The Chord P2P Network

Some slides have been borrowed from the original presentation by the authors

Chord vs.Tapestry

- The topology of the Chord network, as defined by the successor pointers, must satisfy a well-defined structure.
- Tapestry (uses Plaxton routing) requires the root of the object to be placed in a designated node, but the object can be placed locally. In contrast, Chord requires the object to be placed at a designated node.

Main features of Chord

- Load balancing via Consistent Hashing
- Small routing tables: **log n**
- Small routing delay: **log n** hops
- Fast join/leave protocol (polylog time)

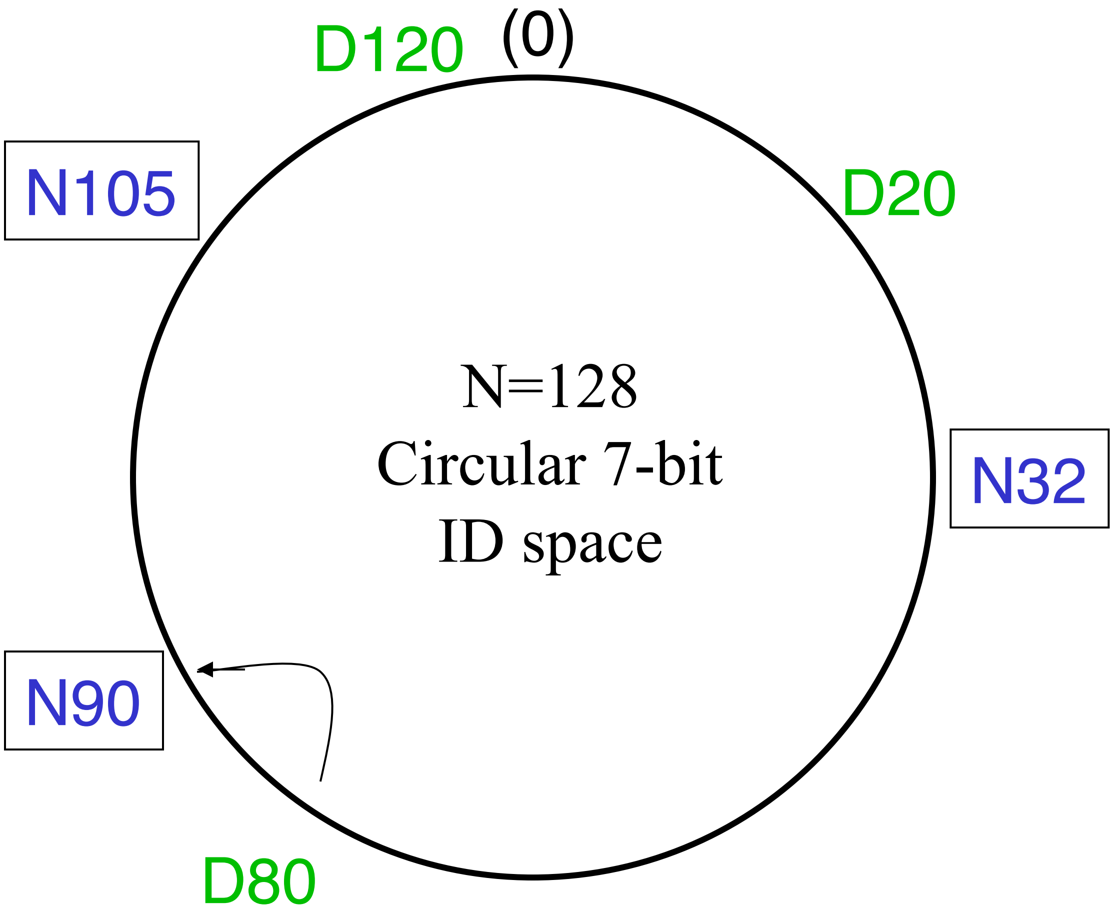
Consistent Hashing

Assigns both nodes and objects from an m-bit key.

Order these nodes around an identifier circle (what does a circle mean here?) according to the order of their keys (0 .. 2^m-1). This ring is known as the Chord Ring.

Object with key k is assigned to the ***first node*** whose key is ≥ k (called the successor node of key k)

Consistent Hashing



Example: Node 90 is the “successor” of document 80.

Consistent Hashing [Karger 97]

Property 1

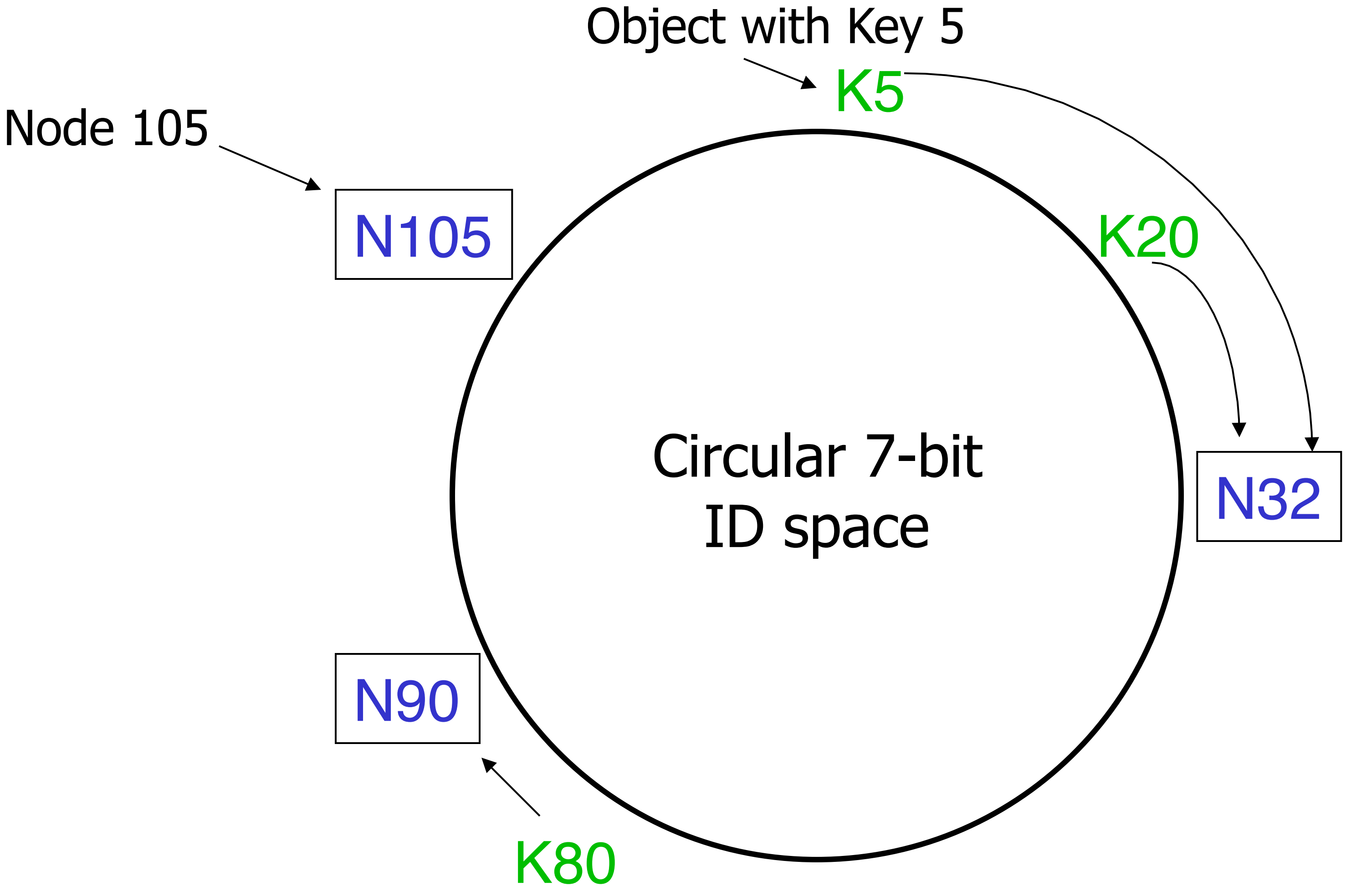
If there are N nodes and K keys, then *with high probability*, each node is responsible for (1+epsilon)K/N keys.

Property 2

When a node joins or leaves the network, the responsibility of at most O(K/N) keys changes hand (only to or from the node that is joining or leaving.

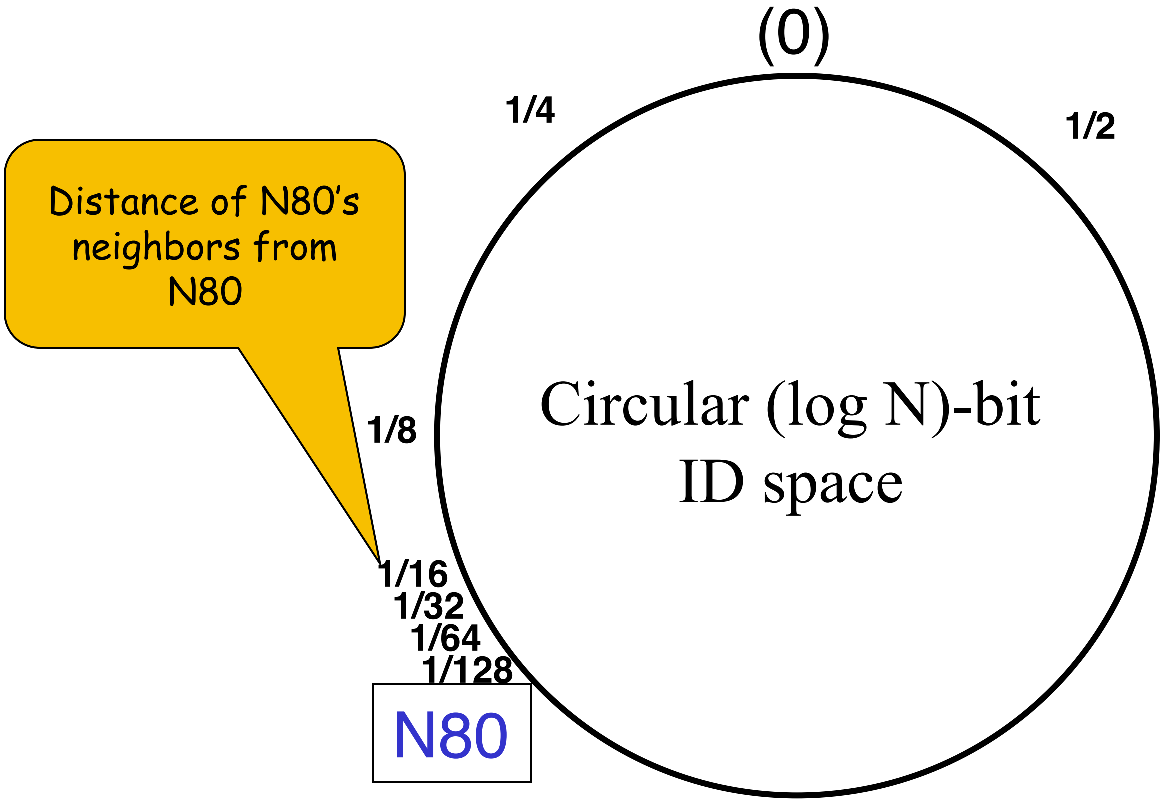
When K is large, the impact on individual nodes is quite small.

Consistent hashing



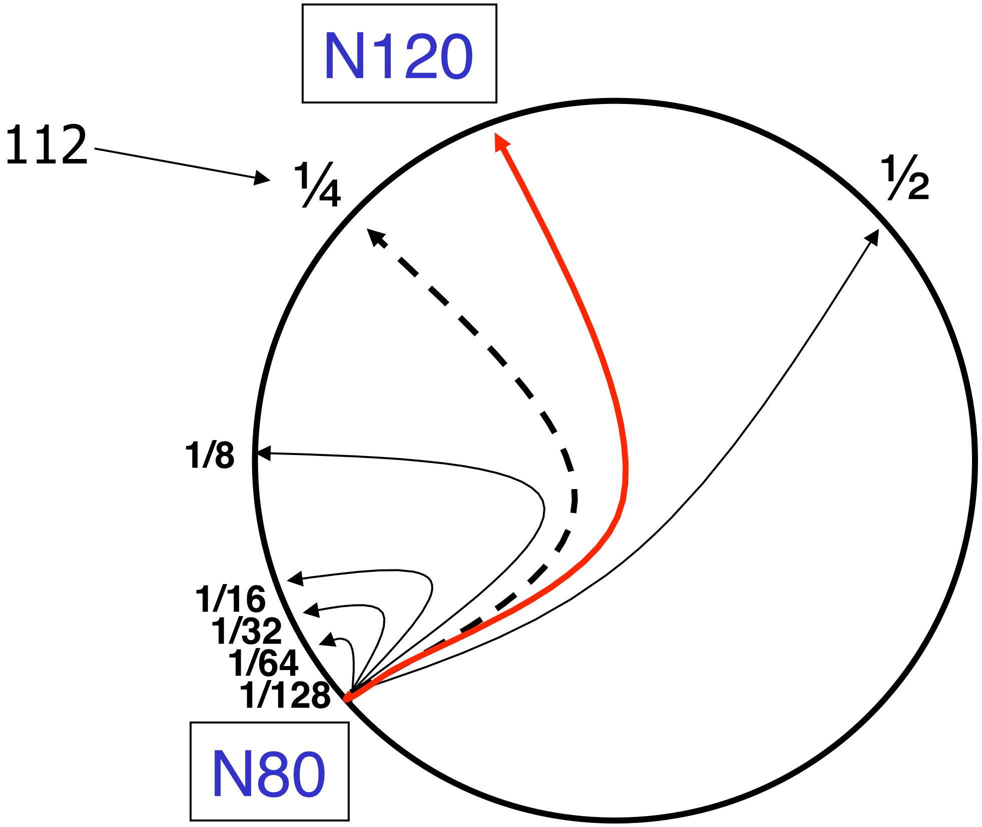
An object with key **k** is stored at its successor (node with key ≥ k)

The log N *Fingers*

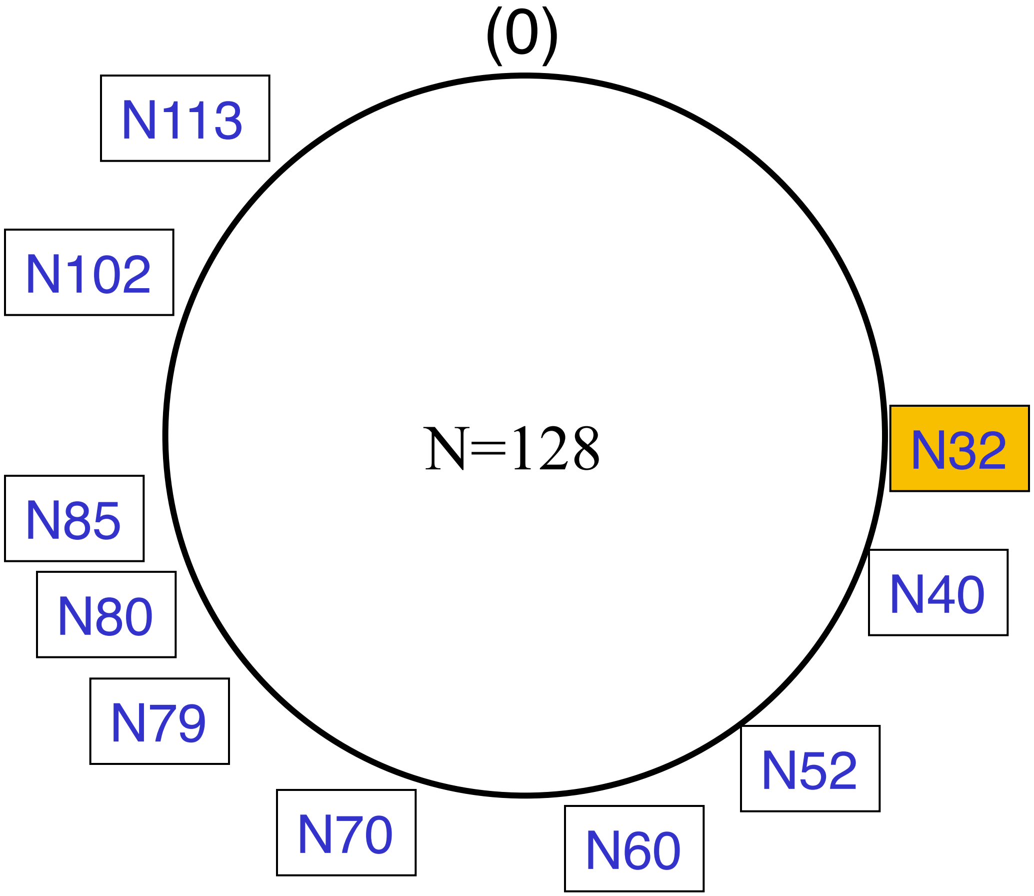


Each node knows of only **log N** other nodes.

Finger i points to successor of $n+2^i$



Chord Finger Table



N32's Finger Table

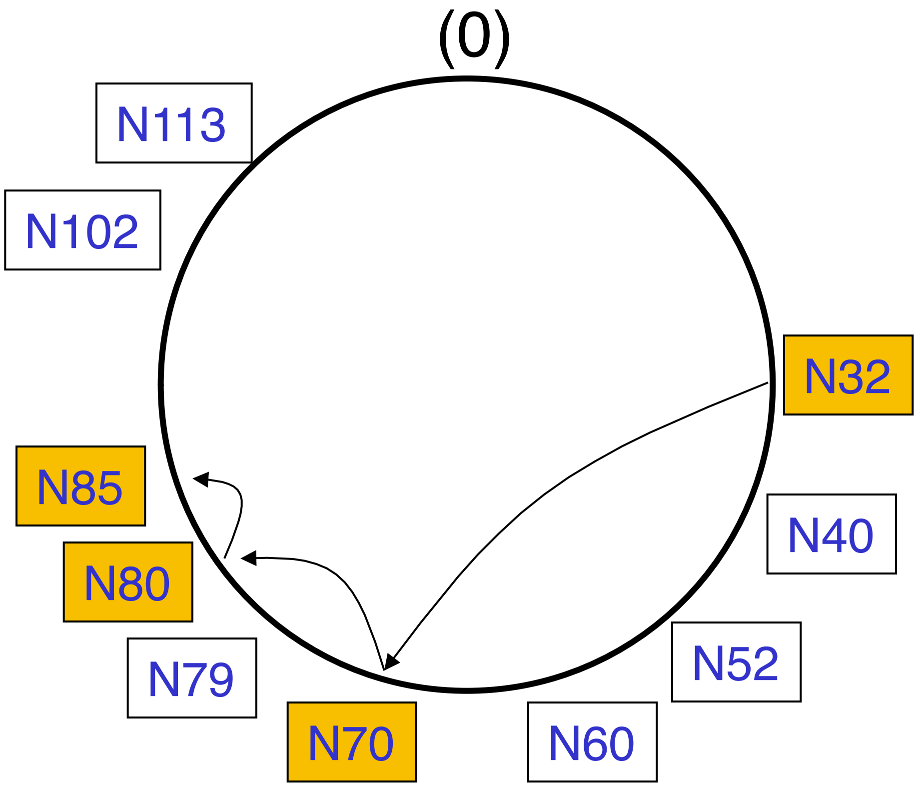
33..33	N40
34..35	N40
36..39	N40
40..47	N40
48..63	N52
64..95	N70
96..31	N102

Finger table actually contains ID and IP address

Node n's i-th entry: first node $\geq n + 2^{i-1}$

Lookup

Greedy routing



N32's
Finger Table

33..33	N40
34..35	N40
36..39	N40
40..47	N40
48..63	N52
64..95	**N70**
96..31	N102

N70's
Finger Table

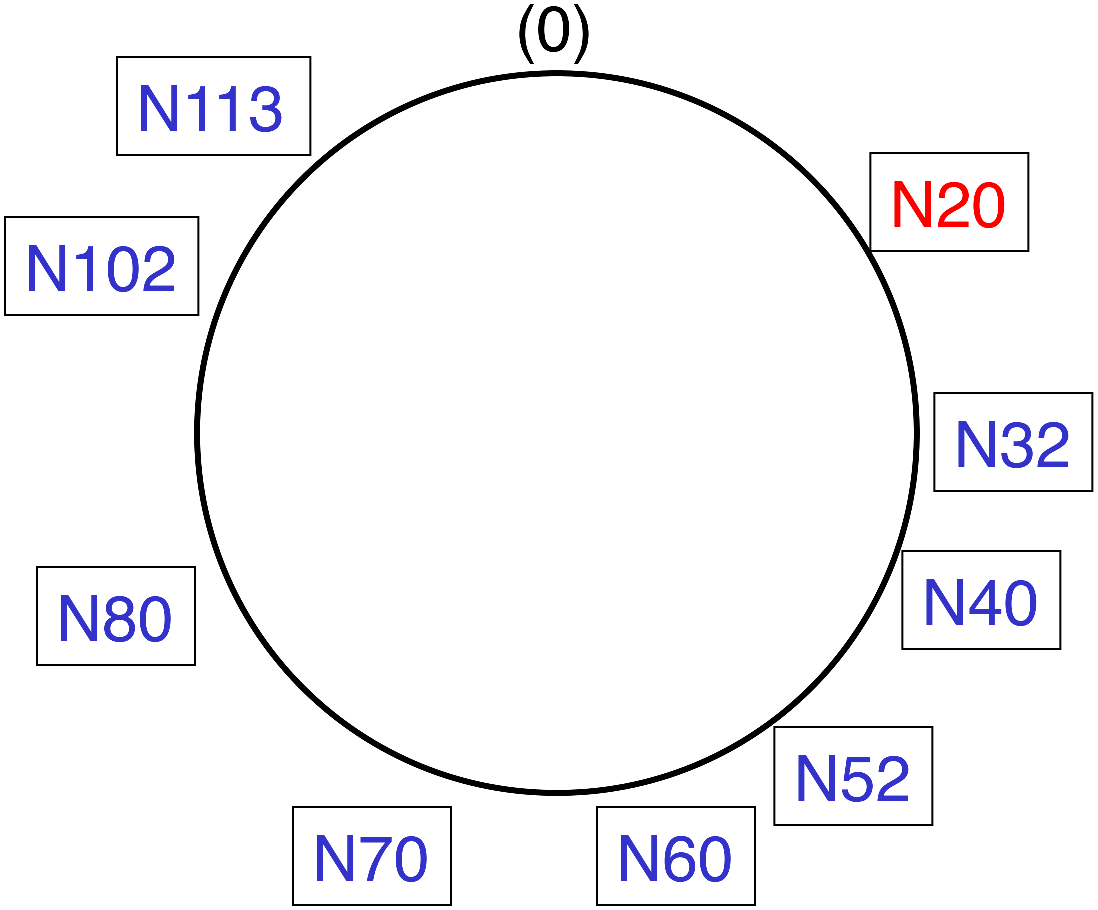
71..71	N79
72..73	N79
74..77	N79
78..85	**N80**
86..101	N102
102..5	N102
6..69	N32

N80's
Finger Table

81..81	N85
82..83	N85
84..87	**N85**
88..95	N102
96..111	N102
112..15	N113
16..79	N32

Node 32, lookup(82): 32 → 70 → 80 → 85.

New Node Joins

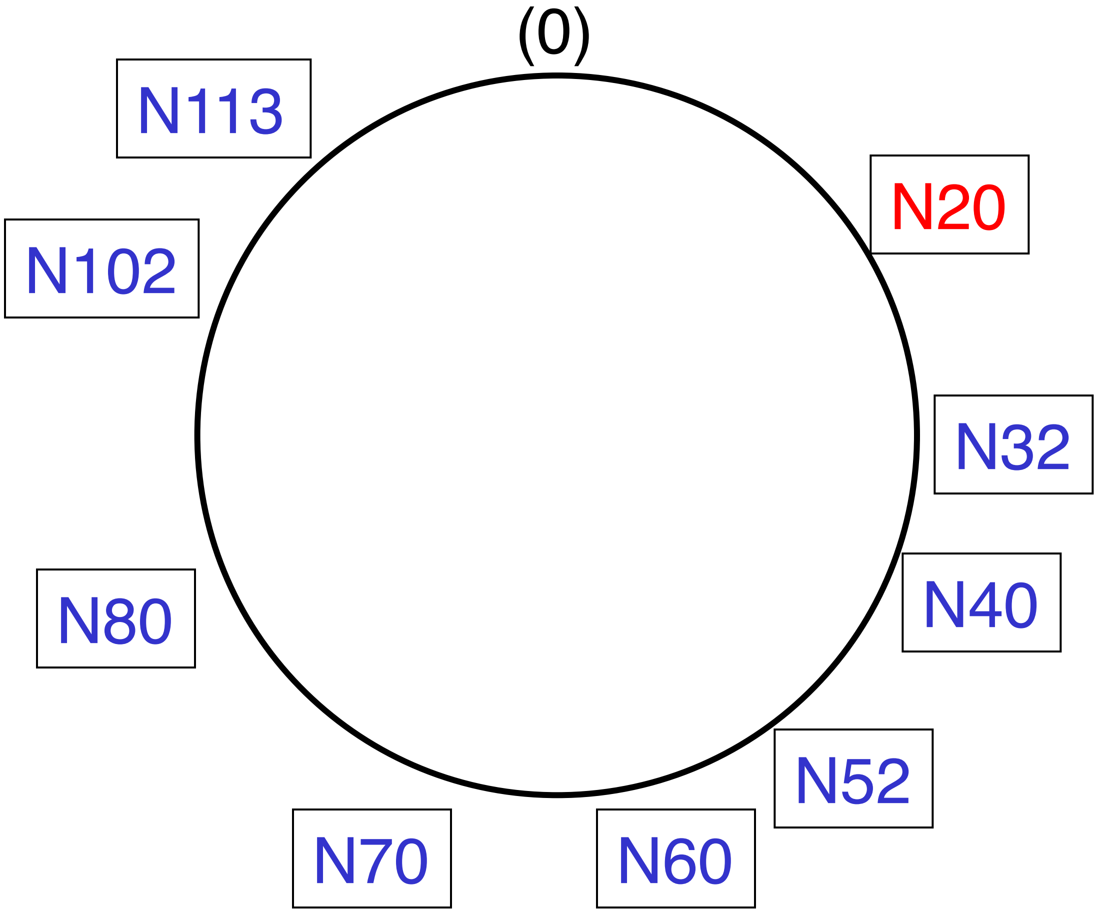


N20's
Finger Table

1	21..21
2	22..23
3	24..27
4	28..35
5	36..51
6	52..83
7	84..19

Assume N20 knows one of the existing nodes.

New Node Joinsw (2)



N20's
Finger Table

21..21	N32
22..23	N32
24..27	N32
28..35	N32
36..51	N40
52..83	N52
84..19	N102

Node 20 asks *that* node for successor to 21, 22, …, 52, 84.

The Join procedure

The new node id asks a gateway node n

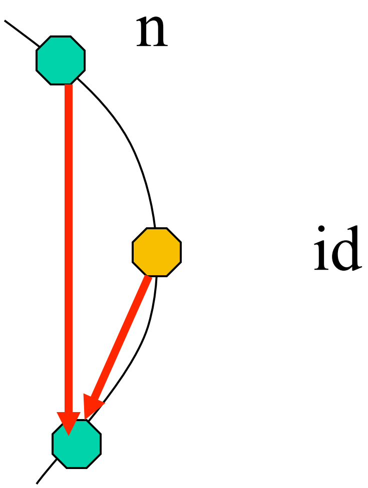
to find the successor of id

```
n.(find_successor(id))
if      id = (n, successor]
        then    return successor
        else    forward the query around the circle
fi
```

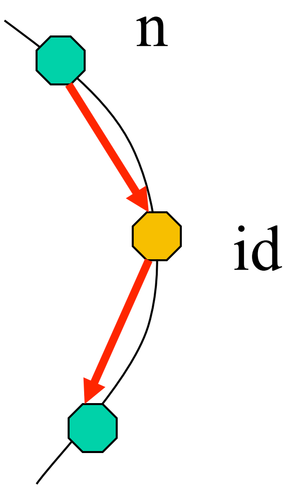
Needs O(n) messages. This is slow.

Steps in join

Linked list insert



Successor(n)



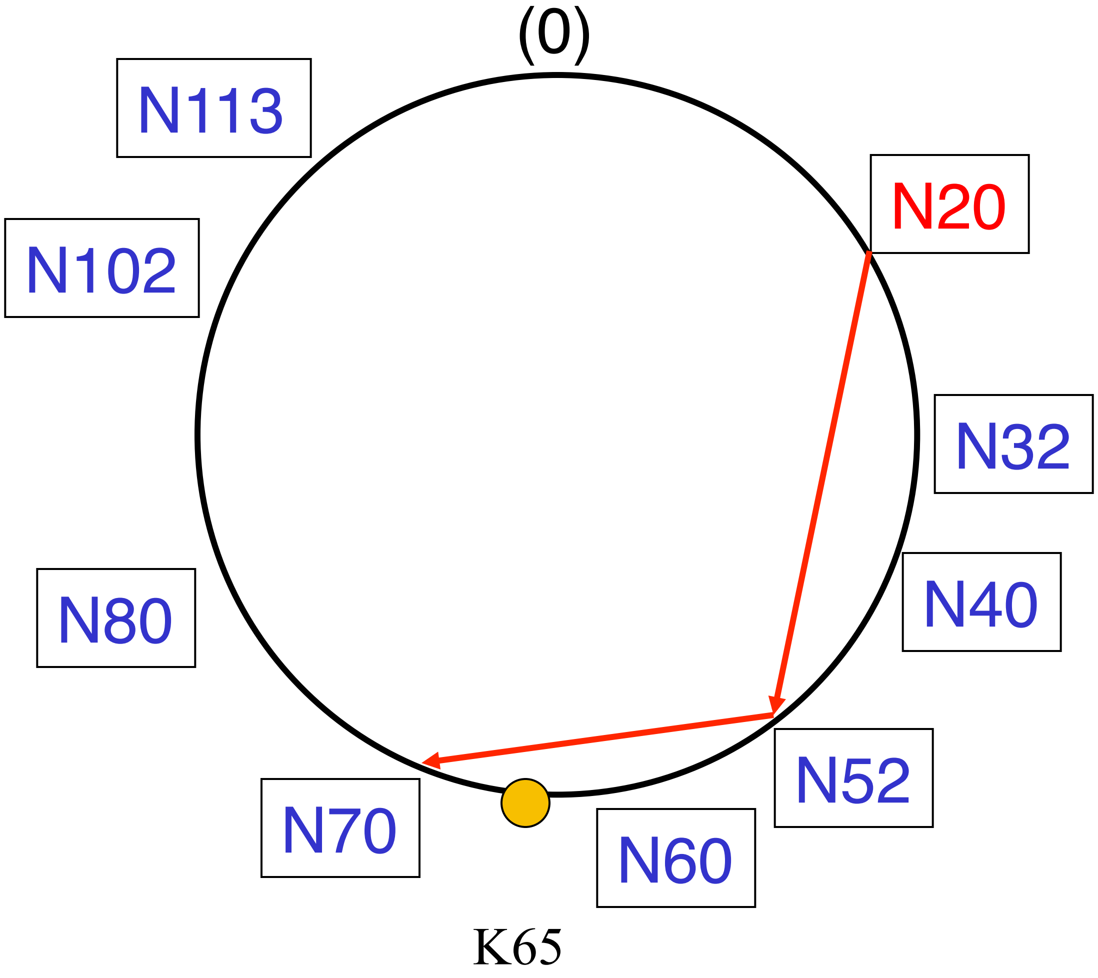
Finally

But the transition does not happen immediately

A More Efficient Join

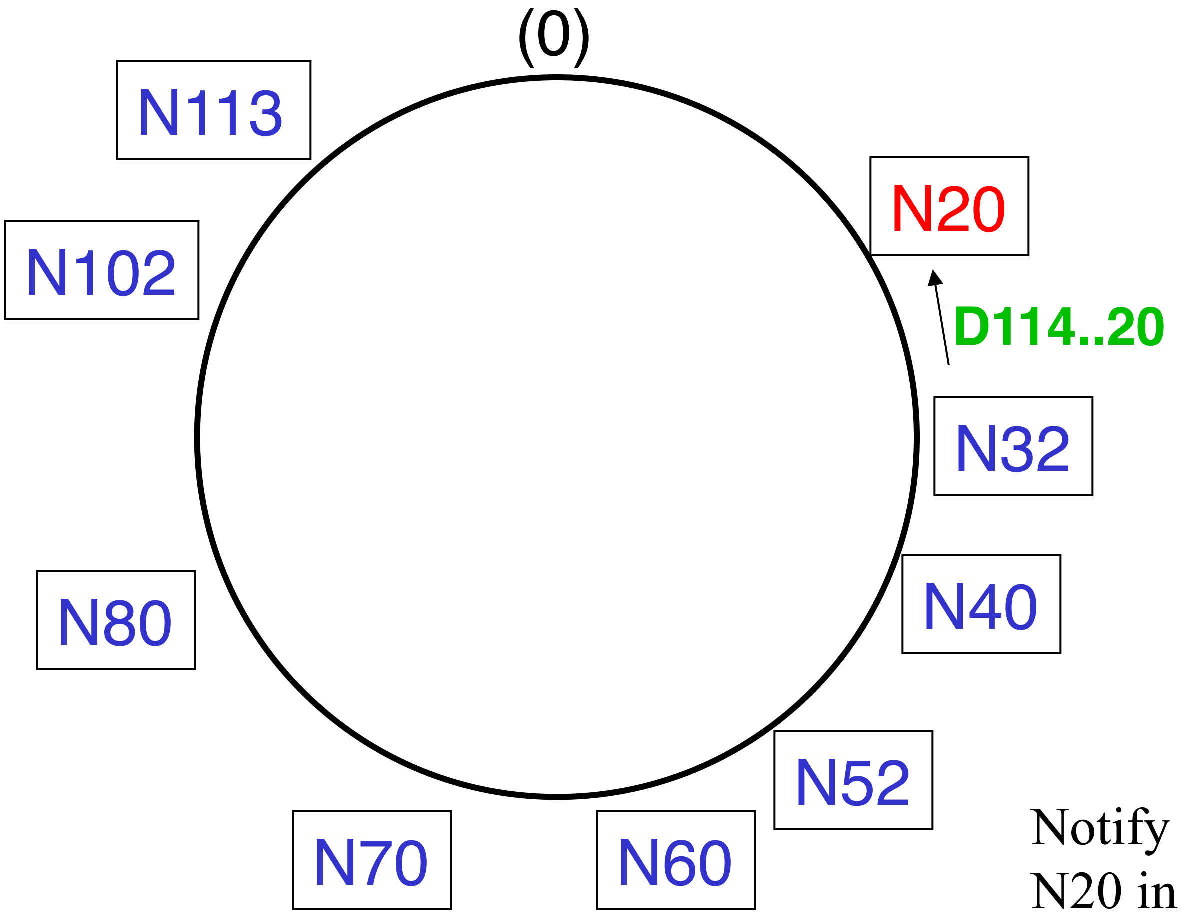
```
// ask n to find the successor of id
if      id = (n, successor]
        then   return successor
        else   n'= closest_ preceding_node (id)
               return n'.find_successor(id)
fi
// search for the highest predecessor of id
        n. closest_preceding_node(id)
        for i = log N downto 1
        if (finger[i] is between (n,id)
        return finger[i]
```

Example



N20 wants to find out the successor of key 65

After join move objects



N20's
Finger Table

21..21	N32
22..23	N32
24..27	N32
28..35	N32
36..51	N40
52..83	N52
84..19	N102

Notify nodes that must include N20 in their table. N113[1]=N20, not N32.

Node 20 moves documents from node 32.

Three steps in join

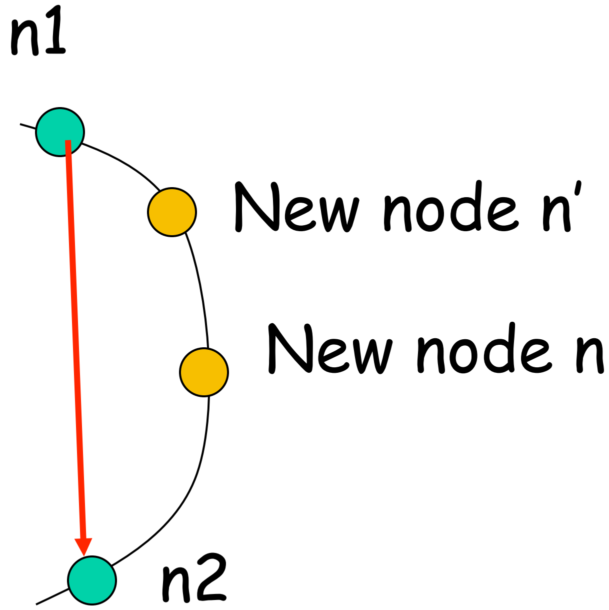
Step 1. Initialize predecessor and fingers of the new node.

(Knowledge of predecessor is useful in stabilization)

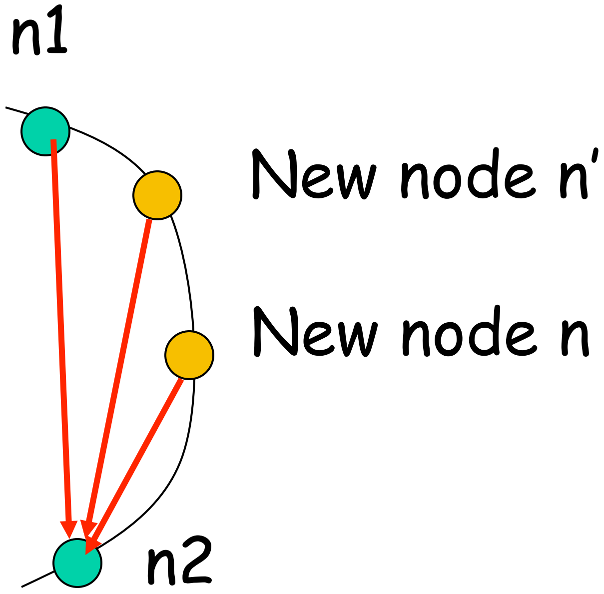
Step 2. Update the predecessor and the fingers of the existing nodes. (Thus notify nodes that must include N20 in their table. N113[1] = N20, not N32.

Step 3. Transfer objects to the new node as appropriate.

Concurrent Join



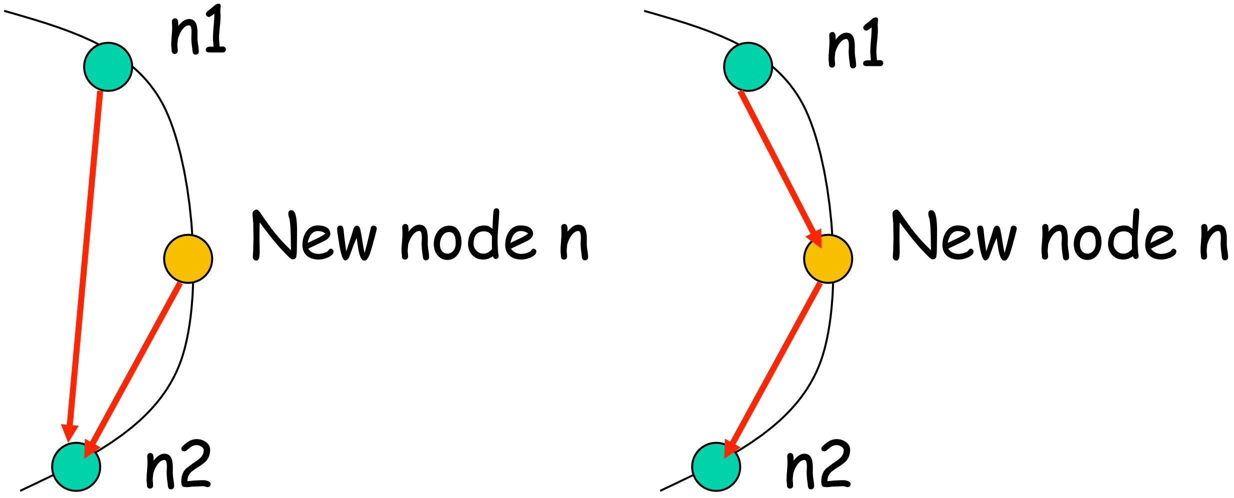
[Before]



[After]

Stabilization

Periodic stabilization is needed to integrate the new node into the network and restore the invariant.



Predecessor.successor(n1) ≠ n1, so n1 adopts predecessor.successor(n1) = n as its new successor

The complexity of join

With high probability, any node joining or leaving an N-*node Chord network will use* O($\log^2$N) *messages to re-establish the Chord routing invariants and finger tables*.

Chord Summary

- Log(n) lookup messages and table space.
- Well-defined location for each ID.
  - No search required.
- Natural load balance.
- No name structure imposed.
- Minimal join/leave disruption.