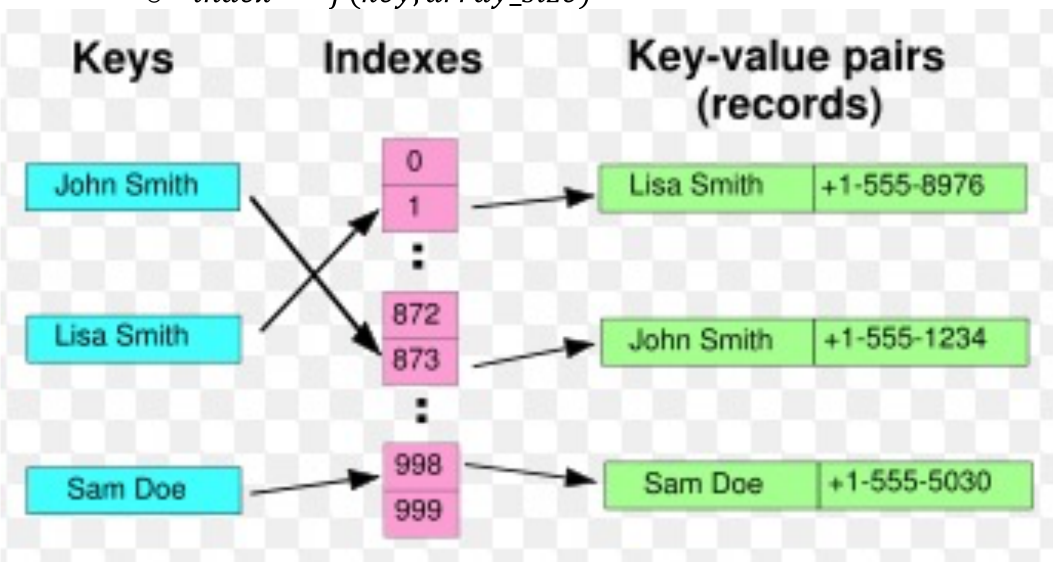
Distributed Hash Table

Outline

- Hashing.
- Distributed Hash Table.
- Chord.

A Hash Table (hash map).

- A data structure implements as an associative array that can map keys to values.
    - Searching and insertions are O(1) in the worst case.
- Uses a hash function to compute an index into an array of buckets or slots from which the correct value can be found.
    - $index\ =\ f(key, array\_size)$



Hash functions

- Crucial for good hash table performance.
- Can be difficult to achieve.
    - WANTED: uniform distribution of hash values.
    - A non-uniform distribution increases the number of collisions and the cost of resolving them.

Hashing for partitioning use-case

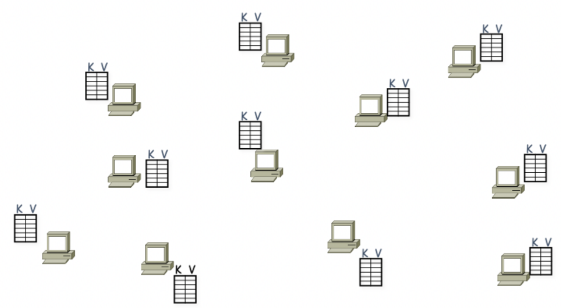
- Objective:
    - Given document X, choose one of k servers to use.
- E.g., using modulo hashing.
    - Number servers 1, 2, …, k.
    - Place X on server i = (X mod k).

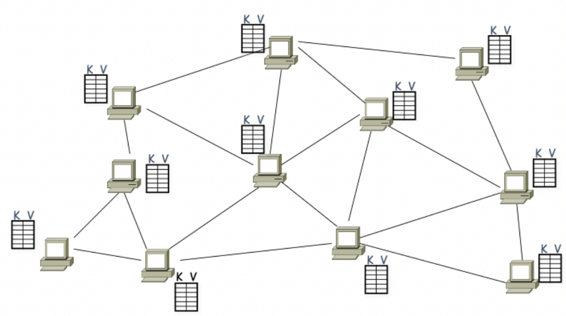
        - Problem? Data may not be uniformly distributed.
    - Place X on server i = hash(X) mod k.
- Problem?
    - What happens if a server fails or joins (k → k±1)?
    - What If different clients have different estimate of k?
    - Answer: All entries get remapped to new nodes!

Distributed hash table (DHT)

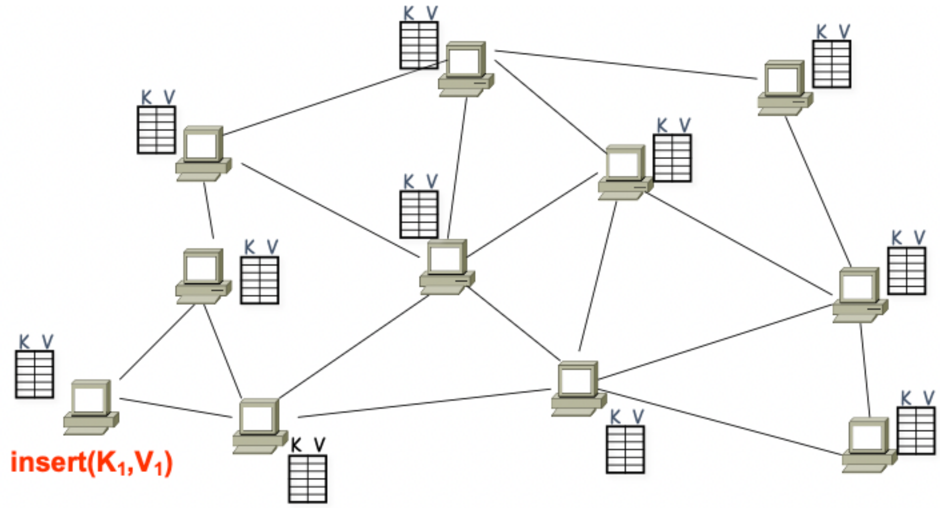
- Distributed Hash Table (DHT) is similar to hash table but spread across many hosts.
- Interface:
    - insert(key, value)
    - lookup(key, value)
- Every DHT node support a single operation:
    - Given key as input; route messages to node holding key.

DHT: basic idea

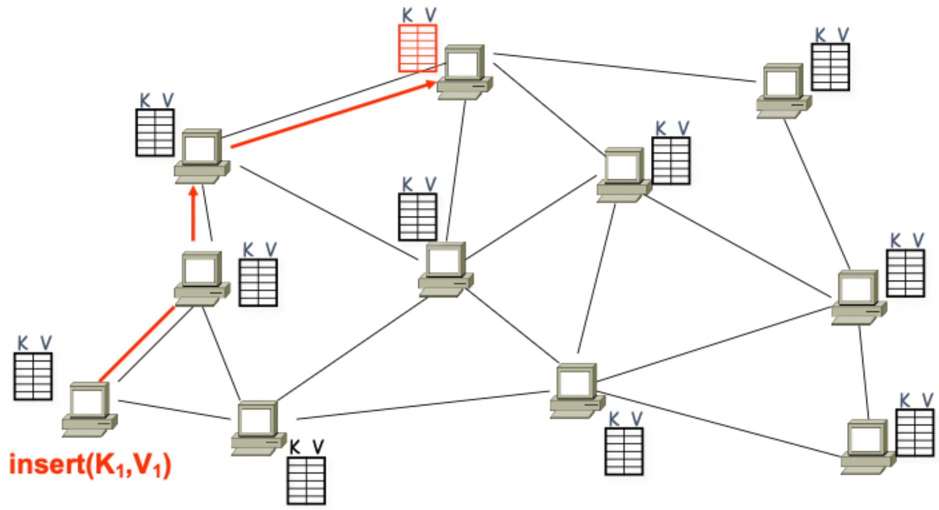




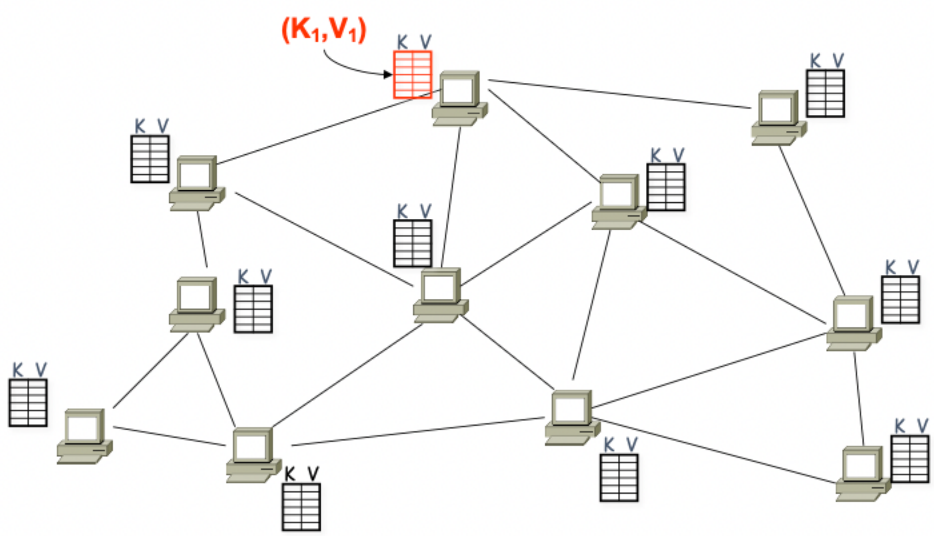
Neighboring nodes are “connected” at the application-level.
Operation: take key as input; route messages to node holding key.



insert(K_1,V_1)



insert(K_1,V_1)



(K_1,V_1)

How to design a DHT?

- State Assignment:
  - What (key, value) tables does a node store?
- Network Topology:
  - How does a node select its neighbors?
- Routing Algorithm:
  - Which neighbor to pick while routing to a destination?
- Various DHT algorithms make different choices:
  - CAN, Chord, Pastry, Tapestry, Plaxton, Viceroy, Kademlia, Skipnet, Symphony, Koorde, Apocrypha, Land, ORDI, etc.

Chord: A scalable peer-to-peer look-up protocol for internet applications

Outline

- What is Chord?
- Consistent Hashing.
- A simple key lookup algorithm.
- Scalable key lookup algorithm.
- Node Joins and Stabilization.
- Node Fails.

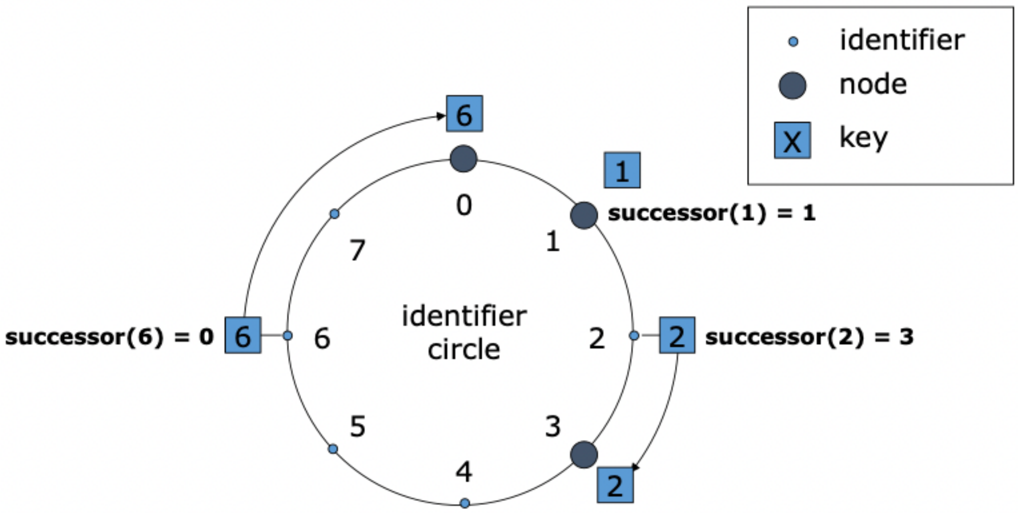
What is Chord?

- In short: a peer-to-peer lookup system.
- Given a key (data item), it maps the key onto a node (peer).
- Uses consistent hashing to assign keys to nodes.
- Solves the problem of locating key in a collection of distributed nodes.
- Maintains routing information with frequent node arrivals and departures.

Consistent hashing

- Consistent hash function assigns each node and key an m-bit identifier.
- SHA-1 is used as a base hash function.
- A node's identifier is defined by hashing the node's IP address.
- A key identifier is produced by hashing the key (chord doesn't define this. Depends on the application).
  - ID(node) = hash(IP, Port).
  - ID(key) = hash(key).

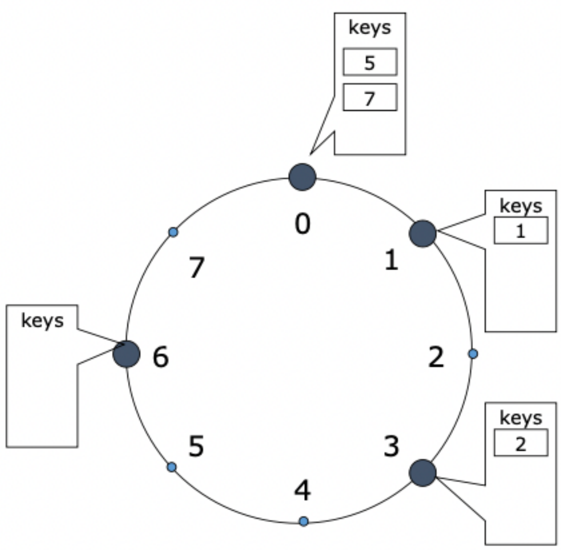
Consistent hashing – Successor nodes



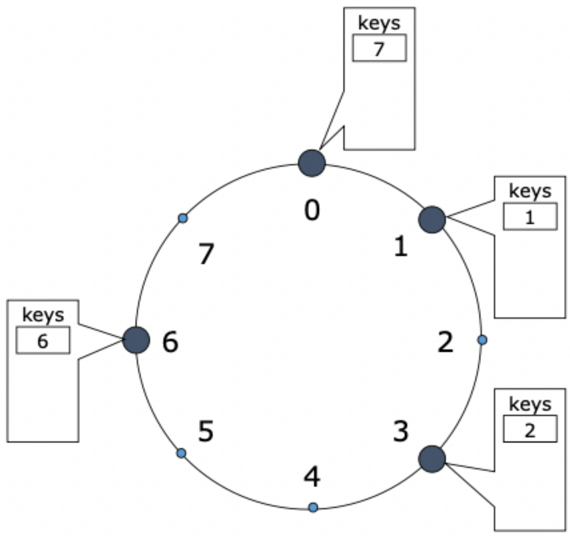
Consistent hashing – Join and Departure

- When a node n joins the network, certain keys previously assigned to n's successor now become assigned to n.
- When node n leaves the network, all its assigned keys are reassigned to n's successor.

Consistent hashing – Node join



Consistent hashing – Node departure

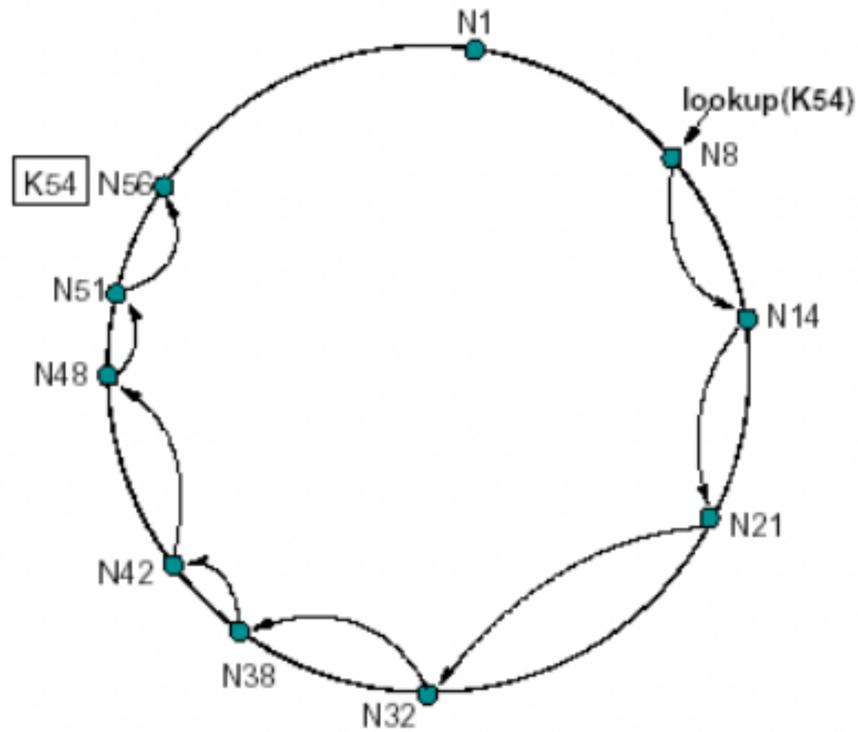


A simple key lookup

- If each node knows only how to contact its current successor node on the identifier circle, all nodes can be visited in linear order.
- Queries for a given identifier could be passed around the circle via these successor pointers until they encounter the node that contains the key.
- Pseudo code for finding 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);
```

- The path taken by a query from node 8 for key 54:

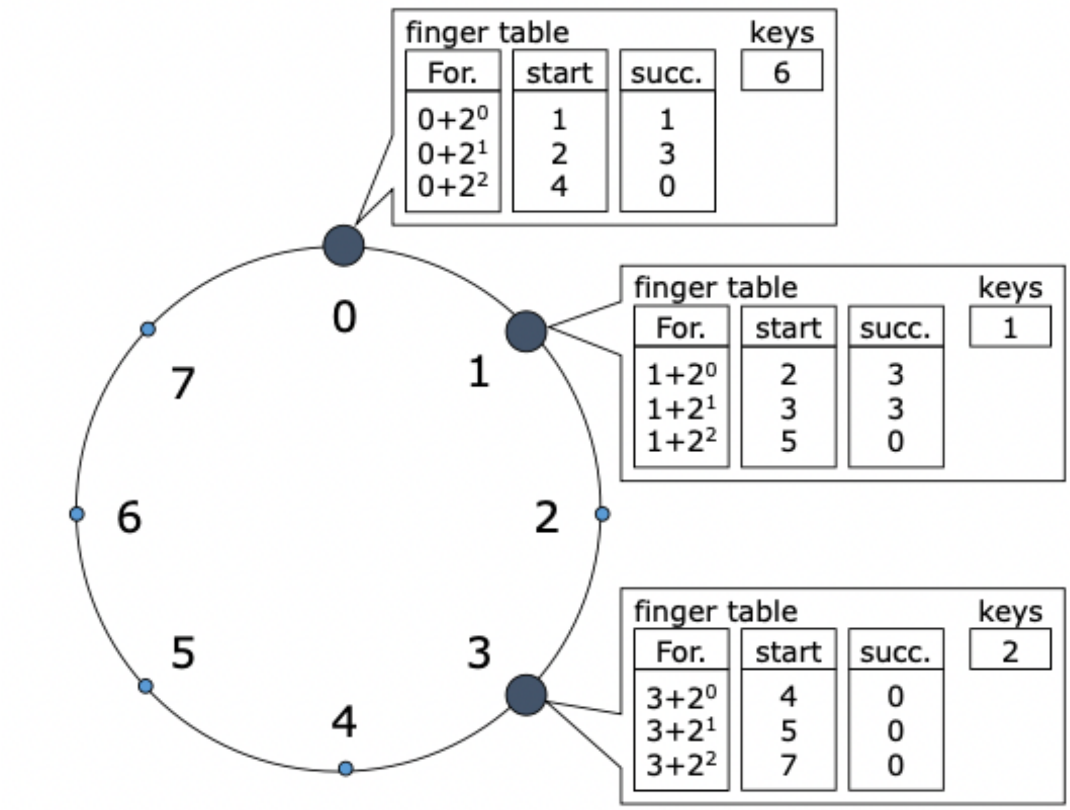


Scalable key location

- To accelerate lookups, Chord maintains additional routing information.
- This additional information is not essential for correctness, which is achieved as long as each node knows its correct successor.

Scalable key location – Finger tables

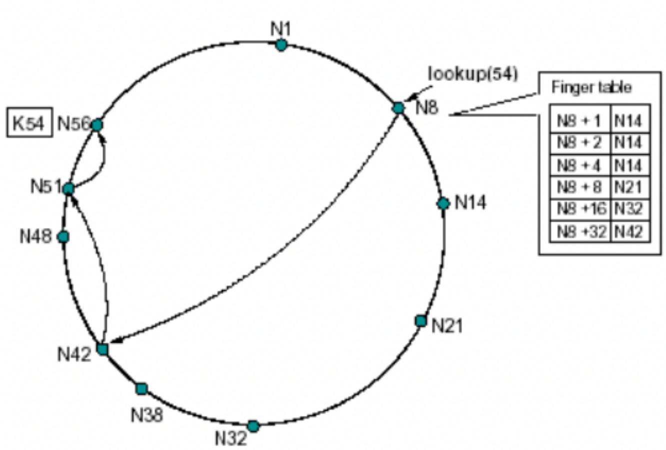
- Each node n’ maintains a routing table with up to m entries (which is in fact the number of bits in identifiers), called finger table.
- The i^{th} entry in the table at node n contains the identity of the first node s that succeeds n by at least $2^i – 1$ on the identifier circle.
- s = successor(n + 2^i -1).
- s is called the i^{th} finger of node n, denoted by n.finger(i).



- A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.
- The first finger of n is the immediate successor n on the circle.

Scalable key location – Example query

- The path a query for key 54 starting at node 8:



Applications: Chord-based DNS

- DNS provides a lookup service.
  - Keys: host names, values: IP addresses.
- Chord could hash each host name to a key.
- Chord-based DNS:
  - No special root servers.
  - No manual management of routing information.
  - No naming structure.
  - Can find objects not tied to particular machines.

What is Chord? Addressed problems

- **Load balance**: chord acts as a distributed hash function, spreading keys evenly over nodes
- **Decentralization**: chord is fully distributed, no node is more important than any other, improves robustness
- **Scalability**: logarithmic growth of lookup costs with the number of nodes in the network, even very large systems are feasible
- **Availability**: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found
- **Flexible naming**: chord places no constraints on the structure of the keys it looks up.

Summary

- Simple, powerful protocol
- Only operation: map a key to the responsible node
- Each node maintains information about O(log N) other nodes
- Lookups via O(log N) messages
- Scales well with number of nodes
- Continues to function correctly despite even major changes of the system

Table of Contents