Kademlia: A Peer-to-peer Information System Based on the XOR Metric

Petar Maymounkov and David Mazières
{petar,dm}@cs.nyu.edu
http://kademlia.scs.cs.nyu.edu

Abstract

We describe a peer-to-peer system which has provable consistency and performance in a fault-prone environment. Our system routes queries and locates nodes using a novel XOR-based metric topology that simplifies the algorithm and facilitates our proof. The topology has the property that every message exchanged conveys or reinforces useful contact information. The system exploits this information to send parallel, asynchronous query messages that tolerate node failures without imposing timeout delays on users.

1 Introduction

This paper describes Kademlia, a peer-to-peer ⟨key,value⟩ storage and lookup system. Kademlia has a number of desirable features not simultaneously offered by any previous peer-to-peer system. It minimizes the number of configuration messages nodes must send to learn about each other. Configuration information spreads automatically as a side-effect of key lookups. Nodes have enough knowledge and flexibility to route queries through low-latency paths. Kademlia uses parallel, asynchronous queries to avoid timeout delays from failed nodes. The algorithm with which nodes record each other's existence resists certain basic denial of service attacks. Finally, several important properties of Kademlia can be formally proven using only weak assumptions on uptime distributions (assumptions we validate with measurements of existing peer-to-peer systems).

Kademlia takes the basic approach of many peer-to-peer systems. Keys are opaque, 160-bit quantities (e.g., the SHA-1 hash of some larger data). Participating computers each have a node ID in the 160-bit key space. ⟨key,value⟩ pairs are stored on nodes with IDs "close" to the key for some notion of closeness. Finally, a node-ID-based routing algorithm lets anyone locate servers near a destination key.

Many of Kademlia's benefits result from its use of a novel XOR metric for distance between points in the key space. XOR is symmetric, allowing Kademlia participants to receive lookup queries from precisely the same distribution of nodes contained in their routing tables. Without this property, systems such as Chord [5] do not learn useful routing information from queries they receive. Worse yet, because of the asymmetry of Chord's metric, Chord routing tables are rigid. Each entry in a Chord node's finger table must store the precise node proceeding an interval in the ID space; any node actually in the interval will be greater than some keys in the interval, and thus very far from the key. Kademlia, in contrast, can send a query to any node within an interval, allowing it to select routes based on latency or even send parallel asynchronous queries.

To locate nodes near a particular ID, Kademlia uses a single routing algorithm from start to finish. In contrast, other systems use one algorithm to get near the target ID and another for the last few hops. Of existing systems, Kademlia most resembles Pastry's [1] first phase, which (though not described this way by the authors) successively finds nodes roughly half as far from the target ID by Kademlia's XOR metric. In a second phase, however, Pastry switches

This research was partially supported by National Science Foundation grants CCR 0093361 and CCR 9800085.

distance metrics to the numeric difference between IDs. It also uses the second, numeric difference metric in replication. Unfortunately, nodes close by the second metric can be quite far by the first, creating discontinuities at particular node ID values, reducing performance, and frustrating attempts at formal analysis of worst-case behavior.

2 System description

Each Kademlia node has a 160-bit node ID. Node IDs are constructed as in Chord, but to simplify this paper we assume machines just choose a random, 160-bit identifier when joining the system. Every message a node transmits includes its node ID, permitting the recipient to record the sender's existence if necessary.

Keys, too, are 160-bit identifiers. To publish and find $\langle$key,value$\rangle$ pairs, Kademlia relies on a notion of distance between two identifiers. Given two 160-bit identifiers, x and y, Kademlia defines the distance between them as their bitwise exclusive or (XOR) interpreted as an integer, $d(x, y) = x \oplus y$.

We first note that XOR is a valid, albeit non-Euclidean, metric. It is obvious that that $d(x, x) = 0$, $d(x, y) > 0$ if $x \neq y$, and $\forall x, y : d(x, y) = d(y, x)$. XOR also offers the triangle property: $d(x, y) + d(y, z) \geq d(x, z)$. The triangle property follows from the fact that $d(x, z) = d(x, y) \oplus d(y, z)$ and $\forall a \geq 0, b \geq 0 : a + b \geq a \oplus b$.

Like Chord's clockwise circle metric, XOR is *unidirectional*. For any given point x and distance $\Delta > 0$, there is exactly one point y such that $d(x, y) = \Delta$. Unidirectionality ensures that all lookups for the same key converge along the same path, regardless of the originating node. Thus, caching $\langle$key,value$\rangle$ pairs along the lookup path alleviates hot spots. Like Pastry and unlike Chord, the XOR topology is also symmetric ($d(x, y) = d(y, x)$ for all x and y).

2.1 Node state

Kademlia nodes store contact information about each other to route query messages. For each $0 \leq i < 160$, every node keeps a list of $\langle$IP address, UDP port, Node ID$\rangle$ triples for nodes of distance between 2^i and 2^{i+1} from itself. We call

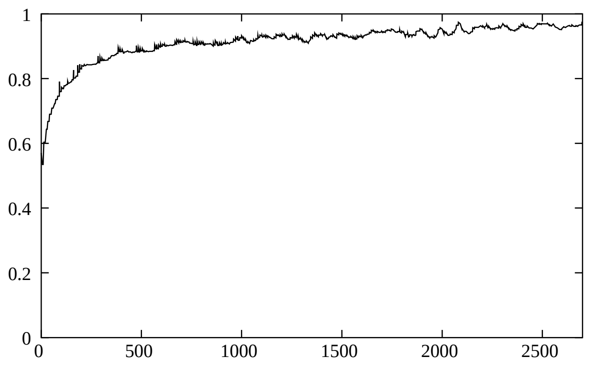

Figure 1: Probability of remaining online another hour as a function of uptime. The x axis represents minutes. The y axis shows the the fraction of nodes that stayed online at least x minutes that also stayed online at least $x + 60$ minutes.

these lists k-buckets. Each k-bucket is kept sorted by time last seen—least-recently seen node at the head, most-recently seen at the tail. For small values of i, the k-buckets will generally be empty (as no appropriate nodes will exist). For large values of i, the lists can grow up to size k, where k is a system-wide replication parameter. k is chosen such that any given k nodes are very unlikely to fail within an hour of each other (for example $k = 20$).

When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender's node ID. If the sending node already exists in the recipient's k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, then the recipient pings the k-bucket's least-recently seen node to decide what to do. If the least-recently seen node fails to respond, it is evicted from the k-bucket and the new sender inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender's contact is discarded.

k-buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the list. This preference for old contacts is driven by our analysis of Gnutella trace data collected by Saroiu et. al. [4]. Figure 1 shows the percentage of Gnutella nodes that stay online another hour as a function of current uptime. The longer a node has been up, the more likely it is to remain up another hour. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online.

A second benefit of k-buckets is that they provide resistance to certain DoS attacks. One cannot flush nodes' routing state by flooding the system with new nodes. Kademlia nodes will only insert the new nodes in the k-buckets when old nodes leave the system.

2.2 Kademlia protocol

The Kademlia protocol consists of four RPCs: PING, STORE, FIND_NODE, and FIND_VALUE. The PING RPC probes a node to see if it is online. STORE instructs a node to store a $\langle$key, value$\rangle$ pair for later retrieval.

FIND_NODE takes a 160-bit ID as an argument. The recipient of a the RPC returns $\langle$IP address, UDP port, Node ID$\rangle$ triples for the k nodes it knows about closest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about).

FIND_VALUE behaves like FIND_NODE—returning $\langle$IP address, UDP port, Node ID$\rangle$ triples—with one exception. If the RPC recipient has received a STORE RPC for the key, it just returns the stored value.

In all RPCs, the recipient must echo a 160-bit random RPC ID, which provides some resistance to address forgery. PINGs can also be piggy-backed on RPC replies for the RPC recipient to obtain additional assurance of the sender's network address.

The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID. We call this procedure a *node lookup*. Kademlia employs a recursive algorithm for node lookups. The lookup initiator starts by picking α nodes from its closest non-empty k-bucket (or, if that bucket has fewer than α entries, it just takes the α closest nodes it knows of). The initiator then sends parallel, asynchronous FIND_NODE RPCs to the α nodes it has chosen. α is a system-wide concurrency parameter, such as 3.

In the recursive step, the initiator resends the FIND_NODE to nodes it has learned about from previous RPCs. (This recursion can begin before all α of the previous RPCs have returned). Of the k nodes the initiator has heard of closest to the target, it picks α that it has not yet queried and resends the FIND_NODE RPC to them.[1] Nodes that fail to respond quickly are removed from consideration until and unless they do respond. If a round of FIND_NODEs fails to return a node any closer than the closest already seen, the initiator resends the FIND_NODE to all of the k closest nodes it has not already queried. The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. When $\alpha = 1$ the lookup algorithm resembles Chord's in terms of message cost and the latency of detecting failed nodes. However, Kademlia can route for lower latency because it has the flexibility of choosing any one of k nodes to forward a request to.

Most operations are implemented in terms of the above lookup procedure. To store a $\langle$key,value$\rangle$ pair, a participant locates the k closest nodes to the key and sends them STORE RPCs. Additionally, each node re-publishes the $\langle$key,value$\rangle$ pairs that it has every hour.[2] This ensures persistence (as we show in our proof sketch) of the $\langle$key,value$\rangle$ pair with very high probability. Generally, we also require the original publishers of a $\langle$key,value$\rangle$ pair to republish it every 24 hours. Otherwise, all $\langle$key,value$\rangle$ pairs expire 24 hours after the original publishing, in order to limit stale information in the system.

Finally, in order to sustain consistency in the publishing-searching life-cycle of a $\langle$key,value$\rangle$ pair, we require that whenever a node w observes a new node u which is closer to some of w's $\langle$key,value$\rangle$ pairs, w replicates these pairs to u without removing them from its own database.

To find a $\langle$key,value$\rangle$ pair, a node starts by performing a lookup to find the k nodes with IDs closest to the key. However, value lookups use FIND_VALUE rather than FIND_NODE RPCs. Moreover, the proce-

[1]Bucket entries and FIND replies can be augmented with round trip time estimates for use in selecting the α nodes.

[2]This can be optimized to require far fewer than k^2 messages, but a description is beyond the scope of this paper.

dure halts immediately when any node returns the value. For caching purposes, once a lookup succeeds, the requesting node stores the $\langle$key,value$\rangle$ pair at the closest node it observed to the key that did not return the value.

Because of the unidirectionality of the topology, future searches for the same key are likely to hit cached entries before querying the closest node. During times of high popularity for a certain key, the system might end up caching it at many nodes. To avoid "over-caching," we make the expiration time of a $\langle$key,value$\rangle$ pair in any node's database exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID.[3] While simple LRU eviction would result in a similar lifetime distribution, there is no natural way of choosing the cache size, since nodes have no *a priori* knowledge of how many values the system will store.

Buckets will generally be kept constantly fresh, due to the traffic of requests traveling through nodes. To avoid pathological cases when no traffic exists, each node refreshes a bucket in whose range it has not performed a node lookup within an hour. Refreshing means picking a random ID in the bucket's range and performing a node search for that ID.

To join the network, a node u must have a contact to an already participating node w. u inserts w into the appropriate k-bucket. u then performs a node lookup for its own node ID. Finally, u refreshes all k-buckets further away than its closest neighbor. During the refreshes, u both populates its own k-buckets and inserts itself into other nodes' k-buckets as necessary.

3 Sketch of proof

To demonstrate proper function of our system, we need to prove that most operations take $\lceil \log n \rceil + c$ time for some small constant c, and that a $\langle$key,value$\rangle$ lookup returns a key stored in the system with overwhelming probability.

We start with some definitions. For a k-bucket covering the distance range $\left[2^i, 2^{i+1}\right)$, define the *index* of the bucket to be i. Define the *depth*, h, of a node to be $160 - i$, where i is the smallest index of a non-empty bucket. Define node y's *bucket height* in node x to be the index of the bucket into which x would insert y minus the index of x's least significant empty bucket. Because node IDs are randomly chosen, it follows that highly non-uniform distributions are unlikely. Thus with overwhelming probability the height of a any given node will be within a constant of $\log n$ for a system with n nodes. Moreover, the bucket height of the closest node to an ID in the kth-closest node will likely be within a constant of $\log k$.

Our next step will be to assume the invariant that every k-bucket of every node contains at least one contact if a node exists in the appropriate range. Given this assumption, we show that the node lookup procedure is correct and takes logarithmic time. Suppose the closest node to the target ID has depth h. If none of this node's h most significant k-buckets is empty, the lookup procedure will find a node half as close (or rather whose distance is one bit shorter) in each step, and thus turn up the node in $h - \log k$ steps. If one of the node's k-buckets is empty, it could be the case that the target node resides in the range of the empty bucket. In this case, the final steps will not decrease the distance by half. However, the search will proceed exactly as though the bit in the key corresponding to the empty bucket had been flipped. Thus, the lookup algorithm will always return the closest node in $h - \log k$ steps. Moreover, once the closest node is found, the concurrency switches from α to k. The number of steps to find the remaining $k-1$ closest nodes can be no more than the bucket height of the closest node in the kth-closest node, which is unlikely to be more than a constant plus $\log k$.

To prove the correctness of the invariant, first consider the effects of bucket refreshing if the invariant holds. After being refreshed, a bucket will either contain k valid nodes or else contain every node in its range if fewer than k exist. (This follows from the correctness of the node lookup procedure.) New nodes that join will also be inserted into any buckets that are not full. Thus, the only way to violate the invariant is for there to exist $k+1$ or more nodes in the range of a particular bucket, and for the k actually contained in the bucket all to fail with no intervening

[3] This number can be inferred from the bucket structure of the current node.

lookups or refreshes. However, k was precisely chosen for the probability of simultaneous failure within an hour (the maximum refresh time) to be small.

In practice, the probability of failure is much smaller than the probability of k nodes leaving within an hour, as every incoming or outgoing request updates nodes' buckets. This results from the symmetry of the XOR metric, because the IDs of the nodes with which a given node communicates during an incoming or outgoing request are distributed exactly compatibly with the node's bucket ranges.

Moreover, even if the invariant does fail for a single bucket in a single node, this will only affect running time (by adding a hop to some lookups), not correctness of node lookups. For a lookup to fail, k nodes on a lookup path must each lose k nodes in the same bucket with no intervening lookups or refreshes. If the different nodes' buckets have no overlap, this happens with probability 2^{-k^2}. Otherwise, nodes appearing in multiple other nodes' buckets will likely have longer uptimes and thus lower probability of failure.

Now we look at a $\langle$key,value$\rangle$ pair's recovery. When a $\langle$key,value$\rangle$ pair is published, it is populated at the k nodes, closest to the key. It is also re-published every hour. Since even new nodes (the least reliable) have probability $1/2$ of lasting one hour, after one hour the $\langle$key,value$\rangle$ pair will still be present on one of the k nodes closest to the key with probability $1 - 2^{-k}$. This property is not violated by the insertion of new nodes that are close to the key, because as soon as such nodes are inserted, they contact their closest nodes in order to fill their buckets and thereby receive any nearby $\langle$key,value$\rangle$ pairs they should store. Of course, if the k closest nodes to a key fail and the $\langle$key,value$\rangle$ pair has not been cached elsewhere, Kademlia will lose the pair.

4 Discussion

The XOR-topology-based routing that we use very much resembles the first step in the routing algorithms of Pastry [1], Tapestry [2], and Plaxton's distributed search algorithm [3]. All three of these, however, run into problems when they choose to approach the target node b bits at a time (for acceleration purposes). Without the XOR topology, there is a need for an additional algorithmic structure for discovering the target within the nodes that share the same prefix but differ in the next b-bit digit. All three algorithms resolve this problem in different ways, each with its own drawbacks; they all require secondary routing tables of size $O(2^b)$ in addition to the main tables of size $O(2^b \log_{2^b} n)$. This increases the cost of bootstrapping and maintenance, complicates the protocols, and for Pastry and Tapestry prevents a formal analysis of correctness and consistency. Plaxton has a proof, but the system is less geared for highly faulty environments like peer-to-peer networks.

Kademlia, in contrast, can easily be optimized with a base other than 2. We configure our bucket table so as to approach the target b bits per hop. This requires having one bucket for each range of nodes at a distance $[j2^{160-(i+1)b}, (j+1)2^{160-(i+1)b}]$ from us, for each $0 < j < 2^b$ and $0 \leq i < 160/b$, which amounts to expected no more than $(2^b - 1)\log_{2^b} n$ buckets with actual entries. The implementation currently uses $b = 5$.

5 Summary

With its novel XOR-based metric topology, Kademlia is the first peer-to-peer system to combine provable consistency and performance, latency-minimizing routing, and a symmetric, unidirectional topology. Kademlia furthermore introduces a concurrency parameter, α, that lets people trade a constant factor in bandwidth for asynchronous lowest-latency hop selection and delay-free fault recovery. Finally, Kademlia is the first peer-to-peer system to exploit the fact that node failures are inversely related to uptime.

References

[1] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Accepted for Middleware, 2001*, 2001. http://research.microsoft.com/ãntr/pastry/.

[2] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: an infrastructure for fault-

tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U.C. Berkeley, April 2001.

[3] Andréa W. Richa C. Greg Plaxton, Rajmohan Rajaraman. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA*, pages 311–320, June 1997.

[4] Stefan Saroiu, P. Krishna Gummadi and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical Report UW-CSE-01-06-02, University of Washington, Department of Computer Science and Engineering, July 2001.

[5] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.