIN`). After the node completes joining the SDIMS, it will update those three attributes with a new value (IPADDR$_A$, `ALREADY_JOINED`). SDIMS will be pre-installed with an aggregation function for attribute type `DOMAIN-NODES` that picks a few representatives, given a set of tuples, prefering values with `ALREADY_JOINED` over values with `WILL_JOIN`. In case of a tie, it chooses a value with lower IP address. The joining node A, after performing updates on node B, performs probe operations for global aggregates for those attributes to locate a node that is closer in terms of domain-nearness metric. So in the example, Node A will perform probe operation on node B requesting to locate other nodes in the `cs.utexas.edu` domain. If that probe returns any non-null IP address vector, then node A uses one of the nodes in that vector as a bootstrapping node. If null value is returned, then node A continues probing for nodes in higher domains — in this case for nodes in the `utexas.edu` domain and then in the `edu` domain.

5.3.4 Maintaining Consistent Leafsets

Note that although mechanisms described in the previous section provide one way for rendezvous between nodes in same domain, they do not guarantee that a new joining node always finds other nodes in its domain. For example, if nodes in a new domain concurrently join an existing ADHT, they might not find each other during the join phase and might use some node outside their domain as the bootstrap node. Even though using the SDIMS-based technique reduces the chances of such a scenario, this can happen when reconfigurations happen in the system and SDIMS is in the state of repairing aggregation trees and aggregate values at different levels in

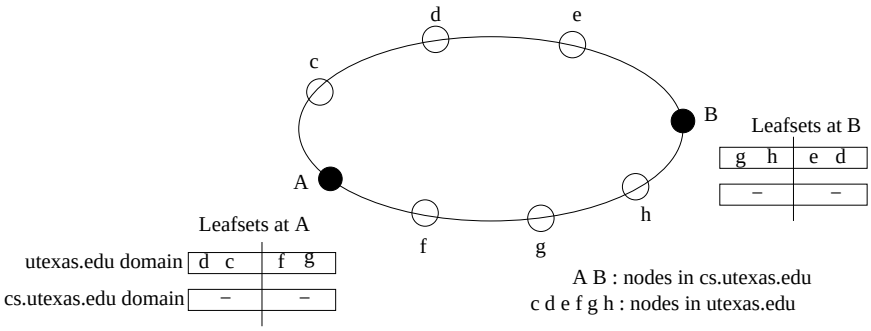


Figure 5.7: Concurrent joins leading to path convergence property violations. Nodes A and B in `cs.utexas.edu` join concurrently using nodes in `utexas.edu` domain as bootstrap nodes. Observe the incorrect leafset tables at node A and B corresponding to `cs.utexas.edu` domain.

the aggregation trees (explained in detail in Chapter 6). This scenario where nodes in a single domain use nodes outside their domain as bootstrap nodes can lead to incorrect leafset tables at those nodes. For example, we show a case in Figure 5.7(a) where two nodes A and B in domain `cs.utexas.edu` in join simultaneously and use some nodes outside `cs.utexas.edu` as bootstrap nodes. Hence, the lowest domain leafset tables for node A and node B end up in an incorrect state in this case which leads to the violation of path convergence and locality properties. In our system, we ensure path convergence and path locality properties are met by using the following mechanism — each node periodically searches for other nodes in all domains that it is part of using DHT-based or SDIMS-based methods, contacts those nodes, and corrects any incorrect entries in its routing table or leafset tables. In the following, we first describe the *Zippering* mechanism that is useful for any DHT to handle partitions and then describe how we use this to achieve leafset consistency in ADHT.

Zippering for Mending Partitions Several DHT systems, like SkipNet [48] and Willow [102], provide a way to merge partitioned components. In Pastry, nodes periodically perform a leafset maintenance task where each node checks for the live-

ness of its leafset table entries and broadcasts its current leafset to members of that leafset if it finds any dead entries. Though this maintenance protocol is appropriate to mend machine crash failures, it fails during network partitions, leading to possible partitions in the DHT. Even after the network heals, Pastry does not have a mechanism where these different partitions can merge back together.

To mend partitions in a DHT, we need a way to rendezvous between nodes in those partitions and a way to merge a partition with another after a node in a partition discovers a node in the second partition. In ADHT, nodes keep a log of dead nodes and occasionally ping them to check if they became alive. The entries in the log expire after a certain timeout period. This mechanism will ensure the rendezvous between nodes in different partitions for network failures lasting shorter than the above mentioned timeout period. For handling longer network failures, we provide an interface for an administrator to initiate the rendezvous. This interface also allows administrators to merge two separately formed DHTs into a single DHT.

After a node, say node A, discovers another node B in a different partition, node A routes a *join* message from node B for target key ID_A, similar to the join algorithm, using node B as the bootstrap node. Node A then receives leafset and routing information from nodes on the routing paths. Note that if the route for ID_A from node B does reach the node A, then it implies that node B indeed belongs to the same partition as node A. In case of a partitioned DHT, the final node on the path will be unaware of node A. Node A then exchanges its leafset with that final root node and both nodes correct their leafsets. Note that nodes in ADHT perform periodic leafset exchange with their neighbors in the leafsets. This periodic leafset exchange ensures that the information about nodes in one partition reaches nodes in the other partition thus integrating two partitions into one.

We illustrate the zippering mechanism in Figure 5.8. We show the partitions as two different circles corresponding to the logical ID space and show nodes in

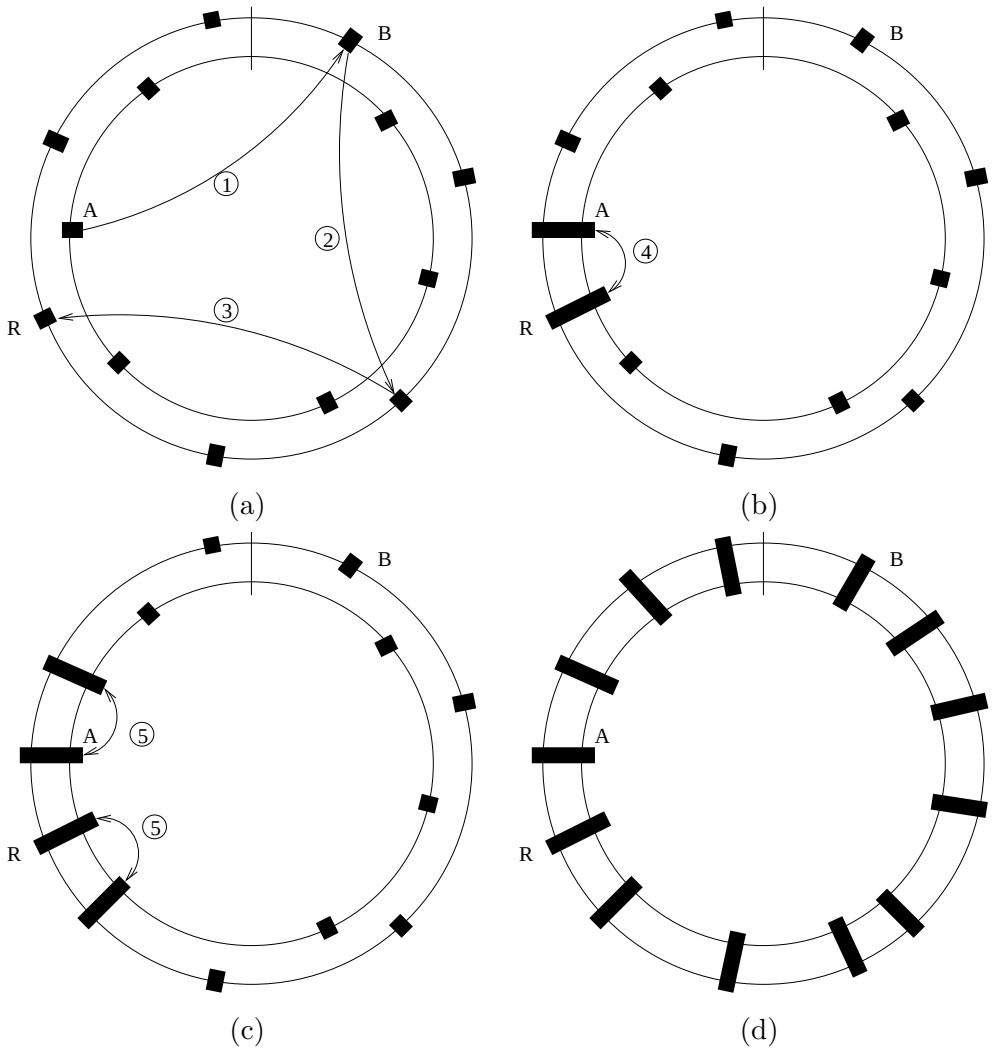


Figure 5.8: Zippering steps: Node A in a partition discovers node B and starts the zippering procedure. (a) Node A with ID ID_A starts a join procedure using node B as the bootstrap node. Node B routes that request towards node D which is the current root for ID_A in B's partition. (b) Node A and Node D detect the partitions and exchange their leafsets. (c) Node A and Node D propagate the information about partitions during their periodic leafset exchanges with neighbors in their leafsets. (d) Finally, the partition information spreads around the whole ID space and the partitions are merged together.

different partitions on those two different circles. In this figure, node A with say ID_A initially discovers node B and starts a join procedure using node B as the bootstrap node (message 1 in the figure). Node B forwards the message using ADHT routing towards the root for key ID_A, which is routed to node D that is currently responsible for key ID_A in the ring to which node B belongs. This join message routing is depicted as messages 2 and 3 in the figure and will take $O(\log N)$ hops in a partition with N nodes. When node D returns an answer to node A, node A then knows that D is in a different partition. Node A and node D exchange their leafsets so that both of them now are in the same partition. Thus the logical ID space near node A and node D is mended (depicted in the figure as node A and node D participating in both rings). As node A and node D perform their periodic leafset exchanges with other nodes in their leafsets, the mending of logical ID space spreads around the ring (shown in Figure 5.8(c)) and finally the partitions are zippered together when the mending is done at all nodes in the partitions (depicted in the Figure 5.8(d)).

Fast Zippering The method for zippering in the last few paragraphs can take $O(N)$ time steps before two partitions with N nodes are merged together. Note that the healing of the partitions is spread around the ID ring linearly in time. To hasten the zippering process, we propose the following scheme. Upon detecting partitions and mending leafsets, a node picks a constant number of random nodes in its current partition (from its previous leafsets and routing table) and informs them about the nodes in the other partition (new entries in the leafset). These nodes then start zippering procedure to mend their leafsets. With each node informing a constant number of other nodes after it completes zippering, all nodes will get to know of partitions in $O(\log N)$ such rounds with high probability. Overall, all nodes will complete performing zippering step by $O(\log^2 N)$ time steps. This procedure will incur $O(N \log N)$ messages as each node might perform the zippering step in

contrast to $O(N)$ messages in slow zippering procedure described previously.

Leafset Maintenance using Zippering In ADHT, we also use the above zippering mechanism to correctly maintain leafsets at different levels. Periodically each node looks for other nodes in each domain it is part of using the DHT-based method or the SDIMS-based method described in the previous section. Once it finds such nodes, it uses the zippering mechanism to mend any possible partitions. Note that in the description of the zippering mechanism, we described that a node only considers the final root in the routing path. To maintain correct leafsets at all levels, a node in ADHT looks at leafsets it receives from other nodes on the path (root nodes for subdomains) to fill and correct its leafset tables.

An important issue that needs to be addressed in the above approach is how to decide the frequency with which a node performs the probing step. If all nodes frequently perform such operation, then one or few nodes that are chosen as representatives for a large domain will be inundated with a large number of join requests. To avoid such hot spots in the system, each node chooses the frequency of such checks to be proportional to the estimated number of nodes in the domain. The number of nodes in a domain can be estimated from the leafset for that domain at a node leveraging approaches used in Viceroy [63] and Symphony [64]: If X_s denote the sum of segment lengths (length of logical ID space) managed by any set of s nodes, then $\frac{s}{X_s}$ is an unbiased estimator for the number of nodes.

5.3.5 Extracting Aggregation Trees from ADHT

Extracting aggregation trees in ADHT is similar to the process presented in Chapter 4. In Figure 5.3, we present an example showing how extracting aggregation trees from the routing structure of Pastry violates the administrative isolation requirement. For the same example, the ADHT routing tree and the corresponding aggregation tree we build in SDIMS are shown in Figure 5.9. Since the ADHT

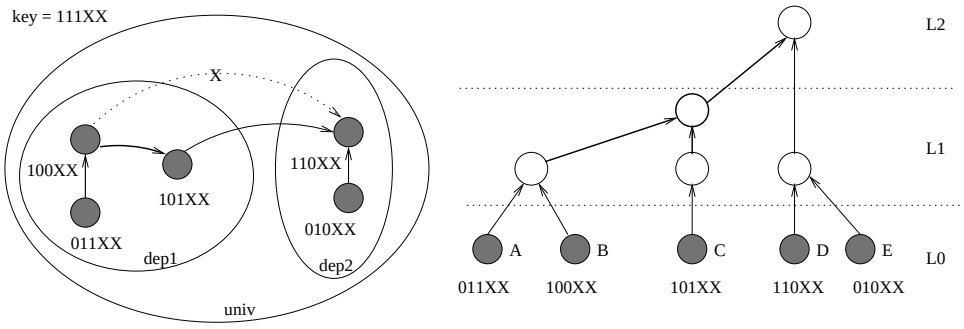


Figure 5.9: Autonomous DHT satisfying the isolation property. Also the corresponding aggregation tree is shown.

algorithm routes to reach a domain root node before jumping out of the domain, the routing path does not necessarily correct any bits during intermediate routing steps. In the original Pastry, such case happens only on the last few steps of the routing. As explained in Chapter 4, we insert extra virtual nodes to handle those cases.

5.4 Properties

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

5.4.1 Correctness

We discuss correctness of ADHT from the perspective of three properties — (1) Consistent Routing [22]: A lookup operation for a key always ends at the current root node responsible for the key in the system, (2) Path Convergence, and (3) Path Locality.

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

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

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

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

Lemma 2 *If the global leafset at all nodes is consistent and after the system becomes stable (no further node and network failures), eventually all leafsets at different domain levels at all nodes become consistent.*

Even when global leafset is consistent, other domain level leafsets can be inconsistent due to partitions in the domain (example in Figure 5.7. But in ADHT, each node looks for other nodes in the same domain periodically and performs a

zippering step. When the global leafset is consistent and the system is stable, if we have a partition in nodes of a domain, then at least one node in one of the partitions will discover a node in another partition. The zippering step following such discovery attaches those two partitions into one and the periodic leafset exchange procedure ensures that the leafset corresponding to this domain on all nodes in those both partitions become consistent.

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

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

Theorem 1 *If all network partitions and node failures are of duration less than $T_{deadNodePurge}$ and after the system is stable, eventually consistent routing, path convergence, and path locality properties are met.*

5.4.2 Performance

Increased Path Length in ADHT In contrast to Pastry routing, ADHT routes to a domain root node before jumping out of the domain to ensure path convergence. In Pastry routing, when an entry is found in teh routing table that is closer to the key than the current node, then the request is forwarded to that node. In ADHT, entries in a domain leafset are given priority over a routing table entry when the routing

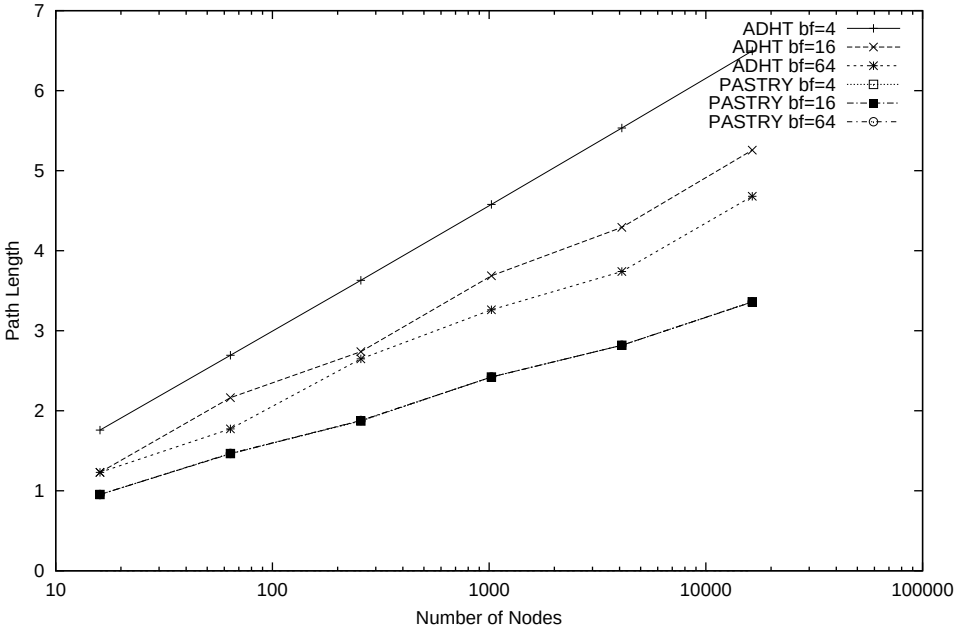


Figure 5.10: Average path length to root in Pastry versus ADHT for different branching factors. All Pastry lines overlap as the branching factor does not effect the Pastry routing procedure.

table entry corresponds to a node outside the domain. This routing procedure leads to an increase in the hop count for reaching a root node for a key from any node in the system. This path length increases to $O(\log N + l)$ from $O(\log N)$ in Pastry, where N is the number of nodes in the system and l is the number of levels in the administrative hierarchy, which in practice will grow no faster than $\log N$.

5.5 Experimental Evaluation

Though the routing protocol of ADHT might lead to an increased number of hops to reach the root for a key as compared to the original Pastry, the algorithm conforms to the path convergence and locality properties and thus provides administrative

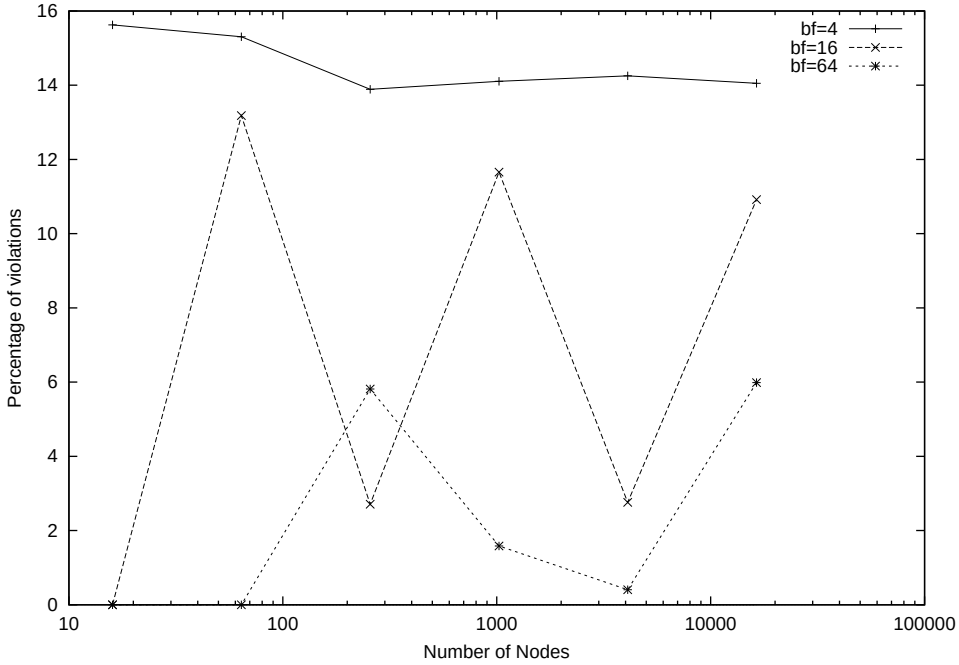


Figure 5.11: Percentage of probe pairs whose paths to the root did not conform to the path convergence property in Pastry. We do not show lines for ADHT as paths of all probe pairs conform to the path convergence property in ADHT.

isolation. In Figure 5.10, we quantify the increased path length by comparisons with unmodified Pastry for different sized networks with different *branching factors* of the administrative hierarchy. The branching factor denote the maximum degree of each internal node in the hierarchy. Note that the number of levels in the administrative hierarchy for an N node system with a branching factor b is $log_b N$. All lines corresponding to the Pastry overlap as the branching factor does not affect the performance of the original Pastry. Observe that the difference between average path length in ADHT compared to the path length in Pastry increases with decreasing branching factor because the depth of the administrative hierarchy increases with decreasing branching factor.

To quantify the performance of Pastry and ADHT with respect to the path

convergence property, we perform simulations with a large number of probe pairs — each pair probing for a random key starting from two randomly chosen nodes. In Figure 5.11, 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.

5.5.1 Zippering

For demonstrating the performance of ADHT Zippering, we build a DHT ensuring that nodes in the DHT get partitioned into two equal sized partitions. Then we inform some nodes of a partition about a node in the second partition (simulating the node discovery mechanism mentioned in Section 5.3.4), which starts the zippering activity. In Figure 5.12, we plot performance results for fast zippering described in Section 5.3.4 and compare it with slow linear zippering. We measure performance as time taken in terms of simulation steps and number of messages incurred during zippering. We plot for two cases where in one case only one node discovers a node in the other partition and in another case where randomly chosen 1% nodes in a partition discover a node in the other partition. As described earlier, fast zippering mends partitions in $O(\log^2 N)$ steps in contrast to slow zippering which can take $O(N)$ steps; in terms of the number of messages, fast zippering incurs $O(N \log N)$ messages in contrast to $O(N)$ messages in slow zippering. Note that in ADHT, all nodes actively search for nodes in other partitions. Hence, more than one node might discover the partition and start the mending process. In the same figure, we also plot the metrics comparing fast and slow zippering when the mending is initiated by one percent of the nodes in the system. Note that when an f fraction of nodes start the zippering process, then mending of partitions takes $O(1/f)$ simulation steps in

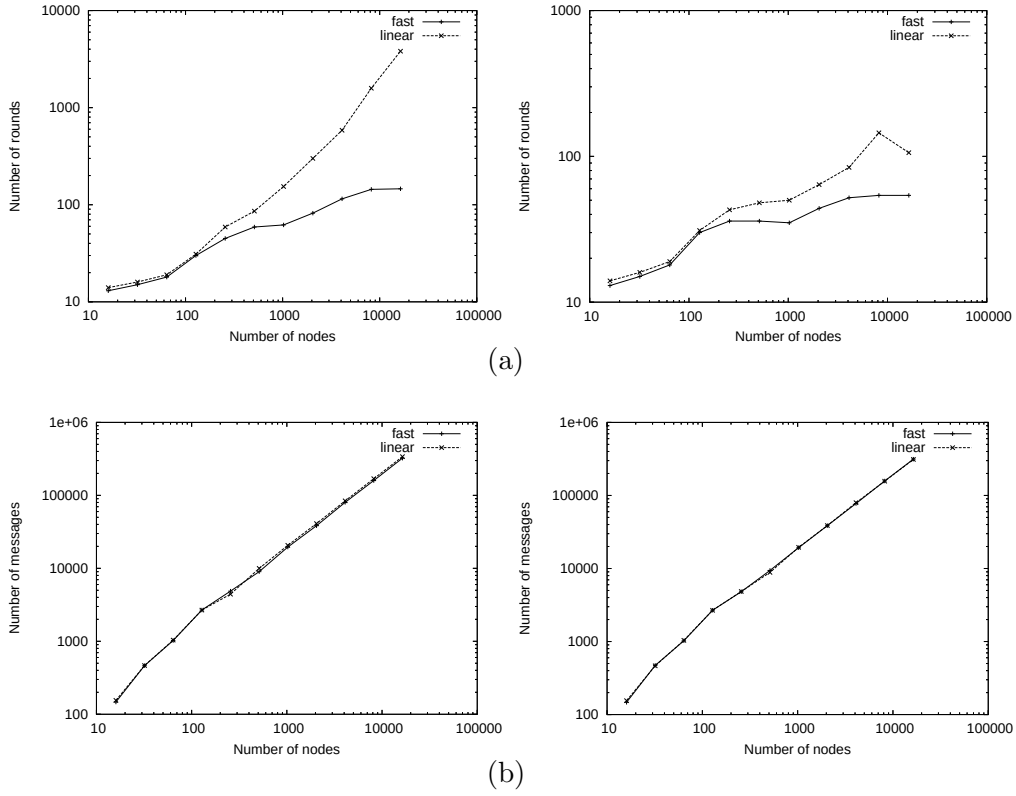


Figure 5.12: Performance of ADHT in merging two equal sized partitions in two cases — when only one node of a partition discovers a node in another partition and when 1% of nodes in a partition discover a node in another partition. We compare performance in case of both fast and slow zippering mechanisms described in Section 5.3.4. (a) Time taken (in terms of number of simulation time steps) to achieve leafset consistency. (b) Communication cost incurred for leafset consistency.

slow zippering and $O(\log^2(1/f))$ in fast zippering.

5.6 Related Work

Internal DHT trees typically do not satisfy domain locality properties required in our system. Castro et al. [23] and Gummadi et al. [44] point out the importance of path convergence from the perspective of achieving efficiency and investigate the

performance of Pastry and other DHT algorithms, respectively. In the later study, domains of size 256 or more nodes are considered and their studies show that path convergence is satisfied with high probability. In SDIMS, we expect the size of administrative domains at lower levels to be much less than 256 and it is at these small sizes that the path convergence fails more often (Refer to Graph 5.11 — smaller branching factors incur higher percentage of violations).

SkipNet [47] provides domain restricted routing where a key search can be limited to a specified domain. This interface can be used to ensure path convergence by searching in the lowest domain and moving up to the next domain when the search reaches the root in the current domain. Although this strategy guarantees path convergence, it loses the aggregation tree abstraction property of DHTs as the domain constrained routing might touch a node more than once (as it searches forward and then backward to stay within a domain). Also the search can be quite inefficient as it searches linearly through the nodes in a domain once the search reaches a node in the required domain.

Mislove et al. build multiple rings [67] to provide administrative control and autonomy in structured peer-to-peer networks. Nodes in a administrative domain form a ring and few nodes of a domain act as gateways for that domain and participate in a ring corresponding to the next higher domain. Each ring is assigned an ID and lookups involve both a key and a ring ID. To enable locating a gateway responsible for a ring ID, gateways of a ring advertise the corresponding ring ID in higher level rings using standard DHT `put` interface. In contrast to ADHT, this requires that a node, say acting as gateway at all levels, participate in $O(l)$ separate DHTs implying a maintenance overhead of $O(l.\log N)$, where l is the number of levels in the administrative hierarchy and N is the number of nodes in the system.

An extension of Chord considers multiple virtual rings [53] focusing on efficiently supporting multiple subgroups using an existing Chord ring. Their ideas

might be applicable to achieve administrative isolation for a two-level administrative hierarchy but it might be inefficient for multi-level hierarchy.

Coral [37, 38] is a peer-to-peer content distribution network which uses a decentralized hierarchical clustering algorithm by which nodes can find each other and form clusters of varying network diameters. Coral then builds different DHTs at each level in the hierarchical clustering tree. In contrast to ADHT's focus on supporting administrative isolation, Coral focuses on finding a nearby node in terms of network locality. They consider shallow hierarchies (three-level hierarchy) and mainly focus on automatically building such hierarchies based on observed round-trip times between nodes.

Similar mechanisms to Zippering are proposed for handling organizational disconnects in SkipNet [48] and for merging two separately formed DHTs in Willow [102].

5.7 Summary

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

ADHT builds upon and augments an existing DHT, Pastry, to achieve administrative isolation through the following four key ideas: (i) Each node in ADHT has multiple leafsets corresponding to levels of administrative hierarchy in which that node participates, (ii) ADHT employs a novel key space assignment and a novel routing algorithm that ensure search paths from nodes in a domain for key

converge at a node within that domain, (iii) A node joining ADHT locates a nearest node in terms of domain nearness and uses that node as the bootstrap node to join ADHT, and (iv) Each node in ADHT periodically tests for partitions in each domain it participates and uses a *Zippering* mechanism to mend partitions.

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

Chapter 6

Robustness

6.1 Introduction

In large scale distributed systems, reconfigurations are a norm [13, 14, 82, 111]. Reconfigurations lead to changes in the DHT routing layer which in turn implies changes in aggregation trees for different attributes. In SDIMS, aggregate values are recomputed at the virtual nodes that are affected by changes in an aggregation tree. Also the aggregate values are then propagated according to the install time `UP` and `DOWN` parameters. Though this reaggregation mechanism is a safe default action, it can incur high communication costs when reconfigurations happen at a high rate (e.g., $O(m \log N)$ messages when a new node joins or an existing node leaves a system with N machines and m attributes installed with Update-Up strategy). Also probes that are being evaluated during reconfigurations might be delayed, fail, or return incorrect results. In this chapter, we propose three techniques for masking many failures — flexible API based, K-way hashing, and supernodes. We also evaluate these three methods analytically and compare them to the basic strategy of reaggregation on failures.

6.2 Reaggregation

In large scale distributed systems, reconfigurations are a norm [13, 14, 82, 111]. Even though individual machine failure rates might be very small, the rate of failures in the whole system can be quite high. For example, in a system with ten thousand machines, if a machine's average life time is three years, then the system will experience a reconfiguration once in every three days. But individual machines also experience transient failures during their lifetime due to network failures, hardware failures, or simply due to an user shutting down her machines during night. If individual machines experience transient failures at an average rate of once in a week then we will observe reconfigurations in the whole system at an average rate of one machine reconfiguration per minute.

Each reconfiguration in the system affects the aggregation trees in an SDIMS. In the underlying Autonomous DHT, each node periodically checks the liveness of the nodes in its routing table and the nodes in its leafsets. Also each node periodically exchanges its leafsets and routing table information with its neighbors in the leafsets. Each node maintains several backup nodes for each slot in the routing table to replace the primary when it goes down or moves away from this node in terms of network distance. Also when a node observes that a neighbor in one of its leafsets is discovered failed, it removes the neighbor from the leafset. Thus, the next hop for different keys might change over time due to reconfigurations in the system, which in turn cause changes in the aggregation trees in the SDIMS.

Here we present an example to illustrate the effect of reconfigurations in the system on the aggregation trees. Consider an ADHT with the next hop pointers for each node shown in Table 6.1. For a key 111, the DHT tree and the aggregation tree for this example are shown in Figure 6.1. Now consider a reconfiguration where a node with ID 111 joins the system. We show the resulting pointer table after node with ID 111 joins in Table 6.2. Note that the next hop pointers for some of

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

Table 6.1: Example pointer table for a DHT comprising of six nodes and addresses drawn from a 3-bit ID space

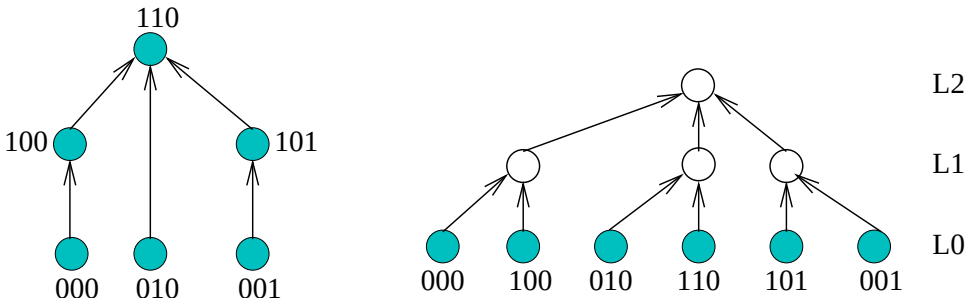


Figure 6.1: DHT tree for key=111 based on the pointer table in Table 6.1. We also show the corresponding aggregation tree.

the nodes in the system change. This affects DHT trees and the aggregation trees. The changes to the DHT tree and the aggregation tree corresponding to key 111 are shown in Figure 6.2. Similarly, a reconfiguration involving a node leaving the system affects the DHT trees and the aggregation trees. The pointer table when the node with ID 101 leaves the system is shown in Table 6.3 and the corresponding changes in the DHT tree and the aggregation tree for key 111 are shown in Figure 6.3.

6.2.1 Reaggregation Procedure

When an aggregation tree topology changes, the aggregate values at virtual nodes might not correspond to the correct values. Note that aggregate value at a node is computed by performing aggregation function across values at child nodes. When a

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

Table 6.2: Pointer table for the DHT shown in Table 6.1 after a node with ID 111 joins the DHT. The changed entries are highlighted in boxes.

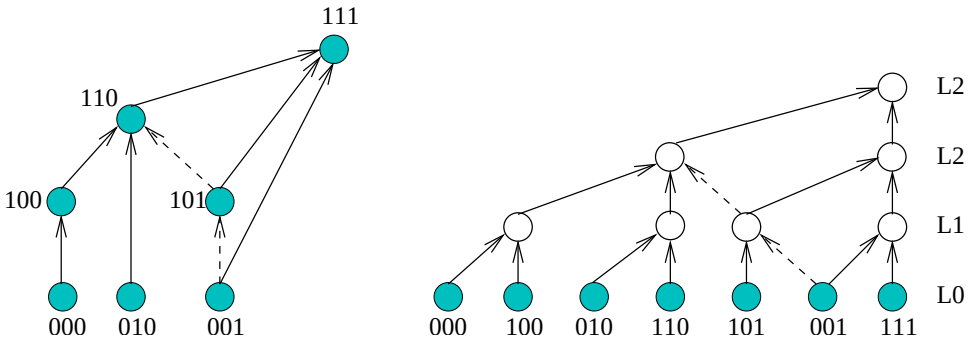


Figure 6.2: DHT tree for key=111 after a node with ID 111 joins the system. We also show the corresponding aggregation tree. The dotted arrows denote the edges in the DHT tree and the aggregation tree that change when the new node joins the system.

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

Table 6.3: Pointer table for the DHT shown in Table 6.1 after the node with ID 101 leaves the DHT. The changed entries are highlighted in boxes.

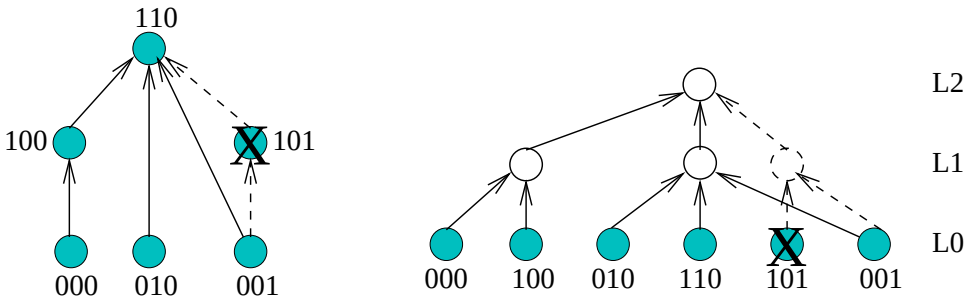


Figure 6.3: DHT tree for key=111 after the node with ID 101 leaves the system. We also show the corresponding aggregation tree. The dotted arrows denote the edges in the DHT tree and the aggregation tree that change when the new node joins the system.

level-l virtual node gets a new child, the child will forward its level-$(l-1)$ aggregate value to the virtual node if the attribute type is installed with `UP`$> (l-1)$. So the virtual node needs to recompute its level-l aggregate value. Also if the virtual node loses an existing child, then also it needs to recompute aggregate value. Similarly, when a level-l virtual node gets a new parent, it needs to send its level-l aggregate value to the new parent `UP`$> l$ so that the parent can compute its aggregate value. For example, consider the aggregation tree for key 111 in the DHT corresponding to the pointer table presented in Table 6.1. We show the aggregation tree for this key before and after the node with ID 101 leaves the system in Figure 6.4. Consider an aggregation function that computes summation of given integer values and suppose it is installed with Update-Up strategy. The values for the attributes at the individual machines (leaf nodes) in the aggregation tree are shown next to those nodes and the aggregate values next to the virtual nodes correspond to the aggregate values before the node with ID 101 leaves the system. The values in boxes next to virtual nodes A and R show the correct aggregate values that those nodes should have after the node 101 leaves the system. Note that the node A obtained a new child in this case, the root virtual node R lost a child, and the node with ID 001 got a new parent.

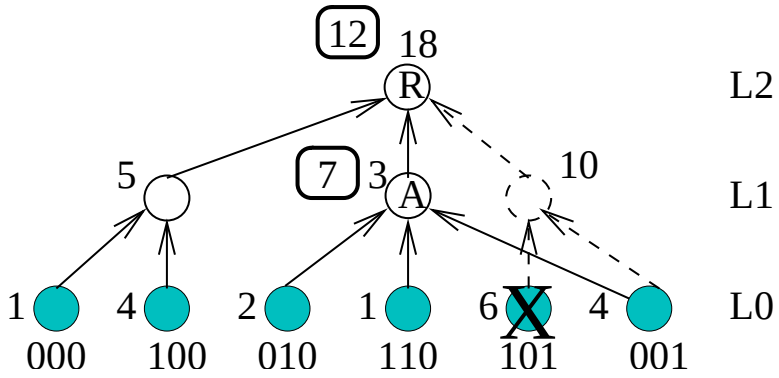


Figure 6.4: We consider a simple summation operation as aggregation function. Here we show the aggregation tree for an attribute with key=111 before and after the node with ID 101 leaves the system for the example in Table 6.1. We show the aggregate values next to each node that are computed before the node 101 leaves the system. The values in boxes next to nodes A and R are the aggregate values after node 101 leaves the system and after the aggregation tree structure changes.

Algorithm 2 onNewParent(*parent*)

```
1: probeSet ← probes waiting for response from previous parent
2: if probeSet ≠ ∅ then
3:     send probeSet to the new parent
4: end if
5: Forward the aggregation function along with install parameters
6: if UP > myLevel then
7:     Send local aggregate value to the new parent
8: end if
```

The pseudo-code for the actions of a virtual node in an aggregation tree upon reconfigurations is shown in Algorithms 2, 3, and 4 — *onNewParent(parent)* is invoked when a virtual node obtains a new parent, *onFailedChild(child)* is invoked when a virtual node loses an existing child, and *onNewChild* is invoked when a virtual node gets a new child.

Algorithm 3 onFailedChild(*child*)

```
1: Remove child from the children set
2: probeSet ← probes waiting for response from child
3: if probeSet ≠ ∅ then
4:    re-evaluate probes
5: end if
6: Recompute the local aggregate value
7: if local aggregate value changed then
8:    if UP > myLevel AND parent exists then
9:       Send local aggregate value to the parent
10:   end if
11:   if DOWN > 1 then
12:      Send local aggregate value to all children
13:   end if
14: end if
```

Algorithm 4 onNewChild(*child*)

```
1: Add child to the children set
2: Forward the aggregation function along with install parameters
3: for all level-l′ aggregate values this node has do
4:    if DOWN > l′ − l then
5:       Send level-l′ aggregate value to the child
6:    end if
7: end for
```

6.2.2 Reaggregation Costs

Reaggregation of data is performed when reconfigurations happen in the system leading to changes in the aggregation trees. Reaggregations can be expensive in terms of communication costs depending on how extensively the system is reconfigured. Also probes that are outstanding when reconfigurations happen can be affected — they may fail or be delayed as aggregation trees are reconfiguring. In the following, we analyze reaggregation costs both in terms of message cost and in terms of probe success probability during reconfigurations.

Consider m attributes and N nodes in a system. Suppose all attributes have the same aggregation strategy of Update-Up. We consider a reconfiguration event

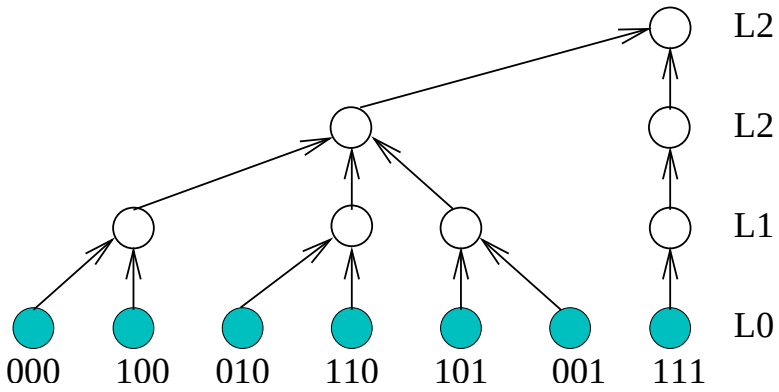


Figure 6.5: Best case example for changes in an aggregation tree on a node join. Here we show the aggregation tree for key 111 before and after node 111 joins. Note that adding node 111 did not change the aggregation tree for key 111 before node 111 joins.

with a single node join. We present best case, worst case, and average case message costs for this reconfiguration event.

Best case: When a machine A with ID ID_A joins, it is chosen as the next hop by only one node that was previously for key ID_A and no other node in the system uses A as the next hop for any bit correction. In this case, each virtual node hosted on this new node has only one child, a lower level virtual node hosted on the same machine. Figure 6.5 illustrates an aggregation tree for an attribute with key 111 before and after a node with ID 111 joins. The number of messages that machine A sends to other machines is $(m - \frac{m}{N})$ (machine A will host the root for aggregation trees corresponding to $\frac{m}{N}$ attributes). Machine A will send $\frac{m}{2}$ messages to its first bit correcting neighbor in the DHT. Each of these messages can further cause up to $\log N$ messages as this message causes recomputation and further propagation of the aggregate value at the parent virtual node. Machine A will also send $\frac{m}{4}$ messages to its second bit correcting neighbor and each of those messages can further cause $(\log N - 1)$ messages, send $\frac{m}{8}$ to the third bit correcting neighbor which can further

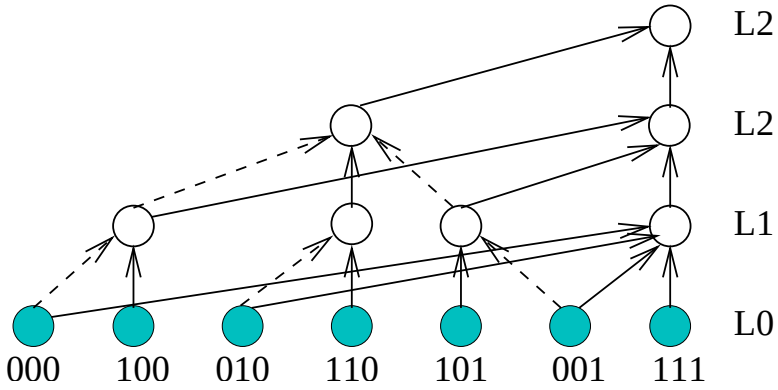


Figure 6.6: Worst case example for changes in an aggregation tree on a node join. Here we show the aggregation tree for key 111 before and after node 111 joins.

cause $(\log N - 2)$ messages, and so on. Also the previous root for ID_A in the DHT will send about $\frac{m}{N}$ messages to machine A as A will host the new root for the aggregation trees corresponding to about $\frac{m}{N}$ attributes. Thus the total number of messages in the system due to this node's join can be

$$\frac{m}{2}\log N + \frac{m}{4}(\log N - 1) + \cdots + \frac{m}{N} = O(m \log N)$$

Worst case: In the best case, only one node chooses machine A as the next hop. In the worst case, machine A can be chosen as first bit correcting neighbor by about $\frac{N}{2}$ machines, as second bit correcting neighbor by about $\frac{N}{4}$ machines, and so on. An example aggregation tree change in a worst case is depicted in Figure 6.6. Note that the machines that chose A as the first bit correcting neighbor will send about $\frac{m}{2}$ messages each of which causes recomputation of aggregate value at level-1 virtual node hosted on machine A and propagation of the aggregate value upwards till root causing about $\log N$ messages. Thus the total number of messages due to this node's join can be

$$\left(\frac{N}{2}\right)\left(\frac{m}{2}\right)\log N + \left(\frac{N}{4}\right)\left(\frac{m}{4}\right)(\log N - 1) + \cdots + \left(\frac{m}{N}\right) = O(Nm\log N)$$

Average case: In an average case, a new machine A is chosen as an ith bit correcting neighbor by a small number of other machines. Suppose the number of such machine be a small constant c for each position of the ID. Reasoning similarly to the analysis in the previous cases, the total number of messages due to this reconfiguration event can be

$$c\left(\frac{m}{2}\right)\log N + c\left(\frac{m}{4}\right)(\log N - 1) + \cdots + \left(\frac{m}{N}\right) = O(cm\log N)$$

Message Cost with Churn: Consider a system with ten thousand machines and ten thousand attributes. With reconfigurations happening at a rate of one in a minute, the reaggregation will incur about 260000 msgs/min (with c=2, $cm\log N = 2 \times 10000 \times 13$). Assuming a message size of 256 bytes, this will cause on average about 4 Mbps traffic in the system. Note that if we did perform reaggregations only when machines really left the system or joined the system (which happens much rarely), then the cost will be significantly smaller. For reconfigurations happening at a rate of one in three years, reaggregation will incur an average of only 1Kbps traffic.

Probe failure probability Reconfigurations in the system not only incur communication costs for repairing the system but also affect the probes happening in the system. Here we compute the probability of a probe failure due to reconfigurations. Assume p be the probability of a single node failure in an aggregation. We assume independence in the failures of the individual nodes[1]. We consider a probe

[1]Note that multiple virtual nodes can be hosted on a single machine. Hence there is a correlation in the failure of the nodes in an aggregation tree. To simplify the analysis, we ignore this correlation.

to be failed if any node on the path from the probing leaf node to the nodes in an aggregation tree from which data is collected is affected by the reconfiguration. For example, in case of Update-Up, we consider a probe to be failed if any node on the path from the probing leaf node to the root node in an aggregation tree is affected by the reconfiguration. Hence the probability of a probe failure, $P_{Update-Up}$, is

$$\begin{aligned} P_{Update-Up} &= 1 - \text{all nodes on the path do not fail} \\ &= (1-(1-p)^{\log N}) \end{aligned} \tag{6.1}$$

If the attribute is installed with `UP`=all and `DOWN`=d with $d > 0$, then the probability of failure will be lesser than in the case of Update-Up. The probability of probe failure, P_{DOWN}, in this case is

$$\begin{aligned} P_{DOWN} &= 1 - \text{initial } (\log N - d) \text{ nodes on path do not fail} \\ &= (1-(1-p)^{(\log N - d)}) \end{aligned} \tag{6.2}$$

If the attribute is installed with `UP`=u with $u < maxLevel$ and `DOWN`=0, then the probe failure probability increases as the probe has to access more nodes to obtain the global aggregate value. With `UP`=u, a probe has to access $u+2^{(\log N - u)}$ nodes in the system. Hence the probability of probe failure, P_{UP}, is

$$P_{UP} = (1-(1-p)^{u+2^{(\log N - u)}}) \tag{6.3}$$

6.2.3 Lazy Reaggregation

Reaggregation of data can be expensive in terms of the communication cost as analyzed in the previous section. Moreover, a single reconfiguration event in the system can lead to multiple reconfigurations of an aggregation tree. During such reconfigurations, the system can cause high traffic in the network possibly disrupting

current updates and probes happening in the SDIMS and disrupting other flows in the system. To circumvent this, we perform *lazy* reaggregation where aggregate values computed in response to reconfigurations at a virtual node are propagated to its parent and/or children lazily in the background. Also for applications to tradeoff communication costs with consistency in the responses for probes, in our flexible API we allow applications to perform fast reaggregation.

A single reconfiguration event in the system can cause multiple reconfigurations in an aggregation tree. In the previous examples, we show all aggregation tree changes as if happening in one step. In reality, a node in a DHT loses an entry in its routing table and it might take some time for that node to discover another node to fill that entry. Losing an entry cause changes in some aggregation trees and filling with a new entry causes further changes. For example, in Figure 6.3, when 101 leaves, it might happen that node 001 have an empty entry for its first bit correcting pointer (1XX entry). In this case node 001 will route to node 010, its immediate neighbor on the ring, for key 111. But once the routing entry correcting the first bit is filled with a pointer to node 110, then the aggregation tree structure changes once more and is connected now. Note that the reaggregation procedure described above starts reaggregation as soon as any changes are observed in the aggregation tree and hence might perform multiple reaggregations on an aggregation tree for a single reconfiguration event.

Reaggregating data immediately upon observing reconfigurations in an aggregation tree, as described in the previous sections, can cause considerable traffic in the system. This is further exacerbated as a single reconfiguration event in the system can cause multiple reconfigurations of an aggregation tree. To alleviate this problem, the virtual nodes of an aggregation tree in SDIMS do not propagate reaggregated values as soon as they detect changes to the aggregation tree. Instead the messages are sent lazily over time. For example, on detecting a new parent for a

prefix, say $1XXX$, a node with ID $0XXX$ has to resend all attributes whose keys start with 1 and installed with Update-Up strategy to the new parent, which can be about half of the attributes at that node. Instead of sending all aggregate values for all those attributes at available bandwidth, the node will send them lazily as a background flow.

The lazy aggregation scheme propagates the aggregates in the system in a lazy fashion. Until the time the lazy reaggregation is in progress, probes in SDIMS might return inconsistent values. To allow applications to tradeoff communication costs with better consistency in the probe results, SDIMS provides `UP` and `DOWN` knobs in the Probe API (refer to Table 3.3) that applications can use to force fast re-aggregation of aggregate values. This is particularly useful when applications can detect that the answers returned by the SDIMS are stale. For example, consider a file location application built on the SDIMS. Suppose a probe for a file *foo* returns a machine A as the answer. Now, when contacted, if that machine A informs that it does not host file *foo* any more, then we know that the SDIMS aggregate values are currently inconsistent. If an application detects or suspects the answer for a probe as stale, then it can re-issue the probe setting `UP` and `DOWN` parameters to force fast reaggregation of the data.

6.3 Masking Temporary Reconfigurations

The reaggregation procedure described in the previous section is a necessary basic mechanism for handling reconfigurations in SDIMS. But in practice, many reconfigurations are transient that happen due to network failures like a router outage taking down connectivity to a set of machines or high loads on a machine filling network buffers and slowing down the machine [18, 54, 111]. For example, in the PlanetLab testbed [76] that does not have any dedicated administrative support, we observe that 50% of the network failures span less than an hour and 70% of network

failures span less than six hours [111]. Also most of these temporary reconfigurations do not affect the state of the SDIMS at the machines. Hence, reaggregating data for every reconfiguration might be unnecessary and expensive. We refer to temporary reconfigurations as *ups/downs* in contrast to *joins/leaves* which require reaggregation of data in SDIMS.

Reaggregation cost on transient reconfigurations can be avoided by not repairing aggregation trees on *ups/downs* but repair only on *joins/leaves*. This can be achieved by controlling the timeout used to detect a machine failure. In the underlying ADHT, each machine periodically exchanges *keepAlive* messages with $O(\log N)$ other nodes that are in its routing table or in the leafsets to check the liveliness of those machines. A machine is considered failed when that machine fails to respond to a *keepAlive* message within a specified timeout value. Thus, by choosing a large timeout value, unnecessary short failures of machines can be masked.

Though increasing the failure timeout reduces the number of reconfigurations and hence the cost for reaggregations, some probes in the system might fail or incur a long response latency. In SDIMS, nodes re-evaluate probes when they detect the failure of the node to which they have forwarded a probe. If we mask the recovery actions for temporary failures by increasing the timeout for failure detection, then the outstanding probes will either wait until the aggregation tree is reconfigured, which can be long, or timeout quickly and respond with a *probeFailure* message. Thus, with longer failure timeout periods, more probes will be affected. Also note that many nodes on a probe path simply help in forwarding probe and probe responses. For example, with Update-Up strategy, a probe from a node traverses $O(\log N)$ nodes to reach the root virtual node hosting the global aggregate value. Failure of any node on the path will result in failure of the probe.

In SDIMS, we propose three techniques that replicate either the aggregate values at all levels or the whole virtual nodes to reduce the effect of long failure

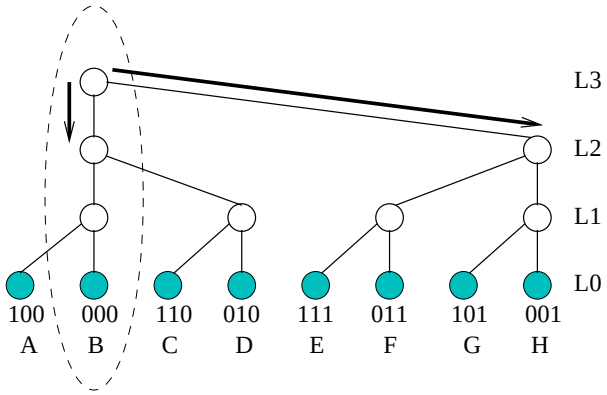


Figure 6.7: Using SDIMS flexible API: For an attribute with `UP`=all and `DOWN`=1 set at install time, the global aggregate and all intermediate aggregates are propagated down by one level. In the figure, we only show the propagation of global aggregate values. When node B fails taking down all virtual nodes it is hosting (shown in ellipse), response latency of probes at nodes E, F, G, and H is not affected as the global aggregate value is replicated on node H. The responses might be stale depending on the aggregation function and the values for those attributes.

timeout on probes — (1) By exploiting the flexible API, (2) K-way hashing, and (3) Supernodes. We detail these techniques in the following sections and analyze and compare their robustness and performance properties in Section 6.4.

6.3.1 Exploiting Flexible API

In our system, applications can control replication of aggregate values at different levels using `UP` and `DOWN` knobs in the Install API; with large `UP` and `DOWN` values, aggregates at the intermediate virtual nodes in an aggregation tree are propagated to more nodes in the system. By reducing the number of nodes that have to be accessed to answer a probe, applications can reduce the probability of incorrect results occurring due to the failure of nodes that do not contribute to the aggregate. For example, in a file location application, using a non-zero positive `DOWN` parameter in install ensures that a file's global aggregate is replicated on nodes other than the

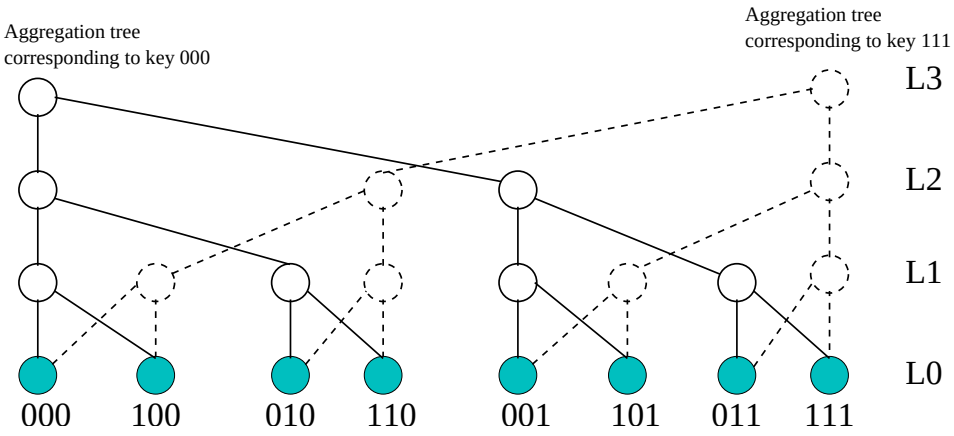


Figure 6.8: K-way hashing: Aggregate an attribute along multiple aggregation trees corresponding to different keys for robustness in the face of reconfigurations. Note that virtual nodes of different aggregation trees are hosted on different machines in the system improving robustness to machine failures.

root. Probes for the file location can then be answered without accessing the root; hence they are not affected by the failure of the root. In Figure 6.7, we present an example illustrating the usefulness of the `DOWN` parameter in SDIMS install API. Note that even when node B fails taking down all virtual nodes it is hosting (shown in ellipse), response latency of probes at nodes E, F, G, and H is not affected as the global aggregate value is replicated on node H. However, note that this technique is not appropriate in some cases. An aggregate value in file location system is valid as long as the node hosting the file is active irrespective of the status of other nodes in the system, but an application that counts the number of machines in a system may receive incorrect results irrespective of replication. If reconfigurations are only transient (like a node temporarily not responding due to a burst of load or short network failures), the replicated aggregate closely or correctly resembles the current state.

6.3.2 K-way Hashing

In k-way hashing, each attribute key is aggregated along k different aggregation trees corresponding to keys generated from using k different hash functions, chosen a priori, on the attribute key. In SDIMS, an attribute is aggregated along an aggregation tree corresponding to key = hash(attribute) and the intermediate virtual nodes of different aggregation trees are hosted on different physical machines in the system. For example, we show the aggregation trees corresponding to keys 111 and 000 in a eight node system in Figure 6.8. Observe that different nodes host virtual nodes of the aggregation trees and hence the failure of a node might drastically affect one aggregation tree but will not affect the other aggregation tree.

In K-way hashing, we exploit the load balancing property of SDIMS for robustness. Instead of aggregating an attribute along one aggregation tree, we aggregate an attribute along multiple aggregation trees. For example, when an update is performed for an attribute *(attrType, attrName)*, we perform K updates at the leaf node for attributes *(attrType, attrNamePAD*$_1$*), (attrType, attrNamePAD*$_2$*),...,* *(attrType, attrNamePAD*$_K$*)* with same value. Similarly, when a probe for aggregate of that attribute is performed, we probe along all K trees for the requested aggregate value and return all probe responses to the application. The application will have different options to interpret those return values — compute a simple majority, a median, etc., Also, instead of waiting for all the answers to arrive, SDIMS returns the responses as they arrive to reduce probe latency effectively.

Reconfigurations in the system still affect the K-way hashing strategy. Individual trees need to be repaired when a machine joins or leaves the system. But the frequency with which such repairs are done is reduced by choosing a large timeout value for failure detection. When reconfiguration is indeed detected, the tree and the aggregate values are repaired as described in the reaggregation section.

K-way hashing masks unavailability for probes at all stages of the reconstruc-

tion in the face of reconfigurations as the probes can respond without needing to wait for repairs to be completed. Note that responses for the probe might or might not be consistent during reconfigurations depending on whether those reconfigurations affected the consistency of previously aggregated value (as discussed in the previous section).

6.3.3 Supernodes

Instead of hosting a virtual node on a single physical machine, the *Supernode* approach is to replicate each virtual node in an aggregation tree on multiple physical nodes using either the state machine replication approach [55, 89] or the primary-backup protocols [20], two fundamental paradigms for implementing fault-tolerant distributed systems. The state machine replication approach would further involve delivering update and probe operations that access or modify the state of the virtual node to all replicas in the same order, which can be achieved by employing the Paxos algorithm of Lamport [56, 57]. Whereas K-way hashing is a simple scheme for replicating aggregate values in space without requiring any changes to SDIMS or underlying ADHT algorithms, the Supernode approach involves replicating all virtual nodes and thus need changes in SDIMS and ADHT algorithms. But Supernodes have better robustness properties (refer to Section 6.4) than K-way hashing while both incur similar communication costs for achieving the same amount of replication. In Figure 6.9, we illustrate an example where each virtual node in an aggregation tree is replicated on two physical machines.

Previously, several researchers have proposed node replication approaches for achieving robustness in structured peer-to-peer networks, like Super-Peer [112], super nodes [62], and Brocade [117]. In all these proposals, a DHT node is replicated at multiple physical machines which differs from our approach where *virtual nodes* are replicated on different machines. A key advantage in our approach is the flexibility

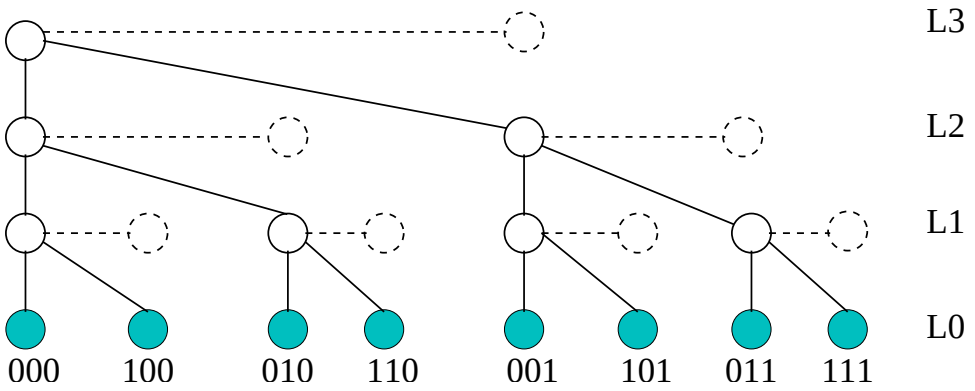


Figure 6.9: Supernodes: Example with each virtual node replicated on two physical machines. The dotted virtual nodes replicate the corresponding virtual nodes shown in solid circles connected by dotted lines.

in choosing replicas. Note that a virtual node that corrects a few bits of a key (near leaves in an aggregation tree) aggregates information about nearby nodes and hence it is efficient to replicate such virtual nodes on nearby nodes; a virtual node that is higher up in an aggregation tree aggregates information about a large number of nodes in the system and hence should be replicated on machines distributed widely to handle correlation in failures like subnetwork disconnections. Simply replicating a full DHT node restricts these options on how machines are chosen to host replicas. Astrolabe [101] employs a similar technique as Supernodes to achieve robustness to reconfigurations in the system. In Astrolabe, a single aggregation tree is used to aggregate all attributes and this aggregation tree is isomorphic to the administrative hierarchy. A virtual node in this aggregation tree corresponds to aggregating values from nodes in a domain. Each such virtual node is replicated on all machines in that domain and nodes use gossiping to maintain the replicas.

Based on the information at a virtual node that is replicated, we have two variants of Supernode approach: (A) replicate only the aggregate tree structure: In this variant, only the aggregation tree structure is replicated but the local state

of a virtual node (local aggregate values, cached child and ancestor values) is not replicated. Since the structure is maintained even on failures, queries can be answered by re-aggregating data on the structure. (B) replicate both structure and local state of a virtual node: this is full behavioral replication at both the ADHT layer and the local state of a virtual node and hence applications do not perceive any difference during a node failure. Though approach A masks DHT repair time, data still needs to be reaggregated - but data reaggregation can be performed on demand only for attributes that are probed and this needs to be done only for failed virtual nodes and reaggregation will not affect the whole path from the failed node to the root in an aggregation tree (that happens in case of non-node replication approaches). Approach B masks even those reaggregation costs but at the expense of extra communication cost for data replication during normal operation.

6.4 Analytic Comparison

In this section, we analyze message costs for different techniques proposed in the previous section to handle reconfiguration in an SDIMS during normal operation when there are no failures. We also analyze robustness of these approaches in terms of probability of a probe failure.

6.4.1 Cost analysis

We analyze the number of messages incurred by different approaches described in previous sections and present communication costs for the underlying ADHT, average number of messages per update operation, and average number of messages per probe operation.

Techniques that simply use the flexible API by varying `UP` and `DOWN` parameters in the install interface and K-way hashing do not incur any extra cost at the ADHT layer as they do not change the way ADHT works. The ADHT cost in these

Strategy	ADHT Cost	Update Cost	Probe Cost
Update-Up	$O(\log N)$	$O(\log N)$	$O(\log N)$
DOWN=d	$O(\log N)$	$O(2^d \log N)$	$O(\log N - \log d)$
K-way hashing	$O(\log N)$	$O(k \log N)$	$O(k \log N)$
Supernode	$O(k^2 \log N)$	$O(k \log N)$	$O(\log N)$

Table 6.4: Message costs for ADHT maintenance and for update and probe operations during normal operation in different techniques.

cases remain $O(\log N)$ cost per node that each node incurs to track liveliness of nodes in its routing table and its leafsets. Supernode approaches incur extra cost at the ADHT layer as each virtual node is replicated on multiple machines. For k-way replication, ADHT on each node will track $O(k^2 \log N)$ nodes. Note that each node in ADHT in the Supernodes case hosts about $O(k \log N)$ virtual nodes — at each level l, it will replicate level-l virtual node of about k other nodes. Since each virtual node itself is hosted on k machines, each machine in total will track $O(k^2 \log N)$ other machines. We tabulate ADHT costs and the average cost for update and probe operations for different techniques in Table 6.4.

6.4.2 Robustness analysis

Let p be the probability for failure of any node and suppose we have N nodes in the system. Thus the average path length from a leaf node to the root in an aggregation tree is $\log N$. Here also, to simplify the analysis, we assume that failures are independent.

The probability of an access failure at a leaf node in the case of k-way hashing, P_{hash}, is

$$\begin{aligned} P_{hash} &= \text{all k paths to root fail} \\ &= (\text{one path fails})^k \\ &= (1 - (\text{all nodes on a path are good}))^k \end{aligned}$$

Strategy	Probe failure	Probe latency(normal)
Update-Up	$(1-(1-p)^{\log N})$	$O(\log N)$
DOWN=d	$(1-(1-p)^{(\log N - \log d)})$	$O(\log N - \log d)$
K-way hashing	$(1-(1-p)^{\log N})^k$	$O(\log N)$
Supernode	$1-(1-p^k)^{\log N}$	$O(\log N)$

Table 6.5: Performance in terms of probe failure probability and probe latency during normal operation for different techniques.

$$= \quad (1-(1-p)^{\log N})^k \tag{6.4}$$

In case of Supernodes with k replicas, probability of access failure, $P_{supernode}$, is

$$\begin{aligned} P_{supernode} &= 1 - \text{Prob. of successful access} \\ &= 1 - (\text{at least one replica good at all levels}) \\ &= 1 - (\text{at least one replica is good out of k replicas})^{\log N} \\ &= 1-(1-p^k)^{\log N} \end{aligned} \tag{6.5}$$

These functions are compared in Figure 6.10 for a network with $2^{13} = 8192$ machines and with a replication factor $k = 4$. Clearly, the Supernode approach is more robust than either K-way hashing or using `DOWN`=2 (`DOWN`=2 setting achieves a similar amount of replication as k=4).

The probe failure probabilities in different techniques are tabulated in Table 6.5. We also present probe latencies during normal operation in that table. We tabulate probe latencies for the successful probes during reconfiguration events in Table 6.6. Observe that the probe latencies during failures in case of the Supernode approach can be higher as it depends on the timeout, T_{swicth}, used for detecting when a replica is down. Though the probe latency might be higher, the number of probes that fail in the Supernode approach is smaller than the other techniques.

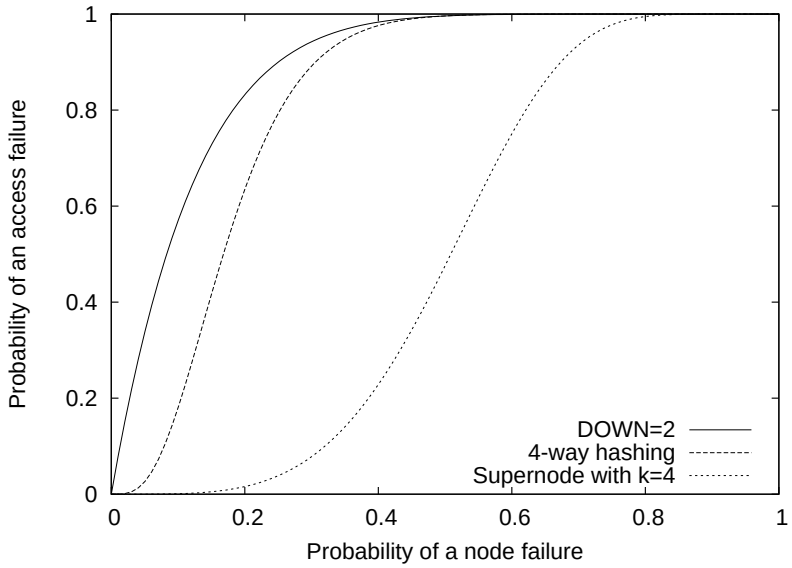


Figure 6.10: The probability of a probe getting affected by failure of nodes in the system for three different replication in space strategies. These graphs are based on analytic models shown in Equations 6.2, 6.4, and 6.5 for k=4 and $\log N$=13. `DOWN`=2 causes replicating a virtual node's aggregate value at about 4 other virtual nodes.

Strategy	Probe Latency (failures)
Update-Up	$O(\log N)$
DOWN=d	$O(\log N - \log d)$
K-way hashing	$O(\log N)$
Supernode	$O(T_{switch} \log N)$

Table 6.6: Performance in terms of probe latency during failures. The latency refers to latencies for only successful probes. In the Supernode approach, T_{switch} corresponds to the timeout that is used to detect when a replica is down.

6.5 Discussion

Though K-way hashing and flexible API approaches are simpler than the Supernode approach, the probability of an access failure is greater than in the Supernode approach for a same replication factor. DHT maintenance overheads in the k-way

hashing and the flexible API approach during normal operation are lower than the Supernode approach because in the Supernode approach each node needs to track liveness of neighbors of k other nodes. A key advantage of k-way hashing and DOWN based approach is that applications can decide the individual replication factor for attributes they care about. But, the replication in the Supernode approach is done for all attributes, irrespective of the fault-tolerance required for an attriute; hence, this approach might incur higher overheads. Though the flexible API approach has higher probability of probe access failure compared to the K-way hashing, it incurs lower probe response latency during normal failure-free operation as the probes need to traverse fewer nodes in an aggregation tree.

The prototype currently supports `DOWN` parameter-based replication in space and the K-way hashing. Further investigation is underway to design and evaluate the Supernode approach for robustness in SDIMS.

Chapter 7

Shruti: Dynamic Adaptation

7.1 Introduction

Most existing aggregation systems use a static aggregation strategy that can perform well for some workloads but poorly for others, which prevents any system from being a truly general solution. For example, *read-dominated* attributes like *numCPUs* rarely change in value, whereas *write-dominated* attributes like *numProcesses* change quite often. An aggregation approach tuned for read-dominated attributes will consume high bandwidth when applied to write-dominated attributes. Conversely, an approach tuned for write-dominated attributes will suffer from unnecessary query latency or imprecision for read-dominated attributes. Also, the read-write access patterns may differ for different sets of nodes (spatial heterogeneity) and may change over time (temporal heterogeneity) requiring different aggregation strategies at different times and at different parts of the system [19, 91, 106].

Most aggregation systems have a single fixed aggregation mechanism. In Astrolabe [101], aggregation is performed at all levels in the aggregation tree and the aggregated values at each level are propagated to all nodes in the subtree on writes. Such strategy might incur high communication overheads when aggregating

attributes that change often. To limit the communication cost, Astrolabe throttles the rate at which information is propagated around in the system which might lead to unnecessary inconsistency in probe responses for attributes that rarely change. In Ganglia [41] and DHT based systems, aggregation is performed up to root on writes. In MDS-2 [36], no aggregation is performed on writes but the information is aggregated on reads.

SDIMS is the first aggregation framework that allows applications to control the aggregation aggressiveness in the system. SDIMS provides two knobs `UP` and `DOWN` that applications set at an aggregation function install time to denote how far up aggregation is performed in an aggregation tree and how far down the aggregate values at a level are propagated, respectively. SDIMS also allows applications to perform *continuous* probes to handle spatial and temporal heterogeneity. Though SDIMS exposes such flexibility to applications, it requires applications to know the read and write access patterns a priori to choose an appropriate strategy.

In this chapter, we present Shruti, a system built on SDIMS, that tracks reads and writes at different levels in an aggregation tree and dynamically decides for how many levels aggregation is performed and how far aggregate values are propagated downwards on writes. Shruti adapts the aggregation aggressiveness based on the observed read and write patterns aiming to optimize the overall communication costs (the sum of read and write message costs). We propose a *lease*-based mechanism in which a node grants a lease for an aggregate value to its parent or a child to inform that it will forward any changes to that aggregate value.

Our simulation results show that at any static globally uniform read-write operation ratio in the system, Shruti incurs similar cost as an optimal static `UP` and `DOWN` strategy. Also, in comparison to a single strategy across attributes of same type, Shruti through adapting to different strategies for different attributes achieves half-an-order magnitude lower average messages per operation. Our results

show that Shruti outperforms static aggregation strategies at almost all read-write ratios when there is a spatial heterogeneity in the access patterns. Finally we also show that Shruti efficiently adapts to temporal heterogeneity in the read and write patterns.

We describe the architecture of Shruti in Section 7.2, and then present our initial simulation results comparing Shruti to SDIMS static strategies in Section 7.3. Section 7.4 details related work and we summarize this chapter in Section 7.5.

7.2 Architecture

Shruti dynamically alters the propagation of aggregate values on writes in the system so that overall communication cost — number of messages incurred on probes and updates — is minimized. Shruti tracks updates and probes happening at all nodes in an aggregation tree and chooses an appropriate aggregation strategy based on that information. Shruti runs on each node (all intermediate and leaf nodes) in the aggregation tree and decides when that node will send any updates in the aggregate value to the node's parent and node's children. We propose a *lease*-based architecture where a node conveys its intention to keep forwarding any updates for an aggregate to another node through granting a lease for that aggregate. In the following sections, we present more details about the lease architecture, data structures that Shruti maintains to track updates and probes, how Shruti makes leasing decisions, and how Shruti handles reconfigurations.

7.2.1 Leases

In SDIMS, applications set `UP` and `DOWN` values at the install time of an aggregation function for an attribute type letting all nodes in the aggregation tree know how far to propagate the aggregate values on writes. All nodes need to know this information so that when a node gets a probe it can decide whether its local state includes the

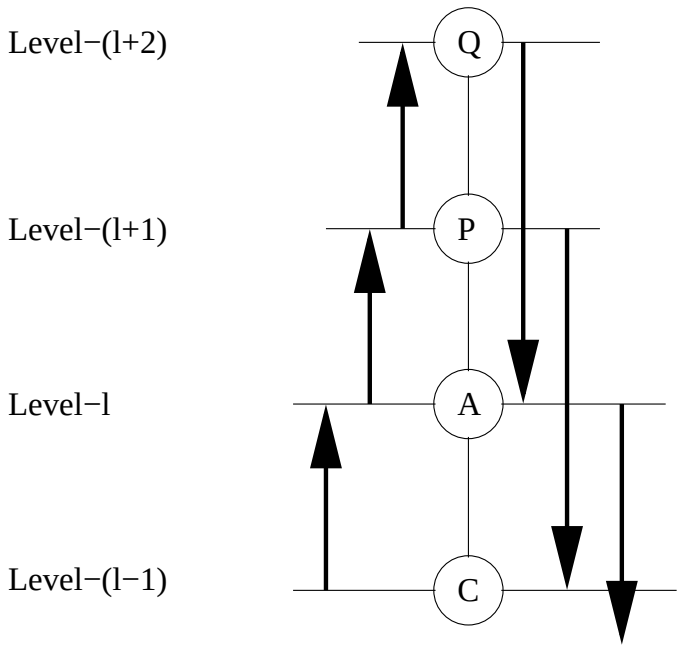


Figure 7.1: An example illustrating propagation of aggregate values in a part of an aggregation tree for the SDIMS static strategy `UP`=all, `DOWN`=2.

aggregate value needed for answering the probe. For example, consider an attribute with `UP`=all and `DOWN`=2. We illustrate how aggregates are propagated for a part of an aggregation tree in Figure 7.1. Since `UP` is set to all, the level-l node A knows that it has to propagate any updates to the level-l aggregate to its parent. Since `DOWN` is two, it propagates the level-l aggregate and also any updates it receives from its parent for the level-$(l+1)$ aggregate to its children. Similarly, this node's parent will propagate down any changes to either the level-$(l+1)$ aggregate or the level-$(l+2)$ aggregate. Now, if this node receives a probe for the level-$(l+2)$ aggregate, then it can respond without needing to further probe its parent. By knowing `UP` and `DOWN` a priori for an attribute, each node knows the rendezvous points between updates and probes for that attribute.

If we want to dynamically adapt the aggregation aggressiveness based on the

workload for an attribute, then nodes in the tree need to have enough information for responding to probes. Shruti employs a *lease*-based scheme to control the level of aggregation upon updates and to let each node know how to respond to probes. A node at level l that has leases from all its children can respond to probes for the level l aggregate without needing to probe its children. If it also has the lease from the parent for a level $l' > l$ aggregate, then it can respond to the probes for the level-l' aggregate without needing to send a probe to its parent. In Shruti, after a node A grants a lease for level l aggregate to another node B, then node A will send any changes to the level-l aggregate value until node B relinquishes that lease or node A revokes the lease.

Shruti does not exclusively work with leases to decide the propagation of aggregate values in an aggregation tree. Shruti also respects any application specified install time `UP` and `DOWN` parameters for an attribute type. Shruti uses the lease-based technique to extend the propagation of aggregate values in an aggregation tree beyond the amount of propagation an SDIMS static strategy allows. For example, if `UP`=all and `DOWN`=0 setting is chosen at install time of an attribute type, Shruti will always at least propagate changes in aggregate values up to the root of an aggregation tree. Shruti might further propagate aggregates down the tree to optimize communication bandwidth.

Invariants To respond correctly to probes, the leases that Shruti sets need to satisfy certain invariants. Consider the state of leases for an attribute in the Figure 7.2. We assume that the corresponding type is installed with `UP`=0 and `DOWN`=0. We show leases with thick arrows and denote the levels corresponding to leases in parenthesis next to the arrows. Note that in this state, node A cannot lease the level-1 aggregate to its parent as it does not have a level-0 lease from one of its children B. Consider a situation where node A indeed issued a lease for level-1 to its parent P. Since updates to the level-0 aggregate at node B are not propagated

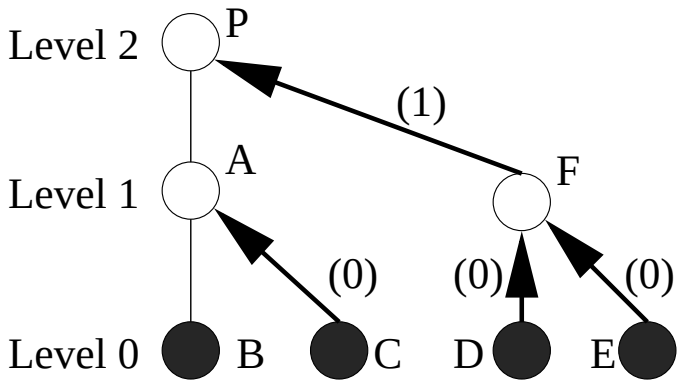


Figure 7.2: An example lease state in an aggregation tree. Thick arrows represent the leases and the corresponding lease levels are shown in parenthesis next to the arrows.

to node A, updates to the level-1 aggregate that node A propagates to its parent P will be incorrect. Any probe initiated at machines in the subtrees rooted at siblings of A (e.g., machine D) will receive an incorrect probe response. Another point to observe in Figure 7.2 is that node F cannot grant a lease for the level-2 aggregate to its children as it does not have a lease for the level-2 aggregate from its parent.

In the following, we present two invariants that Shruti maintains during setting up and tearing down leases to ensure correct rendezvous between probes and updates.

Invariant 1 *A node at level l can lease level l to either parent or a child only if it has leases for level $(l-1)$ from all its children or if* `UP` *for the corresponding attribute type is $\geq l$.*

Invariant 2 *A node at level l can lease level $l' > l$ to a child only if it has the lease for level l' from its parent, if it has no parent, or if* `UP` $\geq l'$ *and* `DOWN` $\geq (l'-l)$ *holds.*

The above invariants also imply the following two conditions regarding when a node can relinquish a lease it has obtained from a neighbor of that node.

1. A node can relinquish a lease for a level l' aggregate received from its parent only if the level l' aggregate is not leased down to any child.

2. A node at level l can relinquish a lease acquired from child only if the level l aggregate is not leased to the parent or to any child.

7.2.2 Leasing Policy

Shruti keeps track of the updates and probes at each node and dynamically sets up and tears down leases between nodes to optimize overall communication costs (sum of read and write costs). Briefly, it is useful for a node A to grant a lease for a level l aggregate to another node B only if the number of messages that A and B exchange on probes when the lease is not granted is greater than the number of update messages that A has to forward to B after granting the lease. Note that each probe causes exchange of two messages between A and B — one for the query and one for a response; whereas each update causes only one message between two nodes. So a lease can be granted if we expect the number of probes to be even half of the number of updates. When the number of updates a node receives when a lease is set goes beyond two times the number of probes, then it is better to remove the lease.

We exploit dynamic adaptation of the aggregation strategies not only to optimize overall communication costs but also to trade-off bandwidth with probe response latencies. For example, if leases are set aggressively (say even when we expect probes to be far less than the half of the number of updates) but removed lazily (say remove only when the number of updates is four times more than the number of probes), then the average probe response times will be smaller but at the cost of increased communication costs. Shruti provides two knobs for the applications to control the lease aggressiveness - one knob to decide how leases are granted and another to decide how they are removed. Overall, Shruti supports and extends the

flexibility provided by the static `UP` and `DOWN` SDIMS strategies.

In the following, we first describe the data structures that Shruti maintains on each node, and then describe the knobs that Shruti exposes to the applications to set a leasing policy.

Data Structures Shruti on each node maintains several logs for tracking updates and probes happening at that node. On a level-l node in an aggregation tree, Shruti maintains the following logs with timestamps:

- **LocalHistory** On each node, Shruti maintains timestamps of the most recent probe and update for each level from either its parent or any of its children. On a level l node, the updates and probes for level $l' \geq l$ are accounted for the respective level l'. But updates received from the node's children for level $(l-1)$ aggregate are accounted as updates for level l, since those updates affect the level-l aggregate.

- **NeighborHistory** This data structure on a node is used to track all probes and updates for all levels from each neighbor of a node (parent and children). A node maintains neighbor history for a level l' aggregate with respect to another node B (aka history of updates and probes from node B for level l') only if node A can grant the level-l lease or has received the level-l' lease from node B.

- **LeasesGranted** and **LeasesReceived** We maintain the leases a node acquired from and leases granted to either its parent or its children in these data structures. These data structures are indexed on the neighbor.

Granting and Relinquishing leases Shruti at a node uses the history of updates and probes from another node to predict the future update-probe patterns from that node. We assume that the patterns observed in the near past reflect the near future

Algorithm 5 OnProbe(*fromNode, level*)

```
1: timestamp ← current time
2: Add (probe, timestamp) to LocalHistory[level]
3: /* Check if invariants allow us to grant a lease for this level to any node */
4: if canLease(level) then
5:    Add (probe, timestamp) to NeighborHistory[fromNode][level]
6:    lastUpdateTime ← timestamp of the most recent update from LocalHis-
      tory[level]
7:    numProbes ← numProbes in NeighborHistory[fromNode][level] with times-
      tamp > lastUpdateTime
8:    if numProbes >= m then
9:       Grant lease for level aggregate to fromNode
10:   end if
11: else
12:   clear all probes from NeighborHistory[fromNode][level]
13: end if
```

Algorithm 6 canLease(*level*)

```
1: myLevel ← this node's level
2: if level > myLevel then
3:    if UP ≥ level AND DOWN ≥ (level−myLevel) then
4:       return true
5:    else if level ∈ LeasesReceived(parent) then
6:       return true
7:    else
8:       return false
9:    end if
10: else
11:   if UP ≥ myLevel then
12:      return true
13:   else
14:      for each child C ∈ children do
15:         if (level−1) ∉ LeasesReceived(C) then
16:            return false
17:         end if
18:      end for
19:      return true
20:   end if
21: end if
```

Algorithm 7 OnUpdate(*fromNode, level*)

```
1: timestamp ← current time
2: Add (update, timestamp) to LocalHistory[level]
3: /* Check if invariants allow us to relinquish a lease for this level. */
4: if level ∈ LeasedFrom(fromNode) AND canRelinquish(level) then
5:    Add (update, timestamp) to NeighborHistory[fromNode][level]
6:    checkLevel leftarrow (level < myLevel)?myLevel:level
7:    /* Note that children of a node at level l send updates for their level (l − 1)
      and those updates affect level l aggregate */
8:    lastProbeTime ← timestamp of the recent most probe from LocalHis-
      tory[checkLevel]
9:    numUpdates ← number of updates in NeighborHistory[fromNode][level] with
      timestamp > lastProbeTime
10:   if numUpdates >= k then
11:      Relinquish lease for level aggregate to fromNode
12:   end if
13: else
14:   clear all updates from NeighborHistory[fromNode][level]
15: end if
```

Algorithm 8 canRelinquish(*level*)

```
1: myLevel ← this node's level
2: if level < myLevel then
3:    level ← myLevel
4: end if
5: if level ∈ LeasesGranted(parent) then
6:    return false
7: end if
8: for each child C ∈ children do
9:    if level ∈ LeasesGranted(C) then
10:      return false
11:   end if
12: end for
13: return true
```

behavior. When invariants for granting a lease for a level l aggregate are satisfied at a node A, we use the following general rule to decide whether to grant a level-l lease to a node B or not — grant the lease if `m` probes are received from node B while no updates happen to the level-l aggregate. In Algorithm 5, we present pseudo-code

for the actions performed by Shruti on receiving a probe from another node. The pseudo-code for checking whether granting a lease violates any invariant is shown in Algorithm 6.

Similar to the rule for setting a lease, we use the following rule for relinquishing a level-l' lease granted by a parent P to node A — relinquish the lease if `k` updates are received from the parent P while no probes are received for the level-l' aggregate. Similarly a level-$(l-1)$ lease granted by a child C is relinquished if `k` updates are received from the child C while no probes are received for the level-l aggregate. Note that updates for level-$(l-1)$ from a node's children affect the value of level-l aggregate at the node. The pseudo-code for Shruti's actions on receiving an update is shown in Algorithm 7. We present pseudo-code checking whether relinquishing a lease violates any invariant in Algorithm 8.

Consider an example with `k`=2 and `m`=1. A node A that can grant a level-l lease to node B grants that lease to B as soon as it gets a probe from B. And node B relinquishes that lease if it gets two updates from node A for that aggregate while not receiving any probes for that aggregate from any other node.

The two knobs `k` and `m` control how aggressively leases are set and how aggressively they are removed. A setting where `k` is about twice the value of `m` performs optimally in terms of number of messages. A large value for `k` and a small value for `m` cause leases to be set aggressively but to be removed lazily leading to better probe response times but at increased bandwidth.

7.2.3 Default Lease State

To be efficient and be scalable with *sparse attributes*, Shruti on each node at level-l for an attribute starts with a state where it assumes that it has level-$(l-1)$ leases from all its children and has granted a lease for the level-l aggregate to its parent.

Sparse attributes are attributes that are of interest to only few nodes in

the system and only those nodes perform write or read accesses to that attribute. In most practical systems with a large number of attributes, all nodes will likely not be interested in all attributes. For example, reads and writes to an attribute corresponding to a multicast session with small membership will generally be done only by its members.

If Shruti starts in a state where no leases are granted for an attribute, the first read operation will incur $2.N$ messages in an N node network, as the read has to collect information from all nodes in the system. Thus, even for sparse attributes that are of interest to only a handful of nodes, all nodes in the system are touched by the first few reads for that attribute till leases are set. Once leases are set those uninterested nodes do not receive any more messages regarding the attribute. But explicitly setting leases implies that all nodes have to maintain some information about all attributes whether they are interested in that attribute or not. Hence the default initial state undermines the scalability of the SDIMS to large numbers of attributes.

Instead, by starting in a lease state in which Shruti on each node for all attributes assumes that it has leases granted from all its children and has granted a lease for the local aggregate to its parent, the unnecessary leasing information need not be explicitly maintained. Hence only nodes that are interested in an attribute and nodes helping these interested nodes in aggregation of the attribute will ever maintain any explicit information about the leases (when leases are relinquished or when leases are further granted down to children), thus achieving scalability with sparse attributes.

7.2.4 Reconfigurations

Reconfigurations will be a norm in any large distributed system and in the face of reconfigurations, the invariants for leases might not continue to hold at one or more

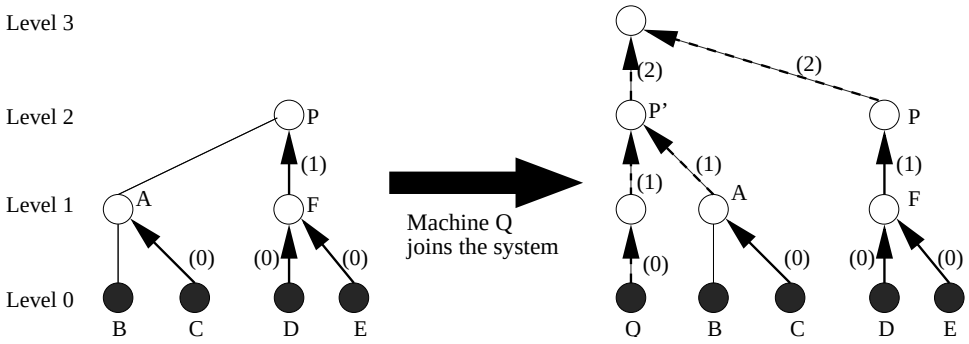


Figure 7.3: Invariant violation on reconfiguration - machine Q joins the system. The dotted arrows represents the leases assumed by default for newly created nodes in the aggregation tree. Note that the Invariant 1 is violated at node A, which has a lease granted to its parent while it does not have a lease from one its children B.

nodes in the system. Invariant 1 is violated when a node at level l in an aggregation tree acquires a new parent when it does not have a lease for level $(l-1)$ from one of its children. For example, consider an aggregation tree in a four machine system shown on the left in Figure 7.3. In this aggregation tree, all invariants are satisfied. Suppose machine Q joined this system and the aggregation tree is modified as shown on the right in the figure. As discussed in the previous section, new nodes start with a lease state where each node at level l assumes that it has leases for level $(l-1)$ from all its children. The dotted arrows show the default leases assumed when machine Q joins the system. At node A, Invariant 1 is violated. The parent of node A, node P', has lease for level 1 from node A while node A does not have a lease for level 0 from one of its children.

Similar to violation of Invariant 1, Invariant 2 might be violated during reconfigurations. We show an example case where such violation occurs in Figure 7.4. Note that node A before reconfiguration has a lease for level 2 from its parent P which it granted further down to one of its children B. When it acquired a new parent P', node A no longer has a level-2 from its parent but has an outstanding

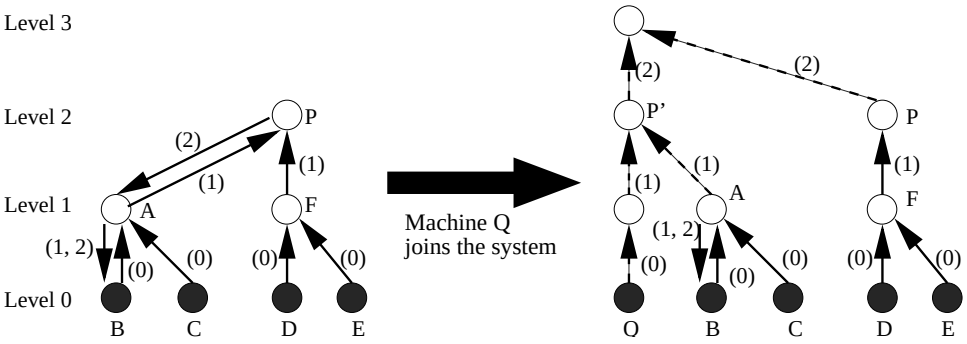


Figure 7.4: Invariant violation on reconfiguration - machine Q joins the system. Note that Invariant 2 is violated at node A, which granted a lease for level 2 to its child while it does not have a lease for level 2 from its parent.

level-2 lease to its child, violating Invariant 2.

In the face of reconfigurations, the goal of Shruti is to revert to a state conforming to all invariants. Observe that invariant violations occur only when a node gets a new parent. Acquiring a new child or losing an existing child does not affect the consistency of leases at a node. At each node where an invariant violation occurs due to a new parent, Shruti revokes leases that violate invariants. For example, when Invariant 1 is violated at a node due to acquiring a new parent, that node revokes the lease to its parent. In the example shown in Figure 7.3, node A revokes the level-1 lease to its new parent P'. Similarly, when Invariant 2 is violated at a node due to getting a new parent, that node revokes leases it granted to its children that violate invariants. For example, node A in Figure 7.4 revokes its level-2 lease to node B after it gets new parent P'.

A node that receives revocation of a lease from its parent or one of its children might have to further revoke some leases that this node granted to other nodes. For instance, when a node receives revocation of a level-l lease from its parent, it has to further revoke any level-l leases it has granted to its children. Also, when a node at level-l receives revocation of level-$(l-1)$ lease from one of its children, then it

should revoke any level-l leases it has granted to its parent or its children.

Note that while lease revocal is in progress, some of the probes might receive incorrect responses. But once the system becomes stable (no machine or network failure events or no new machine join events), Shruti on each node attains a state satisfying invariants through revoking zero or more leases it has granted to other nodes. Once invariants are satisfied in the lease state, all probes receive correct responses.

7.3 Evaluation

We present our experimental results on Shruti comparing its performance to static up and down strategies in SDIMS for a wide range of read-to-write ratios and with spatial and temporal heterogeneity in the access patterns. We measure two metrics — communication cost and operation latency. Our initial evaluation shows that Shruti (i) adapts to the access patterns and approximates the optimal SDIMS static strategy for a static and globally uniform read-write ratio, (ii) adapts to heterogeneity in the access patterns across nodes to outperform any single SDIMS static strategy (spatial heterogeneity), and (iii) quickly adapts to changing access patterns (temporal heterogeneity).

Due to the limitation of memory on our simulating machines, in all of the following experiments, we use a 512-node system. We compensate the small number of nodes by using 1-bit correction in each routing hop instead of default 4-bit correction in FreePastry; thus increasing the depth of the DHT. We simulate 50000 operations for each strategy in each experiment. For Shruti experiments, where not specified, we use `UP`=0 and `DOWN`=0.

In all of the following experiments, we use a simple summation operation as the aggregation function. An update at a node for an attribute increments the previous value for the attribute at that node. We initially update each attribute at

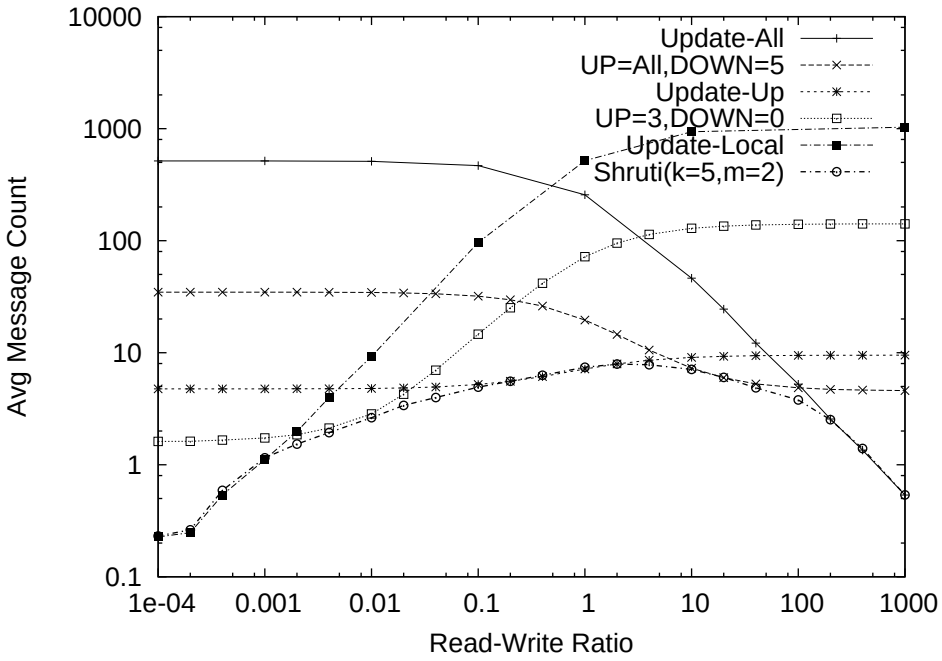


Figure 7.5: Average message cost per operation in Shruti compared to different static up and down settings in SDIMS for a wide range of read-to-write ratios.

each node with a value of one. All probes are for the global aggregate value. Note that SDIMS static strategies with `UP`=all and `DOWN`> 0 propagates down aggregate values at all levels to `DOWN` number of levels. But Shruti adapting to probes will propagate only the global aggregate values. So to be fair to the SDIMS static strategies, we do not count messages corresponding to the downward propagation of intermediate aggregate values.

Single attribute, uniform read-write ratios across nodes In this first set of experiments, we consider a single attribute and uniform read-write ratios across all nodes. In Figure 7.5, we plot the measured average message cost per operation in Shruti compared to different static up and down strategies in SDIMS for a wide range of read-write ratios. We use values of 5 and 2 for k and m (recall that k and

m determine the aggressiveness with which leases are set and removed as explained in Section 7.2.2, respectively, in Shruti. Note that at any read-write ratio, Shruti approximates the behavior of an optimal up-down SDIMS strategy at that ratio.

In Figure 7.6, we plot the average latency per operation (both reads and writes considered) in Shruti and in SDIMS with static up-down strategies. Note that whereas any static strategy that behaves well at some read-to-write ratios incurs a high latency at other read-to-write ratios, Shruti performs well at all read-to-write ratios. For example, although the Update-all strategy performs optimally in terms of operation latency for read-to-write ratios greater than one, it incurs a high communication cost for read-write ratios less than one when compared to Shruti (Figure 7.5).

In Figure 7.7, we plot the average probe response latencies with different read-to-write ratios for static up-down strategies and Shruti. We assume that each overlay hop has unit latency. Note that Shruti adapts to reduce overall communication bandwidth and hence incurs different latencies at different read-write ratios. All static strategies have fixed average probe response latencies.

Varying k and m in Shruti In Figure 7.8, we plot the average message cost observed for different values of k and m in Shruti while varying the read-to-write ratios. As expected, for large values of k and small values of m, the system adapts quickly to probes but slowly to writes; hence, they perform better at large read-write ratios but suffer at small read-write ratios. In Figure 7.9, we compare the average read latency for these different strategies. Observe that the read-favoring higher k compared to m strategies result in smaller read latencies. We conclude two key points from these set of results: (1) k=5 and m=2 or k=5 and m=3 is a good default value set for k and m as Shruti performs better with these values than any other set of values for k and m and (2) Applications that intend to reduce the response latency at the cost of higher bandwidth can use a more aggressive leasing policy by

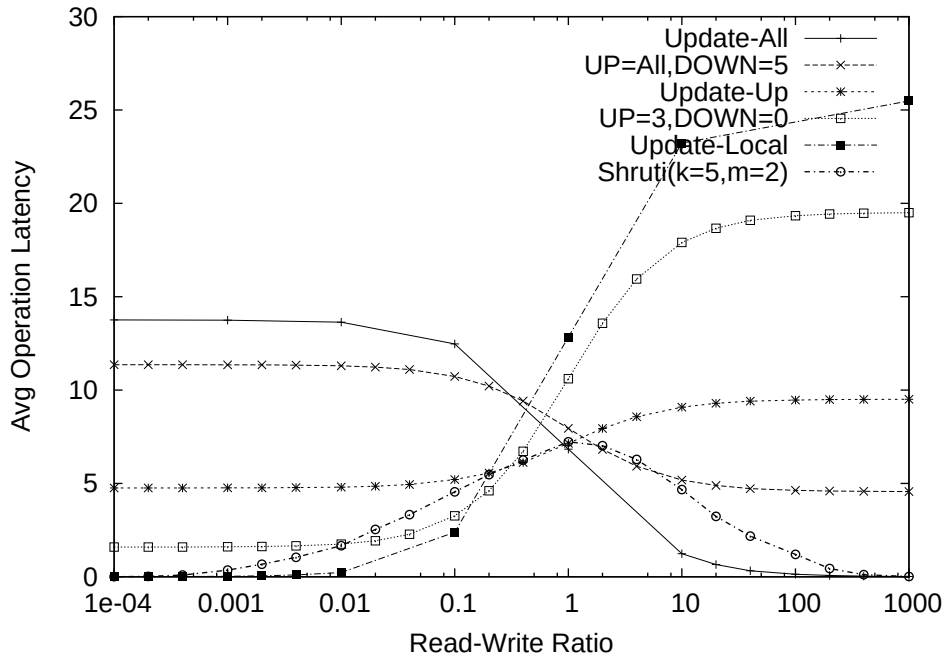


Figure 7.6: Average latency per operation in Shruti compared to different static up and down settings in SDIMS for a wide range of read-to-write ratios. All overlay links have one unit latency.

setting a high value for k and a small value for m.

Shruti in tandem with an application specified static strategy As explained in Section 7.2, Shruti abides by the install time application specified `UP` and `DOWN` parameters. By default, we assume that those parameters are set to zero. When specified, Shruti performs aggregate value propagation at least as much specified according to those parameters but might propagate more to decrease the overall communication costs. In Figures 7.10 and 7.11, we plot the performance of Shruti with difference `UP` and `DOWN` parameters. This hybrid approach allows applications to specify a required latency guarantees and allows Shruti to minimize communication costs while satisfying those response latency guarantees. Shruti with `UP`=0 and

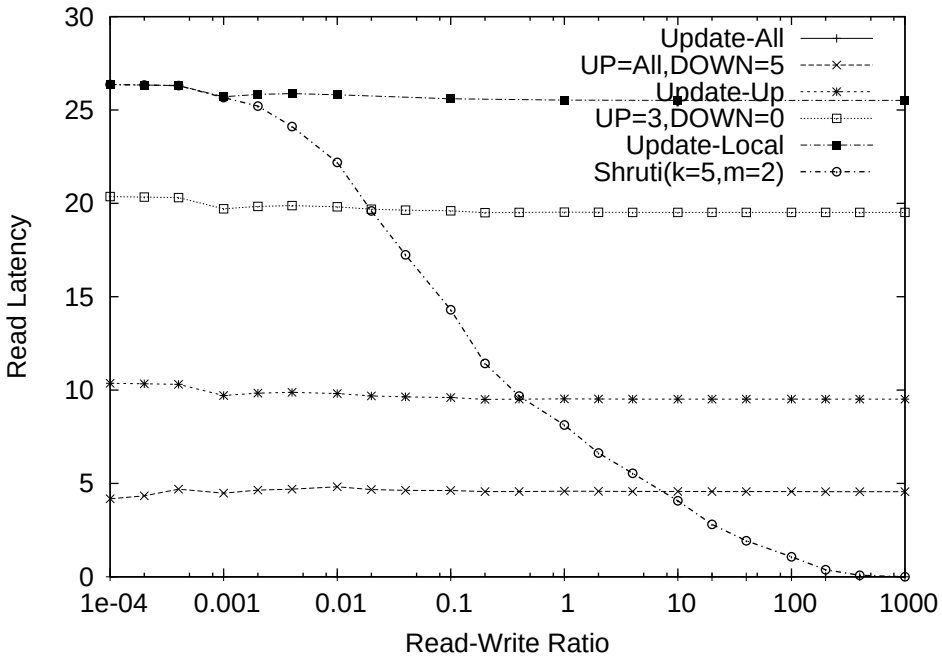


Figure 7.7: Average probe response latency in Shruti compared to different static up and down settings in SDIMS for a wide range of read-to-write ratios. All overlay links have one unit latency.

`DOWN`=0 incurs lower or similar communication cost compared to any static SDIMS strategy (Figure 7.5); but it incurs a higher read latency at small read-to-write ratios when compared to static SDIMS strategies with `UP`>0. But Shruti with same `UP` and `DOWN` values specified as a static SDIMS strategy performs better or similar both in terms of probe latency and communication costs.

Multiple attributes, Zipf-like distribution in reads Studies have shown that web accesses and P2P queries follow Zipf-like distribution [19, 91] with respect to the objects. Here, we study the performance of Shruti when reads to attributes follow Zipf-like distributions [19] (the ith popular attribute gets C/α^i fraction of reads) with $\alpha = 1.3$. The write operations are assigned to different attributes in a

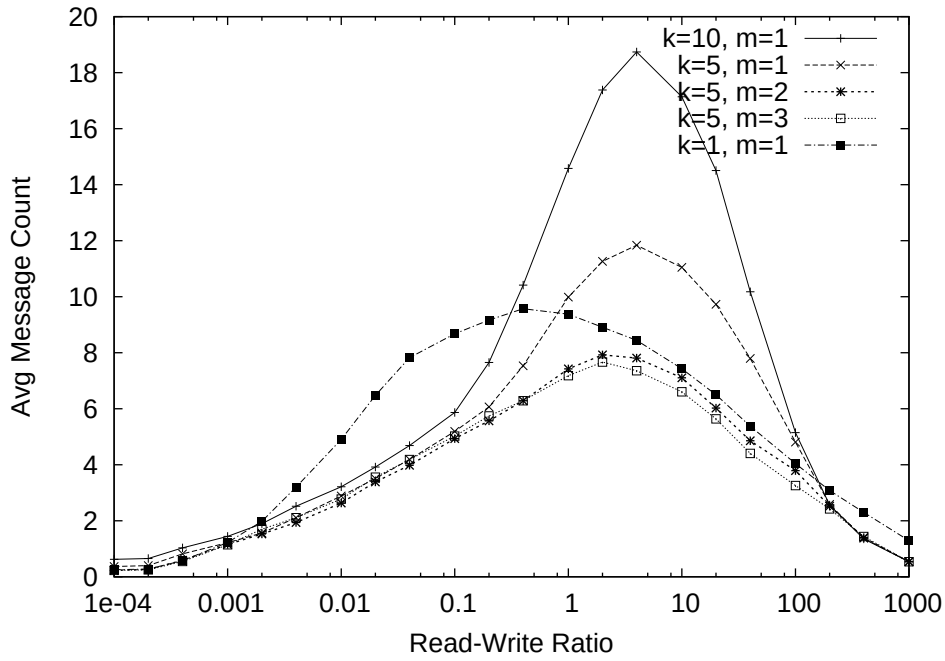


Figure 7.8: Average message cost per operation in Shruti for different values of k and m for a wide range of read-to-write ratios.

uniform way. We simulate 100 attributes of the same type and a global average read-to-write ratio of 100. In Figure 7.12, we present the average number of messages per operation and average read latency incurred by Shruti compared to a set of SDIMS strategies with different static same up and down values across all attributes. Clearly, Shruti achieves both least communication cost and the smallest average read response time through adapting aggregation aggressiveness separately for each individual attribute.

Spatial heterogeneity The distribution of reads and writes for an attribute will not be uniform across nodes in a real system. For example, for an attribute corresponding to a multicast session, typically only members of that multicast session perform most read and write operations. We simulate a single attribute opera-

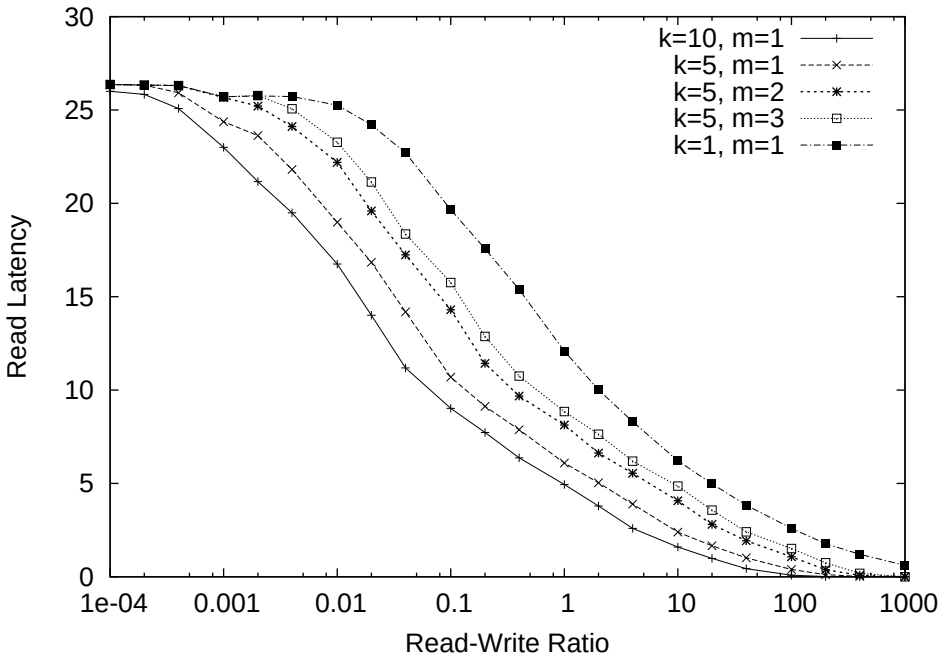


Figure 7.9: Average read latency in Shruti for different values of k and m for a wide range of read-to-write ratios.

tion rates at nodes following a Zipf-like distribution with $\alpha = 1.3$. In Figures 7.13 and 7.14, we plot the average number of messages per operation and average operation latency incurred in Shruti in comparison to that incurred by a set of SDIMS strategies for different read-to-write ratios. Note that Shruti achieves lower communication costs compared to the first set of results in Figure 7.5 as it exploits the spatial heterogeneity to set leases such that updates and probes are propagated to only nodes interested in that attribute.

Temporal heterogeneity The read-write ratio for attributes change over time as attributes become popular and then as popularity fades. In this experiment, we change the read-write ratio every 20000 operations and measure the performance of Shruti and a SDIMS Update-UP (`UP`=all and `DOWN`=0) strategy. In Figure 7.15,

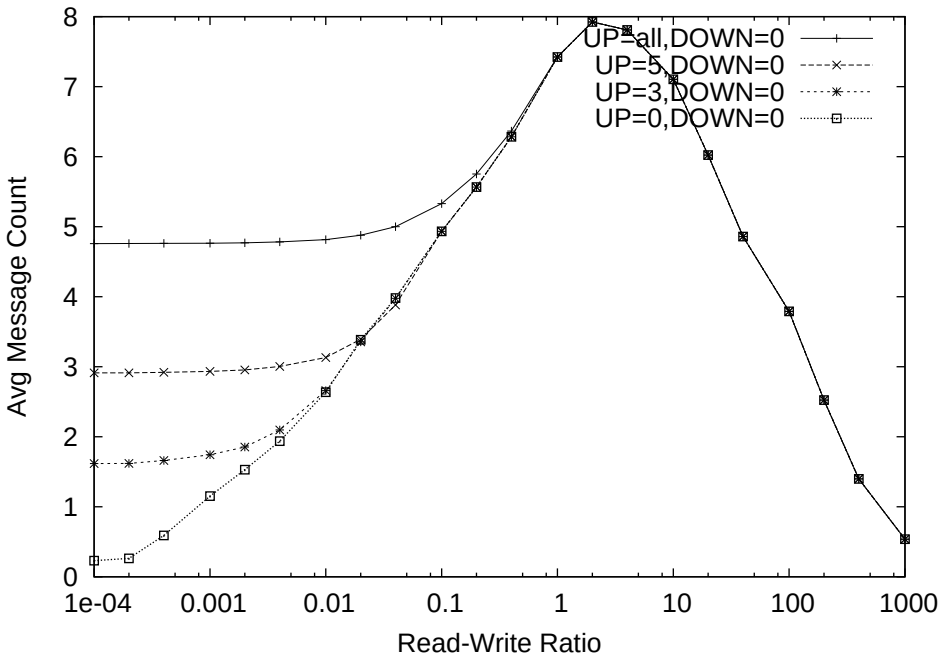


Figure 7.10: Average message cost per operation in Shruti with different UP and DOWN parameters for a wide range of read-to-write ratios.

we show the average number of messages incurred per read and write for these two mechanisms. We change the read-to-write ratio from 0.01 to 100 after first 20000 operations and then revert back to 0.01 after another 20000 operations. Note that whereas the SDIMS static scheme incurs a similar number of messages on every operation irrespective of the read-write-ratio, Shruti adapts to the observed read-write ratios to incur lower communication costs. In this test, SDIMS Update-Up incurs average message cost per operation of 4.6 during the first and third phases and a cost of 9.1 during the second phase; Shruti incurs an average message cost of 2.8 during the first and third phase and a cost of 3.8 during the second phase. Overall, the average message cost per operation is 6.11 in case of the SDIMS Update-Up strategy compared to a cost of 3.23 in case of Shruti.

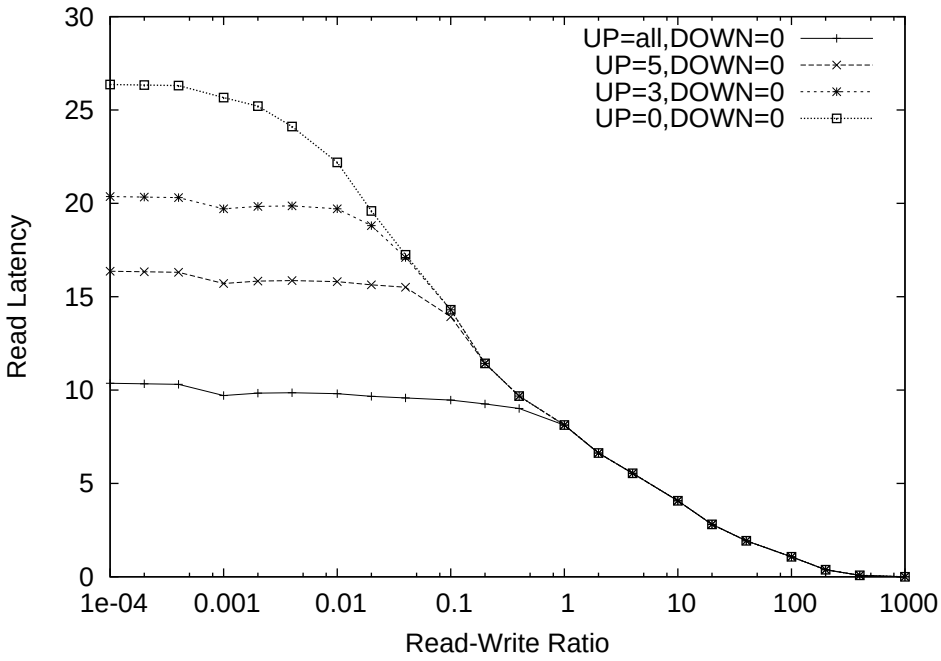


Figure 7.11: Average read latency in Shruti with different UP and DOWN parameters for a wide range of read-to-write ratios.

Reconfigurations As described in Section 7.2.4, when a virtual node in an aggregation tree loses its parent or obtains a new parent during a reconfiguration event, then the virtual node might revoke some of the leases it has granted to other nodes so that invariants are satisfied in the aggregation tree. As leases are revoked due to the reconfiguration events, reads might experience increased latency. We demonstrate this effect on reads due to reconfigurations in the following simulation experiment. With Shruti running on 1024 nodes, we first perform several write operations for an attribute at all nodes so that all leases that are set by default (refer to Section 7.2.3) are removed. We then perform several read operations from a single machine so that leases are set from all nodes till the root for that aggregation tree and from the root towards the probing leaf node. Then we kill the root node so that leases are revoked and again we perform several reads from the chosen leaf node. In Figure 7.16, we

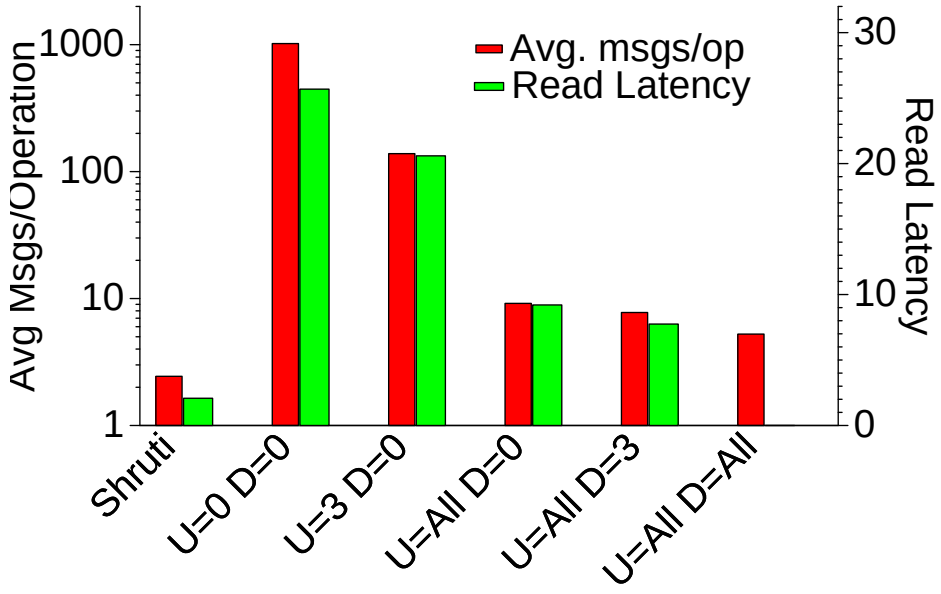


Figure 7.12: Average messages per operation and read latency observed with Shruti (k=5, m=2) and a set of SDIMS static up and down strategies. These metrics are computed across 100 different attributes where reads follow a Zipf-like distribution with $\alpha = 1.3$.

plot the number of simulation rounds (denoting the latency) of each probe operation. After sixty read operations, we kill the root node in the aggregation tree for this attribute. This event revokes all downward leases from the root node to the probing node. Hence the following reads suffer from higher latency.

7.4 Related Work

Existing aggregation systems use a static aggregation strategy that can perform well for some workloads but might perform poorly for others. Astrolabe [101] employs an *Update-All* type aggregation mechanism, DHTs and DHT based systems [66, 77, 80, 86, 94, 118] use an *Update-Up* mechanism, and MDS in Globus tool kit [30] uses an *Update-None* mechanism.

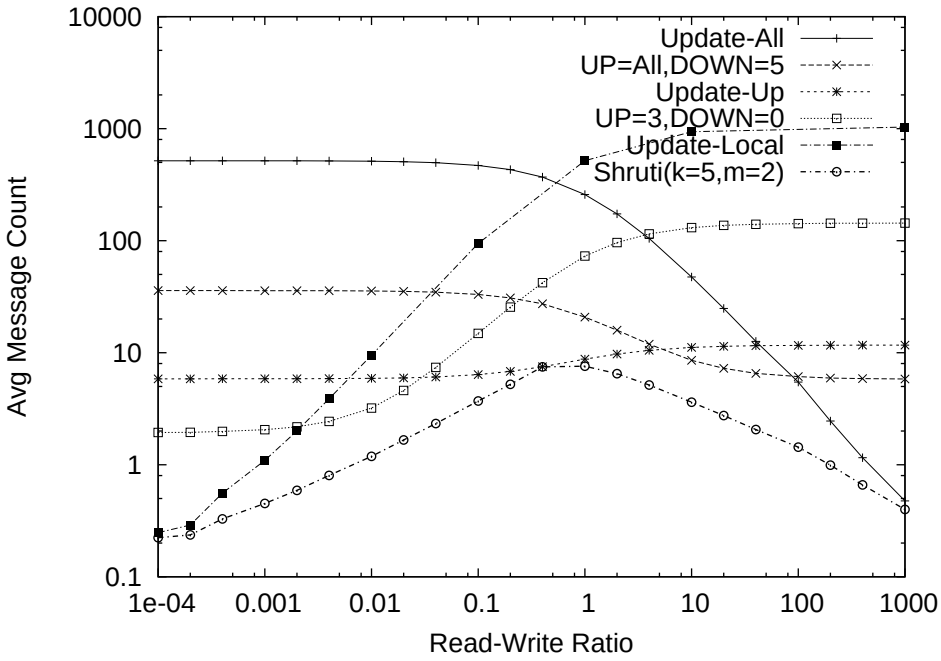


Figure 7.13: Spatial heterogeneity: Average number of messages per operation in Shruti compared to a set of SDIMS strategies for a single attribute where the operation rates across nodes follow a Zipf-like distribution with $\alpha = 1.3$.

The Controlled Update Propagation (CUP) protocol by Roussopoulos et al. [85] and Overlook [98], a name service system, are closely related to Shruti. CUP addresses a similar problem in the context of updating cached results of *get* operations in DHTs and Overlook replicates name service content along a tree to reduce lookup latency of DNS resolve queries. Though CUP, Overlook, and Shruti share the idea of using lease-based techniques, they differ in the design choices leading to different tradeoffs. First, CUP and Overlook only consider replicating the root content at other nodes; they build upon the DHT architecture and hence assume that aggregation is performed up to the root on writes. So they dynamically control the propagation of updates only downwards where as Shruti controls update propagation even towards the root. For Overlook, which is a naming service where

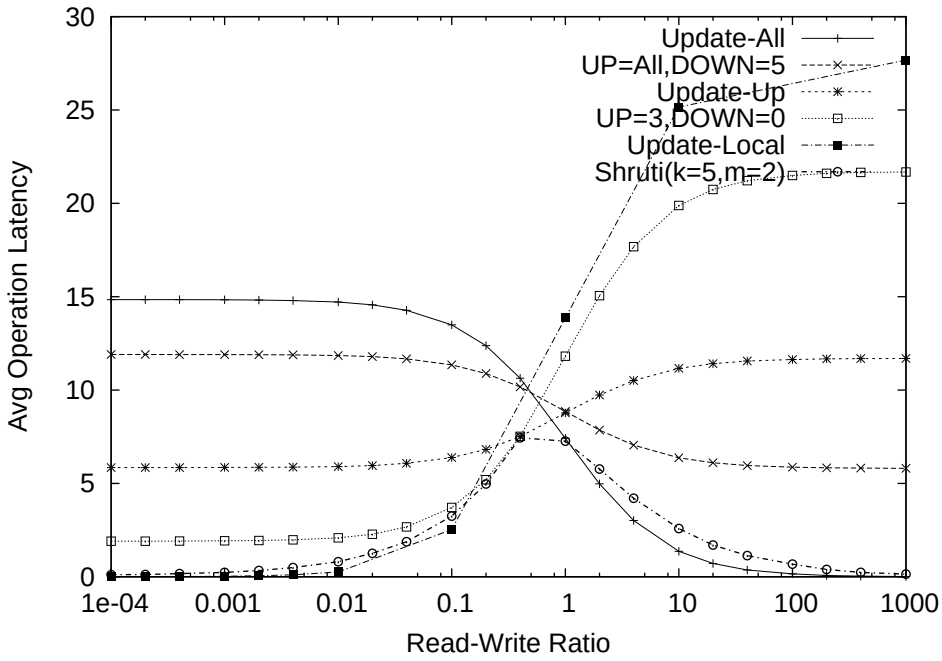


Figure 7.14: Spatial heterogeneity: Average operation latency in Shruti compared to a set of SDIMS strategies for a single attribute where the operation rates across nodes follow a Zipf-like distribution with $\alpha = 1.3$.

the number of updates will be far less than the number of probes for any entry, such downward only propagation control is appropriate. In SDIMS, we consider different attributes with different behaviors. For an attribute that is updated by only a single node and no other node probes for that attribute, it is inefficient to even aggregate the data up to the root in the corresponding aggregation tree. Second, the maintenance overheads are different in these three systems. In CUP, each replicated object at a node expires unless refreshed by the parent for that object. So, the maintenance overhead is on the order of the number of objects. In Overlook and Shruti, a replicated object at a node expires if the parent that gave the lease fails. So, lease maintenance overhead involves tracking liveness of the parents of a node; hence an overhead in the order of the number of parents a node has. Whereas a

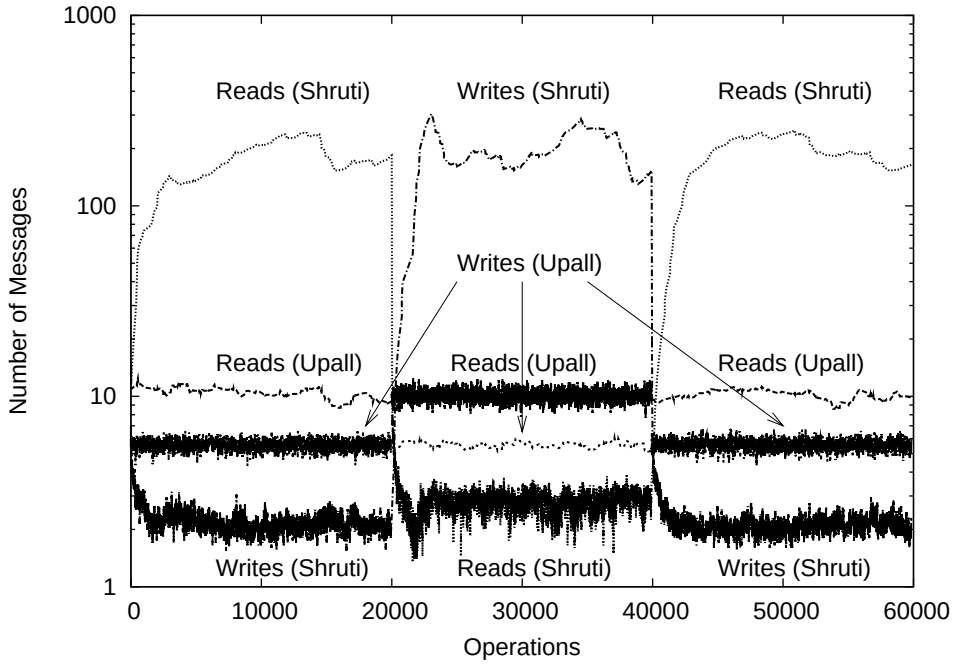


Figure 7.15: Temporal heterogeneity: The number of messages incurred on read and write operations in Shruti compared to a SDIMS strategy of `UP`=all and `DOWN`=0. We change the read-to-write ratio from 0.01 to 100 after 20000 operations and revert back to the same ratio after another 20000 operations

node might have O(N) parents in the worst case in Overlook, each node in Shruti has to track only O($\log N$) other nodes, where N is the number of nodes in the system.

Other closely related projects are Beehive by Rama et al. [79] and SCAN by Chen et al [27]. In Beehive, no updates are considered and the goal is to place a minimum number of replicas such that all queries are satisfied with a constant communication cost, assuming queries follow a Zipf distribution. Chen et al. solve a similar problem of placing a minimum number of replicas while satisfying client QoS requirements and server constraints. Cohen et al. study replication strategies in unstructured P2P networks [28].

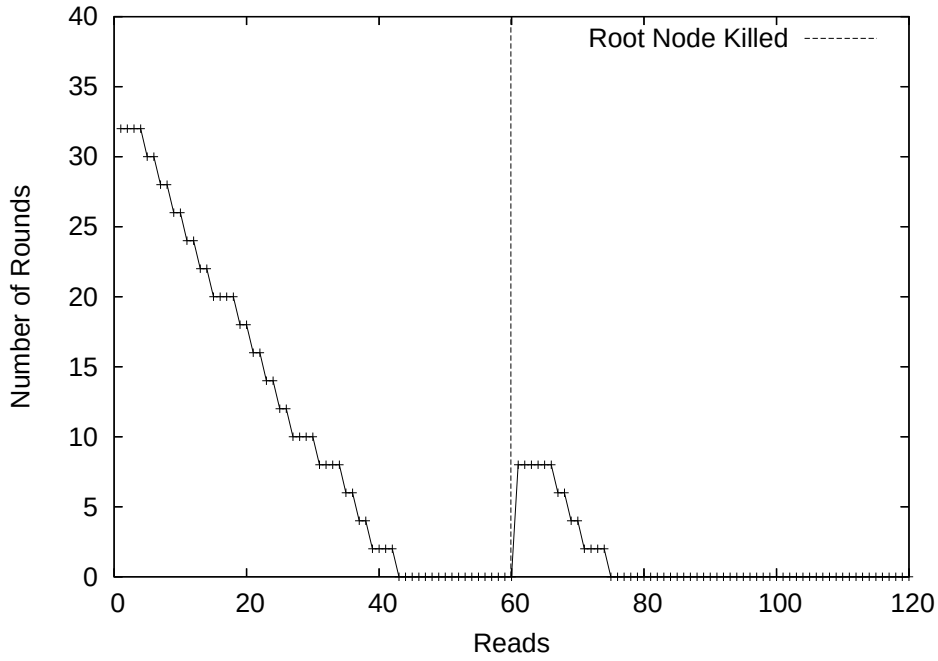


Figure 7.16: Shruti in the face of reconfigurations: The number of simulation rounds taken for each read operation. After sixty read operations, we kill the root node of the aggregation tree.

Lease-based techniques are employed in many distributed systems like replicated file systems [42] and web caching and replication [60, 33, 70, 114, 113]. All web replication research consider the case where updates to objects happen at a single server. Yin et al. propose volume leases [114, 113] where a server issues a volume lease with a short timeout period and object leases with a long timeout period. This helps in reducing the communication from the server to the replicas while ensuring strong consistency guarantees. In Shruti, each node pings its neighbors frequently to check their liveness (similar to short volume lease timeout) and consider all leases that a neighbor issued to be valid as long as the neighbor is alive or the lease is either relinquished or revoked (similar to long object lease timeout).

7.5 Summary

Though SDIMS exposes a wide-range of aggregation strategies, it requires that applications have knowledge of the read/write load patterns a priori. In this chapter, we have presented Shruti, a subsystem in SDIMS, that tracks reads and writes at individual nodes in an aggregation tree and adjusts the aggregation strategy to minimize the communication costs. Shruti uses a lease-based mechanism in which a node grants a lease for an aggregate value to its parent or a child to inform that it will forward any changes to that aggregate value. We have also presented the invariants that Shruti maintains in setting and removing leases that guarantee correct responses for probes. On reconfigurations, Shruti might revoke leases that violate the invariants to reach a state satisfying all invariants.

Through extensive simulations, we show that at any static globally uniform read-write operation ratio in the system, Shruti incurs similar cost as an optimal static `UP` and `DOWN` strategy. Also, in comparison to a single strategy across attributes of the same type, Shruti, through adapting to different strategies for different attributes, achieves half an order magnitude lower average messages per operation. Our results show that Shruti outperforms static aggregation strategies at almost all read-write ratios when there is a spatial heterogeneity in the access patterns. Finally we also show that Shruti efficiently adapts to temporal heterogeneity in read and write patterns.

Chapter 8

Prototype

The design of our SDIMS system comprises of three layers, as shown in Figure 8.1. The Autonomous DHT (ADHT) layer implements ADHT algorithms described in Chapter 5. The Aggregation Management Layer (AML) supports the flexible API described in Chapter 3. The AML maintains attribute tuples, performs aggregations, stores and propagates aggregate values on aggregation trees. The DHT-AML Interface layer is the glue between the DHT and the AML layer that obtains information about state of the overlay from the ADHT layer and uses that information to maintain aggregation trees as described in Chapter 4. In the present chapter, we briefly describe the DHT-AML Interface layer and then describe the internal state and operation of the AML layer in the system.

We refer to a store of (attribute type, attribute name, value) tuples as a Management Information Base or MIB, following the terminology from Astrolabe [101] and SNMP [92]. We refer to an (attribute type, attribute name) tuple as an *attribute key*.

In Figure 8.2, we show aggregation trees corresponding to keys 000 and 111 for an eight node system example from Table 4.1 in Chapter 4. Note that level-l virtual nodes corresponding to different aggregation trees hosted on a same machine

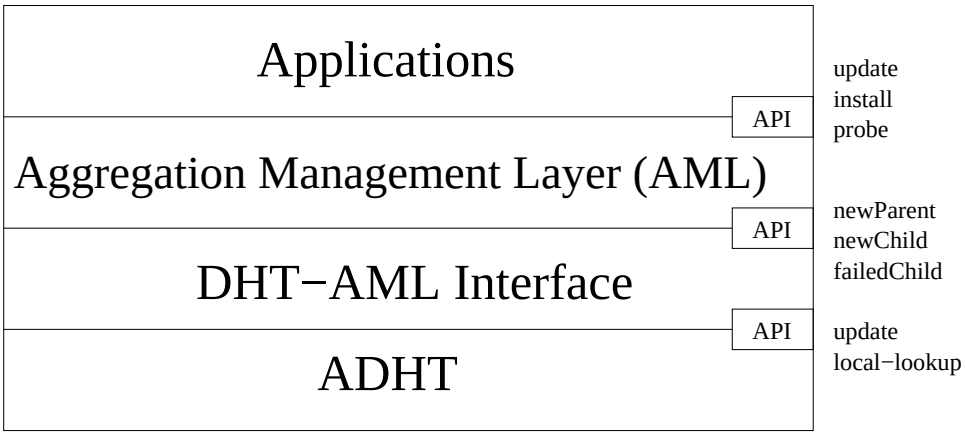


Figure 8.1: Layered SDIMS prototype design and interfaces.

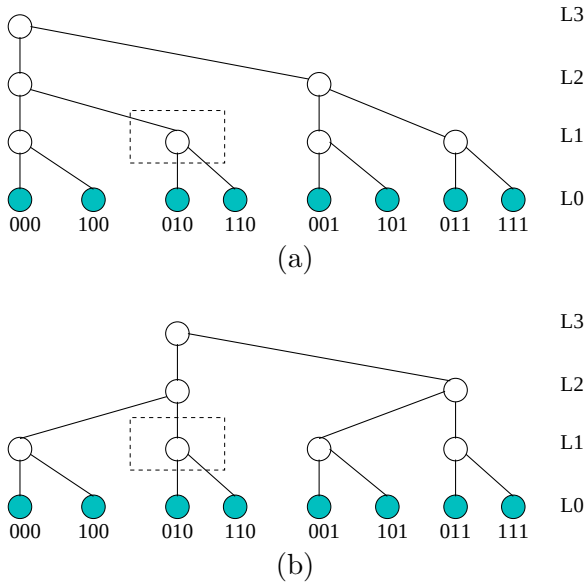


Figure 8.2: Aggregation trees for key 000 (a) and for key 111(b). Note that the virtual node at level 1 on node with ID 010 has a different parent in those aggregation trees corresponding to the second bit in the key.

might have different parents. In the example, the level 1 virtual node on the machine with ID 010 has a level 2 parent on the machine with ID 000 in the aggregation

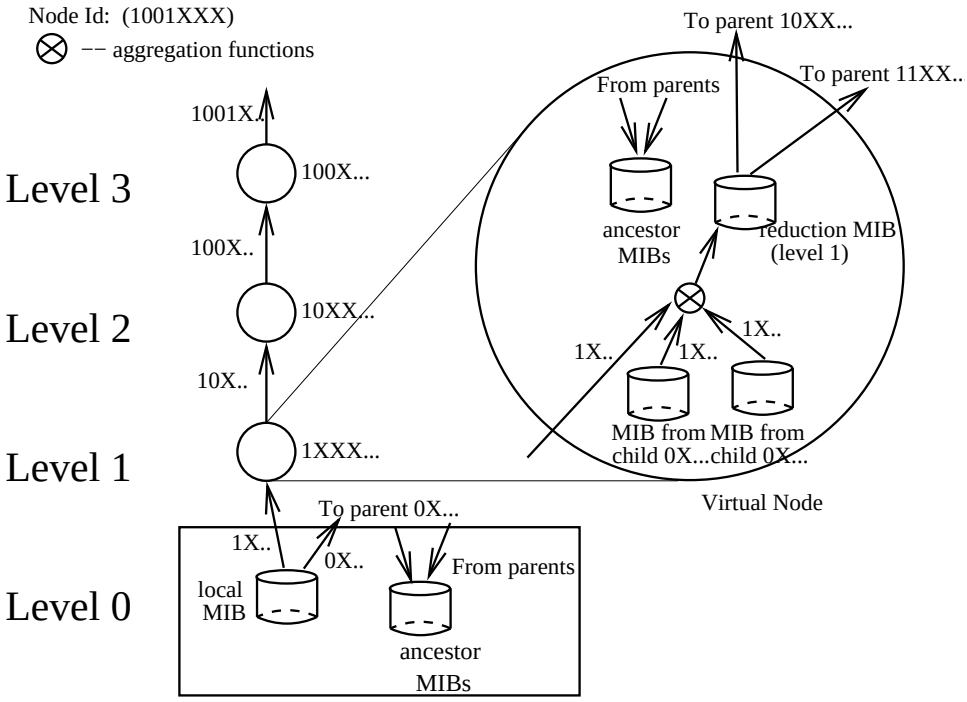


Figure 8.3: Example illustrating the data structures and the organization of them at a node.

tree corresponding to key 000 and a level 2 parent on the same machine in the aggregation tree corresponding to key 010. In the AML implementation, we merge virtual nodes at the same level from different aggregation trees hosted on a single machine into a single virtual node. Depending on the bit of an attribute key a virtual node is correcting, it will have a different parent.

Figure 8.3 illustrates the internal structure of the AML layer on each machine in SDIMS. Each physical node in the system acts as several virtual nodes in the AML: a node acts as leaf for all attribute keys, as a level-1 subtree root for keys whose hash matches the node's ID in b prefix bits (where b is the number of bits corrected in each step of the ADHT's routing scheme, $b = 1$ in the above example), as a level-i subtree root for attribute keys whose hash matches the node's ID in

the initial $i * b$ bits, and as the system's global root for attribute keys whose hash matches the node's ID in more prefix bits than any other node (in case of a tie, we use the key assignment policy described in Chapter 5 to break ties). Each virtual node has 2^b parents corresponding to b-bit correction of attribute keys and one of those parents will be hosted on the same machine.

8.1 DHT-AML Interface Layer

This layer is the glue between the underlying DHT system and the Aggregation Management Layer — it interfaces with the DHT layer to track the state of the overlay and based on that information maintains the aggregation tree parent/child relationships in the AML layer. The DHT common API [31] only provides information about the next hop for any given key and information about any changes to the local leafset (neighboring nodes on the logical ID space). The parent information for individual virtual nodes hosted on a machine can be extracted from using those interfaces. But to maintain aggregation trees, the AML layer at a node needs information about other nodes that use this node as the next hop for any key (i.e., children for that key) and also need to know whenever such nodes fail so that it can repair the aggregation trees. Below we describe how this interface layer gathers and maintains information about children for virtual nodes hosted on a machine.

For a level m virtual node on a machine with ID $i_1 i_2 i_3 ... i_d$ where d is the number of digits in the machine's ID ($= n/b$ where n is the number of bits in ID and b is the digit length in bits), for each key of the form —

$$i_1 i_2 i_3 ... i_m j z_1 z_2 ... z_{d-m-1} \text{ where } j \in [0, 2^b] \setminus i_m \text{ and } \forall_j z_j = 0,$$

the DHT-AML Interface layer uses the DHT API **local_lookup(key)** [31] to determine the next hop neighbor in the ADHT. These next hop neighbors will be parents

for that virtual node. After determining a parent for such a key, the DHT-AML Interface Layer sends an `IAmYourChild` message to the parent node informing about this parent-child relationship. When the DHT-AML Interface Layer on a machine receives such a message, it exposes that information to the corresponding virtual node in the AML and also starts monitoring liveness of the sending node.

8.2 Aggregation Management Layer

8.2.1 Data Structures

To support hierarchical aggregation, each virtual node at the root of a level-i subtree maintains several MIBs that store (1) *child MIBs* containing raw aggregate values gathered from children, (2) a *reduction MIB* containing locally aggregated values across this raw information, and (3) an *ancestor MIB* containing aggregate values scattered *down* from ancestors. This basic strategy of maintaining child, reduction, and ancestor MIBs is based on Astrolabe [101], but our structured propagation strategy channels information that flows up according to its attribute key and our flexible propagation strategy only sends child updates *up* and ancestor aggregate results *down* as far as specified by the attribute key's aggregation function. Note that in the discussion below, for ease of explanation, we assume that the routing protocol is correcting single bit at a time ($b = 1$). Our system, built upon Pastry, handles multi-bit correction ($b = 4$) and is a simple extension to the scheme described here.

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

for synchronization after disconnections [75].

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

A virtual node n_i at level i also maintains an *ancestor MIB* to store the tuples containing attribute key and a list of aggregate values at different levels scattered down from ancestors. Note that the list for a key might contain multiple aggregate values for a same level but aggregated at different nodes (see Figure 5.9). So, the aggregate values are tagged not only with level information, but are also tagged with the ID of the node that performed the aggregation.

Level-0 differs slightly from other levels. Each level-0 leaf node maintains a *local MIB* rather than maintaining child MIBs and a reduction MIB. This local MIB stores information about the local node's state inserted by local applications via *update()* calls. We envision various "sensor" programs and applications inserting data into local the MIB. For example, one program might monitor local configuration and perform updates with information such as total memory, free memory, etc., A distributed file system might perform updates for each file stored on the local node.

Along with these MIBs, a virtual node maintains two other tables: an aggregation function table and an outstanding probes table. An aggregation function table contains the aggregation function and installation arguments (see Table 3.1) associated with an attribute type or an attribute type and name. Each aggregation function is installed on all nodes in a domain's subtree, so the aggregation function table can be thought of as a special case of the ancestor MIB with domain functions always installed *up* to a root within a specified domain and *down* to all nodes within the domain. The outstanding probes table maintains temporary information regarding in-progress probes.

8.3 API Support

Given these data structures, it is simple to support the three API functions described in Section 3.2.

8.3.1 Install

The *Install* operation (see Table 3.1) installs on a domain an aggregation function that acts on a specified attribute type. Execution of an install operation for function $aggrFunc$ on attribute type $attrType$ proceeds in two phases: first the install request is passed up the ADHT tree with the attribute key *(attrType, null)* until it reaches the root for that key within the specified domain. Then, the request is flooded down the tree and installed on all intermediate and leaf nodes.

8.3.2 Update

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

8.3.3 Probe

A *Probe* collects and returns the aggregate value for a specified attribute key for a specified level of the tree. As Figure 3.1 illustrates, the system satisfies a probe for a level-i aggregate value using a four-phase protocol that may be short-circuited when updates have previously propagated either results or partial results up or

down the tree. In phase 1, the *route probe phase*, the system routes the probe up the attribute key's tree to either the root of the level-i subtree or to a node that stores the requested value in its ancestor MIB. In the former case, the system proceeds to phase 2 and in the latter it skips to phase 4. In phase 2, the *probe scatter phase*, each node that receives a probe request sends it to all of its children unless the node's reduction MIB already has a value that matches the probe's attribute key, in which case the node initiates phase 3 on behalf of its subtree. In phase 3, the *probe aggregation phase*, when a node receives values for the specified key from each of its children, it executes the aggregation function on these values and either (a) forwards the result to its parent (if its level is less than i) or (b) initiates phase 4 (if it is at level i). Finally, in phase 4, the *aggregate routing phase* the aggregate value is routed down to the node that requested it. Note that in the extreme case of a function installed with `UP` = `DOWN` = 0, a level-i probe can touch all nodes in a level-i subtree whereas in the opposite extreme case of a function installed with `UP` = `DOWN` = ALL, a probe is a completely local operation at a leaf.

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

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

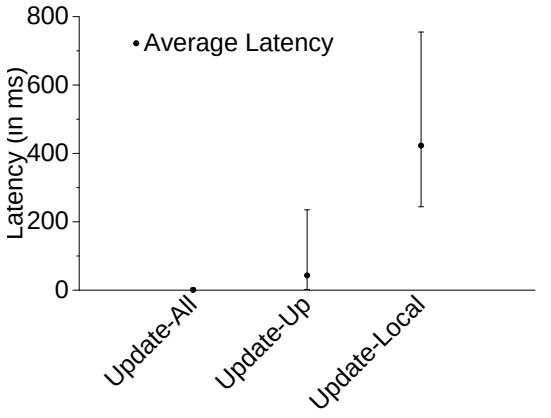
In our prototype, a probe for a level l aggregate at a leaf node returns aggregate values at all levels on the path from the leaf node to the l virtual node. Thus a probe response consists of a set of tuples of the form (*level*, *aggrValue*).

Our current prototype does not implement access control on install, update, and probe operations but we plan to implement Astrolabe's [101] certificate-based restrictions. Also our current prototype does not restrict the resource consumption in executing the aggregation functions; but, techniques from research on resource management in server systems and operating systems [7, 10] can be applied here.

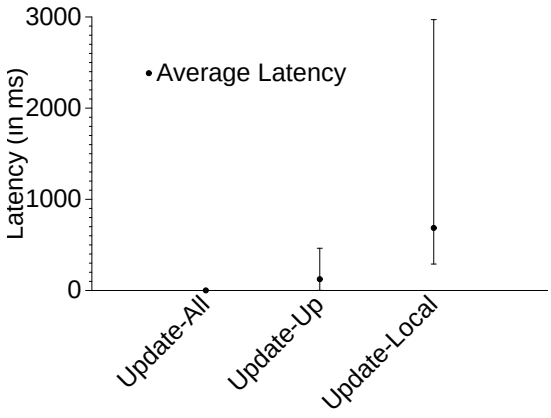
8.4 Testbed Experiments

We run our prototype on 180 department machines (some machines ran multiple node instances, so this configuration has a total of 283 SDIMS nodes), on 69 machines of the PlanetLab [76] testbed, and on 256 nodes in the Emulab testbed [107]. 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 8.4 shows the observed latencies for all three testbeds. Notice that the latency in Update-Local is high compared to the Update-UP policy. This is because latency in Update-Local is affected by the presence of even a single slow machine or a single machine with a high latency network connection.

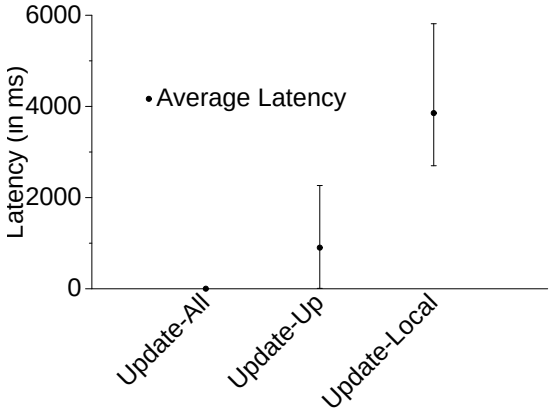
In the second benchmark, we examine robustness. We install one aggregation function of type Update-Up that performs the sum operation on an integer valued attribute. Each node updates the attribute with the value 10. Then we monitor the latencies and results returned on the probe operation for the global aggregate on one chosen node, as we kill a node after every few probes. Figure 8.5 shows the results



(a) CS Department



(b) Planetlab testbed



(c) Emulab testbed

Figure 8.4: Latency of probes for aggregate at global root level with three different modes of aggregate propagation on (a) department machines, (b) PlanetLab machines, and (c) Emulab setup. We also show the maximum and minimum latency observed in each experiment.

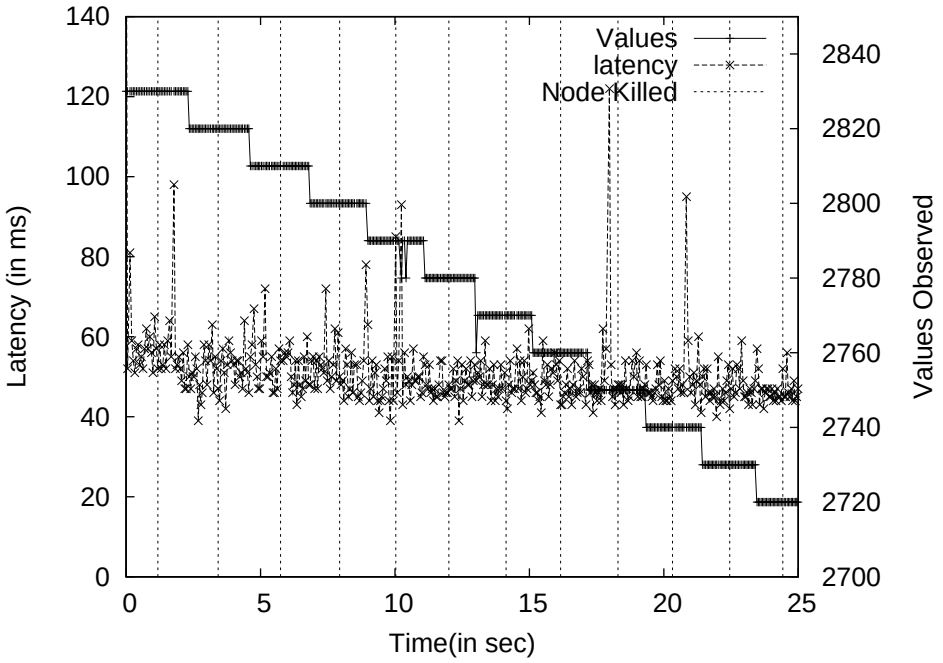


Figure 8.5: Micro-benchmark on department network showing the behavior of the probes from a single node when failures are happening at some other nodes. All 283 nodes assign a value of 10 to the attribute.

on the departmental testbed. Due to the nature of the testbed (machines in a department), there is little change in the latencies even in the face of reconfigurations. In Figure 8.6, we present the results of the experiment on the PlanetLab testbed. The root node of the aggregation tree is terminated after about 275 seconds. There is a 5X increase in the latencies after the death of the initial root node as a more distant node becomes the root node after repairs. Figure 8.7 shows the results on the Emulab testbed. In all testbeds, the values returned on probes start reflecting the correct situation within a short time after the failures.

From the above testbed benchmark experiments and the simulation experiments on flexibility and scalability (Chapter 4), administrative isolation (Chapter 5), we conclude that (1) the flexibility provided by SDIMS allows applications to trade-

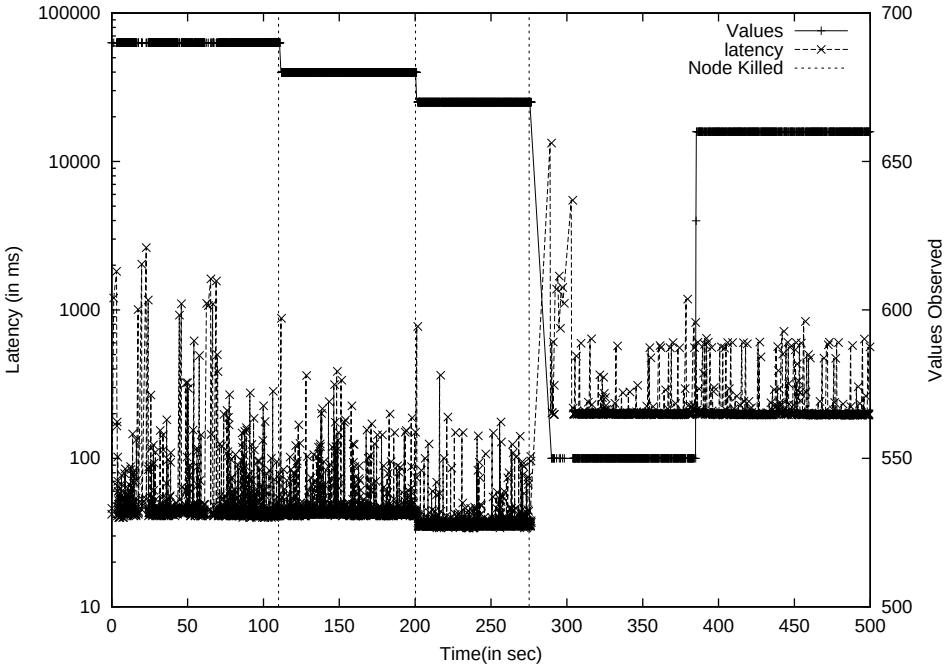


Figure 8.6: Probe performance during failures on 69 machines of the PlanetLab testbed

off read-write overheads (Figure 4.8), read latency, and sensitivity to slow machines (Figure 8.4), (2) a good default aggregation strategy is *Update-Up* which has moderate overheads on both reads and writes (Figure 4.8), has moderate read latencies (Figure 8.4), and is scalable with respect to both nodes and attributes (Figure 4.9), and (3) small domain sizes are the cases where DHT algorithms fail to provide path convergence more often and SDIMS ensures path convergence with only a moderate increase in path lengths (Figure 5.11).

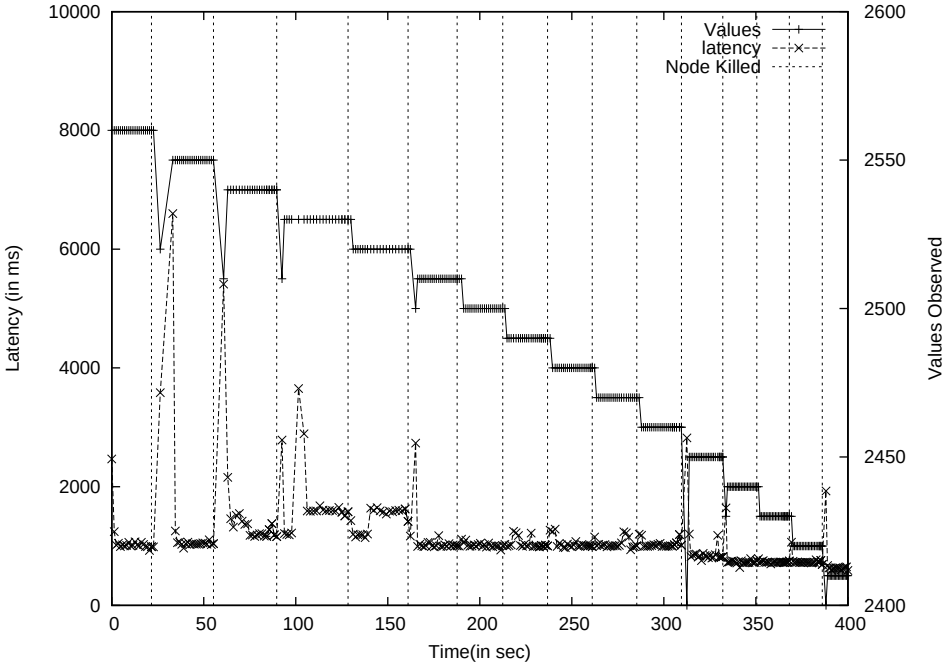


Figure 8.7: Probe performance during failures on 256 nodes in Emulab testbed

Chapter 9

Applications and Case Studies

The goal of this dissertation is to build a scalable information management middleware for large distributed systems that is useful for a wide range of applications. In this chapter, we first present examples of building large scale applications using SDIMS — a file location system and a multicast system — to illustrate how applications can exploit the SDIMS flexible API and scalability features. Then, we present case studies on how SDIMS is being used in two other research works — to build a controller for a distributed file replication system and to build a network monitoring system.

9.1 System Usage

In this section, we describe the usage of our SDIMS system through two example large distributed applications — a file location system and a multicast system. These applications illustrate advantages of the aggregation abstraction and the flexible API exposed by SDIMS.

Algorithm 9 $f_{\texttt{fileLocation}}$(childValueSet)

```
1: if childValueSet ≠ ∅ then
2:     return randomlyChoose(childValueSet)
3: else
4:     return NULL
5: end if
```

9.1.1 File Location System

The goal of a file location system is to track files located on machines in a system and provide that information to users and other applications. On a machine with network address *IPaddr*, for each file *foo*, we update an SDIMS attribute (`fileLocation`, *foo*) with a value *IPaddr* denoting that the file *foo* is available at machine with IP address *IPaddr*. Before the updates, we also install in SDIMS an aggregation function $f_{\texttt{fileLocation}}$ associating it with attribute type `fileLocation` and with *Update-Up* aggregation strategy (`UP`=all and `DOWN`=0). For attribute (`fileLocation`, *foo*), aggregation is done along the aggregation tree corresponding to the attribute key hash(`fileLocation`, *foo*). The aggregate value at a virtual node in that aggregation tree is computed by evaluating function $f_{\texttt{fileLocation}}$ with a set of non-null values for the attribute from the virtual node's children. Note that the aggregation function in this case simply picks one of the non-null values from children as the aggregate value. On a machine, an user or application locates file *foo* by performing a series of probes for the aggregate value starting at level 0 and increasing the level number until a non-null aggregate value is found or the maximum level in the system is reached. Another way to perform this location is to probe for the maximum level directly which will return aggregate values at all intermediate virtual nodes and then pick a non-null lowest level aggregate value.

In Figure 9.1, we illustrate the aggregation tree for attribute (`fileLocation`, *foo*) along with aggregate values at individual nodes in an eight node system. Suppose node G wishes to locate file *foo*. When it performs probe for the root level

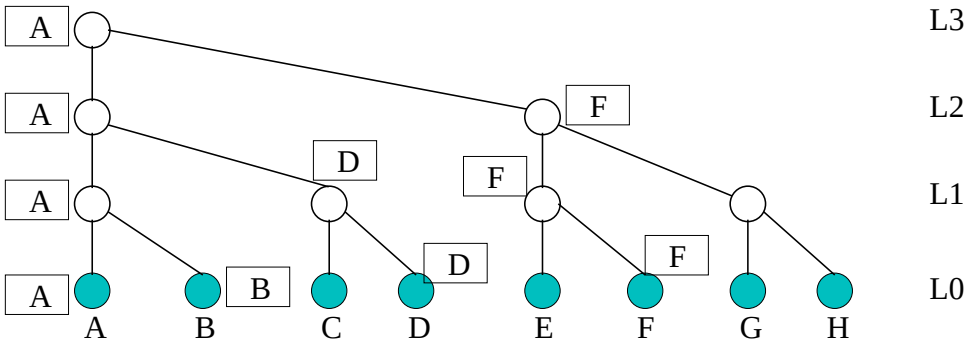


Figure 9.1: The aggregation tree for attribute (`fileLocation`, *foo*) along with the aggregate values. We denote the IP addresses of individual machines with capital alphabets and the aggregate value for this attribute at individual nodes is shown in a box next to a node. Absence of the aggregate value at a node indicates a NULL value for the aggregate at that node.

aggregate of the attribute (`fileLocation`, *foo*), it gets the following response: ((0, NULL), (1, NULL), (2, F), (3, A)). It uses machine F as the answer as that is the lowest level non-NULL aggregate value, and contacts machine F to obtain the file. Note that different machines will obtain different answers for their probes. Choosing lowest level non-NULL value ensures that nodes access different machines that host a file.

Note that the aggregation function $f_{\texttt{fileLocation}}$ randomly picks one of the non-null child values as the aggregate value at a virtual node. In some case, it might be preferable to choose a machine on the basis of different metrics like computing capacity of the machine, upload bandwidth available at a machine, etc. An aggregation function that considers the upload bandwidth is shown in Algorithm 10. In this case, a machine that hosts file *foo* performs update for attribute (`fileLocationBW`, *foo*) with a tuple $\langle IPaddr,\ BW \rangle$ as value, where BW is the upload bandwidth available at the node. The aggregation function at an intermediate virtual node chooses a tuple with maximum BW field. A machine wishing to locate a file can then choose either lowest level non-null aggregate or an aggregate with a required upload

Algorithm 10 $f_{\texttt{fileLocationBW}}$(childValueSet)

```
1: maxBW ← 0
2: maxBWValue ← NULL
3: for all ⟨IPAddr, BW⟩ ∈ childValueSet do
4:    if  BW > maxBW then
5:       maxBW ← BW
6:       maxBWValue ← ⟨IPAddr, BW⟩
7:    end if
8: end for
9: return  maxBWValue
```

bandwidth and contact the machine by connecting to the *IPaddr* specified in the aggregate value. Similar techniques can be used for building a more general resource or service location applications.

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

9.1.2 Multicast Tree Construction

The goal of a multicast system is to construct a spanning tree between interested machines to propagate messages from a source machine to all interested machines. Each machine that is interested in a multicast session, say *sessOne*, updates the attribute (`multicast`, *sessOne*) with a tuple ⟨*IPAddr*, { *IPAddr* }⟩, where *IPAddr* is the IP address of the machine. The format of an aggregate value tuple at a node in the aggregation tree is ⟨*repAddr*, set of reps at children⟩ where *repAddr* is the IP address of a machine that is chosen as representative for the subtree rooted at this node. The second field contains all representatives chosen at the children. An intermediate virtual node computes its aggregate value using the function shown in Algorithm 11 installed for attribute type `multicast` with the *Update-Up* aggregation strategy. At a virtual node, given a set of aggregate values at the children of this

Algorithm 11 $f_{\texttt{multicast}}$(childValueSet)

```
1: childRepList ← Φ
2: for all ⟨rep, childrepset⟩ ∈ childValueSet do
3:    childRepList ← childRepList ⋃ {rep}
4: end for
5: if childRepList = Φ then
6:    return NULL
7: else
8:    rep ⇐ chooseRep(childRepList)
9:    return ⟨rep, childRepList⟩
10: end if
```

node, the aggregation function picks one of the representatives chosen by children to act as the representative at this level (first field in the aggregate value tuple) and also forms a set of representatives chosen by children (second field in the aggregate value tuple).

A machine interested in session *sessOne*, say machine A with IP Address $IPAddr_A$, performs a series of probes for attribute (`multicast`, *sessOne*) starting from level zero and increasing the level number until it finds an aggregate value, say at level l, where the chosen representative's IP address, say $IPAddr_B$, is not same as this machine's IP address. Machine A then deems the machine with IP address $IPAddr_B$ as the parent in the spanning tree for this multicast session. Also machine A can extract the IP addresses of machines that will be this node's children in the spanning tree by taking union of the second field in the aggregate value tuples for all level $l' < l$. To get updates whenever any changes happen to the structure of the spanning tree, a machine performs continuous probes instead of one-shot probes when retrieving aggregate values at different levels in the aggregation tree. We show the aggregate values for session *sessOne* with four interested nodes in Figure 9.2 and we also show the resulting spanning tree in Figure 9.3.

A node wishing to send a multicast message invokes the *multicastRoutine* shown in Algorithm 12 locally (with the node's IP address, *localIPAddr*, passed for

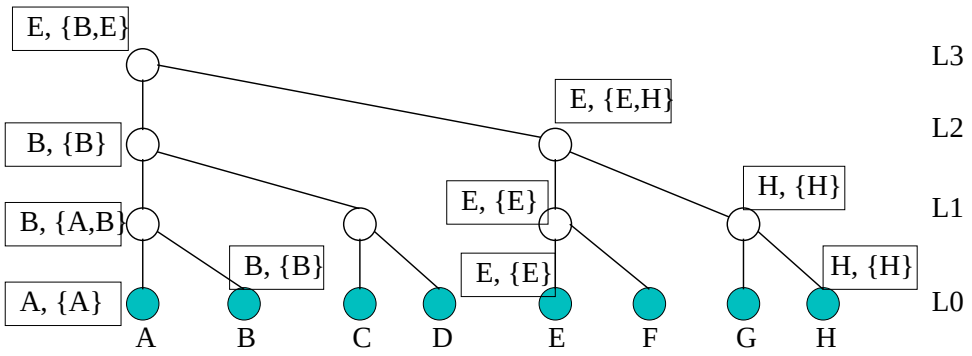


Figure 9.2: Aggregation tree and aggregate values for (`multicast`, *sessOne*) attribute. We denote the IP addresses of individual machines with capital letters and the aggregate value at individual nodes is shown in a box next to a node. Absence of the aggregate value at a node indicates a NULL value.

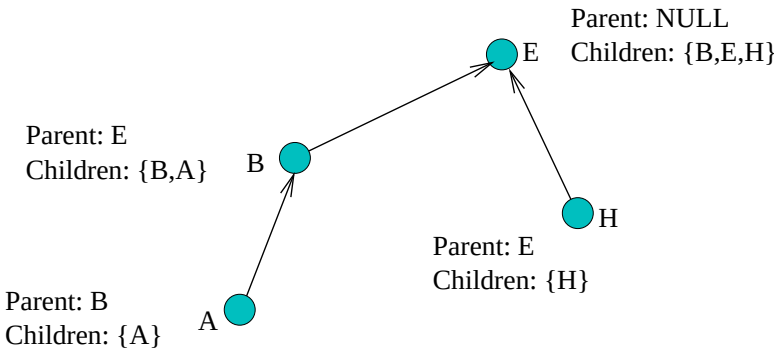


Figure 9.3: The resulting spanning tree for session *sessOne* built based on SDIMS aggregation shown in Figure 9.2. We alse show that parent and child set computed at each interested node obtained by probing the SDIMS.

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. Thus this multicast algorithm uses SDIMS as a control plane to maintain a spanning tree and forms a separate data plane which incurs communication cost on only nodes interested in that session.

Algorithm 12 multicastRoutine(sessName, message, fromIPAddr)

```
1: parent ← currentParent(sessName)
2: childSet ← currentChildSet(sessName)
3: neighs ← childSet ⋃ { parent}
4: for all peer ∈ neighs do
5:    if  peer ≠ fromIPAddr then
6:       Forward message to peer
7:    end if
8: end for
```

In Algorithm 11, a representative is chosen from the childRepList by invoking the *chooseRep* routine. Similar to enhanced $f_{\texttt{fileLocationBW}}$ shown in the Algorithm 10, nodes can provide the locally available upload and/or download bandwidth in the values and let *chooseRep* routine pick a representative with the highest bandwidth available from the childRepList.

9.2 Case Studies

SDIMS is designed as a general distributed monitoring and control infrastructure for a broad range of applications. Above, we discussed a couple of large distributed applications — building a file location system and a multicast membership service. Van Renesse et al. [101, 100] provide detailed examples of how such a service can be used for a peer-to-peer caching directory, a data-diffusion service, a publish-subscribe system, barrier synchronization, and voting. Additionally, we have initial experience using SDIMS to construct two significant applications: a control plane for a large-scale distributed file system [32] and a network monitor for identifying "heavy hitters" that consume excess resources.

9.2.1 Distributed file system control

The PRACTI (Partial Replication, Arbitrary Consistency, Topology Independence) replication system [32] provides a set of mechanisms for data replication over which

arbitrary control policies can be layered. We use SDIMS to provide several key functions in order to create a file system over the low-level PRACTI mechanisms.

First, nodes use SDIMS as a directory to handle read misses. We can use SDIMS similar to the file location application example explained in Section 9.1.1. But in practice, a single file might be accessed within a short period or almost concurrently by a large number of machines (e.g., the slashdot effect [2, 46], a wide-area distributed system experiment downloading executables and input files, etc.) If a file is located on one or a few nodes in the system, then file access latencies will be severely affected during such high load. Below we detail how we handle such flash crowds in PRACTI using SDIMS.

When a node n receives an object o, it updates the *(ReadDir, o)* attribute with the value $\langle n, \texttt{RECEIVED} \rangle$; when n discards o from its local store, it resets *(ReadDir, o)* to NULL. Upon a read miss at a node m for object o, it updates the *(ReadDir, o)* attribute with the value $\langle n, \texttt{WILLGET} \rangle$ denoted that it will soon fetch and cache the file. At each virtual node, the *ReadDir* aggregation function selects a random non-null child value with `RECEIVED` status. If no such child value exists, then it selects a random non-null child value with `WILLGET` status. In the absence of even such values, the aggregate value is simply set to NULL. We use the Update-Up policy for propagating updates. To locate a nearby copy of an object o, a node n_1 issues a series of probe requests for the *(ReadDir, o)* attribute, starting with $level = 1$ and increasing the $level$ value with each repeated probe request until a non-null tuple $\langle n_2, status \rangle$ is returned. Node n_1 then sends a demand read request to n_2, and n_2 sends the data if it has it. If n_2 is waiting to fetch the file, it will send a $waitingToFetch$ message to node n_1 and n_1 retries after a small time period. Conversely, if n_2 cannot serve the file (incorrect SDIMS state), it sends a nack to n_1, and n_1 issues a retry probe with the `DOWN` parameter in probe set to a value larger than used in the previous probe in order to force on-demand re-aggregation, which

will yield a fresher value for the retry. Note that this strategy effectively forms a tree among machines simultaneously accessing a file and hence effeciently transfers files with low response latencies.

Second, nodes subscribe to invalidations and updates to *interest sets* of files, and nodes use SDIMS to set up and maintain a per-interest-set network-topology-sensitive spanning trees for propagating this information. To subscribe to invalidations for interest set i, a node n_1 first updates the *(Inval, i)* attribute with its identity n_1, and the aggregation function at each virtual node selects one non-null child value. Finally, n_1 probes increasing levels of the *(Inval, i)* attribute until it finds the first node $n_2 \neq n_1$; n_1 then uses n_2 as its parent in the spanning tree. n_1 also issues a continuous probe for this attribute at this level so that it is notified of any change to its spanning tree parent. Spanning trees for streams of pushed updates are maintained in a similar manner. Note that this tree construction is similar to the multicast application example in Section 9.1.2 but differs in the following aspect: intermediate virtual nodes do not maintain set of child representative sets. Hence each node in this case only gets to know about their parent. But a node A obtains its children information as the children connect to the node after learning that node A is their parent.

In the future, we plan to use SDIMS for at least two additional services within this replication system. First, we plan to use SDIMS to track the read and write rates to different objects; prefetch algorithms will use this information to prioritize replication [103, 104]. Second, we plan to track the ranges of invalidation sequence numbers seen by each node for each interest set in order to augment the spanning trees described above with additional "hole filling" to allow nodes to locate specific invalidations they have missed.

Grid Microbenchmark We examine a 3-phase benchmark that represents running an experiment on a multi-machine cluster: in phase 1 *Disseminate*, each node

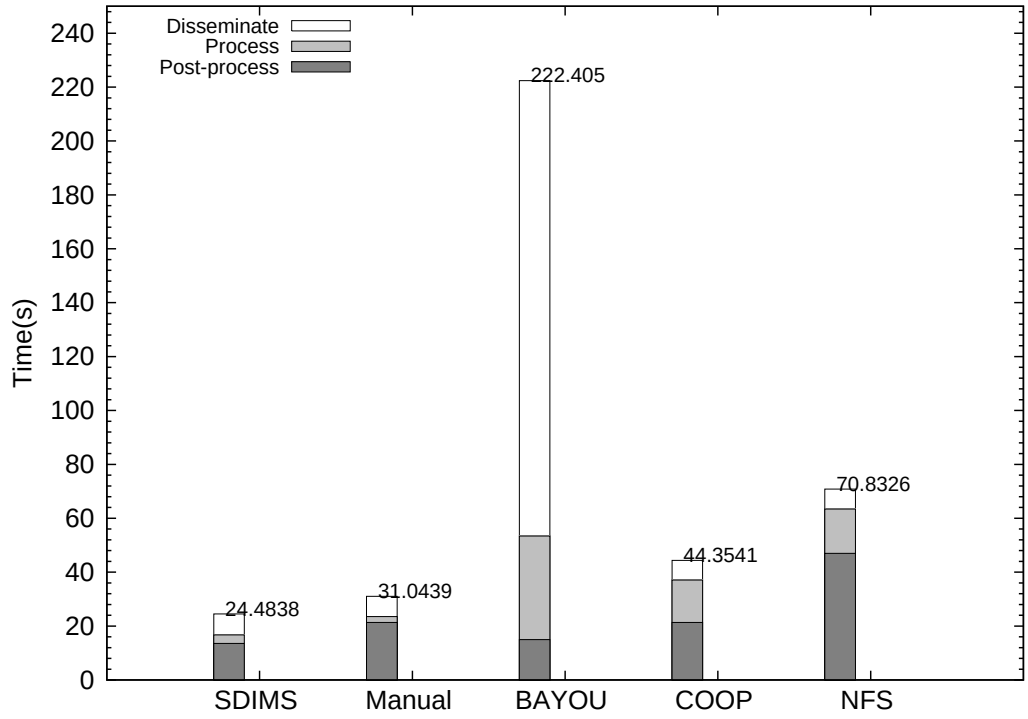


Figure 9.4: Efficacy of PRACTI with SDIMS controller in a grid micro-benchmark compared to four other controllers.

fetches 10MB of new executables and input data from the user's home node; in phase 2 *Process*, each node writes 10 files each of 100KB and then reads 10 files from randomly selected peers; in phase 3, *Post-process*, each node writes a 1MB output file and the home node reads all of these output files. We compare several different controllers with SDIMS-based controller built on PRACTI low level mechanisms — a controller similar to SDIMS-based controller but that is manually configured instead of using SDIMS for location or multicast tree construction for propagating invalidations and updates, a client-server system (e.g., NFS [74]), client-server with cooperative caching of read-only data (e.g., a Shark-like system [5]), and server-replication (e.g., a Bayou-like system [75]). All these five systems are compared

for grid microbenchmark in Figure 9.4. The SDIMS-based controller outperforms all other controllers — it outperforms the client-server system type controller, the cooperative caching based controller, and the server replication type controller as it uses PRACTI low level mechanism efficiently for the type of application, and it outperforms manually configured controller as it creates a different multicast tree for each file during initial dissemination and during update and invalidate propagation where as the manual controller follows a simple approach of using a single tree for multicast and a single tree in the initial dissemination phase.

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

9.2.2 Distributed heavy hitter problem

The goal of the heavy hitter problem is to identify network sources, destinations, or protocols that account for significant or unusual amounts of traffic. As noted by Estan et al. [35], this information is useful for a variety of applications such as intrusion detection (e.g., port scanning), denial of service detection, worm detection and tracking, fair network allocation, and network maintenance. Significant work has been done on developing high-performance stream-processing algorithms for identifying heavy hitters at one router, but this is just a first step; ideally these applications would like not just one router's views of the heavy hitters but an ag-

gregate view.

We use SDIMS to allow local information about heavy hitters to be pooled into a view of global heavy hitters. For each destination IP address IP_x, a node updates the attribute $(DestBW, IP_x)$ with the number of bytes sent to IP_x in the last time window. The aggregation function for attribute type $DestBW$ is installed with the Update-UP strategy and simply adds the values from child nodes. Nodes perform continuous probe for global aggregate of the attribute and raise an alarm when the global aggregate value goes above a specified limit. Note that only nodes sending data to a particular IP address perform probes for the corresponding attribute. Also note that techniques from [71] can be extended to the hierarchical case to tradeoff precision for communication bandwidth.

Chapter 10

Related Work

10.1 Aggregation Frameworks

The aggregation abstraction we use in our work is heavily influenced by the Astrolabe [101] project. Astrolabe adopts a Propagate-All and unstructured gossiping techniques to attain robustness [16]. However, any gossiping scheme requires aggressive replication of the aggregates. While such aggressive replication is efficient for *read-dominated* attributes, it incurs high message cost for attributes with a small read-to-write ratio. Our approach provides a flexible API for applications to set propagation rules according to their read-to-write ratios. Other closely related projects include Willow [102], Cone [12], DASIS [3], and SOMO [116]. Willow, DASIS and SOMO build a single tree for aggregation. Cone builds a tree per attribute and requires a total ordering on the attribute values.

Several academic [41, 73, 61, 105] and commercial [99] distributed monitoring systems have been designed to monitor the status of large networked systems. Some of them are centralized where all the monitoring data is collected and analyzed at a central host. Ganglia [41, 65] uses a hierarchical system where the attributes are replicated within clusters using multicast and then cluster aggregates are further

aggregated along a single tree. SWORD [73, 72] is a resource discovery tool deployed on PlanetLab [76]. SWORD collects reports about available resources on nodes, and answers queries from users requesting nodes matching user-defined criteria. Sophia [105] is a distributed monitoring system designed with a declarative logic programming model where the location of query execution is both explicit in the language and can be calculated during evaluation. This research is complementary to our work. TAG [61] 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 [77] 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 [86, 118] and upon other strategies [66, 80, 94]—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 [86], 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. Dabek et al [31] 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 the KBR API, it does not provide any interface to access the list of children for different prefixes.

Although object location is a predominant target application for DHTs, several other applications like multicast [24, 25, 90, 119] and DNS [29] are also built using DHTs. All these systems implicitly perform aggregation on some attribute,

and each one of them must be designed to handle any reconfigurations in the underlying DHT. With the aggregation abstraction provided by our system, designing and building of such applications becomes easier.

10.2 Different Types of Queries

10.2.1 Composite Queries

There are some ongoing efforts to provide the relational database abstraction on DHTs: PIER [49] and Gribble et al. [43]. These works mainly focus on supporting "Join" operation for database 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 queries – e.g., *find a nearest machine with file "foo" and has more than 2 GB of memory*. PeerDB [69] is another peer-to-peer data sharing system that supports content-based searches, allows users to share data without any shared global schema, and employs mobile agents to assist in query processing.

10.2.2 Arbitrary Range Queries

Arbitrary range queries are common in some distributed applications like sensor network monitoring systems [58], grid resource monitoring and management systems [4, 11, 72, 88], and distributed multi-player games [15]. In SDIMS, applications install an aggregation function to aggregate data from machines in the system and they can only probe for aggregate values computed at different levels in an aggregation tree. One simple way SDIMS can support arbitrary range queries is through maintaining all values in each intermediate aggregate. Range queries can then be resolved by probing for the global aggregate which will return a vector of values for all nodes in the system and pick values that satisfy the query. Though this scheme is simple, it might incur high bandwidth costs as size of the update messages in

the system can become quite large. Recently, several researchers proposed DHT based schemes for handling range queries [78, 15, 88, 110] and techniques from this research can be exploited in SDIMS to support arbitrary range queries in a more efficient way.

10.2.3 Stream Processing Queries

Stream processing applications, an emerging new class of applications, process data that is generated in a distributed environment and is pushed asynchronously to servers for processing. Recently, several systems like STREAM [93], AURORA [115], TELEGRAPH [96], and Borealis [1, 9, 109] are proposed for handling stream processing. These systems model the bulk of the processing required in these scenarios in terms of standard well-defined streaming operators, such as filters, windowed aggregates, windowed joins, etc. Most of this research focuses on defining appropriate language to express the stream processing queries. The Medusa and Borealis [1, 115] projects focus on achieving high availability and load management issues in stream-processing environments. While the design of SDIMS does not focus on efficiently supporting stream-processing queries, we are currently exploring to use SDIMS in building INSIGHT [51], a distributed network monitoring system that tracks streams of network flows data from a large number of routers.

Chapter 11

Conclusions

Information management is one of the key tasks of any large-scale distributed application. The goal of this dissertation is to design and build a general and scalable information management middleware for large distributed systems that will facilitate designing, developing, and deploying new distributed applications, and that will assist application developers in exploring the design space and tradeoffs in communication costs, response latencies, and consistency. This dissertation presents a Scalable Distributed Information Management System (SDIMS) that aggregates information in large-scale networked systems and that can serve as a basic building block for a broad range of applications. For large scale systems, *hierarchical aggregation* is a fundamental abstraction for scalability.

We design SDIMS by extending ideas from Astrolabe and DHTs to achieve (i) *scalability* with respect to both nodes and attributes through a new aggregation abstraction that helps leverage DHT's internal trees for aggregation, (ii) *flexibility* through a simple API that lets applications control propagation of reads and writes and through providing a mechanism that dynamically adapts the propagation based on observed read and write load in the system, (iii) *administrative isolation* through designing a novel Autonomous DHT (ADHT) that guarantees path convergence and

path locality properties, and (iv) *robustness* to node and network reconfigurations through reaggregation mechanisms and application tunable spatial replication.

We have built a prototype of SDIMS in Java using FreePastry [39] framework. Through extensive simulation experiments and micro-benchmarks on several real testbeds, we observe that SDIMS incurs an order of magnitude less node stress than Astrolabe, exposes several data propagation mechanisms and allows applications to decide a policy, incurs no administrative isolation violations whereas flat DHTs incur violations in up to 14% of pairs of paths, and is robust to failures by reconfiguring and reaggregating data in a timely manner. We have designed and built several applications on SDIMS including a file location system and a multicast tree construction system. We have also used SDIMS in two other research projects in our lab — as a controller for a distributed file replication system PRACTI and for a network monitoring system.

Our current work on SDIMS is a first step towards achieving a general and scalable information management middleware that will be a distributed operating systems backbone. Our experience in using SDIMS for designing and developing several applications has been quite promising. Our work also opens up many research issues in different fronts that need to be solved. Below we discuss some of these future research directions.

Handling Composite Queries As discussed in Chapter 4, different attributes in SDIMS are aggregated along different aggregation trees. Whereas this technique allows SDIMS to scale to a large number of attributes, queries involving multiple attributes poses a challenge. We have outlined some of the directions in Section 4.5.

Error Bounds In any large distributed systems, where reconfigurations are a norm, it is hard to guarantee that an aggregate value returned on a probe corresponds to the current state of the system. A more appropriate solution is to an

aggregate value along with some error bounds that will denote some form of confidence in the aggregate value. For example, consider computing the average CPU load in a system of thousand machines. Suppose probes return the average along with the total number of machines used in computing it. If a probe returns an answer denoting that only a hundred machines are used for computing the answer, then an user or application can deem that value to be incorrect and reissue a new probe.

Resource Management and Security One of the important aspects that SDIMS does not address is how many resources should be allocated for an attribute — like memory for computing the aggregation function, amount of time an aggregation function is allowed to run, storage space for an aggregate value, etc. Also another important issue is what other data on a machine should an aggregation function be allowed to access. Techniques for authentication from Astrolabe [101] and techniques for resource management in operating system research [7, 10, 34] can be employed in SDIMS too.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research*, Asilomar, California, January 2005.

[2] S. Adler. The Slashdot Effect: An Analysis of Three Internet Publications. `http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html`, 1999.

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

[4] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 33, Washington, DC, USA, 2002. IEEE Computer Society.

[5] S. Annapureddy, M. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation*, May 2005.

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

[7] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct 2000.

[8] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architect ure for Intentional Resource Discovery. In *Pervasive 2002 - International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[9] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD*, 2005.

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

[11] S. Basu, S. Banerjee, P. Sharma, and S.-J. Lee. NodeWiz: Peer-to-Peer Resource Discovery for Grids. In *Fifth International Workshop on Global and Peer-to-Peer Computing (GP2PC)*, May 2005.

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

[13] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *The Second International Workshop on Peer-to-peer systems*, February 2003.

[14] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. TotalRecall: System Support for Automated Availability Management. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2004.

[15] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, Portland, OR, August 2004.

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

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

[18] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Santa Clara, USA, June 200.

[19] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, 1999.

[20] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Primary-backup Protocols: Lower Bounds and Optimal Implementations. In *Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, 1992.

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

[22] M. Castro, M. Costa, and A. Rowstron. Performance and Dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft Research Cambridge, UK, December 2003.

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

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

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

[26] 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.

[27] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. SCAN: a Dynamic Scalable and Efficient Content Distribution Network. In *First Intl. Conf. on Pervasive Computing*, Aug 2002.

[28] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 2002.

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

[30] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, Aug 2001.

[31] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *IPTPS*, February 2003.

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

[33] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *Proceedings of IEEE Infocom*, Mar. 2000.

[34] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, Dec. 1995.

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

[36] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Intl Workshop on Quality of Service*, 1999.

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

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

[39] FreePastry. `http://freepastry.rice.edu`.

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

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

[42] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[43] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do for Databases, and Vice Versa? In *Proceedings of the WebDB*, 2001.

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

[45] 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.

[46] A. M. C. Halavais. *The Slashdot Effect: Analysis of a Large-Scale Public Conversation on the World Wide Web*. PhD thesis, University of Washington-Seattle, 2003. `http://alex.halavais.net/research/diss.pdf`.

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

[48] 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.

[49] 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.

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

[51] N. Jain, P. Yalagandula, M. Dahlin, and Y. Zhang. INSIGHT: A Distributed Monitoring System for Tracking Continuous Queries. In *Work-in-Progress Session at ACM SOSP*, Brighton, UK, October.

[52] 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.

[53] D. R. Karger and M. Ruhl. Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks. In *Third International Workshop on Peer-to-Peer Systems*, San Diego, CA, February 2004.

[54] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *FTCS99*, June 1999.

[55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[56] L. Lamport. Part time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[57] L. Lamport. Paxos made simple. *ACM SIGACT News Distributed Computing Column*, 32(4), Dec. 2001.

[58] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries

in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 63–75. ACM Press, 2003.

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

[60] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

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

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

[63] 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.

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

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

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

[67] A. Mislove and P. Druschel. Providing Administrative Control and Autonomy in Peer-to-Peer Overlays. In *Third International Workshop on Peer-to-Peer Systems*, San Diego, CA, February 2004.

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

[69] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proceedings of the 19th International Conference on Data Engineering*.

[70] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamrithem, and R. Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *International World Wide Web Conference*, May 2002.

[71] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Twenty-Sixth International Conference on Very Large Data Bases*, pages 144–155, Sept. 2000.

[72] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2004.

[73] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the 14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[74] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[75] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

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

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

[78] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief Announcement: Prefix Hash Tree. In *Proceedings of ACM PODC*, July 2004.

[79] V. Ramasubramanian and E. G. Sirer. "beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays". In *NSDI*, March 2004.

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

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

[82] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, June 2004.

[83] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, , and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *ACM SIGCOMM*, August 2005.

[84] 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.

[85] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

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

[87] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In J. Crowcroft and

M. Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, Nov. 2001.

[88] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[89] F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. *Computing Surveys*, 22(3):299–319, Sept. 1990.

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

[91] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability, 2001.

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

[93] Stanford Stream Data Manager Group (STREAM). `http://www-db.stanford.edu/stream/index.html`.

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

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

[96] The telegraph project at uc berkeley. `http://telegraph.cs.berkeley.edu/`.

[97] 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.

[98] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

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

[100] R. van Renesse. The importance of aggregation. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Springer-Verlag Lecture Notes in Computer Science*, Heidelberg, Germany, April 2003. Springer-Verlag.

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

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

[103] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a wan. In *Proceedings of the 20th Symposium on the Principles of Distributed Computing*, Aug. 2001.

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

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

[106] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[107] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Dec. 2002.

[108] 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.

[109] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *The 21st International Conference on Data Engineering*, 2005.

[110] P. Yalagandula and J. C. Browne. Solving Range Queries in a Distributed System. Technical Report TR-04-18, Department of Computer Science, The University of Texas at Austin, 2004.

[111] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Real Large Distributed Systems. In *First Workshop on Real Large Distributed Systems (WORLDS)*, December 2004.

[112] B. Yang and H. Garcia-Molina. Designing a Super-peer Network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*.

[113] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.

[114] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.

[115] S. Zdonik, M. Stonebraker, M. Cherniack, U. Centintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa Projects. In *Bulletin of the Technical Committe on Data Engineering*, pages 3–10. IEEE Computer Society, March 2003.

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

[117] B. Y. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. "brocade: Landmark routing on overlay networks". In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[118] 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.

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

Vita

Praveen Yalagandula was born in Khammam, India on April 14, 1977, the son of Seetha Rama Rao Yalagandula and Premalatha Yalagandula. After completing high school at Jawahar Navodaya Vidhyalaya, Paleru, Khammam, India in 1994, he studied at the Indian Institute of Technology, Kharagpur where he received a Bachelor of Technology degree in Computer Science & Engineering in May 1998. Praveen earned Master of Science in Engineering degree from the Electrical and Computer Engineering department at the University of Texas at Austin in August 2000. He then joined the Ph.D. program in the Department of Computer Sciences at the University of Texas at Austin. During his graduate studies, Praveen did summer internships at Cadence Berkeley Labs in summer 1999, at HRL in summer 2001, and at Intel Research in Pittsburgh in summer 2004. In Spring 2005, Praveen received James C. Browne Graduate Fellowship from the Computer Sciences Department awarded every year to outstanding graduate students.

Permanent Address: 6805 Wood Hollow Dr Apt 233
Austin, TX 78731

This dissertation was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.