UNIVERSITY OF CALIFORNIA, SAN DIEGO

Exploiting Heterogeneity in Peer-to-Peer Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science

by

Kiran Tati

Committee in charge:

Professor Geoffrey M. Voelker, Chair
Professor Rene L. Cruz
Professor Tara Javidi
Professor Keith Marzullo
Professor Stefan Savage

2006

The dissertation of Kiran Tati is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2006

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank many people without them I am not who I am. First, a big thank you for my parents who encouraged me to join in graduate program in their own way. Kishore, my brother, for his ever positive approach, Kranth, my last brother, for his joyful spirit of never giving up. Swetha, my wife, for all the patience.

I am very grateful to have so many friends who helped me throughout my life. Ashok, who is a phone away to give his ear for my problems. Sreekanth, for his wonderful theories. Venkat Sama, for the field trips. All the kids in my neighborhood at Chityala for all the games especially cricket. Srinivas Vijay and Surya Prakash for wonderful experiences at REC Warangal. Venkat Reddy for introducing me to the joy of programming. Kishore Kottapalli and Sree Ram for badminton games and late night discussions at IITK.

Andrew Chien for offering me a research assistantship at UCSD. Geetanjali and Luis for mentoring me at initial stages in my graduate program. Ranjitha for the fun working on TotalRecall. Priya for evening walks to Mesa. Kashi for the discussions on Indian cricket team performances. Omkar and Hidehetho for Friday night dinners. Marvin, for teaching American ways.

I am very lucky to have Geoffrey M. Voelker as my adviser without him my research work is not complete. I am happy to meet Stefan Savage who taught how to have fun while working. I would like to thank all members of my committee, Professor Geoffrey M. Voelker, Professor Rene L. Cruz Professor Tara Javidi, Professor Keith Marzullo and, Professor Stefan Savage for kindly agreeing to be in the committee and for the feedback during research exam and thesis proposal.

Chapter 3, in full, is a reprint of the material as it appears in the Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04) 2004, by Kiran Tati and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in the Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS) 2006, by Kiran Tati and

Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

VITA

2006	Doctor of Philosophy in Computer Science University of California, San Diego San Diego, CA
1998	Master of Technology in Computer Science & Engineering Indian Institute of Technology Kanpur Kanpur, India
1996	Bachelors of Technology in Computer Science & Engineering Regional Engineering College Warangal, India
2000-2006	Graduate Student Researcher Department of Computer Science and Engineering University of California, San Diego

PUBLICATIONS

Kiran Tati and Geoffrey M. Voelker, "On Object Maintenance in Peer-to-Peer Systems." In *Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS)*, Santa Barbara, California, February 2006.

Kiran Tati and Geoffrey M. Voelker, "ShortCuts: Using Soft State To Improve DHT Routing." In *Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04)*, Beijing, China, October 2004.

Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker, "TotalRecall: System Support for Automated Availability Management." In *Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

FIELDS OF STUDY

Computer Systems

ABSTRACT OF THE DISSERTATION

Exploiting Heterogeneity in Peer-to-Peer Systems

by

Kiran Tati
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Geoffrey M. Voelker, Chair

Peer-to-peer overlay networks provide a distributed, fault-tolerant, scalable architecture on which wide-area distributed systems and applications can be built. Two fundamental services provided by peer-to-peer overlay networks are a routing protocol to map keys in a large, virtual ID space to values associated with individual hosts in the overlay network, and a storage service to maintain data objects on behalf of higher-level services, applications, and users. In existing peer-to-peer routing protocols and object maintenance strategies, the system typically gives each host equal *responsibilities* in terms of routing messages and storing object data and metadata. In reality, however, hosts in the system have unequal *capabilities*.

In this thesis, we propose enhancements to peer-to-peer routing protocols and object maintenance strategies that tailor them to take advantage of resource heterogeneity. Using an approach called "ShortCuts", we use three caching techniques to improve routing performance while adjusting the overhead of maintaining consistent state to the available bandwidth of individual hosts. Combined, these caches achieve routing performance that approaches the aggressive performance of one-hop schemes, but with an order of magnitude less communication overhead on average.

Compared to previous approaches that place data randomly among hosts, we explored refining object maintenance strategies according to host uptimes and lifetimes. As a first step, we performed an extensive measurement study of permanent host failures

in the KAD peer-to-peer file sharing system. Our analysis have shown that, while a significant fraction of the entire population has a very short lifetime (e.g., one connection for a few hours), the network contains a very stable subpopulation of peers with lifetimes of months. As a second step, we propose new object maintenance strategies that bias the placement of redundant data on hosts with longer lifetimes and higher availabilities. Using trace-driven simulation of a peer-to-peer storage system and our trace of hosts in the KAD network, we show that peer-to-peer storage systems can reduce object maintenance overheads by biasing placement on long-lived and highly-available hosts.

Chapter 1

Introduction

This dissertation proposes that peer-to-peer storage systems can take advantage of heterogeneity in host availability and lifetime characteristics to improve performance and reduce overhead. Peer-to-peer overlay networks provide a distributed, fault-tolerant, scalable architecture on which wide-area distributed systems and applications can be built. An increasing trend has been to design Internet-based services on peer-to-peer overlays, including cooperative Web caches [30], Web indexing and searching [29, 41, 45], content delivery systems [87, 20, 36, 9], Usenet news [17], archival storage systems [15, 12, 65, 54, 31, 28], digital research library [75], event notification services [10, 63, 21, 59], version control repository [85], Email service [51, 22] and Domain name service (DNS) [81, 56, 61].

Popular designs of these overlay networks implement a distributed hash table (DHT) interface to higher level software. DHTs map keys in a large, virtual ID space to associated values stored and managed by individual hosts in the overlay network. DHTs divide this functionality into two layers. As described in more detail in Chapter 2, the lower layer implements a lookup operation, and the upper layer implements data storage. For example, the Squirrel Web cache [30] uses the Pastry [70] lookup functionality to locate the host that is responsible for a given HTTP object. The NextGen DNS service [61] uses the Beehive [60] lookup functionality to find the name server that is responsible for a given domain name. Archival storage systems [12, 54, 65] use the

DHT storage subsystem to reliably store file data. Similarly, the email service Post [51] uses the Glacier overlay [28] to reliably store email messages.

The traditional client-server architecture assumes that the server is reliable and always available, whereas clients are unreliable. Peer-to-peer overlays try to provide similarly reliable services, but using only cooperating clients which contribute resources to the system only when the clients are online and connected to the system. As a result, the number of failures in the peer-to-peer environments are an order of magnitude larger than server failures in client-server systems [78, 40]. Hence, a critical challenge for designing DHTs is to provide services with high availability in spite of the large number of failures.

DHTs use a distributed routing protocol to implement the lookup operation. Each host in the overlay network maintains a routing table. When a host receives a request for a particular key, it forwards the request to another host in its routing table that brings the request closer to its destination. The routing table needs to be consistent with the system current state to provide a highly available lookup service. Hence, DHTs continuously update their routing tables to add hosts that arrive in the system and remove hosts that leave. These updates are frequent and consume bandwidth. A natural tradeoff in the design of these routing protocols is the size of the routing table and the latency of routing requests. Larger routing tables can reduce routing latency in terms of the number of hops to reach a destination, but at the cost of additional route maintenance overhead. Because the performance and overhead of DHT overlay networks fundamentally depend upon the distributed routing protocol, significant work has focused on the problem of balancing the degree of routing state and maintenance with route performance.

Similarly, a primary challenge for DHT storage sub systems is to efficiently maintain object availability and reliability in the face of host failures. Hosts in peer-to-peer systems exhibit both temporary and permanent failures, requiring the use of redundancy to mask and cope with such failures (e.g., [84, 1, 38, 73, 5]). The cost of redundancy, however, is additional storage and bandwidth for creating and replenishing redundancy on hosts that leave the system. Since bandwidth is typically a much more

scarce resource than storage in peer-to-peer systems, strategies for efficiently maintaining objects focus on reducing the bandwidth overhead of managing redundancy, trading off storage as a result. Typically, these strategies create redundant versions of object data using either replication or erasure coding as redundancy mechanisms, and either react to host failures immediately or lazily as a repair policy.

In existing peer-to-peer routing protocols and object maintenance strategies, the system typically gives each host *equal responsibilities* in terms of routing messages, storing data and metadata, maintaining system structure, handling client requests, etc. In practice, however, hosts in the system have *unequal capabilities*. Network delay, available bandwidth, host availability, host lifetime, storage capacity, and memory and CPU resources among hosts have distributions that range across orders of magnitude. For instance, in file sharing systems such as OverNet, the availability of hosts vary from few hours a day to an entire day [4]; in the KAD network, host lifetime varies from a few hours to hundreds of days (Chapter 4). Similarly, the upstream and downstream bottleneck bandwidth of hosts in Gnutella vary from 10 Kbps to 100 Mbps [71].

The goal of this thesis is to acknowledge resource heterogeneity among hosts and tailor systems to take advantage of it. We make three contributions: (1) reducing lookup latency using extra state in routing tables, adjusting the overhead of maintaining consistent state to the available bandwidth of individual hosts; (2) understanding the interaction of hosts availabilities, lifetimes, and storage capacities on object maintenance overhead; and (3) reducing object maintenance overhead by allocating objects according to host availability and lifetime.

In Chapter 3, we describe three caching techniques to improve routing performance while adjusting the overhead of maintaining consistent state to the available bandwidth of individual hosts. Local hint caches use large successor lists to short cut final hops. Path hint caches store a moderate number of effective route entries gathered while performing lookups for other hosts. And global hint caches store direct routes to peers distributed across the ID space. Combined, these hint caches achieve routing performance that approaches the aggressive performance of one-hop schemes, but with

an order of magnitude less communication overhead on average.

Next we explored refining previous object maintenance strategies that place data randomly among hosts to place data according to host availabilities and lifetimes. As a first step, Chapter 4 performs an extensive measurement study of permanent host failures in the KAD peer-to-peer file sharing system. Previous studies of peer behavior in such systems have been relatively short in duration (1–2 weeks), and consequently have focused on the temporary churn characteristics of peers. However, the long-term overhead of object maintenance in such systems depends more on the permanent failures of peers. To observe such behavior, we have monitored hosts in a region of the KAD identifier space for over six months. Our analysis of peers over such a long time scale have shown that, while a significant fraction of the entire population has a very short lifetime (e.g., one session for a few hours), the network contains a very stable subpopulation of peers with lifetimes of months. This stable subpopulation accounts for most of the uptime of peers in the entire network.

As a second step, Chapter 5 uses insight gained from the KAD trace analysis to refine the object maintenance strategy used in peer-to-peer storage systems like TotalRecall. Cleanly separating how temporary and permanent churn impact the overheads associated with object maintenance, we demonstrate how different environments exhibit different degrees of temporary and permanent churn, and show how churn in different environments affects the tuning of object maintenance strategies. Using redundancy to mask temporary churn has three implications: (1) an object maintenance strategy can determine a sufficient degree of redundancy to completely mask temporary churn; (2) the amount of redundancy required to mask temporary churn is inversely proportional to the fraction of simultaneously available hosts storing object data; and (3) the bandwidth overhead for coping with temporary churn is dominated by object creation, not by repairs. Once a system has a sufficient degree of redundancy to mask temporary churn, permanent churn drives repairs. When the system permanently loses hosts storing redundant object data, the system must eventually repair the redundancy to ensure data reliability. As a result, in these environments tuning repair strategies to deal with

permanent churn will have the greatest impact on minimizing bandwidth overhead.

When a system decides to repair object data, it must decide how much redundancy to restore. The more redundancy a system restores during a repair the longer it can delay the next repair, thereby trading off storage to reduce the frequency of repairs. In terms of bandwidth overhead, though, it is not immediately clear what the best choice is. An object maintenance strategy can either make "smaller" repairs more frequently, or "larger" repairs less frequently. We show that there exists an optimal balance between the amount of redundancy restored at each repair and the frequency of repair, and that a storage system system like TotalRecall can dynamically measure system behavior to determine this optimal balance when making repairs. Interestingly, the amount of redundancy to restore on a repair that minimizes bandwidth overhead depends upon the degree of temporary churn in the system, but not on the degree of permanent churn; the bandwidth overhead certainly scales with the rate of permanent churn, but the rate does not affect the choice of how much redundancy to repair.

Using trace-driven simulation of a peer-to-peer storage system and our trace of hosts in the KAD network, we confirm our analytic results for determining an optimal amount of redundancy to use at each repair. Further, we explore variants of this object maintenance strategy that bias the placement of redundant data on those hosts with high availability, high lifetimes, or both. We then use simulation to show that, using our trace of KAD hosts as input, peer-to-peer storage systems can reduce object maintenance overhead to 58% of a random strategy by placing on high-available hosts, 49% of random by placing on long-lived hosts, and only 33% of random by placing on both highly-available and long-lived hosts.

In practice, using a placement strategy that biases towards hosts with high availability and/or lifetimes will causes the storage on those hosts to reach capacity at a faster rate. Indeed, even in the uniform random policy, hosts with longer uptimes will be selected more often than hosts with shorter uptimes and, as a result, store more blocks over time. Object maintenance strategies will need to explicitly respect host storage capacities when making placement decisions. Respecting capacities has two

consequences: (1) since object maintenance strategies bias towards hosts with higher uptimes and/or high lifetime, data for newer objects gets placed on hosts with lower uptimes and/or lower lifetime; and as a result, (2) repair overhead increases more than linearly as system storage grows towards capacity. We show that the bandwidth overhead of various object maintenance strategies converge towards each other as the system reaches full capacity.

Finally, Chapter 6 summarizes the contributions of this dissertation and outline future work in the area.

Chapter 2

Background and Related Work

This chapter provides background on the operation of DHT routing protocols and summarizes the various protocols that have been proposed for DHTs, as well as the tradeoffs they make in terms of lookup performance and route maintenance overhead. It also provides background on peer-to-peer storage systems and object maintenance strategies, as well as a survey of various approaches to providing highly-available storage in these systems in the face of failures.

2.1 DHT Routing

Distributed hash tables (DHTs) assign IDs to objects as well as hosts from the same ID space; these ID typically range in size from 128 bits (e.g., Kademlia [48] and the KAD network) to 168 bits (e.g., Chord). Each online host is the "owner" for the IDs for the range between its ID to the next host ID, and is responsible for handling requests addressed to IDs in that range. The lookup operation takes an ID as input and returns the ID of the owner. In the following section we describe the general strategy used to implement the DHT lookup operation as exemplified, for example, by Chord.

Every host in the DHT maintains addresses of other hosts in the system and these entries are referred to as routing entries and, collectively, as routing state or the routing table.

Whenever a host receives a lookup operation for an ID x, it finds the closest

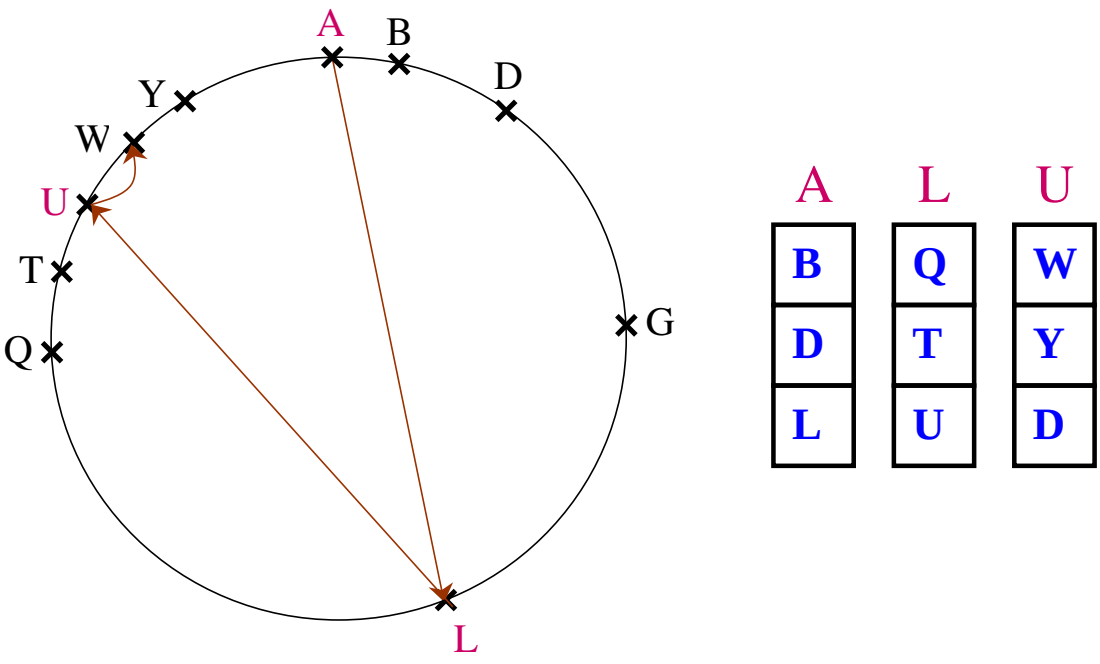


Figure 2.1: DHT lookup operation.

host in its routing table to x and forwards the lookup request to that host. This process continues until the request reaches the owner host. Figure 2.1 is an example DHT with hosts $A, B, D, G, L, Q, T, U, W$, and Y; we are using simple alphabets instead of 128 bit IDs for illustration purposes. This figure also shows the routing tables for hosts A, L, and U. When a host A receives a lookup operation for ID X, it first looks into its routing table and forwards the lookup operation to host L since L is the closest host ID to X in its table. The host L performs the same algorithm, forwarding the lookup operation to host U which, in turn, forwards to the owner W.

The number of hops the lookup operation requires depends on the amount of state each host maintains in its routing table. In the simple case, if each host maintain just its successor or predecessor in the ID space, then the number of hops a lookup requires is proportional to the number of hosts in the system. On the other extreme, if each host maintains full information about all other hosts in the system, then a lookup requires only one hop. As we mentioned earlier, more state information requires more bandwidth to maintain due to host churn in the system (host arrivals or departures). Ideally we

would like to reduce the number of hops as much as possible while maintaining as little state as possible. In the following section we describe some of the DHTs described in the literature and how they improve lookup performance.

2.2 DHT Routing Survey

A number of algorithms seek to minimize route maintenance overhead by using only constant-size $O(1)$ routing tables per host. A set of *randomized* algorithms have $O(1)$ degree and achieve scalable routing performance on average. Viceroy [46] organizes peers into a ring, and uses $O(\log n)$ additional concentric rings that roughly correspond to layers in Butterfly networks. Peers belong to one of these additional concentric rings, and maintain $O(1)$ routes to its neighboring peers in the primary ring, its concentric ring, two peers in a lower ring, and one peer in a higher ring. On average, Viceroy routes requests in $O(\log n)$ hops up and down the rings.

Symphony [47] is another randomized algorithm that builds upon the work of Kleinberg [35] that models the Small World phenomenon. In Symphony, peers maintain routes to their immediate neighbors, known as *short links*, as well as k *long-distance links* to other peers in the system. Since k is a small fixed parameter in the system, each peer has $O(1)$ degree. Each host chooses its long-distance links randomly from a well-chosen probability distribution so that, on average, the system routes requests in $O(\log \log n)$ hops.

For *deterministic* routing algorithms, Kaashoek and Karger [32] prove that, for a constant-size k routing table per peer, $\theta(\log n)$ hops is optimal. They then present a tunable routing algorithm Koorde that combines features of Chord and de Bruijn graphs [8]. Koorde is a tunable algorithm. It can be configured such that peers have only degree 2 and route optimally in $O(\log n)$ hops. In addition, to provide fault-tolerance, Koorde peers can have $O(\log n)$ degree and route in an optimal $O(\log n / \log \log n)$ number of hops.

Several efforts have been made to achieve constant-time $O(1)$ hops to route

requests at the cost of high-degree routing tables. Kelips [27] divides all peers into *affinity groups*. Every peer maintains a route to every other peer in its affinity group, plus a route to at least one peer in every other affinity group. Kelips uses a gossip protocol to propagate route updates among hosts to maintain consistency. Requests from one peer to another in the same group require only a single hop, and requests to peers in another group require two hops. Kelips partitions peers into $O(\sqrt{n})$ affinity groups with $O(\sqrt{n})$ peers per group. As a result, each peer has degree $O(\sqrt{n})$.

Mizrak et al. [52] introduced a hierarchical routing algorithm for DHTs. They observe that host resources in a peer-to-peer overlay network are heterogeneous in terms of capacity, and propose that high-capacity peers serve as *structured superpeers* and handle additional request load in the system. Each superpeer manages a subset of all peers in the system, and all superpeers have a complete mapping of the ID space to other superpeers to achieve one hop routing among superpeers. To route, a peer forwards its request to its superpeer, which can either route it directly to another peer it manages or indirectly through one other superpeer. Each peer has degree $O(1)$. With $O(\sqrt{n})$ superpeers, each superpeer has degree $O(\sqrt{n})$.

Gupta et al. [26] propose a design in which each host maintains a local copy of the complete routing table with degree n. With a complete table, any host can route a request in a single hop. The cost of this approach is the overhead in keeping the routing tables on all hosts consistent. To reduce this overhead, they propose a hierarchical update protocol that enables the system to scale to 10^6 hosts with tolerable update traffic overhead. Peers are divided into *slices*, and each slice is further subdivided into *units*. Every slice and unit has a leader. Peers send updates to their slice and all unit leaders of their slice. Slice leaders both aggregate and pace update logs to all other slices in the system, which in turn propagate the updates to their unit leaders and finally to their peers.

Chord/DHash++ [13] exploits the fact that lookups for replicated values only need to reach a peer near the owner of the key associated with the value (since the peer will have a replica). Although this is appropriate for locating any one of a number

of replicas, many applications require exact lookup. Beehive exploits the power-law property of lookup traffic distributions [60] to achieve constant-time lookups. However, a large class of applications induce different types of lookup traffic. Li et al. [42] exploits heterogeneity in the hosts availability to reduce the state maintenances bandwidth.

Roussopolis and Baker introduce the Controlled Update Protocol (CUP) for managing path caches on peers [69]. CUP uses a query and update protocol to keep path caches consistent with peer arrivals and departures. CUP was implemented in the context of the CAN overlay network, and evaluated relative to straightforward path caches with expiration. Although the CUP path caches are analogous to the path hint caches in our work, our work differs in a number of ways with the CUP approach. Rather than providing an update mechanism to keep caches consistent, we instead combine the use of local hint caches with path and global hint caches to improve performance and tolerate inconsistency. We also evaluate hint caches with a baseline DHT routing algorithm that routes in $O(\log n)$ hops (rather than a range of coordinate dimensions).

2.3 Peer-to-Peer Storage Systems

Peer-to-peer (P2P) storage systems use some form of redundancy (either simple replication or encoding) and continuous replenishment (repair) to cope with failures. We describe two such systems, DHash [12] and TotalRecall [5], in detail and we then summarize other P2P storage systems in the following section.

DHash and TotalRecall are both based on the Chord [73] DHT. Chord arranges all hosts in the system into a circular ID space with each host having a successor. A successor of a host x is online host whose ID is the successor to x in the ID space (except for the highest ID host, whose successor is the lowest ID host). DHash uses simple replication and stores blocks on the owner of the block ID (also known as the master) as well as on four successors of the owner. Finding an object is straightforward in this system. Performing a lookup on the object ID using the Chord DHT will return the owner. DHash also eagerly copies stored data onto new hosts if the successor list

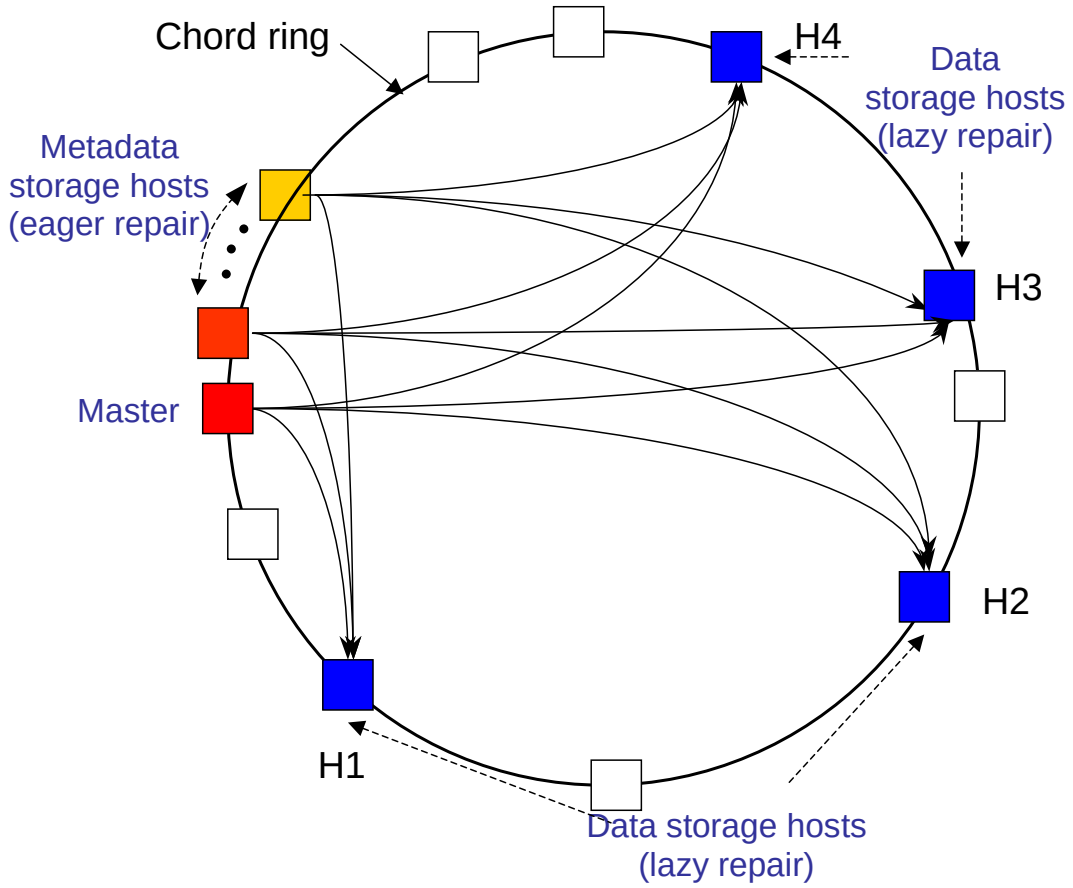


Figure 2.2: P2P Storage System Data Layout.

changes, either due to a new host arriving into the system or a host leaving the system whose ID is in the successor list range. We refer to this strategy of maintaining object availability as *eager repair*.

TotalRecall uses rateless coding to encode object data to reduce storage overhead while achieving high availability. It first encodes the object and then divides the object into n (usually 32) fragments, and stores these fragments onto randomly chosen online hosts. It also stores a metadata object with information about hosts on which it stored the fragments; this metadata uses the DHash eager repair mechanism as explained above for availability. To find an object, the system first gets the metadata object of the object using DHash, and then it gets enough fragments to decode the object. The owner/master of the metadata object continuously monitors the object fragments for their availability. Whenever the number of fragments falls below a specified threshold (typically 16 fragments), TotalRecall creates additional fragments to increase the number of available fragments.

Figure 2.2 shows an example storage layout using TotalRecall. The Master host in the Figure stores the metadata object and replicates the metadata object on the successors; again, these fragments are maintained using the DHash eager repair mechanism. The encoded object is divided into the four fragments ($H1 \cdots H4$) and stored on random hosts in the system. The metadata object stores pointers to the hosts on which the fragments have been placed so that a lookup on the metadata returns the information for reaching these fragments.

2.4 Peer-to-Peer Storage Survey

FarSite [1, 19] is a distributed storage system for corporate environments where the amount of churn is less than churn in the file sharing environments. It uses simple replication, and the number of replicas is determined by the mean time to permanent failure. It does not actively monitor and does not replenish the replicas because permanent failures happen at a very slow rate in these corporate environment.

PASIS [84, 57] is another survivable storage system and uses encoding to provide high availability at a reduced storage cost. They studied the tradeoffs between security, availability, and performance. It also does not have an active storage replenishment mechanism, which is essential to the peer-to-peer environment.

Rowstron proposed a P2P storage system, Past [15, 16], at the same time as DHash. All the files in Past are immutable and the system stores k copies near the owner; k is calculated depending on the expected number of failures. Past does not replace lost copies, though it aggressively caches the replicas onto other hosts to reduce the access time.

Glacier [28] is another distributed storage system aimed at providing high availability even in the presence of correlated failures. It uses abundant storage to protect against correlated failures. It encodes the data using erasure coding. It operates in a corporate or university environment and stores fragments in a fixed position based on the number of hosts in the system. It actively replenishes the lost fragments continuously, and it moves fragments to the proper host as new hosts join or dead hosts leave the system.

OceanStore [38, 65, 55] is a distributed storage system that takes advantage of continuous reduction in storage costs (dollar/byte). The environment consists of stable core servers augmented with home users connected to the Internet with huge disks (similar to the hosts in peer-to-peer environments). It uses erasure codes to reduce the storage costs while providing high availability as well as simple block copies to provide fast access time by eliminating the decoding cost. It has a mechanism to add new hosts to replace old failed hosts. However, it does not specify the policy about when to replace the failed hosts and how often to do it.

Finally, the load balancing mechanisms [62, 34, 24, 25, 39] proposed for DHTs to address the load imbalances due to consistent hashing are similar in spirit to our work since they focus on mechanisms to exploit heterogeneity. We on the other hand are focused identifying which characteristics we should choose to balance and how these different characteristics interact with each other.

Chapter 3

ShortCuts: Using Soft State To Improve DHT Routing

Peer-to-peer overlay networks provide a distributed, fault-tolerant, scalable architecture on which wide-area distributed systems and applications can be built. An increasing trend has been to propose content delivery services on peer-to-peer networks, including cooperative Web caches [30], Web indexing and searching [41, 45], content delivery systems [36, 9], and Usenet news [17]. Popular designs of these overlay networks implement a distributed hash table (DHT) interface to higher level software. DHTs map keys in a large, virtual ID space to associated values stored and managed by individual hosts in the overlay network. DHTs use a distributed routing protocol to implement this mapping. Each host in the overlay network maintains a routing table. When a host receives a request for a particular key, it forwards the request to another host in its routing table that brings the request closer to its destination.

A natural trade off in the design of these routing protocols is the size of the routing table and the latency of routing requests. Larger routing tables can reduce routing latency in terms of the number of hops to reach a destination, but at the cost of additional route maintenance overhead. Because the performance and overhead of DHT overlay networks fundamentally depend upon the distributed routing protocol, significant work has focused on the problem of balancing the degree of routing state and

maintenance with route performance.

Early systems like Chord [73], Pastry [70], Tapestry [86], and CAN [64] use routing tables of degree $O(\log n)$ to route requests in $O(\log n))$ hops, where n is the number of hosts in the network. Newer algorithms improve the theoretical bounds on routing state and hops. Randomized algorithms like Viceroy [46] and Symphony [47] achieve small, constant-degree routing tables to route requests on average in $O(\log n)$ and $O(\log \log n)$ hops, respectively. Koorde [32] is a tunable protocol that can route requests with a latency ranging from $O(\log n)$ to $O(\log n / \log \log n)$ hops for routing tables of constant size to $O(\log n))$ size, respectively. Other approaches, such as Kelips [27], Structured Superpeers [52], Beehive [60], and CUP [69] focus on achieving constant-time $O(1)$ hops to route requests at the expense of high degree routing tables, hierarchical routing, tailoring to traffic distributions, or aggressive update protocols to maintain consistency among the large routing tables in each peer.

In this chapter, we argue that the appropriate use of cached routing state within the routing protocol can provide competitive improvements in performance while using a simple baseline routing algorithm. We describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to successors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

These hint caches require state similar to previous approaches that route requests in constant-time hops, but they do not require the complexity and communication overhead of a distributed update mechanism to maintain consistency among cached routes. Instead, the hint caches do not explicitly maintain consistency in response to peer arrivals and departures other than as straightforward extensions of the standard operations of the overlay network. We show that hint cache inconsistency does not degrade their performance benefits.

We evaluate the use of these hint caches by simulating the latest version of the

Chord DHT [13] and extending it to use the three hint caches. We evaluate the effectiveness of the hint caches under a variety of conditions, including highly volatile peer turnover rates and relatively large network sizes. Based upon our simulation results, we find that the combination of the hint caches significantly improves Chord routing performance. In networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like "OneHop" that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

The remainder of the chapter is organized as follows. Section 3.1 describes how we extend Chord to use the local, path, and global hint caches. Section 3.2 describes the use of detour routes to take advantage of asymmetries in Internet routing to reduce lookup latency further. Section 3.3 describes our simulation methodology for evaluating the hint caches, and presents the results of our evaluations. Finally, Section 3.4 summarizes our results and concludes.

3.1 Design

Distributed hash tables (DHT) increasingly serve as the foundation for a wide range of content delivery systems and applications. The DHT lookup operation is the fundamental operation on which applications base their communication. As a result, the performance of these applications directly depends on the performance of the lookup operation, and improving lookup performance improves performance for all applications layered on DHTs.

The primary goal of our work is to reduce lookup performance as close to direct routing with much less overhead than previous approaches and without relying upon specific traffic patterns. We also integrate the cache update mechanism to refresh cached route entries into the routing protocol to minimize the update complexity as well as overhead. To achieve this goal, each peer employs three *hint caches*. *Local hint caches* store direct routes to neighbors in the ID space. *Path hint caches* store direct

routes to peers accumulated during the natural processing of lookup requests. Finally, *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space. We call them hint caches since the cached routes are hints that may potentially be stale or inconsistent. We also consider them soft-state hints since they can be reconstructed quickly at any time and they are not necessary for the correctness of the routing algorithm.

The following sections describe the behavior of each of the three hint caches. Although these caches are applicable to DHTs in general, we describe them in the context of integrating them into the Chord DHT as a concrete example. So we start with a brief overview of the Chord lookup operation and routing algorithm as background.

3.1.1 The Chord DHT

In Chord, all peers in the overlay form a circular linked list. Each peer has one successor and one predecessor. Each peer also maintains $O(\log n)$ successors and $O(\log n)$ additional peers called *fingers*. The owner of a key is defined as a peer for which the key is in between the peer's predecessor's ID and its ID. The lookup operation for a given key returns the owner peer by successively traversing the overlay. Peers construct their finger tables such that the lookup operation traverses progressively closer to the owner in each step. In recursive lookup, the initiator peer uses its routing table to contact the closest peer to the key. This closest peer then recursively forwards the lookup request using its routing table. Included in the request is the IP address of the initiating peer. When the request reaches the peer that owns the key, that peer responds directly to the initiator. This lookup operation contacts $O(\log n)$ application level intermediate peers to reach the owner for a given key.

We augment Chord with the three hint caches. Chord uses these hint caches as simple extensions to its original routing table. When determining the next best hop to forward a request, Chord considers the entries in its original finger table as well as all entries in the hint caches.

3.1.2 Local Hint Caches

Local hints are direct routes to neighbors in the ID space. They are extensions of successor lists in Chord and leaf hosts in Pastry, except that their purpose is to improve routing performance. With a cache of local hints, a peer can directly reach a small fraction of peers directly and peers can short cut the final hops of request routing.

Local hints are straightforward to implement in a system like Chord using its successor lists. Normally, each peer maintains a small list of its successors to support fault-tolerance within Chord and upper layer applications. Peers request successor lists when they join the network. As part of a process called *stabilization* in Chord, each peer also periodically pings its successor to determine liveness and to receive updates of new successors in its list. This stabilization process is fundamental for maintaining lookup routing correctness, and most DHT designs perform similar processes to maintain successor liveness.

We propose enlarging these lists significantly — on the order of a thousand entries — to become local hint caches. Growing the successor lists does not introduce any additional updates, but it does consume additional bandwidth. The additional bandwidth required is $\frac{S}{H}$ entries per second where S is the number of entries in local hint cache, and H is the half life time of peers in the system. Each peer change, either joining or leaving, requires two entries to update. Similar to [44], we define the half life as the time in seconds for half of the peers in the system to either leave or join the DHT. For perspective, a study of the Overnet peer-to-peer file sharing system measured a half life of four hours [4].

The overhead of maintaining the local hint cache is quite small. For example, when S is 1000 entries and H is four hours, then each peer will receive 0.07 extra entries per second during stabilization. Since entries are relatively small (e.g., 64 bytes), this corresponds to only a couple of bytes/sec of overhead.

Local hint caches can be inconsistent due to peer arrivals and departures. When a peer fails or a new peer joins, for example, its immediate predecessor will detect the failure or join event during stabilization. It will then update its successor list,

and start propagating this update backwards along the ring during subsequent rounds of stabilization. Consequently, the further one peer is from one its successors, the longer it takes that peer to learn that the successor has failed or joined.

The average amount of stale data in the local hint cache is $\frac{R*S*(S+1)}{4*H}$, where R is the stabilization period in seconds (typically one second). On average a peer accumulates $\frac{1}{2*H}$ peers per second of stale data. Since a peer updates its x'th successor every $x * R$ seconds, it accumulates $\frac{x*R}{2*H}$ stale entries from its x'th successor. If a peer has S successors, then on average the total amount of stale data is $\sum_{i=1}^{S} \frac{i*R}{2*H}$. If the system half life time H is four hours and the local hint cache size is 1000 peers, then each peer only has 1.7% stale entries. Of course, a peer can further reduce the stale data by using additional update mechanisms, introducing additional bandwidth and complexity. Given the small impact on routing, we argue that such additions are unnecessary.

3.1.3 Path Hint Caches

The distributed nature of routing lookup requests requires each peer to process the lookup requests of other peers. These lookup requests are generated both by the application layered on top of the DHT as well as the DHT itself to maintain the overlay structure. In the process of handling a lookup request, a given peer can get information about other peers that contact it as well as the peer that initiated the lookup.

With Path Caching with Expiration (PCX) [69], peers cache path entries when handling lookup requests, expiring them after a time threshold. PCX caches entries to the initiator of the request as well as the result of the lookup, and the initiator caches the results of the lookup. In PCX, a peer stores routes to other peers without considering the latency between itself and these new peers. In many cases, these extra peers are far away in terms latency. Using these cached routes to peers can significantly add to the overall latency of lookups (Figure 3.6(a)). Hence PCX, although it reduces hops (Figure 3.6(b)), can also counter-intuitively increase lookup latency.

Instead, peers should be selective in terms of caching information about routes to other peers learned while handling lookups. We propose a selection criteria based on

the latency to select a peer to cache it. A peer x caches a peer y if the latency to y from x is less than the latency from x to peer z, where (1) z is in x's finger table already and (2) its ID comes immediately before y's ID if x orders the IDs of its finger table peers. For example, assume y falls between a and b in x's finger table and then peer x contacts a to perform the lookup request for an ID between (a, b]. If we insert y, then x would contact y for the ID between (y, b]. Since the latency to y from x is less than the latency a from x, the lookup latency may reduce for IDs between (y, b]. As a result, x will cache the hop to y. We call the cache that collects such hints the path hint cache.

We would like to maintain the path hint cache without the cost of keeping entries consistent. The following cache eviction mechanism tries to achieve this goal. Since a small amount of stale data will not affect lookup performance significantly (Figure 3.4), a peer tries to choose a time period to evict entries in the path hint cache such that amount of stale data in its path cache is small, around 1%. The average time to accumulate d percentage of stale data in the path hint cache is $2 * d * h$ seconds, where h is the halving time [44]. Hence a peer can use this time period as the eviction time period.

Although the improvement provided by path hint caches is somewhat marginal (1–2%), we still use this information since it takes advantage of existing communication and comes free of cost.

3.1.4 Global Hint Caches

The goal of the global hint cache is to approximate two-hop route coverage of the entire overlay network using a set of direct routes to low-latency, or *nearby*, peers. Ideally, entries in the global hint cache provide routes to roughly equally distributed points in the ID space; for Chord, these nearby routes are to peers roughly equally distributed around the ring.

These nearby peers work particularly well in combination with the local hint caches at peers. When routing a request, a peer can forward a lookup to one of its global cache entries whose local hint cache has a direct route to the destination. With

a local hint cache with 1000 entries, a global hint cache with a few thousand hosts will approximately cover an entire system of few million peers in two hops.

A peer populates its global hint cache by collecting route entries to low-latency hosts by walking the ID space. A peer x contacts a peer y from its routing table to request a peer z from y's local hint cache. The peer x can repeat this process from z until it reaches one of its local hint cache peers. We call this process *space walking*.

While choosing peer z, we have three requirements: minimizing the latency from x, minimizing x's global hint cache size, and preventing gaps in coverage due to new peer arrivals. Hence, we would like to have a large set of peers to choose from to find the closest peer, to choose the farthest peer in the y's local hint cache to minimize the global hint cache size, and to choose the closer peer in y's local hint cache to prevent gaps. To balance these three requirements, when doing a space walk to fill the global hint cache we use the second half of the successor peers in the local hint cache.

Each peer uses the following algorithm to maintain the global hint cache. Each peer maintains an index pointer into the global hint cache called the *refresh pointer*. Initially, the refresh pointer points to the first entry in the global hint cache. The peer then periodically walks through the cache and examines cache entries for staleness. The peer only refreshes a cache entry if the entry has not been used in the previous half life time period. The rate at which the peer examines entries in the global hint cache is $\frac{g}{2*d*h}$, where d is targeted percentage of stale data in the global hint cache, g is the global hint cache size, and h is the halving time. This formula is based on the formula for stale data in the path hint cache (Section 3.1.3).

The value d is a system configuration parameter, and peers can estimate h based upon peer leave events in the local hint cache. For example, if the halving time h is four hours, the global hint cache size g is 1000, and the maximum staleness d is 0.125%, then the refresh time period is 3.6 seconds. Note that if a peer uses an entry in the global hint cache to perform a lookup, it implicitly refreshes it as well and consequently reduces the overhead of maintaining the hint cache.

Scaling the system to a very large number of hosts, such as two million peers,

the global hint cache would have around 4000 entries and peers would require one ping message per second to maintain 0.5% stale data in very high churn situations like one-hour halving times. Such overheads are small, even in large networks.

Peers continually maintain the local and path hint caches after they join the DHT. In contrast, a peer will only start space walking to populate its global hint cache if it receives a threshold explicit lookup requests directly from the application layer (as opposed to routing requests from other peers). The global hint cache is only useful for the lookups made by the peer itself. Hence, it is unnecessary to maintain this cache for a peer that is not making any lookup requests. Since a peer can build this cache very quickly (Figure 3.9), it benefits from this cache soon after it starts making application level lookups. A peer maintains the global hint cache using the above algorithm as long as it receives lookups from applications on the peer.

3.1.5 Discussion

Our goal is to achieve near-minimal request routing performance with significantly less overhead than previous approaches. Local hint caches require $\frac{S}{H}$ entries/sec additional stabilization bandwidth, where S is the number of entries in the local hint cache and H is the half life of the system. Path hint caches require no extra bandwidth since they incorporate information from requests sent to the peer. And, in the worst case, global hint caches require one ping message per $\frac{2*d*h}{g}$ seconds to refresh stale entries.

For comparison, in the "OneHop" approach [26] each peer periodically communicates $\frac{N}{2*H}$ entries to update its routing table, an order of magnitude more overhead. With one million peers at four hour half life time, for example, peers in "OneHop" would need to communicate at least 35 entries per second to maintain the state consistently, whereas the local hint cache requires 0.07 entries per second and one ping per 28 seconds to maintain the global hint cache.

3.2 Shun Pikes

The direct route provided by the Internet between two peers may not be the best route [72]. According to a recent study, 17% of pairs of peers reduce 25 milliseconds in latency if we use alternate paths [37] with some intermediate hops. Our goal is to reduce the DHT lookup latency by exploiting the alternate paths that are better than the direct paths. Each peer tries to forward the lookup request to a peer that is closest to the lookup key from its routing table until the lookup request reaches the owner. We try to use these alternate paths to reach these intermediate hops if alternative paths reduce latency.

The simplest approach to find alternate shortest paths is to run the single source shortest paths algorithm (either Dijkstra or Bellman-Ford) at each peer using the local hint cache and global hint cache as the intermediate hops. The shortest paths from a single host to all other hosts in a complete graph can be approximated closely by the shortest paths that are constructed from a few random subsets of hosts in the graph similar to Internet [79]. In other words, the shortest paths constructed for a single host with some random hosts and edges to these random hosts are almost as good as the shortest paths constructed from the same host to all other hosts in the graph with the complete graph. In our case, the local hint cache has a small set of random subset of peers in the DHT and the global hint cache has the closest peers for a given peer. Hence, the shortest path constructed from a peer x to any other peer y in the DHT using just peers in x's local and global hint caches as intermediate hops is within a constant factor from shortest path from x to y if x uses all other peers in the DHT as intermediate hops to construct the shortest path.

Each peer constructs the shortest paths to all other peers in its caches and tries to uses this information in forwarding the lookup operation. As previously mentioned, a peer x tries to forward the lookup request to the closest peer y in the identifier space. However, if there is a better path from x to y through peer z, then peer x forwards the lookup to peer z, instead of directly forwarding to y, to forward the lookup request to y.

Overall this new lookup algorithm combining with the local and global hint cache performs better than the "OneHop" approach with significantly less overhead. However, "OneHop" could be modified to take advantages of detours to improve the lookup operation and can achieve similar lookup performance.

3.3 Methodology and Results

In this section we describe our DHT simulator and our simulation methodology. We also define our performance metric, average space walk time, to evaluate the benefits of our hint caches on DHT routing performance.

3.3.1 Chord Simulator

Although the caching techniques are applicable to DHTs in general, we chose to implement and evaluate them in the context of Chord [73] due to its relative simplicity. Although the Chord group at MIT makes its simulator available for external use [14], we chose to implement our own Chord simulator together with our hint caching extensions. We implemented a Chord simulator according to the recent design in [13] that optimizes the lookup latency by choosing nearest fingers. It is an event-driven simulator that models network latencies, but assumes infinite bandwidth and no queuing in the network. Since our experiments had small bandwidth requirements, these assumptions have a negligible effect on the simulation results.

We separated the successor list and finger tables to simplify the implementation of the hint caches. During stabilization, each peer periodically pings its successor and predecessor. If it does not receive an acknowledgment to its ping, then it simply removes that peer from it tables. Each peer also periodically requests a successor list update from its immediate successor, and issues lookup requests to keep its finger table consistent. When the lookup reaches the key's owner, the initiating peer chooses as a finger the peer with the lowest latency among the peers in the owner's successor list.

For our experiments, we used a period of one second to ping the successor and

predecessor and a 15 minute time period to refresh the fingers. A finger is refreshed immediately if a peer detects that the finger has left the DHT while performing the lookup operation. These time periods are same as ones used in the Chord implementation [14].

To compare different approaches, we want to evaluate the potential performance of a peer's routing table for a given approach. We do this by defining a new metric called *space walk latency*. The space walk latency for a peer is the average time it takes to perform a lookup to any other peer on the DHT at a given point of time. We define a similar metric, *space walk hops*, in terms of hops rather than latency. The space walk time is a more complete measurement than a few thousands of random sample lookups because space walk time represent lookup times to all peers in the network.

We simulate experiments in three phases: an initial phase, a stabilization phase, and an experiment phase. The initial phase builds the Chord ring of a specified number of hosts, where hosts join the ring at the rate of 50 hosts per second. The stabilization phase settles the Chord ring over 15 minutes and establishes a stable baseline for the Chord routing data structures. The experimental phase simulates the peer request workload and peer arrival and departure patterns for a specified duration. The simulator collects results to evaluate the hint caching techniques only during the experimental phase.

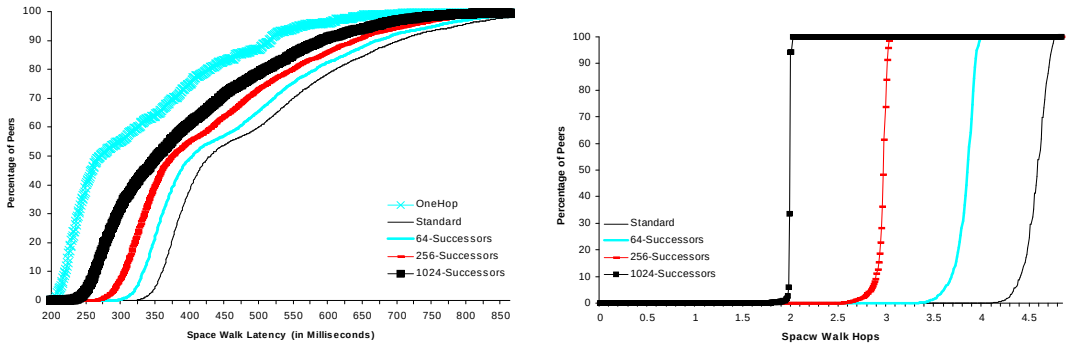
Because membership churn is an important aspect of overlay networks, we study the performance of the hint caches using three different churn scenarios: twenty-four-hour, four-hour, and one-hour half life times. The twenty-four-hour half life time represent the churn in a distributed file system with many stable corporate/university peers [7]. The four-hour half life time represent the churn in a file sharing peer-to-peer network with many home users [49]. And the one-hour half life time represent extremely aggressive worst-case churn scenarios [26].

For the simulations in this chapter, we use an overlay network of 8,192 peers with latency characteristics derived from real measurements. We start with the latencies measured as part of the Vivaldi [11] evaluation using the King [37] measurement technique. This data set has approximately 1,700 DNS servers, but only has complete

all-pair latency information for 468 of the DNS servers. To simulate a larger network, for each one of these 468 DNS servers we create roughly 16 additional peers to represent peers in the same stub networks as the DNS servers. We create these additional peers to form a network of 8,192 peers. We model the latency among hosts within the group as zero to correspond to the minimal latency among hosts in the same network. We model the latency among hosts between groups according to the measured latencies from the Vivaldi data set and we refer this data set as a "King". The minimum, average, and maximum latencies among groups are 2, 165, and 795 milliseconds, respectively. As a timeout value for detecting failed peers, we use a single round trip time to that peer (according to the optimizations in [13]).

Using measurements to create the network model adds realism to the evaluation. At the same time, though, the evaluation only scales to the limits of the measurements. To study the hint caches on systems of much larger scale, we also performed experiments using another network model. We have two different data sets in this model and both of them are significantly larger than above data set. First, we created a matrix of network latency among 8,192 groups by randomly assigning a latency between two groups from the range of 10 to 500 milliseconds and a latency within a group from the range of 1 to 5 milliseconds. We then created an overlay network of 262,144 peers by randomly assigning each peer to one group, keeping the groups balanced. We also created another data set with same latencies among groups and with in a group for 65536 peers that are randomly distributed into 2,048 groups. We refer these data sets as a "Random".

The goal of "Random" data set is to show the performance of various caches at large scale networks that we can simulate on the resources that are available to us. We used two different clusters [50, 80] to scale the computational resources. However, in some of the experiments we couldn't parallelize our simulation easily hence we used smaller "Random" data set with 65,536 peers.



(a) Latency distributions

(b) Hop count distributions

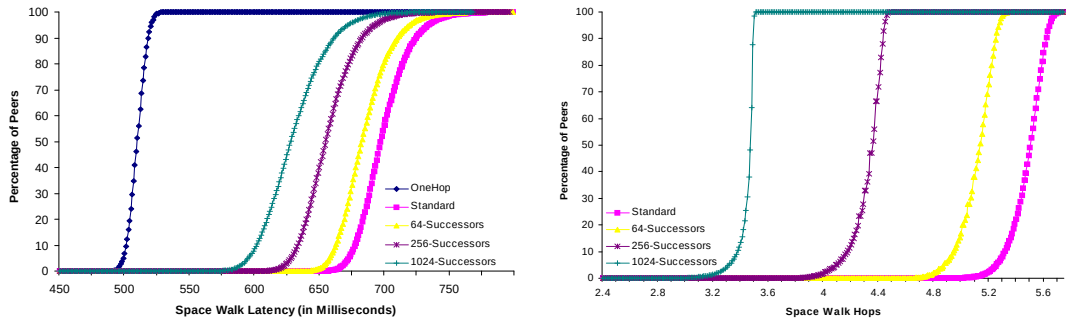
Figure 3.1: Sensitivity of local hint cache size on lookup performance for "King" data set.

3.3.2 Local Hint Caches

In this experiment we evaluate the performance of the local hint cache compared with two baseline routing algorithms, "Standard" and "OneHop." "Standard" is the default routing algorithm in Chord++ [13] that optimized for lookup latency by choosing nearest fingers. "OneHop" maintains complete routing tables on all peers [26].

Figures 3.1(a) and 3.1(b) show the cumulative distributions for the space walk latencies and hops, respectively, across all peers in the system. Since there is no churn in this experiment, we calculate the latencies and hops after the network stabilizes when reaching the experimental phase; we address churn in the next experiment. Figure 3.1(a) shows results for "Standard" and "OneHop" and local hint cache sizes ranging from 64–1024 successors; Figure 3.1(b) omits "OneHop" since it only requires one hop for all lookups with stable routing tables.

Figure 3.1(a) shows that the local hint caches improve routing performance over the Chord baseline, and that doubling the cache size roughly improves space walk latency by a linear amount. The median space walk latency drops from 432 ms in Chord to 355 ms with 1024 successors in the local hint cache (a decrease of 18%). Although an improvement, the local hint cache alone is still substantially slower than "OneHop",



(a) Latency distributions

(b) Hop count distributions

Figure 3.2: Sensitivity of local hint cache size on lookup performance for "Random" data set.

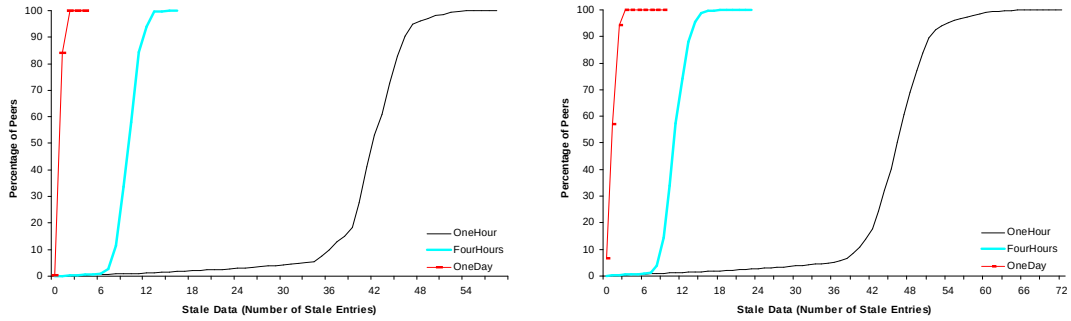
which has a median space walk latency of 273 ms (a decrease of 37% is needed).

Figure 3.1(b) shows similar behavior for local hint caches in terms of hops. A local hint cache with 1024 successors decreases median space walk latency by 2.5 hops, although using such a cache still requires one more hop than "OneHop".

The performance local hint cache for "Random" data set in Figure 3.2(a) and Figure 3.2(b) is similar to the performance of local hint cache for "King" data set in Figure 3.1(a) and Figure 3.1(b) respectively. We used 65,536 peers for the "Random" data set to be consistent with other Figures in this section.

From the graph, we see that doubling the local hint cache size improves the number of hops by at most 0.5. Doubling the local hint cache size reduces hop count by one for half of the peers, and the remaining half does not benefit from the increase. For example, consider a network of 100 peers where each peer maintains 50 other peers in its local hint cache. For each peer, 50 peers are one hop away and the other 50 peers are two hops away. As a result, the space walk hop distance is 1.5 hops. If we increase the local hint cache to 100 peers, then each peer reduces the hop distance for only the 50 peers that were two hops away in the original scenario. In this case, the space walk hop distance is 1.

When there is no churn in the system, the lookup performance when measured



(a) King Data set

(b) Random Data set

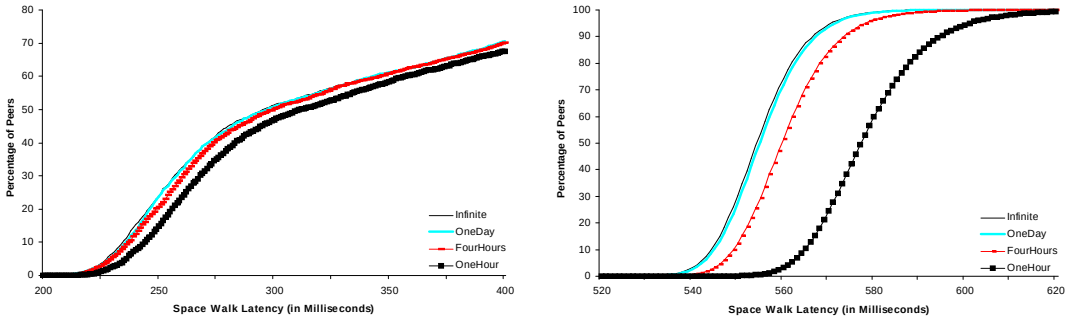
Figure 3.3: Stale data distribution under various churn situations for King and Random data sets.

in terms of hops either remains the same or improves when we double the local hint cache size. The results are more complicated when we measure lookup performance in terms of latency. Most peers improves their lookup latencies to other peers and, on average, increasing local hint cache improves the space walk latency. However, lookup latency to individual peers can increase when we double the local hint cache size in some cases. This is because Internet routing does not necessarily follow the triangular inequality: routing through multiple hops may have lower latency than a direct route between two peers. Since we derive our network latency model from Internet measurements, our latency results reflect this characteristic of Internet routing.

3.3.3 Staleness in Local Hint Caches

The previous experiment measured the benefit of using the local hint cache in a stable network, and we now measure the staleness in terms of stale entries in the local hint cache and the effect of staleness on lookup performance.

In this experiment, we use a local hint cache size of 1024 successors. To calculate stale data in local hint caches, we ran the simulator with King data set and Random data set with 65,536 peers for an experimental phase of 30 minutes. During the experimental phase, the network experiences churn in terms of peer joins and leaves.



(a) King data set

(b) Random data set

Figure 3.4: Space walk latency distributions under various churn situations.

We vary peer churn in the network by varying the half life of peers in the system from one hour to one day; we select hosts to join or leave from a uniform distribution.

Figure 3.3 shows the fraction of stale entries in local hint caches for various system half life times as a cumulative distribution across all peers. We calculated the fraction of stale entries by sampling each peer's local hint cache every second and determining the number of stale entries. We then averaged the samples across the entire simulation run. Each point (x, y) on a curve indicates that y percentage of peers have at most $x\%$ stale data in their local hint caches. As expected, the amount of stale data increases as the churn increases. Note that the amount of stale data is always less than the amount calculated from our analysis in Section 3.1.2 since the analysis conservatively assumes worst case update synchronization.

Figure 3.4 shows the effect of stale local hint cache entries on lookup performance across all peers for King data set and Random data set with 32768 peers. It shows results for the same system half life times as Figure 3.3 and adds results for an "Infinite" half life time. An "Infinite" half life means that there is no churn, no stale entries in the hint cache, and therefore represents the best-case latency. At the end of the experiment phase in the simulation, we used the state of each peer's routing table to calculate the distribution of space walk latencies across all peers. Each point (x, y) in the figure indicates that y percentage of peers have at most x space walk latency. We cut

off the y-axis at 75% of peers to highlight the difference between the various curves.

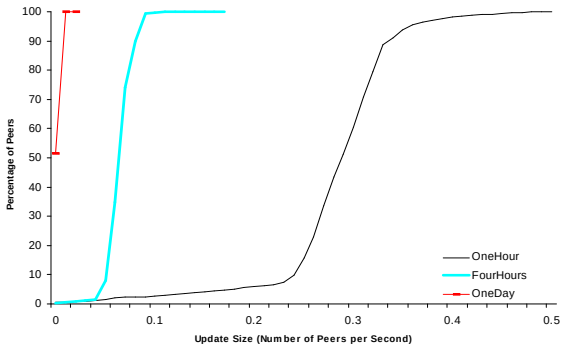
The space walk latencies for a four hour half life time are similar to the latencies from the ideal case with no churn (medians differ by only 1.35%). From these results we conclude that the small amount of stale data (1–2%) does not significantly degrade lookup performance, and that the local hint cache update mechanism maintains fresh entries well. As the churn rate increases, stale data increases and lookup performance also suffers. At an one hour half life, lookup performance increases moderately.

Note that the "Infinite" half life time curves in Figure 3.4(a) and Figure 3.4(b) performs better than the 1024 successors curve in Figure 3.1(a) and Figure 3.2(a) even though one would expect them to be the same. The reason they differ is that the finger table entries in these two cases are different. When we evaluated the local hint cache, we used a routing table with 13 successors and added the remaining successors to create the local hint cache without changing the finger table. When we evaluated the stale data effects in the local hint cache we have 1024 successors from which to choose "nearest" fingers. As a result, the performance is better.

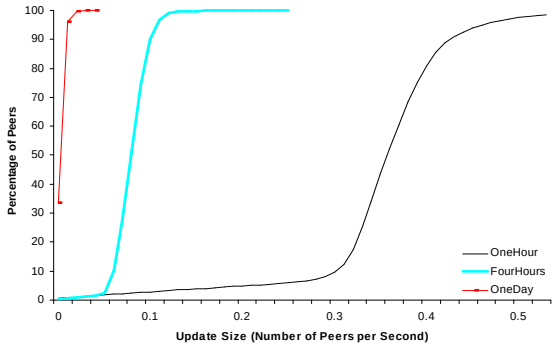
3.3.4 Update Traffic

In the previous section we evaluated the effect of stale data on lookup performance under various churn scenarios. In this section we evaluate the update traffic load under various churn scenarios to evaluate the update traffic bandwidth required by large local hint caches.

We performed a similar experiment as in Section 3.3.3 for King data set and Random data set with 65536 peers. However, instead of measuring stale data entries we measured the update traffic size. We calculated the average of all update samples per second for each peer over its lifetime in terms of the number of entries communicated. Figure 3.5 presents this average for all peers as cumulative distributions. Each curve in Figure 3.5 corresponds to a different churn scenario. A point (x, y) on each curve represent the y percentage of peers that have at most x entries of average update traffic. The average update traffic closely matches the estimate from our analysis in Section3.1.2.



(a) King data set



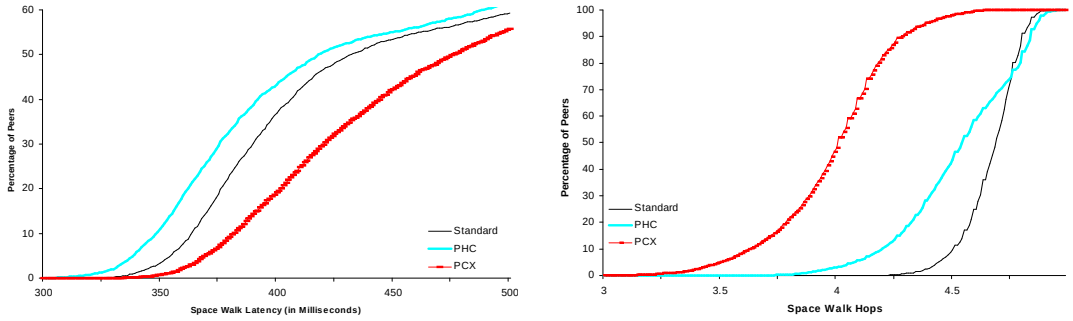
(b) Random data set

Figure 3.5: Update traffic distribution under various churn situations.

The average update traffic (0.4 entries/second) is extremely low even under worst case conditions. Hence, this traffic does not impose a burden on the system.

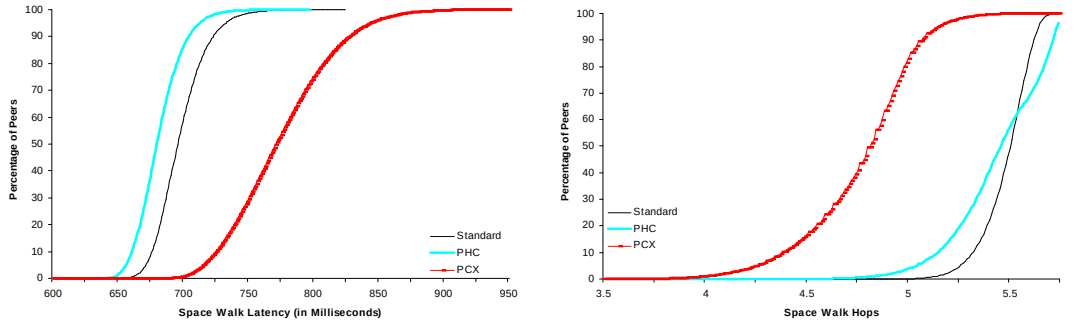
3.3.5 Path Hint Caches

Next we evaluate the performance of the path hint cache (PHC) described in Section 3.1.3 compared to path caching with expiration (PCX) as well as Chord. PCX is the technique of caching path entries described in [69]. When handling lookup requests on behalf of other hosts, PCX caches route entries to the initiator of the request as well as the result of the lookup.



(a) Latency Distribution (b) Hop count distributions

Figure 3.6: Lookup performance of path caching with expiration (PCX), path hint cache (PHC), and standard Chord for "King" data set.



(a) Latency Distribution (b) Hop count distributions

Figure 3.7: Lookup performance of path caching with expiration (PCX), path hint cache (PHC), and standard Chord for "Random" data set.

In this experiment, we simulate a network of "King" data set with a 30-minute experimental phase. We study the lower-bound effect of the path hint caches in that we do not perform any application level lookups. Instead, the path hint caches are only populated by traffic resulting from network stabilization. We did not simulate application level lookup traffic to separate its effects on cache performance; with applications performing lookups, the path hint caches may provide more benefit, although it will likely be application-dependent. Since there is no churn, cache entries never expire. To focus

on the effect of path caches only, we used a local hint cache size of 13, the size of the standard Chord successor list, and no global hint cache. We collected the routing tables for all peers at the end of the simulations and calculated the space walk latencies and hops.

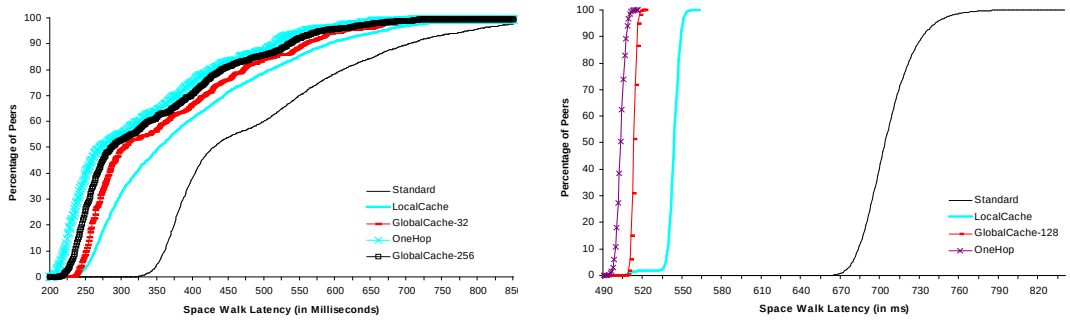
Figure 3.6(a) shows the cumulative distribution of space walk latencies across all peers at the end of the simulation for the various path caches and standard Chord. Each point (x, y) in this figure indicates that y percent peers have at most x space walk latency. From these results we see that, as expected, the path hint cache improves latency only marginally. However, the path hint cache is essentially free, requiring no communication overhead and a small amount of memory to maintain.

We also see that PCX performs worse even than Chord. The reason for this is that PCX optimizes for hops and caches routing information independent of the latency between the caching peer and the peer being cached. The latest version of Chord and our path hint caches use latency to determine what entries to place and use in the caches and in the routing tables. For peers with high latency, it is often better to use additional hops through low-latency peers than fewer hops through high-latency peers.

Figure 3.6(b) shows this effect as well by presenting the cumulative distribution of space walk hops across all peers for the various algorithms. Each point (x, y) in this figure indicates that y percent peers have at most x space walk hops. Using the metric of hops, PCX performs better than both Chord and PHC. Similar to results in previous work incorporating latency into the analysis, these results again demonstrate that improving hop count does not necessarily improve latency. Choosing routing table and cache entries in terms of latency is important for improving performance.

We did the same experiment replacing the ”King” data set with the “Random” data set with 65,536 peers and Figures 3.7 shows the results of this experiment. These results are qualitatively similar to the results of above “King” data set.

The path hint cache are small and each peer aggressively evicts the cache entries to minimize the stale data. Hence the effects of stale data on lookup request performance is marginal.



(a) King data set

(b) Random data set

Figure 3.8: Global hint cache performance in different network models.

3.3.6 Global Hint Caches

Finally, we evaluate the performance of using the global hint cache together with the local and path hint caches. We compare its performance with "Standard" Chord and "OneHop". In this experiment, we simulated 8,192 peers with both 32 and 256 entries in their local hint caches. We have two different configurations of local hint caches to compare the extent to which the global hint cache benefits from having more candidate peers from which to select nearby peers to place in the global hint cache. (Note that the global hint cache uses only the second half of the hosts in the local hint cache to select nearest hosts; hence, the global hint cache uses only 16 and 128 entries to choose nearest peers in the above two cases.) Each peer constructs its global cache when it joined the network as described in Section 3.1.4. We collected the peer's routing tables once the network reached a stable state during the experimentation phase, and calculated the space walk latencies for each peer from the tables.

Figure 3.8(a) shows the cumulative distributions of space walk latencies across all peers for the various algorithms. The "Standard", "OneHop", "LocalCache", "GlobalCache-32", and "GlobalCache-256" curves represent Chord, the "OneHop" approach, a 1024-entry local hint cache, a 32-entry global hint cache with a 256-entry local hint cache, and a 256-entry global hint cache with a 32-entry local hint cache. Compar-

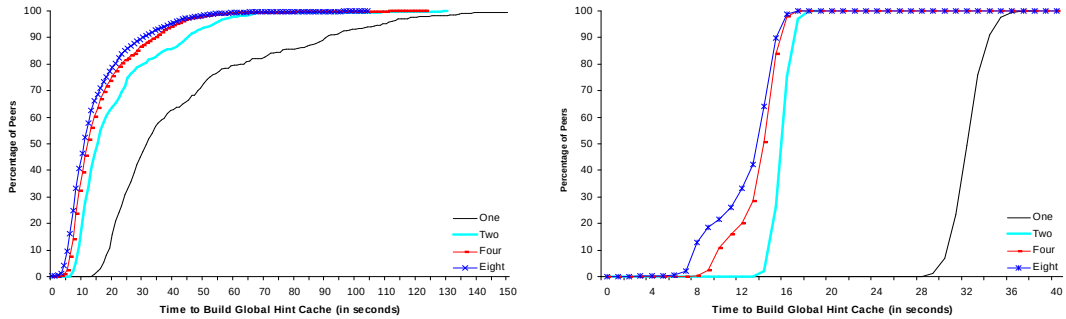
ing the size of local hint caches used to populate the global hint cache, we find that the median space walk latency of "GlobalCache-256"is 287 ms and "GlobalCache-32" is 305 ms; the performance of the global hint cache improved only 6% when it has more peers in the local hint cache to choose the nearest peer.

We also did same experiment with Random data set for 262,144 peers. We set the local hint cache size to 128 instead of 1024 to grow the global hint caches to few thousand peers which is the expected global hint cache size when the network has few million peers. This is the maximum network size we could able to simulate on our cluster machines. At this point, the main memory becomes the bottle-neck for our simulation. The results are presented in Figure 3.8(b) and the global hint cache performed close to the "OneHop" approach (the median space walk latency of global hint cache is 2% more than the median space walk latency of "OneHop" approach).

Comparing algorithms, we find that the median latency of the global hint cache comes within 6% of the "OneHop" approach when the global hint cache uses 128 of 256 entries in the local hint cache to choose nearby peers. Although these results are from a stable system without churn, the amount of stale data in the global hint cache under churn is similar to the local hint cache under churn because both of them use a similar update mechanism. Hence, the global hint cache performance under churn is same or a little better than the local hint cache performance under churn because global hint cache is filled with the nearest peers as opposed to the random peers in local hint cache. As a result, the effect of stale data in the global hint cache is negligible for a one-day system half life time and four-hour system half life time. Overall, our soft-state approach approaches the lookup latency of algorithms like "OneHop" that use significantly more communication overhead to maintain complete routing tables.

Global Hint Cache Build Time

Since we contact closer peers while constructing the global hint cache, one can build this cache within a few minutes. To demonstrate this, we calculated the time to build the global hint cache for King data set with 8,192 peers and Random data set



(a) King data set

(b) Random data set

Figure 3.9: Global hint cache build time

with 262,144 peers. Figure 3.9 presents the results of this experiment as a distribution of cache build times. Each point (x, y) on a curve indicates that y percentage of peers needs at most x seconds to build the cache. Each peer has around 500 peers in its the global hint caches and 16 peers in its local hint cache for King data set and in the Random data set each peer has around 1500 peers in the global hint cache and 128 peers in the local hint cache.

In the King data set, on the average it took 45 seconds to build the global hint cache. A peer can speed up this process by initiating walks from multiple peers from its routing table in parallel. The curves labeled "Two", "Four", and "Eight" represent the cache build times with two, four, and eight parallel walks, respectively. As expected, cache build time reduces as we increase the number of parallel walks. The median reduces from 32 seconds for single walk to 12 seconds for four parallel walks. We see only a small benefit of increasing the parallel walks after four parallel walks.

The global hint cache build times for Random data set has similar trends as the global hint cache build times for King data set. Even though global hint cache sizes are bigger in the Random data set the build time is less comparing with the King data set. The Random data set has more peers in the local hint cache which improve the chances of finding a nearest peer for a given peer which in turn improves the build time.

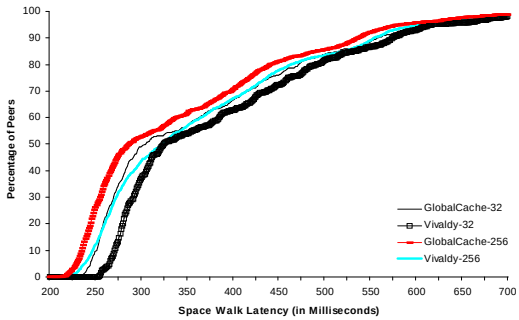


Figure 3.10: Effects of Network Coordinates

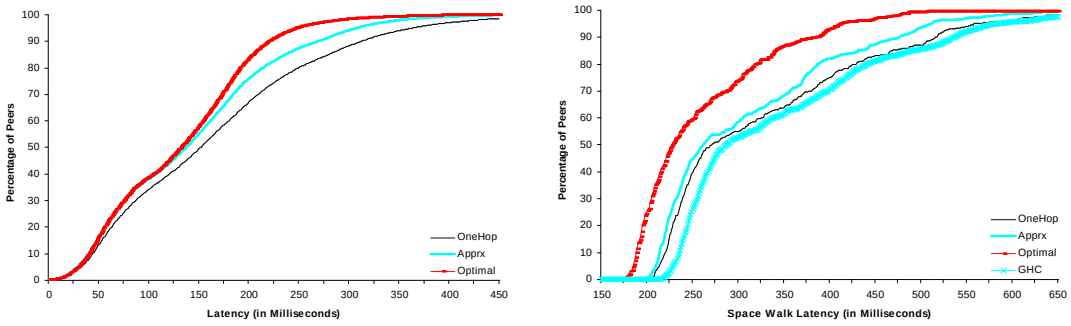
Network Coordinates

So far we have assumed that, when populating the global hint caches, peers are aware of the latencies among all other peers in the system. As a result, the results represent upper bounds. In practice, peers will likely only track the latencies of other peers they communicate with, and not have detailed knowledge of latencies among arbitrary peers. One way to solve this problem is to use a distributed network coordinate system such as Vivaldi [11]. Of course, network coordinate systems introduce some error in the latency prediction. To study the effect of coordinate systems for populating global hint caches, we next use Vivaldi to estimate peer latencies.

In our simulation we selected the nearest host according to network latency estimated according to the Vivaldi network coordinate system, but calculated the space walk time using actual network latency. We did this for the "GlobalCache-32" and "GlobalCache-256" curves in Figure 3.8. Figure 3.10 shows these curves and the results using Vivaldi coordinates as "Vivaldi-32" and "Vivaldi-256". The performance using the coordinate system decreases 6% on average in both cases, showing that the coordinate system performs well in practice.

3.3.7 Shun Pikes

In section 3.2 we described ways to exploit the detours to improve the lookup performance even further than the "OneHop" approach. In this section we evaluated



(a) Latency distributions

(b) Space walk latency distributions

Figure 3.11: Effects of shun pikes

the shun pikes for the King data set. The latency distributions between all pairs are presented in Figure 3.11(a) and Figure 3.11(b) shows the performance results of our new lookup algorithm.

First we calculated the shortest path between all pair of peers in the data set assuming that each peer has full knowledge of other peers. We just simply ran the shortest path algorithm on network latency matrix. The result of this is presented in Figure 3.11(a) as the "Optimal" labeled curve. We then calculated the shortest paths from each peer to all other peers assuming each peer has 64 peers in its local hint cache and approximately 64 peers in the global hint cache. The resultant latency distributed is represented as "Apprx" curve in Figure 3.11(a). The curve "OneHop" shows the network latency distribution. Overall for 40% of pairs shorest path latency improved over the one hop latency. The "Apprx" and "Optimal" performed similarly for smaller latencies and differ a bit at higher latencies. Though theoretically the "Apprx" should be constant factor away from the optimal case in general Internet like graphs. However, in this case we got similar performance as the optimal case when each host has partial information about the entire graph.

The Figure 3.11(b) shows the space walk latency for one hop, shortest path latencies, our new lookup algorithm and our global and local hint caches . The "OneHop" and "Optimal" curves represent the one hop and shortest path latency approaches. The

"GHC" curve represents the space walk latency of our global hint cache along with the local hint cache that is same as the "GlobalCache-256" curve in Figure 3.8(a). The "Apprx" curve represents the space walk latency of our approach where each peer has only partial information about the entire network and each peer tries to optimize the lookup as described in section 3.2. The median space latency for our caches, one hop, our new lookup algorithm and shortest paths is 287, 271, 262 and 228 milliseconds. Our new lookup algorithm improved approximately 10% over our global and local hint caches and it improved around 3% over one hop approach. As expected our lookup algorithm performed approximately 15% slower than the optimal because we need one extra hop to complete the lookup. Overall our new lookup algorithm performed a little better than the one hop approach.

3.4 Conclusion

In this chapter, we describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to neighbors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

We simulate the effectiveness of these hint caches as extensions to the Chord DHT. Based upon our simulation results, we find that the combination of hint caches significantly improves Chord routing performance with little overhead. For example, in networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like "OneHop" that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

3.5 Acknowledgment

Chapter 3, in full, is a reprint of the material as it appears in the Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04) 2004, by Kiran Tati and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Trace Collection

In this chapter, we present the long-term failure characteristics of hosts in a large-scale peer-to-peer file sharing system spanning over six months, as well as the methodology we used to gather such traces.

Object maintenance strategies depend on the general characteristics of peer-to-peer environments, such as the availability of hosts, the rate at which they temporarily depart and arrive back into the system, and their overall lifetime and rate at which they leave the system permanently. The behavior and performance of object maintenance strategies fundamentally depend upon these availability and failure characteristics. In particular, for environments with high permanent churn, object maintenance strategies incur much of their overhead repairing object redundancy to maintain availability. Unfortunately, however, we know little about the long-term permanent failure characteristics of hosts in environments with high permanent churn, such as large-scale cooperative file sharing and storage systems.

Previous work trace peer-to-peer host availability and failures study systems for only a short duration of one day to two weeks [4, 76] that are available ranging from one day to two weeks. From such data, we can measure some failure characteristics of these systems, such as average host availability and the degree of temporary failures over a short time span. However, these traces are not long enough to characterize the permanent failures in the system, as well as the stability of various characteristics such

as average host availability and temporary failures over a long period of time.

To design improved object maintenance strategies, we need characteristics derived from long-term (at least a hundred days) P2P traces with host arrival and leave events. Consequently, we performed a long-term study of a peer-to-peer file sharing system to measure such failure characteristics. In this chapter, we describe the structure of the P2P system we trace, our trace collection methodology, and a validation of our trace collection methodology. We present the long-term failure characteristics that we use to design and evaluate our object maintenance strategies in Chapter 5. Finally, we describe how we used this trace to generate a synthetic trace that extrapolates to two years of host behavior with the same availability and permanent failure characteristics required to evaluate the various object maintenance strategies.

4.1 KAD Network

We trace hosts on the KAD network. KAD is a typical P2P system that is currently actively used by millions of people. The network has approximately 1.5 million to 3 million online hosts at any given point of time [33, 76]. KAD is a P2P network based on Kademlia protocol. eMule [18], aMule [2] and MLdonkey [53] are popular open source clients for the KAD network. We use the aMule Linux-based client to connect to the network and perform our measurements.

When the system has a population of n hosts, each host maintains $log(n)$ routing entries to other hosts in the system, enabling it to route any request in $log(n)$ hops. Each host has a unique 128-bit ID and it always uses this ID whenever it connects to the system, unless the user explicitly regenerates a new ID or reinstalls the client software. We consider each ID as a separate host. Each host maintain this information as a binary tree, with the left edge labeled as '0' and the right edge labeled as '1'. Only the leaf nodes in the tree, referred to as *bins*, store routing entries to hosts. Each bin may have at most 10 entries.

A host is inserted into a bin traversing from the root according to the XOR-

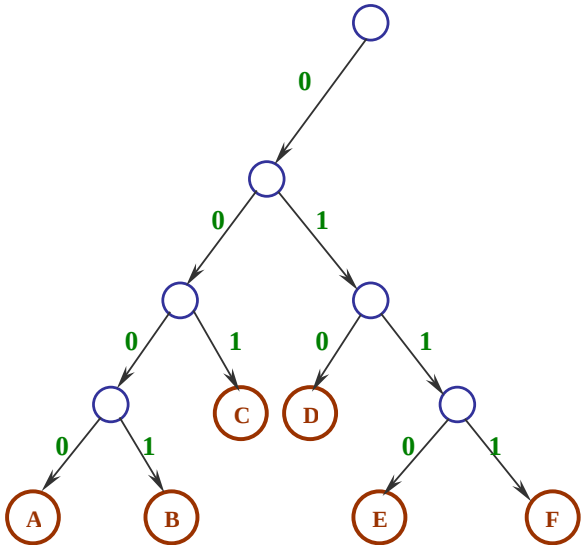


Figure 4.1: A sample routing table for a host.

distance between the ID of host being inserted and the ID of the host whose routing table is being updated. For example, if a host A trying to add host B into its routing table, host A first calculates the distance between A's ID and B's ID ($A_{id} \oplus B_{id}$). It then traverses from the root using the distance in binary form as a path to the leaf node. It inserts an entry for the host into a bin if the bin is not full. Otherwise, it will split the bin by creating children, move the previous hosts to appropriate new bins, and continue until it B fits into a bin.

Figure 4.1 shows an example routing tree for a host. In this routing tree, the bins are A, B, C, D, E, F and store entries representing routes to other hosts. The internal nodes do not store any entries. All the entries in the bin D have distance $010...$ (XOR with the host's ID). This path label from the root to the bin forms a binary number referred to as the *index*. A bin splits only if the index is less than 5. In this sample routing tree, if bin F is full then the node will not split bin F and it will not insert new entries into this bin. This mechanism tries to expand left side of tree as much as possible to get as many closest hosts as possible into the routing table, and also limits the number of right bins (at most 5 bins once it moved to right).

A host uses three mechanisms to maintain its routing table. It periodically

(once in every one hour) pings the hosts in each bin to minimize the staleness of the routing entry. It periodically (once in every hour) makes a lookup request to fill the bins that have few entries (less than four entries). Finally, it also looks for itself periodically (once in every four hours) to discover nearby hosts (in ID space) to expand the left side of the routing tree.

Note that a host cannot join the KAD network fully if it is unable to received unsolicited packets from the other hosts in the system. This situation is true for the hosts behind firewalls. The firewalled hosts cannot be seen by other hosts in the system even though they can contact other hosts. Hence, these hosts are not fully integrated into the KAD network, acting solely as clients, and we neither see them in our traces or consider them as true KAD hosts.

4.2 Methodology

Ideally we would like to capture the join and leave events of all the hosts in the system. There is no central point, however, where we can conveniently monitor all of these events because of the peer-to-peer nature of the environment. To capture all such events we would need to periodically download the routing tables of each and every host in the system, requiring an infeasible amount of bandwidth. Fortunately, the system uniformly distributes hosts across the ID space. As a result, we can choose a certain portion of ID space and monitor all the host join and leave events in this region [76]. We can then extrapolate from this ID region to the full ID space without significant loss of information. We will present evidence to support this claim later in the results section.

We start with a set of a few hosts in the region we are interested in. From each of these hosts routing tables, we download their entries to collect the IDs of all the hosts in the region that we are interested in. We continue this process until we discover no new hosts in the region. This process captures all the host join events in the region we are interested in if (1) we can get all the routing tree entries in a given region from each known host in this region, and (2) each host in the region has a routing entry to it in at

least one another host in the region (i.e., this other host has the host in its routing tree). We repeatedly ping the hosts we are interested in to capture the leave events.

4.2.1 Routing Entries From a Host

We use lookup requests to obtain the routing tree from a host. The lookup request returns the closest entries in a host routing tree for the ID sent in the lookup request. As an example, say host A's routing tree is as shown in Figure 4.1 and its ID is 0000 in binary form. It then receives a lookup request for 0111. It will first calculate the distance (XOR with its ID) and use this distance to find the closest entries in the routing tree. The distance in this example is 0111, hence all the entries in the bin F will be closest to the lookup request ID, then entries in the bin E, and the next bins will be D, C, B and A. It will return the entries from the bin F first, then from bins E, D, C, B and A until it finds 11 entries. Given that each bin in the routing tree stores at most 10 entries, if we carefully craft the lookup requests we can get the entire routing tree in b lookups, where b is the number of bins. We can deduce which bins we got fully by examining the last entry returned in a lookup request. In the above example, if we made a request that matches bin F and the response has an entry returned from bin C, then we know we have all of the entries from bins F, E and D.

Unfortunately, some of the earlier eMule and aMule client versions have a bug in the lookup request processing that returns less than 11 entries even though the routing tree has more than 11 entries. In the worst case, a host will return only one entry. In this case, to get all entries in a bin we need to go deeper in the routing tree, thereby requiring more than one lookup per bin. The worst case is exhaustively searching all the nodes (128 nodes in the 128-bit ID space) in the routing tree, requiring 128 lookups to get one entry from a bin. In this worst case we always get the same entry for all 128 lookup requests.

In this case, as an optimization, we prune the lookup requests when we cross the 28th level (28th bit position). We know that the system has at most 2 million online hosts, and we need only 21 bits to represent such a population. Hence, if we use lookups

that differ in at most the first 28 bits, and we find that there are no other hosts that match the 28 bit prefix, with very high probability we will not obtain new hosts from this bin from further searches. At this point, we stop the bin search and continue probing the rest of a host's routing table.

To understand the implications of this assumption, we allowed our algorithm to continue until it reaches the 40th level (40 bits of ID prefix) and collected hosts for four hours. We also record all the lookup requests we sent and the corresponding replies. We then calculate the number of hosts we might miss if we prune our search at the 28th level. No hosts were missed. Repeating the analysis for pruning at the 24th level, again no hosts were missed. Pruning at smaller prefixes, however, starts to lose routing information. Although 24 levels are enough for this situation, we choose to prune at 28 levels to be conservative. We submitted a patch to the client developers, and newer versions of the client fix the bug and our tracer can work with both cases.

4.2.2 Routing Entries From a Region

At this stage, we have algorithm to obtain the routing table from an individual host. We can get all the routing tree entries in a region by obtaining the entries from only bins that match the required number of bits. In the above sample routing tree in Figure 4.1, if we want to get all the entries that match the first three bits of the host then we need to get bins A and B only.

The second criteria for capturing all the hosts in a region is that each host in the region have a routing entry in the tree of at least one other host in the region being monitored. This criteria depends on how quickly a host builds its routing table so that other hosts in the system take notice of it. To determine the range of times it takes hosts to appear in a routing table of another host in the region, we measured the routing table build time of a few hosts in the system. To estimate the extent to which our methodology might miss hosts as they become settled in the system, we also simulated the dissemination of routing information for thousands of hosts.

We collected routing table build time measurements for 19 aMule clients. We

ran each aMule client for 5 hours between February 21 to March 1st, 2006 while recording changes to their routing table entries. To be conservative, none of the clients have any connection history and they contact well-known servers to bootstrap their routing table; the client stores randomly selected hosts from its routing table to a history file so that it does not have to depend on well-known servers on subsequent executions. Overall, 17 clients have entries in level 9 within 10 minutes, one client took one hour to populate the 9th level, and another one took four hours. The two clients that took a lot of time are on a day when a huge number of well-known servers were shutdown by the RIAA. These measurements indicate that, under stable conditions, clients can very quickly populate its routing table.

Next we determined the extent to which other hosts in an ID region have pointers in their routing tables to a particular host. We simulated the routing maintenance protocol of thousands of hosts to understand the impact of various factors (initial bootstrap hosts, number of hosts with above mentioned bug, and network connectivity) on when one host is noticed by another host in a region. We took an aMule client and replaced all the network code with a network simulator, and then we ran thousands of aMule clients in a single process. We recorded into a file whenever a host adds another host to its routing table to analyze it later.

First we looked at the impact of the client routing bug by varying the percentage of hosts that have the bug from 10–95%; there was no churn in the system, and all hosts are reachable from each other. We then calculate the time for a host to appear in the 9th level of the routing table of at least one other host. All 32,000 hosts in the simulation appeared in the routing table of another host within two hours, even when 95% of hosts have the bug.

We then examined the impact of network loss. We simulated 100,000 hosts and for each host we uniformly assigned a loss rate from 0–90%. A host with x% loss rate could not reach x% of hosts during the simulation. The network connections are asymmetric (i.e., host A can reach host B but host B cannot reach host A). There is no churn in the system, and no hosts have the bug. The results of this simulation are

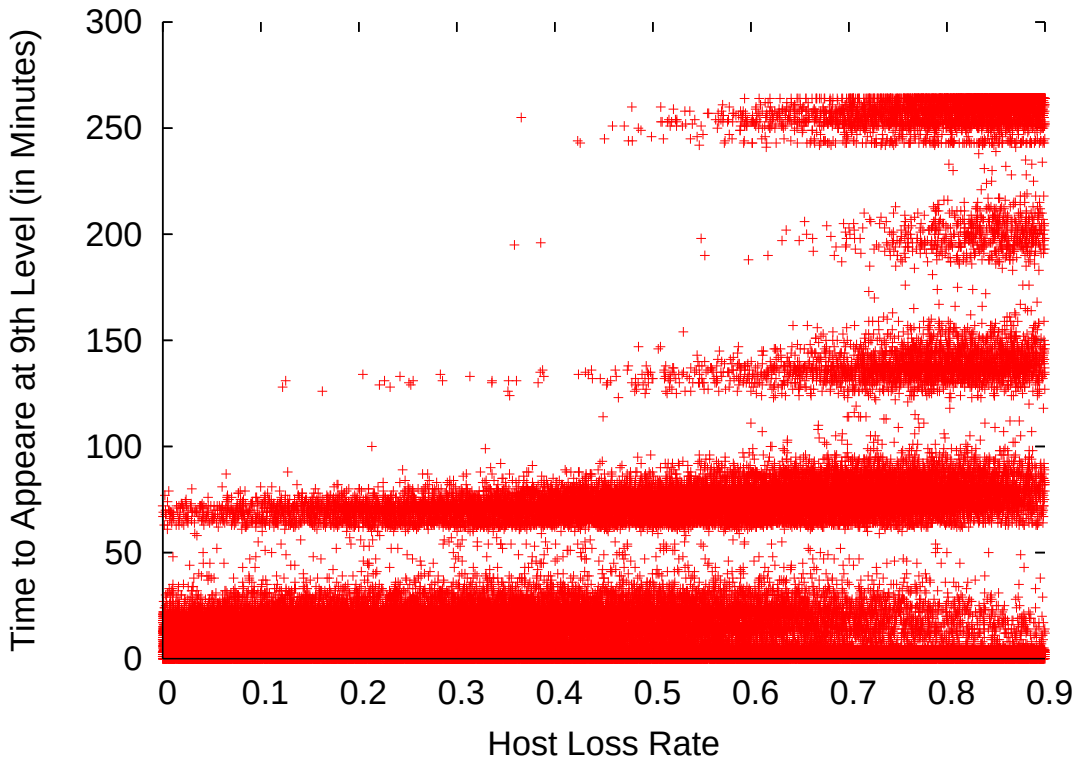


Figure 4.2: Histogram of time to reach 9th level while varying host network loss rate.

presented in Figure 4.2. Only 1821 (1.8%) hosts do not appear in the routing table of any other hosts at the 9th level, and all of these hosts have more than 60% loss rate. Except for a few hosts, all hosts whose loss rate is less than 40% appeared in a routing table at the 9th level within two hours. If a host has good reachability, it will appear in other hosts routing table within an hour.

We then simulated a more realistic, yet still conservative, scenario where each host has a 4% loss rate, 90% of the hosts are buggy, and there is host churn in the system. We initially simulate hosts without any churn for two hours to stabilize the system, and then 50% of the hosts leave at once so that the remaining hosts' routing tables contain approximately 50% stale data. From this point onwards we add and delete one host at a regular interval so that half of the hosts leave within four hours.

Figure 4.3 shows the results of this experiment. This graph is a CDF of the time to appear in the 9th level of all hosts whose lifetime is more than 30 minutes. The

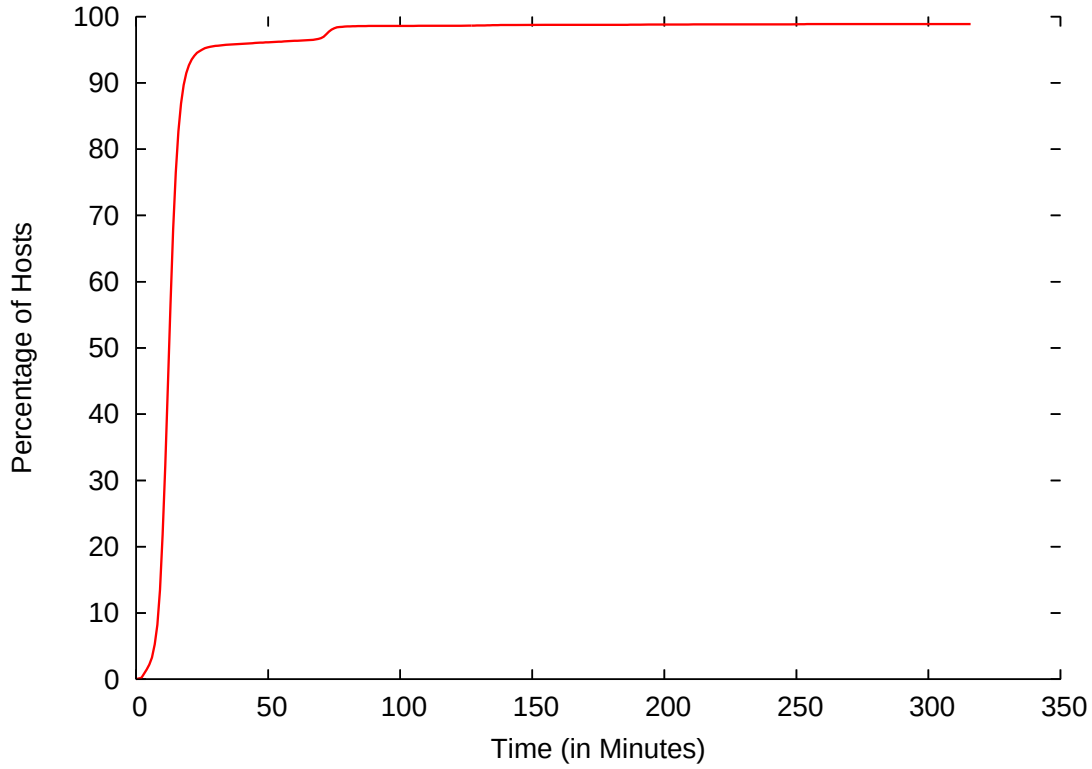


Figure 4.3: CDF of time to reach 9th level of all hosts whose lifetime is more than 30 minutes in a simulation with churn, buggy hosts and network loss.

x-axis is the time in minutes and y-axis is the percentage of hosts that reached 9th level. Only few hosts (463 hosts out of 41,357 hosts, just 1.1%) did not appear in the 9th level at the end of simulation. About 95% of the hosts required just 30 minutes to appear at 9th level in some other host. We also performed a similar experiment for hosts whose lifetime is four hours, and only one host out of 9,706 hosts did not appear in the 9th level of some other host. Hence, if hosts stay long enough they will eventually appear in 9th level of some other host.

Overall, if a host stays only 30 minutes it will appear in a routing table with 0.96 probability. If a host comes twice with 30 minutes session time, the probability improves to 0.998; and if it stays in the system for more than four hours, the probability is 0.9998. Hence, most hosts in the system will appear at the appropriate level in the routing table of at least one other host in an ID region even if it connects to the system

twice with just 30 minute session times.

4.3 Analysis

We implemented the routing table capturing methodology as a separate thread in the aMule client. We randomly chose a 9-bit prefix (101100000) for collecting hosts in the region from $0xC000 \ldots 000$ to $0xC07F \ldots FFF$ (1/512th of the entire ID space). We repeatedly collected routing table entries in this range from all the hosts we discover in this range. We collected routing table entries once every 10 minutes. In the routing table, we collected the IDs of hosts from level 9 and below that match the prefix.

We ran this instrumented aMule client on a machine in our cluster starting on April 24th, 2006. We collected more than 160 days of trace, and we continue to collect data. The machine has a 2.8 GHz CPU with 2 GB main memory and a 1 Gbps Ethernet card. The aMule client used on average 40 - 60% of user CPU load. It takes 50 MB memory at the start and the memory usage increases as it acquires new host information, peaking at 400 MB. On average, it consumed around 600 Kbps bandwidth continuously, including the bandwidth for writing traces files to an NSF server.

For reliability, we also run a watchdog program that constantly monitors the instrument aMule client and restarts it after sleeping for 60 seconds if the aMule client dies for any reason. Our aMule client dies regularly (the time period ranges from few hours to one month) and our watchdog program restarted automatically in all cases. The aMule client stores information about all the hosts that are online in the last four hours into a file and restarts with these hosts in its routing table if it restart from a failure. Hence, it just takes only one round to recapture all the online hosts. This round typically takes 10 minutes to recover from a failure, minimizing data loss due to failures. This state is also used to generate a trace file that consists of host join and leave events. The aMule client also aborts if it loses more than 50% of hosts in its routing table within 10 minutes; this situation accounts for most of our client failures. During the trace period, we lost data just five times for more than one hour due to network and power

Table 4.1: aMule outages that last more than one hour.

Date	Duration	Cause
06/08/2006	60 minutes	Recovery took more than one hour
07/28/2006	80 minutes	No data is written to log
07/31/2006	3 hours	UCSD Network Outage
08/24/2006	75 minutes	No data is written to log
09/04/2006	14 hours	Power Outage

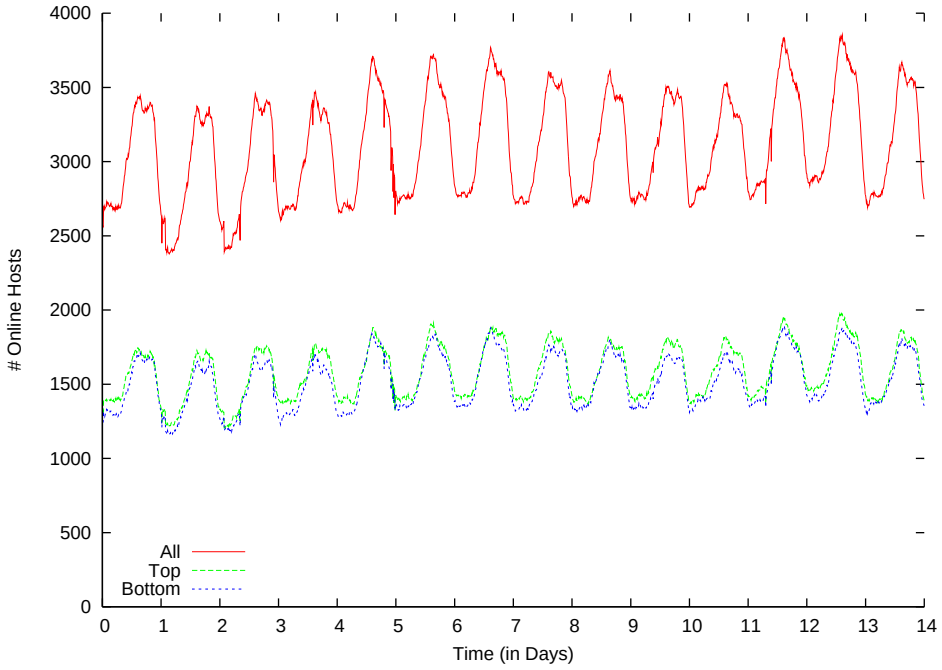
interruptions. We list these outages in Table 4.1 along with the causes of data loss. We introduced a bug that puts the trace collection thread asleep for more than one hour, and this situation happened twice in the entire trace on July 28th and August 24th. In these two occasions, we calculated the sleep times incorrectly.

4.3.1 Churn in the System

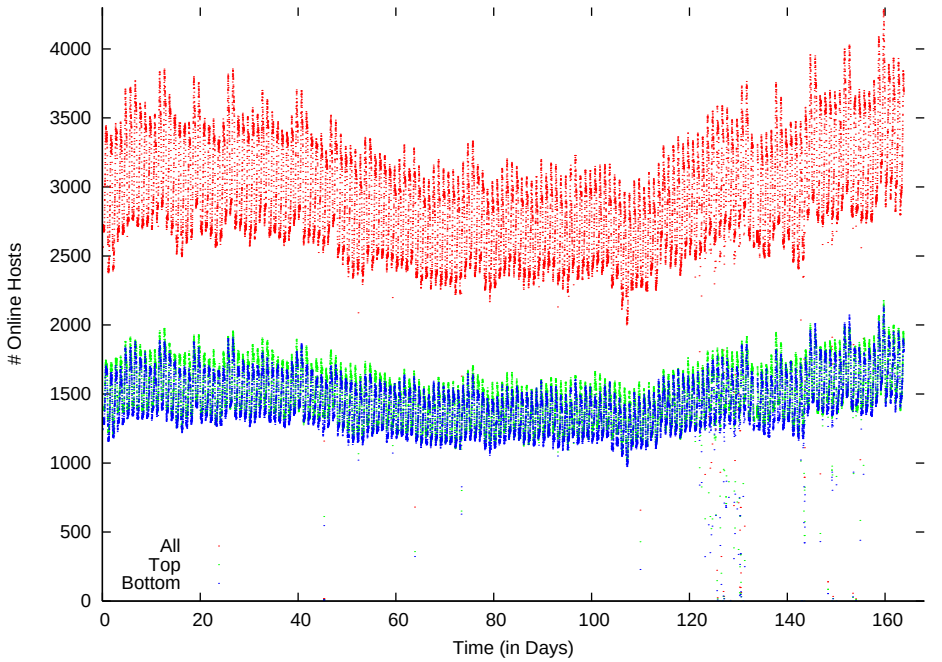
As a first step, we examine the size of the host population in KAD over time and characterize the global churn characteristics. Across the entire trace, we observed 148,333 unique host IDs in the trace, with 74,186 hosts (50% of all hosts) in the top half of the monitored ID space and 74,147 hosts (50% of all hosts) in bottom half of the ID space.

Figure 4.4 shows the number of hosts online connected to KAD whose IDs fall within our monitored ID range at each timestamp in our trace. It shows the number of online hosts in two graphs corresponding to two time scales. In each graph we present three curves, one for the entire ID range we monitored ("All"), and two others for the top ("Top") and bottom ("Bottom") halves of the monitored ID range (a 10-bit prefix).

Figure 4.4(a) shows the first 14 days to show daily churn characteristics with a clear diurnal pattern. Figure 4.4(b) shows the overall trend for entire trace period (168 days). The maximum number of hosts seen each day stayed around 3,500 until the end of May, 2006, falls to around 3,250 for 75 days during the summer (around middle of Aug, 2006), and then continues to increase after summer ends. The maximum number of hosts observed in a day is 4,284 hosts in the monitored ID range, and 2,177 and 2,136 hosts in the top and bottom halves of the ID range, respectively. (In the next chapter,



(a) 14 days



(b) 168 days

Figure 4.4: Overall churn in the system.

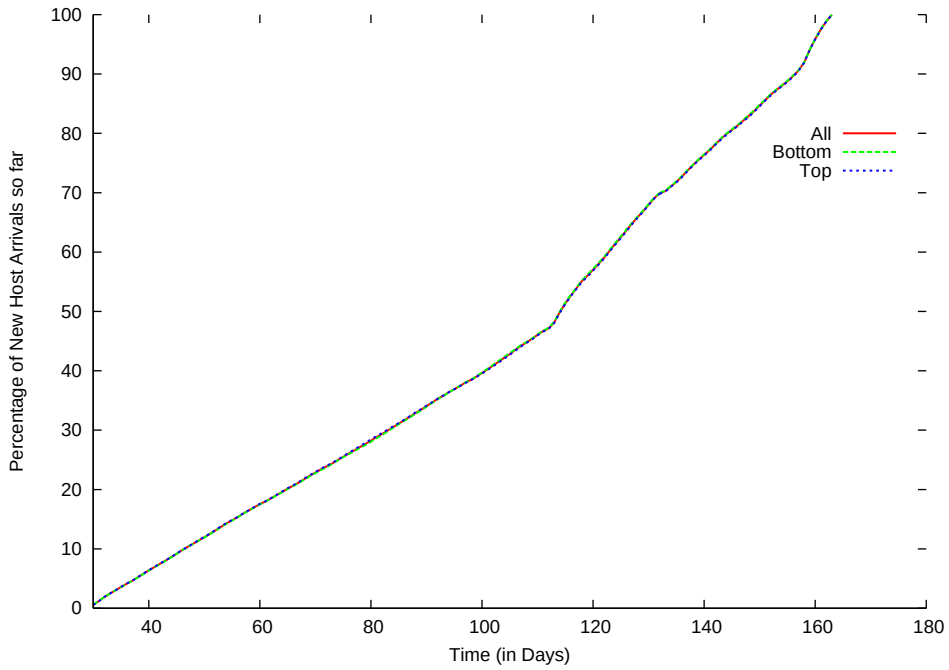


Figure 4.5: New host arrivals in the trace.

we show more high-level trace characteristics in the context of traces of other systems in Table 5.1.)

4.3.2 Arrival and Departure Rates

We created a trace file from the snapshots of current online hosts that are taken every 10 minutes. We define a host as a new host if we did not see it in the first 30 days of the trace, and a host has left the system (permanently failed) if we did not see it in the last 30 days of the trace.

Figure 4.5 shows when hosts appear for the first time in the trace. The x-axis is the time in days and y-axis is the percentage of new hosts seen so far. Note that we do not see any new hosts during the initial 30 day period because of our definition of new hosts. Overall, we saw 114,530 new hosts in the system during the entire trace. We also show the same kind of curve when we halve the collection ID range. The “All” curve represent the new host information from entire ID range, the “Bottom”-labeled curve represent the new host information when we consider only bottom half of the ID

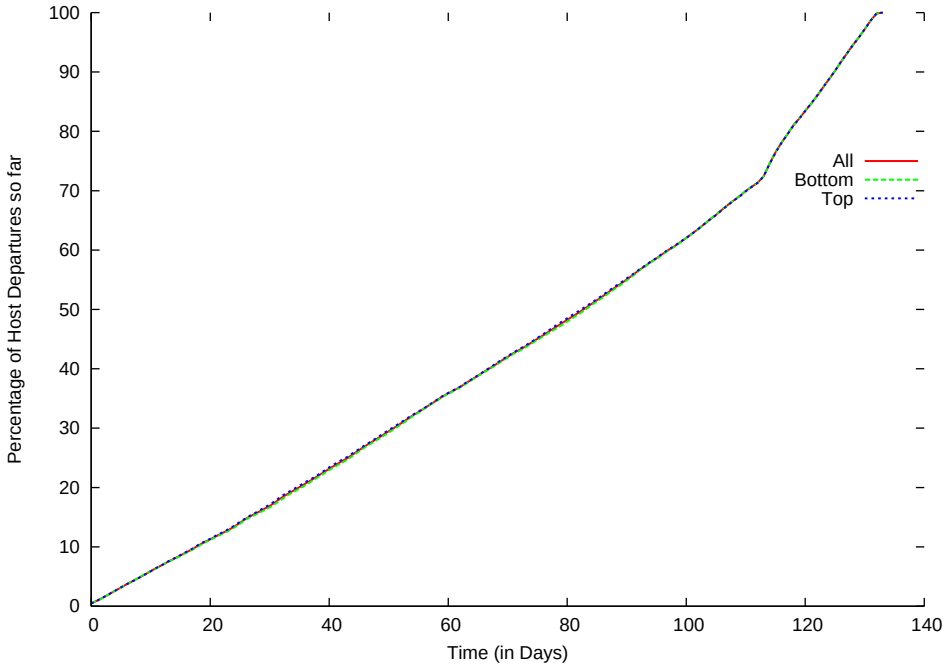


Figure 4.6: Host departures from the system (permanent failures) in the trace.

range, and similarly the "Top"-labeled curve represents the new host information when we consider the hosts from the top half of the ID range. We present these two additional curves to show that the characteristics we observe in a smaller sample space hold for the larger ID range. The curves are so similar that they overlap on the graph.

The slopes of the curves are piece-wise linear with a change in slope at 113 days. At this time, the number of new hosts per day doubled. Even though the magnitude changed after 113 days, the characteristic that the number of new hosts is proportional to the time period, remains the same. This behavior holds for the smaller ID ranges as well, suggesting that this characteristic holds for hosts in other portions of the ID space.

Figure 4.6 shows the characteristics of host departures from the system (permanent failures) in our trace for the entire ID range that we monitored ("All"), as well as the two sub-ranges ("Bottom" and "Top"); again, the curves are so similar that they overlap in the graph. We define that a host has left the system forever (permanent failure) if we do not see the host ID in the last 30 days of the trace; its last departure time is the host death time. The x-axis is the time in days and y-axis is the percentage of dead

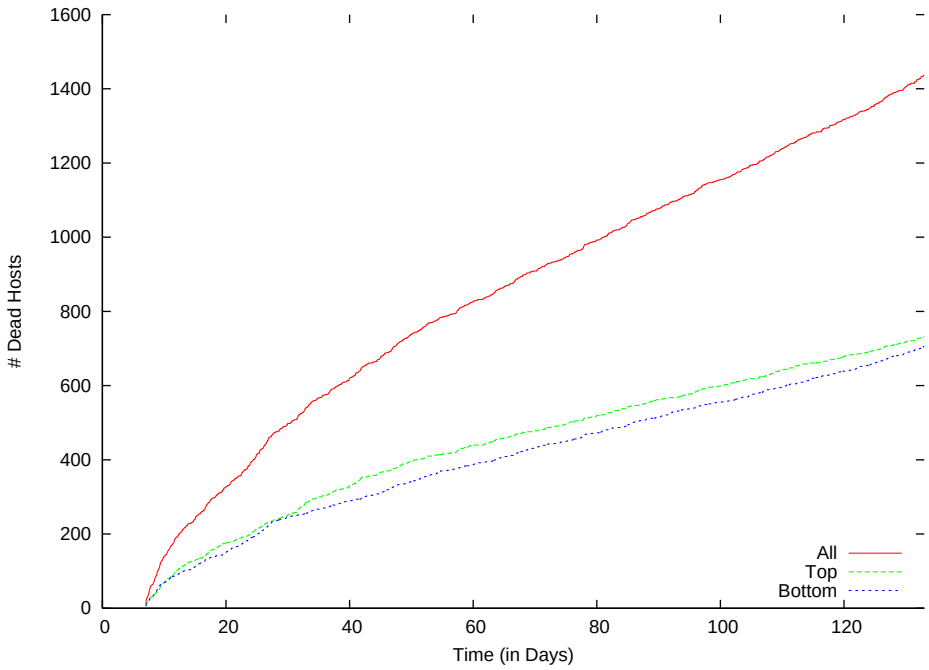


Figure 4.7: Number of dead hosts in a group of all hosts.

hosts so far. As with host arrivals, hosts die at a constant rate over time, and this rate doubles on the 113th day. Overall 94,865 hosts remain in the ID range after 160 days.

4.3.3 Dead Hosts in a Group

In the previous section, we presented characteristics of all hosts in the system. However, the overhead of object maintenance in peer-to-peer storage systems fundamentally depends on the characteristics (especially permanent failures) of the hosts on which an object is stored. Hence, in this section we present the permanent failure characteristics of specific groups of hosts over the lifetime of the trace.

Figures 4.7–4.10 show the number of dead hosts over time in four different groups representing different categories of hosts. In each graph, the x-axis is the time in days and the y-axis is the number of dead hosts in a group. We selected all online hosts at the start of 7th day in the trace and tracked the number of dead hosts in this group (Figure 4.7). This group corresponds to the TotalRecall strategy for randomly choosing hosts to store an object. As with the previous two figures, we show three curves in each

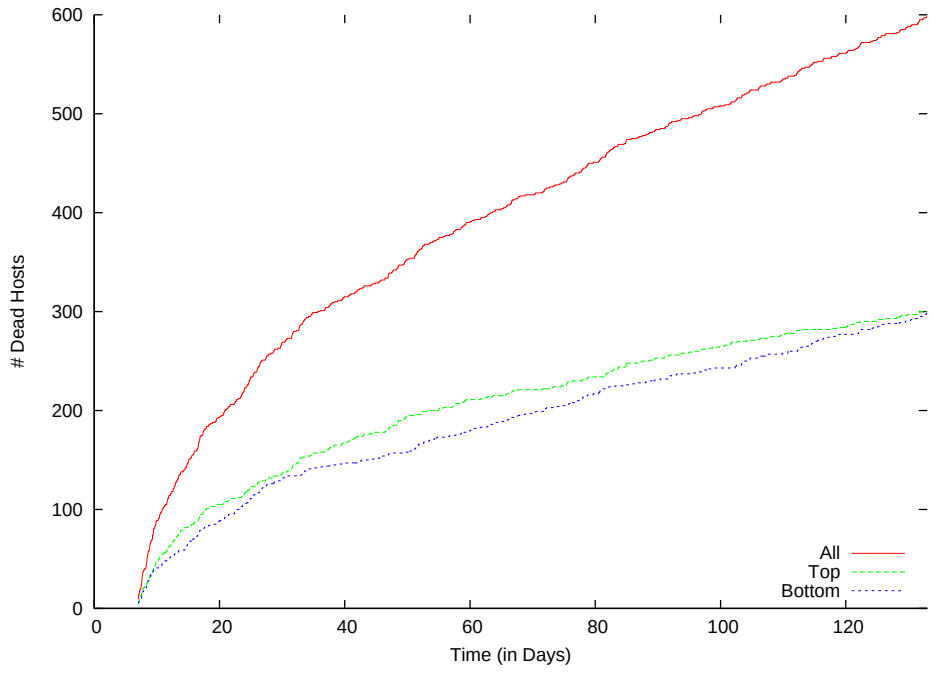


Figure 4.8: Number of dead hosts in a group of highly-available hosts (availability greater than 0.5).

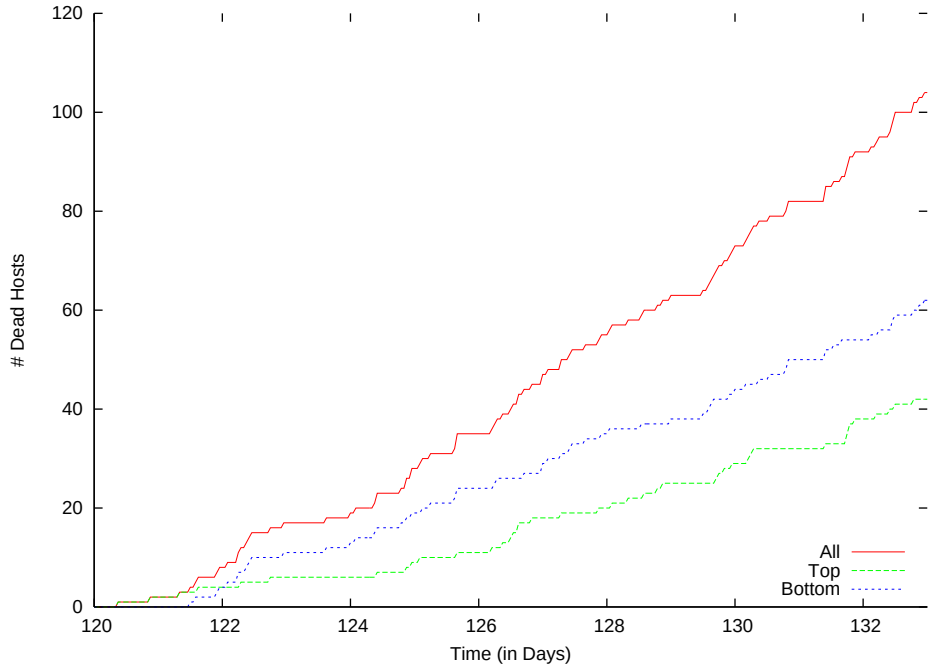


Figure 4.9: Number of dead hosts in a group of long-lived hosts (lifetime greater than 120 days).

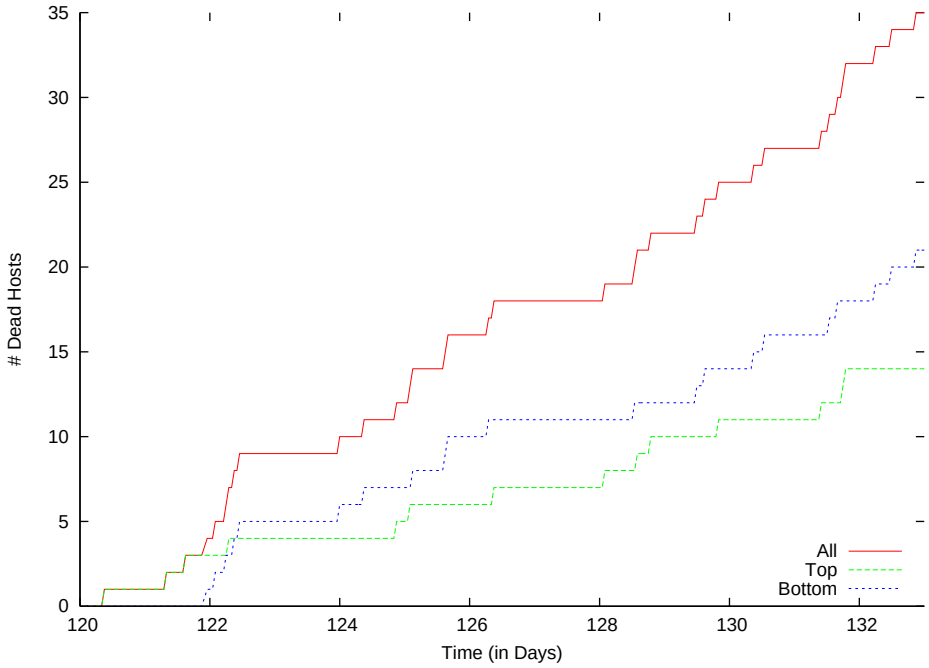


Figure 4.10: Number of dead hosts in a group of long-lived (lifetime greater than 120 days) and highly-available hosts (availability greater than 0.5).

graph: one one for all of the hosts in the ID range we monitored ("All"), another for the bottom half of the ID range where the 10th bit is zero ("Bottom"), and a third for the top half of the ID range where the 10th bit is one ("Top"). The number of hosts at the start is presented as the first row in the Table 4.2 for the three curves with the same column labels. Overall, there are two different phases and both of them are roughly linear. The first phase lasts from the 7th day to the 50th day. In this phase the groups lost hosts with a higher rate than the remaining phase, which lasts from the 50th day to the end of trace. The number of hosts at the start is approximately half when we divided the ID range into halves, and both of the subrange curves have similar shapes. These results suggest that characteristics scale across magnitudes of ID ranges, and that characteristics we find of portions of the ID space we monitor will apply to hosts in other portions of the ID space.

Note that, even after four months only 53% (1,537 out of 2,837 hosts) of hosts in the group are dead. The number of failures per host and per day is 1/247, much smaller than the entire population as previously reported (1/27) in Li's study [40] and

Table 4.2: Number of hosts at the start.

	All		Top		Bottom	
	Online	Dead	Online	Dead	Online	Dead
All hosts	2770	1439	1407	707	1363	732
Long-lived ($\geq$ 120 days)	1437 (52%)	106 (7%)	763 (54%)	63 (9%)	674 (49%)	43 (6%)
Highly-available ($\geq$ 0.5)	(999) (36%)	598 (42%)	496 (35%)	298 (42%)	503 (37%)	300 (41%)
Combined	436 (16%)	35 (2%)	219 (16%)	21 (3%)	217 (16%)	14 (2%)

(1/9) in Tati's study [78] — both of which are based on a 2-week trace of the Overnet file sharing system [4]. The permanent failure rate is, however, similar to PlanetLab's failure rate (1/314) as reported previously in [40] or (1/200) as reported in [78].

In the next chapter we are proposing various ways to select hosts on which to store an object. We would like to know the characteristic of dead hosts in these groups as a basis for analyzing object maintenance overhead. One way of selecting hosts is to choose the long-lived hosts among all online hosts at object creation time. To explore the permanent failure rates of this category of hosts, we selected all online hosts at the start of the 7th day whose lifetime is more than 120 days and tracked the number of dead hosts from this group over time. As before, we tracked dead hosts for both halves of the monitored ID range. The second row in the Table 4.2 shows the number of hosts at the start on the 7th day. Figure 4.9 shows the number of dead hosts in the ID ranges over time. The axis labels and curve labels are same as in the Figure 4.7. We show results from the 120th day onwards because, by definition, there are no dead hosts before that day. The number of dead hosts in all groups is linearly proportional to the time, suggesting that the permanent failure rate is constant over time.

As our third category, we performed a similar experiment by selecting only highly-available hosts (availability is more than 0.5), and Figure 4.8 shows the results. The characteristics are similar to the previous group categories.

Finally, we selected hosts based on both lifetime and availability. We selected

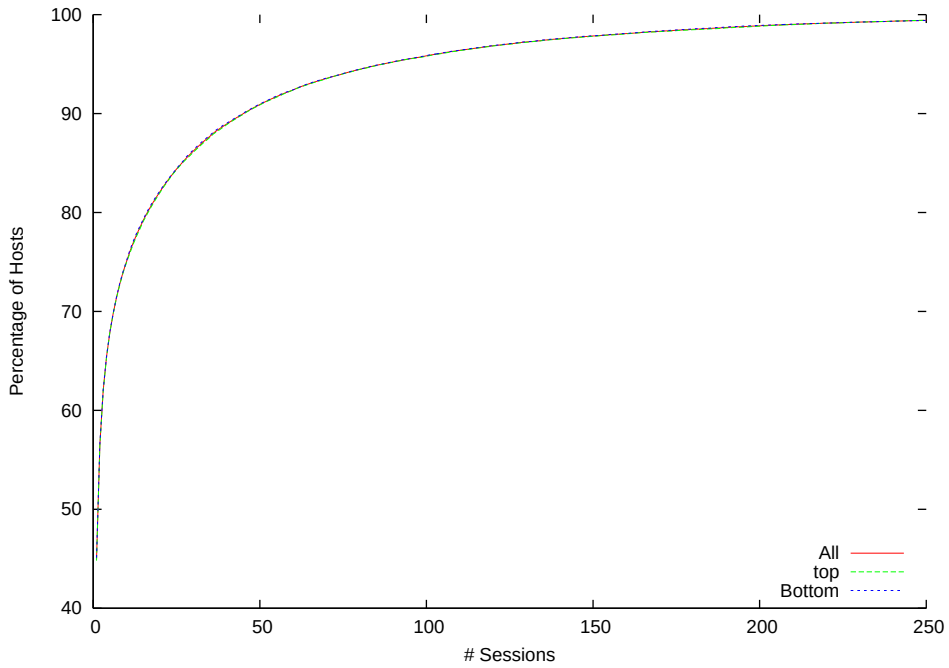


Figure 4.11: Host session durations.

all online hosts at the start of 7th day whose lifetime is more than 120 days and whose availability is higher than 0.5 and tracked the number of dead hosts in this group. We did same for the top half and bottom half of ID range in our trace. Table 4.2 shows the number of hosts in this category at the start in the third row, and Figure 4.10 presents these results over time. The characteristics (constant permanent failure rate) are similar to the previous curves in Figures 4.7 and 4.9. The number of failures per day and per host is 1/1615, which is similar to the disk failure rate reported for PlanetLab hosts (1/1825) [40]. Overall, the number of dead hosts is approximately linearly proportional to the time irrespective of how we selected (all, long-lived, highly-available, or long-lived and highly-available) online hosts at a given time.

4.3.4 Sessions

Lastly, we examine the distribution of host session durations to illustrate the range of client participation in the system. Figure 4.11 shows the CDF of host session durations for all hosts in the trace. A point (x, y) on a curve indicates that y percentage

of hosts or less have at most x number of sessions. As we did in the previous figures, we show the distributions for all hosts in the monitored ID range ("All") as well as the top ("Top") and bottom ("Bottom") halves of the ID range. From the graph, we see that 45% of the hosts have only one session, although the distribution has a long tail; 10% of the hosts have at least 50 sessions. And, as with previous analysis, the distributions of the ID subranges have the same shape as the entire range.

4.4 Conclusions

In this chapter, we describe a study of the availability and lifetime characteristics of hosts in a portion of the KAD peer-to-peer file sharing network over a span of six months. We describe our methodology for performing the active measurement study, analyze the ability of our methodology to capture sufficiently complete data, and show that peer-to-peer systems like KAD have substantial heterogeneity in host availability and lifetime characteristics. In the next chapter, we present object maintenance strategies that take advantage of this heterogeneity to reduce bandwidth overhead.

Chapter 5

Object Maintenance Strategies

Storage is often a fundamental service provided by peer-to-peer systems, where the system stores data objects on behalf of higher-level services, applications, and users. A primary challenge in peer-to-peer storage systems is to efficiently maintain object availability and reliability in the face of host churn. Hosts in peer-to-peer systems exhibit both temporary and permanent failures, requiring the use of redundancy to mask and cope with such failures (e.g., [84, 1, 38, 73, 5]). The cost of redundancy, however, is additional storage and bandwidth for creating and repairing stored data.

Since bandwidth is typically a much more scarce resource than storage in peer-to-peer systems, strategies for efficiently maintaining objects focus on reducing the bandwidth overhead of managing redundancy, trading off storage as a result. Typically, these strategies create redundant versions of object data using either replication or erasure coding as redundancy mechanisms, and either react to host failures immediately or lazily as a repair policy.

In this chapter, we revisit object maintenance in peer-to-peer systems, focusing on how temporary and permanent churn impact the overheads associated with object maintenance. We have a number of goals: to highlight how different environments exhibit different degrees of temporary and permanent churn; to provide further insight into how churn in different environments affects the tuning of object maintenance strategies; and to examine how object maintenance and churn interact with other constraints such

as storage capacity. When possible, we highlight behavior independent of particular object maintenance strategies. When an issue depends on a particular strategy, though, we explore it in the context of a strategy in essence similar to TotalRecall [5], which uses erasure coding, lazy repair of data blocks, and random indirect placement (we also assume that repairs incorporate remaining blocks rather than regenerating redundancy from scratch).[1]

Overall, we emphasize that the degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. Previous work has highlighted how ranges of churn affect object lookup algorithms [66]; in this paper, we explore how these differences impact the source of overheads for object maintenance strategies. In environments with low permanent churn, object maintenance strategies incur much of their overhead when initially storing object data to account for temporary churn. In environments with high permanent churn, however, object maintenance strategies incur most of their overhead dealing with repairs — even if the system experiences high temporary churn. We also present various strategies to reduce the object maintenance overhead. Finally, we highlight additional practical issues that object maintenance strategies must face, in particular dealing with storage capacity constraints. Random placement, for example, unbalances storage load in proportion to the distribution of host uptimes, with both positive and negative consequences.

5.1 Churn

Peer-to-peer systems experience churn as a result of a combination of temporary and permanent failures. A temporary failure occurs when a host departs the system for a period of time and then comes back. Any data stored on the host becomes unavailable during this period, but is not permanently lost. Examples of temporary failures are when home users login to systems in the evening, or when business users use systems during the day but logoff overnight. A permanent failure corresponds to a loss of data on

[1] We choose one strategy to be illustrative more than to advocate a particular approach, and choose this strategy because of familiarity; the tradeoffs between replication and erasure coding, for example, have been well studied [82, 49, 3, 6, 67], and each has its strengths and weaknesses.

Table 5.1: Churn in representative systems.

System		OverNet	KAD Network	PlanetLab	FarSite
Start Date		Jan 15th 2003	Apr 24th 2006	Apr 1st 2004	Jul 1st 1999
Duration		7 days	163 days	406 days	35 days
Total Hosts		1,469	148,333	655	60,000
Average Hosts Per Day		1,028	8,334	318	45,000
Temporary Failures	Total	33,084	2,500,353	13,633	87,500
	Per Day	4,736	15,339	34	2,500
	Per Host	4.61	1.8	0.11	0.05
Permanent Failures	Total	107	94,865	593	7,000
	Per Day	107	713	1.6	200
	Per Host	1/9.6	1/11.8	1/200	1/250

a host, such as when a disk or machine fails, or when a user leaves a file sharing system permanently. Temporary failures directly impact availability, and permanent failures directly impact reliability.

The degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. Systems incorporating home and business hosts tend to experience much higher levels of churn than systems incorporating server hosts maintained in machine rooms. For example, Table 5.1 illustrates the churn characteristics taken from traces of four different host populations, the Overnet file sharing system [4], the KAD Network file sharing system [33], the PlanetLab testbed [74], and hosts in a large corporation [7].

The observation that different environments experience different degrees of churn is not new, although characterizations of churn tend to focus just on temporary churn (e.g., [66]). Characterizing permanent churn in deployed systems has remained an open question, in part because doing so requires long-term measurement as well as assumptions about host behavior; deciding that a host has left permanently within a finite trace essentially requires a threshold for assuming that observing that a host has left the system means that it has left permanently. For the Overnet trace, we consider host departures where the host leaves for more than six days a permanent failure; all other host departures are temporary failures. Given the short period of the trace, using

a larger threshold would result in little permanent churn. As a result, we consider this threshold an upper bound on permanent churn for this population. For the PlanetLab and KAD Network trace, we consider host departures where the host leaves for more than thirty days a permanent failure; all other host departures are temporary failures. For the FarSite study, we use numbers reported in the paper.

Comparing the churn in the different environments, we see that the environments have very different degrees of both temporary and permanent churn. We normalize the metrics per day since user and host behavior tends to be diurnal. The "Ave Hosts Per Day" metric is the number of hosts in a system per day, averaged across all days in the trace. Normalized per day per host in the system, the file sharing trace exhibits an order of magnitude more temporary and permanent churn than the other environments. The wide-area population exhibits twice as much temporary churn as the corporate population, with roughly equivalent permanent churn.

Another way to compare churn in different environments is to consider the impact of churn from the perspective of object maintenance strategies. Systems maintain objects by storing redundant versions of object data among multiple hosts. As a result, the behavior of a maintenance strategy depends on the churn of a set of hosts selected at a particular point in time, such as when the object is initially stored. Figure 5.1 shows the effects of churn on a fixed set of hosts over time in the Overnet, PlanetLab, and KAD traces. We examined all hosts in each trace that were available 24 hours into the trace (530 for Overnet, 300 for PlanetLab, 2643 for KAD), and tracked their availability over six days. The graph plots the percentage of hosts in the original set that are available in the system over time.

The Overnet and KAD hosts exhibit a dramatic drop in availability in the first 24 hours; this drop is due to the high temporary churn in the environment, particularly hosts with short uptimes. All groups of hosts experience daily variations in availability, also due to temporary churn. Both Overnet and KAD exhibit a slow decay in host availability over time. This slow decay is due to permanent host failures slowly reducing the original set of hosts.

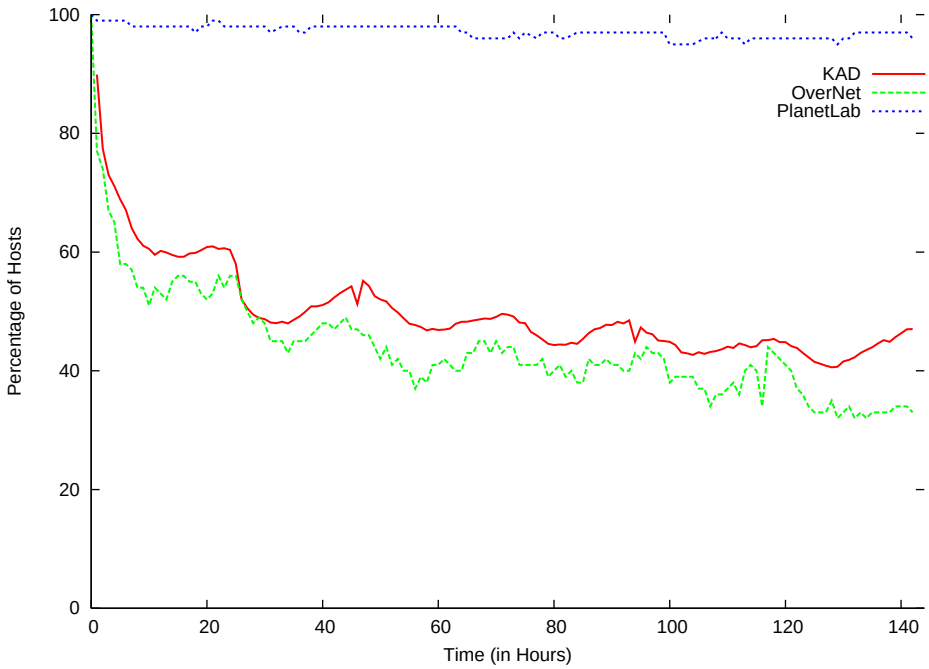


Figure 5.1: Tracking host availability among a set of hosts over time. The monitored set of hosts are those hosts that were in the system at 24 hours into the trace.

Sections 5.2 and 5.3 discuss the consequences of temporary and permanent churn in different environments in more detail. By comparing the characteristics of churn in different environments, we want to emphasize to degree to which environment matters. As we discuss in more detail later in the chapter, differences in environment impact the focus of object maintenance strategies.

5.2 Temporary Churn

In this section we focus on the approach of using redundancy to handle temporary churn. Since our goal is to provide insight into the problem, rather than advocate a particular algorithm, we make some simplifying assumptions to highlight temporary churn issues. In particular, in this section we assume that the host population has no permanent churn, only temporary churn, and that a host's availability characteristics do not significantly vary during its lifetime. Of course, in an actual system these assump-

tions are not realistic: any population experiences permanent churn, and host availability varies over time. Such shifts in host availability over time change the steady-state dynamics of the group of hosts storing object data. A maintenance strategy can detect and adapt to these changes over longer time scales.

Consider the events that occur immediately after storing object data in a system with only temporary churn. A maintenance strategy selects (typically randomly) a set of hosts on which to store the blocks comprising an object. One coincidental characteristic these hosts share is that they are all available at the time of object placement. These hosts, however, vary both in their uptime durations as well as how long they have been active in their current session. As a result, over time a fraction of these hosts will become unavailable due to temporary churn. Eventually, though, the number of simultaneously available hosts will stabilize in a diurnal pattern as hosts depart and arrive on a daily basis.

Given this behavior, a maintenance strategy can create sufficient redundancy to sustain the availability of an object on the minimum set of available hosts during a day. By "minimum", we mean that a sufficient number of hosts are available at any point in time such that the data they store is available for use; reducing the set by a host implies that at some time the data is not available. For strategies that use replication, unavailability occurs when all replicas are simultaneously unavailable; for those that use erasure coding, it occurs when an insufficient number of hosts are available to reconstruct object data. An object maintenance strategy can proactively estimate this amount of redundancy when initially storing the object (e.g., based on past behavior). Or, it can reactively add redundancy as hosts temporarily depart the system until the amount of redundancy is sufficient to mask temporary failures. Either way, eventually the set of hosts storing object data for a particular file will stabilize into a random process where a number of simultaneously available hosts storing data is sufficient to maintain object availability with high probability (although which hosts are simultaneously available varies over time). We call this state "masking" temporary churn, where an object maintenance strategy is using sufficient redundancy such that temporary churn will not

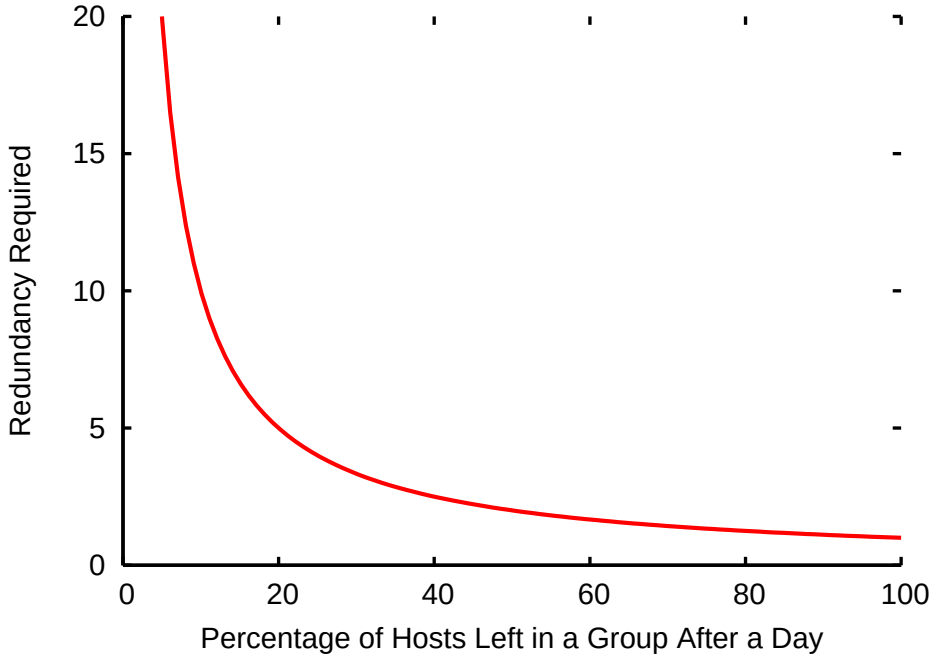


Figure 5.2: Relationship between temporary churn and redundancy (storage overhead) required to mask it.

induce repairs (in our idealized model), and result in only infrequent repairs in practice (due to changing host availabilities, etc.).

We also note that the data availability these hosts provide is probabilistic. Object maintenance strategies estimate the amount of redundancy required to provide object availability with a high probability based upon host availability characteristics (e.g., [12], [5]). If hosts in the system experience a sudden shift in availability (e.g., catastrophic simultaneous failures), the probabilistic availability guarantees will not hold. We also note that placement strategies typically assume that failures are not correlated. There are clear examples when failures are correlated (e.g., [31, 83]); however, even with noticeable diurnal patterns, correlation coefficients of host availabilities indicate that correlation of hosts in the system is not strong [4].

Figure 5.2 illustrates the relationship between temporary churn and the amount of redundancy required to mask it. By amount of redundancy, we mean the storage overhead used by a redundancy technique such as replication or erasure coding; a re-

dundancy of three, for example, means that the storage overhead is three times the file size. For a given group of hosts storing object data, the x-axis varies the percentage of hosts remaining available in the group after one day has passed since storing the object in the system. The y-axis shows the degree of redundancy required to keep the object available. Again, we focus only on temporary churn and assume that no hosts fail permanently. At a high level, it is a straightforward inverse relationship. For example, if any 50% of the original group of hosts are available in steady state and host arrivals are uniformly distributed, then the maintenance strategy will need to store the object with a minimum redundancy factor of two to maintain object availability in the face of just temporary churn.

Once the set of hosts storing object data stabilizes, a system will not need to frequently react to host departures or create further redundancy on additional hosts. As a result, a system incurs primary bandwidth overhead for masking temporary churn when it initially places the object in the system. With only temporary churn, no repairs are necessary. In environments with little permanent churn, tuning redundancy for masking temporary churn will have the greatest impact on minimizing bandwidth overhead. Of course, an actual system will still occasionally have to repair data redundancy due to temporary churn as, for example, host availabilities vary over time; such repairs will likely be incorporated naturally in the handling of permanent failures, discussed below.

In summary, using redundancy to deal with temporary churn has three implications: (1) an object maintenance strategy can determine a sufficient degree of redundancy to minimize repairs due to temporary churn, or "mask" temporary churn; (2) the amount of redundancy required to mask temporary churn is inversely proportional to the fraction of simultaneously available hosts storing object data; and (3) the bandwidth overhead for coping with temporary churn is dominated by object creation, not by repairs induced by temporary churn.

5.3 Permanent Churn

Permanent churn drives repairs. When the system permanently loses hosts storing redundant object data, the system must eventually repair the redundancy to ensure data reliability. The frequency with which the system repairs object data depends on the degree of permanent churn and the amount of redundancy restored during repair for long-lived objects. In environments with substantial permanent churn, like those that incorporate business and home hosts, the overhead for repairing long-lived object data dominates the overhead of establishing sufficient redundancy to mask temporary failures. As a result, in these environments tuning repair strategies to deal with permanent churn will have the greatest impact on minimizing bandwidth overhead.

When a system decides to repair object data, it must decide how much redundancy to restore. The more redundancy a system restores during a repair the longer it can delay the next repair, thereby trading off storage to reduce the frequency of repairs. In terms of bandwidth overhead, though, it is not immediately clear what the best choice is. An object maintenance strategy can either make "smaller" repairs more frequently, or "larger" repairs less frequently.

The choice depends upon the distribution of permanent host failures. We show that there exists an optimal balance between the amount of redundancy restored at each repair and the frequency of repair under the following model. Assume that the object maintenance strategy uses erasure coding and lazy repair [5], and that a repair replenishes any remaining data with new redundant data to maintain reliability (as opposed to regenerating it from scratch). Let x be the threshold at which the system triggers repair in terms of the number of hosts storing object data. An object maintenance strategy will restore redundancy by creating new coded blocks of data on N additional hosts (encoding with a large encoding graph enables the creation of incremental encoded blocks over time to supply repairs). Immediately after a repair, an object has blocks stored on $x + N$ hosts. Since the repair threshold is x hosts, from one repair to the next N hosts will fail permanently. This process takes $\frac{2Nd}{N+x}$ time, where d is the average rate of permanent

failures measured in terms of half death time (similar to half life time [43]), the amount of time it takes for half of the hosts to fail permanently; if we have N hosts, then it takes d time for $N/2$ hosts to fail permanently. The number of permanent failures in a group is linearly proportional to time Figures 4.7–4.10 in Section 4.3.3 show that the number of permanent failures in a group is reasonably modeled as linearly proportional to time. The half death time represents this relationship independent of group size, or group category of interest.

Our goal is to minimize bandwidth requirements for performing repairs. When using erasure coding, for example, the system must first read the object. Doing so requires f bytes, where f is the object size.[2] We also store new encoded blocks on N new hosts, requiring another Nf/a bytes, where a is the number of hosts an object gets fragmented onto. Overall each repair consumes $f + Nf/a$ bytes. The total bandwidth needed for a repair is $\frac{f(1+N/a)(x+N)}{2Nd}$ bytes per second, averaged over the interval of time between repairs. The minimum value occurs at $\sqrt{ax}$ and is $\frac{f(1+\sqrt{x/a})^2}{2d}$. A system will typically keep a, object fragmentation, constant. The value of x, the repair threshold, will depend on the amount of redundancy needed to mask temporary churn since the object needs to be immediately available at the time of repair.

Interestingly, the amount of redundancy to restore on a repair that minimizes bandwidth overhead depends upon the degree of temporary churn in the system, but not on the degree of permanent churn; the bandwidth overhead certainly scales with the rate of permanent churn, but the rate does not affect the choice of how much redundancy to repair. Figure 5.3 illustrates the relationship between the temporary churn experienced by a system and the amount of redundancy to restore on a repair that minimizes repair bandwidth overhead. As an example parameterization, we assume that files have a uniform size (f) of 1 MB, the repair threshold is twice the redundancy required to mask temporary churn, each file is fragmented (a) into 16 blocks, and the hosts in the system permanently fail (d) according to the Overnet trace (which contains both temporary and

[2] Optimizations are possible, such as storing a full replica at one host to eliminate the read [67], although we note that such optimizations increase storage cost and may not be practical for very large objects. We could modify our analysis to incorporate them, but our goal is to understand the trends more than absolute overheads.

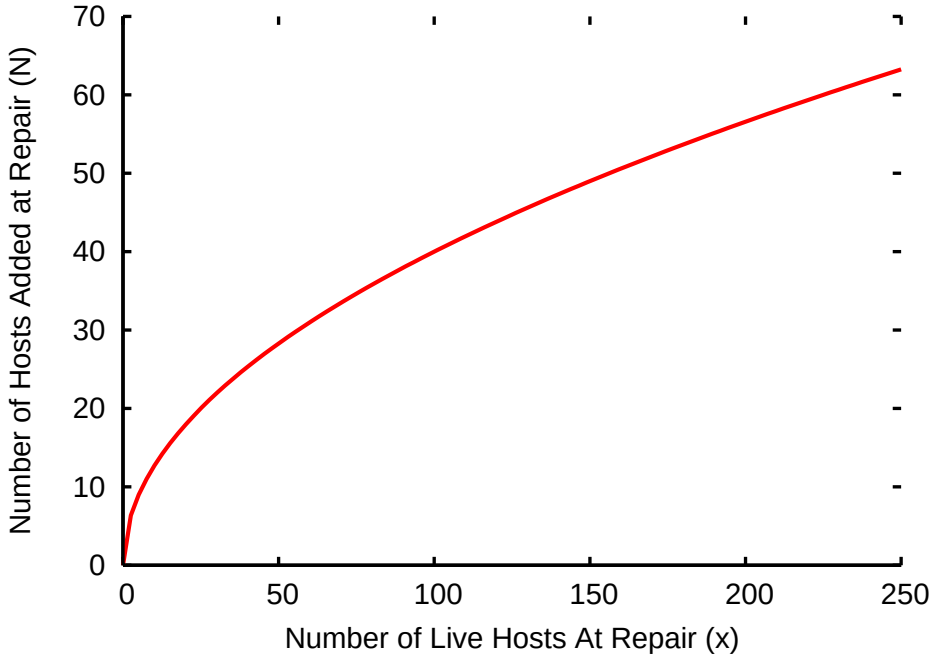


Figure 5.3: Optimal bandwidth required to mask permanent churn depending on degree of temporary churn (number of hosts required to mask temporary failures).

permanent churn). The x-axis shows the repair threshold (x) in terms of the number of hosts remaining that are storing object data; again, think of x as the number of hosts (amount of redundancy) needed to mask temporary churn. The y-axis shows the amount of redundancy restored on each repair that minimizes repair overhead.

In the above section we computed the object maintenance overhead analytically for a typical strategy using the permanent failures characteristics. We achieve an optimal bandwidth overhead if we choose to add small amounts of redundancy at each and repair and perform more repairs (consistent with similar results for a system model focused on permanent churn [58]). The optimal bandwidth for a strategy is $\frac{f(1+\sqrt{x/a})^2}{2d}$, hence the optimal bandwidth proportional to the amount of redundancy to cope up with the temporary churn and inversely proportional to the permanent churn. These two entities depends on the group of hosts on which an object is stored. In other words, if we have two groups of hosts where one group requires less amount of redundancy to cope up with temporary churn than the other group and both groups have similar amounts

of permanent churn then we could reduce the optimal object maintenance overhead by choosing the first group to store an object. Similarly, if one group of hosts has less permanent churn comparing with permanent churn of another group of hosts and both group of hosts has similar temporary churn then we could reduce the optimal object overhead maintenance by selecting the first group of hosts to store an object. We could get even lower the optimal object maintenance overhead if we could find a group of host that lower the temporary churn as well as permanent churn at the same time.

5.3.1 Exploiting Heterogeneity in Availability

The amount of redundancy required to cope with the temporary churn depends on the availabilities of hosts on which we store an object. We could reduce the redundancy required to cope with temporary churn by picking highly available hosts to store an object, which in turn reduces the object maintenance overhead according to the above analysis. This opportunity is possible only if the system has enough variation in host availabilities. For example, the availability of most hosts in the cluster environment [23, 7] is both similar and very high. In this environment picking any group of hosts would yield similar average availability. As a result, there would not be an opportunity to reduce object maintenance overhead by choosing hosts one way or another. However, host availability varies substantially in wide-area peer-to-peer file sharing systems, as observed in Overnet [4], KAD [76], and Gnutella and Napster [71].

Figure 5.4 shows the availability of hosts in KAD Network, OverNet and, PlanetLab. We define availability as the ratio of the amount of time a host is online and the difference between its first and last appearance in the trace. Note that the single session hosts (hosts that appear only once in the entire trace) will have availability 1.0 from this definition. The point (x, y) on a curve in the figure corresponds to $y\%$ of hosts have availability x or less. The curve labeled "KAD" shows the availability of KAD Network hosts from the trace described in the previous chapter, "OverNet" shows the availability of OverNet hosts from [4], and "PlanetLab" shows PlanetLab host availability from [74]. Over 45% hosts in the KAD network has availability 1.0, and are all

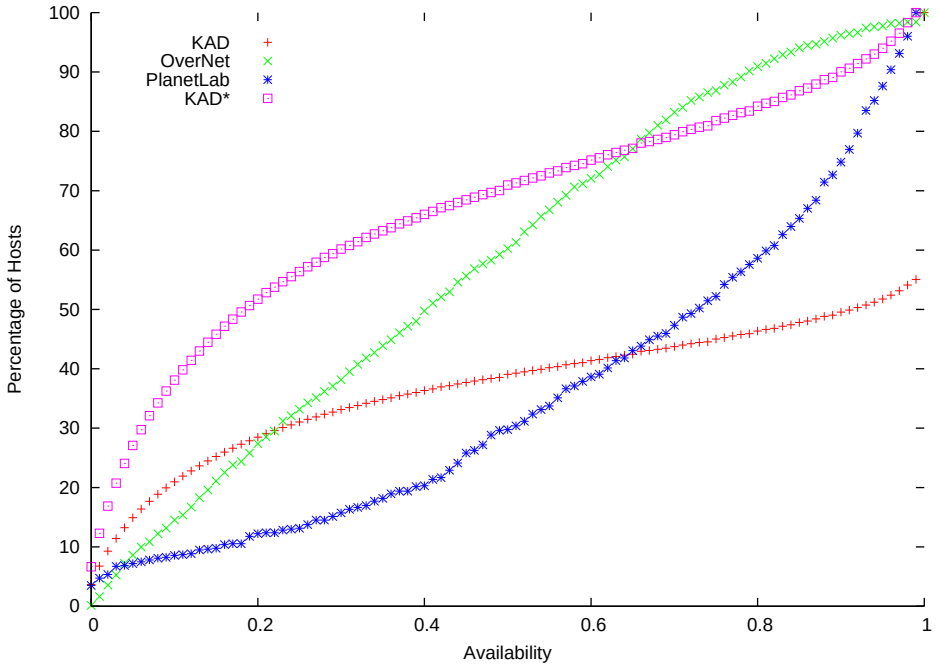


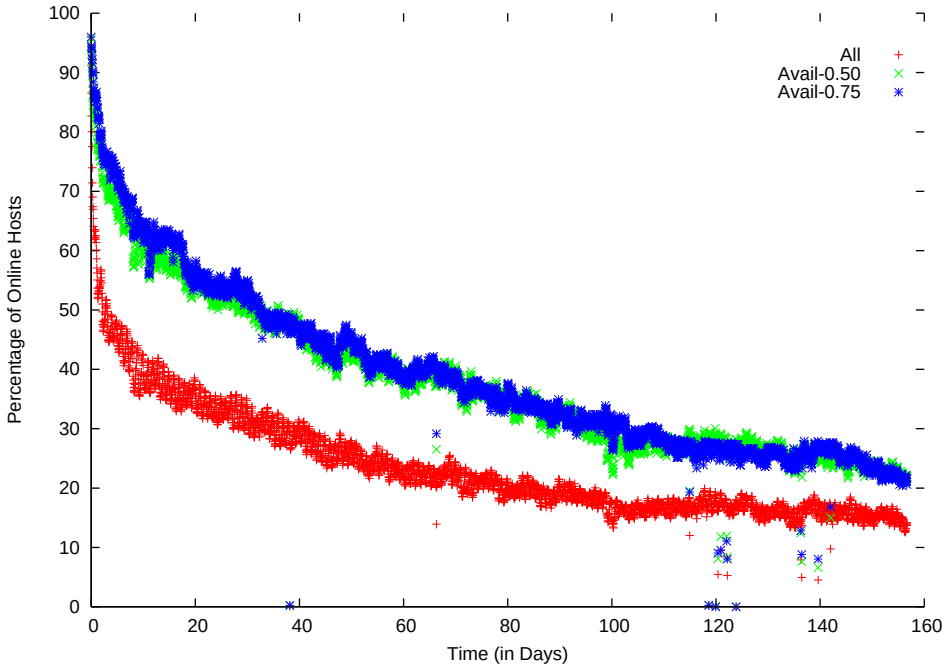
Figure 5.4: Host availability in KAD, OverNet and PlanetLab.

single-session hosts. The curve labeled "KAD*" is availability of KAD hosts with these single-session hosts removed.

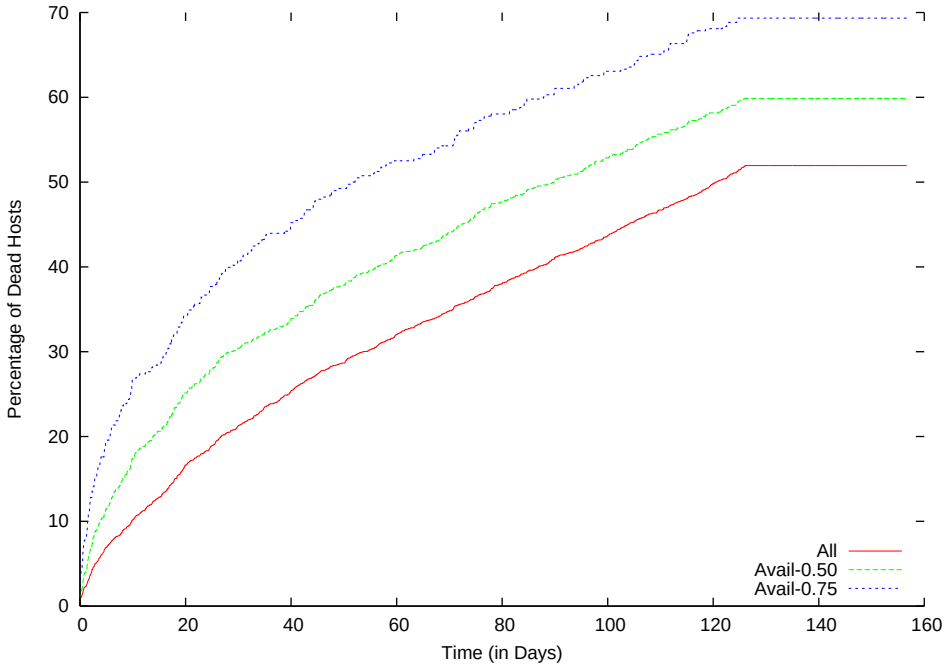
The PlanetLab hosts have greater availability than hosts in the file sharing system. In the KAD network, 60% of hosts have availability less than 0.3, whereas only 15% of PlanetLab hosts have availability less than 0.3. The file-sharing environments have more variety in hosts availability compared with PlanetLab. Hence, file sharing systems might exploit the heterogeneity in availability more than PlanetLab hosts.

Next we would like to evaluate how much benefit, if any, we can achieve if we select highly available hosts when we store an object in file sharing (KAD) and PlanetLab environments. We evaluated this opportunity by keeping track of the number of online hosts after selecting hosts based on their availability at a given point of time for both the KAD and PlanetLab traces.

Figure 5.5 shows the results for the KAD trace in two graphs. We selected different sets of hosts depending upon their availability: all the online hosts at the start of the trace ("All"); all online hosts at the start whose availability is greater than 0.5



(a) Hosts Churn



(b) Dead Hosts

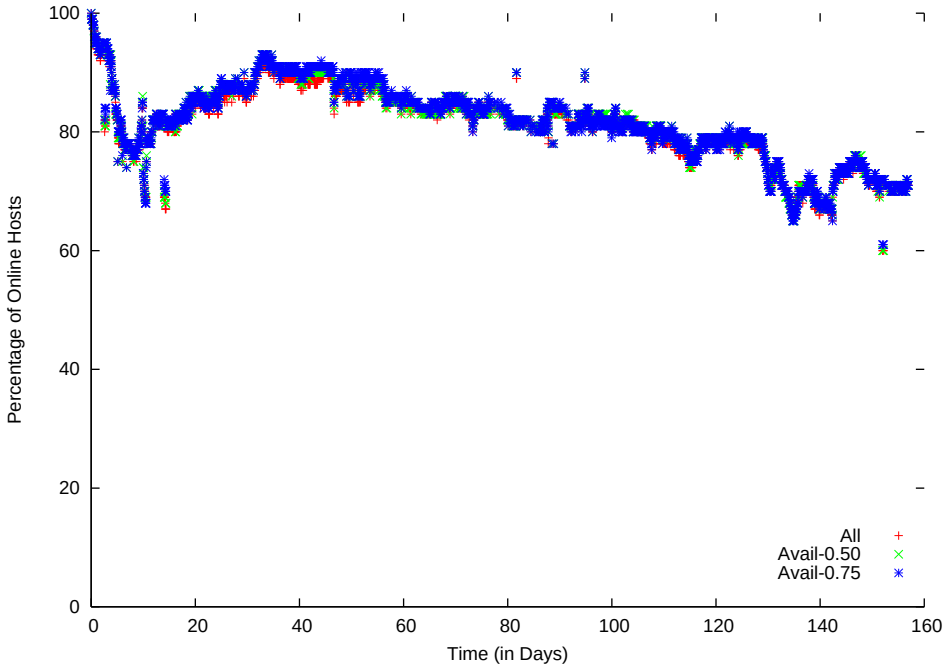
Figure 5.5: Exploiting host availability heterogeneity in KAD.

("Avail-0.5"); and all online hosts whose availability is greater than 0.75. A point (x, y) in Figure 5.5(a) indicates that $y\%$ of hosts among all the hosts selected at the start of the trace are online at xth hour in the trace. We also show the number of hosts that are dead from the groups over time in Figure 5.5(b) to measure the amount permanent churn in these groups.

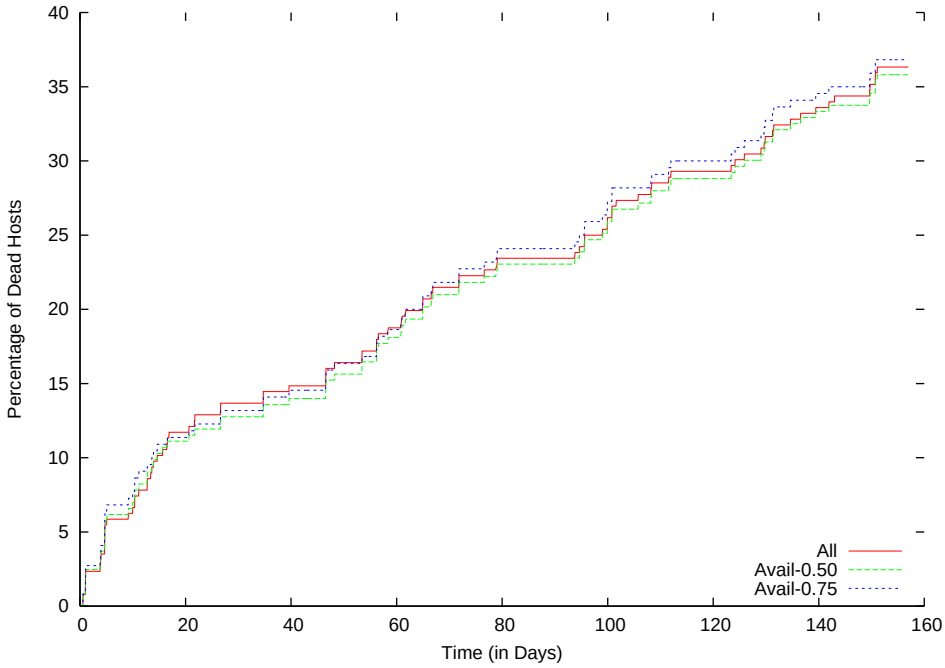
We make two observations from these results. First, choosing highly-available hosts reduces the amount of redundancy required to mask temporary failures. In this example, a redundancy of 8.3 masks failures when hosts are chosen randomly, but with highly-available hosts only a redundancy of 5 is needed. Although the amount of redundancy required to cope with failures is reduced when we choose highly-available hosts, there is no difference between the group of hosts whose availability is more than 0.5 and group of hosts whose availability is more than 0.75.

Second, the amount of permanent churn among these three groups is not the same as shown in Figure 5.5(b). We actually increased permanent churn by choosing highly-available hosts. From our analysis, this result should reduce the benefit we are getting from choosing highly available hosts. However, the benefits (from reducing the amount of churn required to cope with temporary churn) are outweighing the overhead due to more permanent churn when we choose the highly availability hosts. We also experimented with other availability thresholds (0.33 and 0.66), and found that the benefit of choosing highly available hosts levels off after the 0.5 threshold.

Figure 5.6 shows the result of the same experiment for the PlanetLab trace for the same period of time. The trace has 256 hosts that are online at the start; 243 hosts have availability more than 0.5, and 220 of these hosts have availability greater than 0.75. The amount of redundancy required to mask failures for all groups is similar because most of the online hosts have similar availability. The amount of permanent churn is also same for all three groups, as show in Figure 5.6(b). The improvements in reducing object maintenance overhead by exploiting heterogeneity in host availability is marginal in the PlanetLab environment.



(a) Hosts Churn



(b) Dead Hosts

Figure 5.6: Exploiting host availability heterogeneity in PlanetLab.

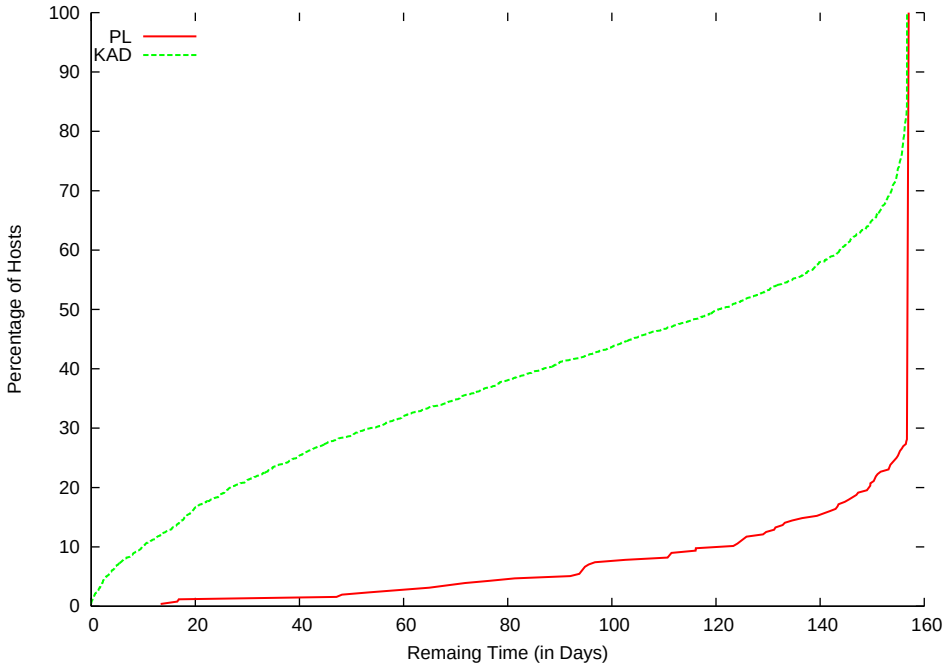


Figure 5.7: Amount of permanent churn in KAD and PlanetLab.

5.3.2 Exploiting Permanent Churn Heterogeneity

An object maintenance strategy can reduce bandwidth overhead to cope with failures by choosing a set of hosts that have less permanent churn. The benefits of doing so depend upon the degree of variation in permanent churn among hosts in the system. To determine the amount of permanent churn among the KAD and PlanetLab hosts, we calculate the remaining lifetime of hosts from the start of the traces. The remaining lifetime of a host is defined as the time difference between its last appearance in the trace and the start time of the group. If the host is online at the end of trace we consider trace end time as its last appearance. We selected all online hosts at a given point of time (7th day) and calculated the remaining lifetimes for these hosts.

Figure 5.7 shows the distributions of these lifetimes as CDFs. The x-axis is the number of days and the y-axis is the percentage of hosts. A point (x, y) on a curve indicates that $y\%$ of hosts have a remaining lifetime of x days or less. In the PlanetLab trace, 70% of hosts are online until the end of the trace. In KAD, only 12% of hosts are online at the end. This indicates that hosts in the KAD network experience more

variation in permanent churn for a group of hosts compared host groups in PlanetLab.

Host lifetime is another metric for characterizing the amount of permanent churn among hosts in the system. We define a host as leaving the system (a permanent failure) if it does not appear in the trace for 30 days. A host is newly joined the system if it does not appear in the first 30 days of the trace; we did not include any hosts that appear in the first 30 days in the trace because we do not know the exact joining time for these hosts.

We then divided the remaining trace into equal parts, and we consider only the hosts in the first half of the trace. This approach to measuring host lifetime is similar to the creation-based block life time as defined in [68]. With this approach, all hosts that appear in the first half but did not appear in the second half ([68] refers to this phase as end margin) has at least end margin minus 30 days lifetime. The observation phase lasts for 30 days and overall trace covers 163 days. Hence, if a host has not left the system by the end of the trace, we can say that this host has a lifetime of at least 73 days (we do not know when these hosts left in the last thirty days, so 163 - 30 (first phase) - 30 (observation phase) - 30 (last phase)). Using this perspective, in the KAD trace we observed 38,722 new hosts and, of those, 4,597 (12%) hosts remain alive at the end of trace. 2802 (7%) hosts have a gap (time between leaving the system and their next appearance) of more than 30 days.

We also measured the lifetimes for hosts in the PlanetLab trace. We looked at the 378 hosts that appeared in the first 30 days of the PlanetLab trace, and 257 of them remain at the end of 400-day trace; 145 hosts have a gap of 30 days or more.

Figure 5.8 shows the distributions of lifetimes for these hosts in the KAD and PlanetLab traces. The x-axis is the lifetime in days in log scale. A very large percentage of hosts (59%) have a lifetime less than one day, and 5% hosts have a lifetime is more than 100 days in the KAD trace. On the other hand, in the PlanetLab trace only 9% of hosts had a lifetime less than even one month, and 34% of hosts have a lifetime of more than one year. Overall, KAD hosts experience substantially more permanent churn than PlanetLab hosts, and we can expect that an object maintenance strategy that

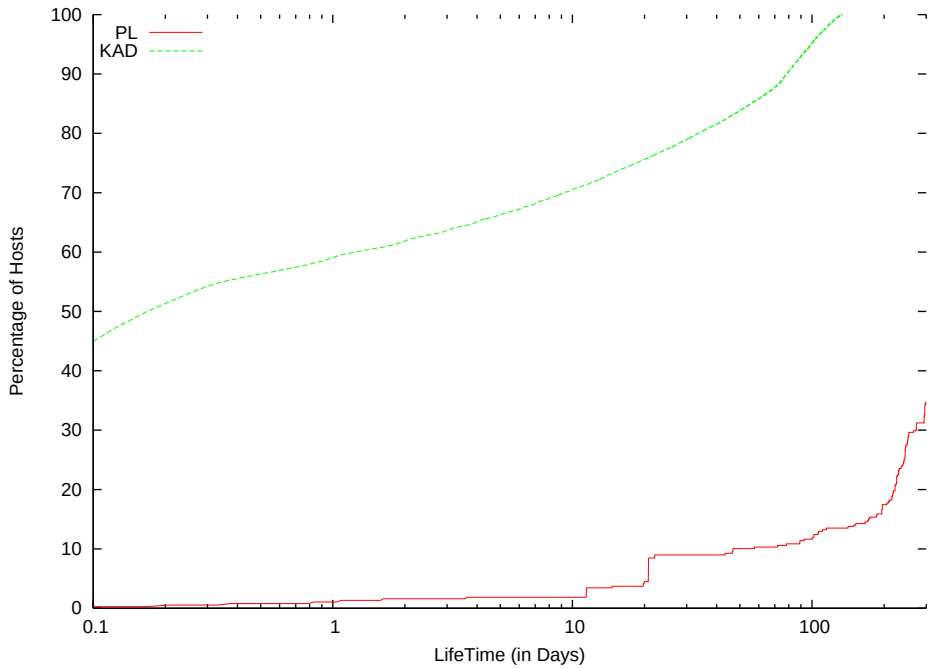


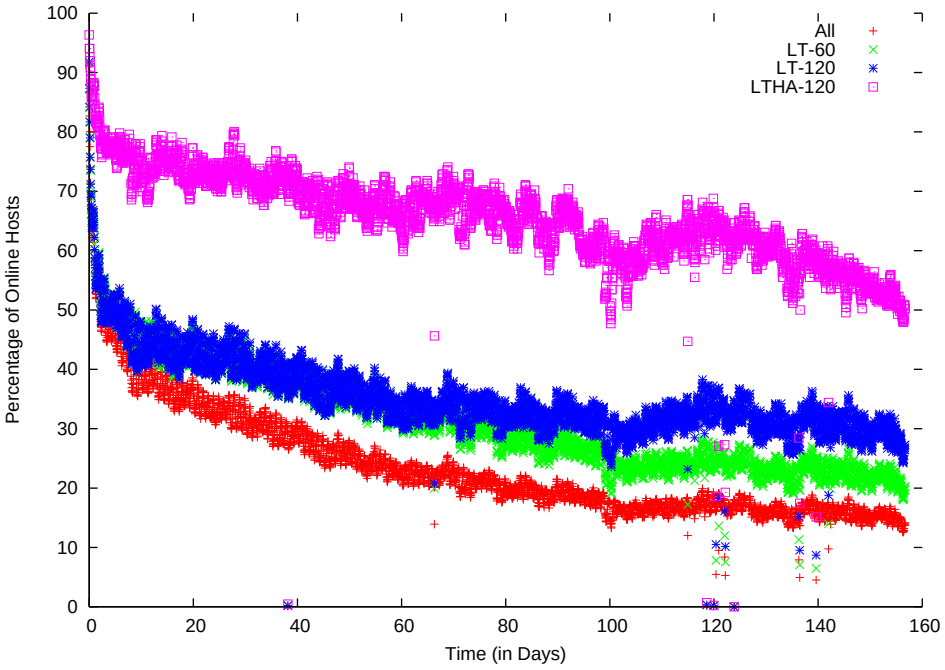
Figure 5.8: Host lifetimes in KAD and PlanetLab traces.

exploits heterogeneity in permanent churn will have more benefit for hosts in a KAD environment.

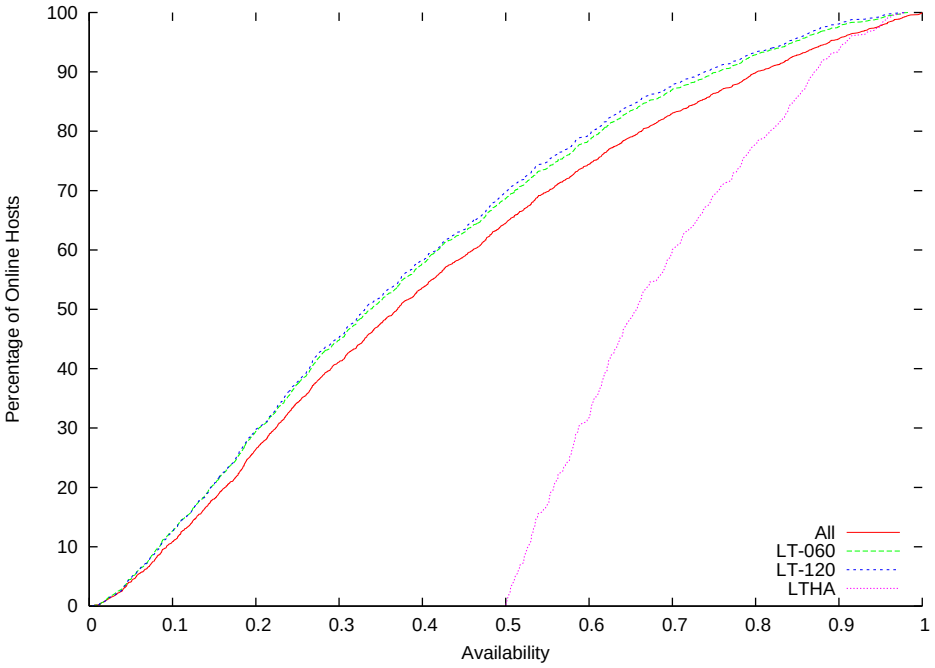
In previous sections, we see that hosts in a KAD environment experience substantial permanent churn. We could bias data placement towards high lifetime hosts (or high remaining lifetime) when we store object data to reduce object maintenance overhead when dealing with churn. We selected hosts starting on the 7th day whose lifetimes are more than 60 and 120 days in the KAD trace. We then tracked the number of those hosts online over time to determine the benefits of selecting hosts with high lifetimes over selecting hosts randomly.

We could expect benefits if we select high lifetime hosts over random hosts given that the two groups have similar availability distributions.

Figure 5.9(a) shows these results. The x-axis is the time in days and the y-axis is the percentage of hosts that are online at a given point of time. The “All” curve represents the group of all online hosts at the 7th day; the “LT-60” and “LT-120” curves represent the groups of online hosts whose lifetimes are at least 60 and 120 days, re-



(a) Hosts Churn



(b) Hosts Availability

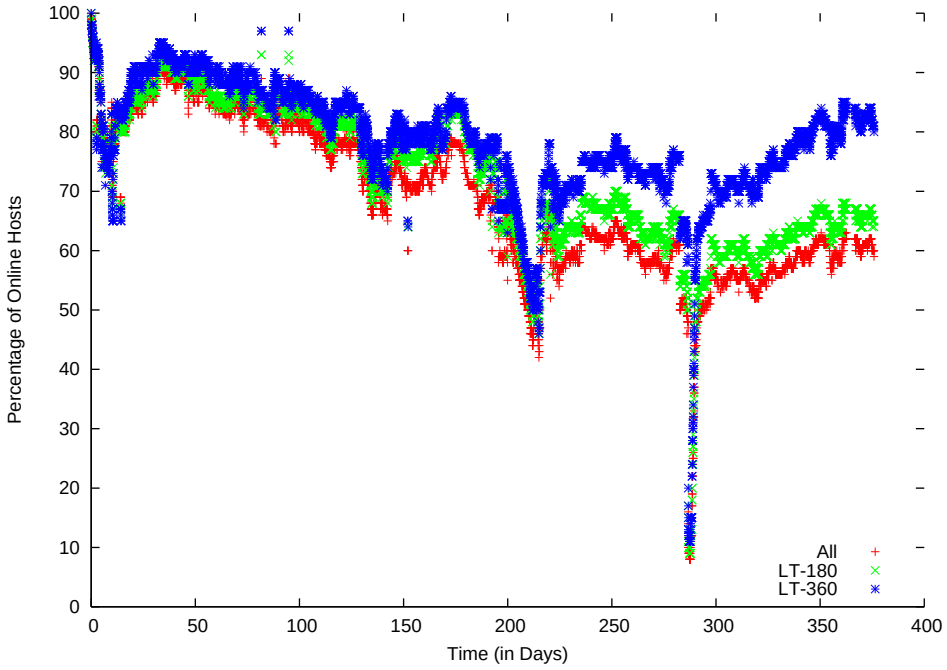
Figure 5.9: Exploiting permanent churn heterogeneity in KAD.

spectively; and the last curve labeled “LTHA-120” represents the group of online hosts whose lifetime is more than 120 days as well as whose availability is at least 0.5. Even though no hosts were permanently lost in the first 60 days, the “LT-60” and “LT-120” group availability gradually decreases along with the “All” curve. Both groups “LT-60” and “LT-120” reduced the amount of redundancy required to cope with permanent failures, from a factor of 7.7 (“All”) to 5.5 (“LT-60”) and 4 (“LT-120”). We can reduce the object maintenance cost even further if we choose hosts based on lifetime as well as availability. Placing object data on the “LTHA-120” group reduces redundancy from 4 for “LT-120” to just a factor of 2.

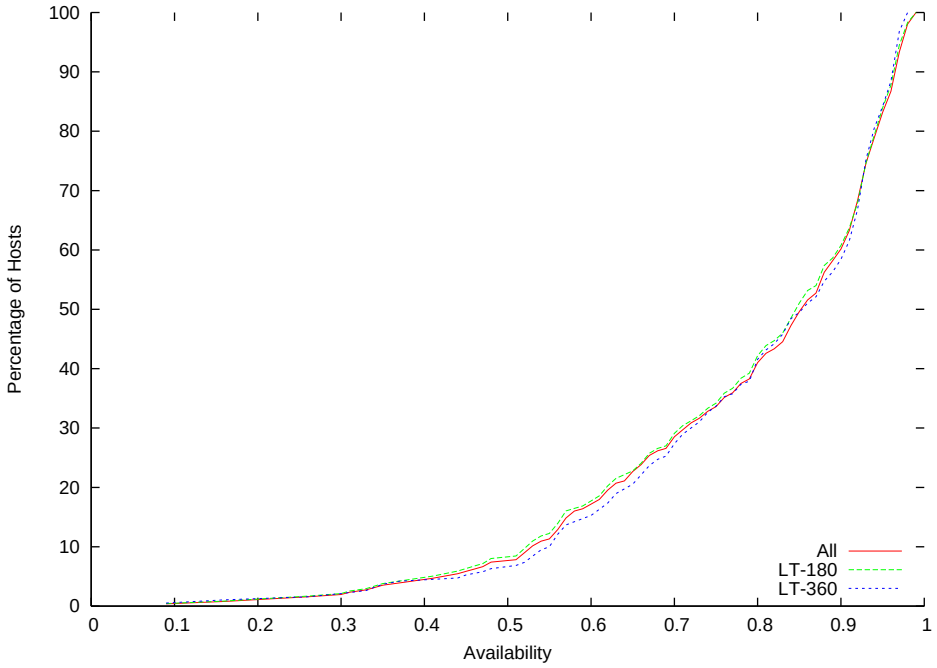
We also present the availability CDF of high lifetime hosts selected at the 7th day in Figure 5.9(b). The curve labeling is same to the Figure 5.9(a). The “LT-60” and “LT-120” groups have similar availability and slightly lower availability than the general population, the “All” curve. Overall, choosing high lifetime hosts over random hosts reduces the object maintenance costs even though the availability of high lifetime hosts is slightly less than availability of all online hosts at the selection time.

From this experiment, object maintenance overhead can be reduced if we choose higher lifetime hosts at object creation time in a file sharing P2P environment. We would like know whether the same is true for the PlanetLab environment, which have hosts with high lifetimes, little variance in lifetime, and high availability. We performed the same experiment: at a given point of time (on 30th day), choose all online hosts and all online hosts with higher lifetimes and keep track of the number of online hosts until the end of the trace.

Figure 5.10(a) shows the results of this experiment. The x-axis is the time in days and y-axis is the percentage of hosts that are online at a given point of time. The “All” labeled curve presents the group of all online hosts (256 hosts), the “LT-180” curve represents all online hosts (237 hosts) whose lifetime is more than 180 days, and “LT-360” represents all online hosts (190 hosts) whose lifetime is more than 360 days. The “All” group required 2.5 redundancy cope with the permanent failures, and the “LT-360” group required only 1.8 redundancy to cope with the failures. As expected the benefits



(a) Hosts Churn



(b) Hosts Availability

Figure 5.10: Exploiting permanent churn heterogeneity in PlanetLab

of selecting high lifetime hosts is small (only 28%) for reducing the object maintenance overhead. Figure 5.10(b) shows the availability CDF for these groups. The availability characteristics are similar for all groups.

5.3.3 Synthetic Trace

From Section 5.3, the object maintenance overhead in terms of bandwidth for a given amount of temporary churn (a) and permanent churn (d) is $\frac{f(1+N/a)(x+N)}{2Nd}$ bytes per second. Under our assumptions, we know from this formula that the minimum overhead is achieved by adding a small amount of redundancy at each repair. We would like to confirm this result from our trace by varying the amount of redundancy at repair time. We can cope with all permanent failures for the entire KAD trace time period if we store objects with a factor of 8 redundancy, even though our trace very long (163 days) (Figure 5.9(a)).

We can only increase the redundancy factor up to 8 for the KAD trace. Hence, we could not confirm the observation that using small redundancy factor at each repair yields an optimum object maintenance overheard using this trace because the trace is still not long enough. We could exploit the heterogeneity in temporary churn as shown in the Figure 5.5 or the heterogeneity in permanent churn as shown in the Figure 5.9 to reduce the object maintenance overhead. However, changing the temporary churn characteristics for a group also changes the characteristics of permanent churn, as shown in the Figure 5.5. Ideally we would like measure the relative benefit of changing the temporary churn characteristics for a group without changing the permanent churn characteristics. To accomplish this, we generated a synthetic trace based on the KAD traffic representing a longer period of time, while maintaining the similar temporary and permanent characteristics as the original KAD traffic. We make the temporary and permanent churn characteristics independent of each other in the synthetic trace, even though these two characteristics depend on each other in the original trace, so that we can study the effects of exploiting temporary churn or permanent churn individually.

In the synthetic trace, each day we introduce new hosts uniformly random

throughout the day as a parameter. In the original trace, this value is 600–1200 from Figure 4.5. We generate host availability and host lifetime using the availability CDF of the original KAD trace, as shown in Figure 5.4, and lifetime CDFs of original KAD trace, as shown in Figure 5.8. We randomly assign these values from these two CDFs to the new hosts. Each host comes into the system and stays for the number of hours that matches its availability. The host lifetime determines the number of days it stays in the system. We assumed all hosts are dead at the end of the original trace (actually there are 11% hosts are still alive in the original trace) and we added 0.16 availability for each host to ensure the synthetic trace has hosts online all the time in the system. The synthetic trace required 120 days to stabilize the number of hosts in the system because more hosts join initially. We generated a trace for two years (730 days) using 150 new hosts per day. We use this trace for the following experiments in this chapter. Although not a perfect reflection of the actual trace, it captures host behavior sufficiently to provide insight into relative performance if not absolute performance of object maintenance strategies.

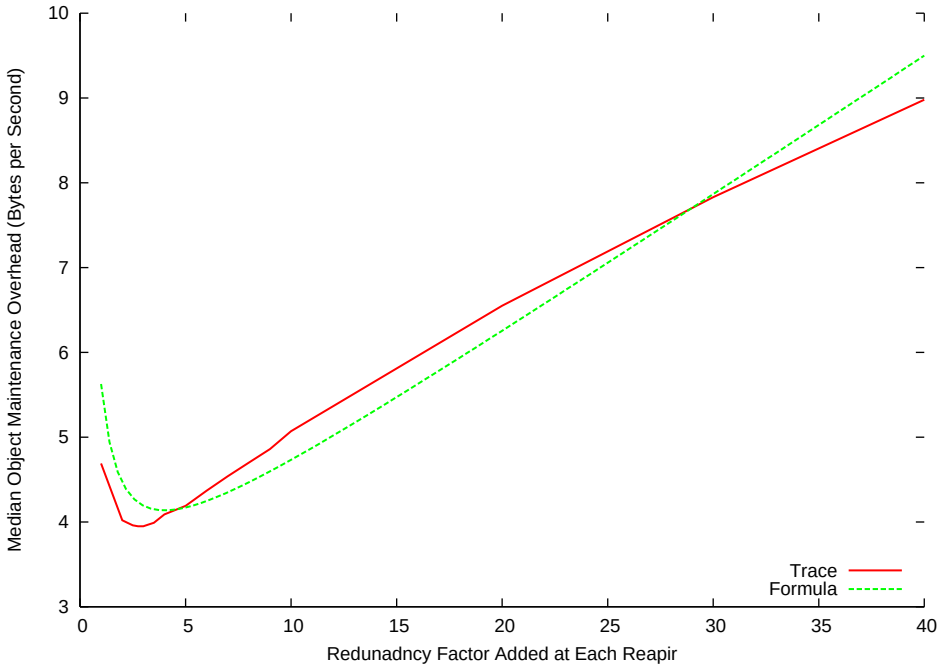
We wrote a simulator that implements the object maintenance strategy for a given trace with host arrival and departure events. This simulator does not simulate the network, and assumes everybody is connected and each host has infinite bandwidth. We did not simulate the network because we are more interested in measuring the overhead due to host churn. Hence, these overheads represents only a minimum value. We implemented an object maintenance strategy based on TotalRecall [5]. The strategy initially places the encoded object with a specified replication factor (the default is 3) onto selected hosts, and keeps track of the number of online hosts for each object. The user also specifies the number of hosts on which an object should be placed for one redundancy factor (parameter a in our object maintenance overhead formula from above; the default is 16). The strategy adds the specified amount of redundancy whenever the number online hosts for an object falls below a threshold (currently a redundancy factor of two (twice a), or 32 hosts). We inserted 1,024 objects at a specified point into the trace (default is 150 days) and then simulated the object maintenance strategy while varying the amount of redundancy added at each repair from one to forty. We calculated the number

of bytes we used to maintain these objects in the system and from this we calculated the bandwidth required to maintain an object by dividing the number of bytes by the time (we maintained objects for 250 days).

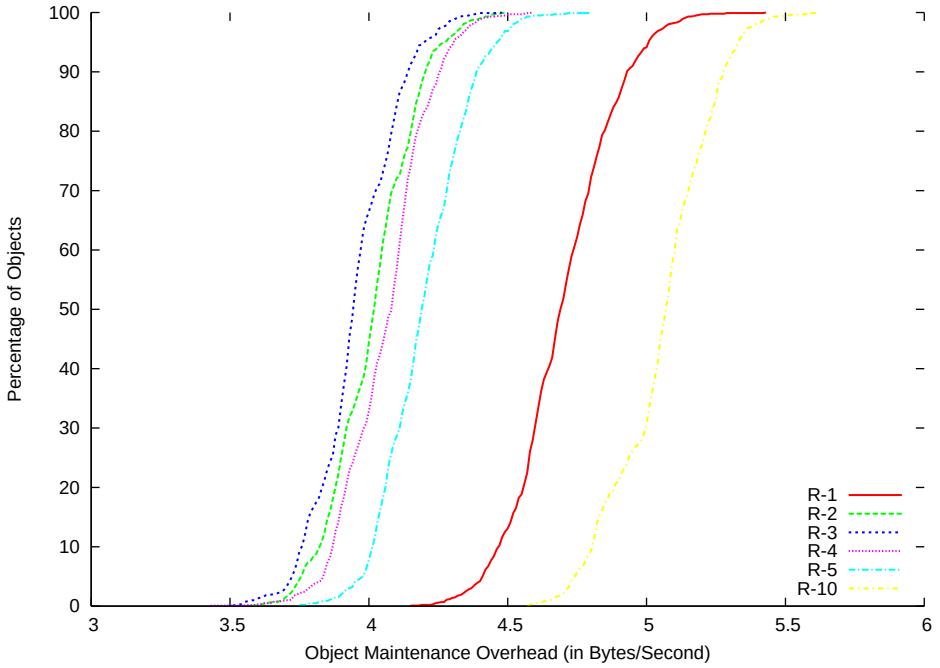
Figure 5.11 presents the results of this experiment in two graphs. Figure 5.11(a) shows the median object maintenance overhead of 1,024 objects while varying the redundancy added at each repair. The x-axis is the redundancy factor added at each repair and the y-axis is the overhead to maintain an object. We measured the object overhead from simulation and it is represented as "Trace" curve in the figure. The "Formula" curve represents the overhead calculated from the formula. We used the same object size and number of hosts to store an object with one redundancy factor. We measured the half death time from the trace and it is 880 hours.

The object maintenance overhead initially starts at infinite at a redundancy factor of zero, and gradually decreases as we increase the amount of redundancy added at each repair. The overhead reaches a minimum value, then it increases toward infinite if we increase the amount of redundancy further. Both curves show this trend and our formula-based calculation closely follows the simulation-based measurement. The measurement shows the minimum overhead is 3.95 at a redundancy factor of three, while the formula predicted the minimum overhead at 4.13 at a redundancy factor of four (overhead is within 4.7%).

Figure 5.11(b) shows the object maintenance overhead for 1,024 objects as a cumulative distribution using different redundancy factors at repair time. A point (x, y) on any curve indicates that $y\%$ of objects have overhead x or less. The "R-r" curves represent the strategy where we added r redundancy factor at repair. Most of the objects for a given strategy have similar overhead. The minimum overhead occurs at redundancy factor of 3. This experiment confirms the result from our analysis that adding a small redundancy factor yields an optimum overhead under the failures experienced in file sharing environments. Next, we would like to measure the benefits of biasing hosts with high availability and/or high remaining lifetime over selecting random hosts at repair time.

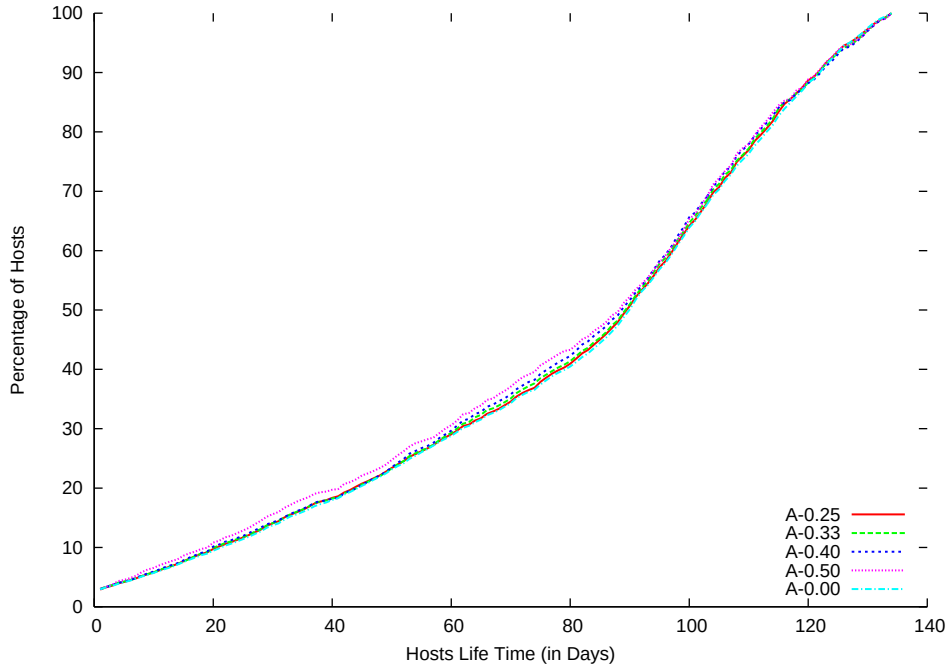


(a) Median Overhead

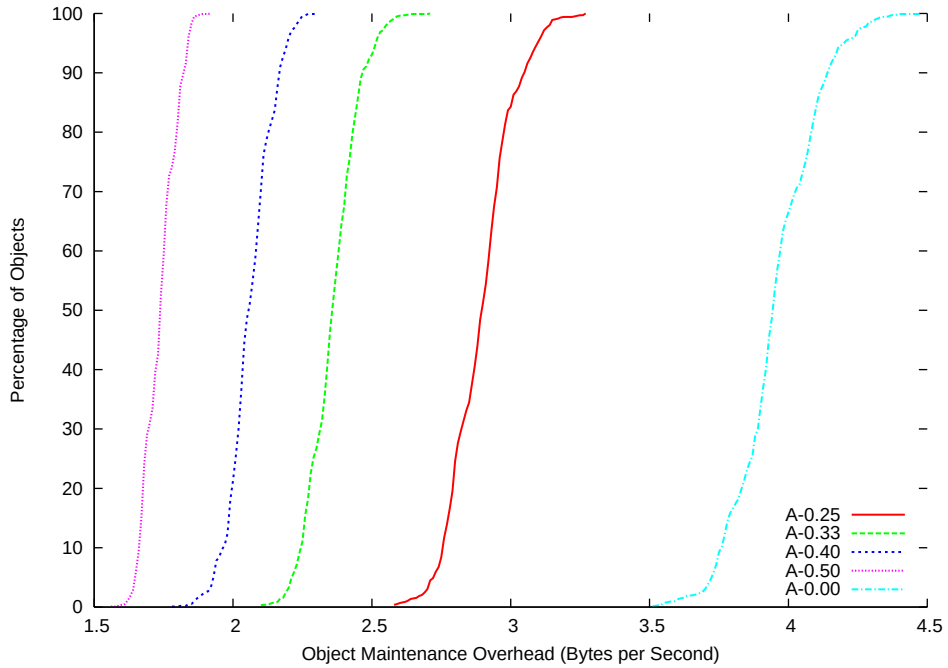


(b) Overhead CDFs

Figure 5.11: Object maintenance overhead.



(a) Object Lifetime



(b) Overhead

Figure 5.12: Object maintenance overhead when selecting highly-available hosts.

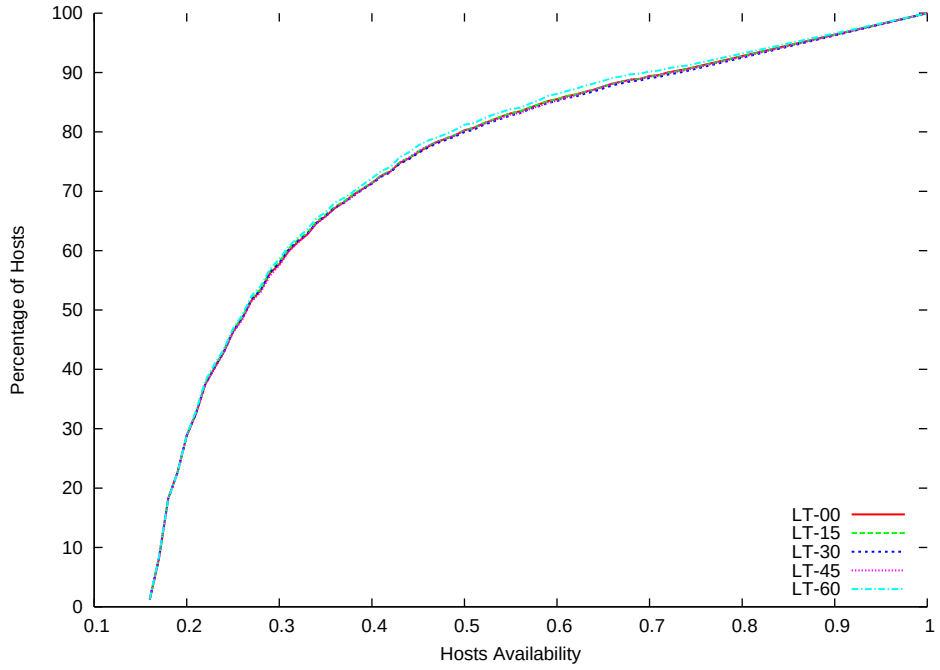
We implemented a strategy that greedily chooses highly-available hosts when selecting hosts on which to store redundant object data at repair time. This strategy selects hosts whose availability is more than a user-specified value. We varied this value from 0.25 to 0.5. We also measured the lifetime of selected hosts while adding the same number of hosts. We presented the object maintenance overhead with this modified strategy and the lifetime of selected hosts as CDFs in Figure 5.12.

Figure 5.12(a) shows all selected host's lifetimes as a cumulative distribution for each experiment. A point (x, y) on any curve indicates that $y\%$ of hosts have lifetime x or less days. The "A-a" curves show the object maintenance overhead when we selected hosts whose availability is more than or equal to a. This figure shows that we did not modify the hosts' permanent churn characteristics when biasing towards highly available hosts. This experiment shows the benefits of selecting highly-available hosts while all other factors remain constant.

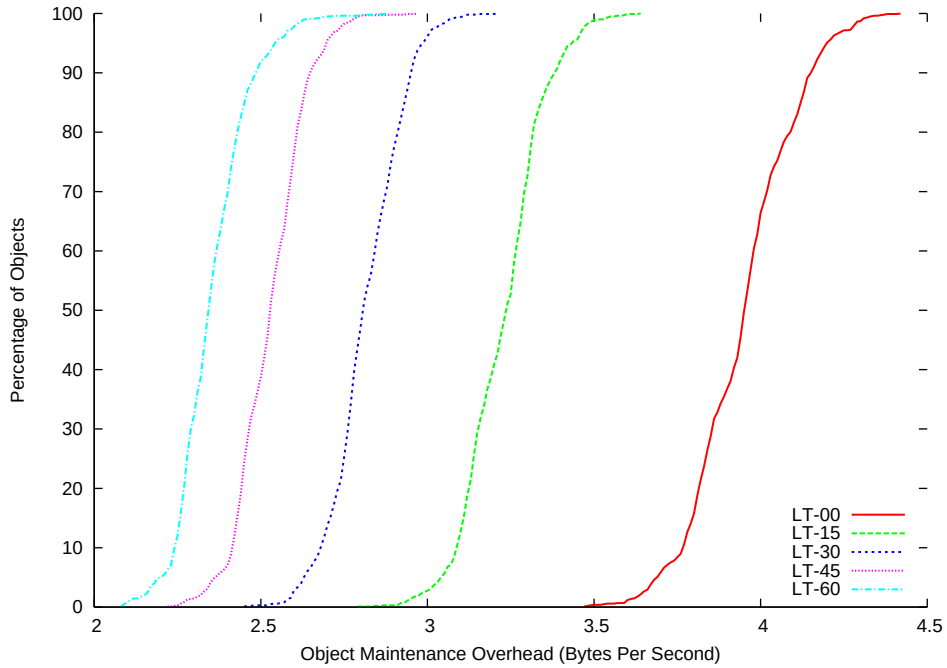
Figure 5.12(b) shows the distribution of object maintenance overheads per host for the various availability choices. The median overhead is reduced from 3.95 for unbiased case, to 1.74 when we selected hosts whose availability is at least 0.5. Overall, selecting hosts with high availability reduces the overhead around 56%.

Next, we fixed the availability of hosts while selecting high remaining lifetime hosts at each repair. We modified our simulator to implement this new strategy. We simulated this strategy by randomly selecting hosts whose remaining lifetime is more than 15, 30, 45 and, 60 days.

Figure 5.13 presents the object maintenance overhead along with the availabilities of the hosts. Figure 5.13(a) shows the availabilities of the selected host during the repair time in each experiment. A point (x, y) on an curve in this graph indicates that $y\%$ of the selected hosts have x availability or less. The "LT-l" curves are the CDFs of availability when selecting hosts that have at least l days of remaining lifetime. We also included a curve, "LT-00", where we do not bias selecting hosts based on remaining lifetime. All the curves have similar characteristics, indicating that we did not change the availability characteristics even though we biased selecting high remaining lifetime



(a) Objects Availability



(b) Overhead

Figure 5.13: Object maintenance overhead when selecting high remaining lifetime hosts.

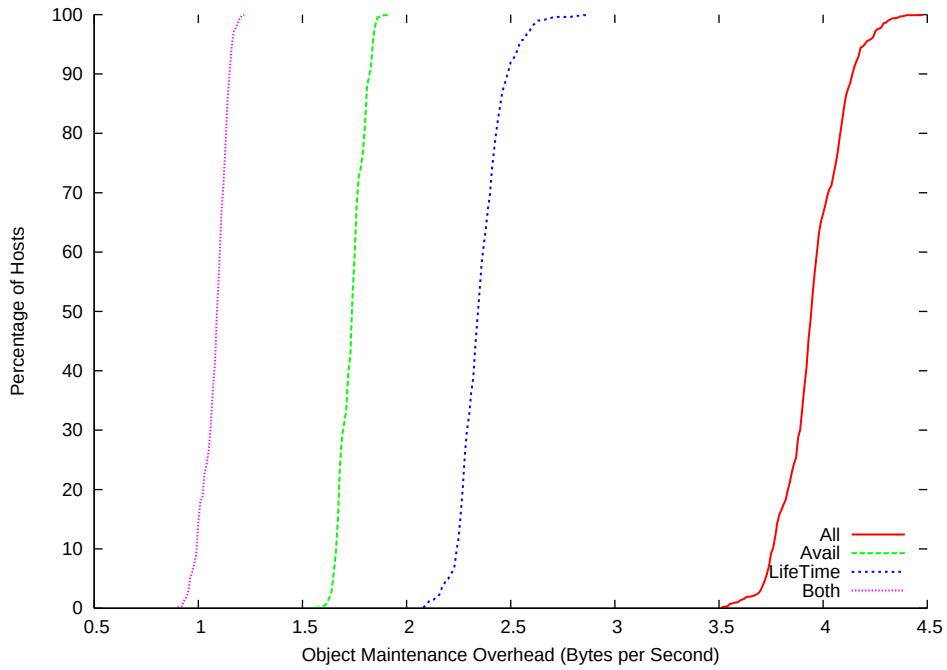


Figure 5.14: Object maintenance overhead when selecting both highly-available and high remaining lifetime hosts.

hosts. This experiment measures the benefits of selecting low permanent churn set of hosts at each repair without any changes in temporary churn characteristics.

Figure 5.13(b) shows the object maintenance costs for the strategy that biases the high remaining lifetime hosts at each repair. The curve labeling is the same as in Figure 5.13(a). As expected, the object maintenance overhead decreases as we increase the remaining lifetime of selected hosts. We did not see any further benefits by increasing the remaining lifetime any further than 60 days. This strategy decreased the median object maintenance cost from 3.95, when we did not bias towards long remaining lifetime hosts, to 2.35 (41% improvement) when we selected hosts that have more than 60 days of remaining lifetime. Overall, we will see benefit when we choose hosts to decrease their permanent churn while fixing the temporary churn, and this benefit is less than the benefit we would get if we fix the permanent churn and decrease the temporary churn of the selected hosts.

Finally, we changed the strategy to choose hosts that have both high availabil-

ity and high remaining lifetimes. We ran the simulator with this new strategy with host availability greater than 0.5 and remaining lifetime is more than 45 days. Figure 5.14 shows the results of this experiment. The "All" curve is the object maintenance overhead when we selected hosts without any bias; it is the same as the "LT-00" curve in Figure 5.13(b) and the "A-0.00" curve in Figure 5.12(b). The "Avail" curve is the object maintenance overhead when we selected hosts with high availability (more than 0.5 availability) and it is the same as the "A-0.5" curve in Figure 5.12(b). The "Lifetime" curve is the object maintenance overhead when we select hosts with high remaining lifetime (more than 60 days); it is the same as the "LT-60" curve in Figure 5.13(b). The "Both" curve is the object maintenance overhead when we selected hosts with high availability (more than 0.5) and high remaining lifetime (more than 45 days). This strategy provides the maximum benefit. As expected, when placing redundant data on hosts hosts with low temporary churn as well as low permanent churn, we can reduce the median object maintenance cost substantially: 1.09 bytes per second per host compared to 3.95 when randomly selecting among all hosts.

Overall, we can exploit the heterogeneity in temporary churn and permanent churn individually and combined in the file sharing environment to reduce object maintenance costs. We halve the median object maintenance cost when we exploit either the heterogeneity in temporary churn or permanent churn individually. We further reduce the median object maintenance costs to one fourth of the uniformly random case when we exploit heterogeneity in both temporary and permanent churn.

5.4 Capacity Constraints

A primary goal of object maintenance strategies is to reduce the bandwidth overhead of making data available and reliable in the face of churn. The strategies trade-off storage to achieve these goals, but they typically still strive to be storage-efficient. Previous work has evaluated the insertion failures rates of peer-to-peer storage systems as the system reaches capacity [15]. Even so, the constraints of both system and host

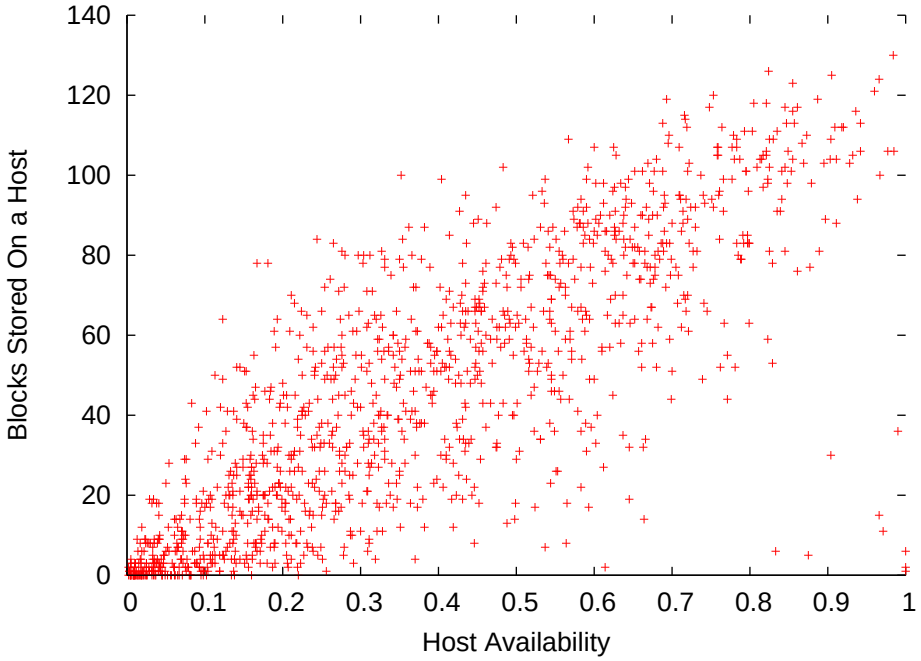


Figure 5.15: Host's used storage capacity with respect to host's availability.

storage capacity on object maintenance strategies and their overheads have not been given much attention, particularly as host availability and churn varies. In this last section, we motivate the need for maintenance strategies to also consider the constraints of capacity.

Object maintenance strategies that use indirect block placement with lazy repair, as in TotalRecall [5], randomly place object data on hosts in the system. A consequence of random placement is that it unbalances storage load in proportion to the distribution of host uptimes. To illustrate this effect, we simulated placing 1,024 1-MB objects into a system of 2,000 hosts paced evenly throughout a day. We then measured the number blocks each host stores at the end of the day when using a redundancy factor of three to store objects. We used the Overnet trace to simulate host arrivals and departures and determine host uptimes; the effect is similar in other environments, although the distribution of uptimes will change. Figure 5.15 shows the results of this experiment in a scatter plot. Note that each object is divided into encoded blocks (the parameter a in Section 5.3). In this experiment we divided objects into 32 blocks (if the replication

factor is 3, the systems stores 96 encoded blocks for a file). For each host in the system, the graph shows the number of blocks stored on the host according to its uptime. The diagonal cluster shows the correlation of host uptime and storage load (outliers, such as in the lower right corner, correspond to hosts entering the system as the simulation ends).

This effect is due to the random selection of hosts when placing object data — both when the object is initially created, as well as during repair. Hosts with longer uptimes will be selected more often than hosts with shorter uptimes and, as a result, store more blocks over time. Storage is a comparatively plentiful resource, but hosts still have finite capacities (particularly if only a small fraction of storage on a host is available for use by other hosts). Consequently, over time hosts with longer uptimes will fill to capacity faster than hosts with shorter uptimes.

This effect has both positive and negative consequences. On the positive side, it is a natural mechanism by which the system will favor storing object data on hosts with good availability. Favoring hosts with long uptimes reduces the amount of storage, and hence bandwidth overhead, required to mask temporary churn. (Note, though, that if permanent failures are independent of host uptimes, this effect does not reduce the rate of repairs.) This effect is somewhat similar to the natural formation of stable cores of supernodes in unstructured networks, also due to the bias of host uptimes [77].

On the negative side, object maintenance strategies will need to explicitly respect host storage capacities when making placement decisions. Strategies that use indirect placement can adapt to storage capacity constraints by simply removing hosts at capacity from random selection. Respecting capacities has two consequences: (1) since indirect placement biases towards hosts with higher uptimes, data for newer objects gets placed on hosts with lower uptimes; and as a result, (2) repair overhead increases more than linearly as system storage grows towards capacity.

To illustrate this effect, we repeat the object maintenance simulation and extend it so that all hosts have equivalent capacity constraints. Figure 5.16 shows the results of capacity constraints on repair overhead. The y-axis shows the average ob-

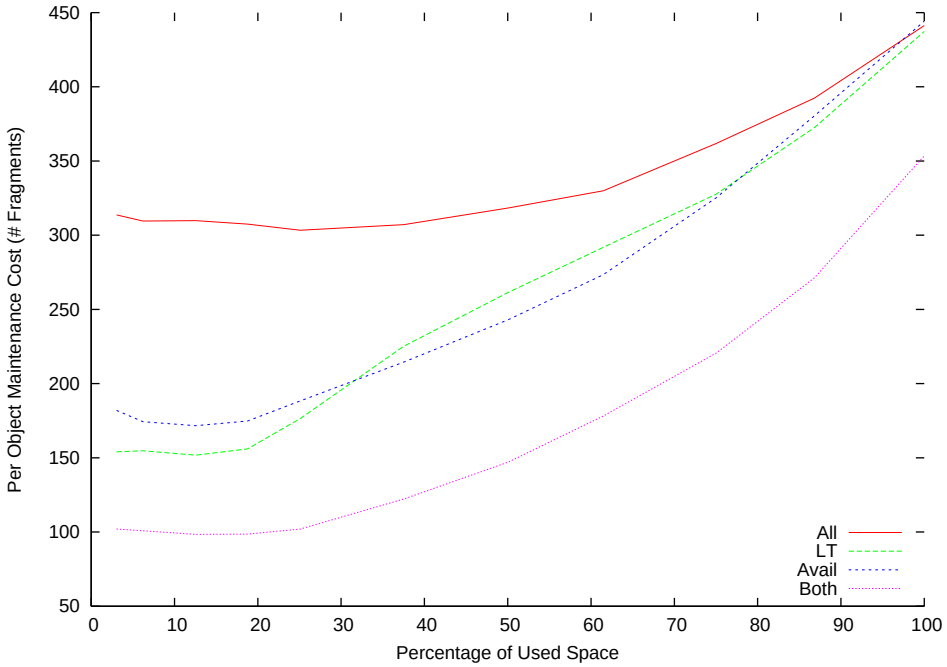


Figure 5.16: Normalized repair cost as a function of used capacity.

ject maintenance overhead per object. The x-axis varies capacity as a percentage of used system capacity. The "All" curve is the object maintenance strategy when we selected hosts without any bias. The "Avail" curve is the object maintenance strategy that greedily selects hosts with highest availability. The "Lifetime" curve is the object maintenance strategy that greedily selects hosts that have the highest remaining lifetime. The "Both" curve is the strategy that selects hosts with the highest product of availability and remaining lifetime. The strategies are greedily selecting the "good" hosts. The per object maintenance cost increases as the system utilization increases for all strategies. The difference between the strategies that bias selection and the standard policy is highest when the system is lightly loaded (up to 20%), and this difference slowly diminishes as the system fills up. The strategy that biases either the availability or remaining lifetime approaches the overhead of the standard strategy when the system capacity is around 90%. However, the strategy that biases availability as well as remaining lifetime performs better than random even when the system is at full capacity, though the difference between these two strategies is reduced from 66% to 33%.

Object maintenance strategies that eagerly repair on successors (e.g., CFS) or leaf sets (e.g., PAST) will not exhibit this bias since hosts store data relative to their position in the ID space, and not relative to uptime. Such placement implicitly assumes that nearby hosts in the ID space can always store data given to it, although in practice some hosts in the middle of a successor list, for instance, may reach capacity before other hosts. One approach to this problem is to use replica diversion [15] to introduce a level of indirection, effectively implementing indirect placement. Alternatively, successor placement can skip successors at full capacity when propagating redundant data down the successor list, effectively treating those successors as "failed" hosts with respect to placement. Doing so, however, will likely require either direct or indirect bookkeeping to track which successors store redundant object data, evolving such placement strategies from direct towards indirect placement.

5.5 Conclusion

In this section, we revisit object maintenance in peer-to-peer systems, focusing on how temporary and permanent churn impact the overheads associated with object maintenance. Overall, we emphasize that the degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. These differences impact the source of overheads for object maintenance strategies.

A system with permanent failures much repair lost redundancy regardless of the initial redundancy. We formulated the object maintenance bandwidth overhead using observations from our KAD trace. Adding small amounts of redundancy at each repair reduces the bandwidth overhead. Interestingly, the amount of redundancy to restore on a repair that minimizes bandwidth overhead depends upon the degree of temporary churn in the system, but not on the degree of permanent churn; the bandwidth overhead certainly scales with the rate of permanent churn, but the rate does not affect the choice of how much redundancy to repair. We validated these results from the simulation based on the trace collected in the KAD file sharing environment.

Hosts in file sharing environments exhibits wide variation in terms of resources (availability and lifetime) that they contribute to the system. The system can exploit these variations to reduce object maintenance overhead. In the KAD environment, at least, we show that object maintenance overhead is reduced to 58% of the standard random policy if the system selects highly-available hosts at each repair. Similarly, the object maintenance overhead reduces to 49% of the random policy if the system selects long-lived hosts at each repair when hosts capacity is unlimited. The system further reduces the object maintenance overhead by combining highly-available and high lifetime hosts. We also confirmed these results from simulations using a synthetic trace derived from the actual trace.

Finally, we highlight additional practical issues object maintenance strategies must face, in particular dealing with storage capacity constraints. Experience with deployments of peer-to-peer storage systems will undoubtedly raise a number of additional practical constraints that object maintenance strategies will need to address.

5.6 Acknowledgment

Chapter 5, in part, is a reprint of the material as it appears in the Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS) 2006, by Kiran Tati and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Summary and Future Work

To improve the lookup performance of distributed hash tables (DHTs) while minimizing update overhead, we propose and evaluate the use of three kinds of *hint caches* containing route hints: *local hint caches* store direct routes to neighbors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space. Combined, these hint caches achieve routing performance that approaches the aggressive performance of one-hop schemes, but with an order of magnitude less communication overhead on average.

To design improved object maintenance strategies, we need to understand the availability and lifetime characteristics derived from long-term (at least a hundred days) traces of peer-to-peer systems with host arrival and departure events. As a result, we performed a 6-month study of the KAD peer-to-peer file sharing system to measure such failure characteristics of a portion of the hosts in the system. We evaluated our measurement methodology to capture sufficiently complete data, and demonstrated that peer-to-peer systems like KAD have substantial heterogeneity in host availability and lifetime characteristics. We found that hosts join and leave the system at roughly a constant rate, the permanent failure rates of hosts are constant and independent of host availability and lifetime, and that a substantial percentage of hosts (around 40%) have lifetimes longer than 150 days.

We then revisit object maintenance strategies in peer-to-peer systems, focusing on how temporary and permanent churn impact the overheads associated with object maintenance. Overall, we emphasize that the degrees of both temporary and permanent churn depend heavily on the environment of the hosts comprising the system. These differences impact the source of overheads for object maintenance strategies.

Using trace-driven simulation of a peer-to-peer storage system and our trace of hosts in the KAD network, we confirm our analytic results for determining an optimal amount of redundancy to use at each repair. Further, we explore variants of this object maintenance strategy that bias the placement of redundant data on those hosts with high availability, high lifetimes, or both. We then use simulation to show that, using our trace of KAD hosts as input, peer-to-peer storage systems can reduce object maintenance overhead to 58% of a random strategy by placing on high-available hosts, 49% of random by placing on long-lived hosts, and only 33% of random by placing on both highly-available and long-lived hosts. Finally, we highlight additional practical issues object maintenance strategies must face, in particular dealing with storage capacity constraints.

In summary, this dissertation shows that peer-to-peer systems based on distributed hash tables (DHTs) can take advantage of heterogeneous availability and lifetime characteristics of hosts in the system.

6.1 Future Work

Experience with deployments of peer-to-peer storage systems will undoubtedly raise a number of additional practical constraints that object maintenance strategies will need to address. Hosts have other characteristics that vary as well, including access bandwidth and capacity. Measuring both characteristics of peer-to-peer host populations at large scale remain open and interesting problems. Adapting object maintenance strategies that take these heterogeneous aspects of peers could further improve the task of maintaining object data with high availability and reliability. Also, other possible op-

timizations for improving object maintenance strategies include placing redundant data closer to the hosts using it, and adapting the amount and kinds of redundancy to the application workloads that access the data.

Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, and M. Theimer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of 5th OSDI Symposium*, Dec. 2002.

[2] http://www.amule.org/.

[3] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, University of California, San Diego, 2002.

[4] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[5] R. Bhagwan, K. Tati, Y. C. Cheng, S. Savage, and G. M. Voelker. Totalrecall: System support for automated availability management. In *ACM/USENIX NSDI Symposium*, Mar. 2004.

[6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of the HotOS*, May 2003.

[7] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. of SIGMETRICS*, June 2000.

[8] D. Bruijn. A combinatorial problem. *Koninklijke Nederlandse akademie van wetenschappen*, 49:758–764, 1946.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[10] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of IEEE Infocom*, Mar. 2003.

[11] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *proceedings of Second Workshop on Hot Topics in Networks*, Nov. 2003.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[13] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[14] http://www.pdos.lcs.mit.edu/chord/.

[15] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001.

[16] P. Druschel and A. Rowstron. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[17] F. D. Emil Sit and J. Robertson. Usenetdht: A low overhead usenet server. In *3rd International Workshop on Peer-to-Peer Systems*, Feb. 2004.

[18] http://www.emule-project.net/home/perl/general.cgi?l=1.

[19] http://research.microsoft.com/sn/farsite/.

[20] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *ACM/USENIX NSDI Symposium*, Mar. 2004.

[21] M. J. Freedman, K. Laskhminarayanan, and D. Mazières. Oasis: Anycast for any service. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[22] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. Re: Reliable email. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[24] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *Proceedings of IEEE Infocom*, March 2004.

[25] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proceedings of IEEE Infocom*, March 2005.

[26] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[27] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[28] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.

[29] J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *29th International Conference on Very Large Data Bases*, Sept. 2003.

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

[31] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. The phoenix recovery system: Rebuilding from the ashes of an internet catastrophe. In *9th Workshop on Hot Topics in Operating Systems*, May 2003.

[32] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[33] http://en.wikipedia.org/wiki/kad_network.

[34] D. R. Karger and M. Ruhl. Simple, efficient load balancing algorithms for peer-to-peer systems. In *3rd International Workshop on Peer-to-Peer Systems*, Feb. 2004.

[35] J. M. Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC*, May 2000.

[36] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[37] S. S. Krishna P. Gummadi and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *2nd Internet Measurement Workshop*, Nov. 2002.

[38] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS 2000*, Nov. 2000.

[39] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *Proceedings of IEEE Infocom*, March 2005.

[40] J. Li and F. Dabek. F2f: Reliable storage in open networks. In *5th International Workshop on Peer-to-Peer Systems*, Feb 2006.

[41] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[42] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth efficient management of dht routing tables. In *2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[43] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.

[44] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[45] B. T. Loo, S. Krishnamurthy, and O. Cooper. Distributed web crawling over dhts. Technical Report UCB/CSD-4-1305, UC Berkeley, 2004.

[46] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.

[47] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *4th USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[48] P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[49] J. McCaleb. http://www.overnet.com/.

[50] M. McNett. https://sysnet.ucsd.edu/group.

[51] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. S. Wallach, X. Bonnaire, P. Sens, J.-M. Busca, and L. Arantes-Bezerra. Post: A secure, resilient, cooperative messaging system. In *9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, May 2003.

[52] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *IEEE Workshop on Internet Applications*, June 2003.

[53] http://mldonkey.sourceforge.net/main_page.

[54] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[55] http://oceanstore.cs.berkeley.edu/.

[56] K. Park, V. S. Pai, L. Peterson, and Z. Wang. Codns: Improving dns performance and reliability via cooperative lookups. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, Mar. 2004.

[57] http://www.pdl.cmu.edu/pasis/.

[58] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *Proceedings of IEEE Infocom*, April 2006.

[59] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[60] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[61] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *proceedings of ACM SIGCOMM*, Aug. 2004.

[62] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[63] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Feedtree: Sharing web micronews with peer-to-peer event notification. In *4th International Workshop on Peer-to-Peer Systems*, Feb. 2005.

[64] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *proceedings of ACM SIGCOMM*, Aug. 2001.

[65] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *2nd USENIX Conference on File and Storage Technologies*. USENIX, Mar. 2003.

[66] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. of the USENIX Annual Technical Conference*, June 2004.

[67] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *Proc. of the IPTPS*, February 2005.

[68] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *USENIX Annual Technical Conference*, June 2000.

[69] M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer-to-peer networks. In *USENIX Annual Technical Conference*, June 2003.

[70] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[71] S. Saroiu, K. P. Gummadi, and S. D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia System*, 9(2):170–184, 2003.

[72] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A case for informed internet routing and transport. *IEEE Micro*, 19, 1999.

[73] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[74] J. Stribling. http://pdos.csail.mit.edu/ strib/pl_app/.

[75] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Overcite: A distributed, cooperative citeseer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[76] D. Stutzbach and R. Rejaie. Characterizing churn in peer-to-peer networks. Technical Report CIS-TR-05-03, University of Oregon, June 2005.

[77] D. Stutzbach, R. Rejaie, and S. Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *Proc. IMC 2005*, June 2005.

[78] K. Tati and G. M. Voelker. On object maintenance in peer-to-peer systems. In *5th International Workshop on Peer-to-Peer Systems*, Feb 2006.

[79] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, 2001.

[80] C. UCSD. http://activeweb.ucsd.edu/.

[81] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from dns. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, Mar. 2004.

[82] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the IPTPS*, March 2002.

[83] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of the International Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[84] J. J. Wylie, M. Bakkaloglu, V. Pandurangan, M. W. Bigrigg, S. Oguz, K. Tew, C. Williams, G. R. Ganger, and P. K. Khosla. Selecting the right data distribution scheme for a survivable storage system. Technical Report CMU-CS-01-120, Carnegie Mellon University, May 2001.

[85] A. Yip, B. Chen, and R. Morris. Pastwatch: a distributed version control system. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, May 2006.

[86] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

[87] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*, June 2001.