DHT

Introduction

Distributed Hash Table:

- Implements a key-value mapping semantic
- Each item assigned to at least one owner node
- Has two main issues:
  1. How to determine the owner of an item
  2. How to find the owner of an item
DHT

Introduction

Common approach for DHT:

1. Determine the owner by hashing the key and the nodes IDs. The "closest" node is the item's owner
   - Requires to define a distance metric

2. Key based routing (KBR), enables a node that is not an owner of an item to forward a lookup request for this object to another node such that eventually, if the value exists, it will be found.
Kademlia

Introduction

Kademlia is a peer-to-peer structured DHT.

- Low latency queries ($O(\log N)$ lookup time)
- Configuration information spreads automatically as a side-effect of key lookup
- Uses parallel asynchronous queries to avoid delays caused by failed nodes.
- Resistant to some kinds of DoS attacks
- A novel symmetric XOR metric
  - Allows nodes to receive lookup queries from the same distribution of nodes contained in their routing table and to learn useful routing information from queries they received.
- A single routing algorithm
- Base protocol for many popular P2P systems
  - BitTorrent
  - eMule
System Description

System Layout

Kademlia treat nodes as leaves in a binary tree

- Each node is assigned with 160-bit node ID
- A node position is determined by the shortest unique prefix of its ID
- Keys in the system are hashed to a 160-bit words (using SHA1)
Kademlia binary tree from 0011 point of view. The black dot shows the location of node 0011 in the tree. Gray ovals show the subtrees in which 0011 must have contacts.
System Description

System Layout

From a node point of view, the tree is divided into series of successively sub-trees

- The highest subtree consists of the half of the tree that not containing the node
- The next subtree consists of the half of the remaining tree not containing the node
- ..
- The 160\textsuperscript{th} subtree contains only the node

The Kademlia protocol ensures that each node knows of at least one node on each of its sub-trees.

- With this guarantee, a node can locate any other node by its ID
Kademlia define the distance between two items $x, y$ as follows:

$$d(x, y) = x \oplus y$$

interpreting the result as an integer.

- A key to an object is a 160-bit long (using SHA1)
- The owner of an object is the closest node according to the above metric

XOR metric has the following properties:

- $d(x, x) = 0$
- $d(x, y) > 0$ if $x \neq y$
- $\forall x, y : d(x, y) = d(y, x)$
- $\forall x, y, z : d(x, y) + d(y, z) \leq d(x, z)$
- $\forall x, \Delta > 0$ there exists exactly one $y$ such that $d(x, y) = \Delta$
System Description
Distance Metric

Because of its properties, XOR captures the notion of distance implicit in the binary tree:

- In a fully populated tree, the distance between two nodes is bounded by the height of the smallest subtree contains them both.
- In a not fully populated tree, the closest leaf to a given ID $x$ is the leaf whose ID shares the longest common prefix of $x$.
- All lookups for the same key converge along the same path. Hence, caching $\langle key, value \rangle$ pairs along the lookup path alleviates hot spots.
Node State

k-bucket

For each $0 \leq i < 160$ every node keeps a list of $\langle ip, port, id \rangle$ for nodes in distance between $2^i$ to $2^{i+1}$. This list is called **k-bucket**.

- k-bucket stores at most $k$ records
- The head contains the least-recently seen node
- The tail contains the last seen node
- $k$ is a system wide replication parameter and chosen such that any given $k$ nodes are unlikely to fail
- Note that each k-bucket is corresponding to a subtree

For smalls $i$ the bucket is likely to be empty (why?)
When a Kademlia node receives any message from another node it updates the k-bucket with that node details according to the following policy:

- If the node is already in the bucket it moved to the bucket’s tail
- If the node is not in the bucket and the bucket is not full, the node is entered to the bucket’s tail.
- If the node is not in the bucket and the bucket is full:
  - The receiver pings the bucket’s head
  - If it fails to respond it is evicted and the new node inserted to the tail
  - Else, the head is moved to the tail and the new node is discarded.

The preference for already known nodes (over new ones) is due to empirical results.

Another important aspect of the above policy is resistance to Routing Table Poisoning (RTP) attacks in which the attacker tries to change the routing tables.
Kademlia Protocol

Overview

0011 performs lookup for 1110 by recursively learn and query closer and closer nodes.
Kademlia Protocol

RPCs

Kademlia protocol consists of four RPCs:

- **PING** - probes a node to see if it is still alive
- **STORE** - stores a \( \langle key, value \rangle \) pair for later retrieval
- **FIND_NODE** - receives an \( id \) and returns a list of \( k \) \( \langle ip, port, id' \rangle \) triples for the \( k \) closest nodes to \( id \) it knows about. Those triples might belong to either the same or multiple buckets. If the node does not know of \( k \) nodes it returns all the nodes it knows about.
- **FINE_VALUE** - behave the same as FIND_NODE except that if the RPC receiver is the owner of the value (it was instructed previously to STORE \( \langle key, value \rangle \)) it returns the value
Kademlia Protocol

Node Lookup

A node lookup is a recursive procedure for finding an object ID in Kademlia. It takes an id as an argument and return the k most closest nodes to id. The procedure performs the following recursive steps:

1. If all id’s k closest \(^1\) nodes were already queried about id and returned an answer, return a list of triples \(\langle ip, port, id' \rangle\) of id’s k closest nodes.

2. Pick the id’s \(\alpha\) closest \(^2\) nodes, that were not queried yet, and initiates in parallel the FIND_NODE RPC.

3. If new closest nodes were discovered, repeat.

4. Else, for every node among the k closest nodes that were not queried yet, initiates in parallel the FIND_NODE RPC and repeat.

\(^1\) nodes that fail to respond quickly enough are removed from consideration.

\(^2\) \(\alpha\) is a system-wide concurrency parameter.
Kademlia Protocol

Operations

Most Kademlia operations uses the lookup algorithm:

- To store a \(\langle key, value\rangle\) a node initiates a lookup with \(key\) and then initiates the STORE RPC on the \(k\) most closest nodes to \(key\).
- To find a \(\langle key, value\rangle\), a node performs a lookup with \(key\). However, value lookup uses FIND_VALUE instead of FIND_NODE.

For caching propose, once a value lookup succeeds, the initiator invokes the STORE RPC with \(\langle key, value\rangle\) on the closet node is observed that did not return the value - spreading a popular values to more nodes. This mechanism handling over-caching by expiration date for each pair which, in return, forces the nodes to republish their values periodically.

- Why is it important to re-publish values anyway?
Kademlia Protocol

Refreshing buckets

To handle the case in which there are no lookups for a particular ID range, a node must refresh the buckets to which it did not recently performed a node lookup (an hour).

- Refreshing means picking a random ID in the bucket’s range and run a lookup for it.
Kademlia Protocol

Joining Kademlia

To join the network, a node must be familiar with at least one already participant node. A node $p$ that wishes to join the network (and knows an already participant node $q$) performs the following:

1. Insert $q$ to the appropriate k-bucket
2. Perform lookup with $p$ (not $q$)
3. Refresh all k-buckets further away than the closest neighbours

Step 2 makes $p$ known to its closest nodes while step 3 do both, populating $p$’s buckets and making $p$ known to other nodes.
Routing Table

Overview

The routing table is a binary tree whose its leaves are known k-buckets (this structure is highly reasonable considering Kademlia’s structure). Each sub-table (subtree) contains all nodes with the largest common prefix of their IDs.

- The routing table overlaps the entire IDs space
- A node cannot be present in two different sub-tables
Routing Table

Building Table

Routing table for a node whose ID is 00...00

11...11  Space of 160-bit ID numbers  00...00

```
1 0
  
1 0
  
1 0
  
1 0
  
1 0
```

Yehonatan Buchnik (Based on Dolev Adas slides)

Distributed Systems 236351

December 13, 2018, 19 / 23
Routing Table

Building Table

The node initially starts with a single routing table that contains only itself and covers the whole IDs space. Upon discovery of a new node $q$, a node $p$ takes the following steps:

1. If $q$’s appropriate bucket already contains $p$, split the bucket into two new buckets (dividing the content accordingly) and repeat

2. Else, if $q$’s appropriate bucket is not full yet, insert $q$. 
Routing Table

Building Table

The above method imposes a problem with an unbalanced trees.

- Consider a node $p$ with unique ID prefix; as such, every node that discovers $p$ allocates a new bucket for $p$
- $p$ itself locates all nodes in the same bucket
  - But the k-bucket is only of size $k$
  - $p$ does not know nodes who does know it - heavily breaking symmetry and connectivity of the network
- To overcome the problem, Kademlia splits a bucket even if its node ID does not resides in it
Relaxed routing table for a node whose ID is 00...00

Space of 160-bit numbers (used for node IDs and keys)
Additional materials