Distributed Systems

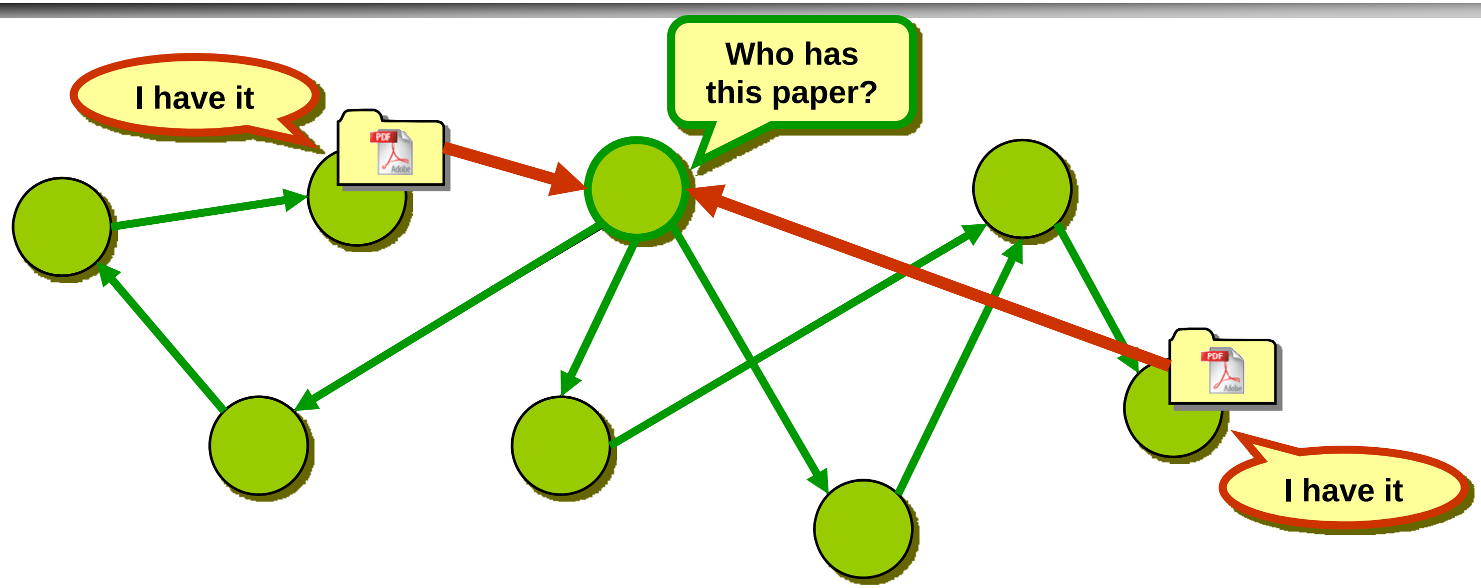
Distributed Hash Tables

Dynamic and Distributed
Information Systems

Today's Agenda

- What are DHTs?
  - Why are they useful?
- Pastry
- Chord

P2P challenge: Locating content



- Simple strategy: flood (e.g., expanding ring) until content is found
  - If ***R*** of ***N*** nodes have a replica, the expected search cost is at least ***N/R***, i.e., ***O(N)***
  - Need many replicas to keep overhead small
- Other strategy: centralized index (Napster)
  - Single point of failure, high load
- Goal: Decentralize the index!

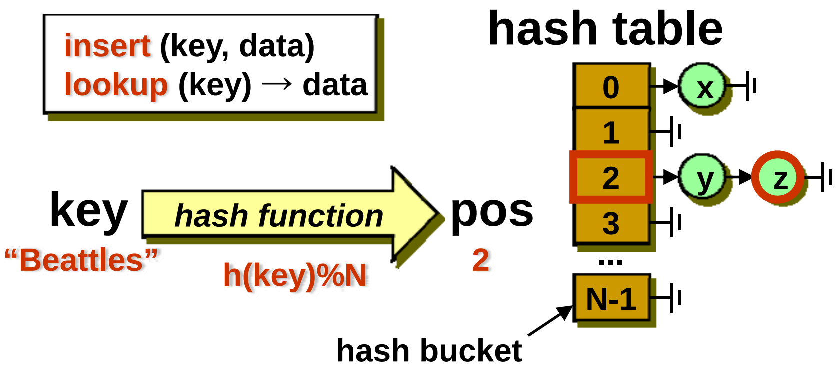
Indexed Search

- Idea
  - Store particular content on particular nodes
    - alternatively: *pointers to content*
  - When a node wants this content, go to the node that is supposed to hold it (or knows where it is)

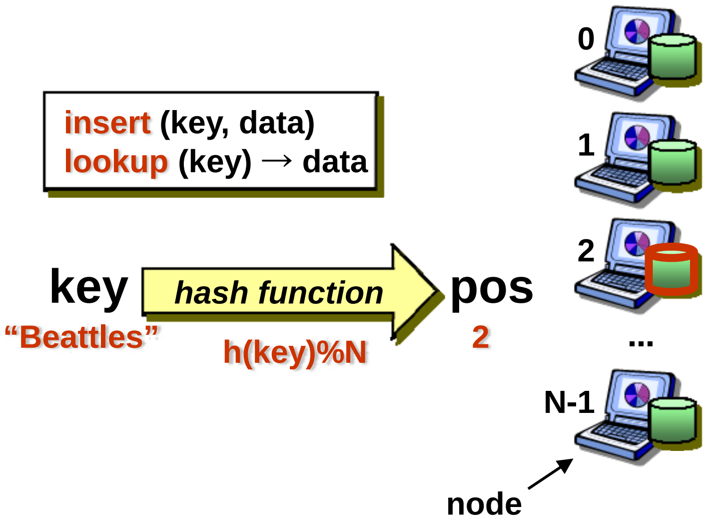
- Challenges
  - Avoid bottlenecks:
    Distribute the responsibilities "evenly" among the existing nodes
  - Self-organization w.r.t. nodes joining or leaving (or failing)
    - Give responsibilities to joining nodes
    - Redistribute responsibilities from leaving nodes
  - Fault-tolerance and robustness
    - Operate correctly also under failures

Idea: Hash Tables

- In a classic Hash Table:
  - Table has *N* buckets
  - Each data item has a key
  - Key is hashed to find bucket in hash table
  - Each bucket is expected to hold 1/N of the items, so storage is balanced

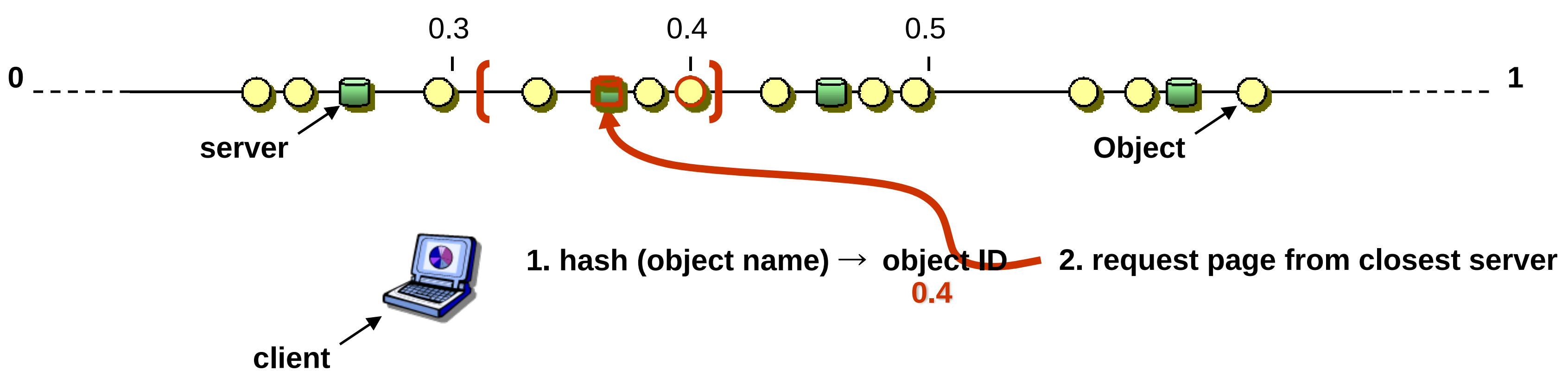


- In a Distributed Hash Table (DHT), nodes are the buckets
  - Network has *N* nodes
  - Each data item has a key
  - Key is hashed to find peer responsible for it
  - Each node is expected to hold 1/N of the items, so storage is balanced
  - <u>Additional requirement</u>: Also balance routing load!!



DHT Hashing

- Based on consistent hashing (designed for Web caching)
  - Each server is identified by an ID uniformly distributed in range [0, 1]
  - Each object maps (via some hash function) to an ID which is uniformly distributed in [0, 1]
- When looking up an object, we hash its ID, and get it from the appropriate server
  - Good load balancing: each server covers roughly equal intervals and stores roughly the same number of objects
  - Adding or removing a server invalidates few keys



What Makes a Good DHT Design?

- Should be able to route to any node in a few hops (small diameter)
  - Different DHTs differ fundamentally only in the routing approach
- DHT routing mechanisms should be decentralized (no single point of failure or bottleneck)
- The number of neighbors for each node should remain "reasonable" (small degree)
- To achieve good performance, DHTs must provide low stretch
  - Minimize ratio of DHT routing vs. IP latency
- Should gracefully handle nodes joining and leaving
  - Reorganize the neighbor sets
  - Bootstrap mechanisms to connect new nodes into the DHT
  - Repartition the affected keys over existing nodes

DHT Interface

- Minimal interface (data-centric)
  Lookup(key) → IP address
- Generality: Supports a wide range of applications
  - Keys have no semantic meaning
  - Values are application dependent
- DHTs do not store the data
  - Data storage can be built on top of DHTs
    Lookup(key) → data
    Insert(key, data)

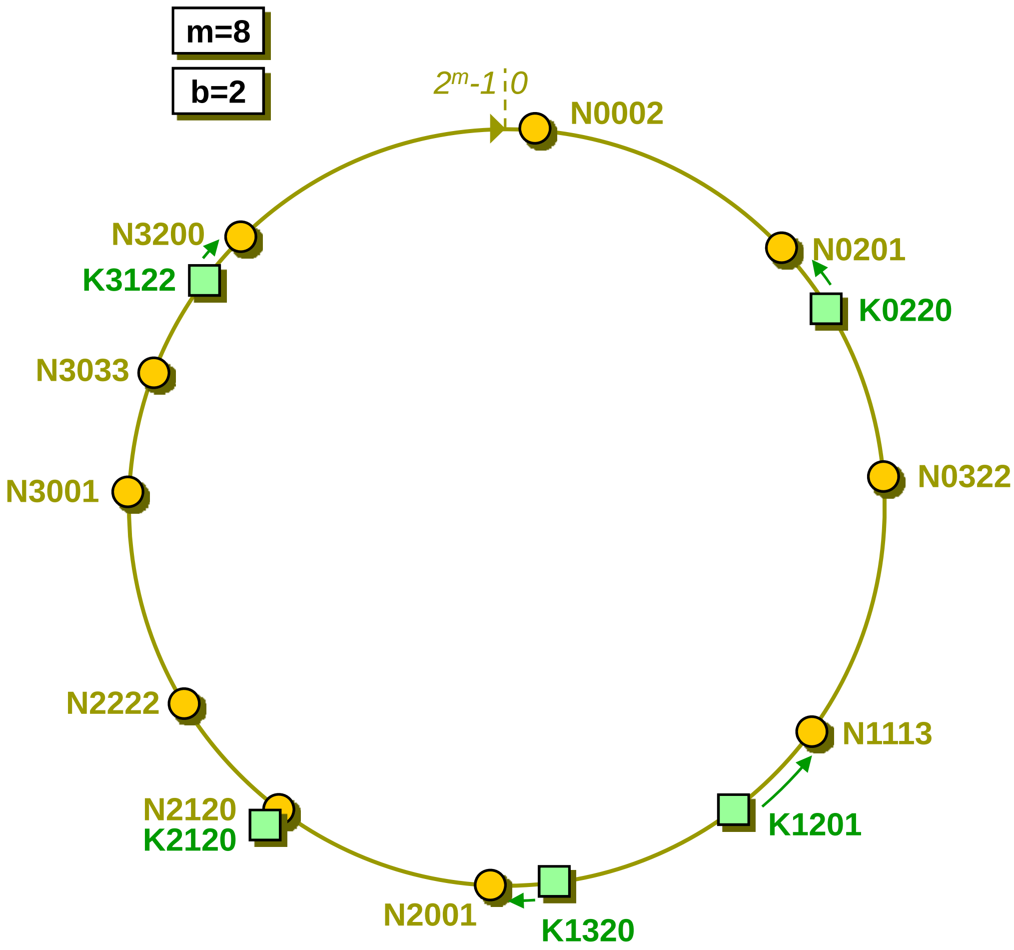
Application spectrum

- DHTs support many applications:
  - Network storage [CFS, OceanStore, PAST, …]
  - Web cache [Squirrel, …]
  - E-mail [e-POST, …]
  - Query and indexing [Kademlia, …]
  - Event notification [Scribe]
  - Application-layer multicast [SplitStream, …]
  - Naming systems [ChordDNS, INS, …]
  - ...

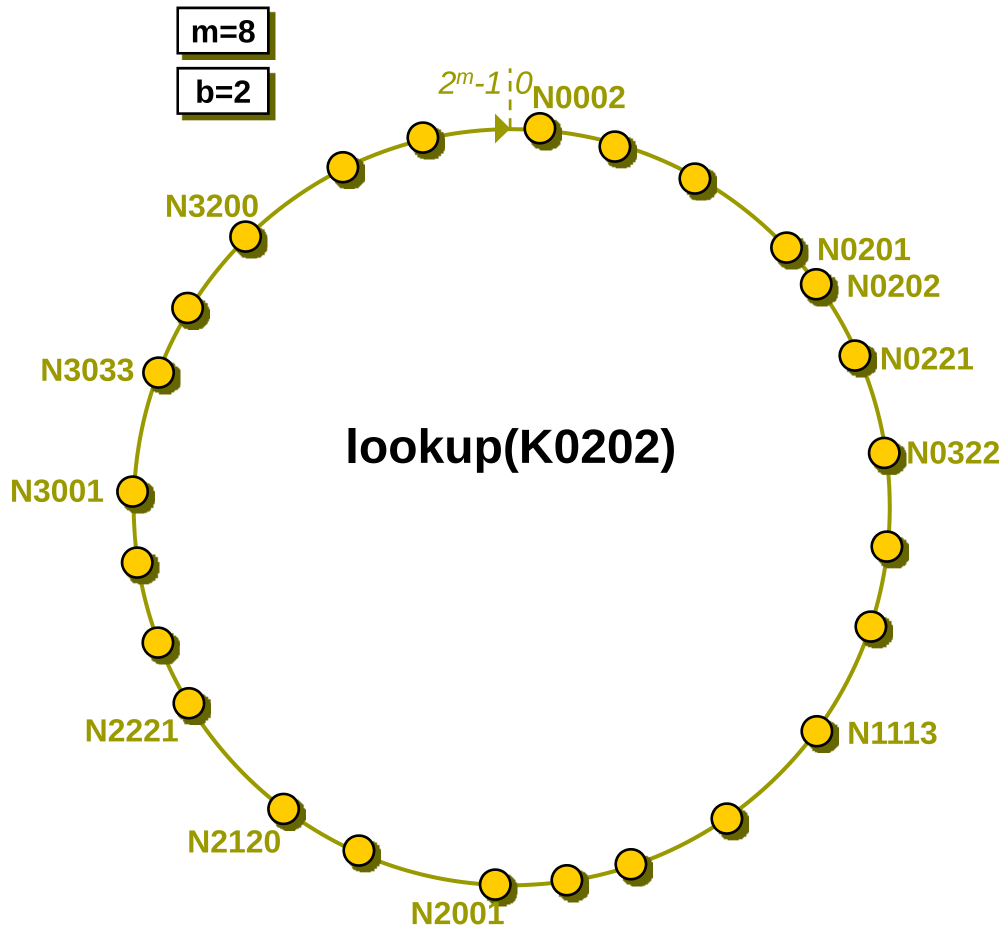
PASTRY (MSR + Rice)

Pastry

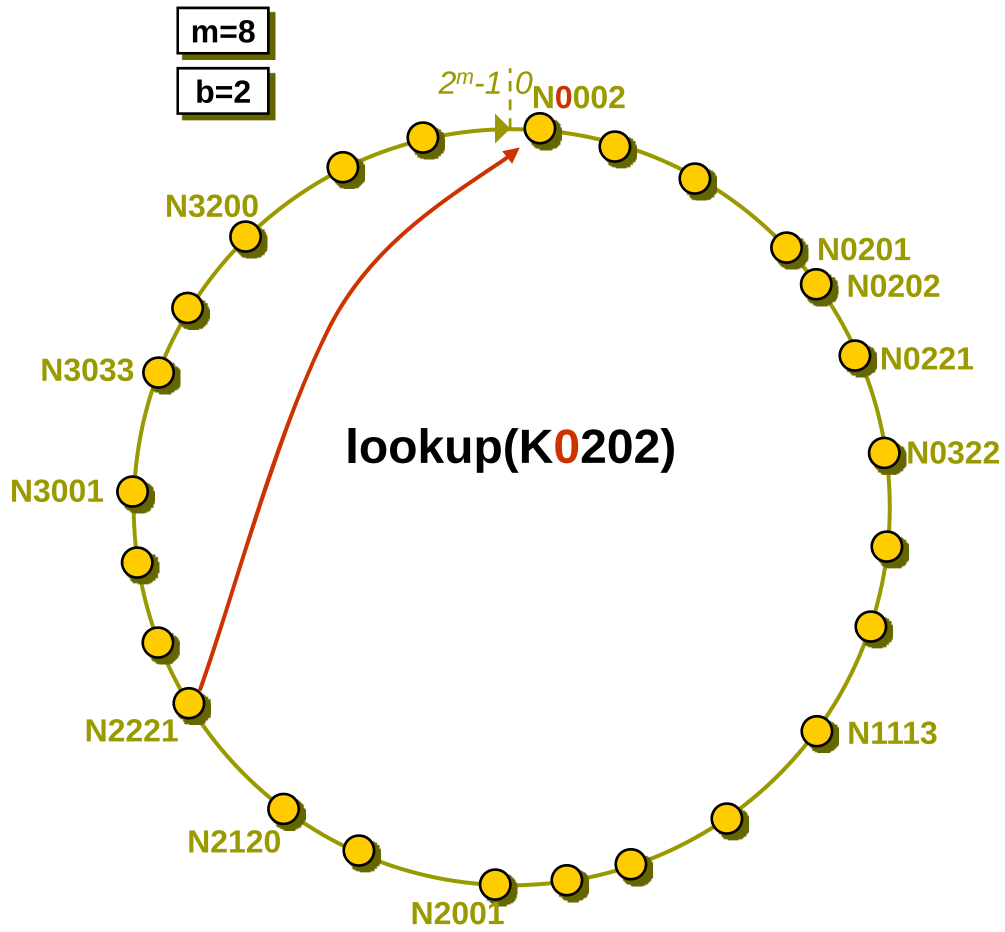
- Circular m-bit ID space for both keys and nodes
- Addresses in base 2^b with m/b digits
  - Address: m bits
  - Digit: b bits
  - ==> Address: m/b digits
- Node ID = SHA-1(IP address)
- Key ID = SHA-1(key)
- A key is mapped to the node whose ID is numerically-closest to the key ID



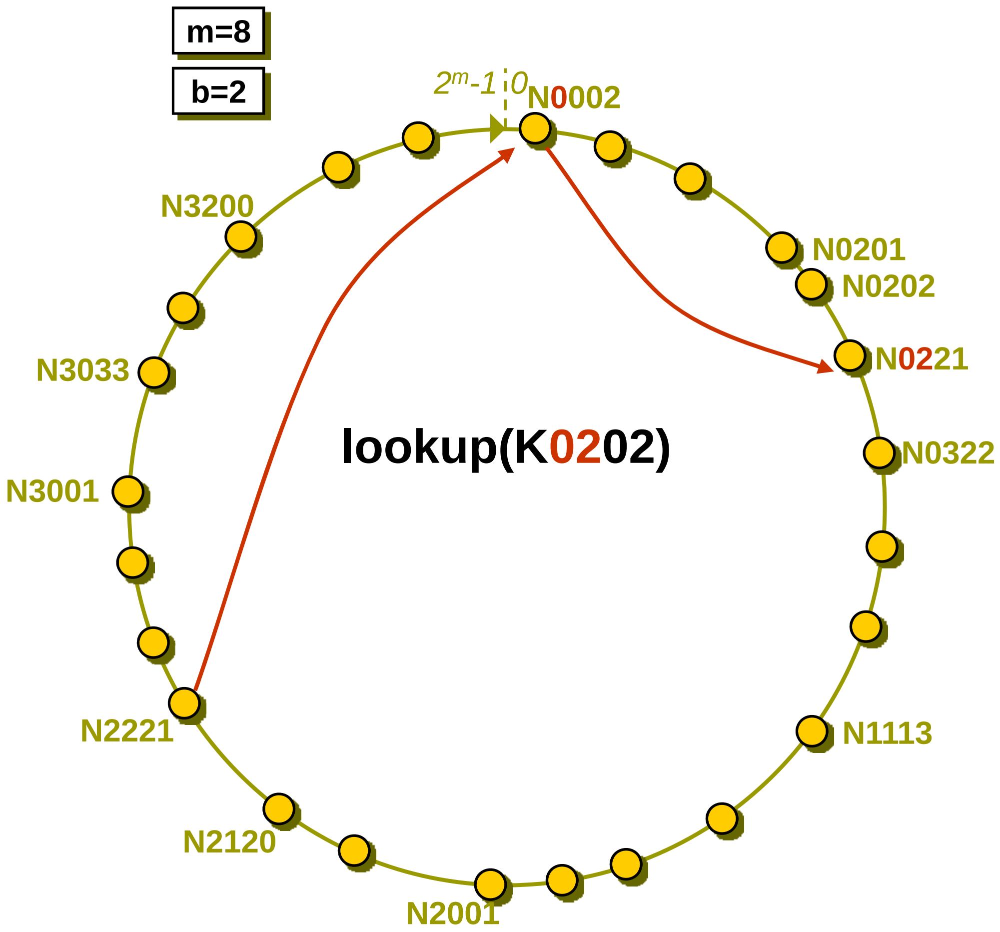
Pastry Routing



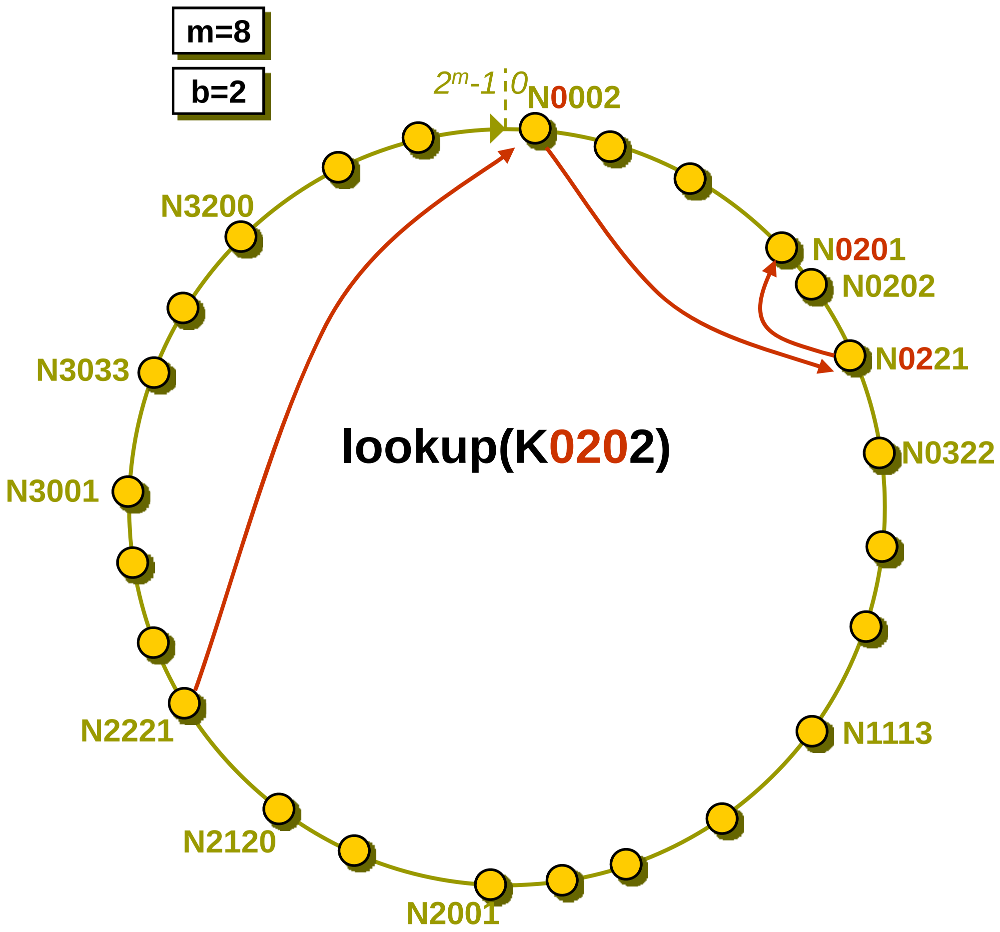
Pastry Routing



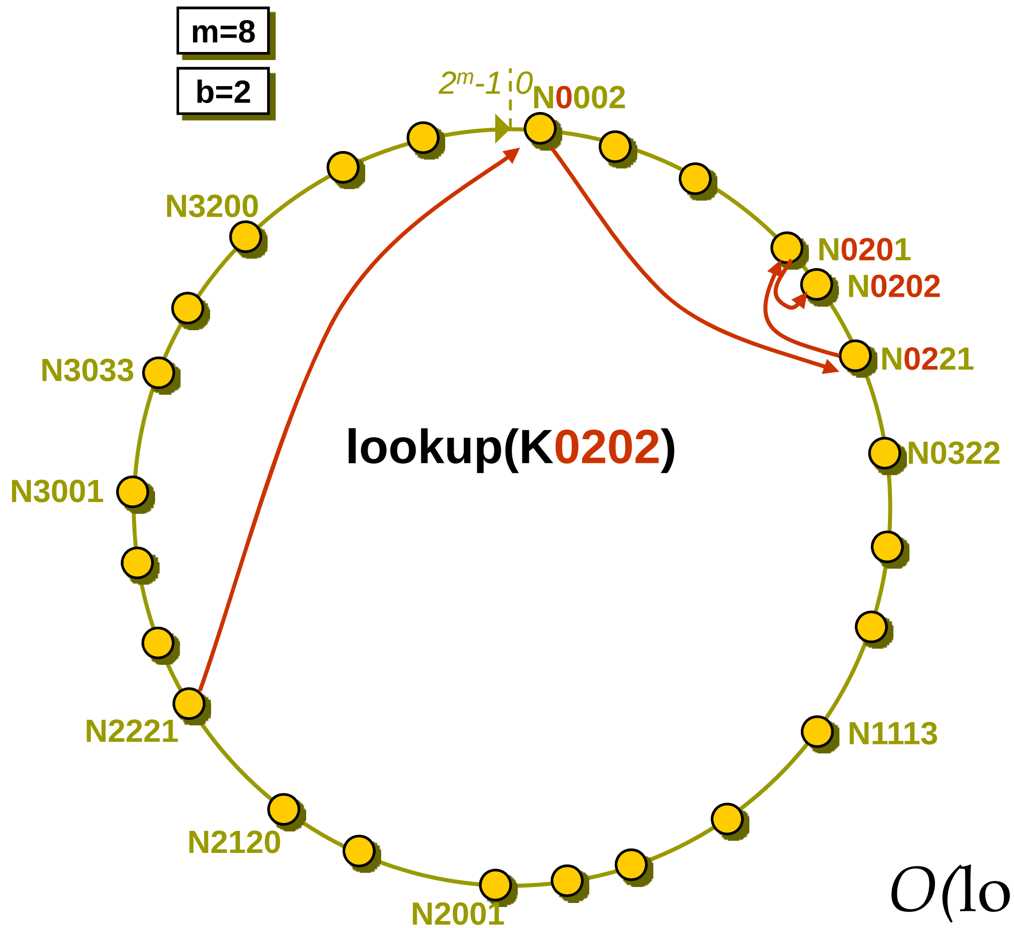
Pastry Routing



Pastry Routing



Pastry Routing



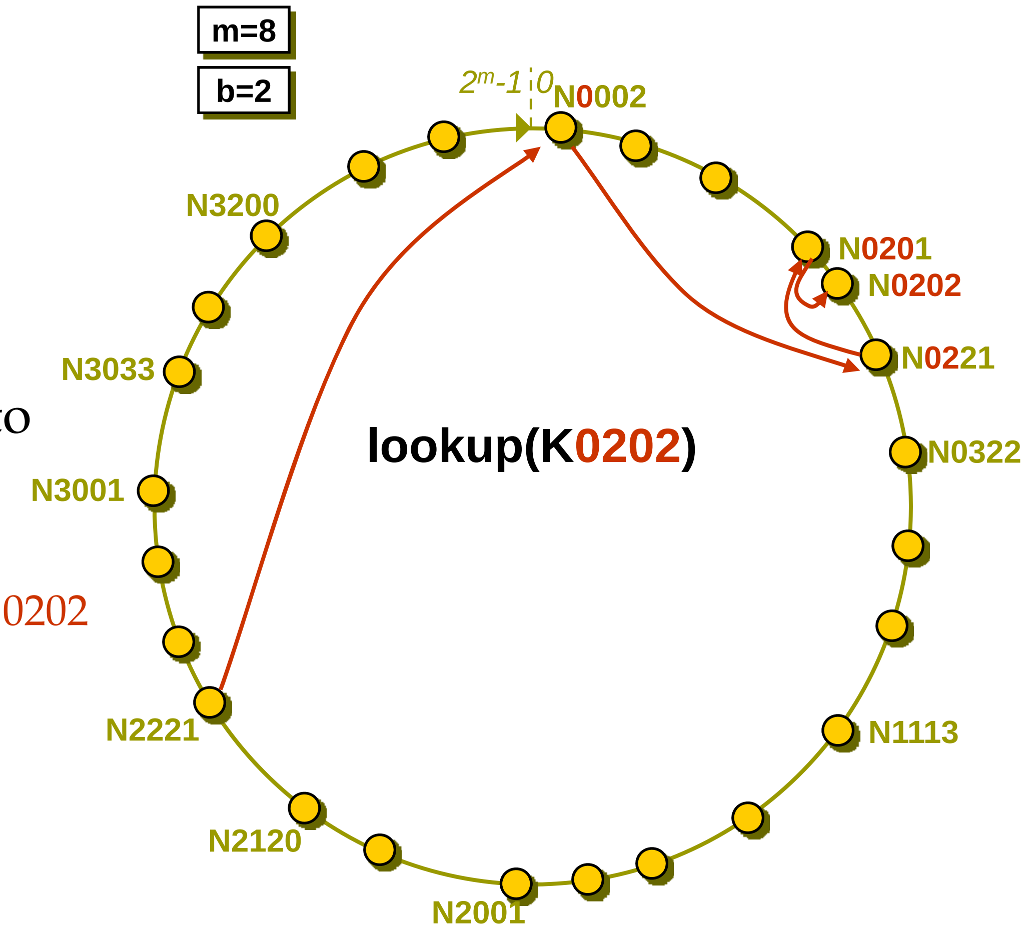
$O(\log N)$ hops

Pastry Routing

- O(log*N*) hops
- Route to 0202:

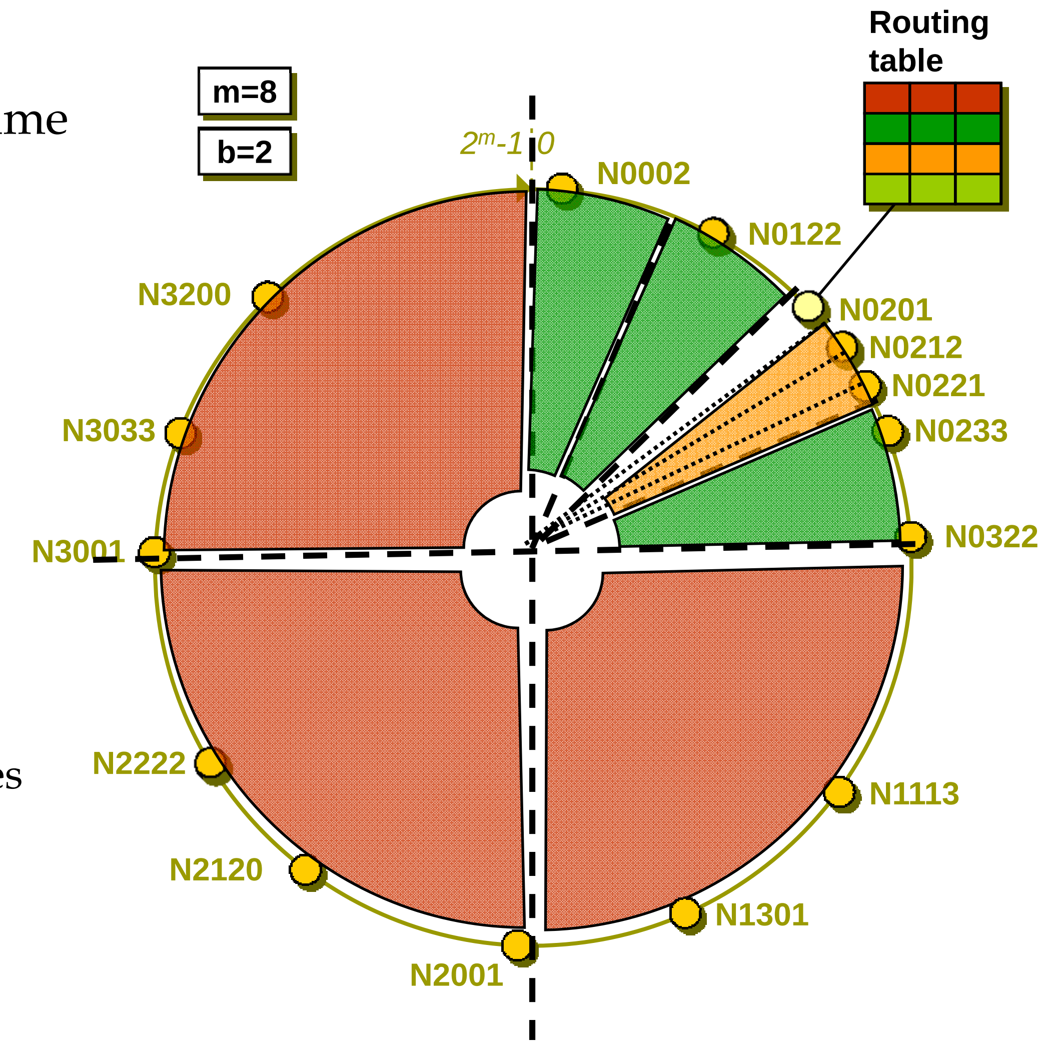
  2221 → 0002 → 0221 → 0201 → 0202
- If chain not complete, forward to numerically closest neighbor (successor)

  2221 → 0002 → 0221 → 0210 → 0201 → 0202



Pastry State and Lookup

- For each prefix, a node knows some other node (if any) with same prefix and different next digit
- For instance, N0201:
  - N-: N1???, N2???, N3???
  - N0: N00??, N01??, N03??
  - N02: N021?, N022?, N023?
  - N020: N0200, N0202, N0203
- When multiple nodes, choose topologically-closest
  - Maintain good locality properties (more on that later)



A Pastry Routing Table

m=16 b=2

b=2, so node ID is base 4 (16 bits)

Node ID 10233102

Contains the ***L*** nodes that are *numerically* closest to local node MUST BE UP TO DATE

Leaf set

< SMALLER		LARGER >	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing Table

m/b rows

02212102	1	22301203	31203203
0	11301233	12230203	13021022
10031203	10132102	2	10323302
10200230	10211302	10222302	3
10230322	10231000	10232121	3
10233001	1	10233232	
0		10233120	
		2	

2^b-*1* entries per row

Entries in the m^{th} column have *m* as next digit

n^{th} digit of current node

Entries in the n^{th} row share the first *n* digits with current node
[common-prefix next-digit rest]

Entries with no suitable node ID are left empty

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Contains the nodes that are closest to local node according to proximity metric

Pastry Lookup (Detailed)

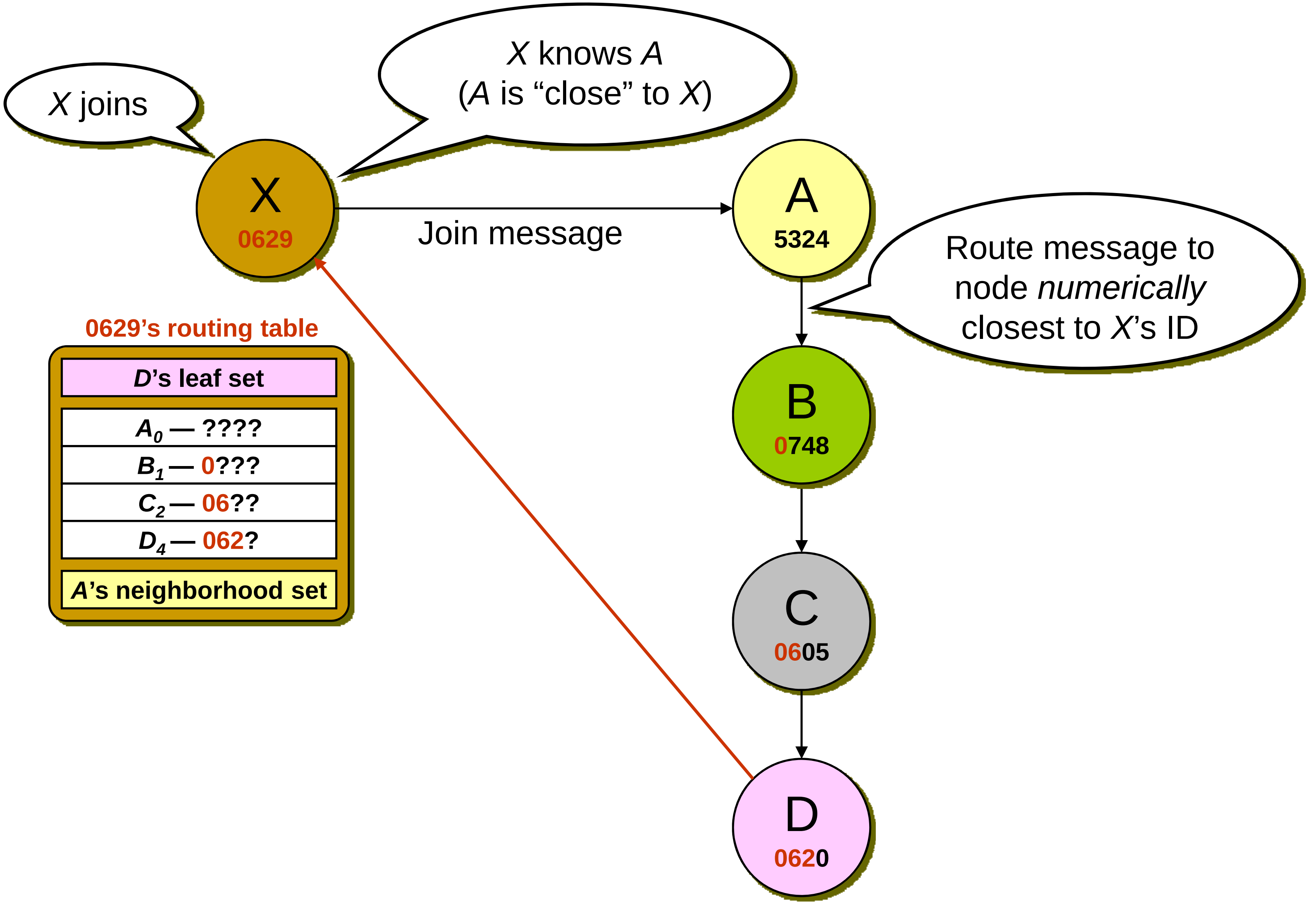
The routing procedure is executed whenever a message arrives at a node

1. IF (*key* in Leaf Set)
   1. If key is in leaf set, destination is 1 hop away, forward directly to destination.
2. ELSE IF (*key* in Routing Table)
   1. Forward to node that matches one more digit
3. ELSE
   1. Forward to a node numerically closer, from Leaf Set

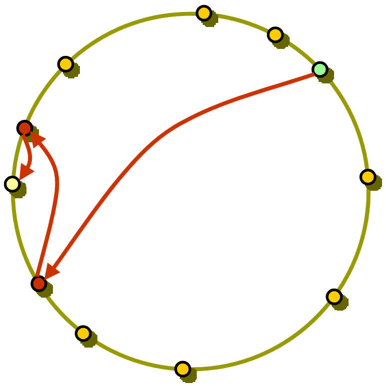
- **The procedure always converges!**

if $(L_{-\lfloor|L|/2\rfloor} \leq D \leq L_{\lfloor|L|/2\rfloor})$ {
    // D is within range of our leaf set
    forward to L_i, s.th. $|D - L_i|$ is minimal;
} else {
    // use the routing table
    Let $l = shl(D, A)$;
    if $(R_l^{D_l} \neq null)$ {
        forward to $R_l^{D_l}$;
    }
    else {
        // rare case
        forward to $T \in L \cup R \cup M$, s.th.
            $shl(T, D) \geq l$,
            $|T - D| < |A - D|$
    }
}

Joining

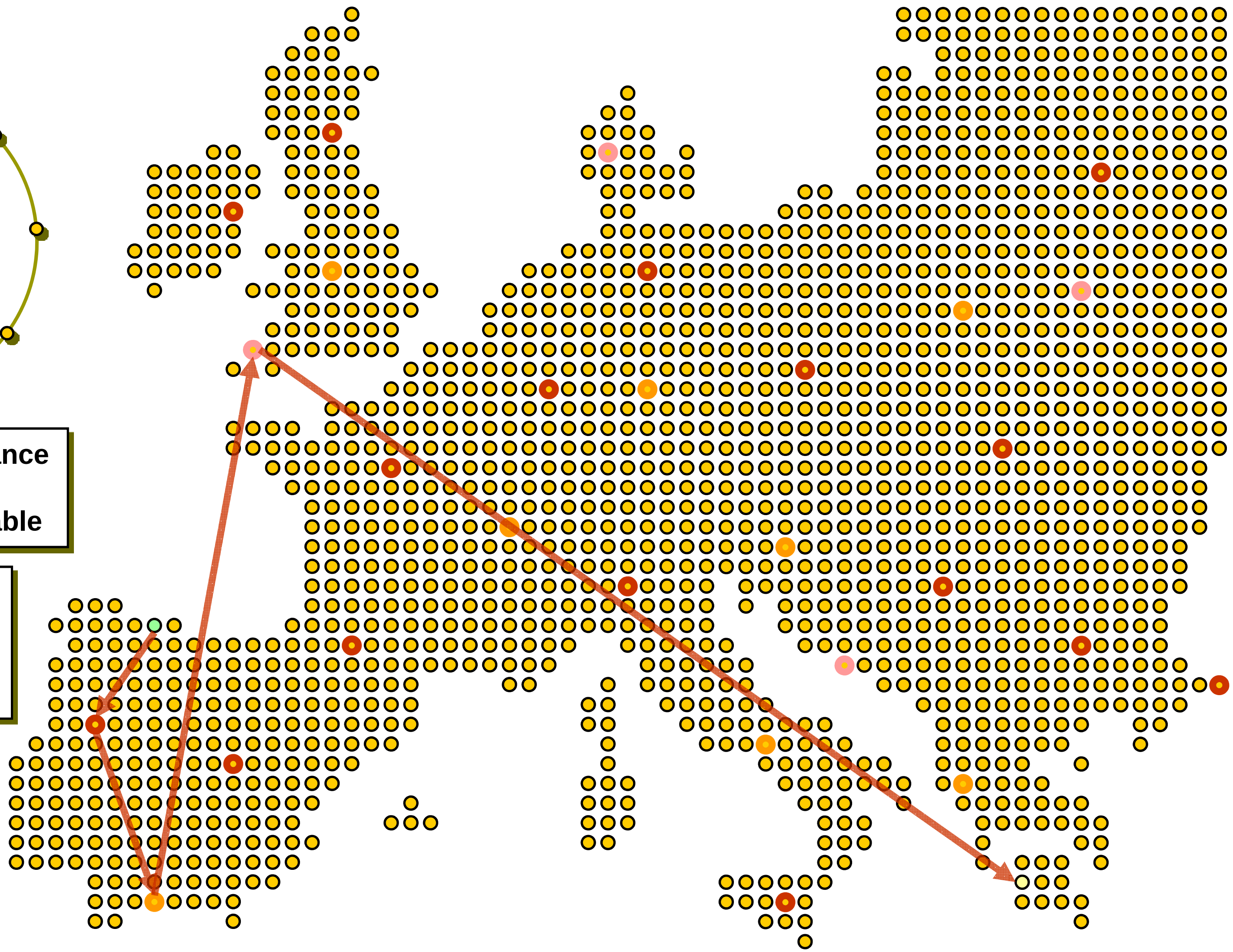

X joins
X knows A
(A is "close" to X)
X
0629
Join message
A
5324
Route message to
node numerically
closest to X's ID
0629's routing table
D's leaf set
A0 — ????
B1 — 0???
C2 — 06??
D4 — 062?
A's neighborhood set
B
0748
C
0605
D
0620

Pastry and Network Topology



Expected node distance increases with row number in routing table

Smaller and smaller **numerical** jumps
Bigger and bigger **topological** jumps

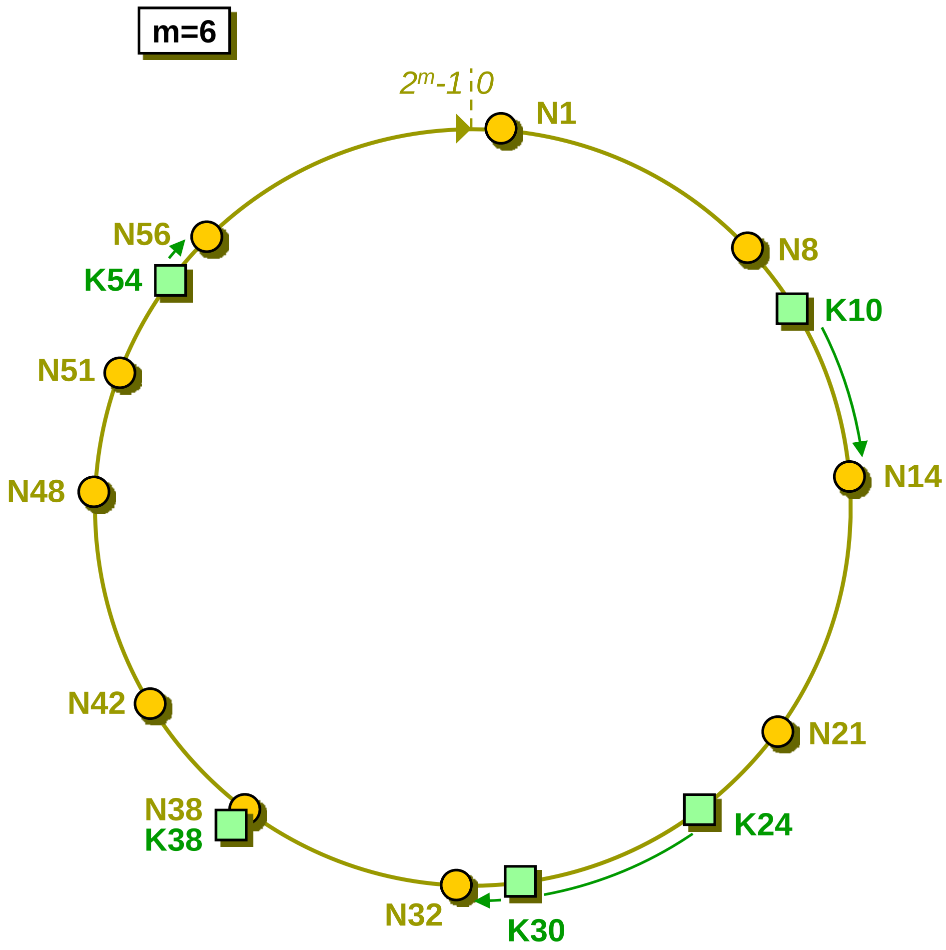


CHORD (MIT)

Dynamic and Distributed Information Systems

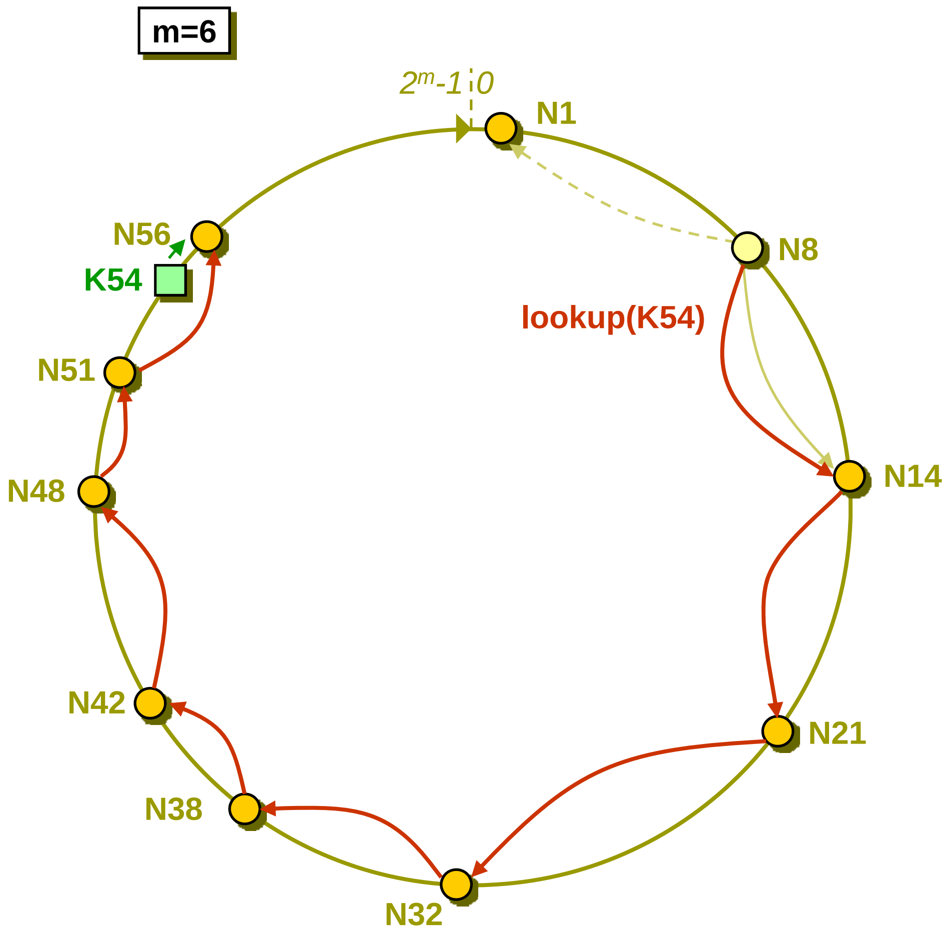
Chord (MIT)

- Circular *m*-bit ID space for both keys and node IDs
- Node ID = SHA-1(IP address)
- Key ID = SHA-1(key)
- Each key is mapped to its successor node
  - Node whose ID is equal to or follows the key ID
- Key distribution
  - Each node responsible for *O(K/N)* keys
  - *O(K/N)* keys move when a node joins or leaves



Basic Chord: State and Lookup

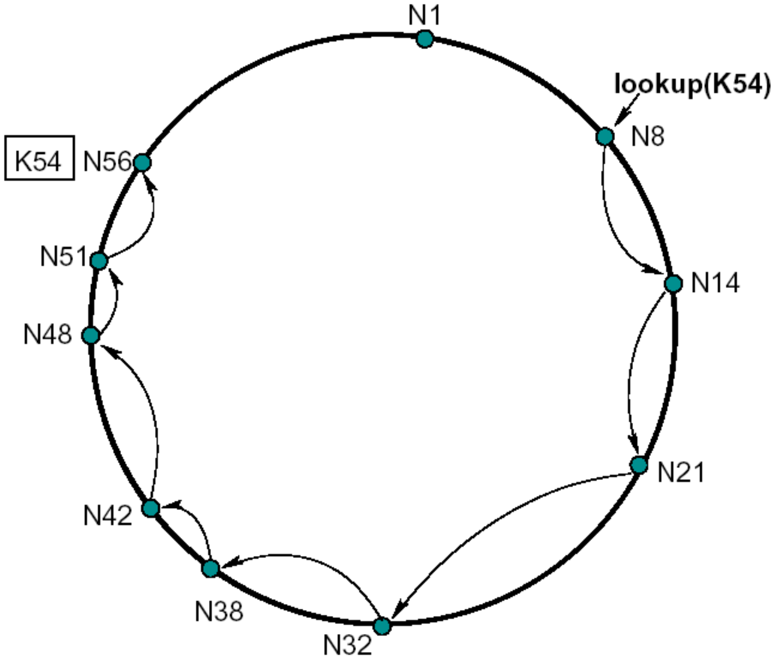
- Each node knows only two other nodes on the ring:
  - Successor
  - Predecessor (for ring management)
- Lookup is achieved by forwarding requests around the ring through successor pointers
  - Requires *O(N)* hops



Basic Chord: State and Lookup

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

(a)



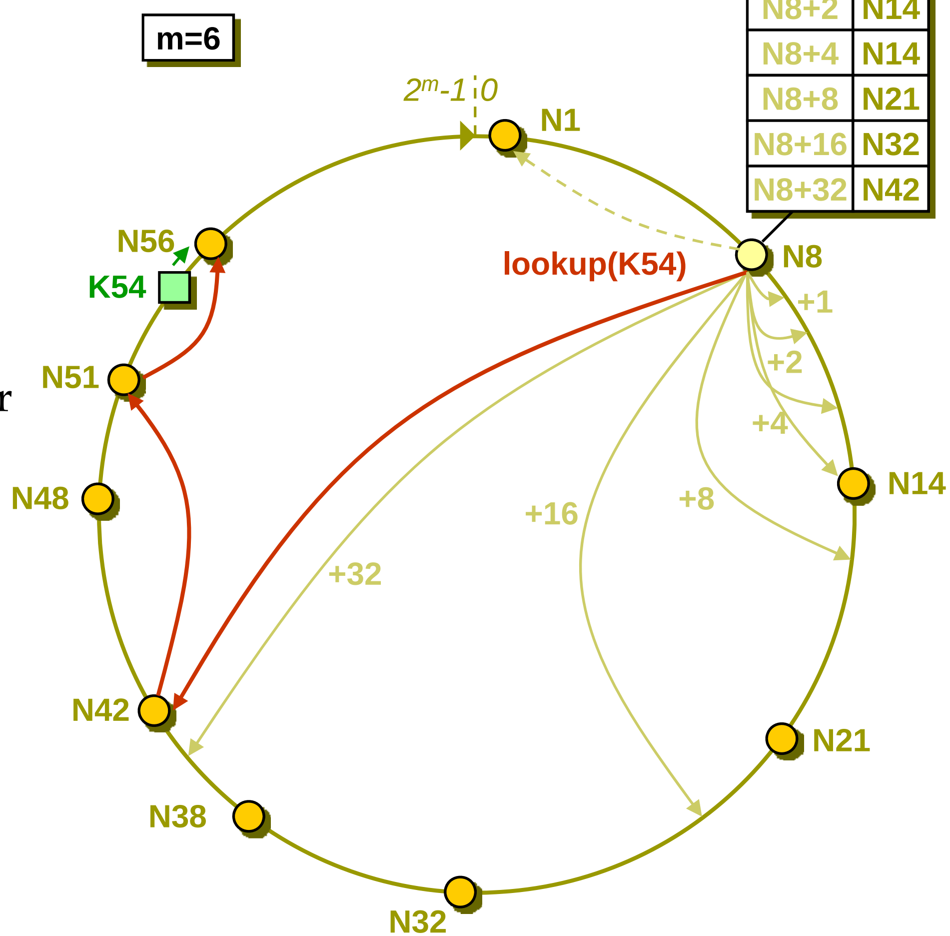
(b)

Complete Chord

- Each node knows these two nodes:
  - Successor
  - Predecessor (for ring management)
- But also: Each node has *m* fingers
  - *n*.finger(*i*) points to node on or after 2^i steps ahead
  - *n*.finger(0) == *n*.successor
  - *O(log N)* state per node
- Lookup is achieved by following *longest preceding fingers*, then the successor
  - *O(log N)* hops

Finger table

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42



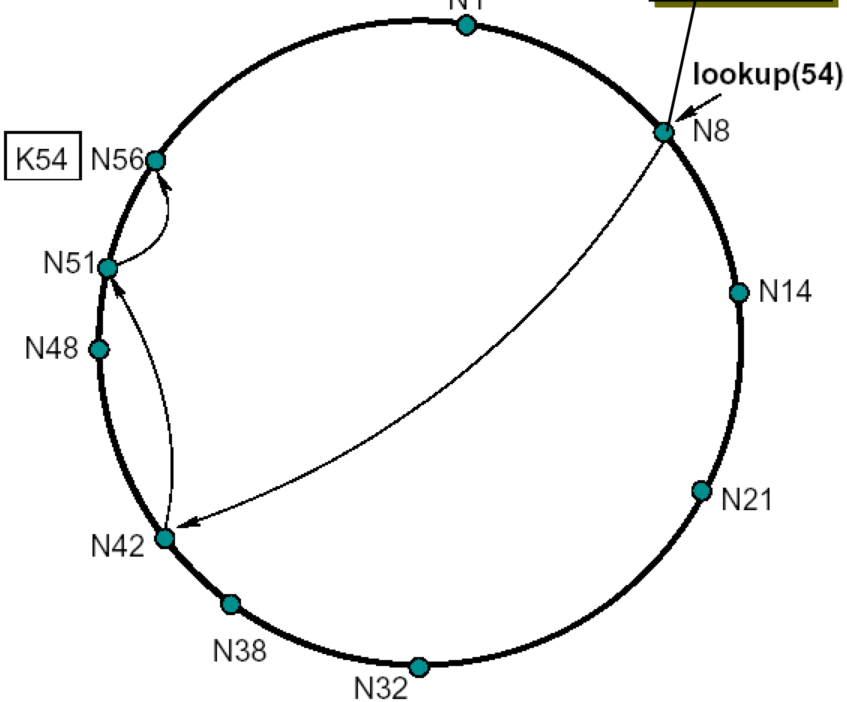
Complete Chord

Finger table

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

```
// ask node n to find the successor of id
n.find_successor(id)
    if (key ∈ (n, n.successor])
        return n.successor;
    else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);
```

```
// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];
    return n;
```



Chord Ring Management

- For correctness, Chord needs to maintain the following invariants
  - Successors are correctly maintained
  - For every key *k*, *succ(k)* is responsible for *k*
- Fingers are for efficiency, not necessarily correctness!
  - One can always default to successor-based lookup
  - Finger table can be updated lazily

Joining the Ring

- Three step process:

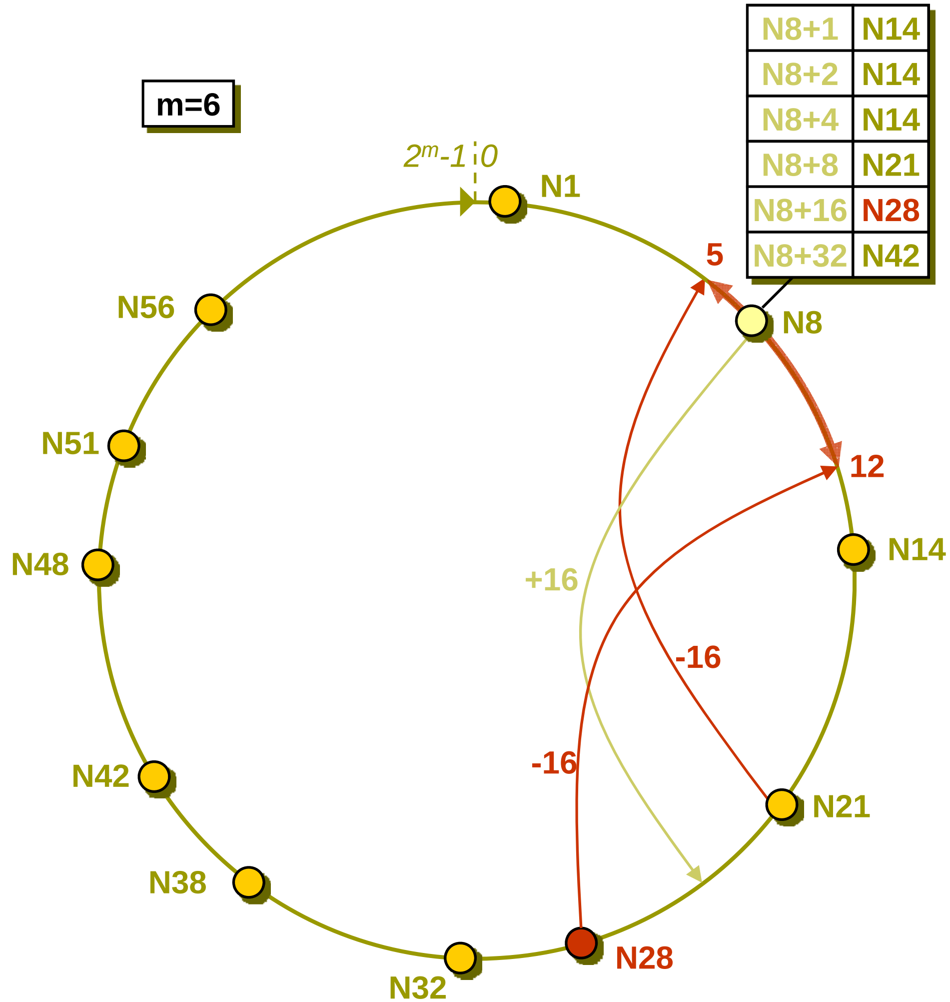
1. Outgoing links
   - Initialize predecessor and all fingers of new node
2. Incoming links
   - Update predecessors and fingers of existing nodes
3. Transfer some keys to the new node

Joining the Ring – Step 1

- Initialize the new node finger table
  - Locate any node *n* in the ring
  - Ask *n* to lookup the peers at $j+2^0$, $j+2^1$, $j+2^2$…
  - Use results to populate finger table of *j*

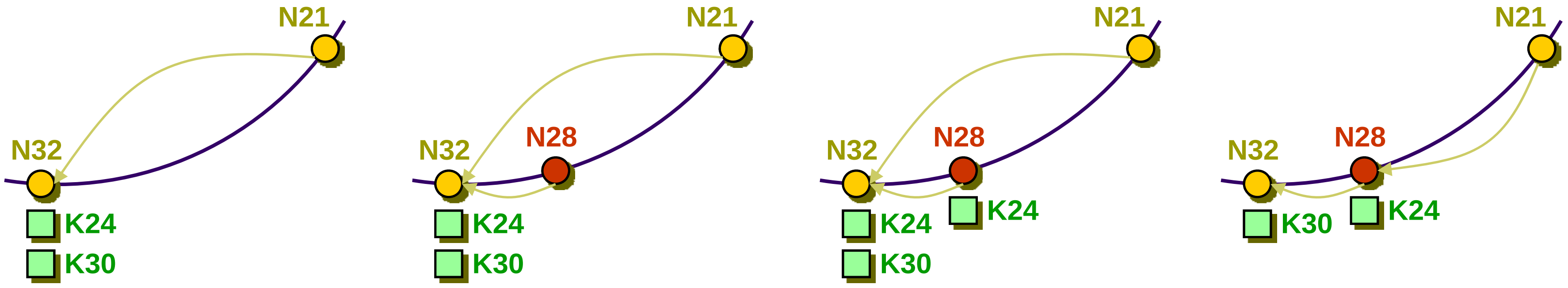
Joining the Ring – Step 2

- Updating fingers of existing nodes
  - New node *j* calls update function on existing nodes that must point to *j*
    - Nodes in the ranges $[j-2^i , pred(j)-2^i+1]$
  - *O(log N)* nodes need to be updated



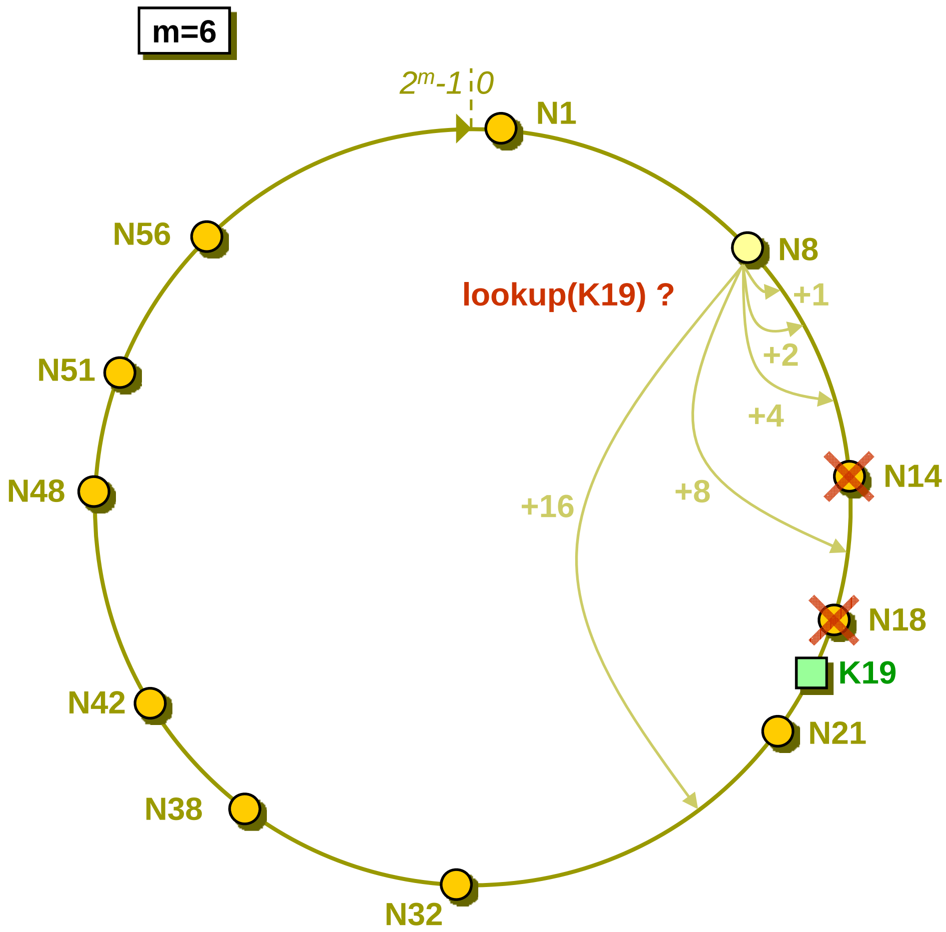
Joining the Ring – Step 3

- Transfer key responsibility
  - Connect to successor
  - Copy keys from successor to new node
  - Update successor pointer and remove keys



Leaving the Ring (or Failing)

- Node departures are treated as node failures
- Failure of nodes might cause incorrect lookup
  - N8 doesn't know correct successor, so lookup of K19 fails
- Solution: successor list
  - Each node *n* knows *r* immediate successors
  - After failure, *n* contacts first alive successor and updates successor list
  - Correct successors guarantee correct lookups

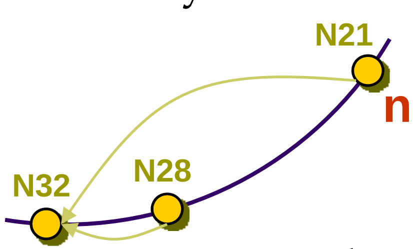


Leaving the Ring (or Failing)

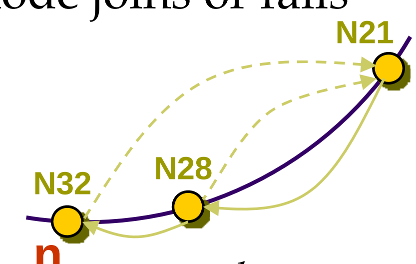
- Successor lists guarantee correct lookup with some probability
  - Can choose *r* to make probability of lookup failure arbitrarily small
- Assume half of the nodes fail and failures are independent
  - *P(n.successorList all dead)* = 0.5^r
  - *P(n does not break the Chord ring)* = $1 - 0.5^r$
  - *P(no broken nodes)* = $(1 - 0.5^r)^{N/2}$
    - $r = 2\log N$ makes probability = $1 - 1/N$
    - With high probability (*1-1/N*), the ring is not broken

Stabilization

- Case 1: finger tables are reasonably fresh
- Case 2: successor pointers are correct, not fingers
- Case 3: successor pointers are inaccurate or key migration is incomplete — MUST BE AVOIDED!
- Stabilization algorithm periodically verifies and refreshes node pointers (including fingers)
  - Eventually stabilizes the system when no node joins or fails

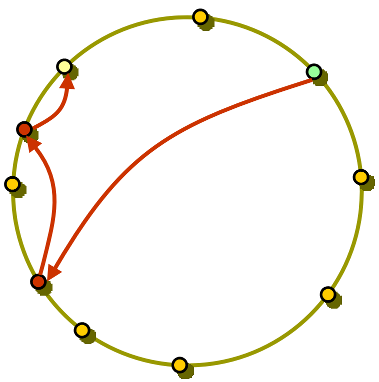


$x = n.succ.pred$
if x in $(n, n.succ)$
$n.succ = x$

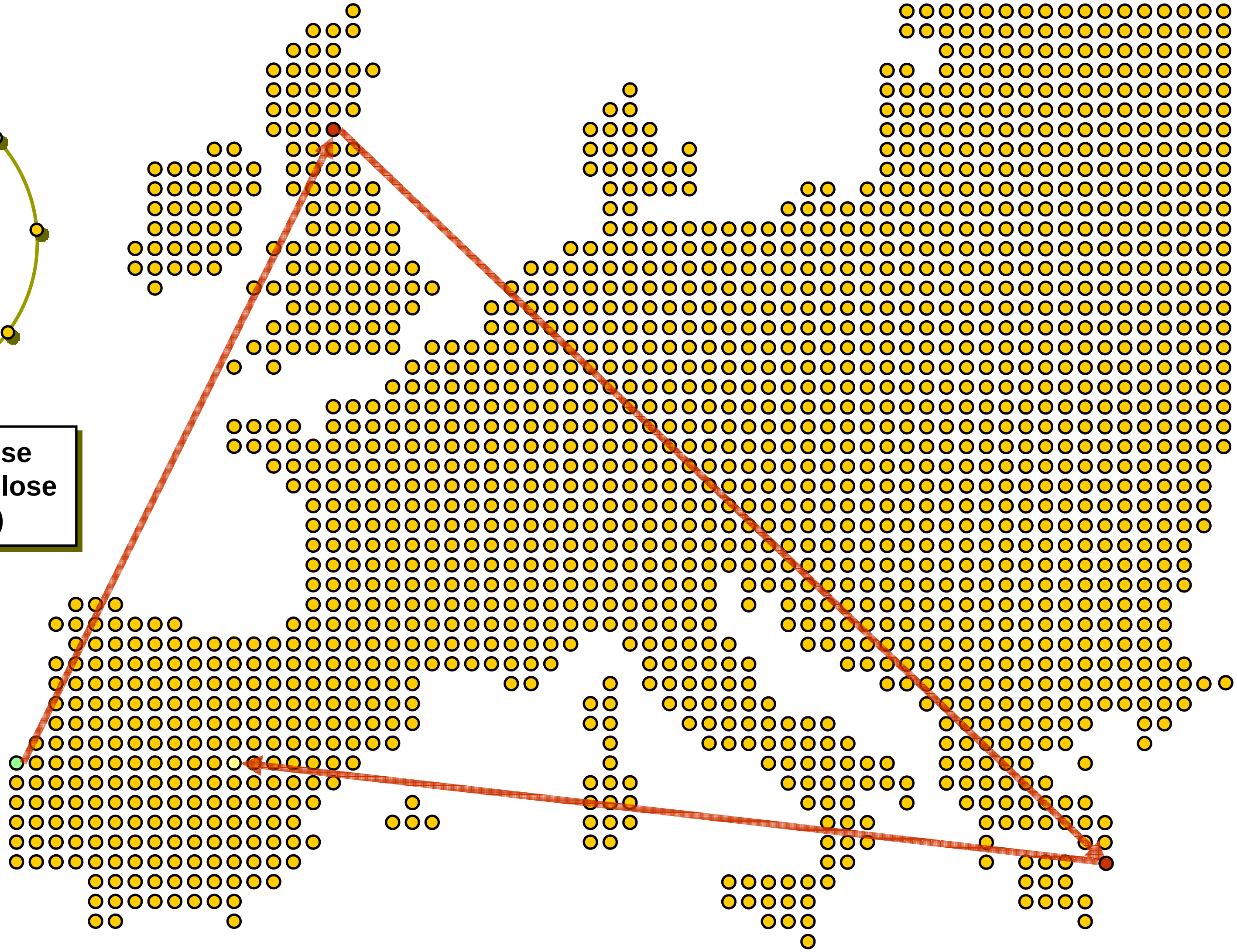


$x = n.pred.succ$
if x in $(n.pred, n)$
$n.pred = x$

Chord and Network Topology

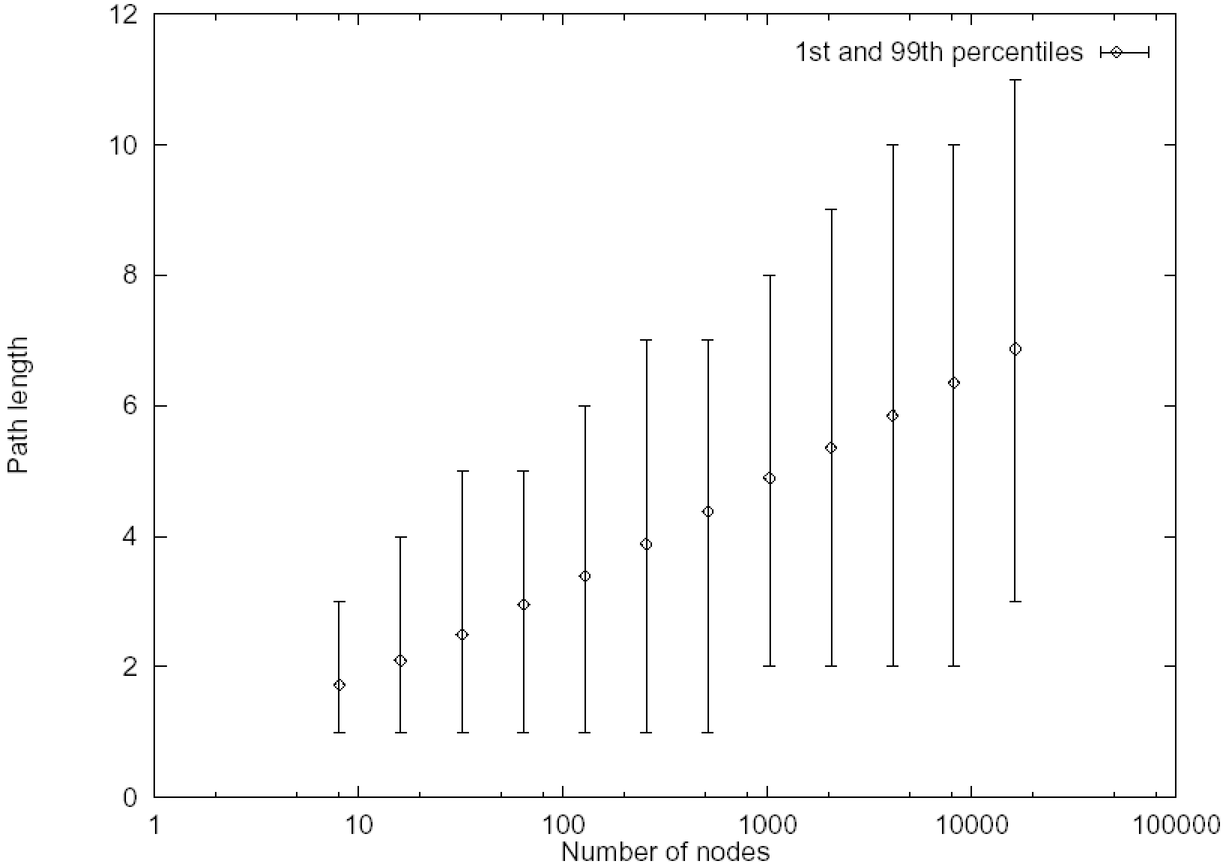


Nodes numerically-close are not topologically-close (1M nodes = 10+ hops)



Cost of Lookup

- Cost is *O(log N)*, constant is 0.5



Conclusions (1/2)

- Search types
  - Only equality
  - (How about ranges?)
- Scalability
  - Diameter (search and update) in $O(\log N)$ w.h.p.
  - Degree in $O(\log N)$
  - Construction: $O(\log^2 N)$ if a new node joins
- Robustness
  - Replication might be used by storing replicas at successor nodes

Conclusions (2/2)

- DHTs are a simple, yet powerful abstraction
  - Building block of many distributed services (file systems, application-layer multicast, distributed caches, etc.)
- Many DHT designs, with various pros and cons
  - Balance between state (degree), speed of lookup (diameter), and ease of management
- System must support rapid changes in membership
  - Dealing with joins/leaves/failures is not trivial
  - Dynamics of P2P networks are difficult to analyze
- Many open issues worth exploring