DHT Routing Tradeoffs in Network with Churn

Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, Thomer M. Gil
{jinyang, strib, rtm, kaashoek, thomer}@csail.mit.edu

Abstract

This paper studies the performance of DHT algorithms with continuous changes in membership. A DHT protocol incurs communication costs to keep up with these changes in order to maintain its ability to route lookups efficiently. The paper explores a unified framework for evaluating cost and performance. Communication costs are combined into a single cost measure (bytes), and performance benefits are reduced to a single latency measure. This approach correctly accounts for background maintenance traffic and timeouts during lookup due to stale routing data; it also correctly leaves open the possibility of different preferences in the lookup time/communication cost tradeoff. All DHTs have parameters that can be tuned to provide better performance with different workloads; these parameters obscure more fundamental performance differences, and our approach is to explore the parameter space looking for the envelope of best performance.

The paper analyzes three DHT protocols, comparing the efficiency with which different protocols obtain performance benefit from each byte of communication cost, and providing insight into how different design choices and protocol structures affect this efficiency.

1 Introduction

The design space of DHT protocols is large. All designs are similar in that hosts forward lookups for keys through routing tables that point to other hosts. Algorithms differ in the amount of state they keep, from $O(1)$ with respect to the number of hosts [5, 6] to $O(\log n)$ [9, 7, 10, 8] to $O(\sqrt{n})$ [4] to $O(n)$ [3]. They also differ in the techniques used to find low-latency routes, in the way alternate paths are found after encountering dead intermediate hosts, in the expected number of hops per lookup, and in choice of parameters such as the frequency with which they check other hosts for liveness.

How is one to compare these protocols in a way that separates incidental details from more fundamental differences? Most evaluations and comparisons of DHTs have focused on lookup hop-count or latency in unchanging networks. Static analysis, however, tends to unfairly favor protocols that keep large amounts of state, since they pay no penalty to keep the state up to date, and more state usually results in lower lookup hop-counts and latencies.

This paper presents a framework for evaluating DHT algorithms in the face of joining and leaving hosts, in a way that makes it easy to compare tradeoffs between state maintenance costs and lookup performance. As an example, the Chord [9], Tapestry [10], and Kelips [4] lookup algorithms are compared with this framework. All three have similar overall performance, but the differences in the way they behave under stress are revealing. For example, a structured ID-space is an advantage when churn is high, since the structure helps guide lookup re-tries and helps focus limited maintenance traffic on the most important routing table entries. On the other hand, when lookups are relatively common, the benefits of a careful routing table maintenance protocol are lost, since the lookup traffic itself suffices to keep routing tables up to date.

2 A Model for DHT Performance Evaluation

There are multiple measures for both *cost* and *performance*. Cost has often been measured as the amount of per-host state. However, an analysis should also include the cost of keeping that state up to date (which avoids timeouts), and the cost of exploring the network to search for nearby neighbors (which allows low-latency lookup). A unified cost metric should reflect consumption of the most valuable system resource; we measure it as the number of bytes of messages sent. This reflects a judgment that network capacity is a more limiting resource than host memory or CPU time.

Lookup performance has often been measured with hop-count, latency, success rate, and probability of timeout. We use lookup latency as the unified performance metric relevant to applications. Lookup hop-count can be ignored except to the extent that it contributes to latency. The cost of trying to contact a dead host during a lookup is accounted for as a latency penalty equal to the round trip time to the dead host, an optimistic simulation of the cost of a timeout before the host pursues the lookup through an alternate route. The DHT is required to keep trying alternate routes indefinitely, which effectively means that failed lookups are converted into long latencies.

Protocol parameters tend to obscure differences among protocols, since the parameters may be tuned for different workloads. A key challenge is to eliminate differences due solely to parameter choice, so that remaining performance differences reflect fundamental design choices rather than accidental parameter choices. We evaluate each protocol over a range of parameter values, outlining a performance envelope from which we can extrapolate an optimal cost/performance tradeoff curve.

Finally, the workload mix may determine the results of comparisons between two DHTs. A workload that has relatively high churn to lookup ratio may favor a DHT that keeps a smaller routing state, while low churn may favor protocols that accumulate large routing tables. The challenge is to explore the workload spectrum and identify the crossover points.

3 Protocol Overviews

This paper compares three DHT protocols: Chord [9], Tapestry [10], and Kelips [4]. The protocols are implemented in a simulator in as much detail as possible, based on published papers and communication with the authors of each DHT. All protocols takes a DHT key as argument and pursues the lookup until the host responsible for the key is contacted.

Chord: In this implementation of Chord, the host whose ID most closely precedes a key (the *predecessor*) is responsible for the key. The *base* b of the ID space is variable for this implementation: each host keeps $(b-1)\log_b(n)$ fingers whose IDs lie at integer powers of $\frac{b-1}{b}$ fraction of ID circle away from itself. Any host whose ID lies within the range $(\frac{b-1}{b})^i$ and $(\frac{b-1}{b})^{i+1}$ can be used as the i^{th} finger. This flexibility allows for Proximity Neighbor Selection [2]. Each host also keeps a *successor list* of the s hosts.

A Chord host periodically pings all current fingers to check their liveliness. If a finger i doesn't respond, the host issues a lookup request for the key $(\frac{b-1}{b})^i$ fraction of the ID circle away, yielding host f. The host retrieves f's successor list, and uses the successor with the lowest latency as the level i finger. A host stabilizes its successor list by periodically retrieving and merging its successor's successor list. Figure 1 lists the Chord parameters

Number of successors	4 – 32
Finger base	2 – 128
Finger stabilization interval	2s – 32s
Succlist stabilization interval	1s – 32s

Figure 1: Chord parameters

Base	2, 4, 16
Stabilization interval	2s – 32s
Number of backup hosts	1 – 4
Backup hosts used in a lookup	1 – 4

Figure 2: Tapestry parameters.

that are varied in the simulations.

Tapestry: A Tapestry host ID can be viewed as a sequence of l base-b digits. A routing table has l levels, each with b entries. Hosts in the m^{th} level table share a prefix of length $m-1$ digits, but differ in the m^{th} digit. Each entry may contain up to c hosts, sorted by latency. The closest is the entry's *primary neighbor*; the others serve as *backup neighbors*.

Hosts forward a lookup message for a key by resolving successive digits in the key (*prefix-based routing*). When no more digits can be resolved, an algorithm known as surrogate routing determines exactly which host is responsible for the key (see [10] for the details of this algorithm).

For lookups to be correct, at least one neighbor in each entry must be alive. Tapestry periodically checks the liveness of each primary neighbor, and if the host is found to be dead, the next closest backup in that entry (if one exists) becomes the primary. If a primary is found to be dead in the course of a lookup (i.e. an RPC to that host timed out), a backup may be used instead; the number of backups used in such a way is a configurable parameter in our implementation. Figure 2 lists the parameters varied in the simulations.

Kelips: Kelips hosts divide themselves into k groups, where k is chosen to be the square root of the number of hosts. A host's group is its ID mod k. Each host's routing table contains an entry for every other host in its group, and "contact" entries for a few hosts from each of the other groups. Thus a host's routing table size is a small constant times $\sqrt{n}$.

In the simulator's variant of Kelips, lookups are only

Gossip interval	0.125s – 24s
Group targets	1 – 64
Contact targets	1 – 16
Group ration	1 – 64
Contact ration	1 – 16
Contacts per group	2 – 8
Times a new item is gossiped	0 – 4
Routing entry timeout	5s – 40s

Figure 3: Kelips parameters.

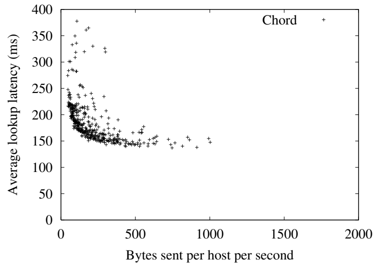

Figure 4: Chord cost/performance tradeoff for the churn-intensive workload (1024 hosts).

defined for host IDs. The originating host executes a lookup for a key by asking a contact in the key's group for the IP address of the target key's host, and then contacting that host. If that fails, the originator tries routing the lookup through randomly chosen hosts.

Hosts periodically gossip to learn about new hosts. A host may also learn about another host in the course of a lookup. Hosts time out routing table entries that aren't refreshed for a while. Hosts learn RTTs and liveness information from each RPC. Hosts preferentially route lookups through low-RTT contacts.

Figure 3 lists Kelips' parameters. Targets are the number of randomly chosen hosts a host sends information to every gossip interval, and rations are the number of hosts mentioned in the gossip messages. Contacts per group is the maximum number of contact entries per group in a host's routing table; if it has value c, then the size of each host's routing table is $\sqrt{n} + c(\sqrt{n} - 1)$.

4 Evaluation

We simulated two scenarios, with *lookup intensive* and *churn intensive* workloads. Hosts issue 100 and 0.1 lookups per second, respectively. Each host crashes and rejoins at exponentially distributed intervals with mean 100 seconds. The simulated network consists of $1,024$ hosts, with inter-host latencies derived from Internet measurements among a set of $1,024$ Internet DNS servers [1]. The average inter-host round trip time is 100 milliseconds. All simulations run for 200 seconds. The lookup-intensive workload uses only 256 nodes.

For each of the graphs below, the x-axis shows the average number of bytes sent per second from each host, including both lookup traffic and routing table maintenance. The y-axis shows the average lookup latency, including timeout and lookup retry penalties. The communication cost is the sum of the sizes of all packets sent divided by the number of hosts and the length of the simulation. A packet is considered to consist of 20 bytes for headers plus 4 bytes for each host ID or IP address mentioned in the packet.

4.1 Churn-Intensive

Chord: Figure 4 shows the performance of Chord in the churn intensive scenario. The best average lookup latency achieved is 140 milliseconds with average 2.1 lookup hops at base 64 or higher.

Base has the most significant impact on both lookup latency and communication cost. A base b Chord achieves $\log_b(n)$ hops with $(b-1)*\log_b(n)$ fingers. For example, when base is 64 and successor list contains 16 hosts, each of a host's 64 fingers knows 16 other hosts from its successor list. Thus, most lookups can complete within 2-hops,

The cost of finger stabilization is a major part of the communication cost. This is approximately proportional to finger stabilization timer. We find that longer stabilization timer saves a large amount of communication cost while only increasing lookup latencies slightly due to timeouts with reasonably amount. For example, at base 32, communication cost range from 141 bytes with 32 second timer to 527 bytes/host/second with 2 second timer with 30 milliseconds saved in timeouts. In comparison, the stabilization cost for successor list is pretty small; even at 2 second interval, only $30 - 35$ bytes/host/second is generated. However, slow stabilization does have an adverse impact on the correctness of lookups which result in retries and significantly higher average latency.

Tapestry: Figure 5 shows the performance of Tapestry in the churn intensive scenario. The best latency achieved is approximately 180 milliseconds (Note

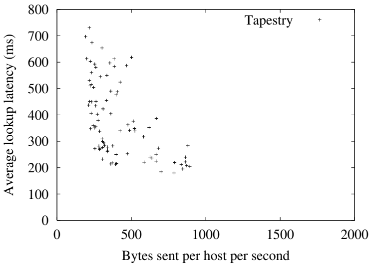

Figure 5: Tapestry cost/performance tradeoff for the churn-intensive workload (1024 hosts). Note that the y-axis scales up to 800 milliseconds.

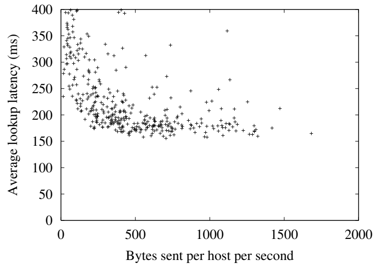

Figure 6: Kelips cost/performance tradeoff for the churn-intensive workload (1024 hosts).

that figure 5 is plotted with a bigger y-axis scale than other graphs). This is higher than the best achieved by Chord. This is due to the fact that we have not explored a wider range of the base parameter. The knee of the curve occurs slightly above 200 milliseconds, with a bandwidth cost of 250 bytes per host per second. The parameters around the knee have an identifier base of 2 or 4 and many backups per entry, allowing all of the backups to be used if a lookup fails.

Similar to Chord, the base parameter in Tapestry has significant impact on latency and communication cost; a routing table with $b * \log_b(n)$ hosts achieves $\log_b(n)$ lookup hops. Tapestry routing table stabilization timer also has dramatic impacts as it affects *b*oth lookup correctness and the likelihood of lookup timeouts. Tapestry depends on its routing table for correctness, therefore to avoid costly re-tries of incorrect lookups, Tapestry should check the liveness of all its primary neighbors relatively frequently. Therefore, Tapestry is not as effective with low communication cost.

Kelips: Figure 6 shows the performance of Kelips in the churn intensive scenario. The best average lookup latency achieved is about 160 milliseconds; this corresponds to one RPC to a contact in the target group, and then an RPC to the target host. The first RPC typically has low latency since any of the target group's members can be used, and Kelips chooses the one with lowest RTT. The second RPC has the 100 ms average expected for talking to a random host. The latencies above 160 ms are due to timeouts and extra hops due to contacting dead hosts; they are not due to higher-latency hops.

Most of the left-right spread in the graph is due to the inter-gossip interval parameter. The other important parameter is the routing entry timeout interval. Shorter timeouts tend to delete entries for hosts that are actually live, and the missing entries increase average hop count.

In extreme cases Kelips can suffer from high lookup failure rates. The problem is that the DHT overlay becomes close to disconnected, so that even after a large number of retries, Kelips cannot find a path to a host that is alive. These situations happen when the churn rate is high compared to the rate at which the parameter values cause gossip to move information; for example, if the inter-gossip interval is high, the number of gossip targets is low, and the routing entry timeout is low.

Churn Summary: These analyses lead to the following observations about DHT design for relatively high-churn environments:

- The main symptom of high churn is high lookup latency due to timeouts for dead hosts. Timely stabilization is required to avoid timeouts. Selection of low-RTT routing table entries is relatively less important.

- A host should be able to look at a key and *know* whether it is responsible for that key. This allows a lookup algorithm to treat routing table entries as hints, since their correctness can be checked. That in turn makes it more reasonable to stabilize the routing table entries relatively slowly, and yet still perform correct lookups.

- An ID space that orders all IDs, along with routing that makes monotonic progress in that ordering, has an advantage over an un-ordered ID space. The ordering allows a host to choose the best remaining alternate route each time it encounters a dead host. Without such an order, a lookup algorithm may have to resort to random search to recover from dead hosts.

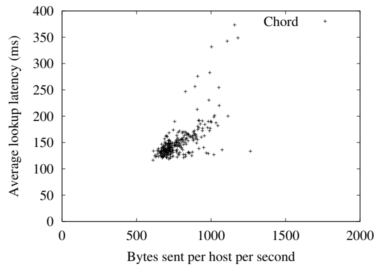

Figure 7: Chord cost/performance tradeoff for a lookup-intensive workload and 256 hosts.

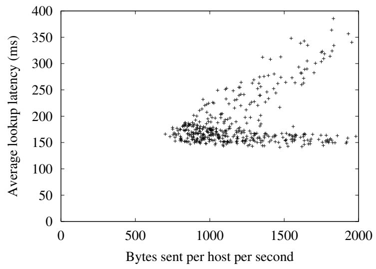

Figure 8: Kelips cost/performance tradeoff for the lookup-intensive workload and 256 hosts.

- Algorithms in which each host is required to have routing table entries that point to specific other hosts (or points in the ID space) tend to retain connectivity better under high churn.

4.2 Lookup-Intensive

Chord: Figure 7 shows the performance of Chord in the lookup intensive scenario. With each lookup forwarded, a Chord host is able to check the validity of routing table entry; at high lookup rates this is a more effective way than stabilization to keep fingers up to date. At 117 milliseconds, the minimum latency is lower than in the churn-intensive scenario. This is due to a combination of fewer timeouts and a more up-to-date successor list which helps to sample lowest latency fingers.

The upper tail of figure 7 corresponds to parameters that cause extra hops, either due to a smaller routing table or more timeouts. Since most of the traffic is due to the lookups, more hops also cause more traffic, so the points lie along a diagonal line.

Kelips: Figure 8 shows the performance of Kelips in the lookup intensive scenario. The best latencies are better than in the churn-intensive scenario for the same reasons as Chord. The reasons for the graph's fish-tail shape are also the same as for Chord.

These analyses lead to the following observations about DHT design for relatively lookup-intensive environments:

- The communication cost is dominated by the lookups themselves, so (in contrast to the churn-intensive scenario) the costs of stabilization are relatively less important.
- Stabilization has relatively little impact on latency, because the protocols learn more from the lookups than they do from stabilization.

Tapestry: Tapestry implementation used too much memory to execute the lookup intensive workload.

5 Conclusions

This paper presents a unified framework for studying the cost vs. lookup latency tradeoffs in different DHT protocols, and evaluates Chord, Tapestry, and Kelips in that framework. We find that protocols with a notion of lookup correctness and routing structure are more robust and can operate with less communication cost. In a lookup intensive workload, DHTs can use the lookup traffic to help stabilize large amounts of state.

References

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

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

[3] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, May 2003.

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

[5] Frans Kaashoek and David Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the Second IPTPS*, 2003.

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

[7] Peter Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc.*

1st International Workshop on Peer-to-Peer Systems, March 2002.

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

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

[10] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. To appear.