Routing Tradeoffs in Dynamic Peer-to-peer Networks

by

Jinyang Li

B.Sc, National University of Singapore (1998)
S.M., Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

November 2005

Author .....
Department of Electrical Engineering and Computer Science
November 4, 2005

Certified by .
.....
Robert Morris
Associate Professor
pervisor

Accepted by ....
.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Routing Tradeoffs in Dynamic Peer-to-peer Networks

by

Jinyang Li

Submitted to the Department of Electrical Engineering and Computer Science
on November 4, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Distributed Hash Tables (DHTs) are useful tools for building large scale distributed systems. DHTs provide a hash-table-like interface to applications by routing a key to its responsible node among the current set of participating nodes. DHT deployments are characterized by churn, a continuous process of nodes joining and leaving the network.

Lookup latency is important to applications that use DHTs to locate data. In order to achieve low latency lookups, each node needs to consume bandwidth to keep its routing tables up to date under churn. A robust DHT should use bandwidth sparingly and avoid overloading the network when the the deployment scenario deviates from design assumptions. Ultimately, DHT designers are interested in obtaining best latency lookups using a *bounded* amount of bandwidth across a wide range of operating environments. This thesis presents a new DHT protocol, Accordion, that achieves this goal.

Accordion bounds its overhead traffic according to a user specified bandwidth budget and chooses a routing table size that minimizes lookup latency, balancing the need for both low lookup hop-count and low timeout probability. Accordion employs a unique design for managing routing tables. Nodes acquire new routing entries opportunistically through application lookup traffic. Large bandwidth budgets lead to big routing table and low lookup hop-count. Nodes evict entries that are likely dead based on past statistics of node lifetimes. Short node lifetimes lead to high eviction rates and a small routing table with low maintenance overhead. The routing table size is determined by the equilibrium of the neighbor acquisition and eviction processes. Accordion's automatic table size adjustment allows it to bound its bandwidth use and achieve latencies similar or better than existing manually tuned protocols across a wide range of operating environments.

Thesis Supervisor: Robert Morris
Title: Associate Professor

Acknowledgments

This thesis is the result of a collaboration with Jeremy Stribling, Robert Morris, Frans Kaashoek and Thomer Gil. Many people helped develop and support the *p2psim* simulator. Thomer Gil and Robert Morris designed and implemented most of the simulation infrastructure. Russ Cox developed and supported the user-level thread library used by *p2psim*. Jeremy Stribling wrote Tapestry. Anjali Gupta implemented OneHop. Robert Morris implemented Kelips. Frans Kaashoek wrote Koorde. Frank Dabek wrote the Vivaldi coordinate system, and Thomer Gil wrote Kademlia. I thank the *DHash* team, especially Frank Dabek and Emil Sit, for helping me incorporate Accordion in the *DHash* software release.

I am very lucky to have Robert Morris and Frans Kaashoek as my advisors. They have provided me enormous help and encouragement throughout this work. Their intellect, rigor, enthusiasm and humor will always inspire me.

This thesis describes research done as part of the IRIS project funded by the National Science Foundation under NSF Cooperative Agreement No. ANI-0225660. I am very grateful to Microsoft Research for supporting the work with a Microsoft Research Fellowship.

I owe my six years of fun in graduate school to a lot of people, especially folks from PDOS.

Robert Morris Frans Kaashoek Scott Shenker

Y. C. Tay David Karger Joseph Hellerstein

Xiaowei Yang Charles Blake

Eddie Kohler David Mazieres

Frank Dabek

Athicha Muthitachereon

Thomer Gil Rodrigo Rodrigues Douglas De Couto

Max Krohn Jeremy Stribling Cliff Frey

Dan Aguayo Sanjit Biswas John Bicket

Joshua Cates

John Jannotti	Emmett Witchel	Benjie Chen
Emil Sit	Bryan Ford	David Andersen
Russ Cox	Alex Yip	James Robertson
Jaeyeon Jung	Magda Balazinska	Michel Goraczko
Dina Katabi	Hari Balakrishnan	Chandrasekhar Boyapati
Boon Thau Loo	Yan Zhang	Chris Laas
Brad Karp	Aditya Akella	Sean Rhea
Michael Walfish	Neena Lyall	
Elisa Alonso	Mingliang Jiang	Jinyuan Li
Hang Pang Chiu	Huiling Huang	Kun Yang
Becky Qiang	Junfeng Yang	Min Tang
Yuxin Yang	Quan Zhao	Lei Cai
Beiyi Jiang	Jingyi Yu	Yuan Mei

Contents

Chapter 1

Introduction

Distributed systems operate in complex and changing environments. System designers strive to build robust systems that work correctly across a wide range of operating conditions and gracefully degrade their performance in unexpected environments [26]. One major environmental factor that is the most difficult to predict in the design stage is the ultimate size of the system. Many systems start out small but become extremely popular quickly. For example, the Usenet [40] bulletin board system has sustained nearly exponential growth over the last two decades [63]. From 1983 to 1984, the number of Usenet sites doubled from 63,000 to 110,000 [1]. A more recent example is the peer-to-peer Internet telephony service Skype [3], which grew to have over two million active users in the year and a half since its introduction in August 2003 [80]. Rapid growth exposes systems to risks such as network overload since the communication overhead grows with system size.

The common paradigm of systems design is to assume a certain deployment scenario and optimize operations for that scenario. As a result, there is often a spectrum of designs: some that work best with millions of nodes but sacrifice performance when the system is small, and others that optimize for a small system but sacrifice robustness when the system grows too large. System builders face the dilemma of choosing one design from the spectrum, sacrificing either performance or scalability.

Given the unpredictable nature of system growth, an ideal design would not force system builders to assume a certain deployment environment *a* priori. The system would automatically adapt itself to achieve the best performance when its size is small but should scale to millions of nodes without overloading the network. This thesis investigates how one can design and build such an adaptive system in the context of one type of distributed protocol, the Distributed Hash Table (DHT), that is widely used to build peer-to-peer applications.

Peer-to-peer networks are a distributed system design in which nodes self-organize into an overlay network and communicate with each other without spe-

Public infrastructure service	distributed storage	DHash [18], OceanStore [44]
	content distribution network	Coral [22]
	domain name server	CoDoNS [64]
	distributed digital library	OverCite [78, 79]
	cooperative Usenet service	UsenetDHT [75]
	decentralized web cache	Squirrel [37]
Enterprise application	serverless email service	ePost [60]
	cooperative backup	Backup [14, 74]
	distributed file system	CFS [17], PAST [73]
Internet architecture	identity based Internet routing	UIP [21]
	delegation oriented architecture	DOA [85]
	reference resolution	SFR [84]
	Internet indirection service	I3 [76]

Table 1.1: Some examples of peer-to-peer applications that use a Distributed Hash Table to locate data or resource among the participating nodes.

cially designated servers. When more peer nodes join the system, they contribute more network and storage resources, increasing the aggregate system capacity. As a result, peer-to-peer applications can potentially scale to many millions of nodes quickly as there is no need to invest in more powerful server infrastructure as the system grows. Examples of commercial peer-to-peer applications include file sharing applications such as Gnutella [32], Kazaa [28] and BitTorrent [8], and Internet telephony applications such as Skype [3].

A major challenge in peer-to-peer systems is the problem of efficiently locating data or resources among the numerous participating nodes. Distributed Hash Tables (DHTs) organize the peer-to-peer network in a structured manner to provide a hash-table-like *lookup* interface [19]. DHTs allow distributed applications to store and locate data items among a large number of participating nodes. DHTs' simple and useful interface makes them a powerful building block for peer-to-peer applications. Table 1 lists a subset of recently developed research applications that are built on top of a DHT. These applications range from public infrastructure services such as distributed storage infrastructure and content distribution networks, to enterprise applications like email and backup, to novel Internet architectures such as identity based Internet routing.

Given the wide-spread use of DHTs in peer-to-peer applications, it is important that the DHTs themselves are robust and work well across a wide range of deployment scenarios. In order to be robust against rapid network growth or surges in failures, DHTs must bound their communication overhead within the network capacity. This thesis provides a fundamental understanding of the design

tradeoffs of DHTs and develops Accordion, a robust DHT that can adapt itself to offer best performance across a wide range of operating conditions with bounded bandwidth consumption.

1.1 DHT background

A DHT lookup protocol aims to consistently route any arbitrary *key*, typically through multiple intermediate nodes, to the node responsible for the key. Figure 1-1 shows the overall architecture of DHT-based applications. The DHT lookup protocol runs on all participating nodes in the system. A DHT-based application queries the underlying DHT service using a lookup key from a flat identifier space. The DHT nodes cooperate to route the query to the node responsible for the lookup key who replies to the query originator with its IP address. The querying node can then publish the data item associated with the key at the responsible node. Subsequently, any node that knows the key can retrieve the data by routing to the same responsible node.

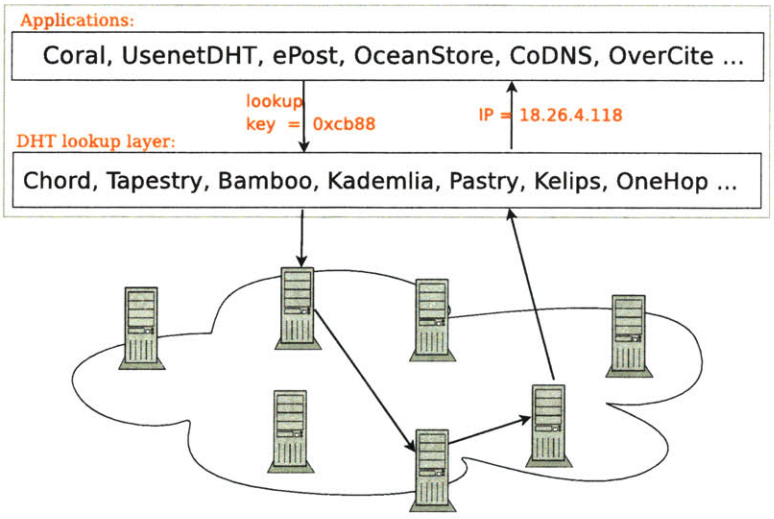
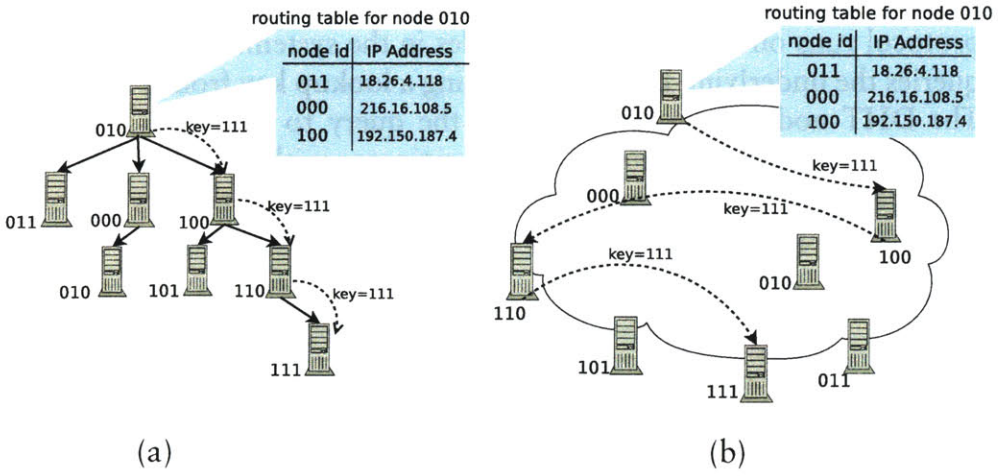


Figure 1-1: Applications use distributed hash tables (DHTs) to route any key to its responsible node among the current set of participating nodes. Data producers in the distributed applications publish data items associated with a known key at the responsible node. Consumers route lookups to the responsible node to retrieve the published data with the same lookup key.

Each DHT node has a unique node key (or node identifier) and all nodes in the system self-organize into an overlay network. An overlay is a network that runs on top of the Internet where a link between two nodes corresponds to an underlying Internet path connecting them. In order to make forwarding decisions, each DHT



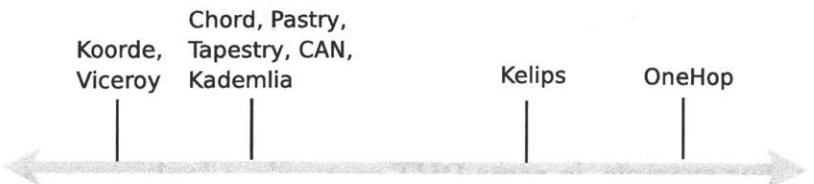
(a) (b)

Figure 1-2: An example of DHT routing with a Plaxton tree routing geometry. In this example, each DHT node maintains a routing table that contains $log(n)$ neighbors, one for each possible prefix of length k ($k = 0...log(n)$) of the node's own identifier. A solid arrow linking two nodes (e.g.010 $\rightarrow$ 100) shows that node 010 contains a routing entry for node 100. The routing tables of all nodes form a Plaxton tree in the identifier space. The graph omits certain links for clarity and only includes those necessary to route any key from node 010. The lookup path for key 111 is shown as dotted links with arrows. Lookups are routed through multiple intermediate nodes based on the routing tables, each hop matching one more prefix digit with the lookup key, to reach the node whose identifier is the same as the key. Figure (a) shows the logical view of the DHT overlay and Figure (b) is the corresponding physical view of how lookups might be routed in the underlying Internet.

node keeps a routing table that contains the identities of some other nodes in the system. If node x appears in node y's routing table, we refer to x as y's neighbor. Typically, a node forwards a lookup to the neighbor that allows the lookup key to make the most progress in the identifier space based on the neighbor's node identifier. For example, in some DHT protocols, a lookup message is forwarded to the neighbor whose node identifier has the most number of matching prefix digits with the key.

If we view the overlay network formed by all routing tables as a static graph, a desirable property is for the graph to have a low diameter so all lookups can finish with few hops. Existing DHTs base their routing structures on a variety of inter-connection graphs such as the Plaxton tree [88], Butterfly networks [56], de Bruijn graph [39], hypercube [66] etc. Figure 1-2(a) shows a logical view of the connections between some nodes which form a Plaxton tree in the identifier space. Each node's routing table has $\log(n)$ entries and each lookup can be routed to its responsible node in $\log(n)$ hops, each hop's node identifier matching one more digit of the lookup key. Figure 1-2(b) shows the corresponding geographic layout of nodes in the underlying Internet. Nodes close to each other in identifier space might be far apart geographically. Therefore, even though lookups finish in only $\log(n)$ hops, each hop may take as long as the average roundtrip time on the Internet.

The latency of a DHT lookup is the time taken to route a message to the responsible node for the key and receive a reply back. Low lookup latency is crucial for building fast DHT-based applications. For example, a UsenetDHT [75] node issues DHT lookups to locate articles stored among many hundreds of nodes distributed over the Internet. The faster DHT lookups can be resolved, the smaller the overall read latency of Usenet articles are. Most existing work on optimizing DHT performance focuses on achieving low latency in static networks. In static networks, lookup latency is determined by the number of lookup hops and the underlying network delay incurred at each hop. Two common strategies are in use to reduce lookup latency; proximity routing and large routing tables. With proximity routing, a node chooses each of its neighbors to be the one with the lowest network delay among a set of qualified nodes. The actual network delay of each hop is reduced even though the number of hops remains the same. An alternative is to increase the per-node routing table size. Intuitively, the more neighbors each node knows about, the fewer hops are required during lookups. For example, each Kelips [34] node has a routing table with $O(\sqrt{n})$ table entries and can finish any lookup in 2 hops. In the extreme, each OneHop [33] node knows all participating nodes in the system and routes a lookup to its responsible node in 1 hop, the lowest achievable static lookup latency. However, these low latencies only apply in a static network when there are no nodes joining or leaving.



Overlay Geometry	de Bruijn, Butterfly	Plaxton tree, Hypercube, XOR	Groups	Complete state
Routing table size	O(1)	O(log n)	$O(\sqrt{n})$	O(n)
Lookup hopcount	O(log n)	O(log n)	2	1

Figure 1-3: Existing DHTs have a fixed routing table size vs. lookup hopcount tradeoffs. n is the total number of nodes in the system. Different DHTs differ mainly on their underlying routing geometry.

1.2 DHT lookups under churn

Real peer-to-peer systems experience *churn*: nodes continuously join and leave the system. Studies of file sharing networks [65] [32] observe that the median time a node stays in the system is on the order of tens of minutes to an hour. Churn poses two problems for DHT routing. First, it causes routing tables to become out of date and to contain stale entries that point to neighbors that are dead or have already left the system. Stale entries result in expensive lookup timeouts as it takes multiple round-trip time for a node to detect a lost lookup message and re-route it through a different neighbor. In static networks, the number of lookup hops and the network delay at each hop determine the end-to-end lookup latency. Under churn, timeouts dominate latency. Second, as new nodes join the system and stale routing entries are deleted, nodes need a way to replenish their routing tables with new entries.

There are many ways to reduce lookup timeouts and bring routing tables close to their ideal state under churn. All techniques to cope with churn incur communication costs in order to evaluate the liveness of existing neighbors and learn about new neighbors. Intuitively, bandwidth consumption increases with the size of a DHT's routing table and the churn rate in the network.

One common cause for a distributed system to break down under rapid growth is when the communication overhead overloads the underlying network to the point of collapse. Examples of such collapse include the wellknown 1986 Internet congestion collapse [38]. Recently, Rhea et. al. [68] have found that certain mechanisms of reacting to failures cause some DHTs to overload the network, albeit

temporarily, with protocol messages and suffer from drastic performance degradation as a result. In order to be robust in scenarios when the networks and failures increase quickly, DHTs must bound their bandwidth use to avoid overloading the network. Furthermore, DHTs should optimize the lookup performance using the bounded bandwidth, adapting to the current deployment environment.

The DHTs proposed in the literature all have a fixed routing table size versus lookup hopcount tradeoff. Figure 1-3 shows the existing DHTs along a spectrum of per-node routing table sizes. As a result, existing DHTs are optimized for a certain deployment scenario. If the available bandwidth is plentiful relative to the churn and size of the network, applications should use protocols with large routing tables like OneHop for best achievable lookup latency. However, to avoid overloading the network when the system grows to millions of nodes or suffers from large degrees of churn, the total bandwidth consumption of the system can not exceed the underlying network's capacity. As a result, applications might prefer protocols with logarithmic routing tables like Chord [77] or Tapestry [88] for scalable performance.

The ideal DHT protocol should be able to adapt its routing table size to provide the best performance using bounded communication overhead. In addition to deciding on the best table size, a DHT should choose the most efficient way of spending bandwidth to keep routing tables up to date under churn. For example, a node could periodically ping each routing entry to check its liveness and search for a replacement entry if an existing neighbor is found to be dead. Chord and Pastry [72] rely on periodic liveness checking. Intuitively, the faster a node pings, the less likely it is that lookups will encounter timeouts. However, periodic pinging generates overhead messages. The more a node pings, the less bandwidth it has for other uses. For example, to reduce lookup timeouts, a node can issue parallel lookup messages to multiple next hop nodes so that one copy of the lookup can make progress even if another is waiting in timeout. The more parallel lookups a node issues, the less likely it is that lookups will be affected by timeouts. Again, redundant parallel lookup messages result in communication overhead. In fact, all techniques to cope with churn require extra communication bandwidth. In other words, churn is a challenge because nodes can only use a finite amount of bandwidth resource. Therefore, the goal of a DHT is not to simply achieve low lookup latency under churn, but to achieve low latency *efficiently* with *bounded* bandwidth overhead. In other words, we are interested in the latency reduction per byte of communication overhead, also referred to as a DHT's latency versus bandwidth tradeoff.

1.3 Evaluating DHT designs with PVC

In order to design a DHT with best lookup latency while consuming only a bounded amount of bandwidth, we need to understand how to efficiently keep routing tables up to date under churn. This thesis develops PVC, a performance vs. cost framework, that helps DHT designers evaluate and compare the efficiencies of different DHTs and design choices. A example of a question that PVC can help answer is whether periodic pinging to check routing entry liveness is a more efficient use of bandwidth than sending parallel lookups.

The main issue that complicates evaluation is the presence of a large number of protocol parameters. A straightforward (but wrong) approach is to first run a DHT with default parameters and record its bandwidth consumption and lookup latency, and then to perform two other experiments: one experiment with nodes issuing more parallel lookups and the other with nodes pinging neighbors more often. It is tempting to conclude that periodic pinging is more efficient than parallel lookups if the second experiment yields more latency reduction per each extra byte of communication overhead. This approach is incorrect because different conclusions might be reached with different initial choices of parameter values. A key challenge is to eliminate differences due solely to parameter choices, so that remaining performance differences reflect fundamental design choices rather than accidental parameter settings.

PVC systematically explores the entire parameter space for a given DHT and plots a family of latency vs. bandwidth tradeoff points, one for each unique parameter setting. The best latency vs. bandwidth tradeoff is not unique: there is a lowest potential lookup latency for each bandwidth consumption. The set of best latency vs. bandwidth tradeoffs can be characterized by a *curve*. PVC extrapolates the best tradeoff curve by computing the overall convex hull segment that lies beneath all tradeoff points. The convex hull segment reveals a DHT's bandwidth efficiency; for any bandwidth consumption x, the corresponding point on the convex hull shows the minimal lookup latency y found after exhaustively exploring a DHT's parameter space. To compare the efficiencies of different DHTs, one needs to examine the relative positions of their corresponding convex hulls. If one convex hull lies completely beneath the other, its corresponding DHT is more efficient. If two convex hulls cross, one DHT is more efficient than the other when limited to low bandwidth use while the other is more efficient if allowed high bandwidth use. In addition to evaluating the efficiencies of entire DHT protocols, PVC can also help DHT designers compare different design choices within a DHT. For example, DHT designers can use PVC to analyze whether parallel lookup uses bandwidth more efficiently than periodic pinging or vice versa.

1.4 Accordion

Accordion is a new DHT protocol that bounds its communication overhead according to a user-specified bandwidth budget and automatically adapts itself to achieve the best lookup latency across a wide range of operating environments. Because Accordion bounds its overhead to avoid overloading the network, it is robust against rapid system growth or unexpectedly high levels of churn.

Accordion's design draws on insights gained from applying PVC analysis to evaluate the efficiencies of existing DHTs and their churn handling techniques. Instead of using a fixed routing table size, Accordion dynamically tunes its table size to achieve the best lookup performance. It maintains a large routing table for best lookup performance when the system is small and relatively stable. When the system grows too large or suffers from high churn, Accordion shrinks its routing table for lower communication overhead.

The problems that Accordion must solve in order to tune itself are how to arrive at the best routing table size in light of the budget and the stability of the node population, how to choose the most effective neighbors to place in the routing table, and how to divide the maintenance budget between acquiring new neighbors and checking the liveness of existing neighbors. Accordion solves these problems in a unique way. Unlike other protocols, it is not based on a rigid data structure such as a de Bruijn graph or hypercube that constrains the number and choice of neighbors. Instead, each node learns of new neighbors as a side-effect of ordinary lookups, but biases the learning so that the density of its neighbors is inversely proportional to their distance in ID space from the node. This distribution allows Accordion to vary the table size along a continuum while still providing the same worst-case guarantees as traditional $O(\log n)$-hop lookup protocols.

A node's bandwidth budget determines the rate at which a node learns. Bigger bandwidth budgets lead to bigger routing tables. Each Accordion node limits its routing table size by evicting neighbors that it judges likely to have failed. It preferentially retains those neighbors that either have been up for a long time or have recently been heard from. Therefore, high churn leads to a high eviction rate. The equilibrium between the learning and eviction processes determines the table size. Compared to existing DHTs, Accordion requires no manual parameter settings. It automatically provides the best lookup performance in a robust way, eliminating the danger of overloading the network as the deployment environment changes.

Performance evaluations show that Accordion keeps its maintenance traffic within the budget over a wide range of operating conditions. When bandwidth is plentiful, Accordion provides lookup latencies and maintenance overhead similar to that of OneHop [33]. When bandwidth is scarce, Accordion has lower lookup latency and less maintenance overhead than Chord [18, 77], even when Chord incorporates proximity and has been tuned for the specific workload.

1.5 Contributions

The contributions of this thesis are:

- A framework for understanding the relationship between communication costs incurred by DHTs and the resulting performance benefits.

- A packet-level simulator, *p2psim* [2], that implements a wide variety of popular DHTs including their performance optimization features under churn.

- A comparative performance study of a variety of different DHTs and an evaluation of the relative efficiencies of different design choices techniques. Some of the main findings include:

    1. A node should expand its routing table to efficiently use extra bandwidth.
    2. Parallelizing lookups is more efficient than periodic pinging of routing tables at reducing the effects of timeouts.
    3. Learning opportunistically from lookup traffic is the most efficient way of acquiring new routing entries.
    4. A node should bound the staleness of its routing entries to reduce lookup timeouts.

- A new DHT protocol, Accordion, which is built on the insights gained from our performance study of existing designs. Accordion is the first DHT protocol that achieves all of the following properties:

    1. bounded bandwidth overhead.
    2. elimination of manual tuning of parameters.
    3. automatic adaptation to churn and workload.
    4. use of systematic efficiency analysis in the protocol design.

- A working implementation of Accordion and an evaluation of the implementation using wide area network emulations.

Apart from the above technical contributions, this thesis presents a different style of designing distributed protocols. We argue that a distributed protocol should be conscious of its communication overhead in order to stay robust across a wide range of operating environments. We demonstrate how to design such a cost aware DHT that uses the bandwidth resource sparingly and efficiently.

1.6 Thesis Organization

The thesis is organized as follows: the first chapter describes the motivation and goal of the thesis. Chapter 2 introduces the PVC evaluation framework and explains how PVC explores the parameter space to find a DHT's latency vs. bandwidth tradeoff and compare the efficiencies of different design choices. Chapter 3 and 4 provide an extensive simulation study of existing DHT protocols using PVC. We simulate five well-known protocols (Tapestry, Chord, Kelips, Kademlia and OneHop) in *p2psim*. We use PVC to evaluate how efficiently different design choices use additional bandwidth for better lookup performance.

Chapter 5 and 6 describe the design and implementation of the Accordion lookup protocol. We explain the design principles of Accordion's routing table maintenance algorithm followed by the actual protocol details. We also present an C++ implementation of Accordion that is integrated in the DHash [18] distributed hash table software distribution. Chapter 7 evaluates Accordion and compares it to existing DHTs both in simulation and with real implementations.

Chapter 8 summarizes related work and Chapter 9 concludes.

Chapter 2

The PVC Framework

In order to design a DHT with best lookup performance, we need to understand how to use a bounded amount of bandwidth most *efficiently*. The *efficiency* of a DHT measures its ability to turn each extra byte of maintenance communication into reduced lookup latency. How does one evaluate and compare the efficiencies of different existing DHTs? Which churn-handling technique offers the most efficient use of bandwidth? This chapter develops a performance vs. cost analysis framework (PVC) that helps DHT designers answer these questions.

2.1 Challenges

Two challenges exist in evaluating DHT lookup protocols. First, most protocols can be tuned to have low lookup latency by including features such as aggressive membership maintenance, faster routing table liveness checking, parallel lookups, or a more thorough exploration of the network to find low delay neighbors. Any evaluation that examines how a DHT performs along one dimension of either cost (in terms of bandwidth consumed) or performance (in terms of lookup latency) is flawed, since a DHT can "cheat" by performing extremely well on the axis being measured but terribly on the other. Thus a comparison of DHT lookup protocols must consider their performance and cost simultaneously, i.e. the *efficiency* with which they exploit bandwidth to reduce latency. A DHT's efficiency can be characterized by a performance vs. cost tradeoff *curve*: at any given bandwidth, there is one best achievable latency. However, the efficiency of a DHT cannot be measured by a single number summarizing the ratio between a protocol's bandwidth consumption and its lookup latency, as the tradeoffs between performance and cost does not necessary follow a linear relationship. Nevertheless, one wants to be able to compare the *efficiency* of one protocol to another.

The second challenge is to cope with each protocol's set of tunable parameters (*e.g.*, liveness checking interval, lookup parallelism etc.). The best parameter values for a given workload are often hard to predict, so there is a danger that a

performance evaluation might reflect the evaluator's parameter choices more than it reflects the underlying algorithm. In addition, parameters often correspond to a given protocol feature. A good framework should allow designers to judge the extent to which each parameter (and thus each feature) contributes to a DHT's overall bandwidth efficiency.

2.2 The PVC Approach

In response to these two challenges, we propose PVC, a performance vs. cost framework and evaluation methodology for assessing DHT lookup protocols, comparing different design choices and evaluating new features. The PVC techniques are general enough to be used in all types of DHT evaluations. However, as the analysis often involves many hundreds or even thousands of different experiments in exploring a DHT's parameter space, it is much more convenient to evaluate DHTs using packet-level simulations.

PVC uses the average number of bytes sent per node per unit time as the cost metric. This cost accounts for all messages sent by a node, including periodic routing table pinging traffic, lookup traffic, and node join messages. PVC ignores routing table storage costs because communication is typically far more expensive than storage. The main cost of routing state is often the communication cost necessary to keep the state up to date under churn.

PVC uses two metrics to characterize a DHT's performance: the average latency of successful lookups and percentage of failed lookups. PVC only incorporates lookup hop-count indirectly, to the extent that it contributes to latency. Thus, our evaluations can easily incorporate latency optimization techniques like proximity neighbor selection (PNS [29]) where nodes preferentially choose low delay nodes as neighbors. Similarly, PVC does not explicitly account for lookup timeouts. Timeouts contribute to increased lookup latency as it requires multiples of roundtrip time for a node to detect a lost lookup packet sent to a dead neighbor.

2.2.1 Overall Convex Hulls

How to extract a DHT's performance vs. cost tradeoff curve while avoiding the danger of arbitrary parameter settings? How does one compare different DHTs?

PVC systematically simulates a DHT with different combinations of parameter values. For each parameter combination, PVC plots the performance and cost measured from the experiment on a graph with total bandwidth usage on the x-axis and average lookup latency or failure rate on the y-axis. For example, in Figure 2-1, each of the many hundred points corresponds to a different parameter combination for Kelips under a particular workload. A point that lies to the lower left of another point is more efficient as its corresponding parameter combination results in both lower lookup latency *and* lower bandwidth consumption.

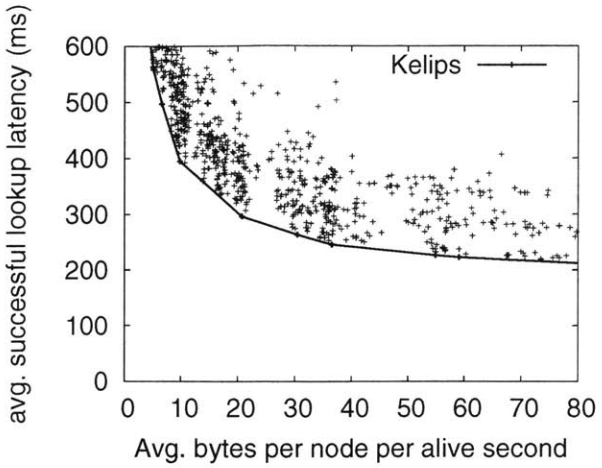


Figure 2-1: The best performance vs. cost tradeoffs in Kelips under a specific workload. Each point represents the average lookup latency of successful lookups vs. the communication cost achieved for a unique set of parameter values. The convex hull (solid line) represents the best achievable performance/cost combinations.

To characterize the efficiency of a DHT, we need to find the best set of performance vs. cost tradeoff points that correspond to the optimal parameter settings. As can be seen in Figure 2-1, there is no single best performance vs. cost tradeoff point. Instead, there is a set of best points: for each cost, there is a smallest achievable lookup latency, and for each lookup latency, there is a smallest achievable communication cost. The curve connecting these best points is the overall *convex hull* segment (shown by the solid line in Figure 2-1) that lies beneath and to the left of all points. A convex hull segment always goes up to the left of the graph as bandwidth decreases. This means that there is no parameter combination that simultaneously produces both low latency and low bandwidth consumption. Figure 2-1 also shows that the ratio of latency to bandwidth does remain constant but decreases as bandwidth increases. Therefore, we cannot use a single ratio value, but have to resort to the entire overall convex hull, to represent the *efficiency* of a DHT.

The relative positions of overall convex hulls can be used to compare different DHTs. A convex hull segment that lies to the lower left of another segment corresponds to a more efficient DHT as it achieves lower latency at any given bandwidth. In the example shown in Figure 2-2, one can conclude that Kelips is more efficient than Kademlia since Kelips' convex hull segment lies beneath the overall hull of Kademlia.

The convex hull in Figure 2-1 and Figure 2-2 only outlines the most efficient parameter combinations found by PVC. It is possible that better combinations exist but that PVC fails to find them. In addition, the convex hull in Figure 2-1

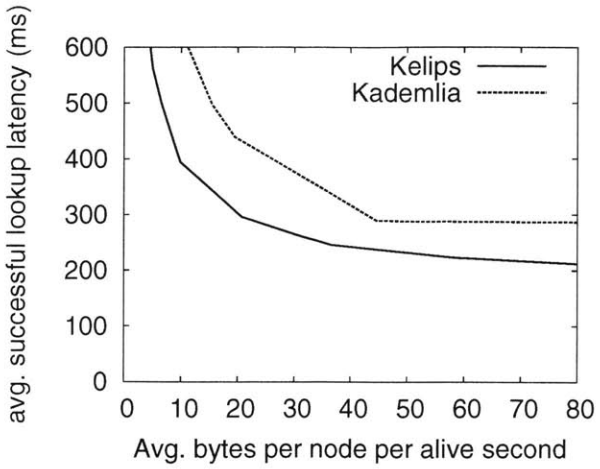


Figure 2-2: The overall convex hulls of Kelips and Kademlia under a specific workload. Kelips is more bandwidth efficient thatn Kademlia as its overall convex hull lies to the lower left of that of Kademlia's.

is only for a specific workload and churn scenario being simulated. The best parameter values (thus the overall convex hulls) might change as workloads or the churn rates change. Therefore, the convex hull only outlines a DHT's maximal efficiency in theory. A DHT operator would have to adjust the protocol parameters manually under known workload and churn scenario in the absence of a self-tuning protocol [10, 71].

2.2.2 Parameter Convex Hulls

Figure 2-1 shows the combined effect of many parameters. What are the actual parameter values that correspond to points on the convex hull? How sensitive is the overall convex hull to the different values of each parameter? In other words, how important is it to tune each parameter to achieve the best efficiency? To answer these questions, we calculate a set of parameter convex hulls, one for each value of the parameter under study. Each parameter convex hull is generated by fixing the parameter of interest and varying all others. Each parameter hull represents the best possible performance vs. cost tradeoffs for a fixed parameter value.

Figure 2-3 and Figure 2-4 present two sets of parameter convex hulls for two different parameters, each compared with the overall convex hull. Each parameter convex hull in Figure 2-3 shows the best latency vs. bandwidth tradeoffs found by fixing $param1$ to a specific value while exploring all other parameters. The parameter convex hull for $param1 = 32$ matches the overall convex hull well which means that $param1$ has a single best value for this workload and does not need to be tuned. In comparison, among the set of parameter convex hulls in

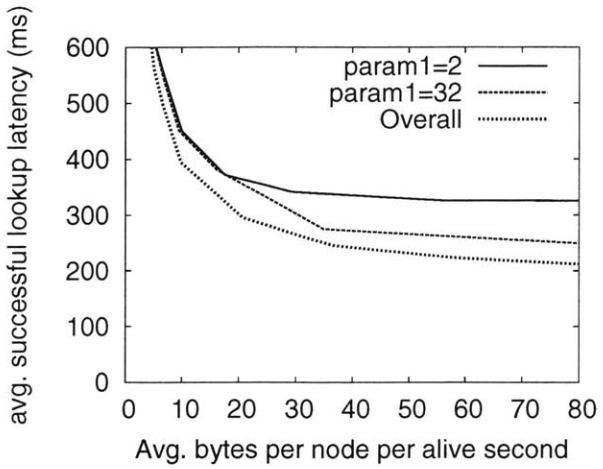


Figure 2-3: Overall convex hull and the parameter convex hulls for two values of param1. Parameter hulls for other values of param1 are omitted for clarity. Overall convex hull always lies to the bottom left of all parameter hulls, however, it is well approximated by the parameter convex hull for $param1 = 32$.

Figure 2-4 for $param2$, no one parameter hull lies entirely along the overall hull; rather, the overall hull is made up of segments from different parameter hulls. This suggests that $param2$ should be tuned to different values depending on the desired latency or bandwidth consumption.

Figure 2-3 and Figure 2-4 show how to use the positions of parameter convex hulls relative to the overall convex hull to find the relative "importance" of a parameter visually: is it necessary to tune the parameter to different values in order to achieve the maximal efficiency as outlined by the overall convex hull? To automate the above visual process to identify relative parameter importance, we calculate the area between the parameter's hulls and the overall convex hull over a fixed range of the x-axis (the cost range of interest). Figure 2-5 shows an example. The smaller the area, the more closely a parameter hull approximates the best overall hull. The minimum area over all of a parameter's values indicates how important it is to tune the parameter. The bigger the minimum area, the more important the parameter since there is a larger potential for inefficiency by setting the parameter to a single value. Figure 2-3 corresponds to a parameter with nearly zero minimum area, while Figure 2-4 shows a parameter with a large minimum area. Hence, it is relatively more important to tune the latter.

There is a relationship between this notion of parameter importance and the efficiency of DHT mechanisms. Suppose that an important parameter affects how much network bandwidth a particular DHT mechanism consumes; for example, the parameter might control how often a DHT pings its routing table entries. If more network capacity becomes available, then any re-tuning of the DHT's

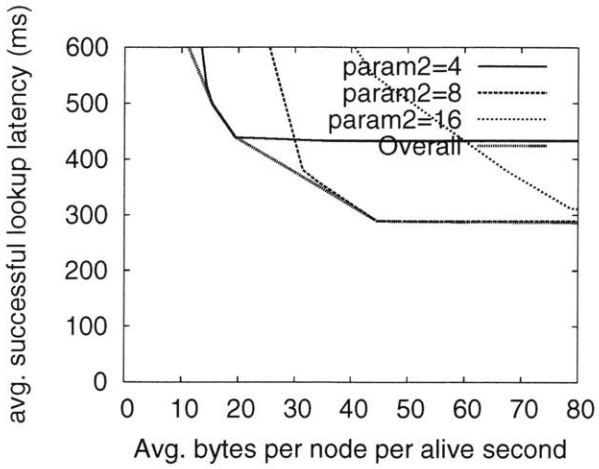


Figure 2-4: Overall convex hull and the parameter convex hulls for two values of param2. Parameter hulls for other values of param2 are omitted for clarity. Unlike Figure 2-3, the overall convex hull cannot be approximated well by any of the parameter convex hulls. Rather, different parameter hulls make up for different portions of the overall convex hull.

parameters to make best use of the new capacity will likely require tuning this important parameter. That is, important parameters have the most effect on the DHT's ability to use *extra* communication bandwidth to achieve low latency, and in that sense important parameters correspond to efficient DHT mechanisms.

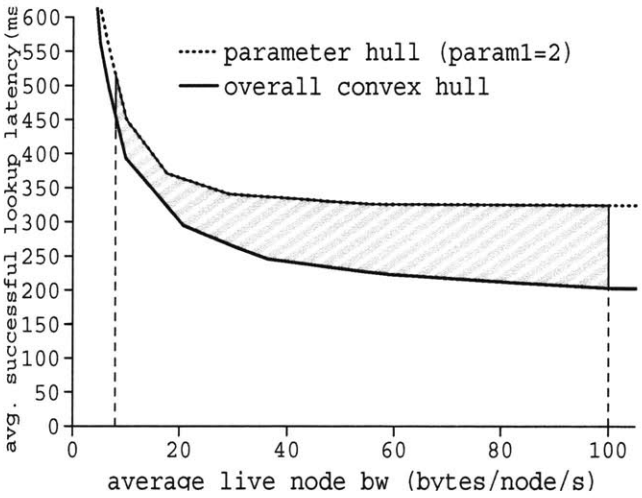


Figure 2-5: The efficiency of param1 with a value of 2 (from Figure 2-3 compared with the efficiency of the overall convex hull. The striped area between the two curves represents the efficiency difference between param1=2 and the best configurations between a cost range of 8 to 100 bytes/node/s.

Chapter 3

Evaluation Methods

We use simulations to study the bandwidth efficiencies of existing DHT lookup protocols. As part of the thesis, we developed *p2psim*, a discrete-event packet-level simulator. We choose to study five existing DHTs: Tapestry [88], Chord [77], Kelips [34], Kademlia [58] and OneHop [33]. Together, these protocols cover a wide range of design choices such as a directed identifier space, parallel lookups, pro-active flooding of membership changes, periodic pinging of routing entries and a spectrum of routing table sizes ranging from $O(\log n)$, to $O(\sqrt{n})$, to $O(n)$. This chapter provides an overview of different DHT lookup protocols, identifying the protocol parameters and relating them to the different design choices. We also describe the *p2psim* software and experimental setup for the performance study in Chapter 4.

3.1 Protocol Overview

A DHT lookup protocol aims to route lookups to the node responsible for the desired key. We are only concerned about lookups, ignoring the actual data fetch from the responsible node. In this section, we describe the five DHTs as they are implemented in *p2psim*. For each DHT, we explicitly identify the protocol parameters and list the range of parameter values explored by the simulation experiments.

§ Tapestry

The ID space in Tapestry is structured as a Plaxton tree. A Tapestry node ID can be viewed as a sequence of l base-b digits. The node with the maximum number of matching prefix digits with the lookup key is responsible for the key.

A Tapestry node's routing table contains approximately $\log_b(n)$ levels, each with b distinct ID prefixes. Routing entries in the m^{th} level share a prefix of length $m-1$ digits with the node's own identifier, but differ in the m^{th} digit. Each entry

Parameter		Range
Base	b	2 – 128
Stabilization interval	t_{stab}	18 sec – 19 min
Number of backup nodes	n_{redun}	1 – 8
Number of nodes contacted during repair	n_{repair}	1 – 10

Table 3.1: Tapestry parameters

may contain up to n_{redun} nodes, sorted by network delay. The most proximate of these nodes is the entry's *primary neighbor*; the others serve as *backup neighbors*. Tapestry also uses a nearest neighbor algorithm [36] to populate its routing table entries with physically nearby nodes.

Nodes forward a lookup by resolving successive digits in the lookup key (*prefix-based routing*). When no more digits can be resolved, an algorithm known as *surrogate routing* determines exactly which node is responsible for the key [88]. Routing in Tapestry is recursive: each intermediate node is in charge of forwarding a lookup to the next hop, retransmitting lost lookups if necessary. A lookup terminates at its responsible node who directly replies to the lookup originator with its own identity.

In a static network, each base-b Tapestry node has a routing table of $(b-1) \cdot \log_b(n) \cdot n_{redun}$ entries and can finish any lookup in $O(\log_b n)$ hops.

For lookups to be correct, at least one neighbor in each routing prefix must be alive. Tapestry's stabilization process pings each primary neighbor every t_{stab} seconds. If the node is found to be dead, the next closest backup in that entry (if one exists) becomes the primary. When a node declares a primary neighbor dead, it contacts some number of other neighbors (n_{repair}) asking for a replacement. Table 3.1 lists the Tapestry parameters varied in our simulations.

§ Chord

Chord structures its identifier space as a clockwise circle. The node responsible for a key y is its successor (*i.e.*, the first node whose ID is equal to y, or follows y in the ID space) using consistent hashing [41]. In Chord, a lookup for a key terminates at the key's *predecessor*, the node whose ID most closely precedes the key. The predecessor returns the identity of the key's successor to the originator node.

A Chord node keeps two types of neighbors: successors and fingers. Each node keeps n_{succ} successors that immediately follow a node's own identifier on the ring. A node also keeps fingers whose IDs lie at exponentially increasing fractions of the ID space away from itself. Chord uses the Proximity Neighbor Selection

Parameter		Range
Base	b	2 – 128
Finger stabilization interval	t_{finger}	18 sec – 19 min
Number of successors	n_{succ}	8,16,32
Successor stabilization interval	t_{succ}	18 sec – 4.8 min

Table 3.2: Chord parameters

(PNS) method discussed in [18, 29, 72]. To obtain each i^{th} PNS finger, a node looks up the node with ID $(\frac{b-1}{b})^{i+1}$ away from itself to retrieve its successor list and chooses the neighbor with the lowest network delay to itself as the i^{th} PNS finger. Chord can route either iteratively or recursively [77]; this thesis presents results for the latter.

In a static network, each base-b Chord node has a routing table of $(b-1)\cdot\log_b(n)$ fingers and n_{succ} successors and can resolve all lookups in $O(\log_b n)$ hops.

A Chord node stabilizes its successors and fingers separately. Each node periodically (t_{succ} seconds) retrieves its immediate successor's successor list and merges with its own. As a separate process, each node also pings all fingers every t_{finger} seconds. For each finger found dead, the node issues a lookup to find a replacement PNS finger. Table 3.2 lists the Chord parameters that we vary in our simulations.

§ Kelips

Kelips divides the identifier space into $g \approx \sqrt{n}$ groups. A node's group is its ID modulo g. Each node's routing table contains an entry for each other node in its own group, and $n_{contact}$ "contact" nodes from each of the foreign groups. Kelips does not define an explicit mapping of a given lookup key to its responsible node. Instead, Kelips replicates key/value pairs among all nodes within a key's group. Lookups that have values stored under the keys terminate when it reaches a node storing the corresponding key/value pair. However, since there is no responsible node for a key, lookups for non-existent keys have higher latency as they cannot terminate properly. For this reason, the variant of Kelips in this thesis defines lookups only for IDs of node that are currently in the network. The originating node executes a lookup by asking a contact in the lookup key's group for the identity of the node whose identifier matches the key, and then (iteratively) contacting that node. If that fails, the originator tries routing the lookup through other contacts for that group, and then through randomly chosen routing table entries.

In a static network, a Kelips node's routing table contains $\sqrt{n}+n_{contact}\cdot(\sqrt{n}-1)$ neighbors and can finish all lookups within two hops.

Nodes gossip periodically every t_{gossip} seconds. A node chooses one random

Parameter		Range
Gossip interval	t_{gossip}	10 sec – 19 min
Group ration	r_{group}	8, 16, 32
Contact ration	$r_{contact}$	8, 16, 32
Contacts per group	$n_{contact}$	2, 8, 16, 32
Routing entry timeout	t_{out}	6, 18, 30 min

Table 3.3: Kelips parameters

Parameter		Range
Nodes per entry	k	2 – 32
Parallel lookups	α	1 – 32
Number of IDs returned	n_{tell}	2 – 32
Stabilization interval	t_{stab}	4 – 19 min

Table 3.4: Kademlia parameters

contact and one neighbor within the same group to send a random list of r_{group} neighbors from its own group and $r_{contact}$ contact nodes. Routing table entries that have not been refreshed for t_{out} seconds expire. Nodes learn round trip times (RTTs) and liveness information from each RPC, and preferentially route lookups through low RTT contacts. Table 3.3 lists the parameters we use for Kelips. We use $g = \sqrt{1024} = 32$ in our Kelips simulations using a network of 1024 nodes.

§ Kademlia

Kademlia structures its ID space as a tree. The distance between two keys in ID space is their exclusive or, interpreted as an integer. The k nodes whose IDs are closest to a key y store a replica of of the key/value pair for y. Each node has $\log_2 n$ routing buckets that each stores up to k node IDs sharing the same binary prefix of a certain length.

In a static network, each Kademlia node has a routing table containing $\log_2(n) \cdot k$ entries and can resolve all lookups in $O(\log_2 n)$ hops.

Kademlia performs iterative lookups: a node x starts a lookup for key y by sending parallel lookup RPCs to the α neighbors in x's routing table whose IDs are closest to y. A node replies to a lookup RPC by sending back a list of the n_{tell} entries that are closest to y in ID space from its routing table. With each lookup RPC, a node learns RTT information for existing neighbors or previously unknown nodes to be stored in its routing bucket. The lookup originator x always tries to keep α outstanding RPCs. A lookup terminates when a replica replies with

Parameter		Range
Slices	n_{slices}	3,5,8
Units	n_{units}	3,5,8
Ping/Aggregation interval	t_{stab}	4 sec – 64 sec

Table 3.5: OneHop parameters

the value for key y, or until the last k nodes whose IDs are closest to y did not return any new node ID closer to y. Like Kelips, Kademlia also does not have an explicit mapping of a key to its responsible node, therefore terminating lookups for non-existent keys requires extra communication with the last k nodes. For this reason, we also use node IDs as lookup keys in Kademlia experiments and the last step in a lookup is an RPC to the target node. Our Kademlia implementation favors communication with proximate nodes.

A node periodically (t_{stab}) examines all routing buckets and performs a lookup for each bucket's binary prefix if there has not been a lookup through it since the past t_{stab} seconds. Kademlia's stabilization only ensures that at least *one* entry in each bucket was alive in the past t_{stab} seconds and a node may still forward lookups through a neighbor that has not been contacted with t_{stab} seconds. In contrast, Tapestry and Chord's ensure *all* routing entries were alive in the past t_{stab} (t_{finger}) seconds and hence there exist no routing entries older than t_{stab}. Table 3.4 summarizes the parameters varied in our Kademlia simulations.

§ OneHop

Similar to Chord, OneHop [33] assigns a key to its successor node on the ID circle using consistent hashing [41]. Each OneHop node knows about every other node in the network and forwards a lookup to its apparent successor node among all routing table entries.

In a static network, each OneHop node has a routing table of size n and all lookups terminate in one hop.

The ID space is divided into n_{slice} slices and each slice is further divided into n_{unit} units. Each unit and slice has a corresponding leader. OneHop pro-actively disseminates information regarding all join and leave events to all nodes in the system through the hierarchy of slice leaders and unit leaders. A node periodically (t_{stab}) pings its successor and predecessor and notifies its slice leader of the death of successor or predecessor. A newly joined node sends a join notification event to its slice leader. A slice leader aggregates notifications within its slice and periodically (t_{stab}) informs all other slice leaders about notifications since the last update. A slice leader disseminates notifications from within its slice and from other slices to each unit leader in its own slice. Notifications are further propagated to all

Design Choices	Tapestry	Chord	Kademlia	Kelips	OneHop
Separate lookup correctness from performance	-	t_{succ}	-	-	-
Vary routing table size	b, n_{redun}	b	k	$n_{contact}$	-
Bound routing entry freshness	t_{stab}	t_{finger}	t_{stab}?	t_{out}?	-
Parallelize lookups	-	-	α, n_{tell}	-	-
Learn new nodes from lookups	-	-	yes	-	yes

Table 3.6: Design choices and their corresponding parameters. We put a question mark next to t_{stab} and t_{out} because Kademlia and Kelips nodes still may forward lookups to neighbors that have not been heard for more than t_{stab} and t_{out} seconds even though the two parameters have some effects on the routing entry freshness.

nodes within a unit through piggybacking on each node's ping messages. Table 3.5 shows the range of OneHop's parameters varied in simulations.

3.2 Summary of Design Choices

Table 3.6 summarizes the correspondence between different design choices and the parameters for all the protocols.

There are a number of common design choices among the DHTs in Table 3.6. First, most protocols (except OneHop) have some flexibility in the size of the per-node routing table. A bigger routing table results in fewer lookup hops at the cost of extra routing table maintenance bandwidth. Chord and Tapestry use the base parameter (b) to adjust the number of neighbors as well as the portions of ID space these extra neighbors are sampled from. In contrast, Kademlia and Kelips just vary its per-node routing table size by keeping more candidate neighbors for each routing entry using k and $n_{contact}$ respectively. Second, most protocols control the freshness of routing entries implicitly or explicitly with some threshold time intervals. Chord and Tapestry check the liveness of each neighbor every t_{finger} or t_{stab} seconds. Therefore, no neighbors exist in routing tables who have not been heard in the last t_{finger} (or t_{stab}) seconds. Kelips nodes evict neighbors after t_{out} seconds of inactivity. However, since a routing entry has already aged in another node's routing table before it is propagated to the current node, the corresponding neighbor can endure a period of inactivity longer than t_{out} seconds before it is evicted. Therefore, unlike t_{finger} and t_{stab} in Chord/Tapestry, Kelips' t_{out} affects but does not bound routing entry freshness. Kademlia tries to ensure at least one neighbor was heard within the last t_{stab} seconds in each bucket, but it may still use an older entry if it points to a more proximate neighbor. Unlike all other protocols, OneHop does not expire neighbors based on timeout threshold. A

OneHop node relies on explicit leave event notifications to evict routing entries. As notification events are delayed or lost, routing entries may be stale and point to neighbors that are dead.

Table 3.6 also shows a number of unique design choices in some DHTs. For example, Chord separates routing entries into those that ensure correctness of lookups (successors) and those that can speed up lookups (fingers). Kademlia parallelizes lookup RPCs if $\alpha > 1$ to reduce the effect of lookup timeouts. A Kademlia lookup originator also learns n_{tell} entries for each lookup message it sends. OneHop learns from lookups differently: the lookup originator regenerates a leave event for each lookup timeout it incurs and a join event for each additional lookup hops it takes. These newly generated events will be propagated to all other nodes as the originator assumes the original events have been lost.

3.3 Simulation Software: *p2psim*

To fairly evaluate and compare existing DHTs under a common framework, we have developed a discrete event packet level simulator, *p2psim*. Because p2psim uses cooperative user-level threads to pass control between protocol executions at different nodes, DHT implementations in *p2psim* resemble their algorithm pseudo-code, which makes them easy to understand. We have implemented six existing DHT protocols in *p2psim*: Chord, Tapestry, Kademlia, Kelips, OneHop and Koorde. Tapestry, OneHop and Koorde are written by their original protocol designers.

To complete one simulation run in *p2psim*, there should be three input configuration files; a protocol parameter file, a topology configuration and a churn/workload specification. The protocol parameter file describes the value of each tunable parameter for the DHT under study. The topology configuration file specifies the pair wise node delay matrix in the system. The churn/workload file describes the sequence of node join and leave events (or node lifetime distribution) and how frequently each node issues a lookup message.

p2psim records the start and completion time of each lookup to calculate its latency, the time required to route a message to a key's responsible node and back. Lookup hopcount, link propagation delay and timeouts are the only factors that contribute to the measured latency. In particular, *p2psim* does not simulate link capacity nor queuing delay as DHT lookups involve only key lookups as opposed to data retrieval. In the interest of a fair comparison, all DHTs implemented in *p2psim* follow a few common guidelines for dealing with lookup timeouts. All DHTs recover from timeouts by retrying the lookup through an alternate neighbor. In p2psim, all protocols time out individual messages after an interval of three times the round-trip time to the target node, though more sophisticated techniques are possible [10,18,68,89]. Following the conclusions of previous studies [10,68],

a node encountering a timeout to a particular neighbor proceeds to an alternate node if one exists, but the neighbor is only declared failed after five consecutive RPC timeouts occur.

When a lookup originator receives a reply from the responsible node, it checks the correctness of the identity of the responsible node against *p2psim*'s global knowledge of the current set of live nodes. A lookup is considered failed if it returns the wrong node among the current set of participating nodes (i.e. those that have completed the join procedure correctly) at the time the sender receives the lookup reply, or if the sender receives no reply within some timeout window. Rhea et. al. [68] proposes to check the consistency of a lookup by performing ten lookups for the same key from different originating nodes. If there is a majority of results in the ten lookups, all nodes in the majority are considered as seeing consistent results. The definition of a correct lookup in *p2psim* has stronger guarantees than Rhea's definition of lookup consistency: a correct lookup in *p2psim* is always consistent, but the converse is not true.

3.4 Simulation Environment

We explore the parameter space of existing protocols in simulation using all combinations of parameter values within the range specified in Section 3.1.

The simulated network, unless otherwise noted, consists of 1024 nodes. For realistic inter-node latencies, we use the measured pairwise latencies between 1024 wide area DNS servers. We "trick" two unmodified recursive DNS servers to report the latency between them using King method [31]. A recursive DNS server y resolves a domain name served by y' by directly sending a DNS lookup message to server y'. In our measurement experiments, we issue a DNS lookup from our local machine x to y, resolving a domain name belonging to server y' and record the latency ($delay(y')$) required. We then separately measure the latency to server y ($delay(y)$) by issuing it a DNS query in its own domain. The difference between the two latencies ($delay(y') - delay(y)$) is the estimated roundtrip time between y and y'.

The median round-trip delay between node pairs in our dataset is 156 ms and the average is 178 ms. Since each lookup for a random key must terminate at a specific, random node in the network, the mean latency of the topology serves as a lower bound for the mean DHT lookup latency.

The amount a protocol must communicate to keep routing tables up to date depends on how frequently nodes join and crash (the churn rate). For the most part, the total bandwidth consumed by a protocol is a balance between routing table maintenance traffic and lookup traffic, so the main characteristic of a workload is the relationship between lookup rate and churn rate. This thesis investigates two workloads, one that is *churn intensive* and one that is *lookup intensive*. In both

workloads, each node alternately crashes and re-joins the network; the interval between successive events for each node is exponentially distributed with a mean of one hour. The choice of mean session time is consistent with past studies of peer-to-peer networks [32]. Each time a node joins, it uses a different IP address and DHT identifier. In the churn intensive workload, each node issues lookups for random keys at intervals exponentially distributed with a mean of 600 seconds. In the lookup intensive workload, the average lookup interval is 9 seconds. Unless otherwise noted, all figures are for simulations done in the churn intensive workload. Each simulation runs for six hours of simulated time; statistics are collected only during the second half of the simulation.

PVC analysis requires explicitly plotting a protocol's bandwidth consumption on the x-axis. We calculate the total number of bytes sent by all nodes during the second half of the simulation experiment divided by the sum of the seconds that all nodes were up. This is the normalized bandwidth consumption per node over the entire experiment and node population and is shown on the x-axis. The bandwidth consumption includes all messages sent by nodes, such as lookup, node join and routing table maintenance traffic. The size in bytes of a message is counted as 20 bytes (for packet overhead) plus 4 bytes for each IP address or node identifier mentioned in the message. The y-axis indicates lookup performance either in the average latency of successful lookups or failure rate.

Chapter 4

A Study of DHT Design Choices

This chapter presents the results of applying PVC analysis in the simulation study of five existing DHTs. We compare DHTs against each other using their overall convex hulls and analyze how effective each churn-handling technique is at improving a DHT's overall bandwidth efficiency.

4.1 Overall comparison

Figures 4-1 and 4-2 present the overall convex hulls for failure rate and the average latency of successful lookups, respectively. Each convex hull is calculated from hundreds or thousands of points derived by simulating all combinations of the parameter values listed in Section 3.1 for each DHT. Each convex hull outlines a DHT's best achievable performance vs. cost tradeoffs with the optimal parameter settings. All convex hulls have similar overall characteristics: latencies go up with smaller bandwidth consumption meaning that there is no combination of parameter values that results in both low lookup latency (or low failure rate) and low bandwidth consumption. The convex hulls go down at higher bandwidth because there are parameter values that improve lookup latency (or failure rate) at the cost of increased bandwidth consumption. The average round trip time ($178ms$) between two nodes in the simulated network lower-bounds the best possible lookup latency since a lookup is issued for a randomly chosen key and hence is routed to a random node in the network.

The convex hulls of different DHTs can be far apart; at any given bandwidth, the best achievable failure rates or latencies of different DHTs can differ significantly. For example, at 20 bytes/node/s, OneHop achieves 200ms average lookup latency and Kademlia achieves 420ms with the best parameter settings. A convex hull that lies to the bottom left of another hull indicates better bandwidth efficiency for its corresponding DHT. Therefore, roughly speaking, the order of protocols in terms of their failure rate vs. bandwidth tradeoffs from better to worse is: OneHop, Chord, Kelips, Tapestry and Kademlia. Similarly, the order of

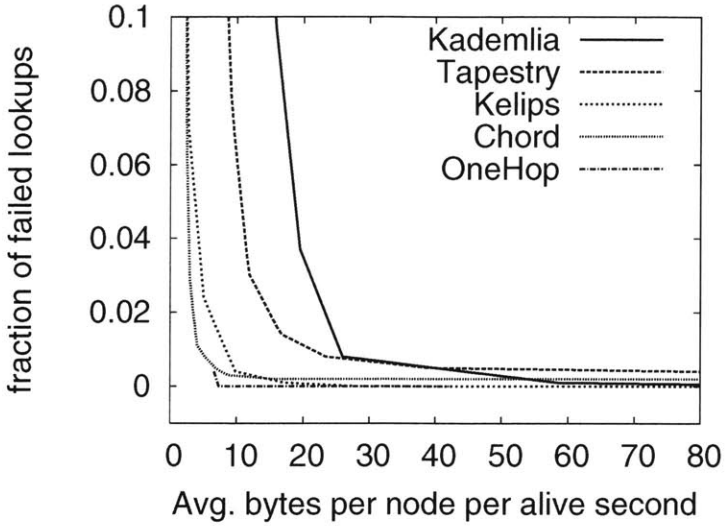


Figure 4-1: Overall convex hulls showing failure rate vs. bandwidth tradeoffs of all DHTs, under the churn intensive workload. The failure rate for OneHop is less than 0.005 for all points on its hull. A convex hull that lies towards the bottom left of another hull indicates that its corresponding DHT can be tuned to have lower failure rate while consuming the same amount of bandwidth.

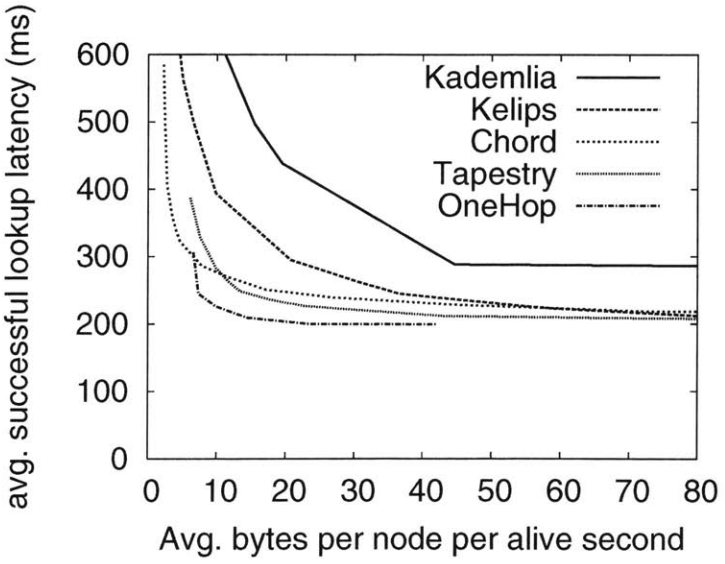


Figure 4-2: Overall convex hulls showing the average latency of successful lookups vs. bandwidth for all DHTs, under the churn intensive workload. The average RTT between two nodes (178ms) serves as a lower bound for the best achievable lookup latency.

protocols in terms of better lookup latency vs. bandwidth tradeoffs is: OneHop, Tapestry, Chord, Kelips, Kademlia. However, there are some caveats. Firstly, not all convex hulls have the same minimal bandwidth use. For example, apart from Chord, all DHTs consume more than 7 bytes/node/s in Figure 4-2. This suggests that Chord is more efficient than others at using a small amount of bandwidth to handle churn. Secondly, some convex hulls cross each other. For example in Figure 4-2, the convex hull of Chord crosses that of Kelips at 50 bytes/node/s. This crossing suggests that Chord is more efficient than Kelips at small bandwidth but the converse is true at large bandwidth. The next section will investigate the efficiencies of different design choices in each protocol that lead these differences in their performance vs. cost tradeoffs.

4.2 PVC Parameter Analysis

None of the existing DHT lookup protocols explicitly controls its performance vs. cost tradeoffs as outlined by Figure 4-1 and 4-2. The various bandwidth consumptions and lookup performance are the indirect consequence of setting a DHT's many parameters to different values. That is, the convex hulls in Figures 4-1 and 4-2 are the result of an exhaustive search for the best parameter values. What parameter values produced the best tradeoffs that make up the convex hull? More importantly, if the available bandwidth changes, what are the parameters that need to be re-adjusted to optimize lookup performance?

Each parameter in a DHT corresponds to a design choice. Different design choices compete with each other in using extra bandwidth to improve lookup performance. For example in Chord, setting a bigger base (b) or a smaller stabilization interval (t_{finger}) can both lower lookup latency at the cost of increased bandwidth consumption. Therefore, measuring the performance benefits by adjusting a single parameter in isolation can be misleading as it ignores other competitive choices of using bandwidth. We solve this problem with PVC parameter convex hull analysis. Instead of measuring the performance benefits of adjusting the parameter of interest, we examine the efficiency *loss* from *not* adjusting the parameter under study and exploring all other parameters. A parameter convex hull (see Section 2.2.2 for its definition) outlines the bandwidth efficiency achieved under a fixed value for the parameter under study while exploring all other parameters. There exist a set of parameter hulls, one for each value of the parameter under study. Since a DHT's overall convex hull always lies beneath all parameter hulls, the area between a parameter hull and the overall convex hull denotes the amount of lost efficiency due to setting the parameter to that fixed value. Therefore, if one had to set the parameter to one specific value, one should choose the value that corresponds to the parameter hull with the minimum area difference (A_{min}). Intuitively, A_{min} reflects the efficiency loss of not tuning a parameter that cannot be

	Tapestry			Chord			Kelips			Kademlia			OneHop		
	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val
1	t_{stab}	1.71	72s	t_{succ}	0.46	36s	t_{gossip}	0.79	72s	n_{tell}	1.00	4	t_{stab}	0.01	16s
2	n_{redun}	0.38	2	t_{finger}	0.12	288s	$n_{contact}$	0.05	8	k	0.76	4	n_{unit}	0.00	5
3	n_{repair}	0.10	3	b	0.07	2	t_{out}	0.03	720s	α	0.32	4	n_{slice}	0.00	3
4	b	0.05	4	n_{succ}	0.04	8	$r_{contact}$	0.01	2	t_{stab}	0.00	1152s			
5							r_{group}	0.00	2						

Figure 4-3: Rankings of the importance of re-adjusting parameters to achieve the best failure rate vs. bandwidth tradeoffs in different bandwidth regions. Each row corresponds to one parameter and rows are ranked from the most important parameter to the least important. For each parameter under study, a set of parameter convex hulls is calculated, one for each specific value of that parameter while exploring optimal settings for other parameters. The A_{min} column denotes the minimum area difference between any parameter hull and a DHT's overall convex hull over the bandwidth range 1-80 bytes/node/s. The *val* column shows the parameter value that produces the parameter hull with the minimum area difference A_{min}. A large A_{min} corresponds to a parameter with no one good "default" value; the best parameter setting changes in different bandwidth regions and fixing the parameter to any specific value results in reduced bandwidth efficiency. The parameters within a DHT are ordered according to decreasing A_{min}.

	Tapestry			Chord			Kelips			Kademlia			OneHop		
	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val
1	b	306	16	b	1271	32	t_{gossip}	517	18s	n_{tell}	5783	8	t_{stab}	1186	8s
2	t_{stab}	298	36s	t_{finger}	581	144s	$n_{contact}$	153	16	α	1139	16	n_{unit}	186	3
3	n_{redun}	177	3	t_{succ}	404	72s	$r_{contact}$	140	16	k	546	8	n_{slice}	0	3
4	n_{repair}	49	5	n_{succ}	186	8	t_{out}	133	360s	t_{stab}	46	1152s			
5							r_{group}	43	2						

Figure 4-4: Rankings of the importance of re-adjusting parameters to achieve the best lookup latency vs. bandwidth tradeoffs in different bandwidth regions. This table has the same format as table 4-3 but uses the average lookup latency (in ms) of successful lookups as the performance metric.

compensated by optimally adjusting other competitive ways of using bandwidth. Thus, A_{min} measures the *importance* of re-adjusting the parameter in different bandwidth regions to achieve a DHT's maximal bandwidth efficiency. A small minimum area suggests that there exists one best default value for the parameter under study. A large minimum area indicates it is important to re-adjust the parameter to optimize performance.

Tables 4-3 and 4-4 show the rankings of parameters according to their respective A_{min} in all DHTs. The tables also list the actual parameter value that corresponds to the parameter hull with the minimum area difference for each parameter. The first row corresponds to the parameter most in need of tuning in each DHT. For example, Table 4-3 shows that the most important parameter to re-adjust in Tapestry is t_{stab} in order to achieve the lowest failure rates at different bandwidth regions. From Table 4-3, we can see that fixing t_{stab} to be its best value of 72 seconds across the bandwidth range from 1 to 80 bytes/node/s results in the largest amount of efficiency loss (as indicated by the largest corresponding $A_{min} = 1.71$). In contrast, setting the base parameter b to its best value of 4 causes the least efficiency loss ($A_{min} = 0.05$) in terms of Tapestry's failure rate vs. bandwidth tradeoffs.

4.3 Understanding DHT design choices

The rankings of parameters shown in Table 4-3 and 4-4 reflect the relative bandwidth efficiencies of different design choices in a DHT. Since different DHTs consist of different sets of design choices, understanding the relative efficiencies of individual design choices help us explain why some DHTs are more efficient at handling churn than others in Figure 4-1 and 4-2. This section presents a number of insights on DHT designs based on the PVC parameter analysis of the design choices using Table 4-3 and 4-4.

4.3.1 When To Use Full State Routing Table

OneHop is the only DHT with no flexibility in the size of its per-node routing table. All nodes always aim to keep a full routing table containing all other nodes. Figures 4-1 and 4-2 show that OneHop has the most efficient convex hull for the most part. When bandwidth is greater than 7 bytes/node/s, OneHop's best achievable failure rate is close to 0% and its lookup latency is about 199ms which is higher than its static performance. In a static network, each lookup takes exactly one hop to reach its responsible node with latency equal to the average network RTT (178ms). In networks with churn, join/leave notifications take time to reach all nodes in the system, causing lookup timeouts or additional hops and hence higher lookup latency.

Although OneHop has no flexibility in its routing table size, it can trade off the freshness of the routing entries with extra bandwidth consumption by adjusting t_{stab}, the delay period during which join/leave events are aggregated before dissemination. According to Table 4-3 and 4-4, t_{stab} is the most important parameter to tune in order to achieve best failure rates and lookup latencies at different bandwidth regions. To optimally consume 8 bytes/node/s of bandwidth, one should set t_{stab} to 32s for lowest lookup latency and t_{stab} should be re-adjusted to 4s for best latency at 40 bytes/node/s. Smaller t_{stab} causes OneHop to consume more bandwidth and disseminate events more promptly so lookups are less likely to incur timeouts or additional hops due to delayed event propagation. However, the converse is not always true, i.e. OneHop's bandwidth consumption does not always decrease with bigger values of t_{stab}. Since OneHop expects full routing tables, nodes treat each lookup timeout or additional hop as *exceptions* and assume that the original join/leave event has been lost. The lookup originator re-generates and propagates a join/leave event to be disseminated for each timeout or additional hop encountered. Therefore, setting t_{stab} to 64s from $16s$ causes OneHop to consume more bandwidth but also results in much worse lookup latency due to timeouts.

Because t_{stab} can not be set to arbitrarily big values, OneHop has a minimum bandwidth consumption of 7 bytes/node/s, which is bigger than Chord's 2 bytes/node/s. Intuitively, OneHop's minimal bandwidth reflects the bandwidth necessary to propagate all events to all nodes to maintain complete routing state. Unfortunately, this minimal bandwidth consumption scales proportionally with the size of the network and the churn rate. We evaluate OneHop in a network of 3000 nodes[1] and show the resulting overall convex hulls in Figure 4-5.

Figure 4-5 shows that OneHop's minimum bandwidth consumption (the left-most point of the OneHop curve) is approximately 20 bytes/node/s for 3000-node networks. The threefold increase in the number of nodes triples the total number of join/leave events that must be delivered to every node in the network, causing OneHop to triple its minimum bandwidth consumption from 7 to 20 bytes/node/s. For comparison, we also include Chord in Figure 4-5. The per-node routing table size in Chord scales as $O(\log n)$ and hence the convex hull of the 3000-node Chord network is shifted from the one for the 1024-node network by only a small amount towards the upper right. Therefore, while a full routing table can produce better performance when bandwidth is plentiful, it also weakens a node's ability to limit its bandwidth consumption under varying network sizes and churn and therefore is primarily attractive in small or low-churn systems.

Another aspect of OneHop's performance is that slice and unit leaders use about 8 to 10 times more network bandwidth than the average. Chord, Ke-

[1]As we do not have King data for our 3000 node topology, we derive our 3000-node pair-wise latencies from the distance between two random points in a Euclidean square. The mean RTT is the same as that of our 1024-node network.

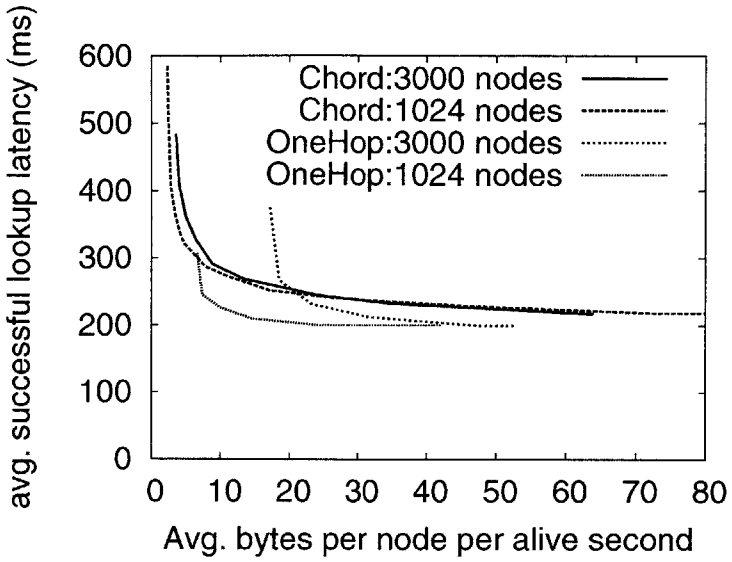


Figure 4-5: Overall convex hulls for Chord and OneHop in 1024- and 3000-node networks, under a churn intensive workload.

lips, Tapestry and Kademlia, on the other hand, have more uniform bandwidth consumption: the 95^{th}-percentile node uses no more than twice the average bandwidth. Therefore, if no node in the network can handle the bandwidth required of slice or unit leaders, one would prefer a symmetric protocol to OneHop.

Since OneHop doesn't allow significant tuning of the performance vs. cost tradeoffs, we do not include this protocol in the rest of our analysis.

4.3.2 Separation of Lookup Correctness from Performance

Figure 4-1 shows that Chord can be tuned to have a lower failure rate than other protocols using only a small amount of bandwidth (< 8 bytes/node/s). The following PVC parameter analysis explains why.

Table 4-3 shows that the Chord parameter most in need of tuning is t_{succ}. Chord separates a node's routing table entries into those that ensure lookup correctness (successors) and those that provide fast routing (fingers). t_{succ} governs how often a node pings its successor and thus determines how quickly a node can realize that its successor is dead and should be replaced with the next live node from its successor list. The correctness of a Chord lookup depends only on nodes' successor entries and not fingers, therefore it is enough to only stabilize successors more frequently for lower failure rates.

The other protocols (except OneHop which uses the same successor stabilization mechanism as Chord) do not separate lookup correctness from performance. Therefore, if a low failure rate is desired, the entire routing table must be checked

frequently. Table 4-3 shows that the most important Tapestry parameter is t_{stab} which determines how fast a node checks the liveness of its *entire* routing table. As the number of successors tends to be only a small fraction of the entire routing table, Chord's successor stabilization leads to more attractive failure rate vs. bandwidth tradeoffs than Tapestry.

4.3.3 Coping with Non-transitive Networks

DHT protocols typically have explicit provisions for dealing with node failure. These provisions usually handle network partitions in a reasonable way: the nodes in each partition agree with each other that they are alive, and agree that nodes in the other partitions are dead. Anomalous network failures that are not partitions are harder to handle, since they cause nodes to disagree on which nodes are alive. For example, if node A can reach B, and B can reach C, but A cannot reach C, then they will probably disagree on how to divide the key ID space among the nodes. A network that behaves in this manner is said to be non-transitive. Non-transitivity is a common source of lookup failures of the DHT deployments on the Planetlab testbed [23, 67].

In order to measure the effects of this kind of network failure on DHTs, we created a topology exhibiting non-transitivity by discarding all packets between 5% of the node pairs in our standard 1024-node topology, in a manner consistent with the observed 4% of broken pairs [25] on PlanetLab [5]. The existence of PlanetLab nodes that can communicate on only one of the commercial Internet and Internet-2, combined with nodes that can communicate on both networks, produces non-transitive connectivity between nodes. We ran both Chord and Tapestry churn intensive experiments using this topology, and measured the resulting failure rates of the protocols. Both protocols employ recursive lookup, and thus nodes always communicate with a relatively stable set of neighbors, eliminating the problem that occurs in iterative routing (*e.g.*, Kelips, Kademlia and OneHop) in which a node hears about a next hop from another node, but cannot communicate with that next hop.

We disable the standard join algorithms for both Chord and Tapestry in these tests, and replace them with an *oracle* algorithm that immediately and correctly initializes the state of all nodes in the network whenever a new node joins. Without this modification, nodes often fail to join at all in a non-transitive network. Our goal is to start by investigating the effect of non-transitivity on lookups, leaving the effect on join for future work. This modification changes the bandwidth consumption of the protocols, so these results are not directly comparable to Figure 4-1.

Figure 4-6 shows the effect of non-transitivity on the failure rates of Tapestry and Chord. Chord's failure rate increases more than Tapestry's with non-transitivity; we can use PVC parameter analysis to shed light on how Tapestry handles non-

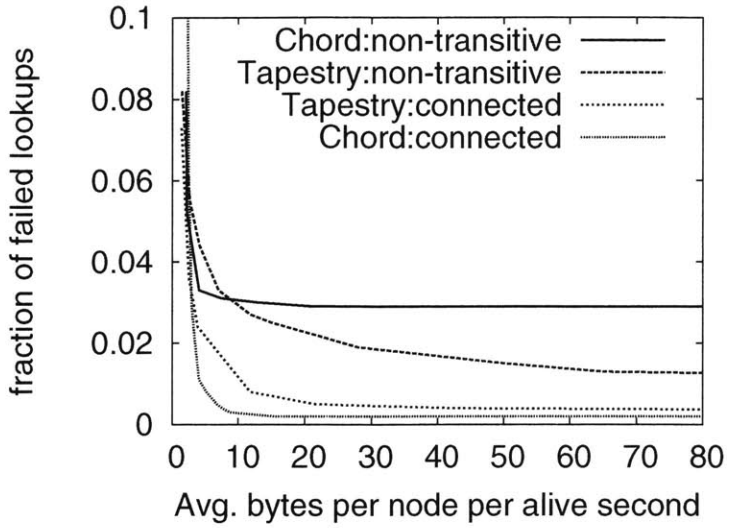


Figure 4-6: Overall convex hulls for lookup failure rates for Chord and Tapestry under connected and non-transitive networks, under the churn intensive workload.

transitivity. As Table 4-3 shows, in a fully-connected network, base (b) was the *least* important parameter for Tapestry, in terms of failure rate vs. bandwidth tradeoff. However, Table 4.1 shows that in the non-transitive network, base (b) becomes a much more important parameter, ranking second behind stabilization interval (which is still necessary to cope with churn). For Chord, however, base (b) remains an unimportant parameter.

We can explain this phenomenon by examining the way in which the two protocols terminate lookups according the structure of their routing tables. The Chord lookup algorithm assumes that the ring structure of the network is correct. If a Chord node n_1 cannot talk to its correct successor n_2 but can talk to the next node n_3, then n_1 may return n_3 for lookups that really should have found n_2. This error can arise if network connectivity is broken between even a single node pair.

Tapestry's surrogate routing, on the other hand, allows for a degree of leniency during the last few hops of routing. Strict progress according to the prefix-matching distance metric is not well defined once the lookup reaches a node with the largest matching prefix in the network. This means that even if the most direct path to the owner of a key is broken due to non-transitivity, surrogate routing may find another, more circuitous, path to the owner. This option is not available in Chord's strict linked-list structure, which only allows keys to be approached from one direction around the ring in ID space. Tapestry does suffer some failures, however. If a lookup reaches a node that knows of no other nodes matching a prefix of the same size with the key as itself, it will declare itself the owner, despite the existence of an unreachable owner somewhere else in the network. A bigger

	Tapestry			Chord		
	param	A_{min}	val	param	A_{min}	val
1	t_{stab}	0.43	36s	t_{succ}	0.20	18s
2	b	0.11	8	t_{finger}	0.13	1152s
3	n_{redun}	0.07	6	n_{succ}	0.07	32
4	n_{repair}	0.04	1	b	0.00	2

Table 4.1: The ranking of parameters in need of tuning in Tapestry and Chord according to the minimum area (A_{min}) between the parameter hulls and the overall convex hull. This table has the same format as that in Table 4-3 except the experiments are done in a non-transitive network.

base results in more entries matching the key with the same largest matching prefix and hence gives more opportunity to surrogate routing to route around broken network connectivity.

In summary, while existing DHT designs are not specifically designed to cope with non-transitivity, some protocols are better at handling it than others. Future techniques to circumvent broken connectivity may be adapted from existing algorithms.

4.3.4 Bigger Routing Table for Lower Latency

In Table 4-4, both Tapestry and Chord have base (b) as the parameter that is most in need of tuning for best lookup latency. Base controls the number of routing entries each node keeps and bigger bases lead to bigger routing tables with $(b-1)\log_b(n)$ entries. Figure 4-7 shows Chord's overall convex hull as well as its parameter hulls for different base values. At the left side of the graph, where the bandwidth consumption is small, the parameter hull for $b = 2$ lies on the overall convex hull which means smaller bases should be used to reduce stabilization traffic at the expense of higher lookup latency. When more bandwidth can be consumed, larger bases lower the latency by decreasing the lookup hop-count.

The highest ranking of the base parameter (b) suggests that expanding a node's routing table is more efficient than other alternatives at using additional bandwidth. For example, one competitive use of extra bandwidth is to check for the liveness of routing entries more frequently as doing so would decrease the likelihood of lookup timeouts. However, when routing entries are "fresh enough", spending extra bandwidth to further reduce the already very low lookup timeout probability has little impact on the overall latency. Instead, when the routing table is fairly fresh, a node should seek to reduce lookup latency by using additional bandwidth to expand its routing table for fewer lookup hops. This explains why base (b) ranks above stabilization intervals (t_{stab} and t_{finger}) as the parameter to

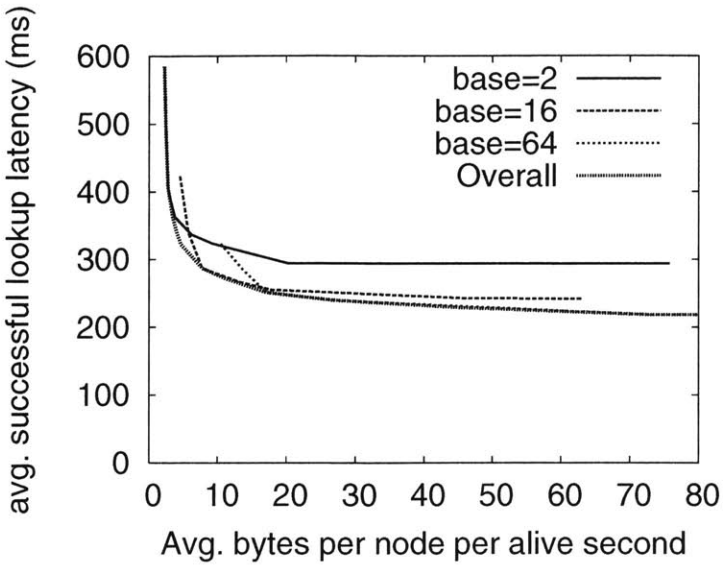


Figure 4-7: Chord, under the churn intensive workload. Each line traces the convex hull of all experiments with a fixed base b value while varying all other parameters.

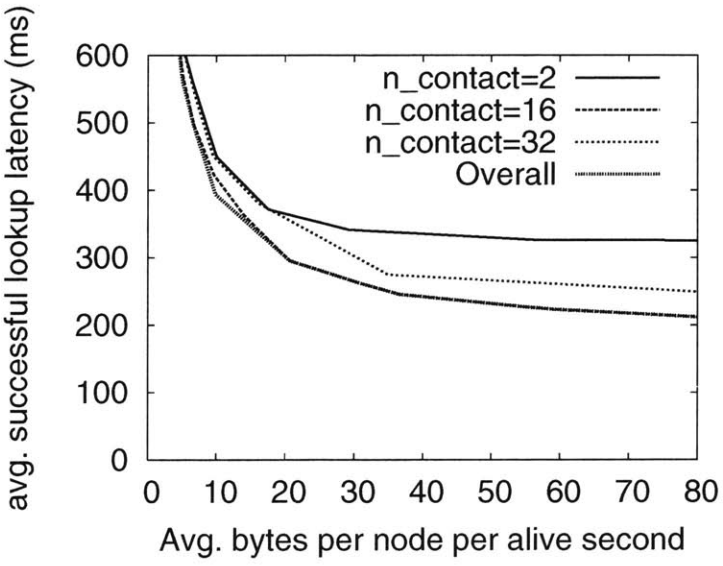


Figure 4-8: Kelips, under the churn intensive workload. Each line traces the convex hull of all experiments with a fixed $n_{contact}$ value while varying all other parameters.

tune at different bandwidth regions.

In contrast to Chord and Tapestry, the Kelips parameter ($n_{contact}$) that controls its routing table size does not appear to require much tuning. Instead, the gossip interval (t_{gossip}) is the most important parameter (see Table 4-4). Figure 4-8 shows the parameter convex hulls for different values of $n_{contact}$ in Kelips. The parameter hull for $n_{contact} = 16$ approaches the overall convex hull over the entire bandwidth range, while a smaller or bigger $n_{contact}$ results in worse performance/cost tradeoffs. In Chord/Tapestry, base (b) governs the routing table size exactly as the stabilization process actively explores the network to find all $(b-1)\log_b(n)$ entries. In contrast, $n_{contact}$ in Kelips determines only the amount of *allowed* state, the actual amount of state acquired by each node is determined by how fast nodes gossip (t_{gossip}) which is the most important parameter in Kelips. It is always beneficial to maximize the amount of *allowed* state so a node can keep any piece of routing information it has learnt and use it later during lookups. Hence, Kelips should allow many contacts for each foreign group as a particular contact is no more or less important than others. This explains why the Kelips parameter hull for $n_{contact} = 16$ is more efficient than that of $n_{contact} = 2$.

A bigger $n_{contact}$ is not always better in Kelips. In Figure 4-8, the efficiency of the parameter hull for $n_{contact} = 32$ is worse than that of $n_{contact} = 16$. We find that changing $n_{contact}$ from 16 to 32 results in more lookup timeouts and hence higher latency for Kelips. Unlike Chord and Tapestry which directly communicate with *all* neighbors, Kelips acquire new routing entries indirectly from a third node through gossips. Ensuring the freshness of routing entries is harder in the face of gossip as a newly acquired entry does not always reflect more recent information than existing information about the same neighbor. As Kelips does not explicitly account for the time interval a routing entry remains un-contacted before it has been gossiped to other nodes, many entries point to neighbors that have not been heard for more than t_{out} seconds even though Kelips expires routing entries after t_{out} seconds of inactivity. Therefore t_{out} is in-effective at bounding the staleness of routing entries. This is in contrast to Chord or Tapestry whose parameters t_{finger} (or t_{stab}) bounds the maximum age of all routing entries as a node always seeks to directly communicate with each of its neighbors every t_{finger} (or t_{stab}) seconds. Therefore, a bigger $n_{contacts}$ may result in worse efficiency in Kelips because the more contacts are kept at each node, the less fresh the routing entries tend to be.

PVC analysis has shown that expanding routing table is necessary to efficiently use extra bandwidth, given the routing entries can be kept relatively fresh. This observation also explains why in Figure 4-2 OneHop's overall convex hull is more efficient than all other protocols at large bandwidth. This is because OneHop keeps a complete routing table which is the more efficient at high bandwidth than $O(\log n)$ routing table kept by Chord, Tapestry and Kademlia.

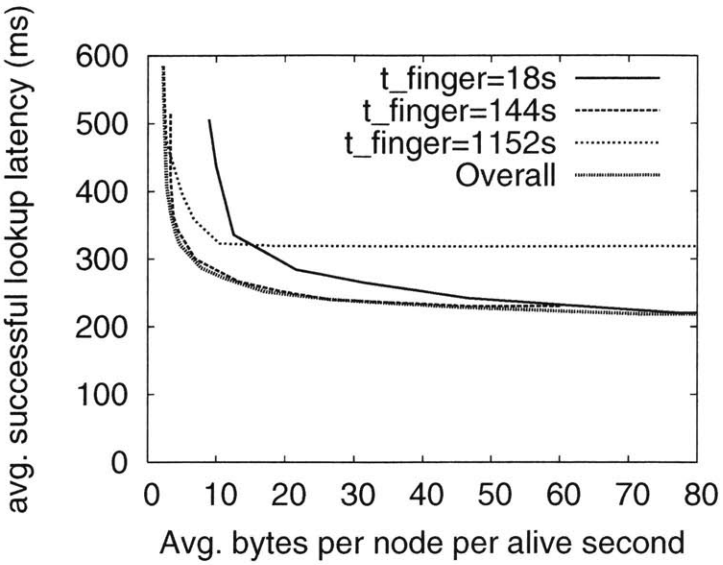


Figure 4-9: Chord, under the churn intensive workload. Each line traces the convex hull of all experiments with a fixed t_{finger} value while varying all other parameters.

4.3.5 Bounding Routing Entries Staleness

Section 4.3.4 suggests that a node should expand its routing table when the existing routing entries are already fresh. What is a good freshness threshold for routing tables? In Chord and Tapestry, routing entries' staleness is bounded by the stabilization interval (t_{finger} and t_{stab}). Table 4-4 shows that the best default value of stabilization interval for Chord is $t_{finger} = 144$s. Figure 4-9 shows the parameter hulls for three stabilization values ($t_{finger} = 18, 144, 1152$ seconds) as well as Chord's overall convex hull. The best value for t_{finger} (144s) corresponds to a parameter hull that approximates the entire overall convex hull. Since parameter hulls are computed by exploring all other parameters including base (b), a less attractive parameter hull indicates the efficiency loss by setting t_{finger} to a wrong value. Making routing entries fresher than necessary ($t_{finger} = 18$s) results in a less efficient parameter hull as the extra bandwidth is wasted on checking the already sufficiently up-to-date routing entries as opposed to expanding a node's routing table. Allowing routing entries to become too stale ($t_{finger} = 1152$ s) also dramatically decreases the efficiency of the convex hull as stale routing entries lead to too many timeouts which can not be compensated by routing via fewer hops with a larger routing table.

The lesson here is that it is important to provide some guarantees for routing entry freshness and there seems to be one best freshness threshold under a given churn rate. In our experiments where nodes' mean lifetime is 1 hour, the optimal freshness threshold for Chord is about 144s ($t_{finger} = 144$ s). Since node

lifetimes are exponentially distributed and a node pings each routing entry every $t_{finger} = 144$ seconds, the probability that any entry points to a departed neighbor is $\Pr lifetime < 144 = \exp^{-\frac{144}{3600}} = 0.96$, i.e. lookup times occur as infrequently as 4% at each lookup hop. Further reducing the already per-hop timeout probability is not as cost efficient as expanding a node's routing table for fewer lookup hops. On the other hand, if timeout probability is significantly higher, as in the case with $t_{finger} = 1152$s, lookup latency increases as there are too many timeouts. A Chord node directly communicates with all its fingers and hence the pinging interval (t_{finger}) determines its routing table freshness. If a node acquires new routing entries indirectly from its existing neighbors such as the case in Kelips and Kademlia, it should also take into account the amount of time a routing entry has aged in other nodes' routing tables in judging its freshness.

4.3.6 Parallel Lookup Is More Efficient than Stabilization

Kademlia has the choice of using bandwidth for either stabilization or parallel lookups. Both approaches reduce the effect of timeouts: stabilization by pro-actively eliminating stale routing table entries, and parallel lookups by overlapping activity on some paths with timeouts on others.

Table 4-4 shows that not only Kademlia's stabilization interval does not need to be tuned, but that its best setting is always the maximum interval (1152s). This implies that stabilization is an inefficient way of using bandwidth to improve latency. In contrast, Table 4-4 shows that n_{tell} and α, which control the degree of lookup parallelism, are the important parameters that determine Kademlia's overall latency vs. bandwidth tradeoffs. Larger values of α keep more lookups in flight, which decreases the likelihood that all progress is blocked by timeouts. Larger values of n_{tell} cause each lookup step to return more potential next hops and thus cause more opportunities for future lookup parallelism. It is no surprise that parallelizing lookups improves lookup performance, however, it is rather counter-intuitive that instead of being wasteful, parallelizing lookups turns out to be a more *efficient* use of bandwidth than stabilization. The reason for parallel lookup's efficiency is twofold. First, rather than pro-actively checking the freshness of each routing entry as in stabilization, parallelizing lookups deal with the staleness of routing entries only when the entries are being used in lookups. Second, in addition to learning about the liveness of existing routing entries as in stabilization, Kademlia nodes also learn new entries from redundant parallel lookup traffic to help expand their routing tables.

4.3.7 Learning from Lookups Can Replace Stabilization

Kademlia relies on lookup traffic to learn about new neighbors: a node learns up to n_{tell} new neighbors from each lookup hop. This turns out to be a more

	Kademlia (no learn)		
	param	A_{min}	val
1	t_{stab}	4.97	576s
2	α	0.84	4
3	n_{tell}	0.16	4
4	k	0.00	2

Table 4.2: The importance rankings of Kademlia's parameters that should be tuned to achieve the best failure rate vs. cost tradeoff, with learning disabled.

	Tapestry			Chord			Kelips			Kademlia			OneHop		
	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val	param	A_{min}	val
1	t_{stab}	241	36s	b	177	32	t_{gossip}	2328	36s	α	1281	4	t_{stab}	82	4s
2	b	170	32	t_{finger}	99	72s	$n_{contact}$	73	32	k	695	4	n_{unit}	0	5
3	n_{redun}	22	4	t_{succ}	73	36s	t_{out}	52	360s	n_{tell}	666	4	n_{slice}	0	5
4	n_{repair}	11	5							t_{stab}	0	1152s			
5															

Table 4.3: The relative importance of tuning each parameter to achieve the best lookup latency vs. bandwidth tradeoffs in different bandwidth regions. This table has the same format as table 4-4 but are obtained from experiments with a lookup intensive workload.

efficient way to acquire new routing entries than explicit exploration. As shown in Table 4-3, n_{tell} is the parameter that determines Kademlia's best failure rate vs. bandwidth tradeoffs. In contrast, stabilization (t_{stab}) does not affect Kademlia's bandwidth efficiency much at all and hence is the lowest ranked parameter in Table 4-3 and is best set to the largest value (1152s). However, with learning disabled, Table 4.2 shows that t_{stab} becomes the most important parameter whose value should be adjusted to achieve the best failure rate vs. bandwidth tradeoffs: a faster stabilization rate is required for a low lookup failure. This shows that learning from lookups can replace explicit exploration as the most bandwidth-efficient way of acquiring new entries.

4.3.8 Effect of a Lookup-intensive Workload

The lookup intensive workload involves each node issuing a lookup request every 9 seconds, almost 67 times the rate of the churn intensive workload used in the previous sections. As a result, the lookup traffic dominates the total bandwidth consumption. Figure 4-10 shows the overall latency convex hulls of all protocols under the lookup intensive workload.

Compared with Figure 4-2, Chord and Tapestry's convex hulls in Figure 4-10 are relatively flat. Table 4.3 shows the relative importance of tuning each

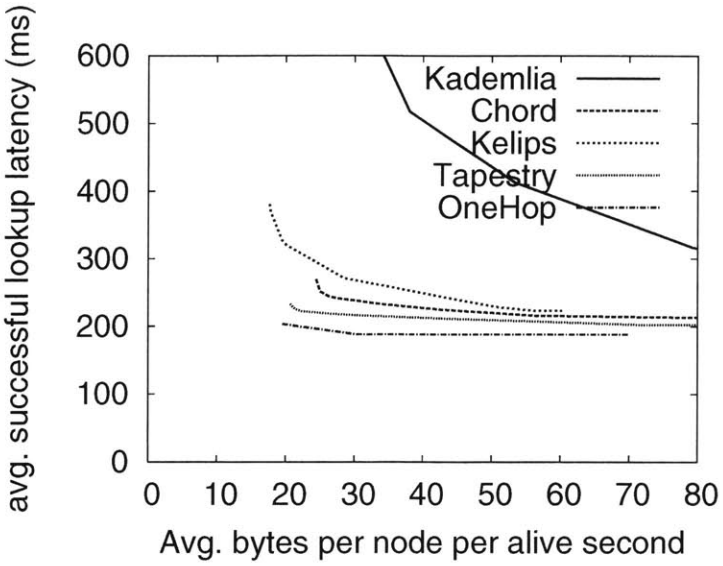


Figure 4-10: Overall convex hulls for the average latency of successful lookups for all DHTs, under the lookup intensive workload.

parameter to achieve the best lookup latency vs. bandwidth tradeoffs under the lookup intensive workload. Compared to the parameter rankings in a churn intensive workload (Table 4-4), there is a decrease in the importance of tuning Tapestry's base parameter (b). Furthermore, a large base (32) corresponds to the best parameter setting for both Chord and Tapestry. In the lookup intensive workload, each node issues and forwards much more lookup messages during its lifetime and hence the amount of churn is relatively low. Thus, it is more efficient to keep a larger routing table for fewer lookup hops when the amount of stabilization traffic is low compared to the amount of lookup traffic. Furthermore, fewer lookup hops translate into a large decrease in forwarded lookup traffic, given the large number of lookups.

For Kademlia, α becomes the most important parameter to tune. Furthermore, compared with Table 4-4, there is a significant decrease in the best α value from 16 to 4. $\alpha = 4$ obtains the best tradeoff in the lookup intensive workload, as opposed to the larger α of 16 which is the best parameter setting for the churn intensive workload. Since Kademlia's stabilization process does not actively check the liveness of each routing table entry, stabilization is ineffective at bounding routing entries' staleness to ensure few lookup timeouts. Therefore, in order to avoid excess timeouts, some amount of lookup parallelism (i.e., $\alpha > 1$) is still needed. However, as lookup traffic dominates in the lookup intensive workload, lookup parallelism is quite expensive as it multiplies the already large amounts of lookup traffic. This partially explains why, in Figure 4-10, the overall convex hull of Kademlia is significantly worst than that of other protocols in the lookup

intensive workload.

4.4 Summary of Insights

Table 4.4 summarizes the insights from the preceding sections. The best use for extra available bandwidth is for a node to expand its routing table. It is important to bound the staleness of routing entries and there seems to be a best freshness threshold under a given churn rate. Compared with periodic pinging to check the liveness of existing routing entries and active exploration of new entries, parallelizing lookups and learning opportunistically through lookup traffic are more efficient at using bandwidth to reduce lookup timeouts and acquire new entries. These results taken together demonstrate the value of PVC as a tool to design and evaluate DHT protocols.

Section	Insights
4.1	Minimizing lookup latency requires complex workload-dependent parameter tuning.
4.3.1	The ability of a protocol to control its bandwidth usage has a direct impact on its scalability and performance under different network sizes.
4.3.2	DHTs that distinguish between state used for the correctness of lookups and state used for lookup performance can more efficiently achieve low lookup failure rates under churn.
4.3.3	The strictness of a DHT protocol's routing distance metric, while useful for ensuring progress during lookups, limits the number of possible paths, causing poor performance under pathological network conditions such as non-transitive connectivity.
4.3.4,4.3.5	Increasing routing table size to reduce the number of expected lookup hops is a more cost-efficient way to consume additional bandwidth than pinging existing entries more often. However, it is crucial to bound the staleness of routing entries. In a fixed churn rate, there seems to be some "good enough" freshness threshold for routing entries.
4.3.6	Issuing copies of a lookup along many paths in parallel is more effective at reducing lookup latency due to timeouts than more frequent pinging under a churn intensive workload.
4.3.7	Learning about new nodes during the lookup process can essentially eliminate the need for actively searching for new entries.
4.3.8	Increasing the rate of lookups in the workload, relative to the rate of churn, favors all design choices that reduce the overall lookup traffic. For example, one should use extra state to reduce lookup hops (and hence forwarded lookup traffic). Less lookup parallelism is also preferred as it generates less redundant lookup traffic.

Table 4.4: Insights obtained by using PVC to evaluate the design choices embedded in five existing protocols: Chord, Kademlia, Kelips, OneHop and Tapestry.

Chapter 5

Accordion Design

With existing designs, a DHT user is forced into choosing a fixed routing table size. A $O(\log n)$-state DHT like Chord consumes a small amount of bandwidth, but has a relatively high lookup latency, thus is suitable for large networks with frequent churn. A $O(n)$-state DHT like OneHop has best lookup latency, but it incurs high communication costs which might exceed the underlying network capacity when the system grows too large or suffers from high churn. Existing DHTs are designed to work well under a certain deployment scenario, which results in either non-optimal lookup performance or the risk of overloading the network with overhead traffic when the current operating environment does not match design assumptions.

Instead of choosing a fixed design that work well under certain assumptions of the operating environment, robust systems seek to adapt itself to have good performance across a wide range of different operating conditions and fail gracefully in an unexpected environment [26]. To have good performance, a DHT node should seek to maintain a large routing table. To be robust against changing deployment environments, a DHT should be able to bound its overhead traffic. The key to designing a robust DHT that also has good performance is to be able to adjust a node's routing table size on the fly. Such a DHT would maintain a large routing table to perform one hop lookup when the bandwidth is plentiful relative to the network size and churn, and shrink its routing table to perform multi-hop lookups when the bandwidth is limited. In this chapter, we present Accordion, a DHT lookup protocol that automatically adapts its routing table size according to the current deployment environment. In order to achieve best lookup performance, the design of Accordion draws from many lessons learnt in Chapter 4.

5.1 Challenges

A robust DHT has bounded bandwidth overhead, in particular, a DHT's routing table maintenance traffic must fit within the nodes' access link capacities. Most existing designs do not try to live within this physical constraint. Instead, the amount of maintenance traffic they consume is determined as a side effect of the total number of nodes, the rate of churn and workload. To ensure robustness, the first design decision is for Accordion to take into account a user-specified bandwidth budget and bound its bandwidth consumption accordingly. A DHT node that controls its own bandwidth usage can choose what packets *not* to send as opposed to the network dropping indiscriminately at times of overload. Thus, such a DHT design is robust against changing operating environments and can degrade its performance more gracefully when the network grows rapidly or the churn surges.

The presence of a bandwidth budget poses several unique challenges. First, an Accordion node needs to bound its bandwidth use according to this user-specified budget. Ideally, it should be able to limit all of its outgoing traffic. However, since a node must initiate and forward application lookup traffic, the amount of which is not under a DHT's control, it really has control only over the portion of leftover budget after sending out all the required lookups. Therefore, to operate within budget, an Accordion node needs to adjust its routing table maintenance traffic to fill up the leftover budget.

Second, in order to optimize lookup latency while observing its budgetary constraint, an Accordion node needs to adjust its routing table size dynamically. The right choice of routing table size depends on both the budget and the amount of churn. Intuitively, when the bandwidth budget is plentiful relative to the level of churn in the system, Accordion should use a large or complete routing table like OneHop. When the converse is true, Accordion should tune itself to use a small routing table like that of Chord, consuming little bandwidth. Furthermore, the fact that the bandwidth is a budgeted resource forces Accordion to consciously use the most bandwidth efficient churn handling technique to maintain its routing table. In designing Accordion, we apply many lessons on efficient design choices from Chapter 4.

Using a variable routing table size brings up two technical issues. First, we need a routing structure that allows nodes to expand and contract its routing table along a continuum and yet still guarantees a small number of lookup hops for all sizes. Second, an Accordion node must figure out which table size is the best one to use. It could calculate the best table size, but that would require it to explicitly measure the current churn rate and workload. Instead, Accordion's routing table maintenance process has two sub-processes: routing state acquisition and eviction. The state acquisition process learns about new neighbors; the bigger the budget, the faster a node learns, resulting in a bigger table size. This reflects the insight

from the previous chapter that a node should always seek to expand its routing table to consume extra bandwidth. The state eviction process deletes routing table entries that are likely to cause lookup timeouts; the higher the churn, the faster a node evicts state. This corresponds to the lesson from the previous chapter that a node should bound the staleness of its routing entries and the best freshness threshold depends on the churn rate. Accordion's routing table size is the result of the equilibrium between the two processes. If the eviction process evicts entries at the right speed, Accordion will reach a good equilibrium with the optimal table size.

5.2 Overview

Accordion uses consistent hashing [41] in a circular identifier space to map a key to its responsible node, same as that in Chord and OneHop. A key is mapped to its successor node whose identifier immediately follows the key on the clockwise identifier ring. Accordion routes a lookup to the key's predecessor node who returns the identity of the key's successor to the lookup originator.

In order to ensure correct lookup termination, each node needs to keep $O(\log n)$ successor entries whose identifiers immediately follow its own. An Accordion node uses the same technique as that in Chord to keep its successor list up to date under churn. Each node periodically obtains the successor list from its current successor and merges this latest information with its existing successor list. In order to provide speedy lookups, Accordion also needs to maintain a routing table with information about many other neighbors. Chapter 4 shows that the bandwidth overhead needed to maintain the routing state needed for correct lookup termination can be much smaller than the bandwidth required to keep the rest of the table up to date. Therefore, we focus the design discussion on how to efficiently maintain those majority of routing entries needed for fast lookups.

New nodes join an existing Accordion network via a set of well-known nodes. The new node looks up its own node identifier to learn the identities of its current set of successors. After an Accordion node has successfully joined the network, it starts to obtain routing entries needed for fast lookups and keep those entries up to date, a process we referred to as the routing table maintenance. The essence of the Accordion protocol is the design of a bandwidth efficient routing table maintenance process that also observes a user-specified bandwidth budget.

We address the following questions in designing Accordion's routing table maintenance process:

1. How do nodes choose neighbors for inclusion in the routing table in order to guarantee at most $O(\log n)$ lookups with a small routing table?

2. How do nodes choose between active exploration and opportunistic learning (perhaps using parallel lookups) to learn about new neighbors in the most efficient way?

3. How aggressively should a node evict stale entries to ensure that the equilibrium reached by the state acquisition and eviction processes will optimize latency? How should nodes efficiently gather the information required to make good eviction decisions?

5.3 Routing State Distribution

In order to route lookups quickly in ID space, Accordion needs to populate its routing table from different regions of the identifier space according to a *routing structure*. The ideal routing structure is both scalable and flexible. With a scalable routing structure, even a very small routing table can route lookups in a few hops. With a flexible routing structure, a node can include any other node in its routing table and use it to route lookups. Furthermore, a node is able to expand and contract its routing table along a continuum to tradeoff lookup hops for routing table sizes. However, as currently defined, most DHT routing structures are not flexible and require a node to only include neighbors from specific regions of ID space. For example, a Tapestry node with a 160-bit identifier of base b maintains a routing table with $\frac{160}{\log_2 b}$ levels, each of which contains $b-1$ entries. The routing table size is fixed at $(b-1)\log_b n$ in a network of n nodes and a node does not augment its table with new information if the levels are already full. Furthermore, The parameter base (b) controls the table size, but it can only take values that are powers of 2, making it difficult to adjust the table size smoothly.

Let the ID distance between two nodes be the clockwise ID interval between them. We can relax a rigid routing table structure by specifying only the desired distribution of ID distances between a node and its neighbors. Viewing routing structure as a probabilistic distribution gives a node the flexibility to include any node (from the probability distribution) in its routing table and to adjust its routing table size smoothly. For example, to increase the routing table size by one, a node simply looks up a neighbor near a random ID sampled from the distribution. To reduce the table size by one, a node just evicts any existing routing entry.

Accordion uses a $\frac{1}{x}$ distribution to choose its neighbors: the probability of a node selecting a neighbor with distance x in the identifier space from itself is proportional to $\frac{1}{x}$. This distribution causes a node to prefer neighbors that are closer to itself in ID space, ensuring that as a lookup gets closer to the target key there is always likely to be a helpful routing table entry. This $\frac{1}{x}$ distribution is the same as the "small-world" distribution originally proposed by Kleinberg [43] and is also used by other DHTs such as Symphony [57] and Mercury [7].

The small world distribution results in an average lookup hop of $O(\frac{\log n \cdot \log\log n}{\log s})$ if each node's routing table contains s entries. We prove this result using techniques shown by Kleinberg [43]. For simplicity of analysis, we assume that total ID ring length is n for a network of n nodes and all nodes are at equal ID distance ($= 1$) apart from each other.

Let y be the node current node to forward a lookup with key k. We say that a lookup is in phase j if node y is at ID distance $d(y,k)$ away from the key k, where $z^j < d(y,k) < z^{j+1}$ for some constant z. Therefore, a lookup requires $O(\frac{\log n}{\log z})$ phases to reach its predecessor node at 1 unit of ID distance away from the key. What is the expected number of hops required for a lookup to successfully transition from any phase j to the next phase $j-1$?

Let U denote the set of nodes whose ID distance to the key is at most z^j. There are z^j nodes in U, each at distance at most z^{j+1} away from node y. If node y picks any node from U as its neighbor, the lookup will transition into phase $j-1$. Because y chooses neighbors from a small world distribution, a node at distance d away from y has a probability $\frac{1}{d} \cdot \frac{1}{\sum_{i=1}^{n} \frac{1}{i}} > \frac{1}{d} \cdot \frac{1}{\ln n}$ of being chosen as y's routing entry. Thus, the probability of at least one node in U being chosen as node y's neighbor is approximately $z^j \cdot \frac{1}{z^{j+1} \cdot \ln n} = \frac{1}{z \cdot \ln n}$. Since y needs to choose s routing entries, the overall probability of y having at least one neighbor from U is $\frac{s}{z \cdot \ln n}$. Therefore, the average number of hops required to transition from phase j to $j-1$ is $O(\frac{z \cdot \ln n}{s})$.

Since a lookup requires $O(\frac{\log n}{\log z})$ phases to complete, the total number of lookup hops is $O(\frac{z \cdot \ln n}{s} \cdot \frac{\log n}{\log z})$. Setting $z = \frac{s}{\log n}$, we obtain the total number of lookup hops as $O(\frac{\log n}{\log s - \log\log n})$ which can be approximated by $O(\frac{\log n \log\log n}{\log s})$ if $\log\log n > 1$ and $s > \log n$.

The small world distribution provides Accordion with a routing structure that is both scalable and flexible. It is scalable because even when each node has a small routing table, lookups can be routed in $O(\log n \cdot \log\log n)$ hops. It is flexible since a node can include any node from the distribution to expand its routing table to trade off for fewer lookup hops.

Not all probability distributions offer scalable lookups as the small world distribution. Intuitively, with a small world distribution, nodes can continue to make good forwarding progress in ID space as a lookup gets closer to its key because nodes preferentially knows more routing entries close to itself than far away in ID space. In contrast, with a uniform random distribution, a lookup approaches its key in ID space quickly in the beginning, but has to rely on successor entries to approach the key by only 1 unit of ID distance at each hop. Specifically, lookups require an average of $O(\sqrt{\frac{n}{s}})$ hops in a network of n nodes if each node's routing table is of size s.

We derive the average number of lookup hops with uniform random routing tables. Each node has a successor entry at 1 unit of ID distance away and s

random neighbors. We calculate how many times nodes use their successor entries to forward a lookup. The number of hops taken with successor entries serves as a lower bound to the total number of lookup hops. Let y be the current forwarding node at distance $d(y,k)$ away from the lookup key k and y' be the next hop node. The probability of y' at distance d away from y is: $\Pr(d(y,y')=d) = \frac{s}{n}(1-\frac{d}{n})^{s-1} < \frac{s}{n}$. The average distance between y and y' is less than $\sum_{d=1}^{d(y,k)} d \cdot \frac{s}{n} < \frac{s}{n} \cdot \frac{d(y,k)^2}{2}$. When $d(y,y') = \frac{s \cdot d(y,k)^2}{2n} < 1$, y has to use its successor to make forwarding progress. Therefore, a total of $\sqrt{\frac{2n}{s}}$ successor hops are used to forward lookups when the current node's distance to the key is less than $\sqrt{\frac{2n}{s}}$. Thus, we have shown that lookups using uniform random routing entries result in $O(\sqrt{\frac{n}{s}})$ hops.

5.4 Routing State Acquisition

After an Accordion node has joined an existing network, the routing state acquisition process continuously collects new entries to replenish its routing table under churn. As Chapter 4 has shown, a node should expand its routing table to consume extra bandwidth more efficiently. The bigger the bandwidth budget, the faster a node should acquire new state to maintain a larger routing table. A straightforward approach to learning new neighbors is active exploration, i.e. nodes explicitly lookup neighbors using IDs sampled from a small world distribution. However, Chapter 4 has also revealed that a more bandwidth-efficient approach is to learn about new neighbors, and the liveness of existing neighbors as a side-effect of ordinary lookup traffic.

Learning from lookup traffic does not necessarily yield new neighbors with the desired small world distribution in ID space. For example, if the DHT uses *iterative routing* [77] during lookups, the original querying node would talk directly to each hop of the lookup. Assuming the keys being looked up are uniformly distributed, the querying node would communicate with nodes in a uniform distribution rather than a small world distribution, resulting in $O(\sqrt{\frac{n}{s}})$ lookup hops. Nodes can filter out neighbors to force a small world distribution, but this leads to a smaller routing table and is not ideal.

With *recursive routing*, on the other hand, intermediate hops forward a lookup message directly to its next hop. Since most lookup hops are relatively short, a node is likely to contact next hop neighbors with closer-by node IDs. In recursive routing, when node y forwards a lookup to y', y' immediately sends back an acknowledgment message to y. Accordion uses recursive routing. Furthermore, an Accordion node piggy-backs additional routing entries whose IDs immediately follow its own node ID in the acknowledgment message to help the previous hop learn new routing entries. If lookup keys are uniformly distributed, then a node is equally likely to use each of its routing entries. Assuming a node's routing entries

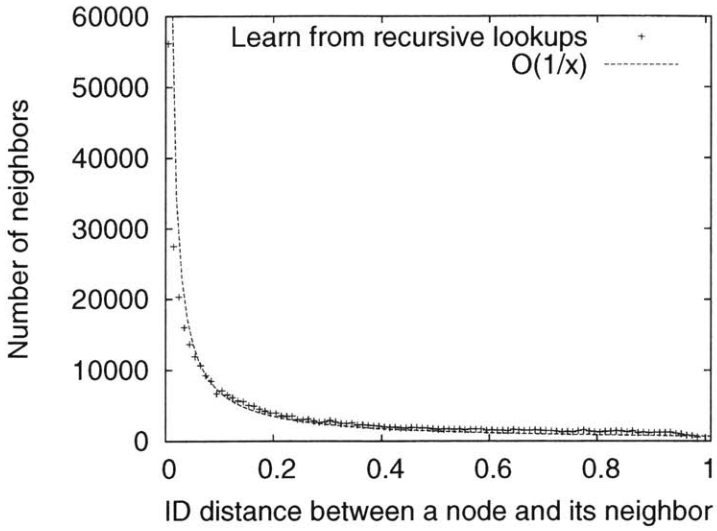


Figure 5-1: The density of ID distances between a node and its neighbors. The simulation experiment is for a 3000-node static network. Each node starts out with 10 random neighbors and 1 successor entry. Nodes learn new routing entries from 30,000 recursive lookups for random keys issued by random nodes in the network. The figure shows the histogram of the resulting probability distribution of the ID distances between nodes and their neighbors. The total ID ring length is normalized to be 1.0 and the bin size of the histogram is 0.01. Each point on the graph shows the total number of routing entries at a certain ID distance away from a node. The resulting probability density distribution matches the ideal $1/x$ distribution well.

already follow a small world distribution, it also learns new entries following a small world distribution from each lookup acknowledgment. A node inserts these newly learnt neighbors in its routing table and uses them to route future lookups and learns new entries from. In practice, even if nodes start out with only random routing entries, learning from recursive lookups makes routing tables converge to a small world distribution. We demonstrate this convergence using a simple simulation experiment in a 3000-node static network. Each node initializes its routing table with 10 random neighbors and 1 successor entry. Figure 5-1 shows the probability density distribution of the ID distances between a node and its neighbor after 30,000 lookups for random keys which matches a $1/x$ distribution well.

In reality lookup keys are not necessarily uniformly distributed, thus Accordion should actively look for a small number of neighbors (e.g. $\log n$ neighbors) with IDs sampled from the small-world distribution. Since routing tables with s entries lead to $O(\frac{\log n}{\log s})$ lookup hops, a node only requires a small number of such neighbors to guarantee a small lookup hopcount. Therefore, the bandwidth consumed in

active exploration can be very small.

Chapter 4 has shown that parallelizing lookups uses bandwidth more efficiently than active neighbor exploration. This is because increased parallel lookup traffic leads to increased opportunities of learning new routing entries, while at the same time reduces the effect of timeouts on the overall lookup latency. Therefore, Accordion adjusts the degree of lookup parallelism based on observed lookup load to use most of its bandwidth budget.

5.5 Routing State Freshness

Chapter 4 shows that all DHT protocols need to ensure the freshness of routing entries in order to minimize the effects of lookup timeouts in the face of churn. For example in Chord and Tapestry, a node pings each neighbor every t seconds and deletes a routing entry if it receives no response from the corresponding neighbor. In other words, a Chord or Tapestry node never uses any routing entries that it has not successfully contacted in the last t seconds. Instead of explicitly pinging each neighbor, a node can simply evict neighbors that are likely dead without extra communications (e.g. if the neighbor has not been contacted in the last t seconds). How aggressively should a node evict its routing entries? On one hand, evicting entries quickly results in a small routing table and many lookup hops. On the other hand, evicting entries lazily leads to a large routing table but lookups are likely to suffer from many timeouts. An Accordion node must make a tradeoff between the freshness and the size of its routing table. In order to design an optimal eviction process, we must be able to characterize the freshness of a routing entry. We derive the optimal eviction threshold and present ways to estimate a routing entry's freshness without explicit communication.

5.5.1 Characterizing Freshness

A node never knows for sure if a neighbor is still alive. Nevertheless, we can characterize the freshness of a routing entry probabilistically by estimating p, the probability of a neighbor being alive. The eviction process deletes a neighbor from the table if the estimated probability of it being alive is below some threshold p_{thresh}.

If node lifetimes follow a memoryless exponential distribution, p is determined completely by Δt_{since}, where Δt_{since} is the time interval since the neighbor was last known to be alive. Intuitively, Δt_{since} measures the "age" of the information that the node was alive Δt_{since} ago. The larger the Δt_{since}, the less likely the node is still alive now. However, in real systems, the distribution of node lifetimes is often heavy-tailed: nodes that have been alive for a long time are more likely to stay alive for an even longer time. In other words, with a heavy-tailed node lifetime distribution, p should be estimated using both how long the node has been in the network, Δt_{alive} as well as Δt_{since}.

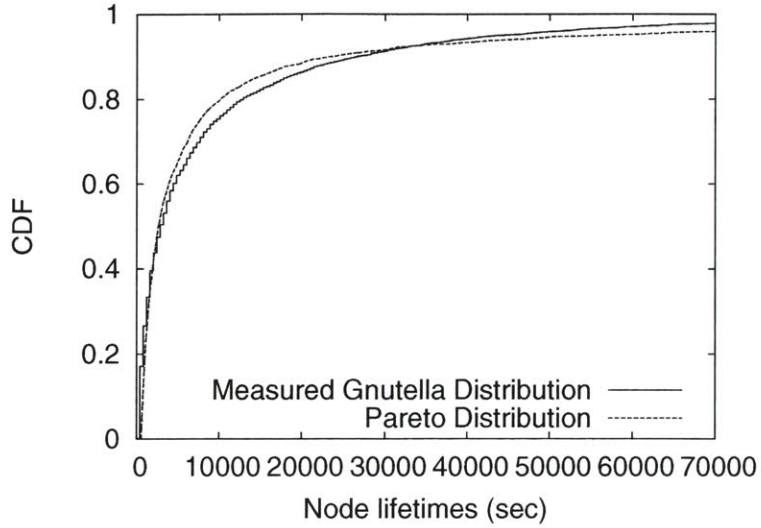


Figure 5-2: Cumulative distribution of measured Gnutella node uptime [32] compared with a Pareto distribution using $\alpha = 0.83$ and $\beta = 1560$ sec.

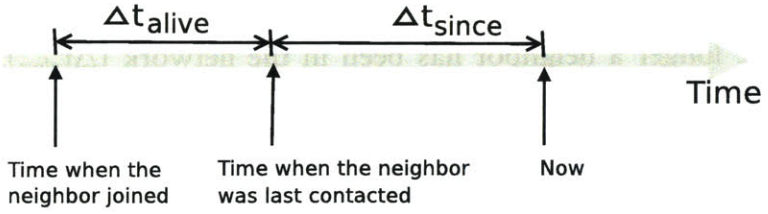


Figure 5-3: The age of a routing entry (Δt_{since}) is the time interval since the corresponding neighbor was last contacted by some node in the system. The known lifetime (Δt_{alive}) of a routing entry is the uptime of the corresponding neighbor when it was last contacted.

If node lifetimes follow a Pareto distribution, a heavy-tailed distribution, the probability of a node dying before time t is

$$\Pr(\textit{lifetime} < t) = 1 - \left(\frac{\beta}{t}\right)^{\alpha}$$

where α and β are the shape and scale parameters of the distribution, respectively. Saroiu et al. observed such a distribution in a study of the Gnutella network [32]; Figure 5-2 compares their measured Gnutella lifetime distribution with a synthetic heavy-tailed Pareto distribution (using $\alpha = .83$ and $\beta = 1560$ sec). We will present our derivation of the best eviction threshold (p_{thresh}) assuming a Pareto node lifetime distribution with $\alpha = 1$.

Let Δt_{alive} be the time for which the neighbor had been a member of the DHT network, at the time it was last heard, Δt_{since} seconds ago. Figure 5-3 illustrates the relationships between the two intervals Δt_{alive} and Δt_{since}. The conditional probability of a neighbor being alive now is:

$$\begin{aligned} p &= \Pr(\textit{lifetime} > (\Delta t_{alive} + \Delta t_{since}) \mid \textit{lifetime} > \Delta t_{alive}) \\ &= \frac{\Pr(\textit{lifetime} > (\Delta t_{alive} + \Delta t_{since}))}{\Pr(\textit{lifetime} > \Delta t_{alive})} \\ &= \frac{(\frac{\beta}{\Delta t_{alive}+\Delta t_{since}})}{(\frac{\beta}{\Delta t_{alive}})} \\ &= \left(\frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}}\right) \end{aligned} \tag{5.1}$$

Equation 5.1 matches our intuition that the bigger a routing entry's age (Δt_{since}) is, the less likely the corresponding neighbor is still alive now (i.e. a smaller p). Furthermore, the longer a neighbor has been in the network (Δt_{alive}), the more likely the neighbor is still alive.

From Equation 5.1, we obtain that $\Delta t_{since} = \Delta t_{alive}(p^{-1} - 1)$. Since Δt_{alive} follows a Pareto distribution, the median lifetime is 2β. Therefore, within $\Delta t_{since} = 2\beta(p_{thresh}^{-1} - 1)$ seconds, half of the routing table are evicted if the eviction threshold is set at p_{thresh}. If s_{tot} is the total routing table size, the eviction rate (E) is approximately:

$$\begin{aligned} E &= \frac{s_{tot}}{2} \cdot \frac{1}{\Delta t_{since}} \\ &= \frac{s_{tot}}{2} \cdot \frac{1}{2\beta(p_{thresh}^{-1} - 1)} \end{aligned} \tag{5.2}$$

The bigger the eviction threshold p_{thresh}, the faster a node evicts to ensure all the remaining neighbors are alive with probability greater p_{thresh}.

Since Accordion nodes aim to keep their maintenance traffic below a certain bandwidth budget, they can refresh or learn about new neighbors only at some finite rate determined by the budget. For example, if a node's bandwidth budget is 20 bytes per second, and learning liveness information for a single neighbor costs 4 bytes (*e.g.*, the neighbor's IP address), then at most a node could refresh or learn routing table entries for 5 nodes per second.

Suppose that a node has a bandwidth budget such that it can afford to refresh or learn about B nodes per second. At equilibrium, the eviction rate is equal to the learning rate ($E = B$). Substituting E with results from Equation 5.2, the routing table size s_{tot} at the equilibrium can be calculated as:

$$\begin{aligned} \frac{s_{tot}}{2} \cdot \frac{1}{2\beta(p_{thresh}^{-1} - 1)} &= B \\ \Rightarrow s_{tot} &= 2B \cdot 2\beta(p_{thresh}^{-1} - 1) \\ &= 4B\beta(p_{thresh}^{-1} - 1) \end{aligned} \tag{5.3}$$

Equation 5.3 implies that if a node evicts routing entries lazily using a smaller p_{thresh}, it will end up with a larger routing table (s_{tot}). However, with probability $1 - p_{thresh}$, each of the s_{tot} routing entries points to dead neighbors and does not contribute to lowering lookup hops. Hence, the *effective* routing table size s, consisting of only live neighbors, is:

$$\begin{aligned} s &= s_{tot} \cdot p_{thresh} \\ &= 4B\beta(p_{thresh}^{-1} - 1) \cdot p_{thresh} \\ &= 2B\beta \cdot (2 - 2p_{thresh}) \end{aligned} \tag{5.4}$$

We have derived Equation 5.4 under the Pareto node lifetime assumption. We can use the same techniques to derive the effective table size (s) under the equilibrium with different node lifetime distributions. For example, with a memoryless exponential distribution with mean lifetime 2β, the effective table size s is:

$$s = 2B\beta \cdot \log \frac{1}{p_{thresh}} \cdot p_{thresh} \tag{5.5}$$

Similarly, with a uniform random distribution with mean lifetime 2β, the effective table size s is:

$$s = 2B\beta \cdot (2 - p_{thresh}) \tag{5.6}$$

5.5.2 Choosing the Best Eviction Threshold

The eviction process deletes routing entries whose estimated probability of being alive (p) is less than some threshold p_{thresh}. Our goal is to choose a p_{thresh} that will minimize the expected number of hops for each lookup including timeout retries.

We know from Section 5.3 that the average number of hops per lookup in a static network is $O(\frac{\log n \cdot \log\log n}{\log s})$; under churn, however, each hop successfully taken has an extra cost associated with it, due to the possibility of forwarding lookups to dead neighbors. When each neighbor is alive with probability at least p_{thresh}, the upper bound on the expected number of trials per successful hop taken is $\frac{1}{p_{thresh}}$ (for now, we assume no parallelism). Thus, we can approximate the expected number of actual hops per lookup, h, by multiplying the number of effective lookup hops with the expected number of trials needed per effective hop:

$$h \propto \frac{\log n \cdot \log\log n}{\log s} \cdot \frac{1}{p_{thresh}}$$

We then substitute the effective table size s using Equation 5.3:

$$h \propto \frac{\log n \cdot \log\log n}{\log(4B\beta(1-p_{thresh}))} \cdot \frac{1}{p_{thresh}} \tag{5.7}$$

The numerator of Equation 5.7 is constant with respect to p_{thresh}, and therefore can be ignored for the purposes of minimization. It usually takes on the order of a few round-trip times to detect lookup timeout and this multiplicative timeout penalty can also be ignored. Our task now is to choose a p_{thresh} that will minimize h^* which is obtained from h without the constant multiplicative factor $\log n \cdot \log\log n$:

$$h^* = \frac{1}{\log(4B\beta(1-p_{thresh}))} \cdot \frac{1}{p_{thresh}} \tag{5.8}$$

The minimizing p_{thresh} depends on the constant $B\beta$. If p_{thresh} varied widely given different values of $B\beta$, nodes would constantly need to reassess their estimates of p_{thresh} using rough estimates of the current churn rate and the bandwidth budget. Fortunately, this is not the case.

Figure 5-4 plots h^* with respect to p_{thresh}, for various values of $B\beta$. We consider only values of $B\beta$ large enough to allow nodes to maintain a reasonable number of neighbors under the given churn rate. For example, if nodes have median lifetimes of 10 seconds ($\beta = 5$ sec), but can afford to refresh or learn one neighbor per second, no value of p_{thresh} will allow its routing table to contain more than 10 entries.

Figure 5-4 shows that as p_{thresh} increases the expected lookup hops decreases due to fewer timeouts; however, as p_{thresh} approaches 1, the number of hops actually increases due to a smaller routing table size. The p_{thresh} that minimizes lookup hops lies somewhere between .8 and .95 for all curves. Figure 5-4 also shows that as $B\beta$ increases, the p_{thresh} that minimizes h^* increases as well, but only slightly. In fact, for any reasonable value of $B\beta$, h^* varies little near its minimum that we can approximate the optimal p_{thresh} for any value of $B\beta$ to be .9.

Equation 5.8 is derived under the assumption of Pareto node lifetime distribu-

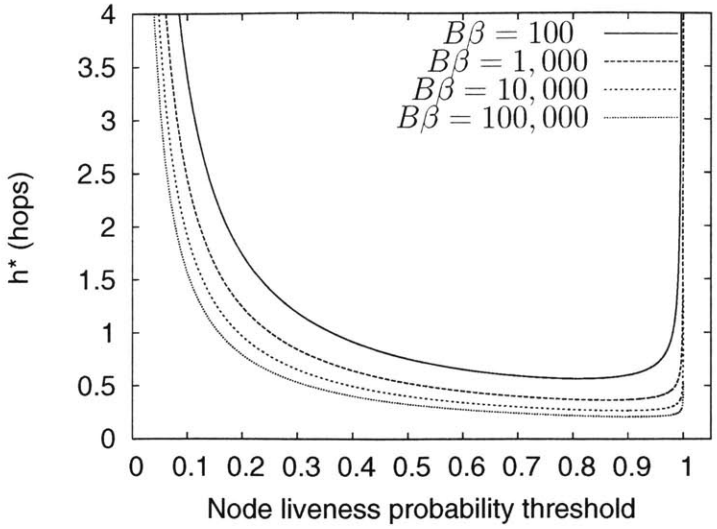


Figure 5-4: The function h^* (Equation 5.8) with respect to p_{thresh}, for different values of $B\beta$. h^* goes up as p_{thresh} decreases due to increased timeouts to stale routing entries. h^* goes to infinity as p_{thresh} approaches 1 due to a smaller routing table leading to more lookup hops.

tion. Analysis with exponential and uniform random node lifetime distributions (using Equation 5.5 and Equation 5.6) results in similar optimal p_{thresh} of .9. This explains our observation in the previous chapter that there seems to be a best staleness threshold for routing entries. In Table 4-4 from Chapter 4, we find the best periodic pinging interval for Chord in simulations to be $144s$, which is equivalent to setting $p_{thresh} = \Pr(\textit{lifetime} > 144) = e^{\frac{144}{3600}} \approx 0.96$ with exponentially distributed node lifetime with a mean of 3600 seconds. In the actual Accordion protocol, we set $p_{thresh} = .9$, because even though this value may not be optimal, it will produce an expected number of hops that is nearly minimal in most deployment scenarios.

The above analysis for p_{thresh} assumes no lookup parallelism. If lookups are sent along multiple paths concurrently, nodes can use a much smaller value for p_{thresh} because the probability that *all* of the next hop neighbors are dead is small. Using a smaller value for p_{thresh} leads to a larger effective routing table size, reducing the average lookup hop count. In fact, allowing nodes to use a less fresh but much bigger routing table without suffering too many timeouts is the key benefit of parallel lookups. An Accordion nodes adjusts p_{thresh} based on its current lookup parallelism such that the probability of at least one next hop neighbor being alive is .9. Therefore, if w_p is the current lookup parallelism, an Accordion node should

set the actual eviction threshold to be:

$$\begin{aligned} 1-(1-p_{thresh})^{w_p} &= 0.9 \\ \Rightarrow p_{thresh} &= 1-e^{\frac{\ln 0.1}{w_p}} \end{aligned} \tag{5.9}$$

5.5.3 Calculating Entry Freshness

Nodes can use Equation 5.1 to calculate p, the probability of a neighbor being alive, and then evict entries with $p < p_{thresh}$. Calculating p requires estimating two values: Δt_{alive} and Δt_{since} for each routing entry. Interestingly, p does not depend on the scale parameter β of the Pareto lifetime distribution, which determines the median node lifetime in the system. This is counterintuitive; we expect that the eviction rate increases when the churn rate is higher (i.e. smaller median node lifetimes). Interestingly, this median lifetime information is implicitly present in the observed values for Δt_{alive}, so β is not required to calculate p.

To help other nodes calculate and update Δt_{alive}, each node can piggyback its current uptime information in each packet it sends out. To calculate Δt_{since}, each node not only counts the time interval since the routing entry has been left un-contacted in its own routing table, but it also takes into account how long the routing entries have aged in other nodes' routing tables at the time it first acquired the routing entry from those nodes. Each node independently calculates Δt_{alive} and Δt_{since} for each routing entry. The estimated values do not reflect the current global information about the neighbor; e.g. the estimating node is not aware that some other node may have contacted the neighbor in the last Δt_{since}. Therefore, p is only a local estimate of the probability of a neighbor being alive.

5.6 Discussions

In this chapter, we have presented a design of Accordion's routing table maintenance process. The design allows a node to have a variable routing table size so it can shrink or expand its routing table to obtain best lookup performance using bounded maintenance traffic. We use many insights from Chapter 4 to choose specific techniques that can use bandwidth most efficiently for low latency lookups. Here we discuss a number of high level design decisions and their consequences.

Under a deployment scenario with high bandwidth budget and relatively low churn, the Accordion table maintenance process will result in complete routing state and one hop lookups. But does Accordion use bandwidth optimally in these scenarios to perform one hop lookups? Let us assume the presence of an oracle that detects nodes join and leave events instantaneously and informs all other nodes immediately to maintain complete and accurate routing state. Suppose the median node lifetime in the system is 2β. Then within 2β seconds, half of the

node population have left the system and are replaced by newly joined nodes, resulting in n total join and leave events. The total bandwidth consumption for the oracle to send all n join and leave events to all nodes in the system is $\frac{n}{2\beta}$ bytes/second per node assuming optimistically that each event takes 1 byte and there is no packet header overhead. This is the minimal bandwidth consumption *any* protocol has to incur in order to maintain complete routing state. According to Equation 5.4, Accordion requires each node to consume $B = \frac{n}{4\beta(1-p_{thresh})} = \frac{5n}{2\beta}$ bytes/second to keep an effective routing table size of n using the recommended eviction threshold ($p_{thresh} = .9$). Therefore, Accordion consumes 5 times the absolute minimal bandwidth to maintain mostly accurate complete routing state. However, the above analysis assumes that the amount of application lookup traffic exceeds Accordion's bandwidth budget so there is no redundant parallel lookups (i.e. no overhead maintenance traffic). For example, if the bandwidth budget allows lookup parallelism to be set at 2, a node requires only $\frac{n}{1.28\beta}$ bytes/second using the eviction threshold $p_{thresh} = 0.68$ (from Equation 5.9) to maintain complete state, only 56% more than the oracle's bandwidth consumption.

In designing Accordion, we decide *not* to send explicit notification messages of node leave events like that in OneHop [33]. This is the main reason Accordion tends to use more bandwidth than the oracle's absolute minimal in scenarios when nodes can afford to keep full state. Each Accordion node independently evicts a routing entry if the neighbor has less than p_{thresh} estimated probability of being alive instead of evicting an entry *only* after receiving an explicit notification message that the neighbor has actually left the network. The decision of not sending explicit node leave notifications is two fold. First, it is hard to detect node failures correctly over the wide area network. Unlike a local area cluster, wide area network connections suffer from transient disruptions [4, 30, 45] caused by link failures, BGP routing updates and ISP peering disputes etc. For example, a certain wide area link failure prevents part of the network from communicating with y directly, but not the rest. As a result, there is going to be an oscillation of join and leave notifications about y as nodes have contradicting beliefs on whether or not y is in the network. Second, Accordion uses a variable per-node routing table size. In OneHop, all leave events are always propagated to all nodes, as OneHop aims to keep complete state. In contrast, Accordion nodes do not try to keep complete state in scenarios when the churn is high relative to the bandwidth budget. Therefore, it is difficult and expensive for a node to find out those nodes that contain the corresponding dead routing entry in order to send the leave event to. Our evaluations in Chapter 7 show that although Accordion is not optimized for those scenarios when nodes can afford to keep full state, its lookup latency still outperforms that of OneHop.

Accordion requires users to explicitly specify a bandwidth budget. Each node bounds its bandwidth consumption according to its budget based on the current workload. In order to be robust against unpredictable and changing environ-

ments, it is essential that DHT nodes have the ability to control its bandwidth use to avoid overloading the network. However, it is less important for nodes to observe a specific bandwidth constraint. What Accordion really needs is a binary signal from the environment that informs a node whether or not the network is currently being overloaded so a node can decide if it should send more maintenance traffic. Unfortunately, it is not sufficient to simply rely on packet losses as an indication of network overload as losses also arise when nodes send packets to failed neighbors. Therefore, more sophisticated mechanisms are required to correctly detect network overload and signal an Accordion node to cut back its bandwidth use. We leave the detection mechanism to future work and adopt the simple design of letting users explicitly specify a desired bandwidth consumption. In practice, users can set a node's bandwidth budget to be some small fraction of the node's access link capacity.

Accordion uses the observation that node lifetime distribution is often heavy tailed to improve the accuracy of its estimate of p, the probability of a neighbor being alive. Therefore, nodes that have been in the network for a longer time appear in more nodes' routing tables because they are judged to have a higher probability of being alive. This causes Accordion to bias lookups towards neighbors that have been in the network for a long time. Fortunately, the imbalance in lookup traffic is not severe as evaluations as Chapter 7 will show.

Chapter 6

Accordion Implementation

After an Accordion node has joined an existing network, it starts to populate and maintain its routing table. Accordion's routing table maintenance process consists of two sub-processes; routing state acquisition and eviction. The overall structure of Accordion is depicted in Figure 6-1. Accordion's routing state acquisition process obtains routing entries predominantly by learning opportunistically through lookup traffic. Each node spends most of its bandwidth budget performing parallel lookups. Accordion's routing state eviction process calculates the known uptime (Δt_{alive}) and age ($talive$) of each routing entry to avoid routing through neighbors that are likely dead. The routing table size is simply the equilibrium of the acquisition and eviction process. The bigger the budget, the faster a node learns, resulting in a larger routing table and low lookup latency. The bigger the churn rate, the faster a node evicts, leading to a smaller routing table with bounded bandwidth overhead.

The pseudocode in Figure 6 outlines Accordion's routing table maintenance and lookup procedures. In this Chapter, we explain in detail how an Accordion node acquires new entries via learning from lookups and active exploration, how it evicts stale entries and how it chooses the best set of next hop neighbors to forward a lookup message to.

6.1 Bandwidth Budget

The owner of each Accordion node independently sets a bandwidth budget for the desired amount bandwidth consumption. Each Accordion node controls how to best consume the "available" portion of budget left over after sending all lookup traffic. Accordion's strategy for using its available bandwidth is to use as much of the bandwidth budget as possible on lookups by exploring multiple paths in parallel. When some bandwidth is left over (perhaps due to bursty lookup traffic), Accordion uses the rest to explore; that is, to find new routing entries according to a small-world distribution.

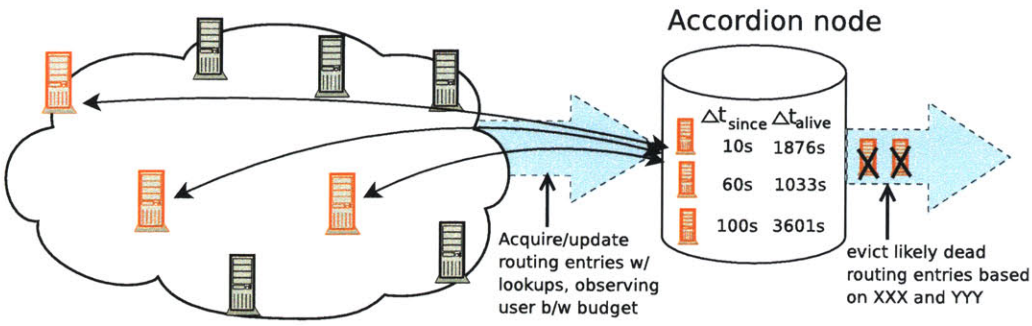


Figure 6-1: The overall architecture of Accordion. Each Accordion node consists of an in-memory routing table. A node acquires and updates its routing entries predominantly by learning through lookup traffic. A node adjusts its lookup parallelism based on observed lookup workload to fill up its bandwidth budget, specified by the user. A node evicts routing entries that are likely dead based on their corresponding Δt_{since} and Δt_{alive}. The size of the routing table is the equilibrium between the routing acquisition and eviction process, reflecting the bandwidth budget and churn rate.

The user specifies the bandwidth budget in two parts: the average desired rate of traffic in bytes per second (r_{avg}), and the maximum burst size in bytes (b_{burst}). Each node maintains an integer variable, b_{avail}, to keep track of how much of the budget is left over from lookups and available for maintenance traffic (including exploration and parallel lookups), based on recent activity. Every t_{inc} seconds, the node increments b_{avail} by $r_{avg} \cdot t_{inc}$ (where t_{inc} is the size of one typical packet divided by r_{avg}). Each time the node sends an RPC packet or receives the corresponding acknowledgment (for any type of traffic), it decrements b_{avail} by the size of the packet. Nodes decrement b_{avail} down to a minimum of $-b_{burst}$. While $b_{avail} = -b_{burst}$, nodes stop sending all maintenance traffic (such as redundant lookup traffic and exploration traffic). Nodes do not decrement b_{avail} for unsolicited incoming traffic, or for the corresponding outgoing acknowledgments. In other words, each packet counts towards only the bandwidth budget at one end.

Every t_{inc} seconds, a node checks if b_{avail} is positive. If so, the node sends one exploration packet, according to the algorithm we present in Section 6.2.2. Thus, nodes send no exploration traffic unless the average traffic over the last b_{burst}/r_{avg} seconds has been less than r_{avg}.

The bandwidth budget aims to limit the maintenance traffic generated by an Accordion node, but does not give the node any control over its incoming traffic or outgoing forwarded lookup traffic. For example, a node must acknowledge all traffic sent to it from its predecessor regardless of the value of b_{avail}; otherwise, its predecessor may think it has failed and the correctness of lookups would be

```
procedure STARTMAINT()
 // update b_avail periodically using budget (Section 6.1)
 UPDATEBAVAIL() periodically

 // adjust lookup parallelism periodically. (Section 6.4.1)
 w_p ← ADJUSTPARALLELISM() periodically

 // evicts stale entries according to the current lookup parallelism. (Section 6.3)
 freshtable ← EVICT(table, w_p)

 // actively explore for more entries if budget permits (Section 6.2.2)
 (n, end_id) ← EXPLORE(freshtable) periodically
 if n not NULL
     v = n.GETNODES(me, w_p, end_id, 5)
     table ← table ∪ v

procedure LOOKUP(lookup_request q)
 if this node owns q.key
     reply to lookup source directly
     return

 // help previous hop learn new entries (Section 6.2.1)
 ack ← GETNODES(q.prev_hop, q.prev_w_p, q.key, 5)
 send ack to q.prev_hop

 // decide on lookup parallelism (Section 6.4.1)
 para ← GETPARALLELISM()

 // pick the best set of next hops (Section 6.4.2, 6.4.3)
 freshtable ← EVICT(table, para)
 nexts ← NEXTHOPS(freshtable, q.key, para)

 // send out copies of the lookup and learn from each reply
 q.prev_hop = me
 q.prev_w_p = para
 for each n in nexts
     v ← n.LOOKUP(q)
     table ← table ∪ v
```

Figure 6-2: The outline of Accordion's routing table maintenance and lookup procedures. n.LOOKUP invokes the LOOKUP function on the remote node n. The global variable *t*able denotes a node's current routing table and w_p denotes a node's current lookup parallelism. Section numbers in the comments refer to the sections where the corresponding functions will be defined.

```
// choose m entries to piggyback in the lookup acknowledgment
// or in the reply to an active exploration message
// m is usually a small constant e.g. 5
procedure GETNODES(src, para, k, m)
 // only consider entries fresher than src node's eviction threshold
 freshtable ← EVICT(table, para)
 s ← neighbors in freshtable between me and k
 if s.SIZE() < m
     v ← s
 else
     v ← m nodes in s with the smallest VIVALDIRTT() to src
 return (v)
```

Figure 6-3: Learning from lookups in Accordion. Accordion includes the m entries with the smallest network delay to the src node in the lookup acknowledgment or the reply to an active exploration message. Furthermore, these m entries have a higher probability of being alive than src node's eviction threshold.

compromised. The imbalance between a node's specified budget and its actual incoming and outgoing traffic is of special concern in scenarios where nodes have heterogeneous budgets. To help nodes with low budgets avoid excessive incoming traffic from nodes with high budgets, an Accordion node biases lookup and table exploration traffic toward neighbors with higher budgets. Section 6.4.2 describes the details of this bias.

6.2 Acquiring Routing State

An Accordion node acquires most routing entries from piggybacked entries in the lookup acknowledgment messages. When an Accordion node has not successfully used up all its budget for performing parallel lookups, it explicitly explores for new routing entries following a small world distribution.

6.2.1 Learning from lookups

An Accordion node forwards a lookup greedily in ID space. Each node chooses a next hop node among its current neighbors with the closest ID distance to the lookup key. We will relax this basic forwarding rule in later sections 6.4.2 and 6.4.3 to accommodate nodes with different bandwidth budget and network proximity. Each intermediate node acknowledges the receipt of every lookup. The acknowledgment serves to indicate that the next hop node is currently alive. In addition, an Accordion node piggybacks m entries from its own routing table in

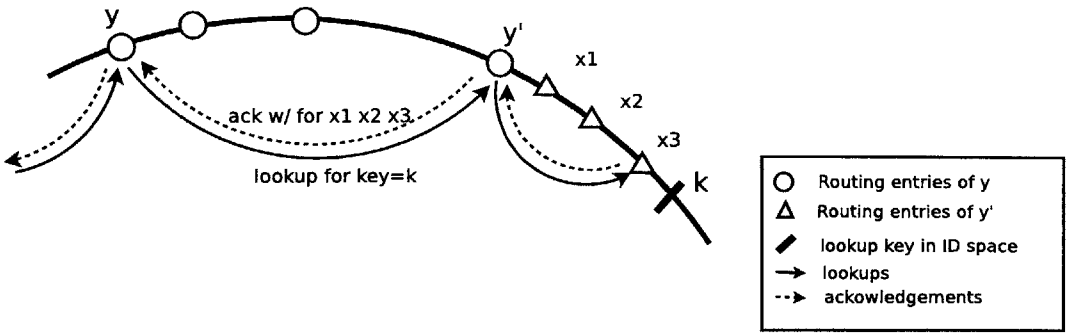


Figure 6-4: Learning from recursive lookups in Accordion. Node y forwards a lookup for key k to the next hop y' that is closest to k in ID space. y' acknowledges the lookup message piggybacking additional routing entries from its own routing table (x_1,x_2,x_3) whose node IDs lie between itself and the lookup key k. The shaded region denotes the ID range from which y' chooses its routing entries to include in the acknowledgment to y.

the acknowledgment message to help the previous hop learn from lookups. In our implementation, m is usually set to be some small constant (e.g. 5).

Figure 6.2.1 shows how an Accordion node learns from forwarding a lookup message. If y forwards a lookup for key k to the next hop y', y' piggybacks a m routing entries in the ID range between y' and k from its routing table. As lookups usually require more than one hop to reach the predecessors, a node forwards more lookup messages than it originates. Since forwarded lookups are for keys in closer-by ID space, acquiring new entries from the next hop allows a node to preferentially learn about new entries close to itself in ID space, the key characteristic of a small-world distribution. Additionally, the fact that y forwards the lookup to y' indicates that y does not know of any neighbors in the ID gap between y' and k, and y' is well-situated to supply routing entries that fill this gap. In Figure 6, we have shown the pseudocode of a node forwarding a lookup to its next hop. Figure 6.2.1 shows how y' chooses a subset of its routing entries to include in its lookup acknowledgment message to y. y' only consider entries whose probability of being alive is higher than y's eviction threshold as calculated using y's lookup parallelism which is included in the lookup message. Node y' chooses entries with the lowest predicted delay to y. Accordion uses Vivaldi [16], a network coordinate system, to predict the round trip delay between any pair of nodes given their synthetic coordinates.

6.2.2 Active Exploration

Accordion attempts to use most its budget for parallel lookups by periodically adjusting the lookup parallelism (w_p). When lookup traffic is bursty, Accordion is not always able to set w_p perfectly for the next time period. As such, parallel lookups would not consume the entire bandwidth budget during that time period. When the parallelism window (w_p) is too small for the current level of lookup traffic, b_{avail} becomes positive indicating there is leftover bandwidth to explore for new neighbors actively. The main goal of exploration is that it be bandwidth-efficient and result in learning nodes with the small-world distribution described in Section 5.3.

For each neighbor at x ID-distance away from a node, the gap between that neighbor and the next successive entry should be proportional to x. A node with identifier y compares the *scaled gaps* between its successive neighbors n_i and n_{i+1} to decide the portion of its routing table most in need of exploration. The scaled gap $g(n_i)$ between neighbors n_i and n_{i+1} is:

$$g(n_i) = \frac{d(n_i, n_{i+1})}{d(y, n_i)}$$

where $d(y, n_i)$ computes the clockwise distance in the circular identifier space between identifiers y and n_i. When node y sends an exploration message, it chooses to send to the neighbor n_i with the largest scaled gap ($g(n_i)$). The result is that the node explores in the area of ID space where its routing table is the most sparse with respect to the desired small world distribution.

An exploration message from node y asks neighbor n_i for m routing entries between n_i and n_{i+1}. Node n_i chooses m entries from its routing table in the same way as it chooses entries for learning from lookups (see Figure 6.2.1). If n_i returns fewer than m entries, node y will not revisit n_i again until it has explored all its other neighbors. Figure 6.2.2 presents the pseudocode for deciding which neighbor to explore new entries from.

6.3 Evicting Stale Routing State

Accordion estimates a routing entry's liveness probability based on Δt_{alive} (the time between when the neighbor last joined the network and when it was last heard) and Δt_{since} (the time between when it was last heard and now). A routing entry's liveness probability (p) is approximated as $p \approx \frac{\Delta t_{alive}}{\Delta t_{since} + \Delta t_{alive}}$. The estimate is exact when the node lifetime follows a Pareto distribution with shape parameter $\alpha = 1$.

Each node keeps track of its own Δt_{alive} based on the time of its last join, and includes its current Δt_{alive} in every packet it sends. Nodes learn (Δt_{alive}, Δt_{since}) information associated with neighbors in one of the following three ways:

```
// decide which neighbor to explore
procedure EXPLORE(freshtable)
 // only explore when there is leftover budget from lookups
 if b_avail ≤ 0
     return (NULL)
 pn = IDSUCC(freshtable, me.id)
 n = IDSUCC(freshtable, pn.id)
 s ← {}
 while n.id not me.id
     if pn not been directly contacted before
         pn.gap = n.id − pn.id
         pn.gap = pn.gap/n.id
         s ← s ∪ {pn}
         pn = n
         n = IDSUCC(freshtable, pn.id)
 n ← node in s with biggest gap
 return (n, IDSUCC(freshtable, n.id))
```

Figure 6-5: Accordion chooses the neighbor with the biggest scaled gap to send an exploration message. IDSUCC($freshtable, n.id$) returns the immediate successor in $freshtable$ after node n.

- When the node hears from a neighbor directly, it records the current local timestamp as t_{last} in the routing entry for that neighbor, and resets an associated Δt_{since} value to 0 and sets Δt_{alive} to the newly-received Δt_{alive} value.
- If a node hears information about a new neighbor indirectly from another node, it will save the supplied (Δt_{alive},Δt_{since}) pair in the new routing entry, and set the entry's t_{last} value to the current local timestamp.
- If a node hears information about an existing neighbor, it compares the received Δt_{since} value with its currently recorded value for that neighbor. A smaller received Δt_{since} indicates fresher information about this neighbor, and so the node saves the corresponding (Δt_{alive}, Δt_{since}) pair for the neighbor in its routing table. It also sets t_{last} to the current local timestamp.

Whenever a node needs to calculate a current value for Δt_{since} (either to compare its freshness with a newly received entry with the same ID, or to pass it to a different node), it adds the saved Δt_{since} value and the difference between the current local timestamp and t_{last}.

The freshness threshold of the routing entries p_{thresh} depends on the lookup parallelism (w_p) a node currently uses (Equation 5.9). Since a node changes its lookup parallelism from time to time based on observed lookup load, it does

```
// return a list of fresh routing entries based on current eviction threshold
procedure EVICT(table, p)
 // calculate eviction threshold based on parallelism
 p_thresh = 1 - (exp * (log(1 - 0.9)/p))
 v ← {}
 for each n in table
     p = n.Δt_alive/(n.Δt_alive + n.Δt_since)
     if p > p_thresh
         v ← v ∪ {n}
 return (v)
```

Figure 6-6: Accordion precludes stale routing entries from being used in lookups based on the current eviction threshold p_{thresh}.

not physically evicts routing entries based on p_{thresh}. Rather, a node simply precludes stale routing entries from being used as next hop nodes if their estimated probability of being alive is less than p_{thresh}. Figure 6.3 shows the pseudocode for the EVICT function that filters routing entries based on their estimated liveness and the current p_{thresh}.

6.4 Accordion Lookups

Accordion needs to adjust lookup parallelism based on observed lookup load. It also carefully chooses the set of next hop nodes, balancing the need to make good lookup progress in ID space and to use low delay and high budget neighbors.

6.4.1 Parallelizing Lookups

Accordion parallelizes recursive lookups by sending a small number of copies of each lookup to a set of neighbors. A node increases the parallelism of lookup messages it initiates and forwards until the point where the lookup traffic nearly fills the bandwidth budget. A node must adapt the level of parallelism as the underlying lookup workload changes and it must also avoid forwarding the same lookup twice.

A key challenge in Accordion's parallel lookup design is caused by its use of recursive routing. Previous DHTs with parallel lookups use iterative routing: the originating node sends lookup messages to each hop of the lookup in turn [48,58]. Iterative lookups allow the originating node to explicitly control the amount of redundant lookup traffic and the order in which paths are explored, since the originating node issues all messages related to the lookup. However, Accordion uses recursive routing to learn more nodes closer-by in ID space, and nodes forward

```
// periodically adjust the desired lookup parallelism
procedure ADJUSTPARALLELISM()
 // increase parallelism by one, subject to a maximum of 6
 if explored more than forwarded unique lookups in last period
      w_p = MIN(w_p + 1, 6)
 else if explored zero time in last period
      w_p = w_p/2

// decide on the parallelism for the current lookup message
procedure GETPARALLELISM()
 if b_avail < -b_burst
      return (1)
 else if
      return (w_p)
```

Figure 6-7: Accordion adjusts lookup parallelism periodically based on observed lookup load. The lookup parallelism for the current lookup message is set to one once a node has used up its budget.

a lookup directly to its next hop. To control recursive parallel lookups, each Accordion node independently adjusts its lookup parallelism to stay within the bandwidth budget and drops redundant lookups when the bandwidth budget is tight.

If a node knew the near-term future rate at which it was about to receive lookups to be forwarded, it could divide the bandwidth budget by that rate to determine the level of parallelism. Since it cannot predict the future, Accordion uses an adaptive algorithm to set the level of parallelism based on the past lookup rate. Each node maintains a "parallelism window" variable (w_p) that determines the number of copies it forwards of each received or initiated lookup. A node adjusts w_p every t_p seconds, where $t_p = b_{burst}/r_{avg}$, which allows enough time for the bandwidth budget to recover from potential bursts of lookup traffic. A node evaluates whether w_p was too aggressive or conservative based on the amount of unique lookup messages a node has originated or forwarded in the past t_p seconds. If the previous w_p was too big, i.e. the total amount of parallel lookup traffic exceeds what is allowed by the budget, a node decreases w_p by half. Otherwise, w_p is increased by 1. This additive increase and multiplicative decrease (AIMD) style of control resembles the congest control mechanisms in TCP [13, 38] and gives a prompt response to w_p overestimation or sudden changes in the lookup load. Figure 6.4.1 shows the pseudocode for adjusting w_p. For each lookup message a node sends out, it decreases b_{avail} by the number of bytes in the lookup and its acknowledgment packet. A node does not parallelize lookups when b_{avail} is less than $-b_{burst}$ indicating it has already exceeded its budget at the moment.

Nodes do not increase w_p above some maximum value, as determined by the maximum burst size, b_{burst}.

When a node originates a lookup, it marks one of the parallel copies with a "primary" flag which distinguishes that lookup message from other redundant copies of the same lookup. If a node receives a primary lookup, it marks one forwarded copy as primary, maintaining the invariant that there is always one primary copy of a query. When a nodes receives a non-primary copy, it is free to drop the lookup if it does not have sufficient bandwidth ($b_{avail} < -b_{burst}$) to forward the lookup, or if it has already seen a copy of the lookup in the recent past. Allowing a node to drop non-primary copies eliminates the danger of parallel lookup traffic increasing uncontrollably at each hop.

6.4.2 Biasing Traffic to High-Budget Nodes

Because nodes have no direct control over the bandwidth consumed by incoming lookups and exploration packets, in a network containing nodes with diverse bandwidth budgets we expect that some nodes will be forced over-budget by incoming traffic from nodes with bigger budgets. Accordion addresses this budgetary imbalance by biasing lookup and exploration traffic towards nodes with higher budgets. Though nodes still do not have direct control over their incoming bandwidth, in the absence of malicious nodes this bias serves to distribute traffic in proportion to the bandwidth budgets of nodes.

When an Accordion node learns about a new neighbor, it also learns that neighbor's bandwidth budget. Traditional DHT protocols (*e.g.*, Chord) route lookups greedily to the neighbor most closely preceding the key in ID space, because that neighbor is expected to have the highest density of routing entries near the key. We generalize this idea to take into account of different bandwidth budgets and forward lookup or exploration messages to neighbors with the best *combination* of high budget and short ID distance to the lookup or exploration key.

Suppose node y is to forward a lookup packet with key k for which n_1 (with budget b_1) is the neighbor whose ID is the closest to k. Let n_i ($i = 2, 3...$) be neighbors preceding n_1 in ID space from y's routing table, each with a bandwidth budget of b_i. Let $d(n_i, k)$ be the distance in identifier space between n_i and k, hence $d(n_1, k)$ is the minimum distance among all $d(n_i, k)$'s. Since y does not know of any node in the ID interval between n_1 and k, it should forward the lookup to a neighbor having the densest routing entries in that interval to make the best lookup progress in ID space. If v_i is an estimate of the density of a neighbor's routing entries in the ID region between n_i and k, then $v_i \propto \frac{1}{d(n_i,k)}$ since the density of a node's routing entries decrease inverse proportionally to the ID distance from itself. However, bigger budgets lead to proportionally bigger routing tables and hence $v_i \propto b_i$. A node uses v_i as the lookup "progress" metric in choosing which

neighbor to forward the lookup to. In Accordion's traffic biasing scheme, y prefers forwarding the lookup packet to the neighbor n_i with the largest value for v_i:

$$v_i = \frac{b_i}{d(n_i, k)} \tag{6.1}$$

In the case of exploring for routing entries in the biggest scaled gap between n_1 and its successive entry n_0, a node also prefers sending the exploration packet to a neighbor n_i with the largest $v_i = \frac{b_i}{d(n_i, n_0)}$. For each lookup and exploration decision, an Accordion node examines a fixed number of candidate neighbors (set to 8 in our implementation) preceding n_1 to choose the best set of next hops. Even though a node does not send lookup packets to the neighbor (n_1) closest to the key, it learns new entries in the ID range between n_1 and k from the next hop node n_i.

6.4.3 Proximity Lookups

When a node does not have a complete routing table, it has to forward a lookup to an intermediate node. Accordion tries to route lookups to nodes with low roundtrip trip delay using both Proximity Neighbor Selection (PNS) and Proximity Route Selection [29] [18].

With PNS, an Accordion node preferentially acquires new routing entries with low roundtrip delay to itself. For example, in Figure 6.2.1, nodes choose routing entries with low network delays to include in the lookup acknowledgment message to the previous hop. In Chord (base $b = 2$), PNS alone is sufficient to achieve most of the lookup latency reduction and proximity route selection helps little [29]. This is because Chord has only a small number of finger entries chosen carefully with low roundtrip delays for forwarding lookups. In contrast, an Accordion node typically has a much larger routing table. As a node's routing table approaches the complete state, it is no longer the case that most routing entries point to neighbors with low delay. Accordion explicitly weights v_i to bias the next hop selection towards proximate neighbors. We extend the "progress" metric in Equation 6.1 so a forwarding node y chooses the neighbor with the largest v_i':

$$v_i' = \frac{b_i}{d(n_i, k) \cdot delay(y, n_i)} \tag{6.2}$$

where $delay(y, n_i)$ is the predicted roundtrip time between node a and n_i based on their Vivaldi coordinates. Figure 6.4.3 shows the pseudocode that an Accordion node uses in choosing the best w_p next hop nodes to forward a lookup message to.

```
// choose p best next hops to forward a lookup with key k
procedure NEXTHOPS(table, key, p)
 v ← {}
 n ← IDPRED(table, k)
 // choose the best p out of a candidate pool of 8*p nodes
 while |v| < 8 * p
     dist = IDDIST(n.id, k)
     // do not send to neighbors whose IDs are too far from the key
     if dist > IDDIST(me.id, k)/2
         break
     delay = VIVALDIRTT(me.coord, n.coord)
     n.vi = n.budget/(dist * delay)
     v ← v ∪ {n}
     n ← IDPRED(table, n.id)
 nexthops ← p nodes with largest vi's in v
 return (nexthops)
```

Figure 6-8: Accordion chooses the next hop nodes based on a combination of their budgets, ID distance to the lookup key and network proximity. IDDIST($a.id, b.id$) calculates the clockwise distance between two IDs. IDPRED(id, p) returns a neighbor that immediately precedes the given id. VIVALDIRTT($a.coord, b.coord$) calculates the predicted roundtrip time between two nodes using their synthetic coordinates $a.coord$ and $b.coord$.

6.5 Implementation

We have implemented Accordion in both the *p2psim* simulator and the MIT DHash [18] software release. DHash is a distributed hash table that distributes and replicates data among a large number of nodes over the wide area network. It relies on a lookup protocol to find the set of responsible nodes to store and retrieve data from based on the lookup key. The original DHash lookup library consists of two protocols, Chord and Koorde. Accordion is implemented as part of the lookup library that DHash can use instead of Chord.

DHash runs as a user-level daemon process, *lsd*. Figure 6-9 shows the overall architecture of DHash and an example application (UsenetDHT [75]) that uses DHash for its distributed storage. The software is developed using the RPC and asynchronous event handling library from the SFS toolkit [59]. The DHT process (*lsd*) communicates with local applications via UNIX domain sockets and communicates with *lsd*s on remote machines via UDP sockets. Accordion provides lookup service to local applications in two ways: applications can directly invoke an Accordion lookup by issuing the *findroute(k)* RPC via a local UNIX domain socket, or applications can issue the *get(k)* and *put(k,v)* RPCs to DHash which

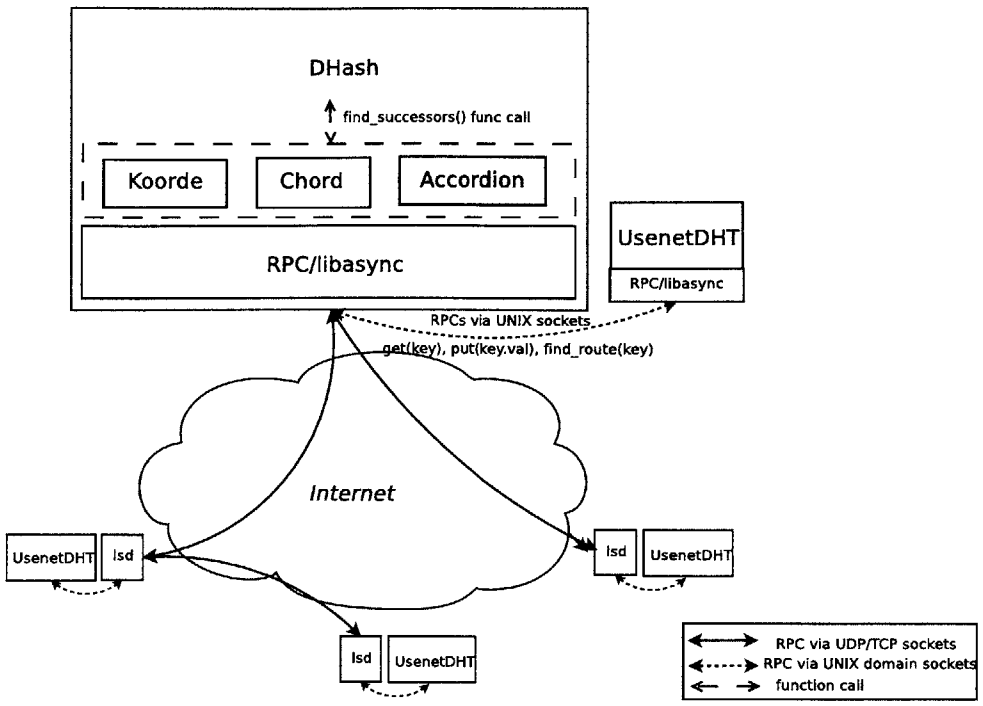


Figure 6-9: The overall architecture of *lsd*, a DHT daemon process that exports the standard get(k), put(k,v) interface to applications. DHash implements the data storage, retrieval and maintenance and use the DHT lookup library to find nodes responsible for a key. Accordion is implemented as part of the lookup library.

further calls the *find_successors(k)* function in Accordion in order to find the responsible node to store or retrieval data from.

Accordion uses consistent hashing to map lookup keys to nodes and therefore re-uses the successor stabilization code from Chord. Accordion obtains its bandwidth budget from a user configuration file at *lsd* startup. The budget only limits Accordion's lookup and routing table exploration traffic and does not apply to the DHash data storage, retrieval and maintenance messages.

Chapter 7

Accordion Evaluation

We demonstrate the important properties of Accordion and compare it to existing DHTs using simulation. In addition, we measure Accordion's performance in a real implementation using wide-area network emulation. Our evaluation shows that Accordion nodes bound their bandwidth consumption effectively according to the bandwidth budget. Accordion automatically tunes itself to provide better latency vs. bandwidth tradeoffs than existing DHT protocols over a wide range of churn and workloads. Specifically, Accordion nodes shrink their routing tables to route lookups in logarithmic number of hops when the budget is small or churn is high. When given a bigger budget or a more stable network, nodes efficiently expand their routing tables to route lookups in one hop to achieve better lookup latency. Furthermore, Accordion's performance degrades only modestly when the node lifetimes do not follow the assumed Pareto distribution.

7.1 Experimental Setup

We use simulations to investigate various properties of Accordion. In simulations, we can run experiments quickly instead of in real time and thus we are able to examine Accordion's performance under a large number of operating environments with different churn rates and workloads. Chapter 4 has shown that Chord has the best latency vs. bandwidth tradeoffs at small bandwidth consumption and OneHop is the best at large bandwidth consumption. Therefore, we compare Accordion's bandwidth efficiencies to that of Chord and OneHop in simulations.

The experimental setups are identical to that in Chapter 4 except for the node lifetime distribution and network topologies. Instead of using exponential lifetime distribution, we uses a Pareto distribution with median lifetime of 1 hour (*i.e.*, $\alpha = 1$ and $\beta = 1800$ sec) to generate node join and leave time. We separately evaluate the scenarios for which the node lifetime distribution does not follow the assumed Pareto distribution. Because of the limited size of our measured roundtrip times between DNS servers (1740×1740), for simulations involving larger networks

we assign each node a random 2D Euclidean coordinate and derive the network delay between a pair of nodes from their corresponding Euclidean distance. The average round-trip delay between node pairs in both the synthetic and measured delay matrices is 178 ms.

All Accordion configurations set $b_{burst} = 100 \cdot r_{avg}$. The size in bytes of each Accordion message is counted as 20 bytes for headers plus 8 bytes for each node mentioned in the message to account for additional routing entry information such as the bandwidth budget, node known uptime (Δt_{alive}), and time since last contacted (Δt_{since}).

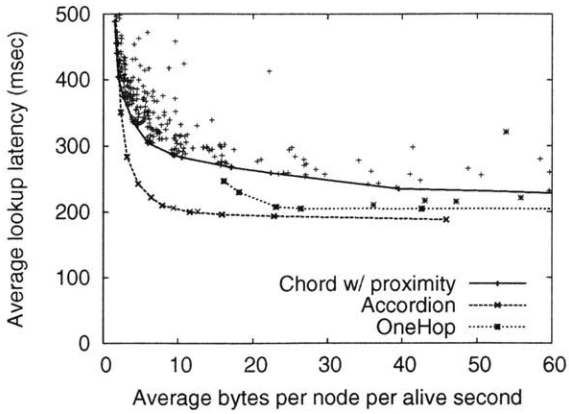
7.2 Latency vs. Bandwidth Tradeoff

A primary goal of the Accordion design is to adapt the routing table size to achieve the lowest latency depending on bandwidth budget and churn. Figure 7-1 plots the average lookup latency vs. bandwidth overhead tradeoffs of Accordion, Chord, and OneHop. The curves for Chord and OneHop are their overall convex hulls obtained by exploring their parameter spaces. There is no parameter exploration necessary for Accordion, we just vary the bandwidth budget (r_{avg}) between 3 and 60 bytes per second. Accordion automatically adapts itself to achieve low lookup latency. We plot the measured bandwidth consumption, not the configured bandwidth budget, along the x-axis. The x-axis values include all traffic; lookups as well as routing table maintenance overhead.

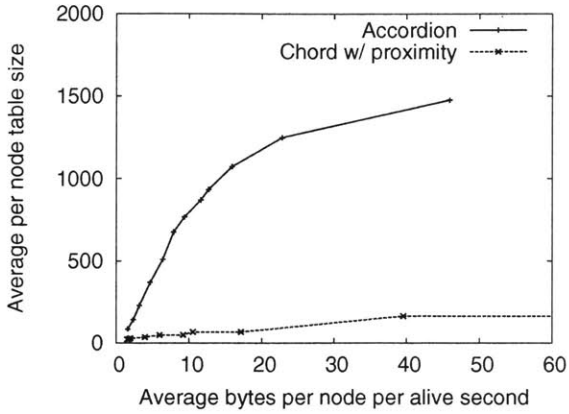
Accordion achieves lower lookup latency that the best OneHop configuration when the bandwidth budget is large, and approximates the latency of the best Chord configuration when bandwidth is small. This is a result of Accordion's ability to automatically adapt its routing table size, as illustrated in Figure 7-1(b).

In Figure 7-1(b), when the budget is limited, Accordion's table size is almost as small as Chord's. As the budget grows, Accordion's routing table also grows, approaching the number of live nodes in the system (on average, half of the 3000 nodes are alive in the system). As protocols use more bandwidth, Chord does not increase its routing table size as quickly as Accordion, even when optimally tuned; instead, a node spends bandwidth on maintenance costs for its slowly-growing table. By increasing the table size more quickly, Accordion reduces the number of hops per lookup, and thus the average lookup latency.

Because OneHop keeps a complete routing table, all join and leave events must be propagated to all nodes in the system. This restriction prevents OneHop from being configured to consume very small amounts of bandwidth. OneHop uses more bandwidth by propagating these events more quickly, so its routing tables are more up-to-date, resulting in fewer lookup hops and timeouts. Accordion, on the other hand, parallelizes lookups to use more bandwidth and lower the eviction threshold of routing entries. Therefore Accordion can cheaply expand its routing



(a) Accordion's latency vs. bandwidth tradeoff compared to Chord and OneHop. Each point represents a particular parameter combination for the given protocol. Accordion's lookup latency is lower than OneHop's when bandwidth is plentiful, and matches that of Chord's when bandwidth is small.



(b) The average routing table size for Chord and Accordion as a function of the average per-node bandwidth. The routing table sizes for Chord correspond to the optimal parameter combinations on Chord's convex hull segment in (a). Accordion's ability to expand its routing table as available bandwidth increases explains why it achieves lower latency than Chord.

Figure 7-1: Accordion's latency vs. bandwidth tradeoff compared to Chord and OneHop, using a 3000-node network and a churn intensive workload.

table size without suffering much from lookup timeouts.

7.3 Effect of a Different Workload

The simulations in the previous section featured a workload that was *churn intensive*; that is, the amount of churn in the network was high relative to the lookup rate. This section evaluates the performance of Accordion under a *lookup intensive* workload. In this workload, each node issues one lookup every 9 seconds, while the rate of churn is the same as that in the previous section.

Figure 7-2 shows the performance results for Chord, OneHop and Accordion. Again, convex hull segments and scatter plots characterize the performance of Chord and OneHop, while Accordion's latency/bandwidth curve is derived by varying the per-node bandwidth budget. Accordion's lookup performance approximates OneHop's when bandwidth is high.

Compared with the churn intensive workload, in the lookup intensive workload Accordion can operate at lower levels of bandwidth consumption than Chord. With a low lookup rate as in Figure 7-1, Chord can be configured with a small

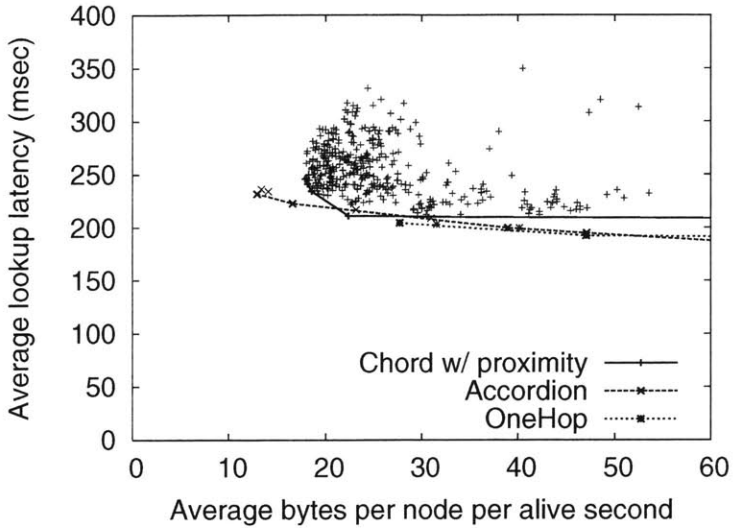


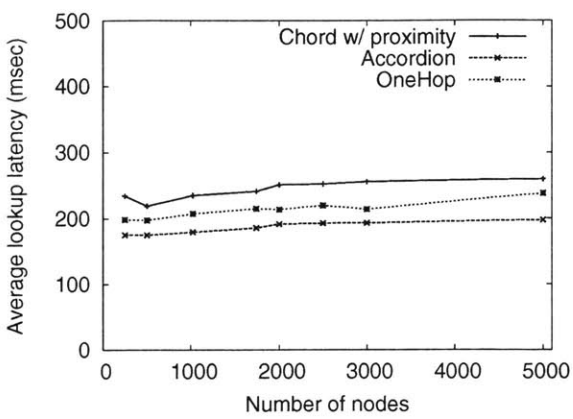
Figure 7-2: Accordion's lookup latency vs. bandwidth overhead tradeoff compared to Chord and OneHop, using a 1024-node network and a lookup intensive workload.

base (and thus a small routing table and more lookup hops) for low bandwidth consumption, with relatively high lookup latencies. However, with a high lookup rate as in Figure 7-2, using a small base in Chord is not the best configuration: it has relatively high lookup latency, but also has a large bandwidth use because each lookup traverses many hops. Because Accordion learns new routing entries from lookup traffic, a higher rate of lookups leads to a larger per-node routing table, resulting in fewer lookup hops and less overhead due to forwarded lookups. Thus, Accordion can operate at lower levels of bandwidth than Chord because it automatically increases its routing table size by learning from the large number of lookups.

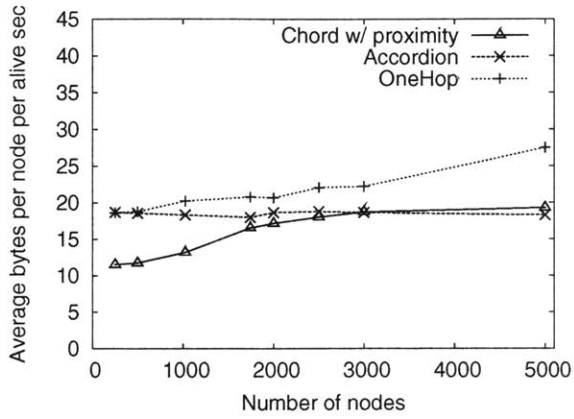
The rest of the evaluation focuses on the churn intensive workload, unless otherwise specified.

7.4 Effect of Network Size

This section investigates the effect of scaling the size of the network on the performance of Accordion. Ideally, we would like to compare Accordion's lookup latency to the best latency that Chord and OneHop can achieve while consuming the same amount of bandwidth by exploring their parameter spaces under different network sizes. However, it is difficult to extract the optimal parameter settings from the overall convex hull for arbitrary bandwidth consumptions, since the convex hull is extrapolated from only a small number of discrete optimal points. Therefore, we choose to fix the parameters for all three protocols under



(a) The average lookup latency of Accordion, Chord and OneHop as a function of network size.



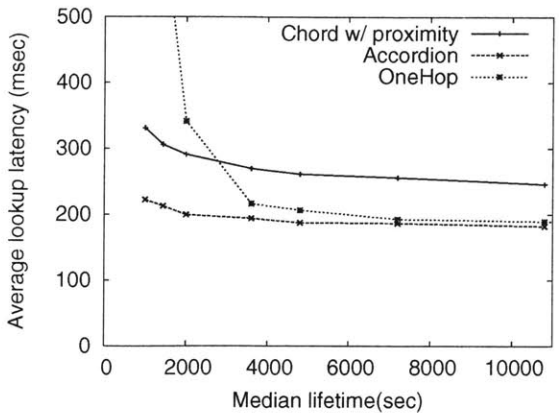
(b) The average bytes consumed per node by Accordion, Chord and OneHop as a function of network size for the same set of experiments as in Figure 7-3.

Figure 7-3: The latency and bandwidth consumption of Accordion, Chord and OneHop as a function of the number of nodes in the system. Accordion uses a bandwidth budget of 6 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 7 and 23 bytes/node/sec in a 3000-node network, respectively.

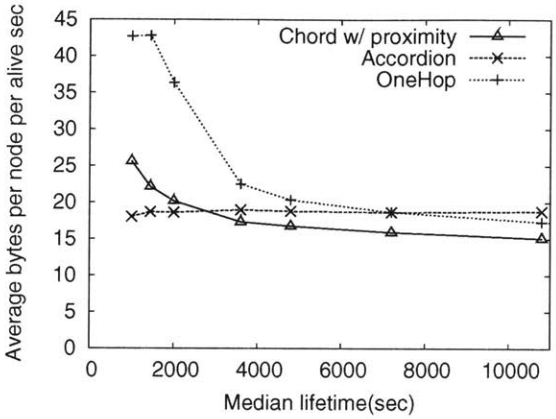
comparison. For Accordion, we fix the bandwidth budget at 24 bytes/sec. For Chord and OneHop, we fix the protocol parameters to be the optimal settings in a 3000-node network for bandwidth consumptions of 17 bytes/node/sec and 23 bytes/node/sec, respectively. These are the parameter combinations that produce latency vs. bandwidth points lying on the convex hull segments and whose corresponding bandwidth consumption most closely approximates Accordion's budget of 24 bytes/sec.

Figures 7-3 shows the average lookup latency and bandwidth consumption of Chord, Accordion and OneHop as a function of network size. With fixed parameter settings, Figure 7-3(b) shows that both Chord and OneHop incur increasing bandwidth overhead that scales as $\log n$ and n respectively, where n is the size of the network. However, Accordion's fixed bandwidth budget results in constant bandwidth consumption regardless of the network size. Despite using less bandwidth than OneHop and the fact that Chord's bandwidth consumption approaches that of Accordion as the network grows, Accordion's average lookup latency is consistently lower than that of both Chord and OneHop.

These figures plot the *average* bandwidth consumed by the protocols, which hides the variation in bandwidth that among different nodes and over time. For Chord and Accordion, the bandwidth consumptions of different nodes are close to the average. OneHop, however, explicitly distributes bandwidth unevenly among nodes: slice leaders typically use 7 to 10 times the bandwidth of average nodes.



(a)The average lookup latency of Accordion, Chord and OneHop as a function of median node lifetime.



(b) The average bandwidth consumption per node by Accordion, Chord and OneHop as a function of median node lifetime for the same set of experiments as in Figure ??(a).

Figure 7-4: The latency and bandwidth consumption of Accordion, Chord and OneHop as a function of median node lifetime, in a 3000-node network with churn intensive workload. Accordion uses a bandwidth budget of 24 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 17 and 23 bytes/node/sec, respectively, with median lifetimes of 3600 sec.

OneHop is also more bursty than Accordion; we observe that the maximum bandwidth burst observed for OneHop is 1200 bytes/node/sec in a 3000-node network, more than 10 times the maximum burst of Accordion. Thus, OneHop's bandwidth consumption varies widely and could at any one time exceed a node's desired bandwidth budget, while Accordion stays closer to its average bandwidth consumption.

7.5 Effect of Churn

Previous sections illustrated Accordion's ability to adapt to different bandwidth budgets and network sizes; this section evaluates its adaptability to different levels of churn.

Figure 7-4 shows the lookup latency and bandwidth overhead of Chord, Accordion and OneHop as a function of median node lifetime. Lower node lifetimes correspond to higher churn. Accordion's bandwidth budget is constant at 24 bytes per second per node. Chord and OneHop uses parameters that achieve the lowest lookup latency while consuming 17 and 23 bytes per second, respectively, for a median node lifetime of one hour. While Accordion maintains fixed bandwidth consumption regardless of churn, both Chord and OneHop's over-

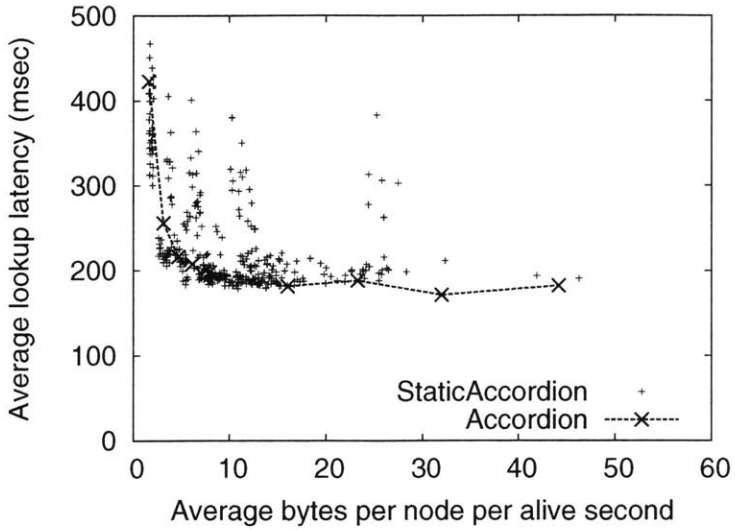


Figure 7-5: Bandwidth versus latency for Accordion and StaticAccordion, using a 1024-node network and a churn intensive workload. Accordion tunes itself nearly as well as the best exhaustive-search parameter choices for StaticAccordion.

head grow inversely proportional to median node lifetime (proportional to churn rates). Accordion's average lookup latency increases with shorter median node lifetimes, as it maintains a smaller table due to higher eviction rates under high churn. Chord's lookup latency increases due to a larger number of lookup timeouts, because of its fixed table stabilization interval. Accordion's lookup latency decreases slightly as the network becomes more stable, with consistently lower latencies than both Chord and OneHop. OneHop has unusually high lookup latencies under high churn as its optimal setting for the event aggregation interval with mean node lifetimes of 1 hour is not ideal under higher churn, and as a result lookups incur frequent timeouts due to stale routing table entries.

7.6 Effectiveness of Self-Tuning

Accordion adapts to the current churn and lookup rate by adjusting the lookup parallelism (w_p), in order to stay within its bandwidth budget. To evaluate the quality of the adjustment algorithms, we compare Accordion with a simplified version (called StaticAccordion) that uses manually adjustable w_p, p_{thresh} and active exploration interval parameters. Simulating StaticAccordion with a range of parameters, and looking for the best latency vs. bandwidth tradeoffs, indicates how well Accordion could perform with ideal parameter settings. Table 7.1 summarizes StaticAccordion's parameters and the ranges explored.

Figure 7-5 plots the latency vs. bandwidth tradeoffs of StaticAccordion for

Parameter	Range
Exploration interval	2-90 sec
Lookup parallelism w_p	1,2,4,6
Eviction threshold p_{thresh}	.6 –.99

Table 7.1: StaticAccordion parameters and ranges.

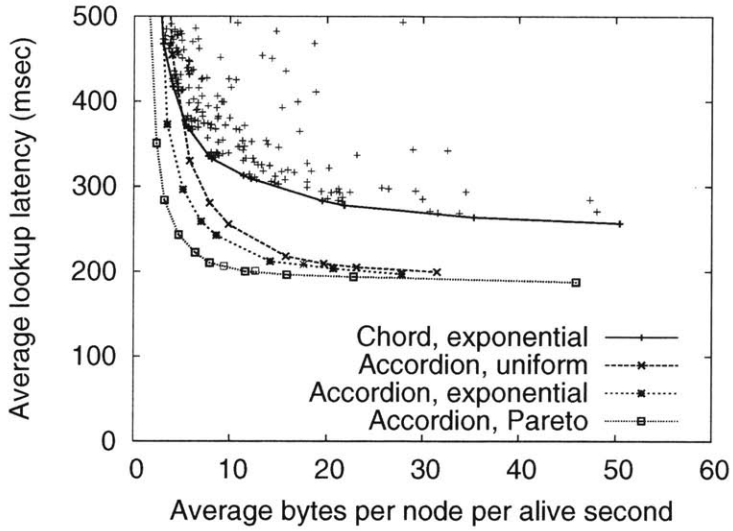


Figure 7-6: The performance of Accordion on three different node lifetime distributions, and of Chord on an exponential distribution, using a 3000-node network and a churn intensive workload. Though Accordion works best with a Pareto distribution, it still outperforms Chord with an exponential node lifetime distribution in most cases.

various parameter combinations. The churn and lookup rates are the same as the scenario in Figure 7-1. The lowest StaticAccordion points, and those farthest to the left, represent the performance Accordion could achieve if it self-tuned its parameters optimally. Accordion approaches the best static tradeoff points, but has higher latencies in general for the same bandwidth consumption. This is because Accordion tries to control bandwidth overhead, such that it does not exceed the maximum-allowed burst size if possible (where we let $b_{burst} = 100r_{avg}$). StaticAccordion, on the other hand, does not attempt to regulate its bandwidth use nor burst size. Its bandwidth consumption is simply the result of using the fixed parameters under the current workload. For example, when the level of lookup parallelism is high, a burst of lookups will generate a large burst of traffic. Accordion will reduce the lookup parallelism w_p to try to stay within the maximum burst size, but StaticAccordion still uses its fixed lookup parallelism. Therefore, StaticAccordion can keep its lookup parallelism constant to achieve lower la-

tencies (by using a bigger routing table with less fresh entries) than Accordion, though the average bandwidth consumption will be the same in both cases. As such, if controlling bursty bandwidth is a goal of the DHT application developer, Accordion will control node bandwidth more consistently than StaticAccordion, without significant additional lookup latency.

7.7 Lifetime Distribution Assumption

Accordion's algorithm for predicting neighbor liveness probability assumes a heavy-tailed Pareto distribution of node lifetimes (see Sections 5.5). In such a distribution, nodes that have been alive a long time are likely to remain alive. Accordion exploits this property by preferring to keep long-lived nodes in the routing table. If the distribution of lifetimes is not what Accordion expects, it may make more mistakes about which nodes to keep, and thus suffer more lookup timeouts. This section evaluates the effect of such mistakes on lookup latency.

Figure 7-6 shows the latency/bandwidth tradeoff with node lifetime distributions that are uniform and exponential. The uniform distribution chooses lifetimes uniformly at random between six minutes and nearly two hours, with an average of one hour. In this distribution, nodes that have been part of the network longer are *more* likely to fail soon. In the exponential distribution, node lifetimes are exponentially distributed with a mean of one hour; the probability of a node being alive does not depend on its join time.

Figure 7-6 shows that Accordion's lookup latencies are higher with uniform and exponential distributions than they are with Pareto. However, Accordion still provides lower lookup latencies than Chord, except when bandwidth is very limited.

7.8 Bandwidth Control

An Accordion node does not have direct control over all of the network traffic it generates and receives, and thus does not always keep within its bandwidth budget. A node must always forward primary lookups, and must acknowledge all exploration packets and lookup requests in order to avoid appearing to be dead. This section evaluates how much Accordion exceeds its budget.

Figure 7-7 plots bandwidth consumed by Accordion as a function of lookup traffic rate, when all Accordion nodes have a bandwidth budget of 6 bytes/sec. The figure shows the median of the per-node averages over the life of the experiment, along with the 10^{th} and 90^{th} percentiles, for both incoming and outgoing traffic. When lookup traffic is low, the median node achieves approximately 6 bytes/sec. As the rate of lookups increases, nodes explore less often and issue fewer parallel lookups. Once the lookup rate exceeds one every 25 seconds there

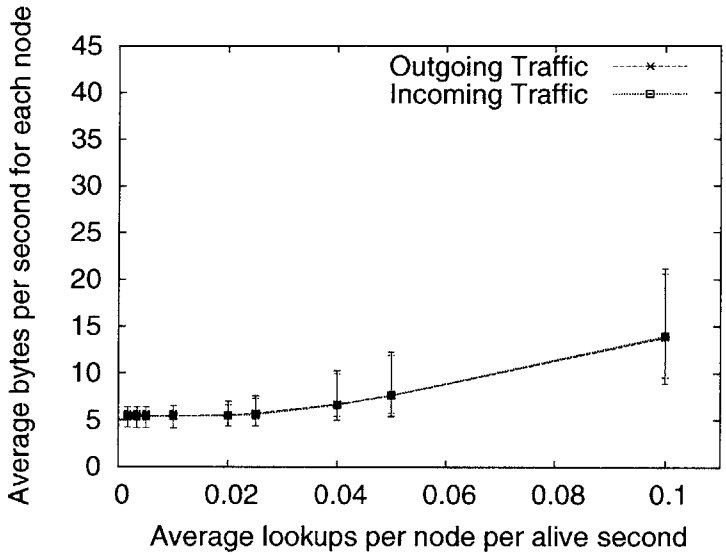


Figure 7-7: Accordion's bandwidth consumption vs. lookup rate, using a 3000-node network and median node lifetimes of one hour. All nodes have a bandwidth budget of 6 bytes/sec. Nodes stay within the budget until the lookup traffic exceeds that budget.

is too much lookup traffic to fit within the bandwidth budget. Each lookup packet and its acknowledgment cost approximately 50 bytes in our simulator, and our experiments show that at high lookup rates, lookups take nearly 3.6 hops on average (including the direct reply to the query source). Thus, for lookup rates higher than 0.04 lookups per second, we expect lookup traffic to consume more than $50 \cdot 3.6 \cdot 0.04 = 7.2$ bytes per node per second, leading to the observed increase in bandwidth.

The nodes in Figure 7-7 all have the same bandwidth budget. If different nodes have different bandwidth budgets, it might be the case that nodes with large budgets force low-budget nodes to exceed their budgets. Accordion addresses this issue by explicitly biasing lookup and exploration traffic towards neighbors with high budgets. Figure 7-8 shows the relationship between the spread of budgets and the actual incoming and outgoing bandwidth incurred by the lowest- and highest-budget nodes. The node budgets are uniformly spread over the range $[2, x]$ where x is the maximum budget shown on the x-axis of Figure 7-8. Figure 7-8 shows that the bandwidth used by the lowest-budget node grows very slowly with the maximum budget in the system; even when there is a factor of 50 difference between the highest and lowest budgets, the lowest-budget node exceeds its budget only by a factor of 2. The node with the maximum budget stays within its budget on average in all cases.

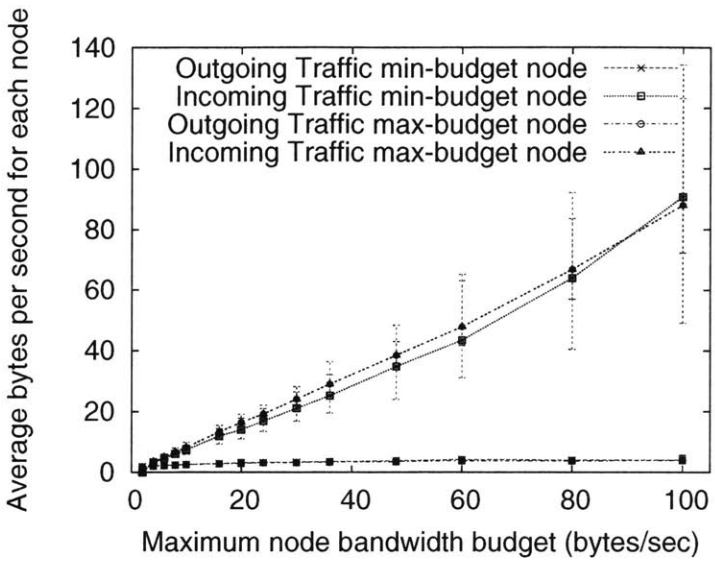


Figure 7-8: Bandwidth consumption of Accordion nodes in a 3000-network using a churn intensive workload where nodes have heterogeneous bandwidth budgets, as a function of the largest node's budget. For each experiment, nodes have budgets uniformly distributed between 2 and the x-value. This figure shows the consumption of the nodes with both the minimum and the maximum budgets.

7.9 Evaluating Accordion Implementation

We evaluate the Accordion implementation in a fully functioning distributed hash table, DHash, using Modelnet [81]. Modelnet is a wide area network emulation toolkit which has also been used by previous studies to evaluate mature implementations of various DHTs [68].

7.9.1 Experimental Setup

Modelnet runs on a cluster of machines on the same LAN. It allows each physical machine to emulate many virtual Internet edge nodes running an instance of *lsd* (the DHash daemon process) with the Accordion lookup protocol. Traffic from all the virtual nodes are forwarded to a single physical machine that acts as the Modelnet "core router" which imposes link capacity constraints, FIFO queuing, propagation delay and loss rate according to a specified network topology.

The Modelnet experimental setup consists of 16 physical machines, one of which is configured as the Modelnet core router that all traffic are routed through. Each of the other 15 machines emulates 68 virtual nodes to create a 1000-node emulated network. Except for the smaller network size (1000 nodes as opposed to 3000 in the simulations), we choose to configure our emulation experiments

similarly to the simulations in the previous sections. The emulated network uses a transit-stub topology generated by the Inet [86] Internet topology generator. The average roundtrip propagation delay between two virtual edge nodes are 164ms. The average bandwidth between a virtual edge node and the ISP stub network is configured to be $500K$bps, a typical DSL link capacity. The capacity between two transit networks is configured to be $150M$bps. In the Accordion implementation, each lookup message and its reply each takes approximately 200 bytes. Therefore, it takes roughly $4*200*8/500 = 12$ ms in transmission delay to traverse the access links between edge nodes and stub networks four times in addition to the 164ms propagation delay to send and receive a lookup in one hop.

All *lsd* processes are started in the first 10 minutes of an experiment. Subsequently, each *lsd* node joins and leaves the network alternatively, with life times generated from a Pareto distribution with median time 3600s ($\alpha = 1.0$ and $\beta = 1800$ s). Each node issues one lookup for a randomly generated key every 600 seconds. All experiments run for 4 hours in real time and statistics are only collected in the second half of the experiments.

In all the graphs shown, the y-axis shows the average lookup latency in milliseconds and the x-axis shows the total number of bytes consumed by all Accordion nodes in the second half of the experiment, divided by the total alive time of all nodes. The bandwidth consumption shown in the graphs includes all lookup traffic, routing table exploration messages and successor list stabilization traffic. In the simulations from the previous section, we count the size of each message to be 20 bytes for the header plus 8 bytes for each node mentioned in the message. In the real implementation, each message header is 170 bytes long and consists of 40 bytes of IP header and 130 bytes of transport headers (containing RPC and the STP [18] transport headers). Furthermore, each node mentioned in the packet takes 36 bytes including information on node's IP address, port number, Vivaldi coordinates [16], bandwidth budget, known uptime (Δt_{alive}), time since last contacted (Δt_{since}) and an unique lookup identifier.

7.9.2 Latency vs. bandwidth tradeoffs

As a baseline comparison, we measure the best possible lookup latency on the Modelnet testbed. We run Accordion with a complete routing table at each node with no churn so each lookup finishes in exactly one hop with no timeouts. The average lookup latency is 220 ms. This is higher than the average RTT it takes for send and receive a 200-byte packets across the network ($164 + 12 = 178$ms). The extra 40 ms is due to a combination of the Modelnet router's processing delay and the relatively high load (load average = 0.49) on each physical machine that each runs 68 *lsd* processes simultaneously. Therefore, 220 ms is the lower bound on the lookup latency in all Modelnet experiments.

Figure 7-9 shows Accordion's lookup latency vs. bandwidth tradeoffs in a

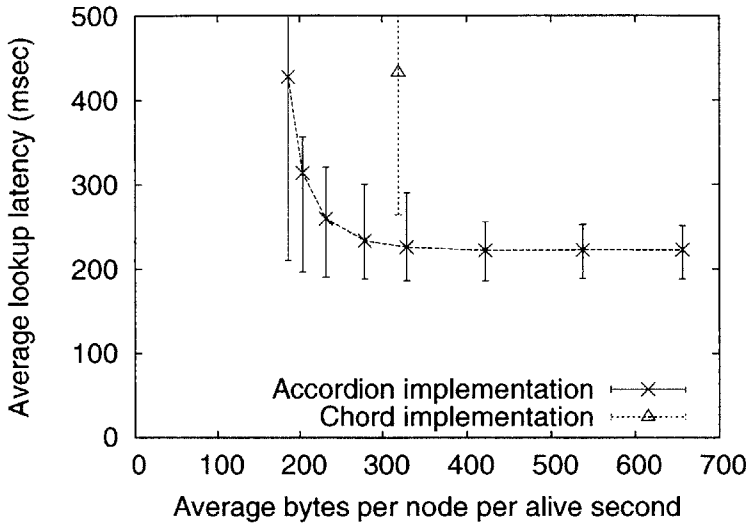


Figure 7-9: Accordion's lookup latency vs. bandwidth tradeoff, in a 1000-node emulated network and a churn intensive workload. The error-bars show the 10-percentile, average and 90-percentile lookup latencies in msec. We use the default parameter settings for evaluating Chord's implementation.

churn intensive workload. Accordion's lookup latency decreases quickly with larger bandwidth consumption. Accordion re-uses Chord's code for successor list stabilization which consumes 170 bytes/node/s in all scenarios and is not subject to Accordion's bandwidth budget constraint. Hence, the minimal Accordion bandwidth consumption approaches 170 bytes/node/s in Figure 7-9.

When the bandwidth budget is small, Accordion slows down its routing state acquisition process with less parallel lookups and active exploration messages. As a result, when consuming 185 bytes/node/s, each node routes lookups with a smaller routing table in 1.96 hops (427.4 ms) and incurring 13.2% timeouts. When the bandwidth budget is plentiful, each Accordion node expands its routing table by learning more quickly through lookups and explorations and using less fresh routing entries with more aggressive parallel lookups. At 328.4 bytes/node/s, Accordion achieves lookup latency as low as 225 ms. Each lookup takes 1.14 hops on average to reach its responsible node and encounters very low timeout probability ($\approx 0.4\%$).

Unlike in p2psim, the Chord implementation in *lsd* does not allow variable base (b) settings. Furthermore, since each Modelnet experiment takes 4 hours, it is impractical to fully explore Chord's parameter space to extrapolate its best latency vs. bandwidth tradeoffs as we did in simulations. We ran *lsd* with Chord using the set of default Chord parameter values. Chord uses $b = 2$ for its finger table. Each node initially checks the liveness of each finger entry every 1 second interval

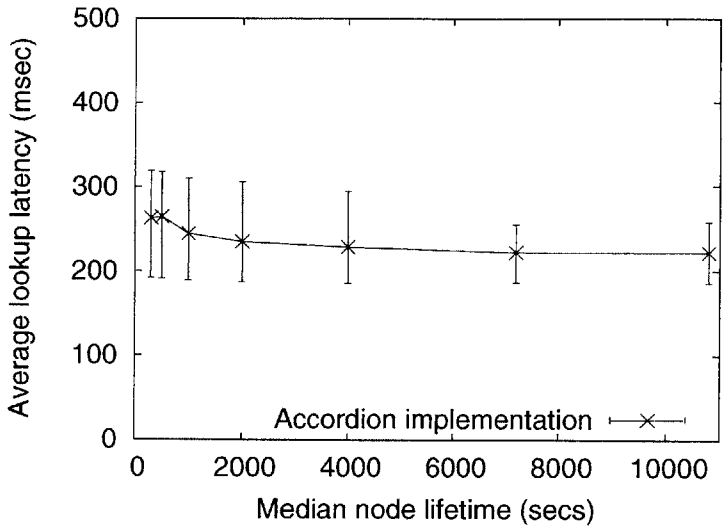


Figure 7-10: Accordion's lookup latency as a function of varying churn rates in a 1000-node emulated network. The x-axis shows the median node lifetime in the experiment. The y-axis shows the 10-percentile, average and 90-percentile lookup latency in msecs.

and doubles the checking interval if it has found all fingers to be alive in the previous period till the maximal checking interval of 16 seconds. With its default parameter setting, each Chord lookup finishes in 431 msec in 3.1 hops on average and incurs 3.1% timeouts. The resulting bandwidth consumption (excluding cost for stabilizing successor lists) is 318.8 bytes/node/s.

7.9.3 Effect of Churn

We measure Accordion's lookup performance under different churn rates in Figure 7-10. We choose a fixed Accordion's bandwidth budget and vary the median node lifetime in each experiment. Each node issues one lookup per 600 seconds. Since the budget remain unchanged, the per-node total bandwidth consumption remains approximately constant at 274 bytes/node/s regardless of churn rates. When churn is high (e.g. median lifetime 500 seconds), an Accordion node contracts its routing table by evicting routing entries more quickly. Hence, nodes route lookups in slightly more hops (1.45) and higher latency (263 msec). When there is low churn, nodes evict entries slower, resulting in bigger routing tables. Hence, when node lifetime is as long as 4000 seconds, lookups have correspondingly lower latency (228 msec) as they finish in fewer hops (1.18).

7.10 Discussions

The performance improvement of the Accordion implementation in DHash with bigger bandwidth budget and larger median node lifetime show the same trend as that in simulations. The absolute bandwidth consumption per node as shown in Figure 7-9 is almost 10 times bigger than that in simulations (Figure 7-1). In simulations, we use the most conservative way to measure each protocol's bandwidth consumption. In particular, each node ID only counts as 4 bytes, the minimal number of bytes required to record a node's IP address. In the actual implementation, each DHT node needs to be identified with much more information than its IP address such as port number, Vivaldi coordinates etc. It is possible to compress the packet payload in the implementation at the cost of increased software complexity, but all protocols will benefit equally from the compression.

The evaluations of Accordion's implementation in DHash shows that Accordion works well as expected. However, our evaluations reveal some issues which are largely ignored by simulations.

DHash uses the STP transport protocol [18] on top of the RPC library to avoid congesting a node's own access link. STP relies on packet losses as an indication of congestion and shrinks its congestion window to slow down sending RPCs. As Accordion aggressively sends out lookup messages to possibly dead neighbors during parallel lookups, the STP congestion window is quickly reduced to its minimal after a sequence of losses. This is because a node has no way to tell if the losses are due to congestion or dead neighbors. In the current implementation, Accordion lookup and exploration messages simply bypass STP's congestion control mechanism. Compared to DHash's data block fetches and stores, Accordion's protocol messages are much smaller and are well bounded by the user specified budget. Therefore, we believe it is unlikely that bypassing STP will result in Accordion congesting a node's access link.

DHT lookup latency is rarely the end-to-end performance metric that applications care about. DHash provides a replicated distributed storage system to applications like UsenetDHT [75] and Overcite [78, 79]. In DHash, a block of data is replicated and stored on a small number of successor nodes that immediate follow the key of the data. The read performance of UsenetDHT and Overcite depends on how fast a data block can be fetched from DHash. A DHT lookup terminates at the immediate predecessor node of the key who returns the identities of its current successors that are storing the data to the lookup originator. Accordion can be used to reduce the lookup latency to approximately one round trip time by forwarding a lookup directly to its predecessor node instead of going through multiple intermediate hops. However, in order to fetch a block of data, the originating node still has to incur another round trip time to issue the fetch RPC to one of the successors in the lookup rely to retrieve the data block. Therefore, even

though Accordion provides one hop lookups, the end-to-end data fetch latency requires two round trip times. We can improve the data fetch performance by allowing nodes to issue data fetches opportunistically before it receives a lookup reply from the key's predecessor node. This requires a node to estimate the total number of nodes in the system so it only issues the opportunistic data fetch when its routing table is nearly complete. One common method for a node to estimate the total number of nodes in the system is to measure the ID segment size occupied by a node's successor list and scale it up to the entire ID ring [57] [56].

Chapter 8

Related Work

There is a large body of related work on designing robust distributed system. The design and applications of DHTs bear resemblance to early work in distributed data structure (LH* [53], DDS [27]) and membership protocols in group communication systems like Horus [82]. One main difference of DHTs from these early work lies in its large scale and wide area deployment. Such a deployment scenario makes precise failure detections impractical. Furthermore, latency and bandwidth [15, 67] optimizations are crucial for good and robust performance over the wide area network. In this chapter, we review related work in DHT evaluations, protocol designs and techniques for handling churn over the wide area network.

8.1 DHT Performance Evaluation

Many ways of evaluating DHTs in static networks exist [29,54]. The most systematic evaluation of DHTs is done by Gummadi et al. [29]. In particular, Gummadi et al. simulate various DHT designs and analyze how different routing geometries affect DHT overlay resilience and proximity routing. They have shown that choosing routing neighbors with low round trip delay (i.e. proximity neighbor selection) can greatly improve the end-to-end lookup latencies of $\log n$ protocols like Chord. They also observe that some routing geometry such as the ring structure of Chord is more resilient to nodes failures than others. The static resilience of the ring structure correlates to our findings in Chapter 4 that Chord can use bandwidth more efficiently to maintain its ring structure for correct lookups. However, the analysis of Gummadi et al. is done in either static networks without churn or networks in which a subset of nodes fail simultaneously before the DHT routing table maintenance procedures have a chance to repair the entries. The analysis does not take into account any overhead traffic and therefore is unable to quantify the *cost* of different DHTs. Our framework, PVC [49, 51], focuses on the performance vs. cost tradeoffs of different DHTs which are affected by design choices

such like parallelizing lookups, varying routing table size and bounding routing entry staleness in addition to a DHT's routing geometry.

Xu [87] studies the tradeoff between the amount of routing state and the resulting overlay network's diameter. The study concludes that existing overlay protocols that maintain $\log(n)$ state have achieved the optimal asymptotic state vs. network diameter tradeoffs. We have not argued against this point; we examined the DHT's lookup latency that was not evaluated by Xu, and found that factors other than the size of the routing state can greatly influence performance.

Liben-Nowell et al. [52] give a theoretical analysis of Chord in a network with churn. The concept of *half-life* is introduced to measure the rate of membership changes. It is shown that $\Omega(\log n)$ stabilization notifications are required per half-life to ensure efficient lookup with $O(\log n)$ hops. The analysis focuses only on the asymptotic communication cost due to Chord's stabilization traffic, whereas our study explores a much broader set of design choices and protocols.

Rhea et al [68] [67] tested three mature implementations of DHTs under churn using Modelnet based emulations: Bamboo [68], FreePastry [24] and an older version of Chord/DHash [17] with iterative lookups. The comparisons focus mostly on lookup consistencies and latency instead of the tradeoffs between a DHT's performance and its bandwidth consumption. Their comparison experiments are done with real implementations using the default protocol parameters. Therefore, unlike PVC, there is no parameter explorations for the DHTs under study. Our study study explores a wider range of protocols and demonstrates that parameter tuning can have a large effect on a protocol's performance vs. cost tradeoffs. On the other hand, we do not model bandwidth congestion in *p2psim* and therefore cannot observe the congestion collapse of FreePastry due to its pro-active recovery in response to churn.

Mahajan et al. studies the reliability vs. maintenance cost tradeoffs of Pastry [55]. Reliability is measured as the probability of forwarding a lookup to a dead neighbor. Reliability affects the lookup latency in the sense that more reliability leads to less timeouts and hence lower lookup latency. However, reliability is not the only factor that determines lookup latency; forwarding a lookup in fewer hops or using proximate neighbors also lead to decrease in overall lookup latency. One of the contribution of PVC is its ability to compare the efficiencies of different design choices each of which uses bandwidth to improve one factor that affects lookup latency. For example, the analysis in [55] provides a self tuning mechanism that achieves a user desired reliability using minimal maintenance cost. In contrast, with PVC we are able to discover that there is one best freshness threshold (i.e. hop-by-hop reliability) and a node should use extra bandwidth to increase its routing state to further reduce lookup hops.

Lam and Liu [46] present join and recovery algorithms for a hypercube-based DHT, and show through experimentation that their protocol gracefully handles both massive changes in network size and various rates of churn. While our work

focuses on lookup latency and correctness, Lam and Liu explore K-consistency, a much stronger notion of network consistency that captures whether or not the network has knowledge of many alternate paths between nodes.

Many existing DHT protocol proposals include performance evaluations [11, 33, 34, 58, 62, 77, 88]. In general these evaluations have focused on lookup hop-count or latency without churn, or the ability to route lookups correctly under a number of simultaneous failures before a DHT's maintenance procedures can repair the stale routing entries. In contrast, PVC helps DHT designers to find out how efficiently different routing table maintenance techniques use extra bandwidth to cope with churn.

8.2 DHT Designs

The first generation of DHT designs focuses on scalable routing geometries that achieve $O(\log n)$ lookup hops with a small routing table with $O(\log n)$ entries. Many different geometries were inspired by the interconnect networks of parallel computing research such as Hypercube (used by CAN [66]), Plaxton tree (used by Pastry [72] and Tapestry [88]), ring with skiplist like fingers (used by Chord [77]).

Subsequent DHT designs have diverged based on different goals. On the one hand, there are designs that explore different tradeoff points between a node's routing table size and lookup hopcount other than $O(\log n)$ state for $O(\log n)$ hops. These include Kelips [34] with $O(\sqrt{n})$ and 2-hop lookups, OneHop [33] with $O(n)$ and 1-hop lookups, Koorde [39] and Viceroy with $O(1)$ and $O(\log n)$-hop lookups etc. However, most of them are evaluated only in simulations with static failures. On the other hand, there are efforts at improving a $O(\log n)$ DHT's performance under churn. One notable example is the design of Bamboo by Rhea et al [68]. They have found that reactive recovery of routing entry failures avoids the danger of congestion collapse associated with pro-active recovery. Calculating timeouts carefully based on virtual coordinates allows a node to re-route a failed lookup via another neighbor quickly. These design lessons turn out to greatly affect a DHT's lookup latency in the actual DHT deployment. In a similar vein, Castro et al. [10] describe how they optimize the Pastry implementation, MSPastry, to route lookups consistently under churn with low bandwidth overhead. Dabek et al [18] evaluate their Chord/DHash implementation in the Planetlab testbed and find that the lookup latency of $\log n$ hops can be reduced dramatically using proximity neighbor selection (PNS) using a small number of samples (also observed by Gummadi et al [29] in simulations). The motivation for Accordion is similar to the second goal, i.e. to understand and improve a DHT's performance and robustness under churn. One main difference of this work with the previous efforts is that Accordion tunes itself with bounded bandwidth overhead to remain robust across different operating scenarios. Therefore, unlike previous work,

Accordion is not tied to one specific routing table size and can reduce its lookup latency more efficiently at additional bandwidth consumption.

Whether the cost of maintaining complete routing state is too overwhelming has been a topic of intense debate. Rodrigues and Blake [70] have argued that for DHTs that provide a distributed persistent data storage, the cost of maintaining data under churn is so high that it is only feasible in networks with very stable node membership. Hence, for these types operating environments, the bandwidth required to maintain complete state is very reasonable. Their analysis does not quantify the effects of lookup timeouts. In practice, timeouts dominate end to end lookup latency, hence they overestimate the benefits brought by a complete but stale routing table. Any argument that nodes *should always* maintain complete routing state assumes a typical deployment scenario with a maximum network size or churn rate. Although we know most deployment scenarios involve small and stable networks, the network growth or the worst case transient churn are ultimately unpredictable. Protocols that always aim to keep complete state degrade badly when the network is overloaded. Accordion does not assume a fixed routing table size and automatically adapts itself under changing conditions to perform low latency lookups while avoid overloading the network.

Accordion [50] shares many aspects of its design with other DHTs or distributed protocols. Specifically, Accordion has borrowed routing table maintenance techniques from a number of DHTs [34, 35, 39, 48, 58, 72, 77], and shares specific goals with MSPastry, EpiChord, Bamboo, and Symphony.

Accordion is not the first DHT that uses a small world distribution to populate a node's routing table. Symphony [57] is a DHT protocol that provides variable routing table size with its small-world distributed routing entries. While Accordion automatically adjusts its table size based on a user-specified bandwidth budget and churn, the size of Symphony's routing table is a protocol parameter. Symphony acquires the desired neighbor entries by explicitly looking up identifiers according to a small-world distribution. Accordion, on the other hand, acquires new entries by learning from existing neighbors during ordinary lookups and active exploration. Evaluations of Symphony [57] do not explicitly account for a node's bandwidth consumption nor the latency penalty due to lookup timeouts. Mercury [7] is another DHT that also employs a small-world distribution for choosing neighbors. Mercury optimizes its tables to handle scalable range queries rather than single key lookups.

Accordion's routing table acquisition process has a flavor of classic epidemic protocols [20]. Accordion learns about routing information indirectly from other nodes, similar in spirit to the way nodes gossip about newly acquired information in an epidemic protocol. There is one major difference between the two; classic gossip algorithms aim to disseminate *all* resources assuming they are equally useful. For example, gossip protocols are also used in distributed systems to detect member failures [83]. The goal of a failure detection service is to allow *all* mem-

bers in the system to learn about *all* failures as quickly as possible.In Accordion, depending on the bandwidth budget or churn rate, nodes may not have complete routing state. Rather, a node tries to learn more routing entries for neighbors closer by in ID space than far away according to a small world distribution. Accordion attaches and updates Δt_{since}, the time since the neighbor was last heard to be alive, for each routing entry a node propagates. Not only does Δt_{since} provides the information required to estimate a routing entry's liveness, it can also be used to distinguish a piece of old information from a new one and hence prevent a dead routing entry from being resurrected, a common technique used by gossip protocols [83]. Routing information can be considered as a non-monotone resource [42] in that it includes both *positive* (a node joins) or *negative* (a node leaves) events. Accordion only propagates positive resources. The resource location protocol shown in [42] uses a spatial gossip algorithm that times out possibly out-of-date resources with timeout bounds same as the information dissemination time which scales with the physical proximity to the resources.

A number of existing DHT designs also use gossip style techniques to acquire routing entries. EpiChord [48] and Kelips [34] also use gossip protocols to populate a node's routing table. EpiChord uses parallel iterative lookups and cache new routing entries from lookups. EpiChord also expires a node's cached entry whose age exceeds some configured limit. Kelips [34] nodes gossip to keep a complete group membership within a node's own group and at least one entry for each of the foreign groups.

There are other DHTs that provide self-tuning. Castro et al. [10] present a version of Pastry, MSPastry, that self-tunes its stabilization period to adapt to churn and achieve low bandwidth. MSPastry also estimates the current failure rate of nodes, using historical failure observations. Accordion shares the MSPastry goal of automatic tuning, but tunes itself with a much wider range of design choices such as routing table size, lookup parallelism to bound its maintenance traffic while providing the best lookup latency under the current churn scenario and workloads.

Kademlia [58] and EpiChord [48] also use parallel lookups to mitigate the effect of lookup timeouts. Both protocols use iterative parallel lookups to control the amount of redundant lookup messages effectively with a user configured parallelism parameter. Accordion prefers recursive lookups in order to allow nodes to preferentially learn routing entries near its own node ID in order to converge to a small world distribution.

There are many other protocols that are concerned with their associated bandwidth cost in a dynamic environment. A number of file-sharing peer-to-peer applications allow the user to specify a maximum bandwidth. Gia [12] exploits that information to explicitly control the bandwidth usage of nodes by using a token-passing scheme to approximate flow control.

8.3 Handling Churn in Other Contexts

This thesis studies the problem of key-based DHT lookups under churn. We use the DHT lookup latency as the performance metric with which a node tries to optimize using a bounded amount of bandwidth. However, depending on the applications that use DHT, the lookup latency might not be the end-to-end performance metric that an application cares about. If an application uses DHT to store data [79] [69] [17] [75] or pointers to data locations [22], the application users are concerned with the latency of the actual data block or data pointer fetches. Dabek et al. [18] have shown that combining the actual data fetch with DHT lookups can reduce the overall data fetch latency.

More importantly than the fetch latency is a DHT's ability to not lose data under churn. The bandwidth required to maintain persistent data storage in a system with churn can be potentially huge. Blake and Rodrigues [9] have modeled the bandwidth cost needed to make new replicas of data when nodes fail and argued that it is not feasible to construct a wide-area DHT-based storage infrastructure using nodes with both limited bandwidth and high churn. TotalRecall [6] and Sostenuto [15] are two systems that use bandwidth efficiently to cope with a common type of churn in which nodes eventually re-join the system after failures.

Chapter 9

Conclusion

Distributed systems should control communication overhead to avoid overloading the network under rapid system growth. Distributed hash tables are widely used by peer-to-peer applications as a building block to locate data. This thesis has shown the design and evaluation of a robust DHT protocol, Accordion, that has bounded bandwidth overhead and best lookup performance across a wide range of environments through self-tuning.

Accordion bounds its bandwidth consumption according to a user-specified bandwidth budget. The key to Accordion's self-tuning is to automatically adjust a node's routing table size. Accordion uses the bandwidth budget to parallelize lookups and acquire new entries from lookups. Bigger budgets lead to bigger routing tables and fewer lookup hops. Accordion estimates routing entries' liveness probabilities based on past node lifetime statistics and evicts possibly dead entries. Larger churn rates lead to faster evictions and hence smaller routing tables that minimize the number of timeouts. Thus, Accordion's final routing table size is the equilibrium between the learning and evicting processes. The evaluations in simulation and experimentation show that Accordion successfully bounds its communication overhead and has matching or better lookup performance than existing manually tuned DHT protocols.

Accordion's bandwidth efficient design comes from insights we have gained by studying existing DHT protocols using the PVC evaluation framework. PVC systematically explores the parameter space of different DHTs to extrapolate the best performance versus cost tradeoffs to provide a fair comparison. PVC also allows DHT designers to evaluate the relative efficiencies of different design choices in their abilities to turn each extra byte of communication into reduced lookup latency.

A self-tuning, bandwidth-efficient protocol such as Accordion has several distinct advantages; First, by controlling and bounding its communication overhead, Accordion avoids overloading the network. Thus, Accordion is robust across different operating environments, especially in scenarios when the system faces rapid

growth or the churn surges. Second, Accordion eliminates the need for manual parameter tuning. Users often do not have the expertise to tune each DHT parameter correctly for a given operating environment. Accordion shifts the burden of tuning from the user to the system by automatically adapting itself to the observed churn and workload to provide the best performance.

9.1 Future Work

We have implemented and evaluated Accordion in the DHash software release. As part of the future work, we plan to deploy Accordion in the context of a specific peer-to-peer application, OverCite [78,79]. OverCite is a cooperative distributed digital library that runs on a few hundred machines distributed over the wide area network to provide a CiteSeer [47]-like digital library service. OverCite uses the DHash software to store and retrieve its meta-data database and research paper repository. It also relies on DHash to retain as accurate and complete information as possible about other nodes in the system. We are in the process of deploying OverCite and hope to gain experiences on the day to day operations of Accordion as a result.

There is a number of improvements that might affect Accordion's performance or usability in the wide area deployment.

Accordion estimates each routing entry's liveness probabilistically, but it uses a calculated threshold value to exclude stale routing entries from being used in lookups. Instead of using a threshold, a more sophisticated approach is to explicitly incorporate the routing entry liveness probability in calculating the "progress" metric during lookups. Let y be the current node to forward a lookup with key k. Node n_i $(i = 1, 2, 3, ...)$ is y's routing entry whose probability of being alive is p_i. One possible way to incorporate p_i in the forwarding decision is for a node to forward the lookup to a neighbor with the biggest v_i'':

$$v_i'' = \frac{v_i'}{p_i}$$

where v_i' is calculated from Equation 6.2. A bigger p_i results in a larger v_i''. Another way to calculate the "progress metric" is to explicitly compute the expected routing delay to the key using each of its neighbors based on p_i, the ID distance of a neighbor to the lookup key, the network delay from y to the neighbor and the neighbor's bandwidth budget.

To work well under the wide area deployment, Accordion has to cope with problems associated with non-transitive network connectivities as described in Chapter 4. We believe that lookup failures due to inconsistent views of key's successors are best solved at the application layer [23]. For applications that require strong data consistency, all successor nodes in a replica set need to run a

distributed consensus protocol (e.g. Paxos [61]) among themselves to ensure an agreement of the replica set's membership. For applications that require only eventual consistency, replica nodes can rely on periodic data repair mechanisms [15] to synchronize data items among each other to reach eventual consistency. Lookup failures due to a broken return path between the predecessor node and the lookup originator should be fixed at the lookup layer. A simple yet effective solution is for the predecessor node to send the lookup reply message via a randomly chosen neighbor to the lookup originator.

Instead of relying on a user-specified bandwidth budget, we would like to include mechanisms that allow Accordion to automatically find out if there is bandwidth available and use it for parallel or active exploration traffic. The biggest challenge here is to distinguish symptoms of network overload from node failures since both situations result in packet losses.

Bibliography

[1] History of the Internet: Internet growth. `http://http://www.funet.fi/index/FUNET/history/internet/en/kasvu.html`, 2005.

[2] p2psim: a packet-level simulator for evaluating peer-to-peer protocols under churn. `http://pdos.lcs.mit.edu/p2psim`, 2005.

[3] Skype: peer-to-peer Internet Telephony Service. `http://www.skype.com`, 2005.

[4] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[5] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating systems support for planetary-scale network services. In *Proceedings of the 1st NSDI*, March 2004.

[6] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, March 2004.

[7] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the 2004 SIGCOMM*, August 2004.

[8] BitTorrent. http://www.bittorrent.com/documentation.html.

[9] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[10] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 DSN*, June 2004.

[11] Miguel Castro, Peter Druschel, Y. C. Hu, and Antony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, June 2002.

[12] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proc. of SIGCOMM*, August 2003.

[13] D. Chiu and R. Jain. Analysis of the increase/decrease algorithms for congestion avoidance in computer networks. In *Journal of Computer Networks and ISDN Vol. 17, No. 1*, 1989.

[14] Landon P. Cox and Brian D. Noble. Pastiche: making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[15] Frank Dabek. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, October 2005.

[16] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the 2004 SIGCOMM*, August 2004.

[17] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[18] Frank Dabek, M. Frans Kaashoek, Jinyang Li, Robert Morris, James Robertson, and Emil Sit. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st NSDI*, March 2004.

[19] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd IPTPS*, 2003.

[20] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, August 1987.

[21] Bryan Ford. Unmanaged internet protocol: Taming the edge network management crisis. In *Proceedings of the Second Workshop on Hot Topics in Networks*, November 2003.

[22] Michael Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, March 2004.

[23] Michael Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and dhts. In *Proceedings of USENIX WORLDS 2005*, 2005.

[24] FreePastry project. `http://freepastry.rice.edu/`, 2004. See wire subdirectory.

[25] Steven Gerding and Jeremy Stribling. Examining the tradeoffs of structured overlays in a dynamic non-transitive network. Class project: `http://pdos.lcs.mit.edu/~strib/doc/networking_fall2003.ps`, December 2003.

[26] Steven Gribble. Robustness in complex systems. In *Proceedings of the 8th workshop on Hot Topics in Operating Systems*, 2001.

[27] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[28] Krishna Gummadi, Richard Dunn, Stefan Saroiu, Steven Gribble, Henry Levy, and John Zahorjan. Measurement, modeling and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 2003 19th ACM Symposium on Operating System Principles*, Bolton Landing, NY, October 2003.

[29] Krishna P. Gummadi, Ramakrishna Gummadi, Steven Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 ACM SIGCOMM*, August 2003.

[30] Krishna P. Gummadi, Harsha Madhyastha, Steven D. Gribble, Henry M. Levy, and David J. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proceedings of the 6th Usenix/ACM Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[31] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop*, November 2002.

[32] P. Krishna Gummadi, Stefan Saroiu, and Steven Gribble. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Multimedia Systems Journal*, 9(2):170–184, August 2003.

[33] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st NSDI*, March 2004.

[34] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd IPTPS*, 2003.

[35] Nicholas Harvey, Michael Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[36] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proceedings of the 14th ACM SPAA*, August 2002.

[37] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st Annual ACM Symposium on Principles of Distributed Computing (PODC).*, July 2002.

[38] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, 1988.

[39] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd IPTPS*, 2003.

[40] B. Kantor and P. Lapsley. Network news transfer protocol. RFC 977, Network Working Group, February 1986.

[41] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th STOC*, May 1997.

[42] David Kempe, Jon M. Kleinberg, and Alan J. Demers. Spatial gossip and resource location protocols. In *ACM Symposium on Theory of Computing*, pages 163–172, 2001.

[43] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd STOC*, 2000.

[44] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, Boston, MA, November 2000.

[45] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. In *Proceedings of ACM SIGCOMM*, 2000.

[46] Simon S. Lam and Huaiyu Liu. Failure recovery for structured P2P networks: Protocol design and performance evaluation. In *Proceedings of the 2004 SIGMETRICS*, June 2004.

[47] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Digital libraries and autonomous citation indexing. *IEEE Computer*, 32(6):67–71, 1999. `http://www.citeseer.org`.

[48] Ben Leong, Barbara Liskov, and Erik D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. In *Proceedings of the 12th International Conference on Networks*, November 2004.

[49] Jinyang Li, Jeremy Stribling, Thomer Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[50] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[51] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the 24th Infocom*, Miami, FL, March 2005.

[52] David Liben-Nowell, Hari Balakrishnan, and David R. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st PODC*, August 2002.

[53] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* — a scalable, distributed data structure. volume 21, pages 480–525, 1996.

[54] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *Proceedings of the 2003 ACM SIGCOMM*, August 2003.

[55] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd IPTPS*, 2 2003.

[56] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable dynamic emulation of the butterfly. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, August 2002.

[57] Gumeet Singh Manku, Mayank Bawa, and Probhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

[58] Peter Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st IPTPS*, March 2002.

[59] David Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 Usenix Technical Conference*, pages 261–274, June 2001.

[60] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. Post: A secure, resilient, cooperative messaging system. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, May 2003.

[61] Athicha Muthitacharoen, Seth Gilbert, and Robert Morris. Etna: a fault-tolerant algorithm for atomic mutable dht data. Technical Report MIT-LCS-TR-993, Massachusetts Institute of Technology, 2005.

[62] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, June 2003.

[63] Vern Paxson. Growth trends in wide-area tcp connections. In *IEEE Network*, August 1994.

[64] Venugopalan Ramasubramanian and Emin Gun Sirer. The design and implementation of a next generation name service for the internet. In *Proceedings of the ACM SIGCOMM*, Portland, Oregon, August 2004.

[65] Stefan Savagen Ranjita Bhagwan and Geoffrey Voelker. Understanding availability. In *Proceedings of the 2003 International Workshop on Peer-to-Peer Sys tems*, February 2003.

[66] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, USA, August 2001.

[67] Sean Rhea. *OpenDHT: A Public DHT Service.* PhD thesis, University of California, Berkeley, August 2005.

[68] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference*, June 2004.

[69] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: A public dht service and its uses. In *Proceedings of ACM SIGCOMM*, August 2005.

[70] Rodrigo Rodrigues and Charles Blake. When multi-hop peer-to-peer routing matters. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[71] Adolfo Rodriguez, Dejan Kostic, and Amin Vahdat. Scalability in adaptive multi-metric overlays. In *Proceedings of the 24th ICDCS*, March 2004.

[72] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[73] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[74] Emil Sit, Josh Cates, and Russ Cox. A DHT-based backup system. In *Proceedings of the 1st IRIS Student Workshop*, Cambridge, MA, August 2003.

[75] Emil Sit, Frank Dabek, and James Robertson. UsenetDHT: A low overhead Usenet server. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[76] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *ACM SIGCOMM*, Pittsburgh, USA, August 2002.

[77] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, pages 149–160, 2002.

[78] Jeremy Stribling. Overcite: A cooperative digital research library. Master's thesis, Massachusetts Institute of Technology, 2005.

[79] Jeremy Stribling, Isaac G. Councill, Jinyang Li, M. Frans Kaashoek, David R. Karger, Robert Morris, and Scott Shenker. OverCite: A cooperative digital research library. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[80] Antti Tapio. Future of telecommunication - internet telephone operator skype. Helsinki University of Technology, HUT T-110.551 Seminar on Internetworking, 2005.

[81] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Fifth Symposium on Operating System Design and Implementation (OSDI 02)*, 2002.

[82] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[83] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP Middleware*, 1998.

[84] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the web from DNS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, March 2004.

[85] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *Proceedings of the 6th Symposium on Operationg Systems Design and Implementation*, December 2004.

[86] Jared Winick and Sugih Jamin. Inet-3.0 internet topology generator. Technical Report UM-CSE-TR-456-02, University of Michigan, 2002.

[87] Jun Xu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. In *Proceedings of the 22nd Infocom*, March 2003.

[88] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[89] Shelley Zhuang, Ion Stoica, and Randy Katz. Exploring tradeoffs in failure detection in routing overlays. Technical Report UCB/CSD-3-1285, UC Berkeley, Computer Science Division, October 2003.