

CSN Final Project - Chord DHT with REST API

Ahmed Nouralla - a.shaaban@innopolis.university

Table Of Contents

1. Introduction
2. Node Design
3. Client Design
4. Lookup Mechanism
5. Results
6. Discussion
7. References

Introduction

- A Distributed Hash Table (DHT) is a regular dictionary (e.g., Java `HashMap` or Python `dict`) in which entries (i.e., keys and their corresponding values) are distributed over multiple nodes (peers).
- Chord is a DHT implementation that is popular for its scalability and performance.
 - It operates over a structured P2P overlay network in which nodes (peers) are organized in a ring
 - Each node should stay up-to-date about the current topology of the ring.
 - Nodes communicate over the network using Remote Procedure Call (RPC).

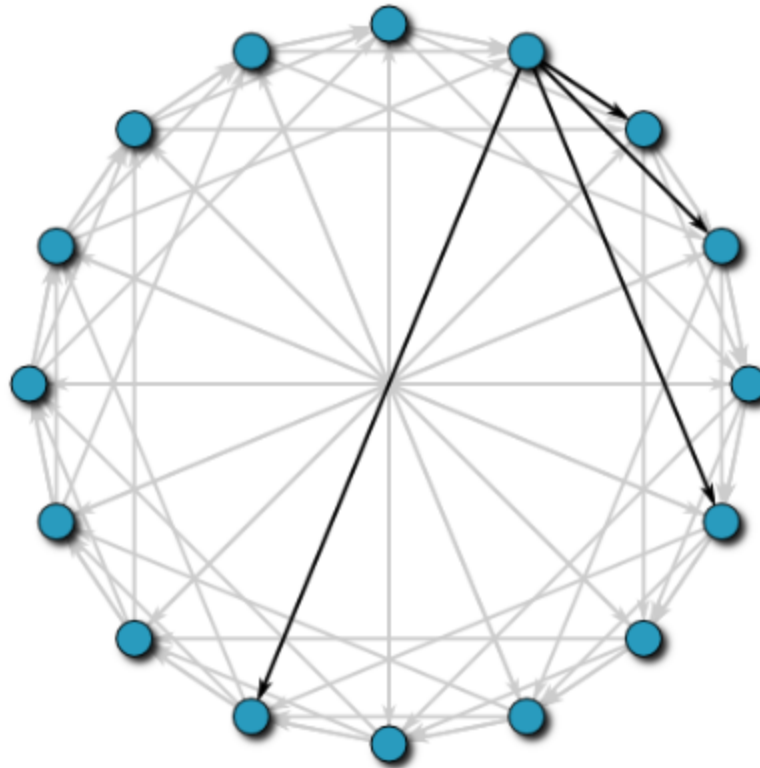
- For a given key, a node either:
 - Holds the value for that key (the key lies within its responsibility).
 - Or uses its finger table (*) to quickly reach the one that does.
- Each node has an integer identifier: $n \in [0, 2^M)$, where M is the key-length in bits
 - There can be up to $N \leq 2^M$ online nodes in the chord at a time.

- Each node maintains a finger table (i.e., a list of identifiers for some other nodes)
 - A finger table contains M entries at max.
 - The value of the i^{th} element in the finger table for node n can be defined as follows:

$$FT_n[i] = \text{successor}((n + 2^i) \bmod 2^M), i \in \{0..M-1\}$$

- Successor function returns the identifier of the next online node in the ring (clockwise direction).
- Finger tables are used by the Chord algorithm to achieve a logarithmic search time. They are the reason behind Chord scalability.

- Sample chord network with $M = 4$. The fingers for one of the nodes are highlighted.

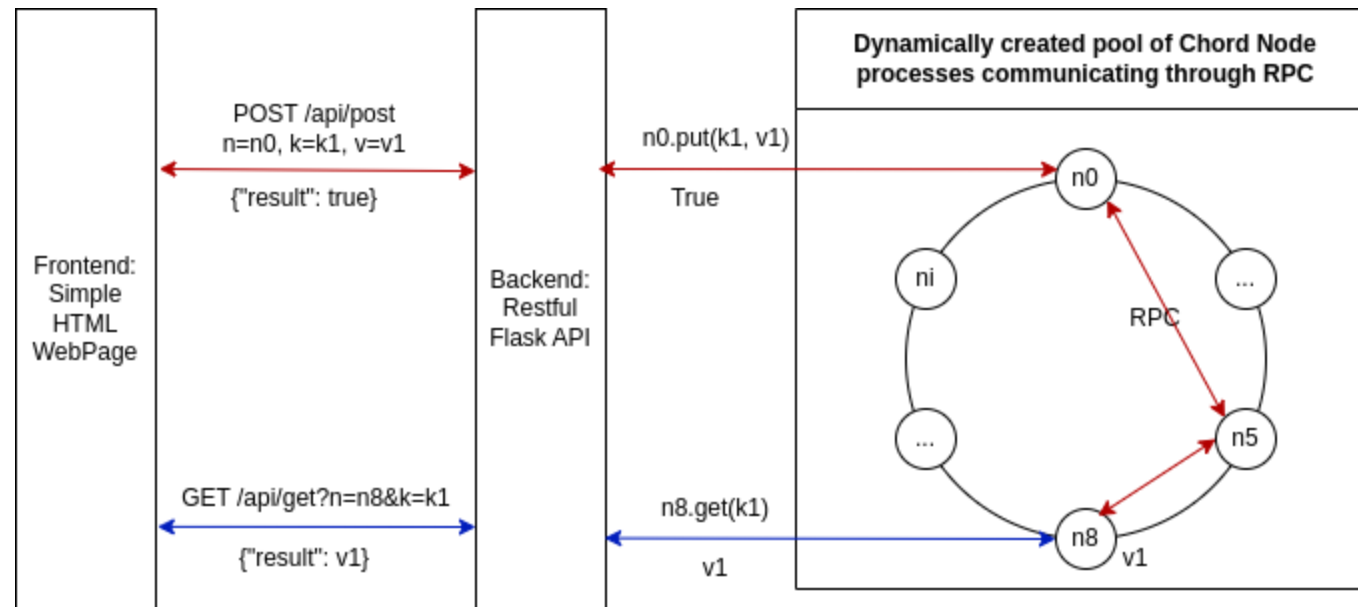


Methods

- This work implements a simplified version of the protocol in Python with a client for testing.
- The overlay network of peers is simulated using Python processes communicating over the local network. However, the same code should work over any IP network.

Architecture Diagram

- Red-colored arrows show a DHT insertion operation
- Blue-colored arrows show a subsequent DHT retrieval operation



Node Design

- Each node is responsible for storing values for keys in the range $\{\text{predecessor_id} + 1: \text{node_id}\}$ except the first node which stores values for keys in the range $\{\text{predecessor_id} + 1, 2^M - 1\}$ and $\{0, \text{node_id}\}$
 - A node's predecessor is the first online node lying before it (counter-clockwise) in the ring.

Client Design

- The designed client provides a RESTful HTTP API written using Flask framework.
- The client allows interacting with the chord network from a nice-looking HTML/CSS web interface.
- The client accepts two types of requests:
 - `GET /api/get?n=...&k=...` : asks node `n` for the value of key `k` .
 - The API returns a JSON response with `result` being `true` on success or a specific `error` on failure (e.g., because value is not in chord or node isn't online)
 - `POST /api/post` : asks `n` to place `k, v` pair in the chord (in the appropriate location).
- For the node to answer a request, it may contact other nodes as explained above.

Lookup Mechanism

Chord algorithm relies on two functions (`find_successor` and `closest_preceding_node`) to determine in which node should an entry (key-value pair) reside. The pseudo-code for these functions is given below:

```
# Recursive function returning the identifier of the node responsible for a given id
n.find_successor(id):
    if id == n.id:
        return id

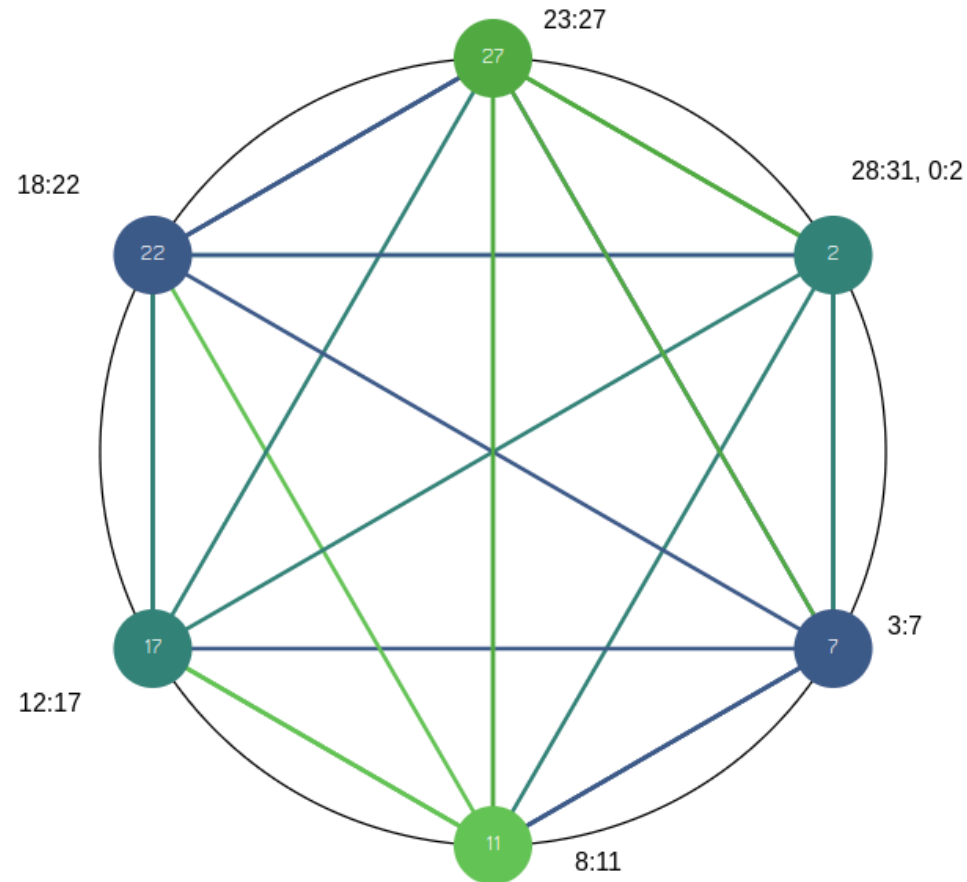
    # Range is evaluated in the circular sense, so L<=R is not necessarily true
    if id ∈ {n.id+1 ... n.successor_id} then
        return n.successor_id

    # Forward the query to the closest preceding node in the finger table for n
    n0 := closest_preceding_node(id)
    return n0.find_successor(id)
```

```
# Returns the closest preceeding node (from n.finger_table) for a given id
n.closest_preceding_node(id):
    # Range is evaluated in the circular sense, so L<=R is not necessarily true
    for i = m downto 1 do
        if finger[i].id ∈ {n.id+1, id-1} then
            return finger[i]
    return n
```

Results

- Example chord with $M=5$ -bit keys (supporting up to 32 nodes). Assume the shown ones (2,7,11,17,22,27) are currently online. Each node will be responsible for the shown range of keys.



- Running the example from the above sample shows

The screenshot displays a web browser window on the left and a terminal window on the right. The browser window shows a RESTful API Client interface for a Distributed Key-Value Store. The PUT Operation section has Node ID 2, Key 13, and Value hello, with a PUT button. Below it, the GET Operation section has Node ID 27 and Key 13, with a GET button. The terminal window shows the execution of a Python script (main.py) in a virtual environment. It displays the initialization of five nodes (2, 7, 11, 17, 22, 27) with their respective finger tables and listening ports. The script then performs a PUT operation (put(13, hello)) and a GET operation (get(13)), showing the request forwarding and retrieval process. Finally, the script is interrupted with a Ctrl-C signal, and all nodes are shown shutting down.

Distributed Key-Value Store

PUT Operation

Node ID: 2

Key: 13

Value: hello

PUT

`{"result":true}`

GET Operation

Node ID: 27

Key: 13

GET

`{"result":"hello"}`

```
(venv) ahmed@ahmed ~/D/M/C/final-project (main)> python3 main.py
Client started
M: 5
RING: 2 7 11 17 22 27
Node 2 initialized! Finger table = [7, 7, 7, 11, 22]. Listening on 127.0.0.1:53649
Node 7 initialized! Finger table = [11, 11, 11, 17, 27]. Listening on 127.0.0.1:44489
Node 11 initialized! Finger table = [17, 17, 17, 22, 27]. Listening on 127.0.0.1:50205
Node 17 initialized! Finger table = [22, 22, 22, 27, 2]. Listening on 127.0.0.1:52395
Node 22 initialized! Finger table = [27, 27, 27, 2, 7]. Listening on 127.0.0.1:53615
Node 27 initialized! Finger table = [2, 2, 2, 7, 11]. Listening on 127.0.0.1:33379
* Serving Flask app 'main'
* Debug mode: off
2: put(13, hello)
2: forwarding request (key=13) to node 11
2: forwarding request (key=13) to node 17
17: storing (13, hello)
27: get(13)
27: forwarding request (key=13) to node 11
27: forwarding request (key=13) to node 17
17: retrieving value for 13
^C
27: Shutting down...
17: Shutting down...
11: Shutting down...
2: Shutting down...
22: Shutting down...
7: Shutting down...
```

Discussion

- Multiple cases need to be considered for more realistic scenarios. For example:
 - The Chord and its finger tables should update dynamically as nodes enter and exit the system.
 - Periodical stabilization procedures are also needed to ensure that nodes stay up-to-date with the current topology of the ring.
 - Additional mechanisms are also in place to handle network partitioning and node failure.

References

- Original paper: https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
- Original implementation: <https://github.com/sit/dht>
- Wikipedia article: [https://en.wikipedia.org/wiki/Chord_\(peer-to-peer\)](https://en.wikipedia.org/wiki/Chord_(peer-to-peer))
- Visualizer tool: <https://msindwan.github.io/chordgen/>