CSN Project - Chord DHT with REST API

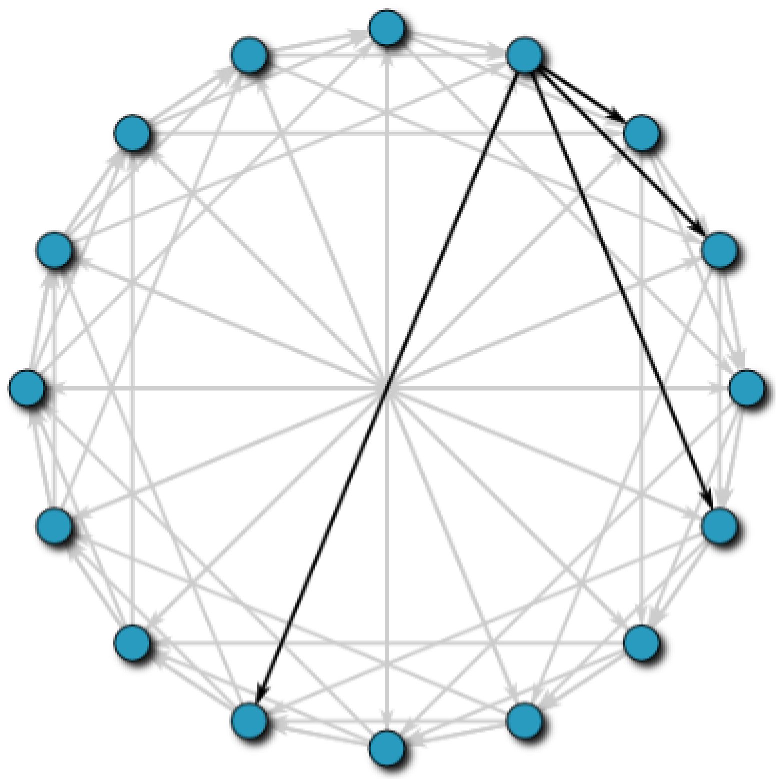
> Ahmed Nouralla - a.shaaban@innopolis.university

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:

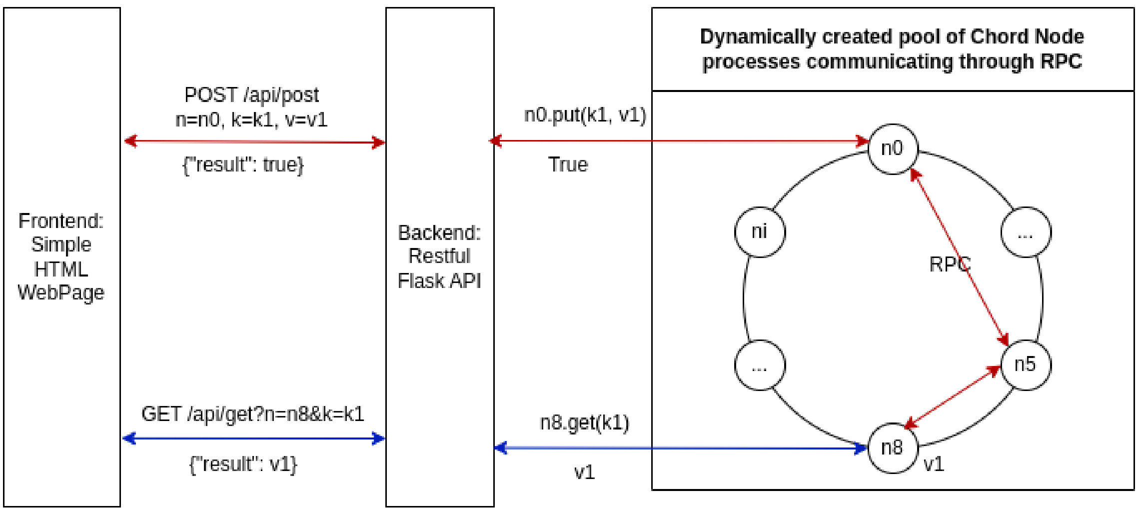
$$FT_n[i] = successor((n + 2^i) \mod 2^M), i \in \{0..M-1\}$$

  - 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.
- The overall architecture with sample usecase is illustrated by this diagram.
  - Red-colored arrows show DHT insertion operation asking node `n0` to place the pair `k1, v1` in the chord. Request content is depicted above the arrow and response content below it. The value ended up being stored at node `n8`.
  - Blue-colored arrows show a subsequent DHT retrieval operation, which happened to ask `n8` - it can ask any node in the chord - for the value associated with `k1`, `n8` immediately returned the value `v1`.



Node Design

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

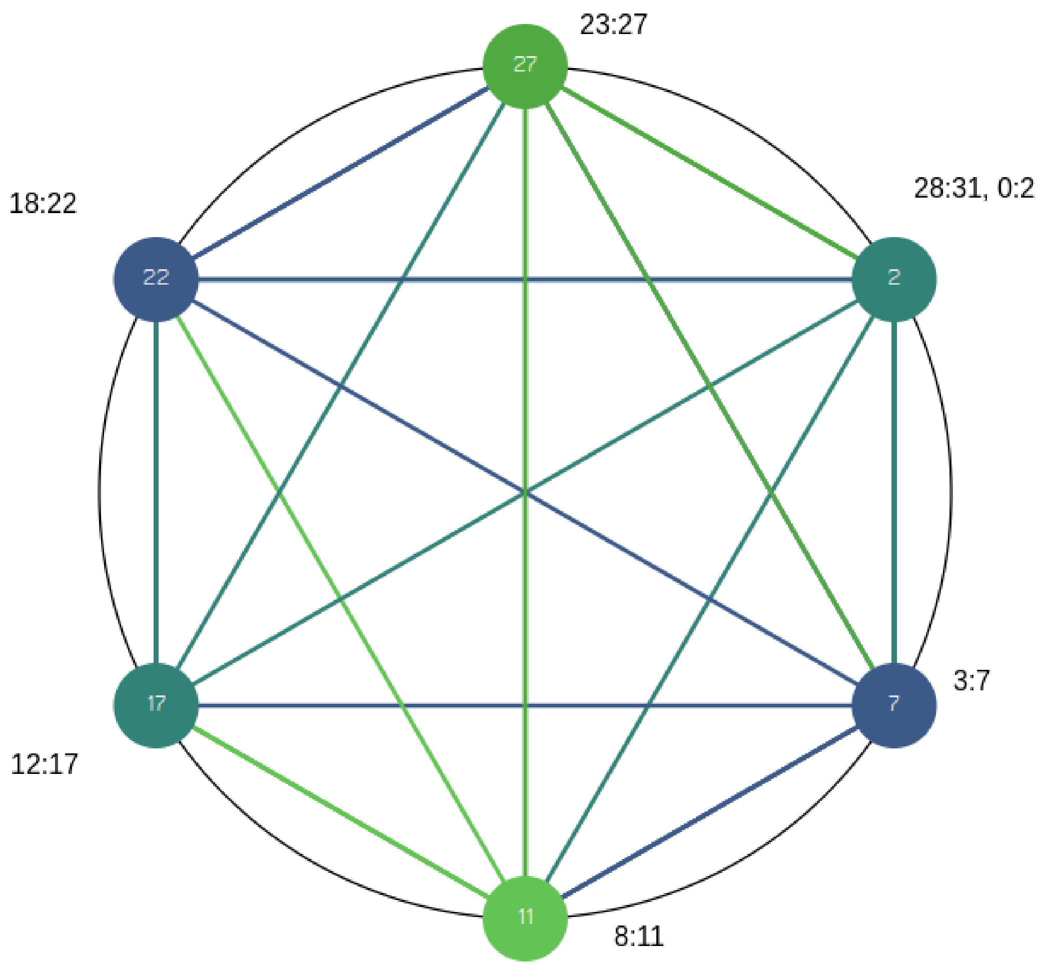
Chord algorithm relies on two functions (`find_successor` and `closest_preceding_node`) to determine in which node should an entry (key-value pair) reside. The pseudo-code for these functions is given below:

```
# 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)
```

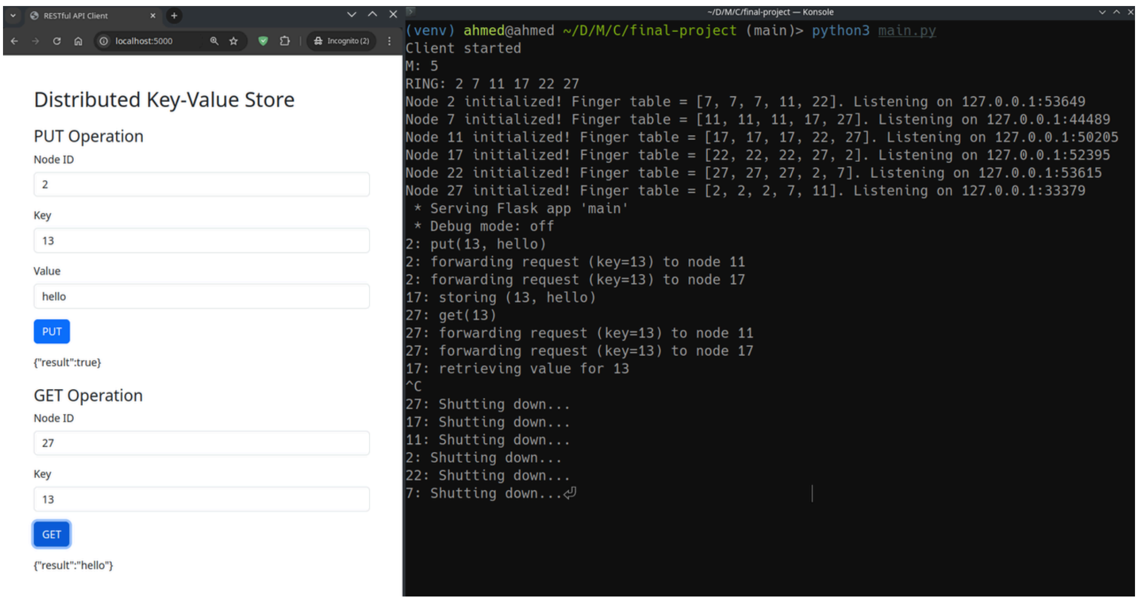
```
# Returns the closest preceeding node (from n.finger_table) for a given id
n.closest_preceding_node(id):
    # Range is evaluated in the circular sense, so L<=R is not necessarily true
    for i = m downto 1 do
        if finger[i].id ∈ {n.id+1, id-1} then
            return finger[i]
    return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

Results

- Let's build the shown 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



Discussion

- Chord is quite complex. The implementation done is simplified to adjust the complexity of the task.
- 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)
- Visualizer tool: https://msindwan.github.io/chordgen/