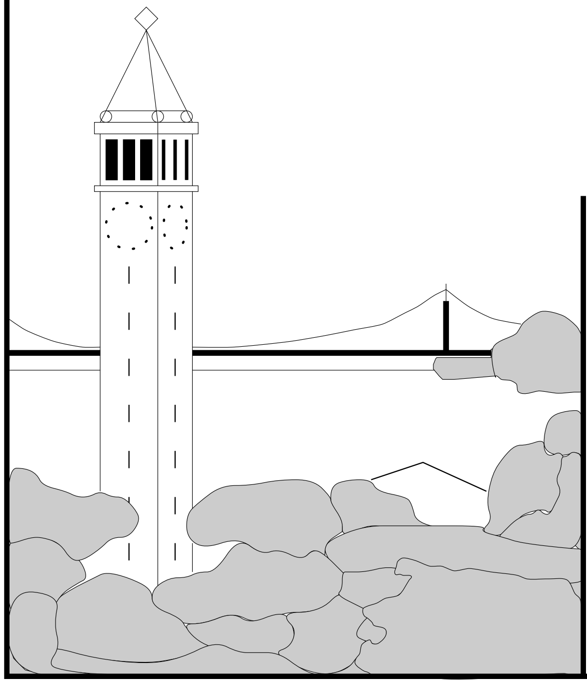
The Cost of Inconsistency in DHTs

Shelley Zhuang *Ion Stoica* *Randy Katz*
{shelleyz, istoica, randy}@eecs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley



Report No. UCB/CSD-5-1394

June 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This technical report is supported by grant number DABT63-98-C-0038

The Cost of Inconsistency in DHTs

Shelley Zhuang   Ion Stoica   Randy Katz
{shelleyz, istoica, randy}@eecs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley

June 2005

Abstract

Distributed Hash Tables (DHTs) support a hash-table-like *lookup* interface: given a key, it maps the key onto a node. One of the crucial questions facing DHTs is whether lookups can route correctly in a dynamic environment where the routing state is inconsistent. The routing state may become inconsistent when a node falsely thinks a failed neighbor is up (false negative), when a node falsely removes a neighbor that is up (false positive), when a new node joins but has not been fully incorporated into the routing state (join hole), and when a node leaves and disrupts the routing state (leave recovery). In this paper we analyze the cost of inconsistency in DHTs. Using the example of Chord, we evaluate the cost of each type of inconsistency on lookup performance. We find that the routing invariant has a first order impact on the relative cost of different types of inconsistencies. In addition, the cost of false negatives is higher than that of false positives, which means it is more important to ensure timely failure detection than a low probability of false positives. The cost of join holes and leave recoveries are also higher than that of false positives due to the routing invariant of Chord. We also make conjectures about the cost of inconsistency in other DHTs based on their routing invariants.

1 Introduction

In the last few years, distributed hash tables (DHTs) have rapidly evolved and emerged as a promising platform to deploy new applications and services in the Internet [12, 7, 9, 10]. DHTs support a hash-table-like *lookup* interface: given a key, it maps the key onto a node.

Each node in the network maintains a routing table containing information about a few other (typically $O(\log n)$) nodes. Because the routing state is distributed, a node communicates with other nodes to perform a lookup. The routing state may become inconsistent as nodes continuously join and leave the network, and cause lookups to fail. Thus, the design of algorithms to construct and maintain consistent routing state under continuous joins and leaves is an important and fundamental issue.

The routing state may become inconsistent when a node falsely thinks a failed neighbor is up (false negative), when a node falsely removes a neighbor that is up (false positive), when a new node joins but has not been fully incorporated into the routing state (join hole), and when a node leaves and disrupts the routing state (leave recovery).

The rate at which inconsistencies are generated and corrected in a DHT depends on its component algorithms. False negatives are generated by leaves and corrected by failure detection algorithms; false positives are generated by failure detection algorithms and corrected by recovery algorithms; join holes are generated by joins and corrected by join algorithms; and leave recoveries are generated by leaves and corrected by recovery algorithms. Minimizing one type of inconsistency may come at the cost of another type of inconsistency. For example, more aggressive failure detection algorithms reduce the duration of false negatives but may increase the rate of false positives.

The relative cost of false negatives, false positives, join holes, and leave recoveries on lookup performance also depend on the routing invariant of a DHT. The routing invariant specifies the information in the routing state that must be correct in order to guarantee correct routing of lookups. As we illustrate in the context of Chord, the routing invariant has a first order impact on the relative cost of inconsistencies on lookup performance.

Our work here focuses on identifying lookup pathologies that can result from routing state inconsistencies and analyzing the relative cost of different types of inconsistencies on lookup performance, rather than proposing new algorithms to minimize false negatives, false positives, join holes, and leave recoveries. Understanding the impact of routing inconsistencies on lookup performance will provide important insights on how to make certain choices and tradeoffs when designing future algorithms to construct and maintain consistent routing state.

The rest of the paper is organized as follows. In Section 2, we identify the possible outcomes of a lookup, and illustrate the lookup pathologies that can result from routing state inconsistencies in Chord. We present our simulation methodology in Section 3. In Section 4, we evaluate the cost of each type of inconsistency on lookup performance in Chord. In Section 5, we make conjectures about the relative cost of false negatives, false positives, join holes, and leave recoveries in several other DHTs based on their routing invariants. We discuss related work in Section 6, and conclude in Section 7.

Inconsistency	Generated by	Corrected by
False negative	Leaves	Failure detection algorithms
False positive	Failure detection algorithms	Recovery algorithms
Join hole	Joins	Join algorithms
Leave recovery	Leaves	Recovery algorithms

Table 1: Routing State Inconsistencies.

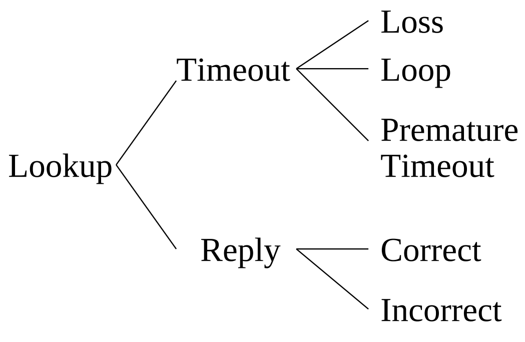


Figure 1: Breakdown of possible outcomes of a lookup.

2 Design

2.1 Routing State Inconsistencies

There are four types of routing state inconsistencies: false negative, false positive, join hole, and leave recovery. A false negative occurs in a node's routing state when a neighbor fails but the node has not yet detected the failure of the neighbor. False negatives are detected and corrected by failure detection algorithms, and the failed neighbor is then removed from the node's routing state. A false positive occurs when a node falsely removes a neighbor that is up. False positives are generated by failure detection algorithms when the algorithms make false detections, and they are corrected by recovery algorithms. A join hole occurs when a new node joins but has not been fully incorporated into the routing state. Join holes are corrected by join algorithms. A leave recovery occurs after a node detects the failure of a neighbor but before it is corrected by recovery algorithms. The different types of inconsistencies, how they are generated and corrected are summarized in Table 1.

2.2 Possible Outcomes of a Lookup

Figure 1 shows the possible outcomes of a lookup. A lookup can either timeout or return a reply. A timeout can be caused by loss, loop, or premature timeout. A lookup loss occurs when it is dropped by the underlying IP network, or when forwarded to a failed neighbor. A lookup loop is caused by inconsistencies in the routing state. A premature timeout occurs when the timeout value is too short and the lookup is still being processed in the network. When a lookup returns a reply, the answer is either correct or incorrect. An incorrect reply is caused by inconsistencies in the routing state.

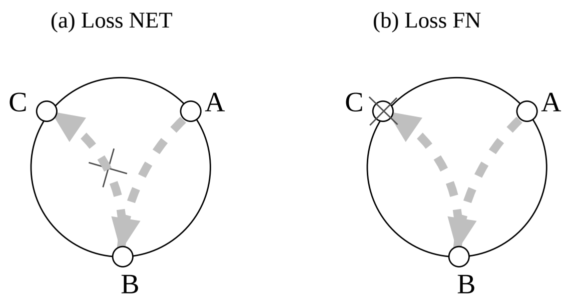


Figure 2: Possible scenarios of a lookup loss in Chord.

2.3 Chord Protocol

Using the example of Chord [10], we illustrate lookup pathologies that can result from routing state inconsistencies.

Chord is a distributed lookup protocol that provides a hash function mapping keys to nodes responsible for them. It assumes a circular identifier space of integers $[0, 2^m)$. Chord ensures that the node responsible for a key is found after $O(\log n)$ hops.

The routing state maintained by each node A consists of a predecessor, successors, and fingers. Predecessor is the node that immediately precedes A on the identifier circle. Successors are the first few nodes that succeed A on the identifier circle. The ith finger is the first node that succeeds A by at least 2^{i-1}, where $1 \leq i \leq m$. The routing invariant in Chord states that lookups will route correctly if each node's predecessor and successor are correctly maintained.

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's routing state is up to date. It does this using a *stabilize* protocol that each node periodically runs every T_s seconds. In each stabilization round, a node updates its immediate successor and another node in its routing state.

2.3.1 Cost of Inconsistency: Loss

A lookup loss occurs when it is dropped by the underlying IP network (Loss-NET), or when forwarded to a failed neighbor (Loss-FN).

Loss-NET Figure 2(a) shows a lookup loss caused by the underlying IP network. Nodes A, B, and C are nodes on the Chord ring. When A forwards a lookup to B and B forwards it on to C, the lookup is dropped by the underlying IP network from B to C.

Loss-FN Figure 2(b) shows a lookup loss caused by a false negative in the routing state of node B. Nodes A, B, and C are nodes on the Chord ring, and C fails, but node B falsely

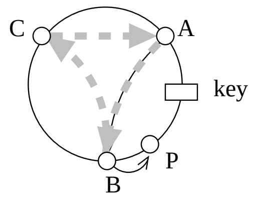


Figure 3: Cost of inconsistency: loop.

thinks its neighbor C is still up. When A forwards a lookup to B and B forwards it on to C, a lookup loss occurs. Such a lookup loss persists until node B detects the failure of C and removes it from its routing state.

Thus minimizing the detection time of a node failure reduces lookup losses caused by false negatives.

2.3.2 Cost of Inconsistency: Loop

Figure 3 shows a lookup loop in Chord, where circles denote nodes on the Chord ring, rectangles denote lookup keys, solid arrows denote neighbor relationships, and dotted arrows denote path taken by a lookup. In Figure 3, node A points to node B as its successor, and node B points to node P as its predecessor. When node A forwards a lookup to its successor B, node B forwards it on because the lookup key is not between B's predecessor P and B. This results in a lookup loop.

Figure 4 shows possible violations of the routing invariant that can cause lookup loops. There are two scenarios to consider. First, node A's successor is incorrect (i.e., A incorrectly points to B as its successor), and B's predecessor is correct or incorrect. Second, node A's successor is correct (i.e., A correctly points to B as its successor), and B's predecessor is incorrect.

In the first scenario, node A's correct successor should be some node S between A and the key (Figure 4(a)), or some node S between the key and B (Figure 4(b)). The above two cases can be caused by false positives, join holes, or leave recoveries in the routing state.

Loop-FP A false positive in the failure detection algorithm at node A may cause A to incorrectly remove node S from its routing state, resulting in node A incorrectly pointing to node B as its successor. Such a routing loop persists until node A reinserts node S in its routing state. Thus, minimizing the number of false positives reduces lookup loops caused by false positives.

Loop-Join A join hole occurs when a new node S joins but has not been incorporated into the routing state of node A. Join holes and false positives are similar in that a node S should, but does not exist in the routing state of node A. A routing loop caused by a join hole persists until node A inserts node S in its routing state. Thus, minimizing the time it takes a node to fully join the network reduces lookup loops caused by join holes.

Loop-Leave A leave recovery occurs after a node A detects the failure of a neighbor but before the recovery algorithm corrects A's routing state to point to the new successor S. Such a routing loop persists until node A inserts node S in its routing state. Thus, minimizing the time it takes a node to recover from leaves reduce lookup loops caused by leave recoveries.

In the second scenario, node B's predecessor is incorrect, and its correct predecessor should be A. This can be caused by a false negative in node B's routing state as illustrated in Figure 4(c).

Loop-FN In Figure 4(c), node B falsely thinks a failed neighbor, node P, is still alive, resulting in node B incorrectly pointing to node P as its predecessor. Such a routing loop persists until node B detects the failure of P and removes it from its routing state. Thus, minimizing the detection time of a node failure reduces lookup loops caused by false negatives.

2.3.3 Cost of Inconsistency: Incorrect Lookup

Figure 5 shows an incorrect lookup in Chord, where circles denote nodes on the Chord ring, rectangles denote lookup keys, solid arrows denote neighbor relationships, and dotted arrows denote path taken by a lookup. In Figure 5, node A points to node B as its successor, and node B points to a node

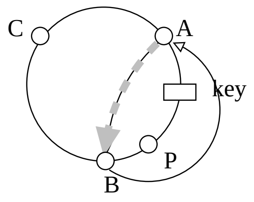


Figure 5: Cost of inconsistency: incorrect lookup.

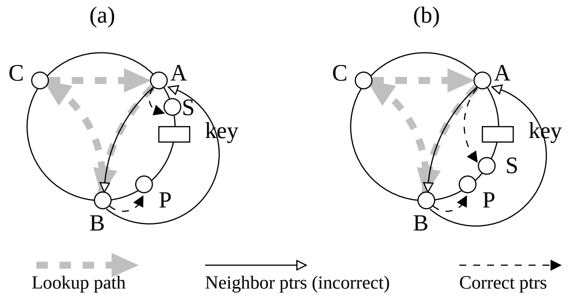


Figure 6: Possible scenarios of an incorrect lookup in Chord.

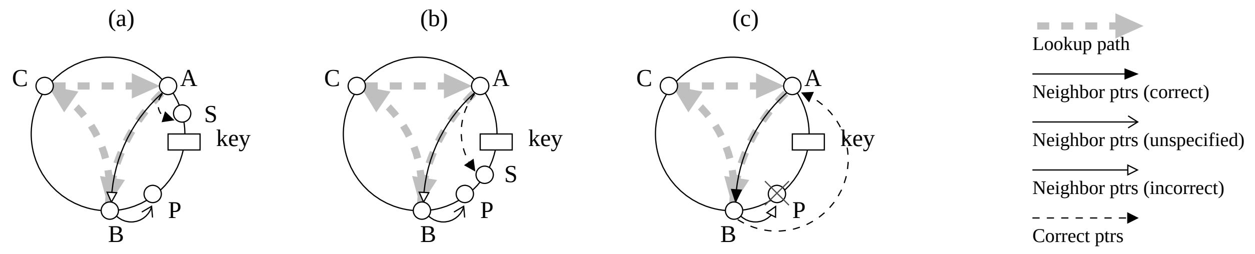


Figure 4: Possible scenarios of a lookup loop in Chord.

(e.g. node A) before the key as its predecessor. When node A forwards a lookup to its successor B, node B accepts it because the lookup key is between B's predecessor and B. This results in an incorrect lookup because there is another node (e.g. node P) between the key and B.

Figure 6 shows possible violations of the routing invariant that can cause incorrect lookups. There is only one scenario to consider. In order to cause an incorrect lookup, node A's successor and node B's predecessor are both incorrect. Node A's successor is incorrect because there is some node (e.g. node S) between the key and node B, and node B's predecessor is also incorrect because B only accepts a lookup if its predecessor is before the key but there is a node (e.g. node P) between the key and node B.

In Figure 6(a), node A's correct successor should be some node S between A and the key. In Figure 6(b), node A's correct successor should be some node S between the key and B. The above two cases can be caused by false positives, join holes, or leave recoveries in the routing state. In both Figures 6(a) and 6(b), node B's correct predecessor should be some node between the key and B (e.g. node P). This can be caused by false positives, join holes, or leave recoveries in the routing state.

3 Simulation Methodology

We now present simulation results evaluating the cost of false negatives, false positives, join holes and leave recoveries in Chord. We make conjectures about the cost of inconsistencies in several other DHTs in Section 5.

In each simulation, we start a Chord network with 1000 nodes by joining a new node to a random bootstrap node once a second. Then we repeatedly kill and replace a random node, timed by a Poisson process.

Key lookups are initiated from random sources to random keys, timed by a Poisson process at a rate of 20 per second. Lookups are routed recursively; each intermediate node forwards a lookup to the next until it reaches the node responsible for the key. The destination node sends back a reply to the source node of the lookup. We examine the causes of lookup timeouts, and incorrect lookups.

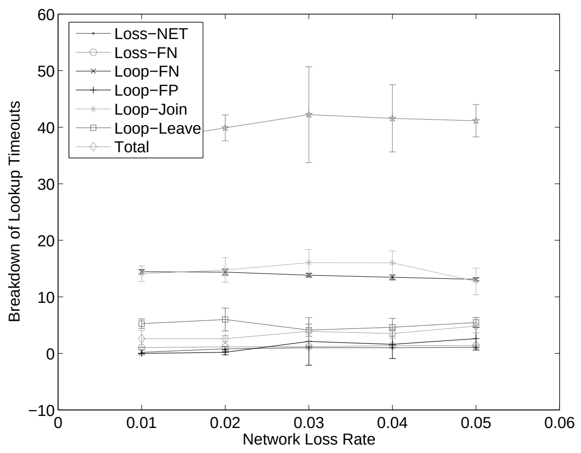


Figure 7: Breakdown of lookup timeouts vs. network loss rate

4 Simulation Results

4.1 Results: Lookup Timeouts vs. Network Loss Rate

Here we examine the breakdown of lookup timeouts as the network loss rate increases from 0.01 to 0.05. For this set of simulations, we hold the churn rate at 0.75 leaves per second, which corresponds to a mean lifetime of 22.22 minutes.

Figure 7 shows a breakdown of the causes of lookup timeouts. As expected, the number of losses due to the underlying IP network (Loss-NET) increases as the network loss rate increases.

The number of losses due to false negatives (Loss-FN) remains approximately constant because the probability of forwarding a lookup to a failed neighbor varies with the churn rate, but does not depend on the network loss rate [13]. For the mean lifetime of 22.22 minutes, a significant number of lookups are lost due to false negatives. On the other hand, the number of loops caused by false negatives (Loop-FN) remains insignificant.

The number of loops due to false positives (Loop-FP) increases slowly as the network loss rate increases because the number of false detections made by failure detection algorithms increases. However, such loops remain insignificant.

The number of loops due to join holes (Loop-Join) and leave recoveries (Loop-Leave) remain approximately constant because such loops only depend on the churn rate.

The results in Figure 7 show that for network loss rates from 0.01 to 0.05, loss due to false negatives causes more timeouts than loop due to false positives in Chord. Thus it is more important to ensure timely failure detection than a low probability of false positives under such conditions. In addition, loop due to join holes causes more timeouts than loop due to false positives in Chord. For example, when the network loss rate is 0.05, the average number of timeouts due to Loop-FP is 0.4 per minute and the average number of timeouts due to Loop-Join is 9.4 per minute. This may be surprising since the average rate of false positives (56 per minute) is greater than the average rate of joins (45 per minute) in the Chord network. This discrepancy can be resolved by looking at Chord's routing invariant, which states that lookups will route correctly if each node's predecessor and successor are correctly maintained [10]. When a new node joins, the routing invariant is violated for both the predecessor and successor of the new node. When a node falsely removes a neighbor, this false positive results in a violation of the routing invariant only if the neighbor is the predecessor or successor of the node. Hence, a join in Chord always violates the routing invariant, whereas a false positive may not necessarily do so. Thus, join protocols such as [3] that can sustain a high rate of node dynamics will improve the rate at which new nodes become fully incorporated into the routing state, and thereby reduce the number of lookup timeouts due to join holes.

Figure 7 also shows that loss (due to the underlying IP network or forwarding to a failed neighbor) causes significantly more timeouts than loop in Chord. Such loss induced timeouts dwarf loop induced timeouts. In Section 4.5, we employ perhop retry to reduce lookup losses, and evaluate the breakdown of lookup timeouts under perhop retry.

4.2 Results: Incorrect Lookups vs. Network Loss Rate

Recall from Section 2.3.3, an incorrect lookup occurs only when node A's successor and node B's predecessor are both incorrect, where A is the node forwarding the lookup to B, and B is the node accepting the lookup. The incorrectness of node A's successor or node B's predecessor can be caused by false positives, join holes, or leave recoveries. In addition, when node B is the source of a lookup and incorrectly believes it is the destination (i.e., B accepts the lookup without forwarding), we include a special type of inconsistency called NO inconsistency because the successor pointer is not used when B incorrectly accepts the lookup. Thus, we have a matrix of four by four entries where each row represents a particular inconsistency that caused the incorrectness of the successor pointer, and each column represents a particular inconsistency that caused the incorrectness of the predecessor pointer, and each entry represents the number of times a particular inconsistency pair caused an incorrect lookup[1].

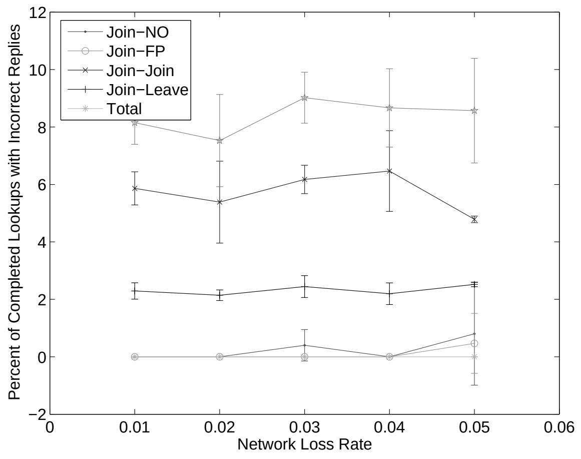


Figure 8: Breakdown of incorrect lookups vs. network loss rate

Figure 8 shows a breakdown of the causes of incorrect lookups. For clarity of presentation, we only plot the Join row in the matrix (i.e., the successor pointer is incorrect due to join holes in the routing state). Figure 8 shows that join holes (Join-Join) and leave recoveries (Join-Leave) cause more incorrect lookups than false positives (Join-FP). This difference is again explained by Chord's routing invariant. For example, when a new node N joins, the routing invariant is violated for both the predecessor P and successor S of the new node N. When a node N leaves, the routing invariant is violated for the successor S of the node N, and may or may not be violated for the predecessor P of the node N. This is because node P maintains a list of successors in Chord, so the routing invariant is not violated for P if it already has node S as a neighbor in its routing state. When a node falsely removes a neighbor, this false positive results in a violation of the routing invariant only if the neighbor is the predecessor or successor of the node. Thus, join holes and leave recoveries are more significant causes of incorrect lookups than false positives in Chord.

4.3 Results: Lookup Timeouts vs. Churn Rate

Overlay networks are intended to scale to at least hundreds of thousands of nodes, where nodes are joining and leaving, putting the network into a continuous state of "churn". Here we observe how the cost of false negatives, false positives, join holes, and leave recoveries vary as churn rate increases. We use mean lifetimes of 88.88, 44.44, 22.22, 11.11, and 5.55 minutes, which correspond to churn rates of 0.1875,

[1]Note that the following entries in the matrix are always zero: NO-NO, FP-NO, Join-NO, Leave-NO.

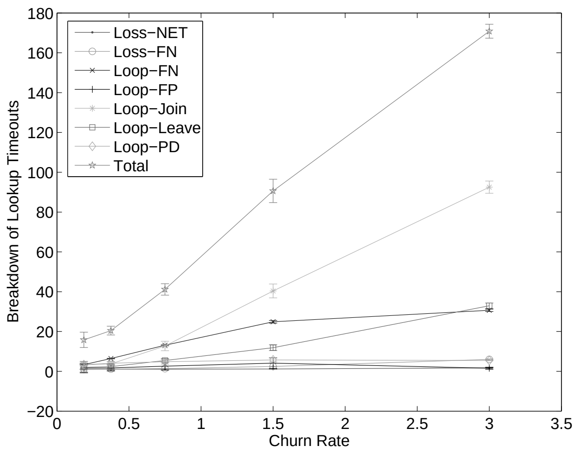


Figure 9: Breakdown of lookup timeouts vs. churn rate

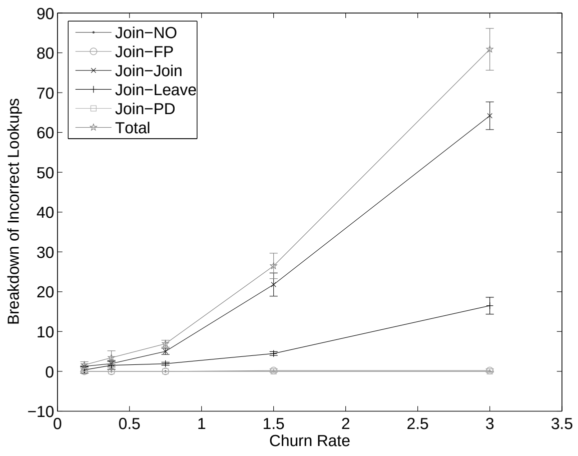


Figure 10: Breakdown of incorrect lookups vs. churn rate

0.375, 0.75, 1.5, and 3 leaves per second. The network loss rate is 0.05 in these simulations.

Figure 9 shows a breakdown of the causes of lookup timeouts. The number of losses due to the underlying IP network (Loss-NET) remains approximately constant as churn rate increases because the network loss rate is held at 0.05.

The number of losses due to false negatives (Loss-FN) increases because the probability of forwarding a lookup to a failed neighbor increases as churn rate increases [13]. On the other hand, the number of loops due to false negatives (Loop-FN) remains insignificant.

The number of loops due to false positives (Loop-FP) remains insignificant as churn rate increases. At a network loss rate of 0.05, the rate of false positives is low (56 per minute [13]). Moreover, a false positive leads to lookup loops only if it is caused by a node falsely removing its successor.

As expected, the number of loops due to join holes (Loop-Join) and leave recoveries (Loop-Leave) increases because the fraction of nodes with incorrect successors increases as churn rate increases. Loop due to join holes becomes a significant cause of timeouts as churn rate increases. Thus, join protocols such as [3] that improve the rate at which new nodes can join a network will allow the network to support higher churn rates.

The results in Figure 9 again show that loss due to the underlying IP network or false negatives causes significantly more timeouts than loop in Chord. In Section 3.4, we employ perhop retry to reduce lookup losses, and evaluate the breakdown of lookup timeouts versus churn rate under perhop retry.

4.4 Results: Incorrect Lookups vs. Churn Rate

Figure 10 plots a breakdown of the causes of incorrect lookups when the successor pointer is incorrect due to join holes in the routing state. The number of incorrect lookups due to false positives (Join-FP) remains insignificant as churn rate increases. At a network loss rate of 0.05, the rate of false positives is low, and a false positive leads to incorrect lookups only if it is falsely removed by its successor.

The number of incorrect lookups due to join holes (Join-Join) and leave recoveries (Join-Leave) increases as churn rate increases because the fraction of nodes with incorrect predecessors and successors increases as churn rate increases. Thus, join holes and leave recoveries are more significant causes of incorrect lookups than false positives in Chord.

4.5 Results: Perhop Retry

In this section, we employ per overlay hop retry to reduce lookup losses, which cause a significant fraction of lookup timeouts. We implement a simple perhop retry mechanism in Chord, where a node forwards a lookup to a neighbor, and waits for an ack. If a lookup is not acknowledged within some timeout period, it is retransmitted to the same neighbor up to a maximum of MAX_NBR_TRIES-1 times. If an ack is not received from the neighbor after MAX_NBR_TRIES transmissions, the neighbor is marked as possibly down, and the node tries another neighbor. Note that a neighbor marked as possibly down is not used in forwarding until the status is reset when the node receives an ack in response to a lookup or when the node receives an ack in response to a keep-alive probe. For this set of simulations, we set the perhop timeout period to 50 milliseconds, and MAX_NBR_TRIES to 3. For more sophisticated implementations of perhop retry that maintains TCP-like state for each neighbor, refer to [8].

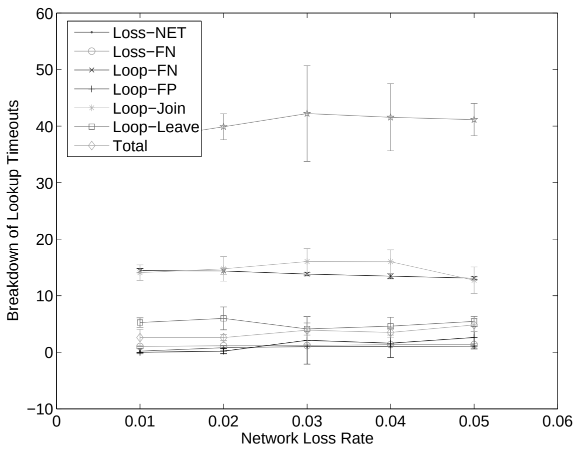


Figure 11: Breakdown of lookup timeouts vs. network loss rate under perhop retry

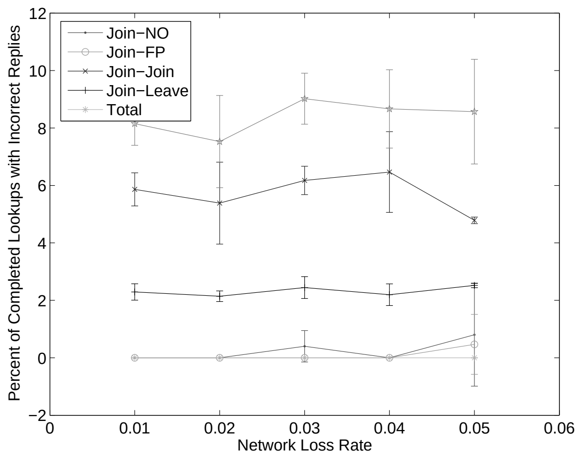


Figure 12: Breakdown of incorrect lookups vs. network loss rate under perhop retry

Perhop retry introduces a new type of inconsistency, possibly down (PD). A PD inconsistency occurs when a node falsely marks a neighbor as possibly down. False positives and possibly downs are similar in that a false positive should but does not exist in the routing state of a node, and a possibly down exist in the routing state but is not used in forwarding.

We now present simulation results evaluating the cost of false negatives, false positives, join holes, leave recoveries and possibly downs in Chord under perhop retry.

4.5.1 *Lookup Timeouts vs. Network Loss Rate*

Figure 11 shows a breakdown of the causes of lookup timeouts versus network loss rate under perhop retry.

Lookup losses due to the underlying IP network and false negatives are significantly reduced. For example, when the network loss rate is 0.05, the average number of timeouts due to Loss-NET and Loss-FN are 1 per minute and 1.5 per minute under perhop retry, compared to 208.1 per minute and 72.8 per minute without perhop retry. This shows that perhop retry is very effective at reducing lookup losses.

Figure 11 shows that the three most significant causes of lookup timeouts are loops due to false negatives (Loop-FN), join holes (Loop-Join), and leave recoveries (Loop-Leave). The number of loops due to possibly downs (Loop-PD) is low. The number of loops due to false positives (Loop-FP) remains insignificant. Thus, it is more important to ensure timely failure detection than a low probability of false positive, and the cost of join holes and leave recoveries is higher than that of false positives due to the routing invariant of Chord.

4.5.2 *Incorrect Lookups vs. Network Loss Rate*

Figure 12 shows a breakdown of the causes of incorrect lookups versus network loss rate under perhop retry. For clarity of presentation, we only plot the Join row in the matrix (i.e. the successor pointer is incorrect due to join holes in the routing state). The results in Figure 12 show that join holes and leave recoveries are more significant causes of incorrect lookups than false positives in Chord.

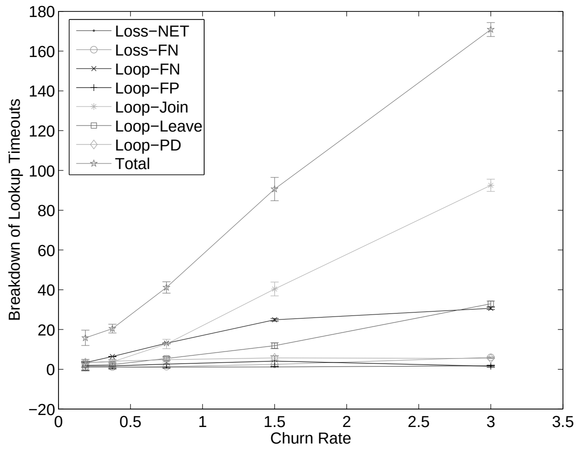


Figure 13: Breakdown of lookup timeouts vs. churn rate under perhop retry

4.5.3 *Lookup Timeouts vs. Churn Rate*

Figure 13 shows a breakdown of the causes of lookup timeouts versus churn rate under perhop retry. Again lookup losses due to the underlying IP network and false negatives are significantly reduced. The three most significant causes of lookup timeouts are loops due to false negatives (Loop-FN), join holes (Loop-Join), and leave recoveries (Loop-Leave).

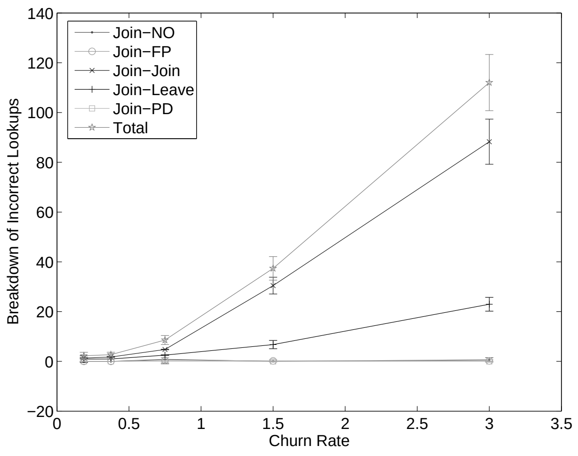


Figure 14: Breakdown of incorrect lookups vs. churn rate under perhop retry

4.5.4 *Incorrect Lookups vs. Churn Rate*

Figure 12 shows a breakdown of thecauses of incorrect lookups versus churn rate under perhop retry. These results again show that join holes and leave recoveries are more significant causes of incorrect lookups than false positives in Chord.

5 Discussion and Future Work

In this section, we make conjectures about the relative cost of false negatives, false positives, join holes, and leave recoveries in several other DHTs based on their routing invariants. A more in-depth quantitative study that makes precise these costs might yield other interesting results. We consider the following DHTs: CAN [7], Tapestry [12], and Pastry [9].

The number of losses due to false negatives depends on the failure detection algorithm and churn rate [13], but does not depend on the routing invariant. Thus, DHTs will experience similar costs of false negatives on lookup performance under similar failure detection algorithms and churn rates.

CAN A virtual d-dimensional Cartesian coordinate space is partitioned among all the nodes in the system such that every node owns its individual, distinct zone within the overall space. A CAN node maintains a routing table that contains the IP address and virtual coordinate zone of each of its neighbors in the coordinate space. Two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension. To route a lookup, the key is deterministically mapped onto a point P in the coordinate space using a uniform hash function. The lookup is then routed to the node that owns the zone within which point P lies. There are no loops because each routing step takes a lookup to a node that is geographically closer to point P than the local node.

The routing invariant of CAN states: a routing table must store at least two neighbors per dimension, one to advance and one to retreat along each dimension. A false positive violates the routing invariant when a node falsely removes the only neighbor that advances or retreats along a dimension. A join hole or leave recovery violates the routing invariant of nodes who are neighbors of the new node or the leaving node (up to $O(d)$ nodes).

If the number of dimensions d in CAN is set to $(\log n)/2$, then each node maintains $O(\log n)$ neighbors as in Chord. Given the greater number of places where the routing invariant could be violated, the cost of false positives and join holes may be higher in CAN than in Chord on lookup performance. However, the cost of false positives and join holes may be comparable in CAN.

Tapestry A Tapestry node ID is a sequence of l digits, where each digit is in base b. An identifier space with a hexadecimal base and 160-bit values is commonly used ($l = 40$, $b = 16$). Each node has a routing table with l levels, where each level contains b entries. Neighbors in the ith level share a prefix of length $i-1$ digits with the local node, but differ in the ith digit. Each entry contains a primary neighbor and a few backup neighbors. A lookup is routed by matching successive digits in the key (prefix-based routing). When a digit cannot be matched, the neighbor with the next higher digit (modulo b) is chosen. There are no loops because each routing step "matches" one more level in the key by forwarding the lookup to a node that either (1) shares a longer prefix with the key than the local node, or (2) shares as long a prefix, but with the next higher digit when a digit cannot be matched.

The routing invariant of Tapestry states: (i) every entry in a routing table must store at least one neighbor if qualified nodes exist; (ii) if there is no qualified node for an entry, then the entry must be empty. A false positive violates the routing invariant when a node falsely removes the only neighbor in a routing entry. In particular, a false positive always violates the routing invariant when a node falsely removes a neighbor in the lth level. A join hole violates the routing invariant of any node (up to $(b-1) \times b^{l-i}$ nodes) who shares a prefix of length $i-1$ with the new node if the new node is the first node in the network with a particular prefix of length i. In particular, since a new node is the first node in the network with a particular prefix length of l, a join hole always violates the routing invariant of any node (up to $b-1$ nodes) who shares a prefix of length $l-1$ with the new node. A leave recovery violates the routing invariant when a neighbor leaves and it is the only neighbor in a routing entry. For example, a leave recovery violates the routing invariant of any node (up to $(b-1) \times b^{l-i}$ nodes) who shares a prefix of length $i-1$ with the leaving node if the leaving node is the only node in the network with a particular prefix of length i. In particular, since a leaving node is the only node in the network with a particular prefix length of l, a leave recovery always violate sthe routing invariant of any node (up to $b-1$ ndoes) who shares a prefix of length $l-1$ with the leaving node.

Given the greater number of places where the routing invariant could be violated, the cost of false positives, join holes, and leave recoveries may be higher in Tapestry than in Chord on lookup performance. However, join holes or leave recoveries are still more costly than false positives in Tapestry because a join hole or leave recovery always violates the routing invariant, whereas a false positive may not necessarily do so.

Pastry Systems like Pastry (and Bamboo [8]) are built almost like Tapestry, but each node also maintains a leaf set in addition to the routing table. A node's leaf set is the set of $2k$ nodes immediately preceding and following it in the circular identifier space. A lookup is routed to the node whose ID is numerically closest to the key. There are no loops because each routing step takes a lookup to a node that either (1) shares a longer prefix with the key than the local node, or (2) shares as long a prefix with, but is numerically closer to the key than the local node.

Because of the leaf set, the routing invariant in Pastry is the same as in Chord. A false positive does not violate the routing invariant unless it is falsely removed by either its predecessor or successor on the circular identifier space. A join hole or leave recovery violates the routing invariant for both the predecessor and successor of the new node or the leaving node.

Given the same routing invariant as Chord, the relative cost of false positives, join holes, and leave recoveries on lookup performance in Pastry should be similar to Chord.

6 Related Work

There are several works which analyze DHTs in the context of static networks. Xu [11] studies the fundamental tradeoff between the size of the routing table and the network diameter in designing a DHT. Gummadi et al. [1] explore the impact of DHT routing geometry on static resilience and proximity. Loguinov [6] examine the effect of graph-theoretic properties of structured peer-to-peer architectures on routing distances and fault resilience. In contrast, we analyze the cost of inconsistency on lookup performance in a dynamic environment with continuous churn.

Liben-Nowell et al. [5] present a theoretical analysis of peer-to-peer networks under continuous churn. They give a lower bound on the rate of maintenance traffic for a network to remain connected, and prove that Chord's maintenance rate is within a logarithmic factor of the optimum rate. The analysis focuses on asymptotic bounds, and assumes perfect failure detection and reliable message delivery. In contrast, we examine the cost of inconsistency on lookup performance under realistic system conditions such as message loss and false failure detections.

Li et al. [4] analyze the performance of lookups in Chord, Kademlia, Kelips, and Tapestry under churn. Lookups that timeout or return incorrect replies are retried up to a maximum of four seconds. Protocol parameters are varied to explore the tradeoff between lookup latency and bandwidth cost. Rather than looking at how protocol parameters affect lookup performance, we study why lookups timeout or return incorrect replies in Chord. In particular, we identify the different types of inconsistencies, and evaluate the cost of each type of inconsistency on lookup performance.

Lam et al. [3] present a new join protocol for hypercube routing that can sustain a high rate of node dynamics by maintaining K-consistent neighbor tables. Join protocols of other DHTs may also be improved to increase the rate at which new nodes become fully incorporated into the routing state, and thereby reduce the number of lookup timeouts due to join holes, a significant factor of lookup timeouts and incorrect lookups.

Krishnamurthy et al. [2] presents a theoretical analysis of Chord using a Master-equation formalism to predict the fraction of failed or incorrect successor and finger pointers and use these quantities to predict number of failed lookups and lookup latency. In contrast, we identify the type of inconsistency that led to incorrect predecessor and successor pointers, and evaluate the cost of each type of inconsistency on lookup performance. By identifying inconsistencies that have significant costs on lookup performance, we can improve certain aspects of a DHT to minimize such inconsistencies.

Zhuang et al. [13] study how the design of various keep-alive algorithms affect their performance in node failure detection time, probability of false positive, control overhead, and packet loss rate. In contrast, we aim to address the question of how node failures, false positives, joins, and leaves actually impact application-level performance by evaluating the cost of inconsistency on lookup performance.

7 Conclusion

This paper studies the cost of inconsistency on lookup peformance in DHTs. Using the example of Chord, we evaluate the cost of each type of inconsistency on lookup performance. Our results indicate that the routing invariant has a first order impact on the relative cost of different types of inconsistencies. In addition, the cost of false negatives is higher than that of false positives, which means it is more important to ensure timely failure detection than a low probability of false positives. The cost of join holes is also higher than that of false positives due to the routing invariant of Chord. We also make conjectures about the cost of inconsistency in other DHTs based on their routing invariants. We believe that these findings will provide important insights on how to make certain choices and tradeoffs when designing future algorithms to construct and maintain consistent routing state in DHTs.

References

[1] GUMMADI, K., AND ET AL. The impact of dht routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM* (2003).

[2] KRISHNAMURTHY, S., EL-ANSARY, S., AURELL, E., AND HARIDI, S. A statistical theory of chord under churn. In *Proc. IPTPS 2005*.

[3] LAM, S., AND LIU, H. Failure recovery for structured p2p networks: Protocol design and performance evaluation. In *Proc. SIGMETRICS 2004*.

[4] LI, J., STRIBLING, J., MORRIS, R., KAASHOEK, F., AND GIL, T. Comparing the performance of distributed hash tables under churn. In *Proc. IPTPS 2004*.

[5] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *Proc. PODC 2002*.

[6] LOGUINOV, D., AND ET AL. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. In *Proc. SIGCOMM 2003*.

[7] RATNASAMY, S., AND ET. AL. A scalable content-addressable network. In *Proc. SIGCOMM 2001*.

[8] RHEA, S., AND ET AL. Handling churn in a dht. In *Proc. USENIX 2004*.

[9] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*.

[10] STOICA, I., AND ET AL. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM 2001*.

[11] XU, J. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. In *Proc. Infocom 2003*.

[12] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal of Selected Area on Communications 22*, 1 (Jan. 2004), 41–53.

[13] ZHUANG, S., GEELS, D., STOICA, I., AND KATZ, R. On failure detection algorithms in overlay networks. In *Proceedings of IEEE INFOCOM'05*.