Kademlia

@arnaucube

2019-04-26

Overview

- nodes self sets a random unique ID (UUID)
- nodes are grouped in `neighbourhoods` determined by the `node ID` distance
- Kademlia uses `distance` calculation between two nodes
    - distance is computed as XOR (exclusive or) of the two `node ID`s

- XOR acts as the distance function between all `node ID`s. Why:
  - distance between a node and itself is zero
  - is symmetric: distance between A to B is the same to B to A
  - follows `triangle inequality`
    - given A, B, C vertices (points) of a triangle
    - AB <= (AC + CB)
      - the distance from A to B is shorter or equal to the sum of the distance from A to C plus the distance from C to B
    - so, we get the shortest path

- with that last 3 properties we ensure that XOR
  - captures all of the essential & important features of a "real" distance function
  - is simple and cheap to calculate
- each search iteration comes one bit closer to the target
  - a basic Kademlia network with `2^n` nodes will only take `n` steps (in worst case) to find that node

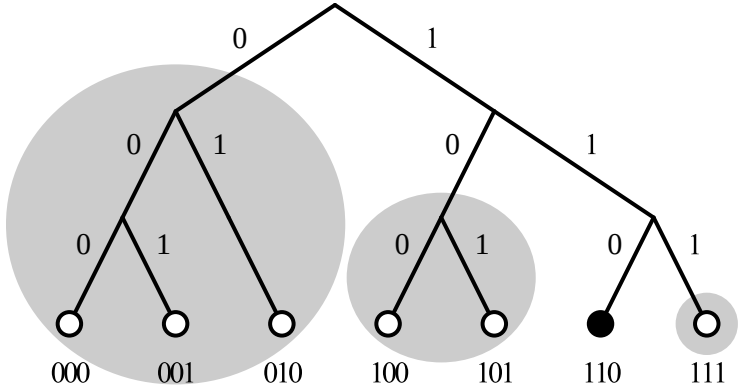
Routing tables

- each node has a routing table, that consists of a `list` for each bit of the `node ID`
  - each entry holds the necessary data to locate another node
    - IP address, port, `node ID`, etc
  - each entry corresponds to a specific distance from the node
    - for example, node in the Nth position in the `list`, must have a differing Nth bit from the `node ID`
    - so, the list holds a classification of 128 distances of other nodes in the network

- as nodes are encountered on the network, they are added to the `lists`
    - store and retrieval operations
    - helping other nodes to find a key
    - every node encountered will be considered for inclusion in the lists
    - keps network constantly updated
        - adding resilience to failures and attacks

- `k-buckets`
  - `k` is a system wide number
  - every `k-bucket` is a `list` having up to `k` entries inside
  - example:
    - network with `k=20`
    - each node will have `lists` containing up to 20 nodes for a particular bit
- possible nodes for each `k-bucket` decreases quickly
  - as there will be very few nodes that are that close
- since quantity of possible IDs is much larger than any node population, some of the `k-buckets` corresponding to very short distances will remain empty

- example:



  - network size: 2^3
  - max nodes: 8, current nodes: 7
  - let's take 6th node (110) (black leaf)
  - 3 `k-buckets` for each node in the network (gray circles)
    - nodes 0, 1, 2 (000, 001, 010) are in the farthest `k-bucket`
    - node 3 (011) is not participating in the network
    - middle `k-bucket` contains the nodes 4 and 5 (100, 101)
    - last `k-bucket` can only contain node 7 (111)

- Each node knows its neighbourhood well and has contact with a few nodes far away which can help locate other nodes far away.
- Kademlia priorizes long connected nodes to remain stored in the `k-buckets`
    - as the nodes that have been connected for a long time in a network will probably remain connected for a long time in the future

- when a `k-bucket` is full and a new node is discovered for that `k-bucket`
  - node sends a ping to the last recently seen node in the `k-bucket`
  - if the node is still alive, the new node is stored in a secondary list (a replacement cache)
    - replacement cache is used if a node in the `k-bucket` stops responding
  - basically, new nodes are used only when older nodes disappear

Protocol messages

- PING
- STORE
- FIND_NODE
- FIND_VALUE
  Each `rpc` msg includes a random value from the initiator, to ensure that the response corresponds to the request

Locating nodes

- node lookups can proceed asynchronously
  - `α` denotes the quantity of simultaneous lookups
  - `α` tipically is 3
- node initiates a FIND_NODE request to the `α` nodes in its own `k-bucket` that are closest ones to the desired key

- when the 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) that receives
    - keeping the `k` best ones (the `k` nodes that are closer to the searched key)
- the requester node will select these `k` best results and issue the request to them
- the proces is repeated again and again until get the searched key

- iterations continue until no nodes are returned that are closer than the best previous results
    - when 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
- node information can be augmented with RTT (round trip times)
    - when the RTT is spended, another query can be initiated
    - always the query's number are <= `α` (quantity of simultaneous lookups)

Locating resources

- data (values) located by mapping it to a key
  - typically a hash is used for the map
- locating data 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

Data replicating & caching

- values are stored at several nodes (k of them)
- the node that stores a value
  - periodically explores the network to find the k nodes close to the key value
  - to replicate the value onto them
  - this compensates the disappeared nodes

- avoiding "hot spots"
    - for popular values (might have many requests)
    - near nodes outside the k closest ones, store the value
        - this new storing is called `cache`
        - caching nodes will drop the value after a certain time
            - depending on their distance from the key
    - in this way the value is stored farther away from the key
        - depending on the quantity of requests
    - allows popular searches to find a storer more quickly
    - alleviates possible "hot spots"
- not all implementations of Kademlia have these functionallities (replicating & caching)
    - in order to remove old information quickly from the system

Joining the network

- to join the net, a node must first go through a `bootstrap` process
- `bootstrap` process
  - needs to know the IP address & port of another node (bootstrap node)
  - compute random unique `node ID` number
  - inserts the bootstrap node into one of its k-buckets

- `bootstrap` process [...]
  - perform a node lookup of its own `node ID` against the bootstrap node
    - this populate other nodes `k-buckets` with the new `node ID`
    - populate the joining node `k-buckets` with the nodes in the path between that node and the bootstrap node
  - refresh all `k-buckets` further away than the `k-bucket` the bootstrap node falls in
    - this refresh is a lookup of a random key that is within that `k-bucket` range
  - initially nodes have one `k-bucket`
    - when is full, it can be split