Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Presented By :
Mahima Dewangan (202411037)
Hari Pala (202411041)
Vrajkumar Patel (202411049)

Submitted To :
Prof. Amit Mankodi
TA : Utsav Pathak

Group ID : 5

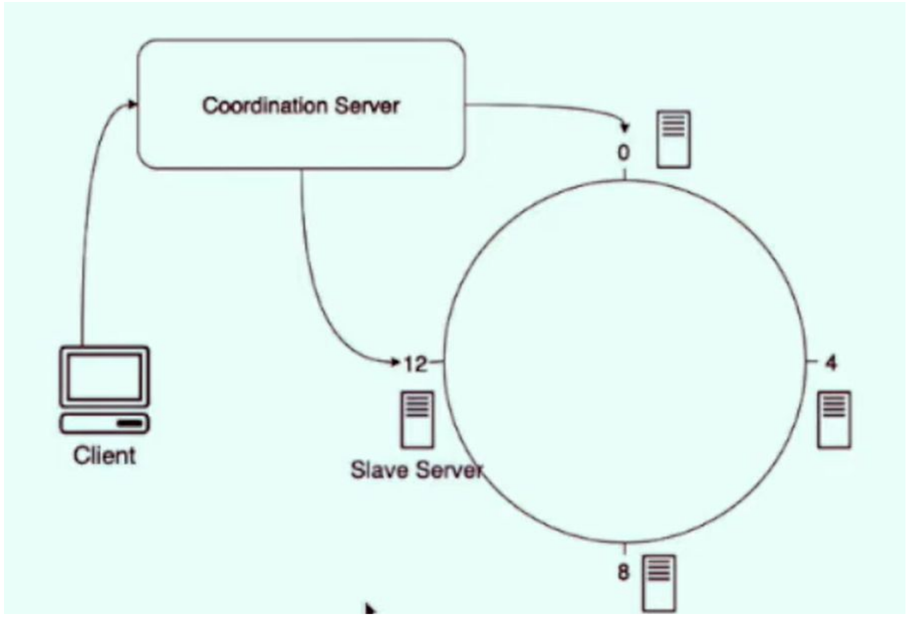
TABLE OF CONTENTS

PROBLEM STATEMENT

- In peer-to-peer distributed systems, a fundamental challenge is efficiently locating the node responsible for storing a particular data item. In a dynamic environment where nodes frequently join, leave, or fail, ensuring fast and reliable lookup operations becomes increasingly difficult. Traditional approaches that rely on centralized directories suffer from scalability limitations and single points of failure, making them unsuitable for large-scale, decentralized systems.

BEFORE CHORD

- Before Chord we had client - server architecture base DHT like Piazza.
- **Problem** : Coordination Server can be Bottleneck.



CHORD

What is Chord?

- It is a peer to peer hash table that uses consistent hashing on peer’s address.
- A hash table is data structure where you can do insert, delete and lookup in O(1)
- DHT is a similar data structure that runs in a distributed setting.
- Supports just one operation : given a key, it maps the key onto a node.

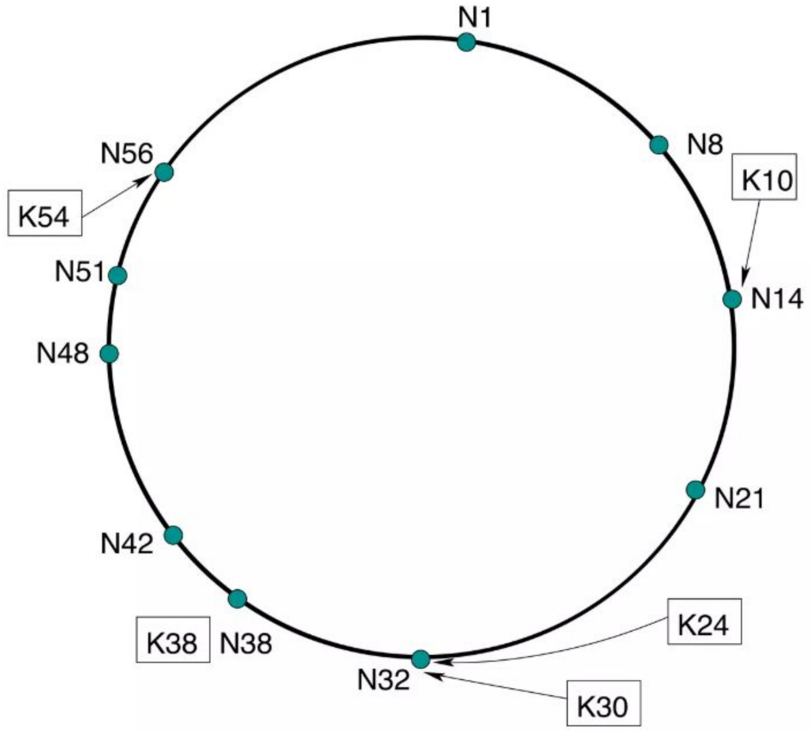
CHORD PROPERTIES

- Load Balancing :
  - Spreads keys evenly over nodes.
- Decentralized :
  - Fully distributed, all nodes equally important
  - Does not take into account heterogeneity of nodes .
- Scalability :
  - Lookup cost : O(log(N))
- Availability :
  - Automatically updates lookup tables as nodes join, leave , fail

CHORD PROTOCOL

- Chord uses consistent hashing to hash the peers to a particular id on the logical ring of size 2^m.
- Assign each node and key an m-bit identifier using SHA-1.
    - Node : hash the IP-Address
    - Key : hash the key.
- Identifiers are ordered on an identifier circle modulo 2^m, the chord ring:
    - Circle of numbers from 0 to (2^m) - 1.

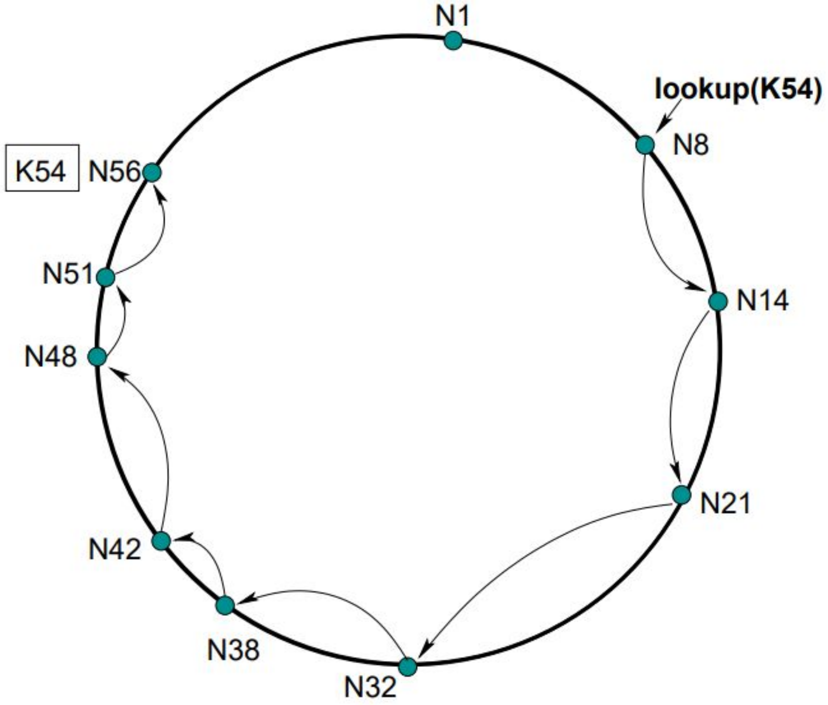
Chord ring (m=6)



Simple Chord lookup

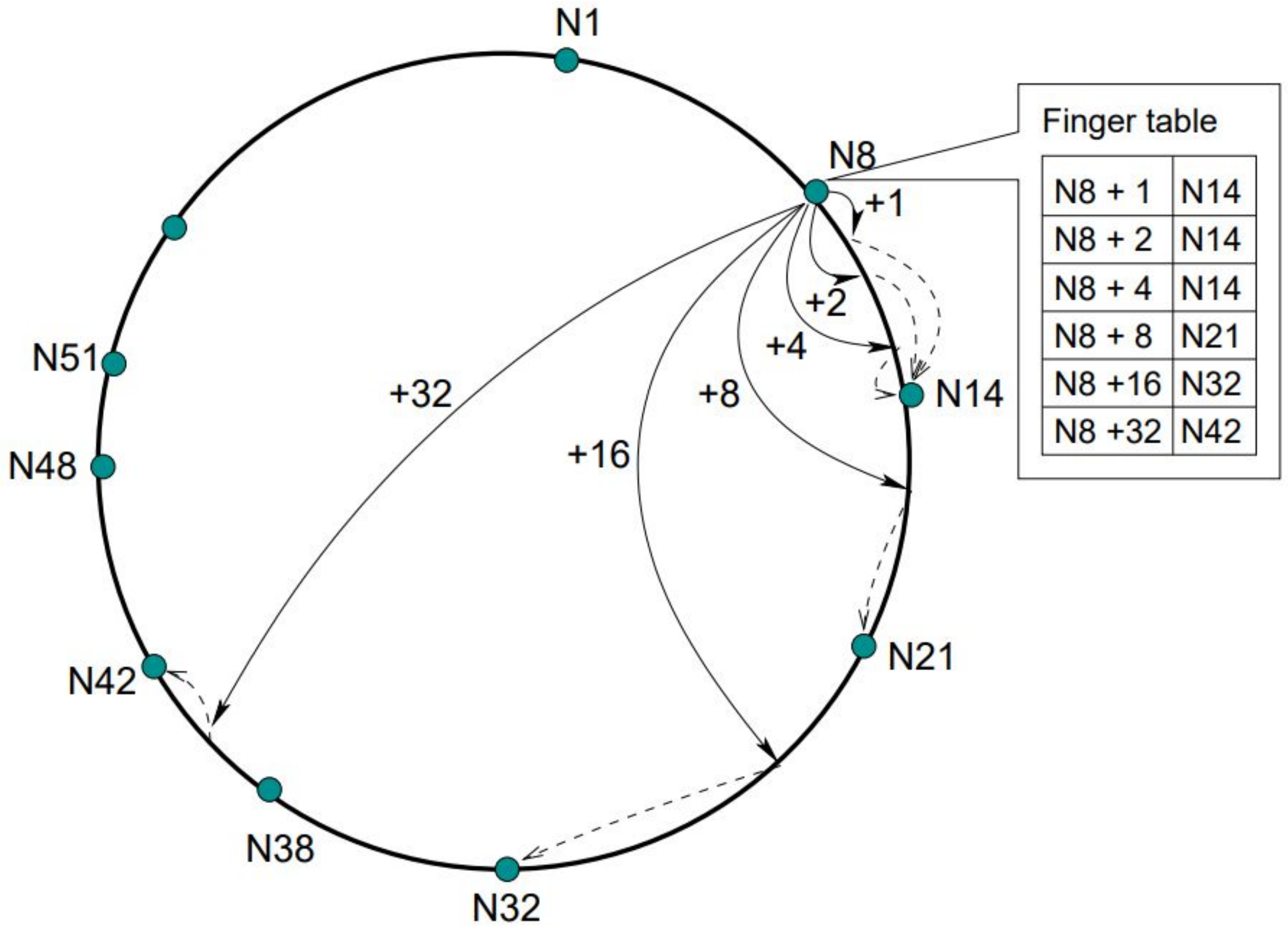
- Simple Chord lookup algorithm:
  - Each node only needs to know its successor on Chord ring
  - Query for identifier id is passed around Chord ring until successor(id) is found
  - Result returned along reverse path
- Easy but not efficient : # messages O(N)

```
// ask node $n$ to find the successor of $id$
$n$.find_successor($id$)
    if ($id \in (n, successor]$)
        return $successor$;
    else
        // forward the query around the circle
        return $successor$.find_successor($id$);
```



Finger table

- Chord maintains additional routing info:
  - Not essential correctness
  - But allows acceleration of lookups
- Each node maintains finger table with m entries:
- N.finger[i] = successor(n + 2^i-1)



Finger table

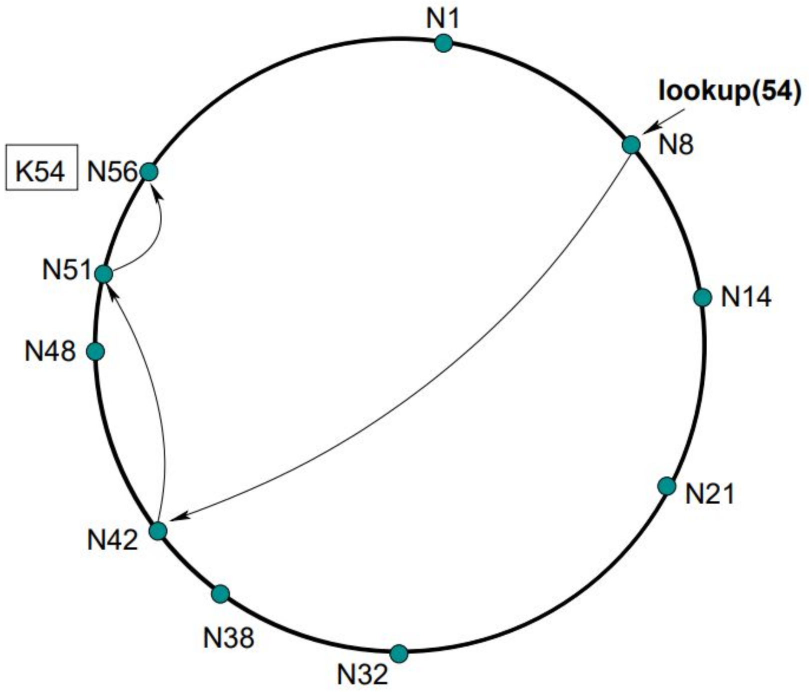
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Improved lookup

- Lookup for id now works as follows:
  - If id fails between n and successor(n), successor(n) is returned
  - Otherwise, lookup is performed at node n’ , which is the node in the finger table of n that most immediately precedes id
- Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier.
- Thus O(log(N)) nodes need to be contacted.

```
// ask node $n$ to find the successor of $id$
$n$.**find_successor**($id$)
  **if** ($id \in (n, successor]$)
    **return** $successor$;
  **else**
    $n' = closest\_preceding\_node(id)$;
    **return** $n'.find\_successor(id)$;

// search the local table for the highest predecessor of $id$
$n$.**closest_preceding_node**($id$)
  **for** $i = m$ **downto** $1$
    **if** ($finger[i] \in (n, id)$)
      **return** $finger[i]$;
  **return** $n$;
```



THANK YOU