Distributed Hash Tables I

Dominic Duggan
Stevens Institute of Technology
Partly based on material by K. Birman

DISTRIBUTED HASH TABLES

Distributed Hash Tables (DHT)

- Goals
  - Enable P2P content sharing
  - Guaranteed lookup success
  - Provable bounds on search time
  - Provable scalability
- Applications
  - Dynamo, Cassandra (NoSQL)

3

DHT: Overview (1)

- Distributed "hash-table" (DHT) data structure:
  - put(id, item);
  - item = get(id);
- Implementation: distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...

4

DHT: Overview (2)

- Structured Overlay Routing:
  - Join: On startup, get a node id
  - Publish: Given key, search for node responsible
  - Search: Given key, search for node responsible
  - Fetch: Two options:
    - Get file
    - Get IP address of file server

5

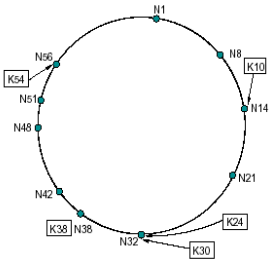
Chord: Consistent Hashing

- Associate to each node and key a unique id in an uni-dimensional space (a Ring)
  - E.g., pick from the range $[0...2^{m-1}]$
  - Usually a hash
- Properties (M = #nodes):
  - Routing table size is O(log M)
  - Key found in O(log M) hops
  - *With finger tables!*

6

Consistent Hashing

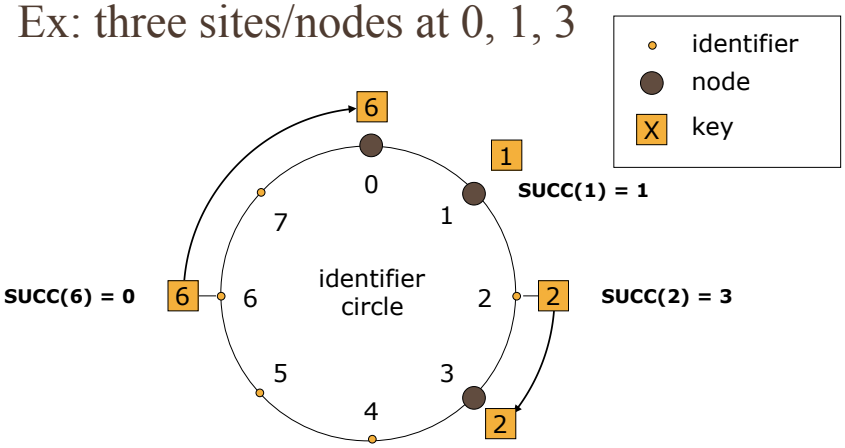
- Store key K in node N with id ID_N where:
  - $ID_N \geq K$
  - $K > ID_{PRED(N)}$
- This node denoted by SUCC(K)



- Notation:
  - SUCC(K): node that holds values for key K
  - SUCC(N): succ node in the ring
  - PRED(N): pred node in the ring

7

Ex: three sites/nodes at 0, 1, 3



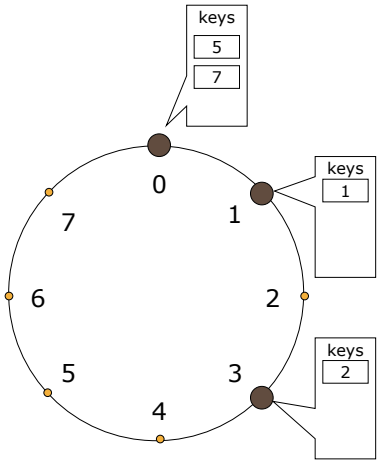
8

Join and Departure

- Node N joins network:
  - Redistribute some keys from SUCC(N)
- Node N leaves network:
  - Reassign keys to SUCC(N)

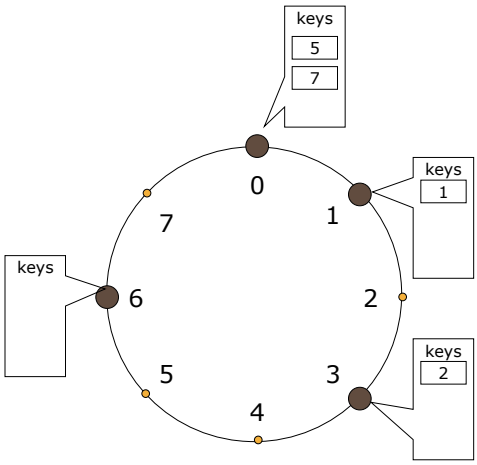
9

Initial Ring



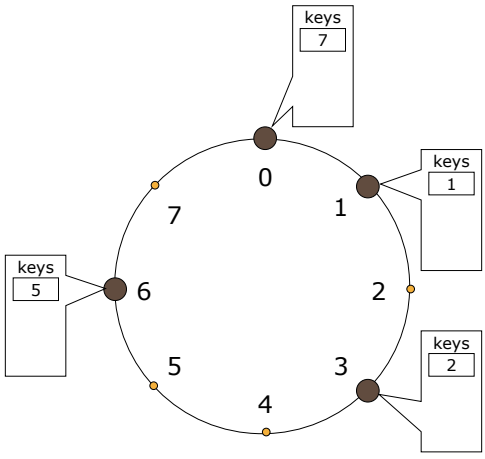
10

Node Join



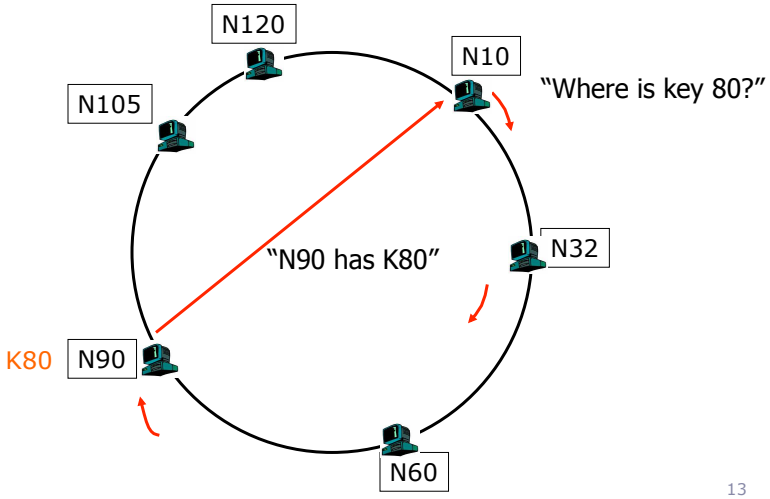
11

Node Departure



12

DHT: Basic Chord Lookup



13

A Simple Key Lookup

- Pseudo code for finding successor:

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

n.find_successor(k):
  n' =  find_predecessor(k)
  return n'.successor
```

14

A Simple Key Lookup

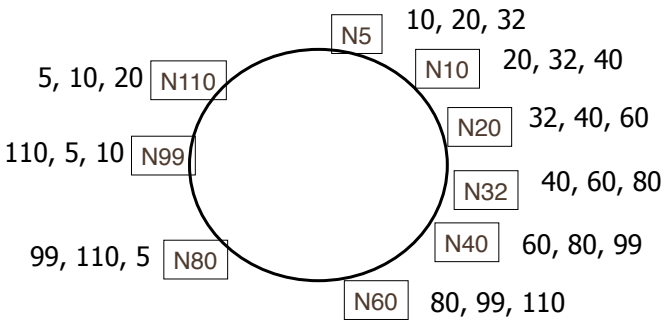
- Pseudo code for finding successor:

```
// ask node n to find the successor of id
n.find_predecessor(k):
  n' = n
  while (k ∉ (n', n'.successor])
      n' = n'.successor
  return n'

n.find_successor(k):
  n' =  find_predecessor(k)
  return n'.successor
```

15

Successor Lists Ensure Robust Lookup



- Each node remembers *r* successors
- Lookup can skip over dead nodes to find blocks
- Periodic check of successor and predecessor links

16

DHT JOIN PROTOCOL

17

Join: Expensive Approach

- New node has to:
  - Fill its own successor, predecessor
  - Notify other nodes for which it can be successor or predecessor
- Can be done in O(log M) time
- But complex
  - Impractical with high churn rate

18

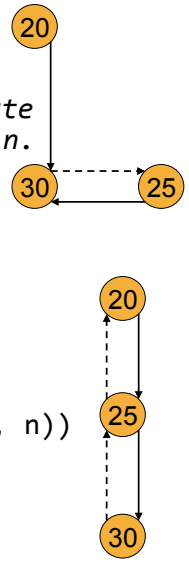
Join: Relaxed Approach

- If ring is correct, then routing is correct
- Stabilization
  - Each node periodically runs stabilization routine

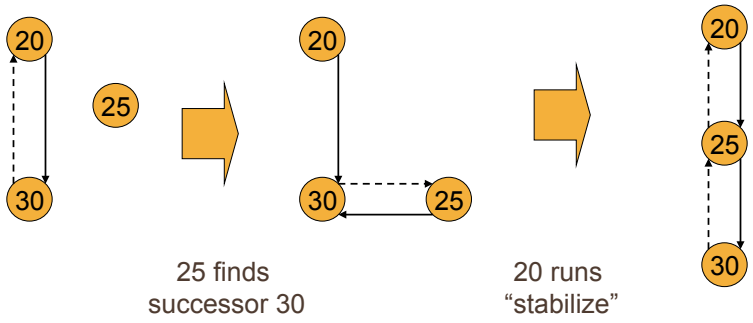
Stabilize

```
// 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';
```



Initial: 25 joins (between 20 and 30)

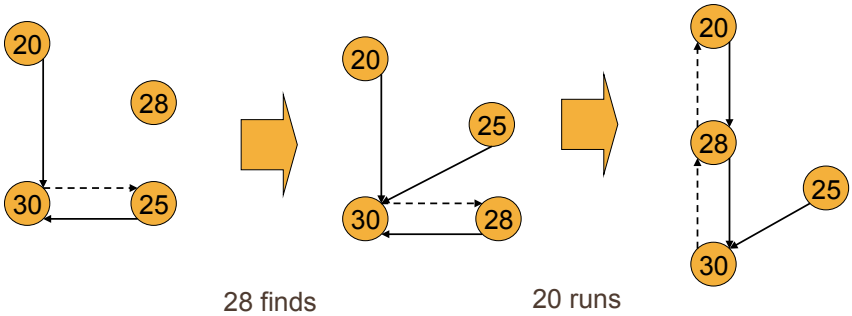


25 finds successor 30

20 runs "stabilize"

21

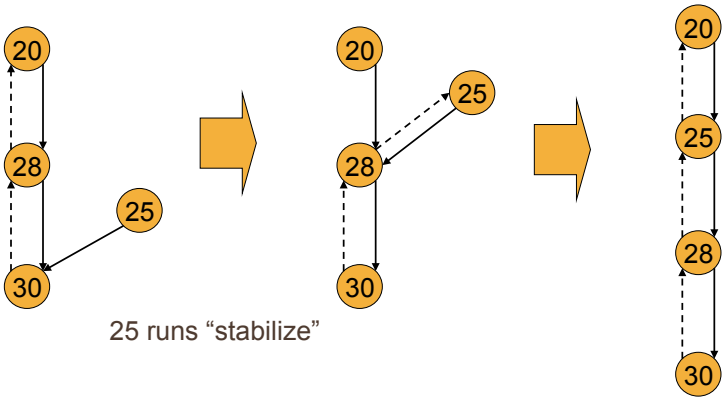
28 joins before 20 runs "stabilize"



28 finds successor 30

20 runs "stabilize"

22


20
28
25
30
25 runs “stabilize”
20
25
28
30
20
25
28
30
20 runs “stabilize”

23