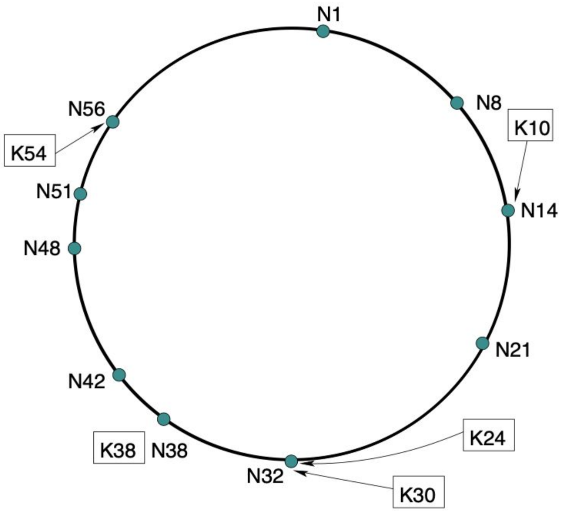
Chord protocol

Michele Papale - Gabriele Puce - Andrea Tassi

Distributed Systems 2018/2019

Introduction to the protocol

Introduction



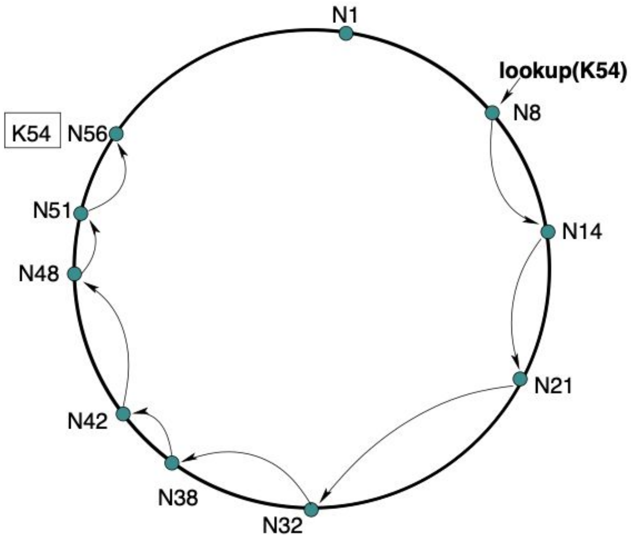
- **Nodes** *and* **keys** *are organized in a logical ring*
- *Each node is assigned a unique m-bit identifier*
  - *Usually the hash of the IP address*
- *Every item is assigned a unique m-bit key*
  - *Usually the hash of the item name*
- *The item with key k is managed by the node with the smallest ID greater than his id (the successor)*

Simple Key Location

Simple Lookup

- Each node need to know only about his successor

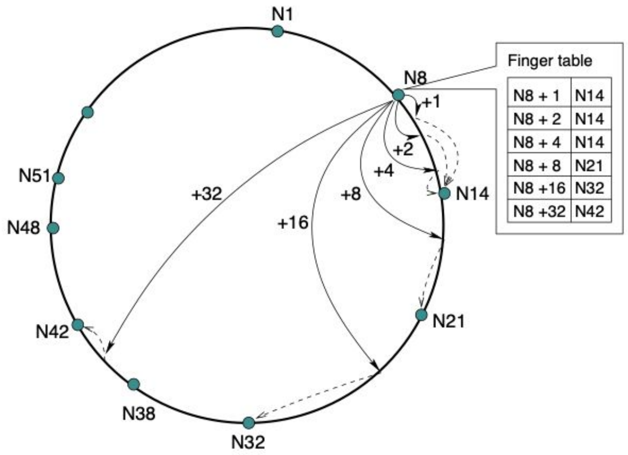
```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    return successor.find_successor(id);
```

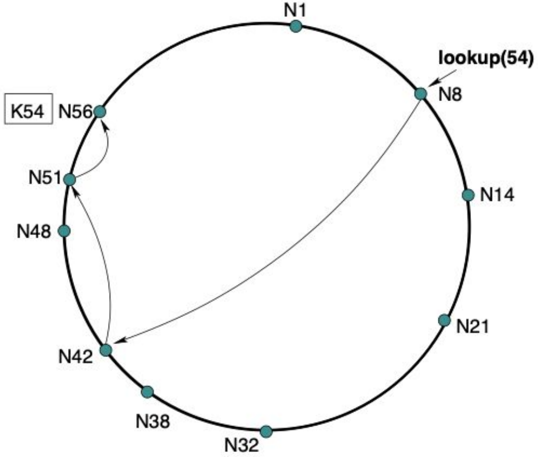


Scalable Key Location

Scalable Key Location

- Each node has a “Finger Table” with m entries (m = log N)
- Entry i in the finger table of node n is the first node whose id is higher or equal than n + 2i (i = 0...m-1)
- Lookup begins from the bottom

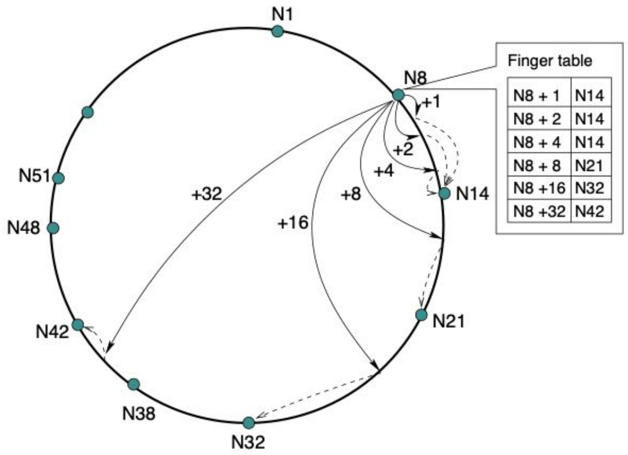


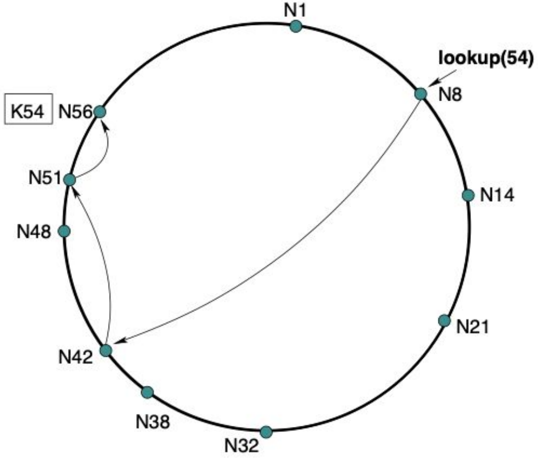


Scalable Key Location

// 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;





Stabilization protocol

Stabilization

// *called periodically. verifi es n's immediate*
// *successor, and tells the successor about n.*
n.**stabilize**()
  $x = successor.predecessor$;
  if $(x \in (n, successor))$
    $successor = x$;
  $successor.notify(n)$;

// n' *thinks it might be our predecessor.*
n.**notify**(n')
  if (*predecessor* **is nil or** $n' \in (predecessor, n)$)
    $predecessor = n'$;

// *called periodically. refreshes fi nger table entries.*
// *next stores the index of the next fi nger to fi x.*
n.**fix_fingers**()
  $next = next + 1$;
  if $(next > m)$
    $next = 1$;
  $finger[next] = find\ successor(n + 2^{next-1})$;

```
def fixLists():
    successor = this.getSuccessor()
    foundLivingSuccessor = False
    while not foundLivingSuccessor:
        if successor is alive:
            successorList = successor.getSuccessorList()
                this.setSuccessorList(successorList)
                successorItems = successor.getSuccessorItems()
                this.setSuccessorItems(successorItems)
                this.setSuccessor(successor)
                foundLivingSuccessor = True
        else:
                //get next successor from successorList
```

```
def fixItems():
    foreach item in successorItems:
        if findSuccessor(item) != thisNode:
            newOwner = findSuccessor(item)
            newOwner.setItem(item)
            thisNode.deleteItem(item)
```

The implementation

Our implementation

- Java library (Simple and Scalable Key Location)
    - Simple Key Location
    - Scalable Key Location
    - Stabilize protocols
        - to handle joins and leaves(voluntary and involutary)
    - implemented SuccessorList and SuccessorItems
- application to show its use (using RMI)
- Tests

Node - main methods

- create(int numBitsIdentifier, boolean isSimpleLookupAlgorithm)
- join(Node knownNode)
- lookUp(int key)
- storeItem(Item item)
- exitFromRing()

Item - main methods

- Item(String name, int module)

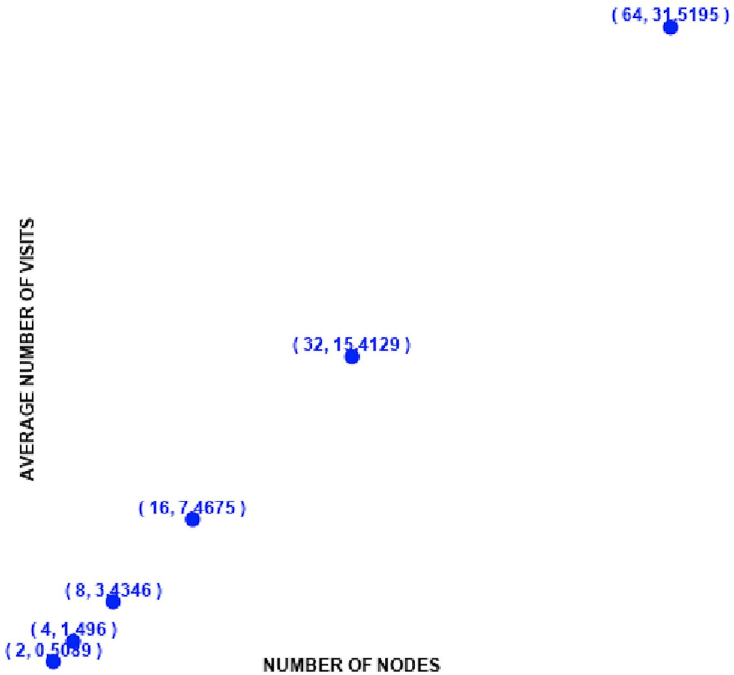
Tests

Complexity analysis

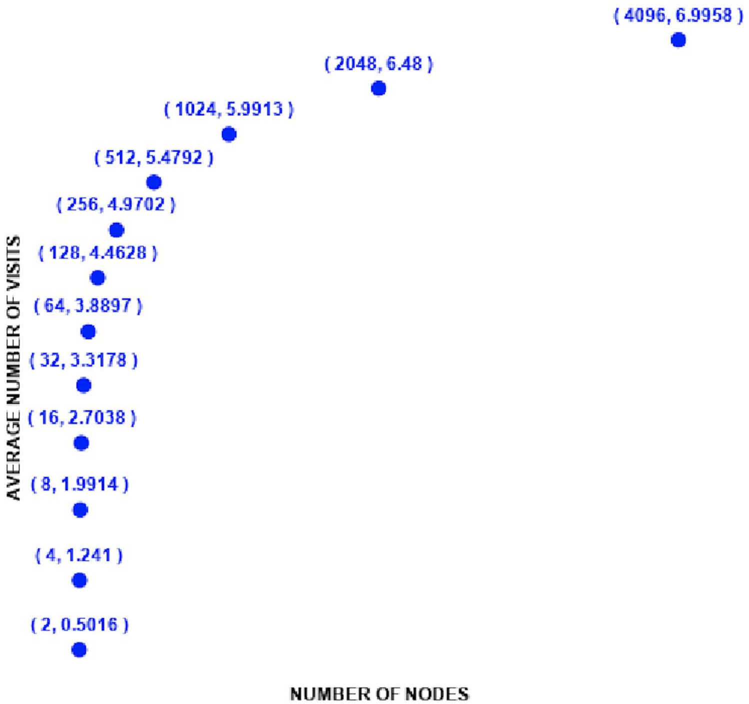
Temporal complexity

- To ensure the correctness of out implementation, we tested the temporal complexity of lookup
  - Simple Key Location - O(N)
  - Scalable Key Location - O(log N)
- We have run 1000 different lookups for 10 ring of different size (starting from a ring containing 2 nodes to a ring containing 1024 nodes)
- Based on incrementing a counter

Simple case



Scalable case



Tests

- **Load Tests:** it creates a ring doing random actions (between join/storeItems/exit) for 20 seconds. After this time, it is printed the info of a node you want to check the correctness.
- **jUnit tests:** we have tested the most important operations done by the Chord protocol (lookUp - findSuccessor - storeItem - create - join - exitFromRing)