CS551
Distributed Hash Tables
Structured Systems

Bill Cheng

http://merlot.usc.edu/cs551-f12

CS551
Chord
[Stoica01a]

Bill Cheng

http://merlot.usc.edu/cs551-f12

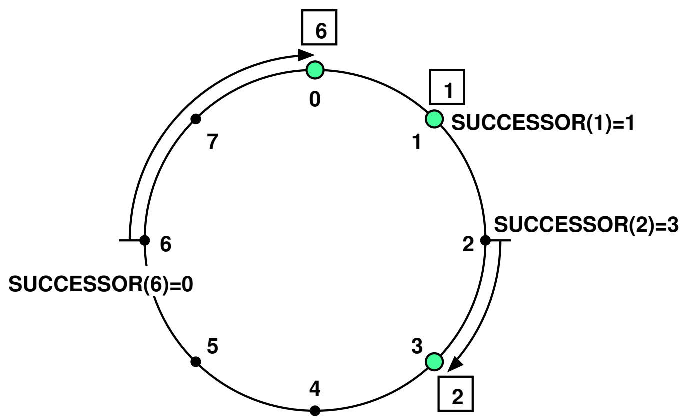
Chord

- A structured peer-to-peer system
- Map key to value
- Emphasis on good algorithmic performance
  - uses *consistent hashing*
  - O(log N) route storage, O(log N) lookup cost, $O(\log^2 N)$ cost to join/leave
  - vs. FreeNet w/emphasis on anonymity
- Easy if static, but must deal with node arrivals and departures

Compare Search in Several Peer-to-Peer Systems

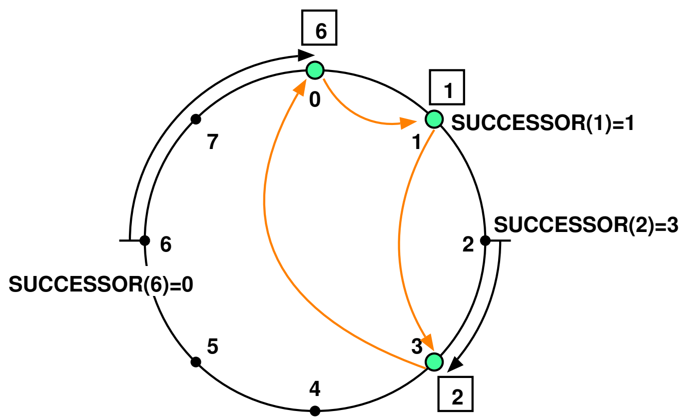
- Napster: central search engine
- Freenet: search towards keys, but no guarantees
- Chord:
  - map keys to linear search space
  - keep pointers (*fingers*) into exponential places around space
  - probabilistic (depends on hashing)

Hashing Nodes and Data



- Nodes hash IP addresses to key space
  - because this hashing is random, can expect nodes to be evenly distributed in key space
- Store data in the ***successor*** of the data item's key
- Property:
  - If each node maintains successor,
  - ... can find any data item

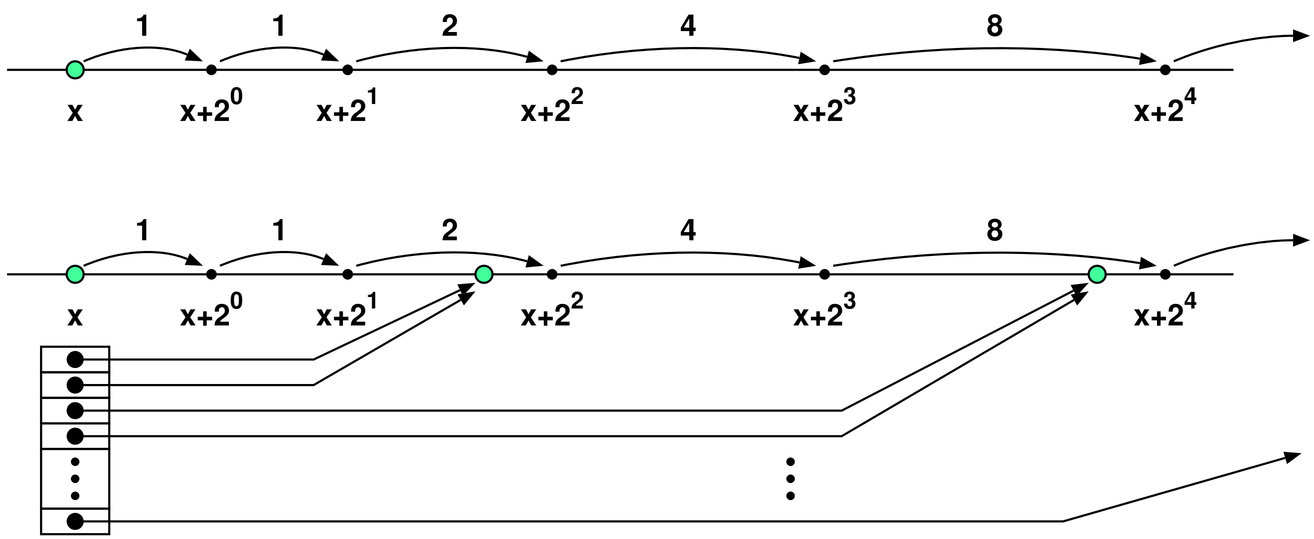
Hashing Nodes and Data



- Nodes hash IP addresses to key space
  - because this hashing is random, can expect nodes to be evenly distributed in key space
- Store data in the ***successor*** of the data item's key
- Property:
  - If each node maintains successor,
  - ... can find any data item
- Nodes have a ***successor*** pointer
  - but O(n) performance

Improving Search Performance with Finger Tables

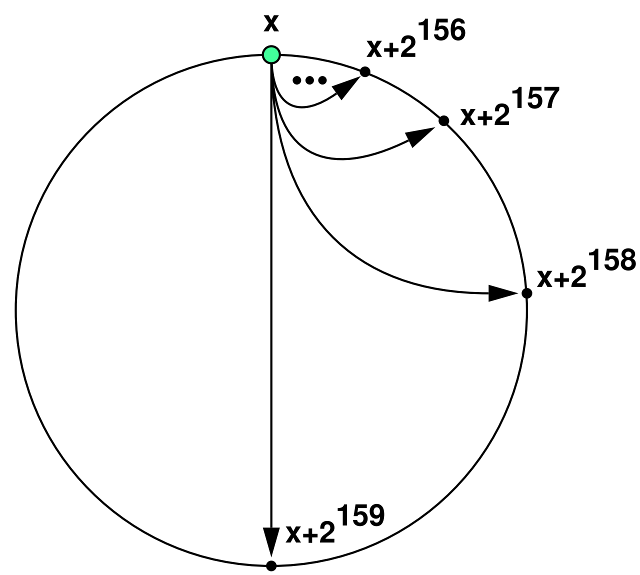
- Finger tables enable logarithmic lookup
  - i-th finger of node x is successor of $x+2^{i-1}$
  - at each step, we halve the remaining distance (in key space) to the target



- Challenge: maintaining finger tables!

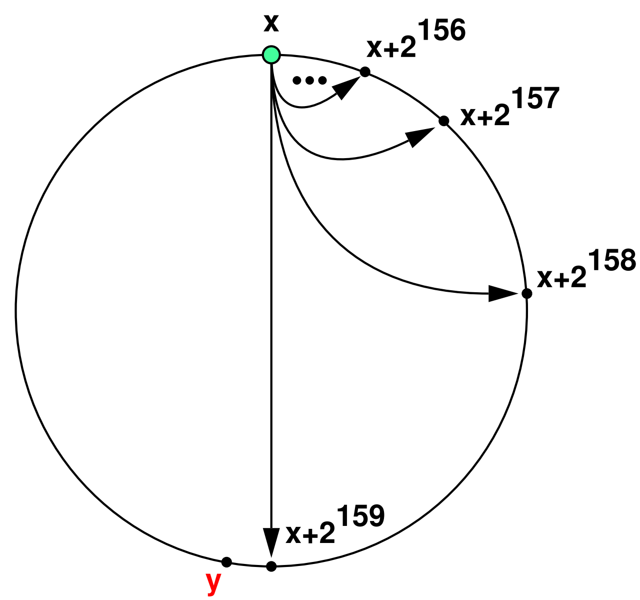
Improving Search Performance with Finger Tables (Cont...)

- Finger tables enable logarithmic lookup
  - i-th finger of node x is successor of $x+2^{i-1}$
  - at each step, we halve the remaining distance (in key space) to the target
    - Ex: look for key y



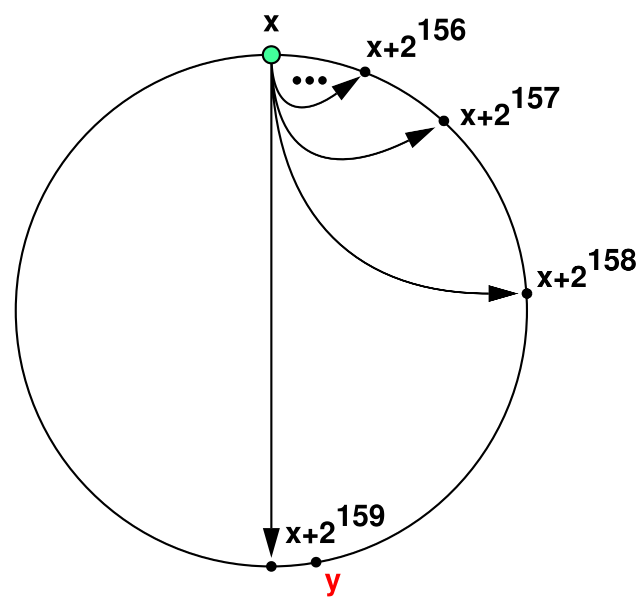
Improving Search Performance with Finger Tables (Cont...)

- Finger tables enable logarithmic lookup
  - i-th finger of node x is successor of $x+2^{i-1}$
  - at each step, we halve the remaining distance (in key space) to the target
    - Ex: look for key y
      - case 1: y is just beyond $x+2^{159}$
      - forward to successor($x+2^{159}$)
      - way more than half the distance to y



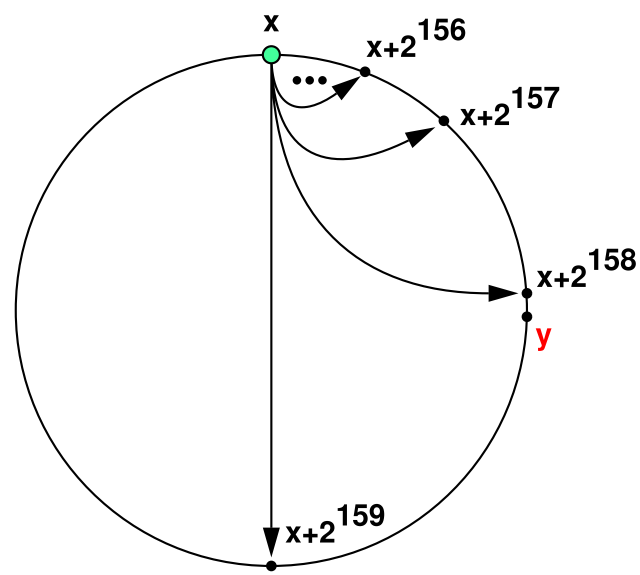
Improving Search Performance with Finger Tables (Cont...)

- Finger tables enable logarithmic lookup
  - i-th finger of node x is successor of $x+2^{i-1}$
  - at each step, we halve the remaining distance (in key space) to the target
    - Ex: look for key y
      - case 2: y is just inside $x+2^{159}$
      - forward to successor($x+2^{158}$)
      - a little over half the distance to y

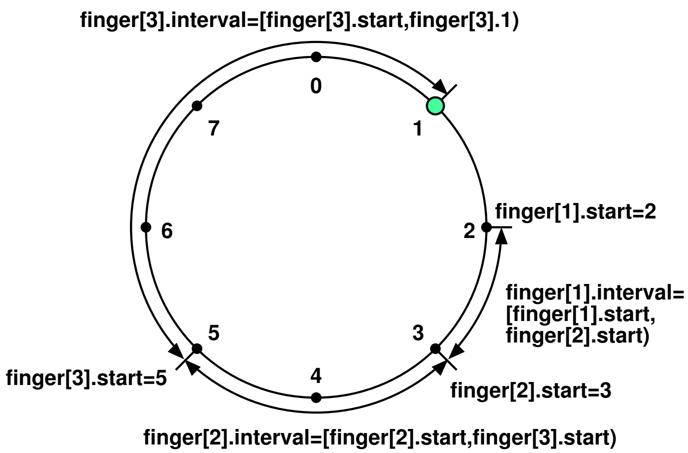


Improving Search Performance with Finger Tables (Cont...)

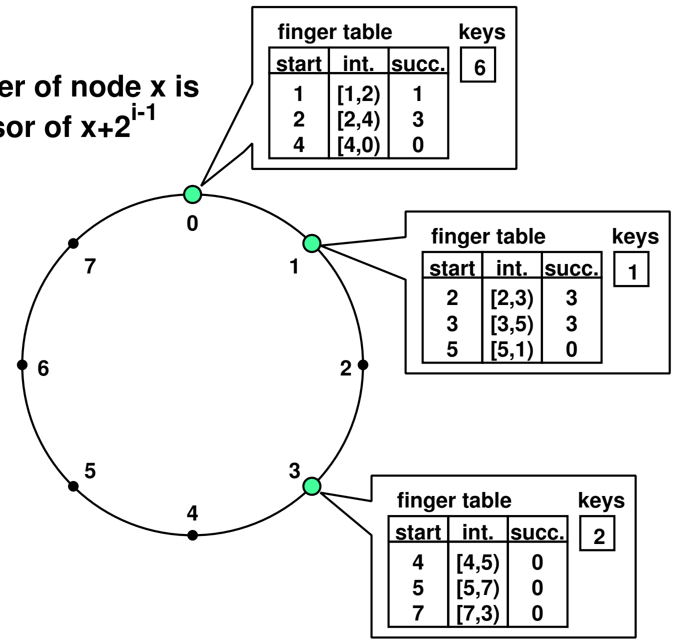
- Finger tables enable logarithmic lookup
  - i-th finger of node x is successor of $x+2^{i-1}$
  - at each step, we halve the remaining distance (in key space) to the target
    - Ex: look for key y
      - case 3: y is just beyond $x+2^{158}$
      - forward to successor($x+2^{158}$)
      - way more than half the distance to y
      - and so on...



Finger Tables Example



i-th finger of node x is successor of $x+2^{i-1}$



Node 0 — finger table (keys: 6)

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

Node 1 — finger table (keys: 1)

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

Node 3 — finger table (keys: 2)

start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

Node Joins

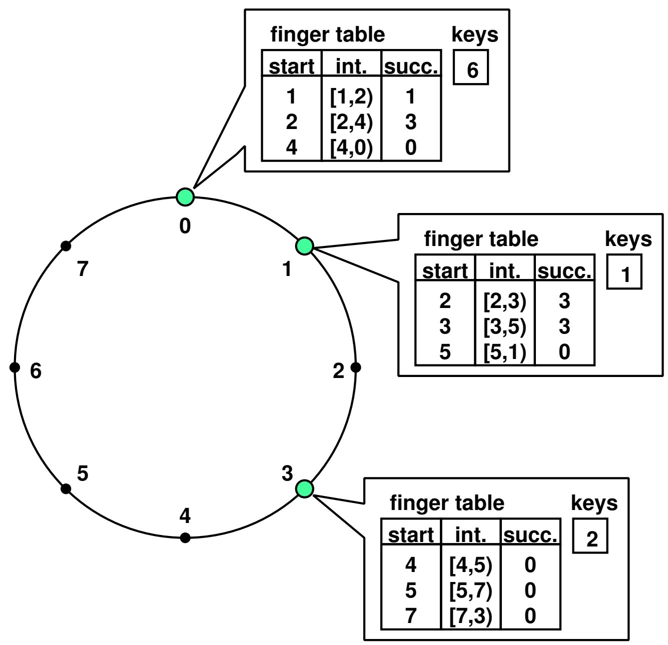
- Must keep successors and finger table current
- Use successors for *correctness*
  - can always fall back on them to find a key
- Use finger table for *performance*
  - must update it, but can tolerate temporary errors
- Keep successor and *predecessor* so we can update our neighbors
- Key observation: can find successors and fingers by doing a lookup on the existing Chord ring!

Finding Predecessor and Successor

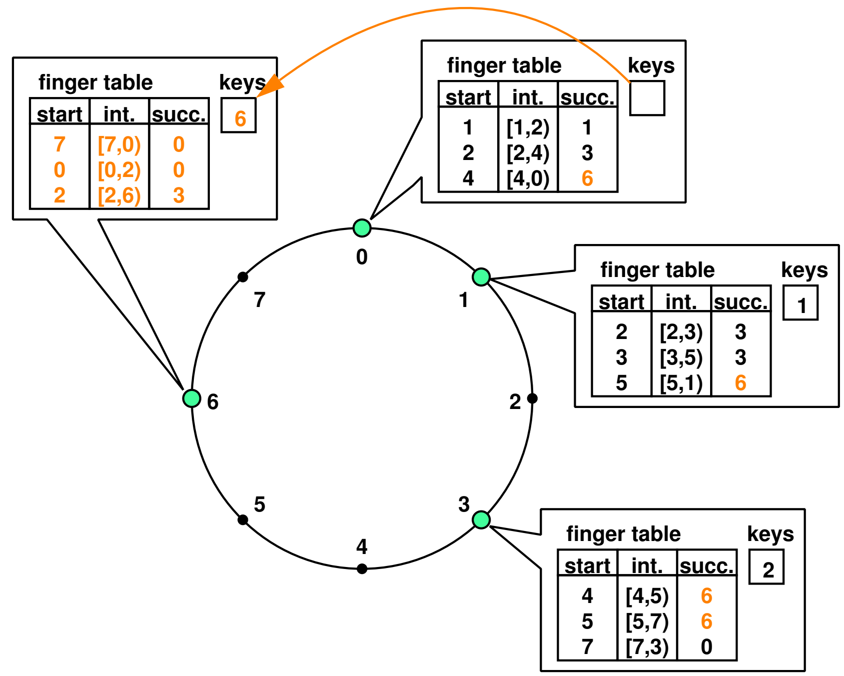
```
node.find_successor(key)
    n = find_predecessor(key);
    return n.successor;

node.find_predecessor(key)
    n = node;
    while (key ∉ (n,n.successor])
        n = n.closest_preceding_finger(key);
    return n;

node.closest_preceding_finger(key)
    for (i=m; i > 0; i--)
        if (finger[i].node ∈ (node,key))
            return finger[i].node;
    return node;
```

Join Example

before node 6 joins



before node 6 joins



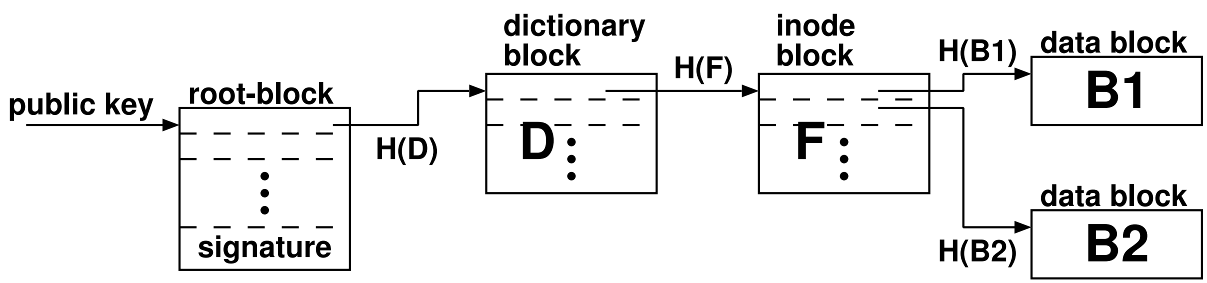
- when new node enters, it establishes its successor and predecessor and then builds its finger table, and moves any keys it now "owns"

Robustness

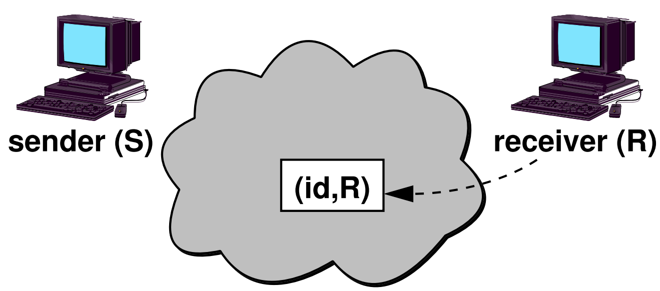
- ***Stabilization*** algorithm to confirm ring is correct
  - every 30s, ask successor for its predecessor
    - fix your own successor based on this
    - successor fixes its predecessor if necessary
  - also, pick and verify a random finger table entry
    - rebuild finger table entries this way
    - important observation: finger tables can be incorrect for some time (between network sizes of N and 2N)
- Dealing with unexpected failures:
  - keep successor list of r successors
  - can use these to replicate data

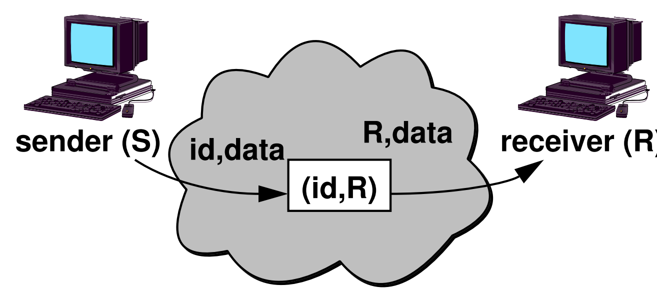
Applications

File Systems



Multicast and Anycast (using rendezvous)





Chord Performance

- Performance dominated by ***lookup*** cost
  - how long does it take to get to the node that stores a key?
- Chord promises few O(logN) hops on the overlay
  - but, on the physical network, this can be quite far
    - this is often the problem with *overlay networks*