Consistent hashing and distributed hash table

Viet-Trung Tran
trungtv@soict.hust.edu.vn

Outline

Hashing

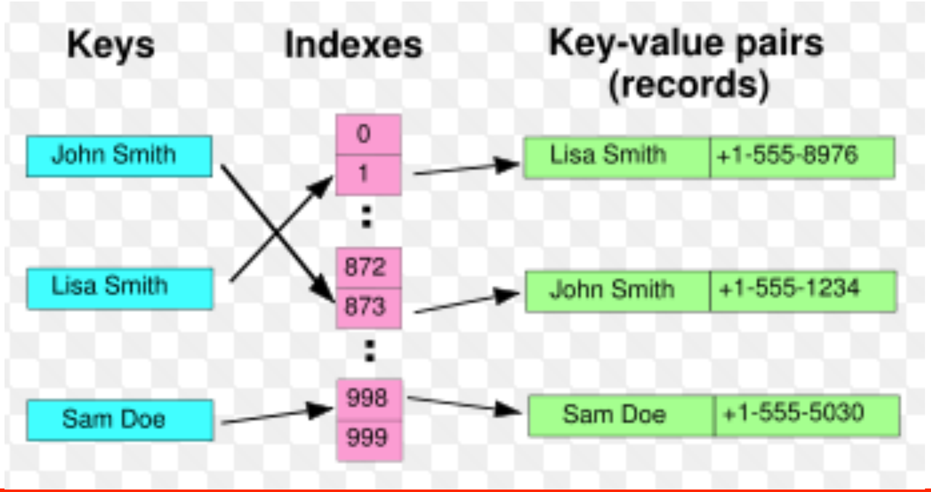
- A data structure for which both searching and insertions are 0(1) in the worse case
- Searching without performing a search ...
  - given an item x, we need to be able to determine directly from x the array position where it is to be stored

Hashing

- The idea of hashing is to distribute the entries of a dataset across an array of buckets.
- Given a key, the algorithm computes an index that suggests where an entry can be found:
  - index = f(key, array_size)
- Often this is done in two steps:
  - hash = hashfunc(key).
  - index = hash % array_size

A Hash Table (hash map)

- A data structure to implement an associative array.
- A structure that can map keys to values.
- Uses a hash function to compute an index into an array of buckets or slots from which the correct value can be found.



Hash function

- Crucial for good hash table performance.
- Can be difficult to achieve.
- A basic expectation is that the function would provide a uniform distribution of hash values.
- A non-uniform distribution increases the number of collisions and the cost of resolving them.

Basic Hashing for Partitioning?

- Consider problem of data partition:
  - Given document X, choose one of k servers to use

- Suppose we use modulo hashing
  - Number servers 1..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 is different clients has different estimate of k?

What is a DHT?

- Hash Table
  - data structure that maps “keys” to “values”
  - essential building block in software systems

- Distributed Hash Table (DHT)
  - similar, but spread across many hosts

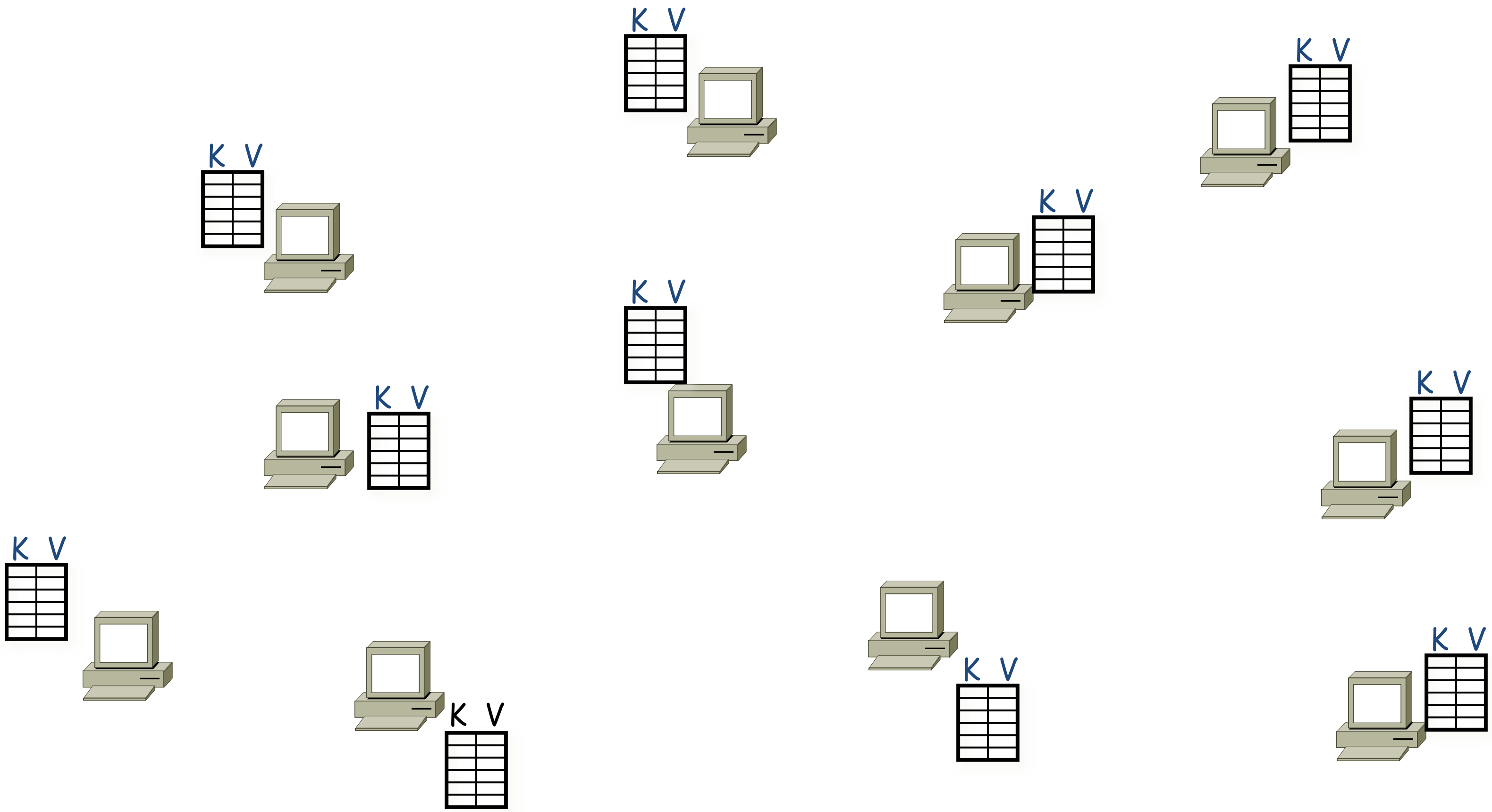
- Interface
  - insert(key, value)
  - lookup(key)

How do DHTs work?

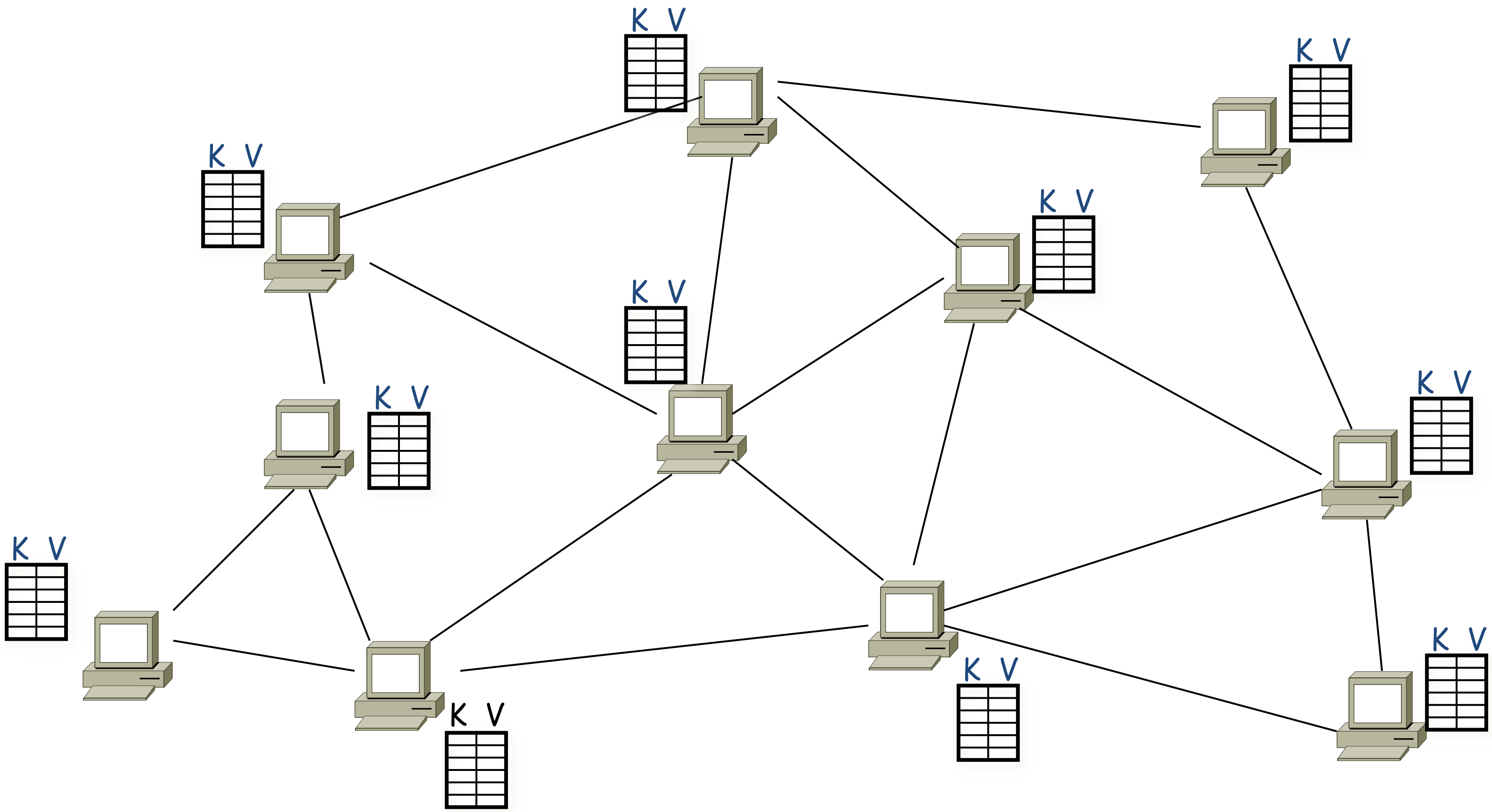
Every DHT node supports a single operation:

- Given *key* as input; route messages to node holding *key*
    - DHTs are *content-addressable*

DHT: basic idea

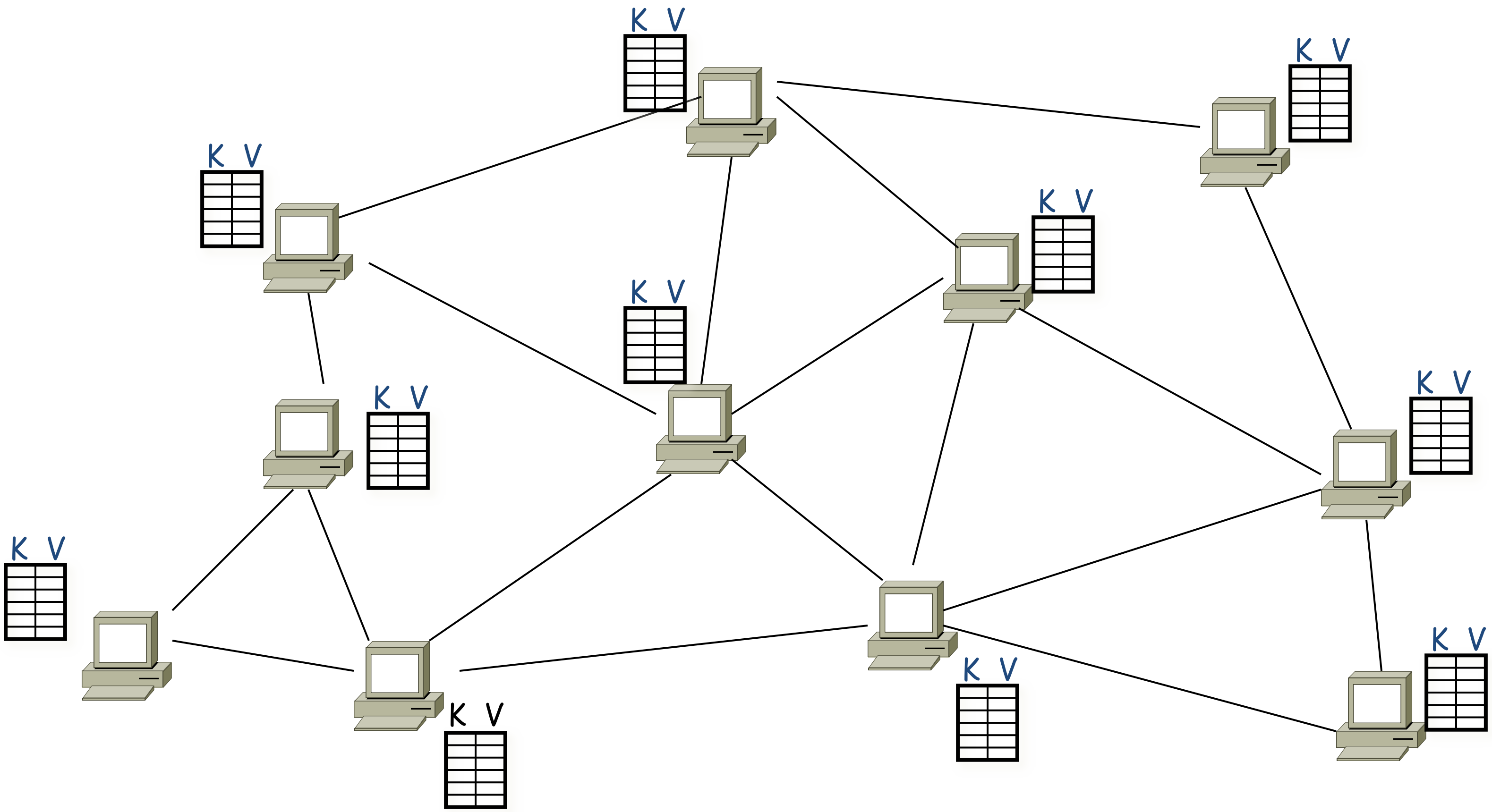


DHT: basic idea



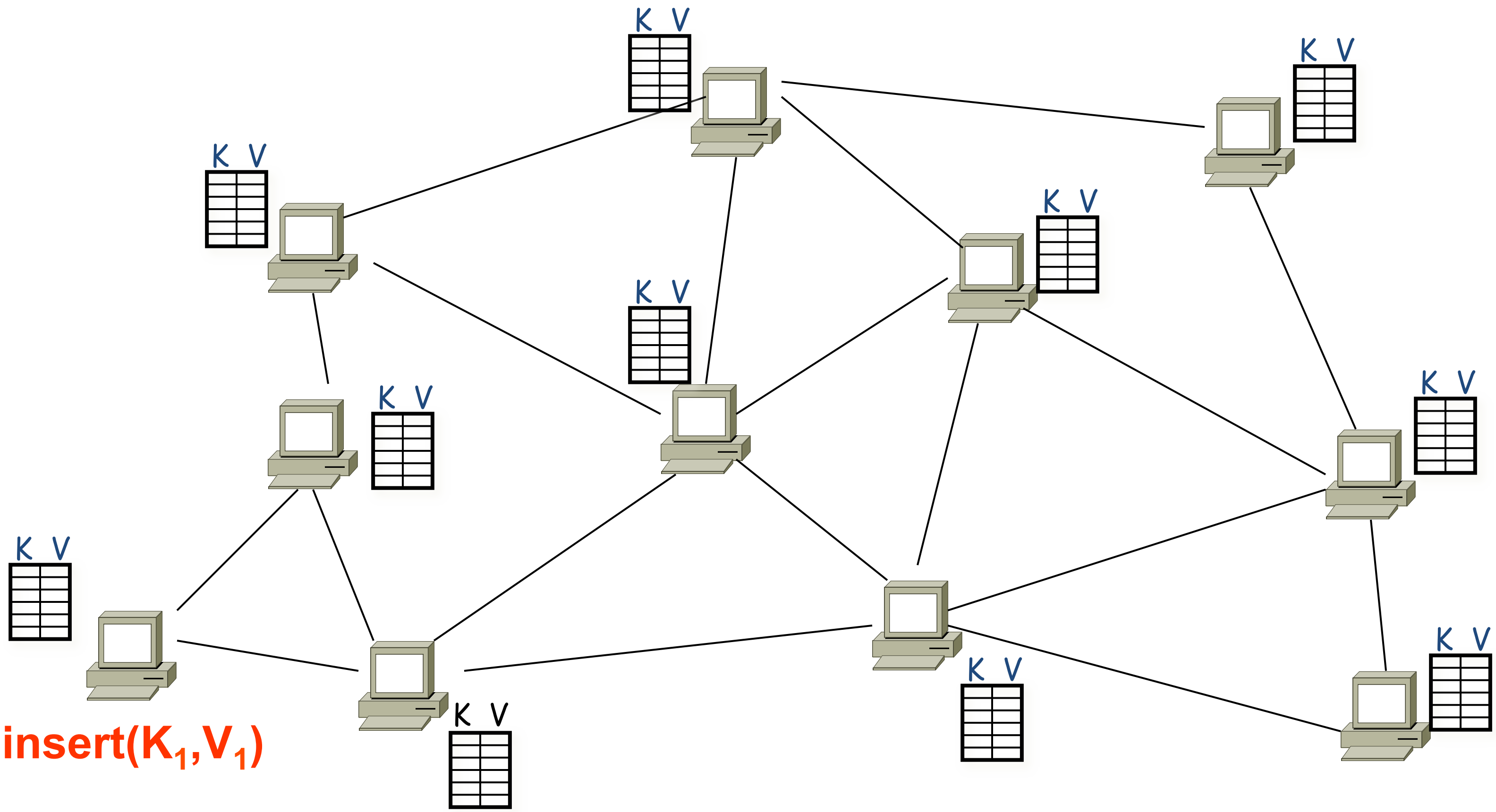
Neighboring nodes are "connected" at the application-level

DHT: basic idea



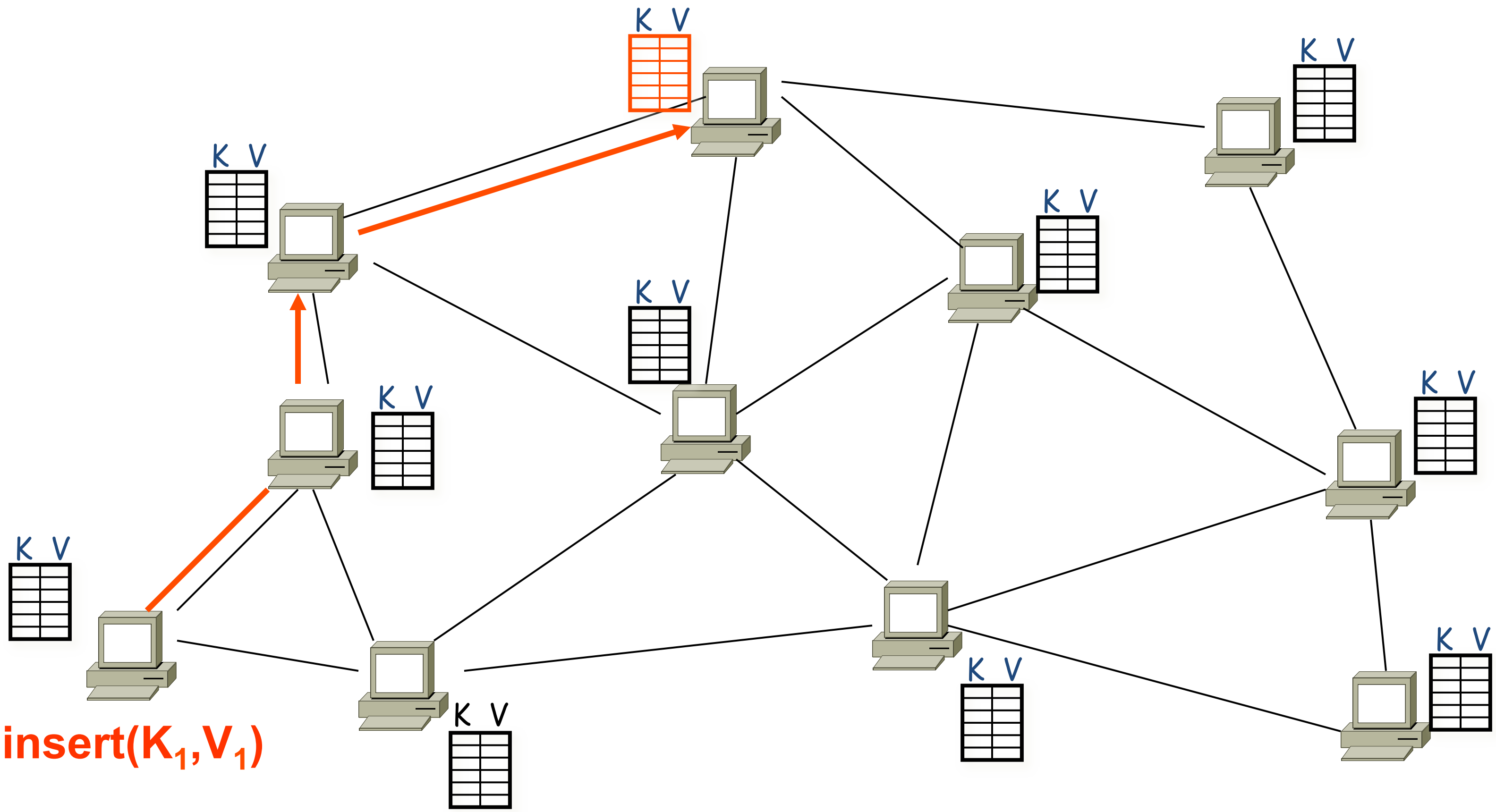
Operation: take *key* as input; route messages to node holding *key*

DHT: basic idea



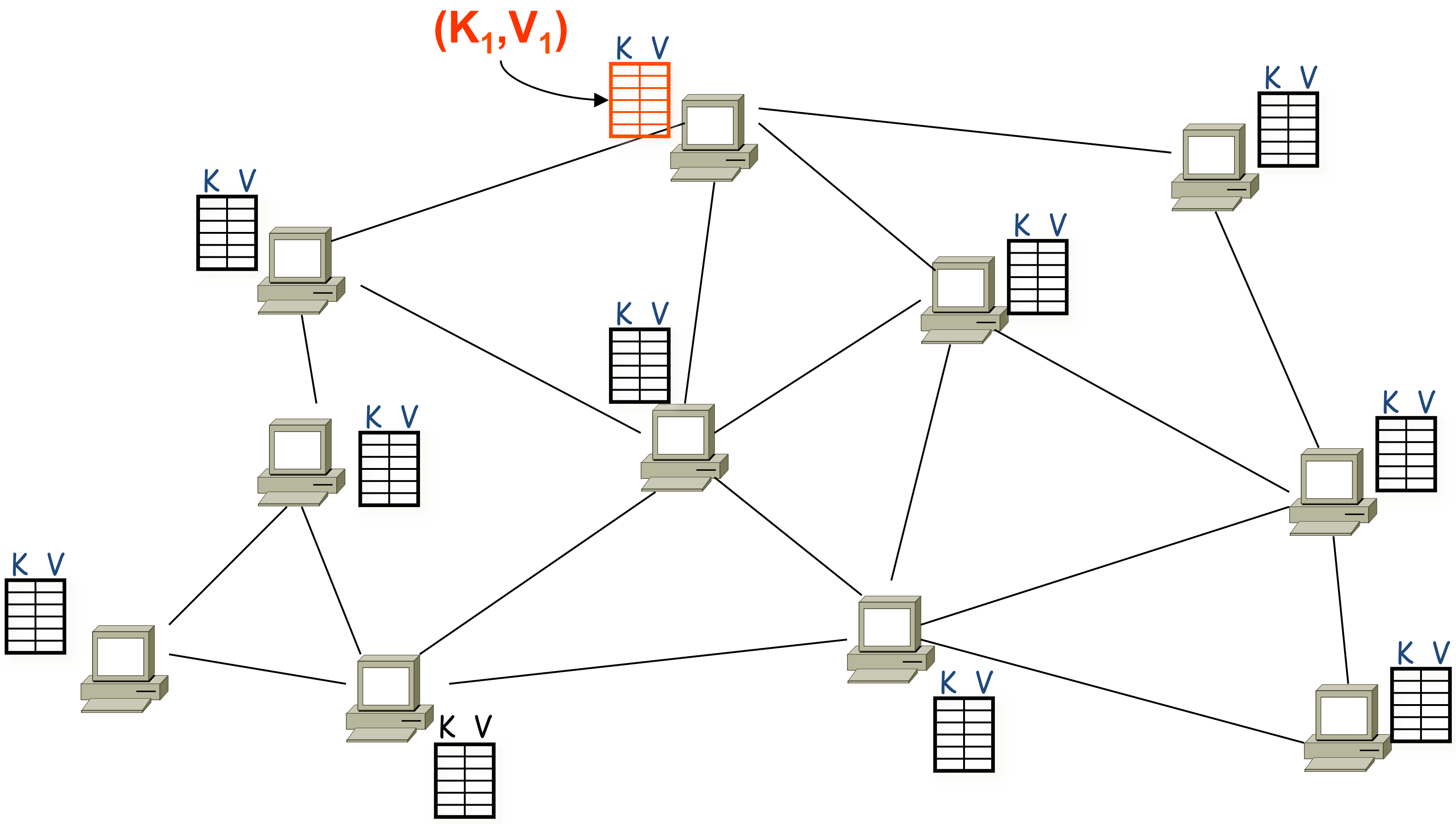
<u>Operation: take *key* as input; route messages to node holding *key*</u>

DHT: basic idea

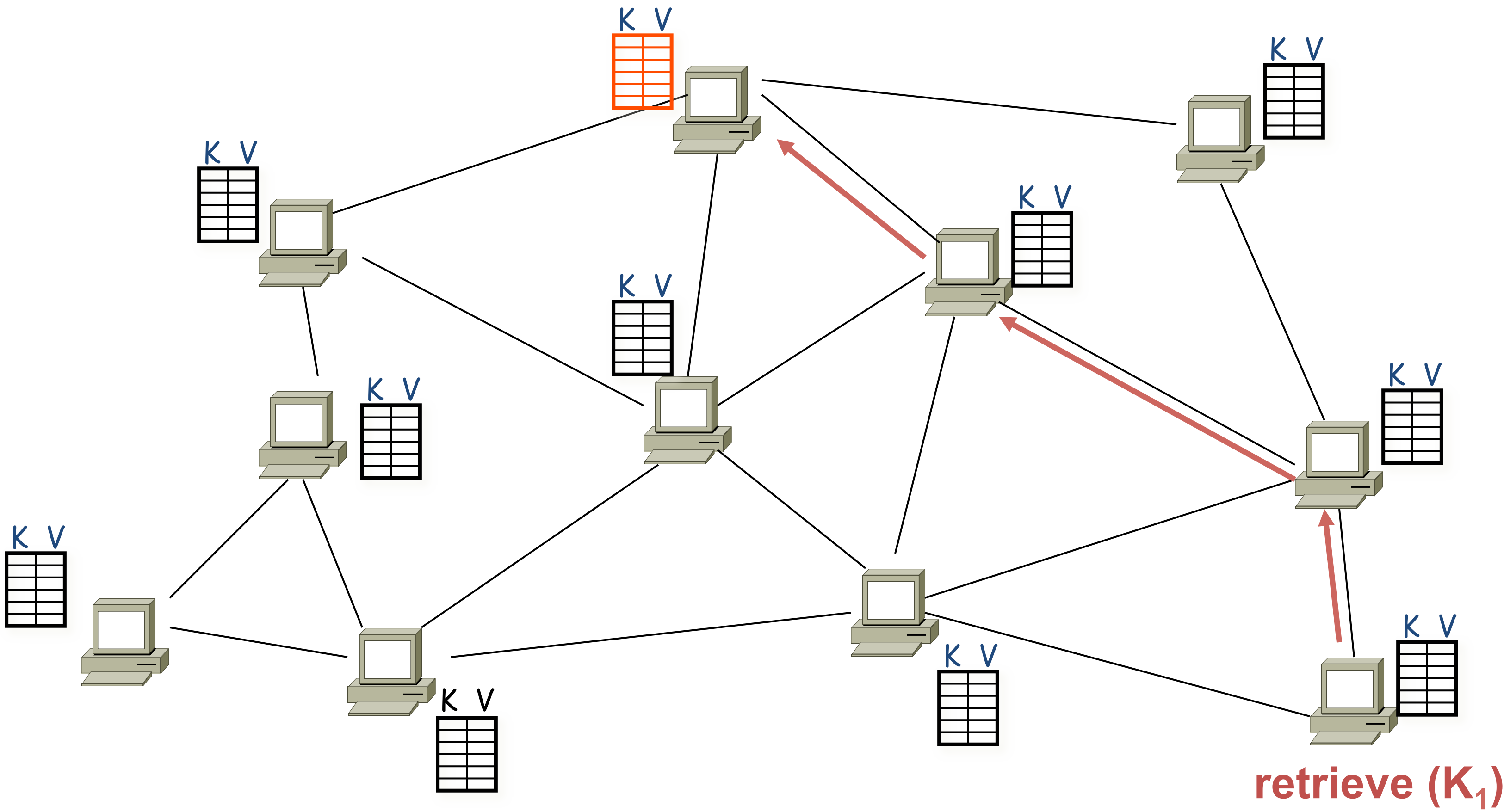


<u>Operation: take *key* as input; route messages to node holding *key*</u>

DHT: basic idea



DHT: basic idea



<u>Operation: take *key* as input; route messages to node holding *key*</u>

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 …

Taken slides from University of California, berkely and Max planck institute

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 Failures

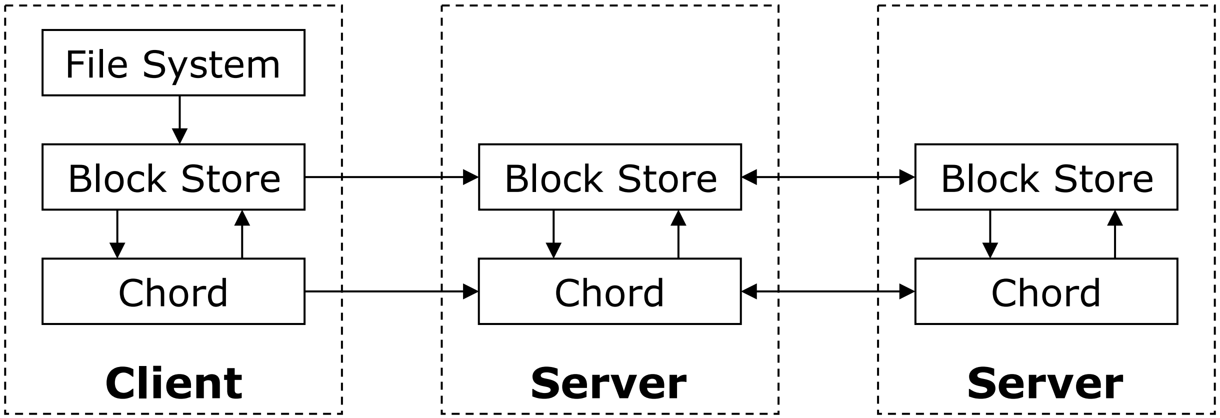
What is Chord?

- 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

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.

What is Chord? - Typical Application



- Highest layer implements application-specific functions such as file-system meta-data
- This file systems maps operations to lower-level block operations
- Block storage uses Chord to identify the node responsible for storing a block and then talk to the block storage server on that node

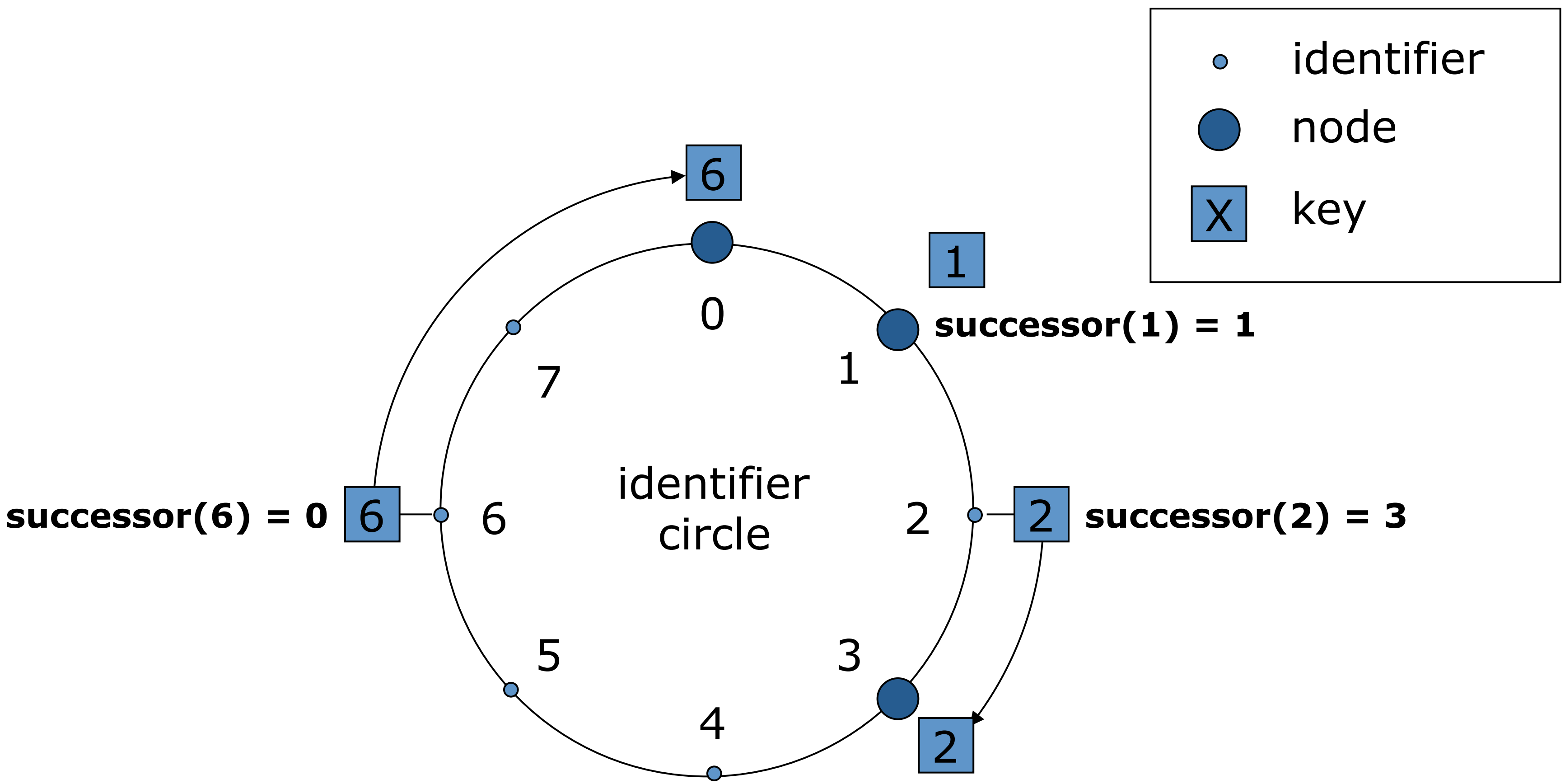
Consistent Hashing

- 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

- In an m-bit identifier space, there are 2^m identifiers.
- Identifiers are ordered on an *identifier circle* modulo 2^m.
- The identifier ring is called *Chord ring*.
- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space.
- This node is the successor node of key k, denoted by *successor(k)*.

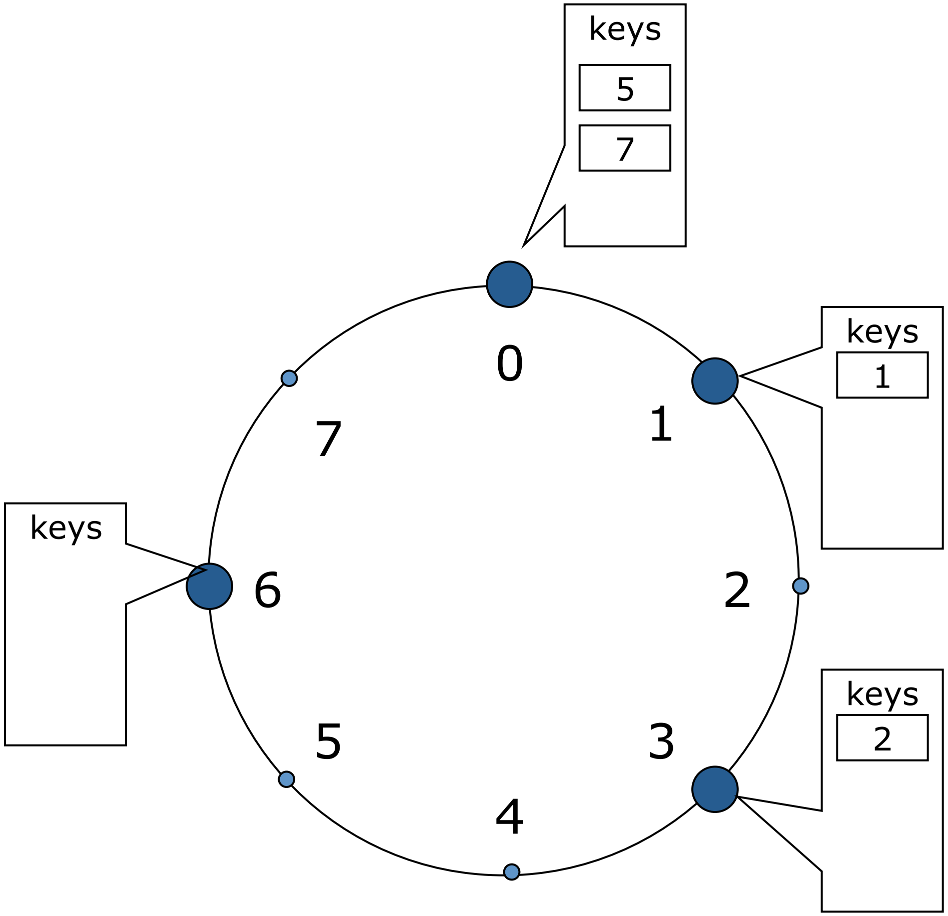
Consistent Hashing - Successor Nodes



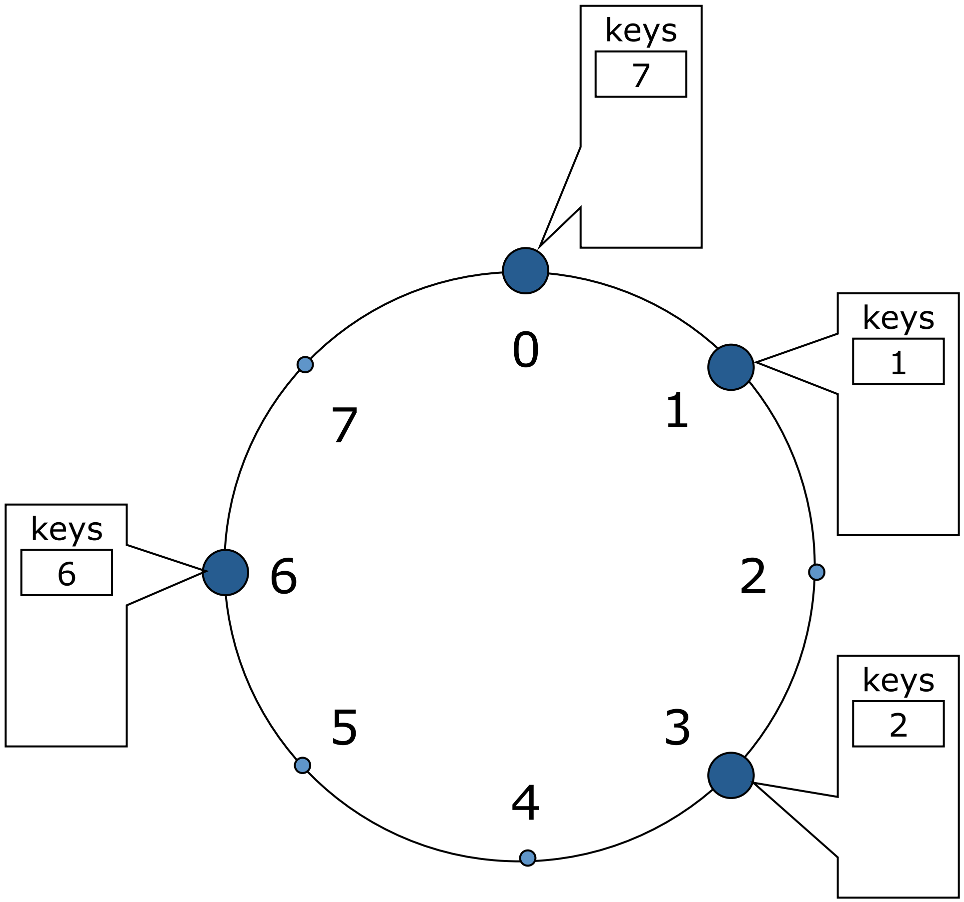
Consistent Hashing – Join and Departure

- 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 of its assigned keys are reassigned to n's successor.

Consistent Hashing – Node Join



Consistent Hashing – Node Dep.



A Simple Key Lookup

- If each node knows only how to contact its current successor node on the identifier circle, all node 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.

A Simple Key Lookup

- Pseudo code for finding successor:

// ask node n *to find the successor of* id

n.find_successor(id)

  if (id ∈ (n, *successor*])

    return *successor*;

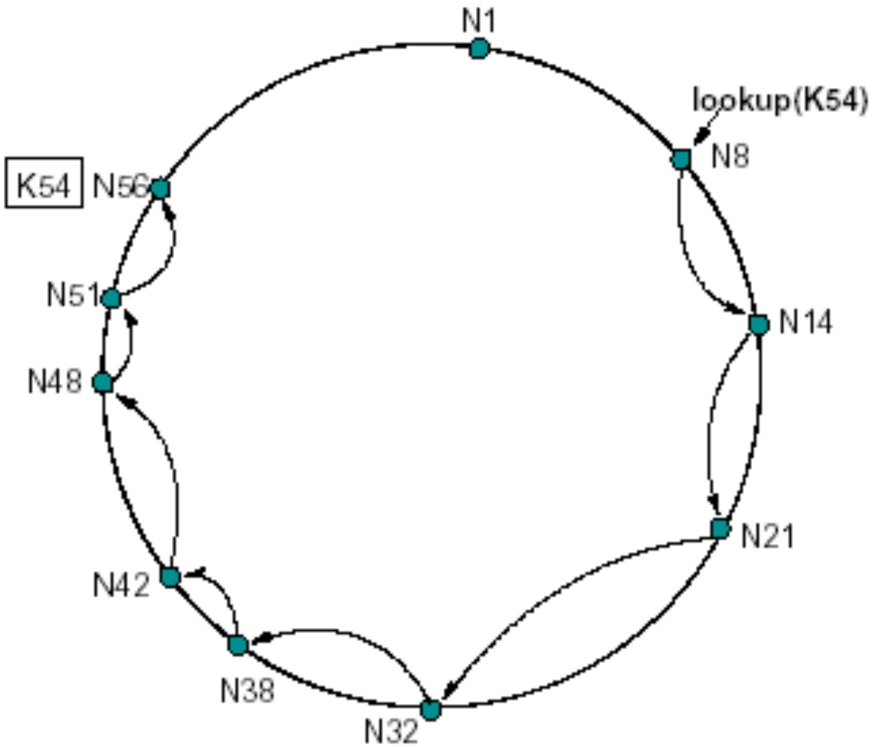
  else

    // forward the query around the circle

    return *successor.find_successor*(id);

A Simple Key Lookup

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



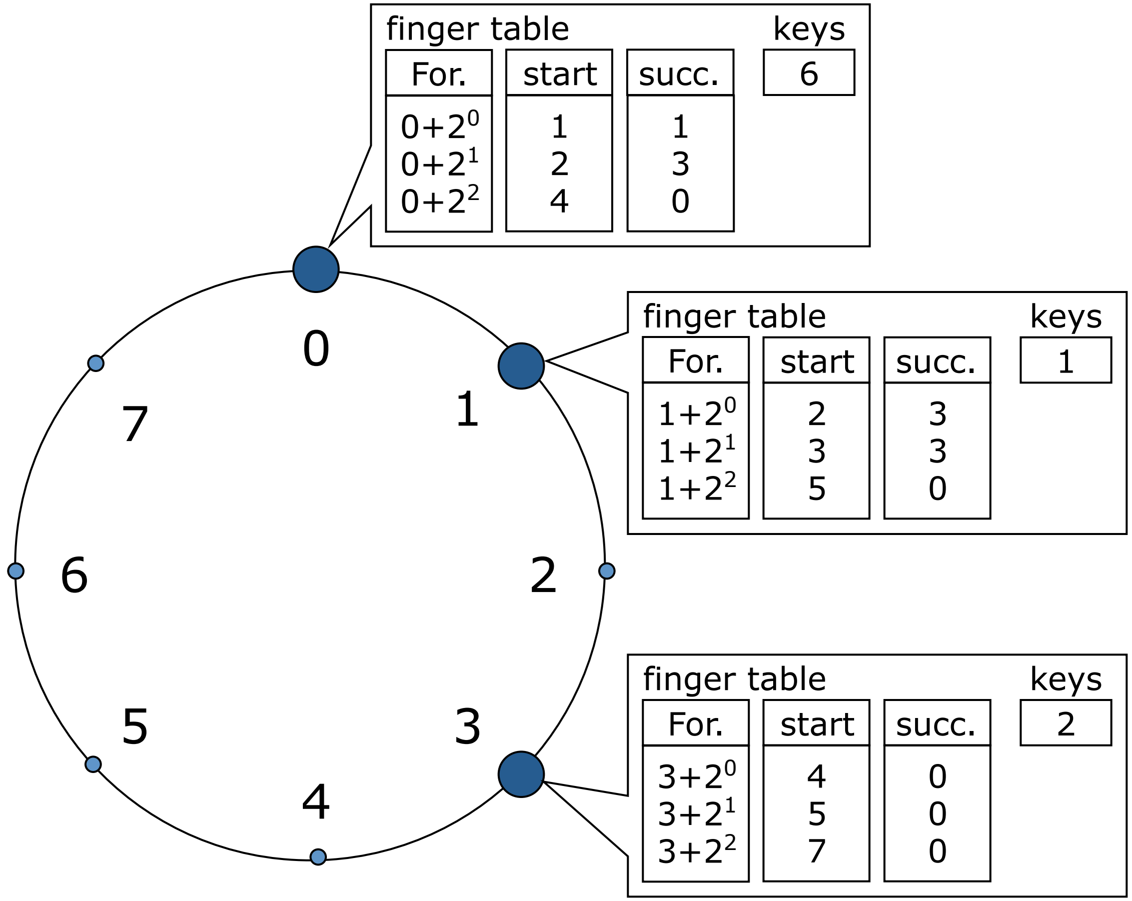
Scalable Key Location

- 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)*

Scalable Key Location – Finger Tables

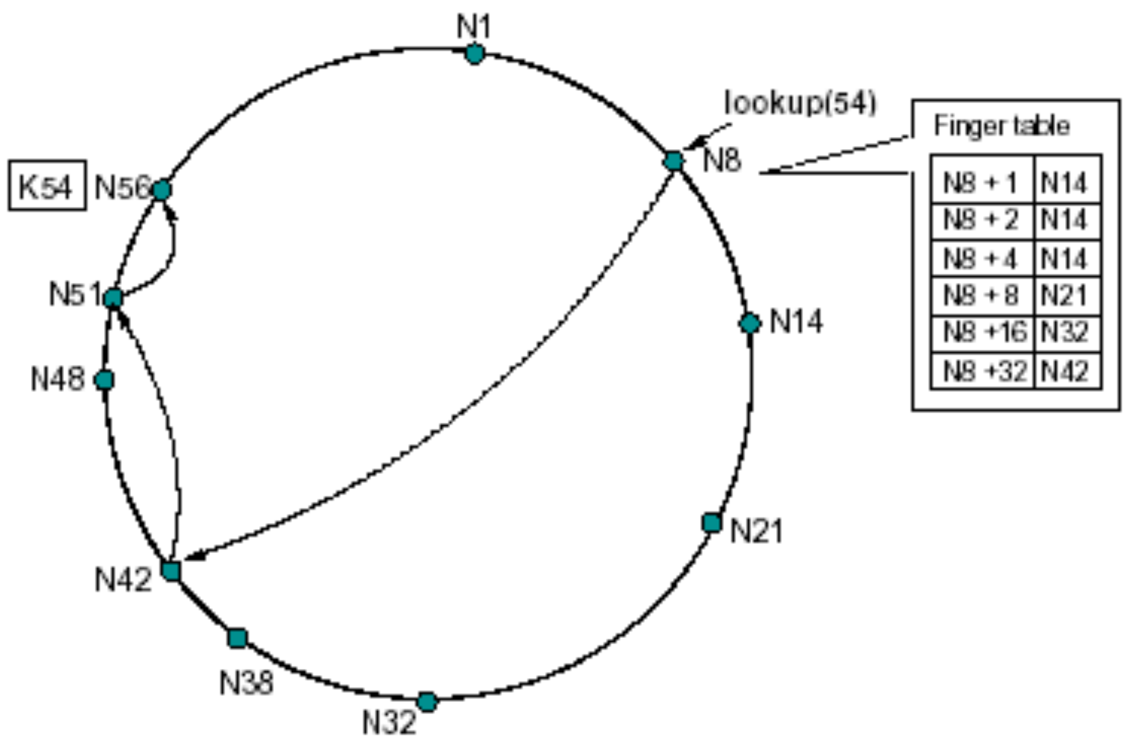


Scalable Key Location – Finger Tables

- 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 of n on the circle.

Scalable Key Location – Example query

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



Scalable Key Location – A characteristic

- 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. From this intuition follows a theorem:

  Theorem: With high probability, the number of nodes that must be contacted to find a successor in an N-node network is O(logN).

Node Joins and Stabilizations

- The most important thing is the successor pointer.
- If the successor pointer is ensured to be up to date, which is sufficient to guarantee correctness of lookups, then finger table can always be verified.
- Each node runs a “stabilization” protocol periodically in the background to update successor pointer and finger table.

Node Joins and Stabilizations

- “Stabilization” protocol contains 6 functions:
  - create()
  - join()
  - stabilize()
  - notify()
  - fix_fingers()
  - check_predecessor()

Node Joins – join()

- When node n first starts, it calls n.*join*(n’), where n’ is any known Chord node.
- The *join*() function asks n’ to find the immediate successor of n.
- *join*() does not make the rest of the network aware of n.

Node Joins – join()

// create a new Chord ring.

n.**create**()

predecessor = nil;

successor = n;

// join a Chord ring containing node n'*.*

n.**join**(n')

predecessor = nil;

successor = n'.*find_successor*(n);

Node Joins – stabilize()

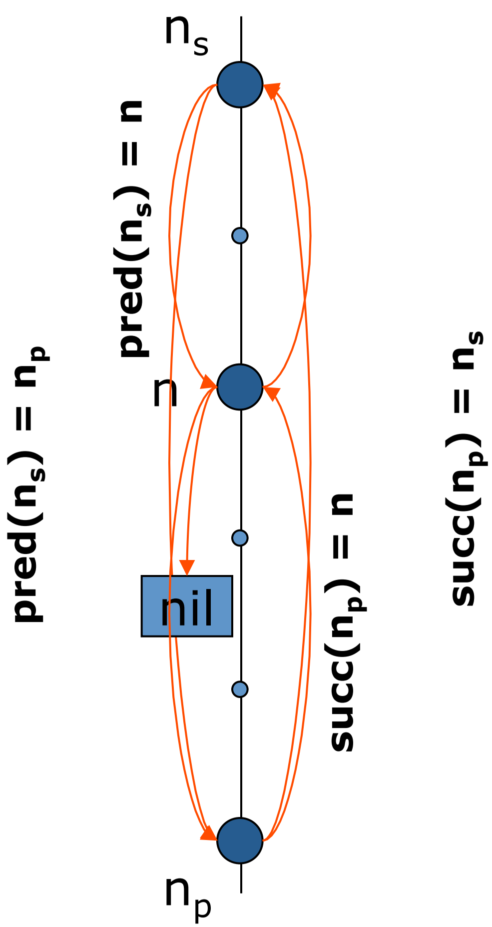
- Each time node n runs *stabilize*(), it asks its successor for it's predecessor p, and decides whether p should be n's successor instead.
- *stabilize*() notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n.
- The successor does this only if it knows of no closer predecessor than n.

Node Joins – stabilize()

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
     successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';
```

Node Joins – Join and Stabilization



- **n joins**
  - predecessor = nil
  - n acquires n_s as successor via some n'
- **n runs stabilize**
  - n notifies n_s being the new predecessor
  - n_s acquires n as its predecessor
- **n_p runs stabilize**
  - n_p asks n_s for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n
  - n will acquire n_p as its predecessor
- **all predecessor and successor pointers are now correct**
- **fingers still need to be fixed, but old fingers will still work**

Node Joins – fix_fingers()

- Each node periodically calls *fix fingers* to make sure its finger table entries are correct.
- It is how new nodes initialize their finger tables
- It is how existing nodes incorporate new nodes into their finger tables.

Node Joins – fix_fingers()

// called periodically. refreshes finger table entries.

n.**fix_fingers**()

next = next + 1 ;

if (next > m)

next = 1 ;

finger[next] = *find_successor*(n + 2^{next-1});

// checks whether predecessor has failed.

n.**check_predecessor**()

if (*predecessor* has failed)

predecessor = nil;

Node Failures

- Key step in failure recovery is maintaining correct successor pointers
- To help achieve this, each node maintains a *successor-list* of its *r* nearest successors on the ring. *Hence, all r successors would have to simultaneously fail in order to disrupt the Chord ring.*
- If node *n* notices that its successor has failed, it replaces it with the first live entry in the list
- Successor lists are stabilized as follows:
  - node n reconciles its list with its successor s by copying s's successor list, removing its last entry, and prepending s to it.
  - If node n notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor.

Chord – The Math

- Each node maintains O(logN) state information and lookups needs O(logN) messages
- Every node is responsible for about *K/N keys* (N nodes, K keys)
- When a node joins or leaves an N-node network, only *O(K/N)* keys change hands (and only to and from joining or leaving node)

Interesting Simulation Results

- Adding virtual nodes as an indirection layer can significantly improve load balance.
- The average path length is about ½(logN).
- Maintaining a set of alternate nodes for each finger and route the queries by selecting the closest node according to network proximity metric improves routing latency effectively.
- Recursive lookup style is faster iterative style

Applications: Time-shared storage

- for nodes with intermittent connectivity (server only occasionally available)
- Store others ' data while connected, in return having their data stored while disconnected
- Data 's name can be used to identify the live Chord node (content-based routing)

Applications: Chord-based DNS

- DNS provides a lookup service
  keys: host names values: IP adresses
  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

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

Thanks for your attention!