WIKIPEDIA

Kademlia

Kademlia is a distributed hash table for decentralized peer-to-peer computer networks designed by Petar Maymounkov and David Mazières in 2002.[1][2] It specifies the structure of the network and the exchange of information through node lookups. Kademlia nodes communicate among themselves using UDP. A virtual or overlay network is formed by the participant nodes. Each node is identified by a number or *node ID*. The *node ID* serves not only as identification, but the Kademlia algorithm uses the *node ID* to locate values (usually file hashes or keywords). In fact, the *node ID* provides a direct map to file hashes and that node stores information on where to obtain the file or resource.

When searching for some value, the algorithm needs to know the associated key and explores the network in several steps. Each step will find nodes that are closer to the key until the contacted node returns the value or no more closer nodes are found. This is very efficient: like many other DHTs, Kademlia contacts only $O(\log(n))$ nodes during the search out of a total of n nodes in the system.

Further advantages are found particularly in the decentralized structure, which increases the resistance against a denial-of-service attack. Even if a whole set of nodes is flooded, this will have limited effect on network availability, since the network will recover itself by knitting the network around these "holes".

Contents

System details

The first generation peer-to-peer file sharing networks, such as Napster, relied on a central database to co-ordinate look ups on the network. Second generation peer-to-peer networks, such as Gnutella, used flooding to locate files, searching every node on the network. Third generation peer-to-peer networks use Distributed hash tables to look up files in the network. *Distributed hash tables* store resource **locations** throughout the network. A major criterion for these protocols is locating the desired nodes quickly.

Kademlia uses a "distance" calculation between two nodes. This distance is computed as the *exclusive or (XOR)* of the two node IDs, taking the result as an integer number. Keys and Node IDs have the same format and length, so distance can be calculated among them in exactly the same way. The node ID is typically a large random number that is chosen with the goal of being unique for a particular node (see UUID). It can and does happen that geographically widely separated nodes—from Germany and Australia, for instance—can be "neighbors" if they have chosen similar random node IDs.

Exclusive or was chosen because it acts as a distance function between all the node IDs. Specifically:

- the distance between a node and itself is zero
- it is symmetric: the "distances" calculated from A to B and from B to A are the same
- it follows the triangle inequality: given A, B and C are vertices (points) of a triangle, then the distance from A to B is shorter than (or equal to) the sum of the distance from A to C plus the distance from C to B.

These three conditions are enough to ensure that *exclusive or* captures all of the essential, important features of a "real" distance function, while being cheap and simple to calculate.[1]

Each Kademlia search iteration comes one bit closer to the target. A **basic** Kademlia network with 2^n nodes will only take *n* steps (in the worst case) to find that node.

Routing tables

This section is simplified to use a single bit; see the section accelerated lookups for more information on real routing tables.

Kademlia routing tables consist of a *list* for each bit of the node ID. (e.g. if a node ID consists of 128 bits, a node will keep 128 such *lists*.) A list has many entries. Every entry in a *list* holds the necessary data to locate another node. The data in each *list* entry is typically the *IP address*, *port*, and *node ID* of another node. Every *list* corresponds to a specific distance from the node. Nodes that can go in the n^{th} *list* must have a differing n^{th} bit from the node's ID; the first n-1 bits of the candidate ID must match those of the node's ID. This means that it is very easy to populate the first *list* as 1/2 of the nodes in the network are far away candidates. The next *list* can use only 1/4 of the nodes in the network (one bit closer than the first), etc.

With an ID of 128 bits, every node in the network will classify other nodes in one of 128 different distances, one specific distance per bit.

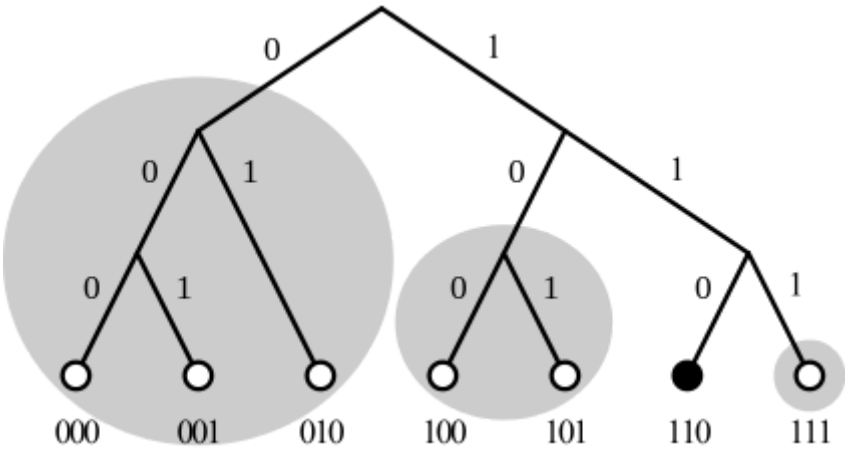
As nodes are encountered on the network, they are added to the *lists*. This includes store and retrieval operations and even helping other nodes to find a key. Every node encountered will be considered for inclusion in the *lists*. Therefore, the knowledge that a node has of the network is very dynamic. This keeps the network constantly updated and adds resilience to failures or attacks.

In the Kademlia literature, the *lists* are referred to as *k-buckets*. *k* is a system wide number, like 20. Every k-bucket is a *list* having up to *k* entries inside; i.e. for a network with k=20, each node will have *lists* containing up to 20 nodes for a particular bit (a particular distance from itself).

Since the possible nodes for each *k-bucket* decreases quickly (because there will be very few nodes that are that close), the lower bit *k-buckets* will fully map all nodes in that section of the network. Since the quantity of possible IDs is much larger than any node population can ever be, some of the k-buckets corresponding to very short distances will remain empty.

Consider the simple network to the right. The network size is 2^3 or eight maximum keys and nodes. There are seven nodes participating; the small circles at the bottom. The node under consideration is node six (binary 110) in black. There are three *k-buckets* for each node in this network. Nodes zero, one and two (binary 000, 001, and 010) are candidates for the farthest *k-bucket* . Node three (binary 011, not shown) is not participating in the network. In the

middle *k-bucket*, nodes four and five (binary 100 and 101) are placed. Finally, the third *k-bucket* can only contain node seven (binary 111). Each of the three *k-buckets* are enclosed in a gray circle. If the size of the *k-bucket* was two, then the farthest *2-bucket* can only contain two of the three nodes. For example, if node six has node one and two in the farthest 2-bucket, it would have to request a node ID lookup to these nodes to find the location (ip address) of node zero. Each node *knows* its neighbourhood well and has contact with a few nodes far away which can help locate other nodes far away.



Network partition for node **110**

It is known that nodes that have been connected for a long time in a network will probably remain connected for a long time in the future.[3][4] Because of this statistical distribution, Kademlia selects long connected nodes to remain stored in the k-buckets. This increases the number of known valid nodes at some time in the future and provides for a more stable network.

When a *k-bucket* is full and a new node is discovered for that *k-bucket*, the least recently seen node in the *k-bucket* is PINGed. If the node is found to be still alive, the new node is placed in a secondary list, a replacement cache. The replacement cache is used only if a node in the *k-bucket* stops responding. In other words: new nodes are used only when older nodes disappear.

Protocol messages

Kademlia has four messages.

- PING — used to verify that a node is still alive.
- STORE — Stores a (key, value) pair in one node.
- FIND_NODE — The recipient of the request will return the k nodes in his own buckets that are the closest ones to the requested key.
- FIND_VALUE — Same as FIND_NODE, but if the recipient of the request has the requested key in its store, it will return the corresponding value.

Each RPC message includes a random value from the initiator. This ensures that when the response is received it corresponds to the request previously sent. (see Magic cookie)

Locating nodes

Node lookups can proceed asynchronously. The quantity of simultaneous lookups is denoted by α and is typically three. A node initiates a FIND_NODE request by querying to the α nodes in its own *k-buckets* that are the closest ones to the desired key. When these recipient nodes receive the request, they will look in their *k-buckets* and return the *k* closest nodes to the desired key that they know. The requester will update a results list with the results (node ID's) he receives, keeping the *k* best ones (the k nodes that are closer to the searched key) that respond to queries. Then the requester will select these *k* best results and issue the request to them, and iterate this process again and again. Because every node has a better knowledge of his own surroundings than any other node has, the received results will be other nodes that are every time closer and closer to the searched key. The iterations continue until no nodes are returned that are closer than the best previous results. When the iterations stop, the best k nodes in the results list are the ones in the whole network that are the closest to the desired key.

The node information can be augmented with round trip times, or RTT. This information will be used to choose a time-out specific for every consulted node. When a query times out, another query can be initiated, never surpassing α queries at the same time.

Locating resources

Information is located by mapping it to a key. A hash is typically used for the map. The storer nodes will have information due to a previous STORE message. Locating a value follows the same procedure as locating the closest nodes to a key, except the search terminates when a node has the requested value in his store and returns this value.

The values are stored at several nodes (k of them) to allow for nodes to come and go and still have the value available in some node. Periodically, a node that stores a value will explore the network to find the k nodes that are close to the key value and replicate the value onto them. This compensates for disappeared nodes.

Also, for popular values that might have many requests, the load in the storer nodes is diminished by having a retriever store this value in some node near, but outside of, the k closest ones. This new storing is called a cache. In this way the value is stored farther and farther away from the key, depending on the quantity of requests. This allows popular searches to find a storer more quickly. Because the value is returned from nodes farther away from the key, this alleviates possible "hot spots". Caching nodes will drop the value after a certain time depending on their distance from the key.

Some implementations (e.g. Kad) do not have replication nor caching. The purpose of this is to remove old information quickly from the system. The node that is providing the file will periodically refresh the information onto the network (perform FIND_NODE and STORE messages). When all of the nodes having the file go offline, nobody will be refreshing its values (sources and keywords) and the information will eventually disappear from the network.

Joining the network

A node that would like to join the net must first go through a bootstrap process. In this phase, the joining node needs to know the IP address and port of another node—a bootstrap node (obtained from the user, or from a stored list)—that is already participating in the Kademlia network. If the joining node has not yet participated in the network, it computes a random ID number that is supposed not to be already assigned to any other node. It uses this ID until leaving the network.

The joining node inserts the bootstrap node into one of its *k-buckets*. The joining node then performs a node lookup of its own ID against the bootstrap node (the only other node it knows). The "self-lookup" will populate other nodes' *k-buckets* with the new node ID, and will populate the joining node's k-buckets with the nodes in the path between it and the bootstrap node. After this, the joining node refreshes all *k-buckets* further away than the k-bucket the bootstrap node falls in. This refresh is just a lookup of a random key that is within that *k-bucket* range.

Initially, nodes have one *k-bucket*. When the *k-bucket* becomes full, it can be split. The split occurs if the range of nodes in the *k-bucket* spans the node's own id (values to the left and right in a binary tree). Kademlia relaxes even this rule for the one "closest nodes" *k-bucket*, because typically one single bucket will correspond to the distance where all the nodes that are the closest to this node are, they may be more than k, and we want it to know them all. It may turn out that a highly unbalanced binary sub-tree exists near the node. If *k* is 20, and there are 21+ nodes with a prefix "xxx0011....." and the new node is "xxx0000*11001*", the new node can contain multiple *k-buckets* for the other 21+ nodes. This is to guarantee that the network knows about all nodes in the closest region.

Accelerated lookups

Kademlia uses an *XOR metric* to define distance. Two node ID's or a node ID and a key are XORed and the result is the distance between them. For each bit, the XOR function returns zero if the two bits are equal and one if the two bits are different. XOR metric distances hold the triangle inequality: given A, B and C are vertices (points) of a triangle, then the distance from A to B is shorter than (or equal to) the sum of the distance from A to C to B.

The *XOR metric* allows Kademlia to extend routing tables beyond single bits. Groups of bits can be placed in *k-buckets*. The group of bits are termed a prefix. For an *m-bit* prefix, there will be 2^m-1 *k-buckets*. The missing *k-bucket* is a further extension of the routing tree that contains the node ID. An *m-bit* prefix reduces the maximum number of lookups from $log_2\ n$ to $log_{2^m}\ n$. These are **maximum** values and the average value will be far less, increasing the chance of finding a node in a *k-bucket* that shares more bits than just the prefix with the target key.

Nodes can use mixtures of prefixes in their routing table, such as the Kad Network used by eMule. The Kademlia network could even be heterogeneous in routing table implementations, at the expense of complicating the analysis of lookups.

Academic significance

While the XOR metric is not needed to understand Kademlia, it is critical in the analysis of the protocol. The XOR arithmetic forms an abelian group allowing closed analysis. Other DHT protocols and algorithms require simulation or complicated formal analysis in order to predict network behavior and correctness. Using groups of bits as routing information also simplifies the algorithms.

Mathematical analysis of the algorithm

To analyze the algorithm, consider a Kademlia network of n nodes with IDs $x_1, \ldots, x_n$, each of which is a string of length d that consists of only ones and zeros. It can be modeled as a trie, in which each leaf represents a node, and the labeled path from the root to a leaf represents its ID. For a node $x \in \{x_1, \ldots, x_n\}$, let $\mathcal{D}_i(x)$ be the set of nodes (IDs) that share a prefix with x of length $d - i$. Then filling the i-th bucket of x can be modeled as adding pointers from the leaf x to k leaves (IDs) chosen uniformly at random from $\mathcal{D}_i(x)$. Thus routing can be seen as jumping among the leaves along these pointers such that each step goes towards the target ID as much as possible, i.e., in a greedy way.

Let T_{xy} be number of jumps needed to go from the leaf x to a target ID y. Assuming that $x_1, \ldots, x_n$ are chosen deterministically from $\{0,1\}^d$, it has been proved that

$$\sup_{x_1,\ldots,x_n} \sup_{x \in \{x_1,\ldots,x_n\}} \sup_{y \in \{0,1\}^d} \mathbb{E}[T_{xy}] \leq (1 + o(1)) \frac{\log n}{H_k},$$

where H_k is the k-th Harmonic Number. Since $H_k / \log k \to 1$ as $k \to \infty$, when k is large $\mathbb{E}T_{xy}$ is bounded from above by about $\log_k n$, however the IDs and the target are chosen.[5] This justifies the intuition that in Kademlia only $O(\log n)$ nodes are contacted in searching for a target node.

To make the model closer to real Kademlia networks, $x_1, \ldots, x_n$ can also be assumed to be chosen uniformly at random without replacement from $\{0,1\}^d$. Then it can be proved that for all $x \in \{x_1, \ldots, x_n\}$ and $y \in \{0,1\}^d$,

$$T_{xy} \xrightarrow{p} \frac{\log n}{c_k},$$
$$\mathbb{E}[T_{xy}] \to \frac{\log n}{c_k},$$

where c_k is a constant depending only on k with $c_k/H_k \to 1$ as $k \to \infty$. Thus for k large, $\mathbb{E}T_{xy}/\log_k n$ converges to a constant close 1. This implies that the number of nodes need to be contact in searching for a target node is actually $\Theta(\log_k n)$ on average.[6]

Use in file sharing networks

Kademlia is used in file sharing networks. By making Kademlia keyword searches, one can find information in the file-sharing network so it can be downloaded. Since there is no central instance to store an index of existing files, this task is divided evenly among all clients: If a node wants to share a file, it processes the contents of the file, calculating from it a number (hash) that will identify this file within the file-sharing network. The hashes and the node IDs must be of the same length. It then searches for several nodes whose ID is close to the hash, and has its own IP address stored at those nodes. i.e. it publishes itself as a source for this file. A searching client will use Kademlia to search the network for the node whose ID has the smallest distance to the file hash, then will retrieve the sources list that is stored in that node.

Since a key can correspond to many values, e.g. many sources of the same file, every storing node may have different information. Then, the sources are requested from all k nodes close to the key.

The file hash is usually obtained from a specially formed Internet magnet link found elsewhere, or included within an indexing file obtained from other sources.

Filename searches are implemented using keywords. The filename is divided into its constituent words. Each of these keywords is hashed and stored in the network, together with the corresponding filename and file hash. A search involves choosing one of the keywords, contacting the node with an ID closest to that keyword hash, and retrieving the list of filenames that contain the keyword. Since every filename in the list has its hash attached, the chosen file can then be obtained in the normal way.

Implementations

Networks

Public networks using the Kademlia algorithm (these networks are incompatible with one another):

- I2P[7]
- Kad Network — developed originally by the eMule community to replace the server-based architecture of the eDonkey2000 network.
- Ethereum — The node discovery protocol in Ethereum's blockchain network stack is based a slightly modified implementation of Kademlia.[8]
- Overnet network: With KadC a C library for handling its Kademlia is available. (development of Overnet is discontinued)
- BitTorrent Uses a DHT based on an implementation of the Kademlia algorithm, for trackerless torrents.
- Osiris sps (all version): used to manage distributed and anonymous web portal.
- Retroshare — F2F decentralised communication platform with secure VOIP, instant messaging, file transfer etc.
- Tox - A fully distributed messaging, VoIP and video chat platform
- Gnutella DHT — Originally by LimeWire[9][10] to augment the Gnutella protocol for finding alternate file locations, now in use by other gnutella clients.[11]
- IPFS – A peer-to-peer distributed filesystem.[12]
- TeleHash is a mesh networking protocol that uses Kademlia to resolve direct connections between parties.[13]

See also

- Content addressable network

- Chord (DHT)
- Tapestry (DHT)
- Pastry (DHT)
- Koorde

References

1. *Kademlia: A Peer-to-peer information system based on the XOR Metric (http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf)
2. http://www.scs.stanford.edu/~dm/home/papers/
3. 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.
4. Daniel Stutzbach and Reza Rejaie. Understanding Churn in Peer-to-Peer Networks (http://www.barsoom.org/papers/imc-2006-churn.pdf) Section 5.5 Uptime Predictability, Internet Measurement Conference, Rio de Janeiro, October, 2006.
5. Cai, X. S.; Devroye, L. (2013). "A Probabilistic Analysis of Kademlia Networks". *Algorithms and Computation*. Lecture Notes in Computer Science. **8283**: 711. arXiv:1309.5866 (https://arxiv.org/abs/1309.5866). doi:10.1007/978-3-642-45030-3_66 (https://doi.org/10.1007%2F978-3-642-45030-3_66). ISBN 978-3-642-45029-7.
6. Cai, Xing Shi; Devroye, Luc (2015). "The Analysis of Kademlia for Random IDs". *Internet Mathematics*. **11**: 1–16. arXiv:1402.1191 (https://arxiv.org/abs/1402.1191). doi:10.1080/15427951.2015.1051674 (https://doi.org/10.1080%2F15427951.2015.1051674). ISSN 1542-7951 (https://www.worldcat.org/issn/1542-7951).
7. https://geti2p.net/en/about/intro
8. Ethereum wiki for Kademlia peer selection (https://github.com/ethereum/wiki/wiki/Kademlia-Peer-Selection)
9. Mojito and LimeWire (http://www.slyck.com/story1235.html)
10. Mojito Wiki (archive.org) (https://web.archive.org/web/20090217070609/http://wiki.limewire.org/index.php?title=Mojito)
11. "Gtk-gnutella changelog" (https://web.archive.org/web/20110723210536/https://gtk-gnutella.svn.sourceforge.net/svnroot/gtk-gnutella/trunk/gtk-gnutella/ChangeLog). *sourceforge.net*. Archived from the original (https://gtk-gnutella.svn.sourceforge.net/svnroot/gtk-gnutella/trunk/gtk-gnutella/ChangeLog) on 23 July 2011. Retrieved 23 January 2010.
12. IPFS Paper (https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf)
13. Francis Irving interviews Jeremie Miller. "#7: Jeremie Miller - TeleHash" (http://redecentralize.org/interviews/2013/10/17/07-jeremie-telehash.html). access=date = 2016-03-12.

External links

- Xlattice projects (http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html) Kademlia Specification and definitions.