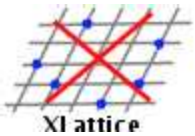


PDF

Kademlia: A Design Specification

Introduction

Kademlia is a communications protocol for peer-to-peer networks. It is one of many versions of a DHT, a **Distributed Hash Table**.

Network Characterization

A Kademlia network is characterized by three constants, which we call **alpha**, **B**, and **k**. The first and last are standard terms. The second is introduced because some Kademlia implementations use a different key length.

- **alpha** is a small number representing the degree of parallelism in network calls, usually **3**
- **B** is the size in bits of the keys used to identify nodes and store and retrieve data; in basic Kademlia this is **160**, the length of an SHA1 digest (hash)
- **k** is the maximum number of contacts stored in a bucket; this is normally **20**

It is also convenient to introduce several other constants not found in the original Kademlia papers.

- **tExpire** = 86400s, the time after which a key/value pair expires; this is a time-to-live (TTL) from the *original* publication date
- **tRefresh** = 3600s, after which an otherwise unaccessed bucket must be refreshed
- **tReplicate** = 3600s, the interval between Kademlia replication events, when a node is required to publish its entire database
- **tRepublish** = 86400s, the time after which the original publisher must republish a key/value pair

> **Note**
>
> The fact that tRepublish and tExpire are equal introduces a race condition. The STORE for the data being published may arrive at the node just after it has been expired, so that it will actually be necessary to put the data on the wire. A sensible implementation would have tExpire significantly longer than tRepublish. Experience suggests that tExpire=86410 would be sufficient.

The Node

A Kademlia network consists of a number of cooperating **nodes** that communicate with one another and store information for one another. Each node has a **nodeID**, a quasi-unique binary number that identifies it in the network.

Within the network, a block of data, a **value**, can also be associated with a binary number of the same fixed length B, the value's **key**.

A node needing a value searches for it at the nodes it considers closest to the key. A node needing to save a value stores it at the nodes it considers closest to the key associated with the value.

NodeID

NodeIDs are binary numbers of length B = 160 bits. In basic Kademlia, each node chooses its own ID by some unspecified quasi-random procedure. It is important that nodeIDs be uniformly distributed; the network design relies upon this.

While the protocol does not mandate this, there are possibleadvantages to the node's using the same nodeID whenever it joins the network, rather than generating a new, session-specific nodeID.

Keys

Data being stored in or retrieved from a Kademlia network must also have a key of length B. These keys should also be uniformly distributed. There are several ways to guarantee this; the most common is to take a hash, such as the 160 bit SHA1 digest, of the value.

Distance: the Kademlia Metric

Kademlia's operations are based upon the use of exclusive OR, XOR, as a metric. The distance between any two keys or nodeIDs x and y is defined as

```
distance(x, y) = x ^ y
```

where ^ represents the XOR operator. The result is obtained by taking the bytewise exclusive OR of each byte of the operands.

> **Note**
>
> Kademlia follows Pastry in interpreting keys (including nodeIDs) as **bigendian** numbers. This means that the low order byte in the byte array representing the key is the most significant byte and so if two keys are close together then the low order bytes in the distance array will be zero.

The K-Bucket

A Kademlia node organizes its **contacts**, other nodes known to it, in **buckets** which hold a maximum of k contacts. These are known as k-buckets.

The buckets are organized by the distance between the node and the contacts in the bucket. Specifically, for bucket j, where 0 <= j < k, we are guaranteed that

```
2^j <= distance(node, contact) < 2^(j+1)
```

Given the very large address space, this means that bucket zero has only one possible member, the key which differs from the nodeID only in the high order bit, and for all practical purposes is never populated, except perhaps in testing. On other hand, if nodeIDs are evenly distributed, it is very likely that half of all nodes will lie in the range of bucket B-1 = 159.

Bucket Size

The Kademlia paper says that k is set to a value such that it is very unlikely that in a large network all contacts in any one bucket will have disappeared within an hour. Anyone attempting to calculate this probability should take into consideration policies that lead to long-lived contacts being kept in the table in preference to more recent contacts.

Contacts

A **contact** is at least a triple:

- the bigendian **nodeID** for the other node
- its IP address
- its UDP port address

The IP address and port address should also be treated as bigendian numbers.

Kademlia's designers do not appear to have taken into consideration the use of IPv6 addresses or TCP/IP instead of UDP or the possibility of a Kademlia node having multiple IP addresses.

Sorting

Within buckets contacts are sorted by the time of the most recent communication, with those which have most recently communicated at the end of the list and those which have least recently communicated at the front, regardless of whether the node or the contact initiated the sequence of messages.

Updates

Whenever a node receives a communication from another, it updates the corresponding bucket. If the contact already exists, it is moved to the end of the bucket. Otherwise, if the bucket is not full, the new contact is added at the end. If the bucket is full, the node pings the contact at the head of the bucket's list. If

that least recently seen contact fails to respond in an *(unspecified)* reasonable time, it is dropped from the list, and the new contact is added at the tail. Otherwise the new contact is ignored for bucket updating purposes.

> **Warning**
>
> In a large, busy network, it is possible that while a node is waiting for a reply from the contact at the head of the list there will be another communication from a contact not in the bucket. This is most likely for bucket B-1 = 159, which is responsible for roughly half of the nodes in the network. Behaviour in this case is unspecified and seems likely to provide an opening for a DOS (Denial of Service) attack.

Rationale

Experience has shown that nodes tend to group into two clearly distinguished categories, the transient and the long-lived. This update policy gives strong preference to the long-lived and so promotes network stability. It also provides a degree of protection from certain types of denial of service (DOS) attacks, including, possibly, Sybil attacks, discussed below.

Protocol

The original Kademlia paper, maymo02, says that the Kademlia protocol consists of four remote procedure calls ("RPCs") but then goes on to specify procedures that must be followed in executing these as well as certain other protocols. It seems best to add these procedures and other protocols to what we call here the Kademlia protocol.

PING

This RPC involves one node sending a PING message to another, which presumably replies with a PONG.

This has a two-fold effect: the recipient of the PING must update the bucket corresponding to the sender; and, if there is a reply, the sender must update the bucket appropriate to the recipient.

All RPC packets are required to carry an RPC identifier assigned by the sender and echoed in the reply. This is a quasi-random number of length B (160 bits).

> **Note**
>
> Implementations using shorter message identifiers must consider the **birthday paradox**, which in effect makes the probability of a collision depend upon half the number of bits in the identifier. For example, a 32-bit RPC identifier would yield a probability of collision proportional to 2^-16, an uncomfortably small number in a busy network.
>
> If the identifiers are initialized to zero or are generated by the same random number generator with the same seed, the probability will be very high indeed.

It must be possible to piggyback PINGs onto RPC replies to force or permit the originator, the sender of the RPC, to provide additional information to its recipient. **This might be a different IP address or a preferred protocol for future communications.**

STORE

The sender of the STORE RPC provides a key and a block of data and requires that the recipient store the data and make it available for later retrieval by that key.

This is a primitive operation, not an iterative one.

> **Note**
>
> While this is not formally specified, it is clear that the initial STORE message must contain in addition to the message ID at least the data to be stored (including its length) and the associated key. As the transport may be UDP, the message needs to also contain at least the nodeID of the sender, and the reply the nodeID of the recipient.
>
> The reply to any RPC should also contain an indication of the result of the operation. For example, in a STORE while no maximum data length has been specified, it is clearly possible that the receiver might not be able to store the data, either because of lack of space or because of an I/O error.

FIND_NODE

The FIND_NODE RPC includes a 160-bit key. The recipient of the RPC returns up to k triples (IP address, port, nodeID) for the contacts that it knows to be closest to the key.

The recipient must return k triples if at all possible. It may only return fewer than k if it is returning all of the contacts that it has knowledge of.

This is a primitive operation, not an iterative one.

> **Note**
>
> The name of this RPC is misleading. Even if the key to the RPC is the nodeID of an existing contact or indeed if it is the nodeID of the recipient itself, the recipient is still required to return k triples. A more descriptive name would be FIND_CLOSE_NODES.
>
> The recipient of a FIND_NODE should never return a triple containing the nodeID of the requestor. If the requestor does receive such a triple, it should discard it. A node must never put its own nodeID into a bucket as a contact.

FIND_VALUE

A FIND_VALUE RPC includes a B=160-bit key. If a corresponding value is present on the recipient, the associated data is returned. Otherwise the RPC is equivalent to a FIND_NODE and a set of k triples is returned.

This is a primitive operation, not an iterative one.

Node Lookup

This section describes the algorithm that Kademlia uses for locating the k nodes nearest to a key. It must be understood that these are not necessarily closest in a strict sense. Also, the algorithm is **iterative** although the paper describes it as recursive.

The search begins by selecting alpha contacts from the non-empty k-bucket closest to the bucket appropriate to the key being searched on. If there are fewer than alpha contacts in that bucket, contacts are selected from other buckets. The contact closest to the target key, **closestNode**, is noted.

> **Note**
>
> The criteria for selecting the contacts within the closest bucket are not specified. Where there are fewer than alpha contacts within that bucket and contacts are obtained from other buckets, there are no rules for selecting the other buckets or which contacts are to be used from such buckets.

The first alpha contacts selected are used to create a **shortlist** for the search.

The node then sends parallel, asynchronous FIND_* RPCs to the alpha contacts in the shortlist. Each contact, if it is live, should normally return k triples. If any of the alpha contacts fails to reply, it is removed from the shortlist, at least temporarily.

The node then fills the shortlist with contacts from the replies received. These are those closest to the target. From the shortlist it selects another alpha contacts. The only condition for this selection is that they have not already been contacted. Once again a FIND_* RPC is sent to each in parallel.

Each such parallel search updates **closestNode**, the closest node seen so far.

The sequence of parallel searches is continued until either no node in the sets returned is closer than the closest node already seen or the initiating node has accumulated k probed and known to be active contacts.

If a cycle doesn't find a closer node, if **closestNode** is unchanged, then the initiating node sends a FIND_* RPC to each of the k closest nodes that it has not already queried.

At the end of this process, the node will have accumulated a set of k active contacts or (if the RPC was FIND_VALUE) may have found a data value. Either a set of triples or the value is returned to the caller.

> **Note**
>
> The original algorithm description is not clear in detail. However, it appears that the initiating node maintains a **shortlist** of k closest nodes. During each iteration alpha of these are selected for probing and marked accordingly. If a probe succeeds, that shortlisted node is marked as active. If there is no reply after an unspecified period of time, the node is dropped from the shortlist. As each set of replies comes back, it is used to improve the shortlist: closer nodes in the reply replace more distant (unprobed?) nodes in the shortlist. Iteration continues until k nodes have been successfully probed or there has been no improvement.

Alpha and Parallelism

Kademlia uses a value of 3 for alpha, the degree of parallelism used. It appears that (see stutz06) this value is optimal.

There are at least three approaches to managing parallelism. The first is to launch alpha probes and wait until all have succeeded or timed out before iterating. This is termed **strict parallelism**. The second is to limit the number of probes in flight to alpha; whenever a probe returns a new one is launched. We might call this **bounded parallelism**. A third is to iterate after what seems to be a reasonable delay (duration unspecified), so that the number of probes in flight is some low multiple of alpha. This is **loose parallelism** and the approach used by Kademlia.

iterativeStore

This is the Kademlia store operation. The initiating node does an iterativeFindNode, collecting a set of k closest contacts, and then sends a primitive STORE RPC to each.

iterativeStores are used for publishing or replicating data on a Kademlia network.

iterativeFindNode

This is the basic Kademlia node lookup operation. As described above, the initiating node builds a list of k "closest" contacts using iterative node lookup and the FIND_NODE RPC. The list is returned to the caller.

iterativeFindValue

This is the Kademlia search operation. It is conducted as a node lookup, and so builds a list of k closest contacts. However, this is done using the FIND_VALUE RPC instead of the FIND_NODE RPC. If at any time during the node lookup the value is returned instead of a set of contacts, the search is abandoned and

the value is returned. Otherwise, if no value has been found, the list of k closest contacts is returned to the caller.

When an iterativeFindValue succeeds, the initiator must store the key/value pair at the closest node seen which did **not** return the value.

Refresh

If no node lookups have been performed in any given bucket's range for tRefresh (an hour in basic Kademlia), the node selects a random number in that range and does a **refresh**, an iterativeFindNode using that number as key.

Join

A node joins the network as follows:

1. if it does not already have a nodeID **n**, it generates one
2. it inserts the value of some known node **c** into the appropriate bucket as its first contact
3. it does an iterativeFindNode for **n**
4. it refreshes all buckets further away than its closest neighbor, which will be in the occupied bucket with the lowest index.

If the node saved a list of good contacts and used one of these as the "known node" it would be consistent with this protocol.

Replication Rules

- Data is stored using an **iterativeStore**, which has the effect of replicating it over the k nodes closest to the key.
- Each node republishes each key/value pair that it contains at intervals of tReplicate seconds (every hour). The republishing node must not be seen as the original publisher of the key/value pair.
- The original publisher of a key/value pair republishes it every tRepublish seconds (every 24 hours).
- When an iterativeFindValue succeeds, the initiator must store the key/value pair at the closest node seen which did **not** return the value.

Expiration Rules

- All key/value pairs expire tExpire seconds (24 hours) after the original publication.
- All key/value pairs are assigned an **expiration time** which is "exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key", where this number is "inferred from the bucket structure of the current node".

The writer would calculate the expiration time when the key/value pair is stored using something similar to the following:

- find the index **j** of the bucket corresponding to the key
- count the total number of contacts **Ca** in buckets 0..j-1
- count the number of contacts **Cb** in bucket **j** closer than the key
- if **C** = Ca + Cb, then the interval to the expiration time is
    - 24 hours if C > k
    - 24h * exp(k / C) otherwise

> **Note**
>
> The requirement that data expires tExpire (one day) after the *original* publication date is more than ambiguous and would seem to mean that no data can ever be republished.
>
> In any case, the system is required to mark the stored key/value pair with an original publication timestamp. If this is to be accurate, the timestamp must be set by the publisher, which means that clocks must be at least loosely synchronized across the network.
>
> It would seem sensible to mark the key/value pair with a time to live (TTL) from the arrival of the data, tExpire (one day) or a fraction thereof.

Implementation Suggestions

Contact

It would seem useful to add to the Contact data structure at least:

- an RTT (round trip time) value or a set of such values, measured in ms
- more IP addresses, together with perhaps
    - protocol used (TCP/IP, UDP)
    - NAT information, if applicable
    - whether the address is local and so reachable by broadcast

Adding an RTT or set of RTTs to the Contact data structure would enable better decisions to be made when selecting which to use.

The round trip time (RTT) to the contact could be as measured using a PING RPC or using a conventional Internet network ping.

Possible Convoy Effects

Implementors should take care to avoid convoy effects. These occur when a number of processes need to use a resource in turn. There is a tendency for such bursts of activity to drift towards synchronization, which can be disasterous. In Kademlia all nodes are requird to republish their contents every hour (tReplicate). A convoy effect might lead to this being synchronized across the network, which would appear to users as the network dying every hour.

Random Number Generation

Implementors should remember that random number generators are usually not re-entrant and so access from different threads needs to be synchronized.

Also, beware of clock granularity: it is possible that where the clock is used to seed the random number generator, successive calls could use the same seed.

STORE

For efficiency, the STORE RPC should be two-phase. In the first phase the initiator sends a key and possibly length and the recipient replies with either something equivalent to OK or a code signifying that it already has the value or some other status code. If the reply was OK, then the initiator may send the value.

Some consideration should also be given to the development of methods for handling hierarchical data. Some values will be small and will fit in a UDP datagram. But some messages will be very large, over say 5 GB, and will need to be chunked. The chunks themselves might be very large relative to a UDP packet,

typically on the order of 128 KB, so these chunks will have to be shredded into individual UDP packets.

tExpire

As noted earlier, the requirement that tExpire and tRepublish have the same value introduces a race condition: data will frequently be republished immediately after expiration. It would be sensible to make the expiration interval tExpire somewhat greater than the republication interval tRepublish. The protocol should certainly also allow the recipient of a STORE RPC to reply that it already has the data, to save on expensive network bandwidth.

Possible Problems with Kademlia

The Sybil Attack

A paper by John Douceur, douceur02, describes a network attack in which attackers select nodeIDs whose values enable them to position themselves in the network in patterns optimal for disrupting operations. For example, to remove a data item from the network, attackers might cluster around its key, accept any attempts to store the key/value pair, but never return the value when presented with the key.

A Sybil variation is the **Spartacus attack**, where an attacker joins the network claiming to have the same nodeID as another member. As specified, Kademlia has no defense. In particular, a long-lived node can always steal a short-lived node's nodeID.

Douceur's solution is a requirement that all nodes get their nodeIDs from a central server which is responsible at least for making sure that the distribution of nodeIDs is even.

A weaker solution would be to require that nodeIDs be derived from the node's network address or some other quasi-unique value.