An Introduction to Kademlia DHT & How It Works

Please forgive the terrible formatting. The article got mixed up somehow. I will re-format and add missing information soon.

Overview

Kademlia is one of the most popular peer-to-peer (P2P) Distributed Hash Table (DHT) in use today.

Kademlia Terms

Node A node is a participating computer on the Kademlia DHT network.

pair (KV) DHTs store data in the form of KV pairs, and this is no different for Kademlia. The key is an identifier to find the value on the Kademlia network and the value is the actual data that needs to be stored on the DHT.

Lookup Lookup is the procedure used by Kademlia to find a value for a given key.

k In Kademlia, k is a system-wide replication parameter. The value of k is chosen such that any given k nodes are very unlikely to fail within an hour of each other.

Data/Content The terms data and content are both used to refer to any data stored on the Kademlia DHT in the form of a KV pair.

Why Kademlia

Kademlia provides many desirable features that are not simultaneously offered by any other DHT [1]. Kademlia has several that make it a preferred choice of DHT. These include:

- Kademlia minimizes the number of inter-node introduction messages.
- Configuration information such as nodes on the network and neighboring nodes spread automatically as a side effect of key lookups.
- Nodes in Kademlia are knowledgeable of other nodes. This allows routing queries through low latency paths.
- Kademlia uses parallel and asynchronous queries which avoid timeout delays from failed nodes.
- Kademlia is resistant to some DOS attacks.

Key

Kademlia uses keys to identify both nodes and data on the Kademlia network. Kademlia keys are opaque, 160-bit quantities. Participating computers each have a key, called a NodeId, in the 160-bit key-space. Since Kademlia stores content in the form of [key, value] pairs, each data on the Kademlia DHT is also uniquely identified by a key in the 160-bit key-space [1].

Overview

Kademlia nodes are represented in the form of a binary tree where nodes are the leaves of the binary tree as shown in Figure 1.1. Note that the tree given is simplified to clearly demonstrate the concepts described. Figure 1.1: Kademlia network The NodeId of each node can be found by tracing the bit value of the edges to that node. The highlighted node in the tree would have a NodeId 0011 [1]. From the perspective of a single node, the tree is divided into subtrees. From the node's location in the tree traversing downwards, the subtrees are successive lower subtrees that don't contain that node. Figure 1.2 shows the subtrees for the node represented by the black dot, node 0011. The highest subtree consists of half of the binary tree not containing the node. The next subtree consists of the half of the remaining tree not containing the node, and so forth. In Figure 1.2, the subtrees of node 0011 consist of all nodes with NodeId prefixes 1, 01, 000 and 0010 respectively [1]. Figure 1.2: A node's subtrees The Kademlia protocol ensures that every node knows at least one node in each of its subtrees, if that subtree contains a node.

XOR Metric & Distance Calculation

Each Kademlia node has a 160-bit NodeId and the key of every data are also 160-bit identifiers. In order to decide which node a KV pair should be stored at, Kademlia uses the notion of distance between two identifiers [1]. Given two 160-bit identifiers, x and y, Kademlia defines the distance between them as their bitwise exclusive or (XOR) [2] interpreted as an integer $d(x, y) = x \oplus y$. XOR captures the notion of distance implicit to the binary tree sketch of the system. In a fully populated binary tree of 160-bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing both. When a tree is not fully populated, the closest leaf to an ID x is the leaf whose ID share the longest common prefix to x [1]. For example, the distance between 0011 and 1001 would be: $0011 \oplus 1001 = 1010$. 1010 is 10 when interpreted as an integer therefore the distance between these nodes is 10.

Node State

Let us use the term Node Network Data (NND) to represent the following information of a Kademlia node:

- IP Address
- UDP Port
- NodeId

In Kademlia, nodes store contact information about each other for routing query messages. For each $0 < i < 160$, every node keeps a list of NND for nodes of distance between 2^i and 2^{i+1} from itself. These lists are called k-buckets. Therefore, a node will know a set of nodes from each of its subtrees. Each k-bucket is kept in sorted order by time last sent with the least recently seen node at the head and most recently seen at the tail. When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender's NodeId. When updating the k-bucket, there are three scenarios:

- 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.

There are two benefits of Kademlia's k-buckets. Firstly, k-buckets implement a least-recently seen eviction policy, except that live nodes are never removed from the list. Analysis of a P2P system by Saroiu et al. [3] show that the longer a node has been up, the more likely it is to remain up for another hour. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain remain online [1]. The second benefit of k-buckets is that they provide resistance to certain DoS attacks. A malicious user cannot flush nodes' routing state by flooding the system with new nodes. Kademlia nodes will only insert new nodes in the k-buckets when old nodes leave the system. Each Kademlia node also has a Routing Table. A routing table is a binary tree whose leaves are k-buckets. A node's routing table contain all of the node's k-buckets and in turn, all of the node's neighbors from the different subtrees of the node.

Kademlia Protocol

The Kademlia protocol consists of four remote procedure calls (RPCs): PING, STORE, FIND_NODE and FIND_VALUE.

- The PING RPC probes a node to see if it's online.
- The STORE RPC instructs a node to store a [key, value] pair for later retrieval.
- The FIND_NODE RPC takes a 160-bit key as an argument, the recipient of the FIND_NODE RPC returns NND information about the k nodes closest to the target id.
- The FIND_VALUE RPC behaves like FIND_NODE returning the k nodes closest to the target Identifier with one exception – if the RPC recipient has received a STORE for the given key, it just returns the stored value.

Lookup Algorithm

A node lookup in Kademlia is a procedure by which a Kademlia node locates the k closest nodes to some given key. 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 to the key). The initiator then sends parallel, asynchronous FIND_NODE RPCs to the α nodes it has chosen. α is a system-wide concurrency parameter. 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. 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. With the guarantee previously mentioned that every node knows at least one node in each of its subtrees, any node can locate any other node by its NodeId. Figure 1.3: Node Lookup Figure 1.3 shows how Kademlia's lookup algorithm works by demonstrating node 0011 locating node 11110 by querying the closest node to node 11110 and then querying successively closer nodes Most Kademlia operations are implemented in terms of the above lookup procedure. To store a KV pair, a participant locates the k closest nodes to the key and sends them store RPCs. To find a KV 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 RPCs rather than FIND_NODE RPCs. Moreover, the procedure halts immediately when any node returns the value.

Caching

For caching purposes, once a lookup succeeds, the requesting node stores the KV 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,” the expiration time of a KV pair in any node’s database is set to be exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID.

Bucket Refreshing

Kademlia buckets are generally kept fresh by the traffic of requests traveling through nodes. To handle pathological cases in which there are no lookups for a particular ID range, each node refreshes any bucket to which it has not performed a node lookup in the past hour. Refreshing a bucket is done by picking a random ID in the bucket’s range and then performing a NODE_LOOKUP for that ID [1].

To join the network, a node u must have a contact to an already participating node w – usually a bootstrap node is available on every network. 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. When a new node joins the system, it must store any KV pair to which it is one of the k-closest. Existing nodes, by similarly exploiting complete knowledge of their surrounding subtrees, will know which KV pairs the new node should store. Any node learning of a new node therefore issues STORE RPCs to transfer relevant KV pairs to the new node. To avoid redundant store RPCs for the same content from different nodes, a node only transfers a KV pair if its own ID is closer to the key than are the IDs of other nodes.

Key Republishing

To ensure the persistence of KV pairs, nodes must periodically republish keys. Otherwise, two scenarios may cause lookups for valid keys to fail:

- First, some of the k nodes that initially get a key-value pair when it is published may leave the network.
- Second, new nodes may join the network with IDs closer to some published key than the nodes on which the key-value pair was originally published.

In both cases, the nodes with a KV pair must republish it so as once again to ensure it is available on the k nodes closest to the key. To compensate for nodes leaving the network, Kademlia republishes each KV pair once an hour. Kademlia introduces two mechanisms to optimize the key republishing operation to ensure efficient use of computing resources:

- First, when a node receives a store RPC for a given key-value pair, it assumes the RPC was also issued to the other k−1 closest nodes, and thus the recipient will not republish the key-value pair in the next hour. This ensures that as long as republication intervals are not exactly synchronized, only one node will republish a given key-value pair every hour.
- A second optimization avoids performing node lookups before republishing keys. In Kademlia, nodes have complete knowledge of a surrounding subtree with at least k nodes. If, before republishing key-value pairs, a node u refreshes all k-buckets in this subtree of k nodes, it will automatically be able to figure out the k closest nodes to a given key.

Examples

Here we look at a full example of a Kademlia network. @todo – do this @todo – add it as a separate blog article!

Kademlia Summary

In Kademlia the term “node” is used to refer to a node on the DHT network and the term “user” to refer to a user of a DOSN. A user runs the DOSN application on its computer, and a node is also created on this user’s computer and used by the DOSN application to connect to the DHT network. The terms “data”, “content” and “content object” are used to refer to any content or information stored on the DHT. The term “key” is used to refer to a DHT key for any content stored on the DHT.

Kademlia [1] is a P2P DHT. Kademlia keys are opaque, 160-bit quantities. Participating computers each have a key, called a NodeId, in the 160-bit key-space. Kademlia stores content in the form of [key, value] pairs. Each [key, value] pair is stored on nodes with NodeId “close” to the key for some notion of closeness. Finally, a node-ID-based routing algorithm lets anyone efficiently locate servers near any given target key. Kademlia uses parallel, asynchronous queries to avoid timeout delays from failed nodes [1].

Kademlia computes the closeness of keys x and y by taking the integer value of the XOR of the two keys. Kademlia’s lookup algorithm finds successively “closer” nodes to any desired ID, converging to the lookup target in logarithmically many steps. Kademlia treats each node as a leaf on a binary tree, with every node having knowledge of nodes on different sections of the tree [1]. Kademlia uses a replication parameter k which specifies on how many nodes a data should be replicated, as well as the size of a node’s routing table [1]. The node state of a Kademlia node contains its routing table and network information of that node.

Each Kademlia node has a routing table that contains contacts (other DHT nodes) at strategic locations on the DHT to facilitate fast lookups. The routing table is made up of 160 buckets, each storing k contacts at different “distances” away from the node for some notion of distance.

Kademlia’s protocol consists of four remote procedure calls (RPCs): PING, STORE, FIND_NODE and FIND_VALUE. The PING RPC probes a node to see if it’s online. The STORE RPC instructs a node to store a [key, value] pair for later retrieval. The FIND_NODE RPC takes a 160-bit key as an argument, the recipient of the FIND_NODE RPC returns information for the k nodes closest to the target id. The FIND_VALUE RPC behaves like FIND_NODE returning the k nodes closest to the target Identifier with one exception – if the RPC recipient has received a STORE for the key, it just returns the stored value [1].

References:

Petar Maymounkov and David Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*," in IPTPS 2002, 2002, pp. 53-65.

(2014, May) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Exclusive_or

Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," University of Washington, Technical Report UW-CSE-01-06-02, 2001.

Article Tags

Distributed Computing (/article-tags/distributed-computing) DHT (/article-tags/dht)

Peer-To-Peer (/article-tags/peer-peer)

Suggestions

Gleamly
113 likes

Be the first of your friends to like this

Newsletter

Stay informed on our latest news!

E-mail *

Subscribe

Previous issues (/newsletter/gleamly-newsletter)

(/taxonomy/term/2/feed)