P2P Network
Structured Networks: Distributed Hash Tables

Pedro García López

Universitat Rovira I Virgili

Pedro.garcia@urv.net

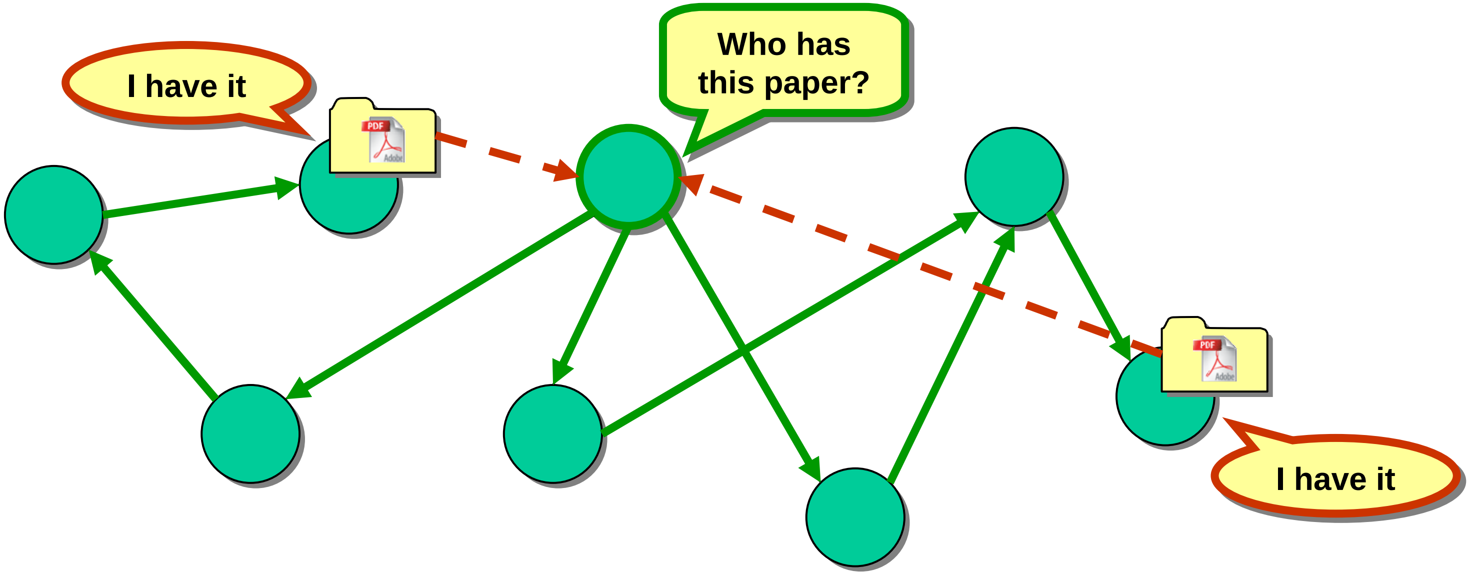
Departament d'Enginyeria Informàtica i Matemàtiques

Universitat Rovira i Virgili

Index

- Introduction to DHT’s
- Origins of structured overlays
- Case studies
  - Chord
  - Pastry
  - CAN
- Conclusions

Introduction to DHT's: locating contents



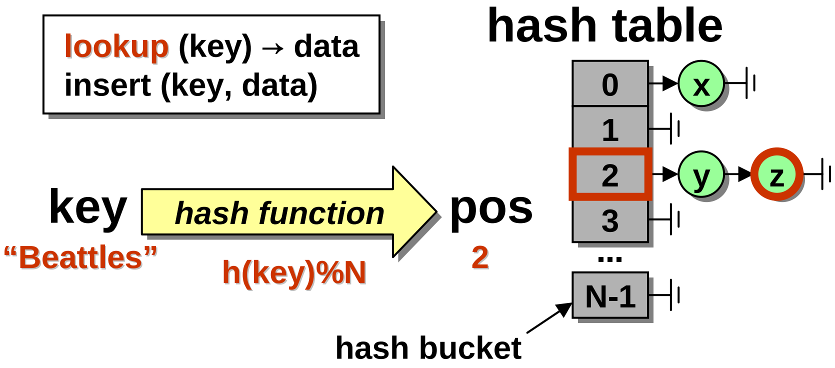
- Simple strategy: expanding ring search until content is found
- If *r* of *N* nodes have copy, the expected search cost is at least *N* / *r*, i.e., *O(N)*
- Need many copies to keep overhead small

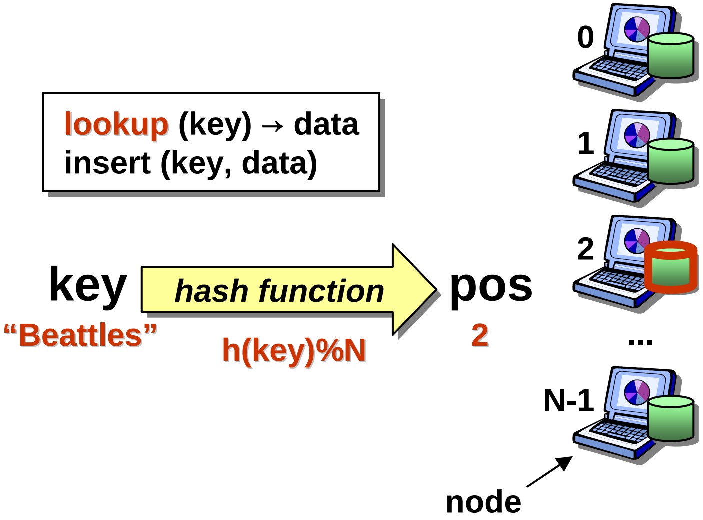
Directed Searches

- Idea
  - Assign particular nodes to hold particular content (or know where it is)
  - When a node wants this content, go to the node that is supposes to hold it (or know where it is)
- Challenges
  - Avoid bottlenecks: distribute the responsibilities “evenly” among the existing nodes
  - Adaptation to nodes joining or leaving (or failing)
    - Give responsibilities to joining nodes
    - Redistribute responsibilities from leaving nodes

Idea: Hash Tables

- A hash table associates data with keys
  - Key is hashed to find bucket in hash table
  - Each bucket is expected to hold #items/#buckets items
- In a Distributed Hash Table (DHT), nodes are the hash buckets
  - Key is hashed to find responsible peer node
  - Data and load are balanced across nodes





DHTs: Problems

- Problem 1 (dynamicity): adding or removing nodes
  - With hash mod N, virtually every key will change its location!

$$h(k) \bmod m \neq h(k) \bmod (m+1) \neq h(k) \bmod (m-1)$$

- Solution: use consistent hashing
  - Define a fixed hash space
  - All hash values fall within that space and do not depend on the number of peers (hash bucket)
  - Each key goes to peer closest to its ID in hash space (according to some proximity metric)

DHTs: Problems (cont'd)

- Problem 2 (size): all nodes must be known to insert or lookup data
  - Works with *small* and *static* server populations
- Solution: each peer knows of only a few "neighbors"
  - Messages are routed through neighbors via multiple hops (overlay routing)

What Makes a Good DHT Design

- For each object, the node(s) responsible for that object should be reachable via a “short” path (small diameter)
  - The different DHTs differ fundamentally only in the routing approach
- The number of neighbors for each node should remain “reasonable” (small degree)
- DHT routing mechanisms should be decentralized (no single point of failure or bottleneck)
- Should gracefully handle nodes joining and leaving
  - Repartition the affected keys over existing nodes
  - Reorganize the neighbor sets
  - Bootstrap mechanisms to connect new nodes into the DHT
- To achieve good performance, DHT must provide low stretch
  - Minimize ratio of DHT routing vs. unicast latency

DHT Interface

- Minimal interface (data-centric)

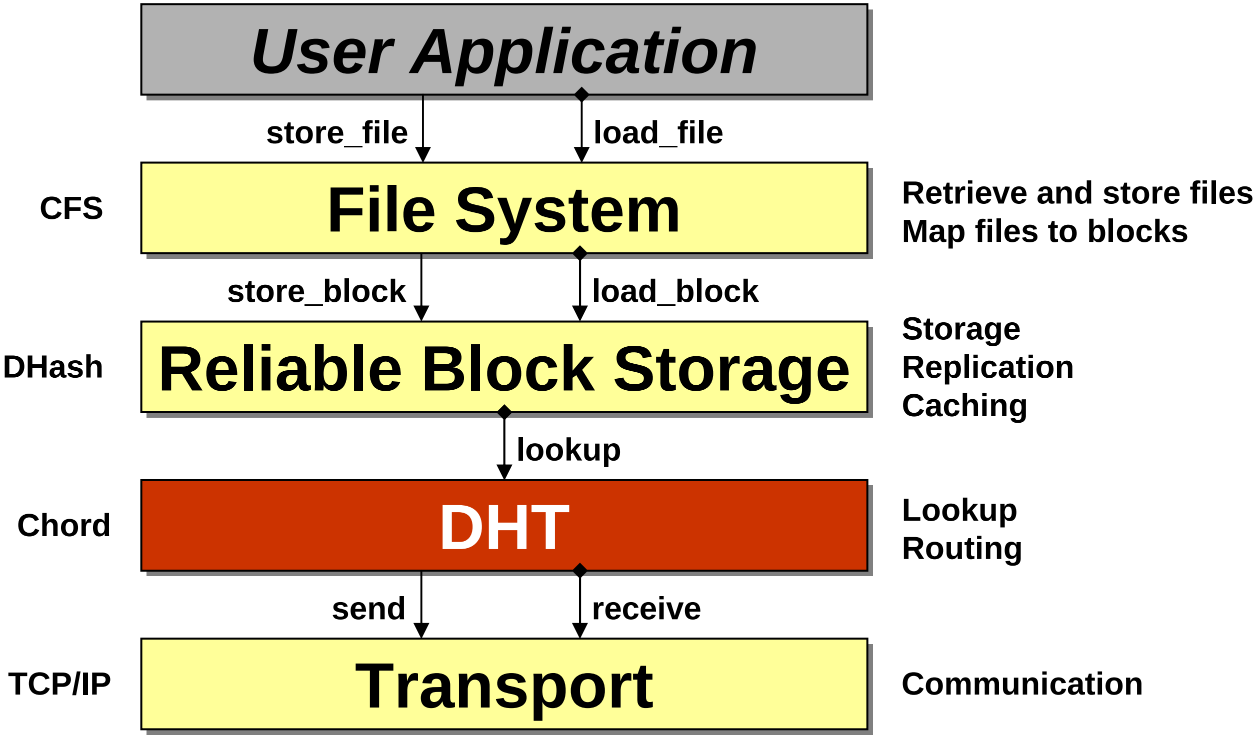
  Lookup(key) → IP address

- Supports a wide range of applications, because few restrictions
  - Keys have no semantic meaning
  - Value is application dependent
- DHTs do not store the data
  - Data storage can be build on top of DHTs

    Lookup(key) → data

    Insert(key, data)

DHTs in Context



User Application

store_file

load_file

CFS

File System

Retrieve and store files
Map files to blocks

store_block

load_block

DHash

Reliable Block Storage

Storage
Replication
Caching

lookup

Chord

DHT

Lookup
Routing

send

receive

TCP/IP

Transport

Communication

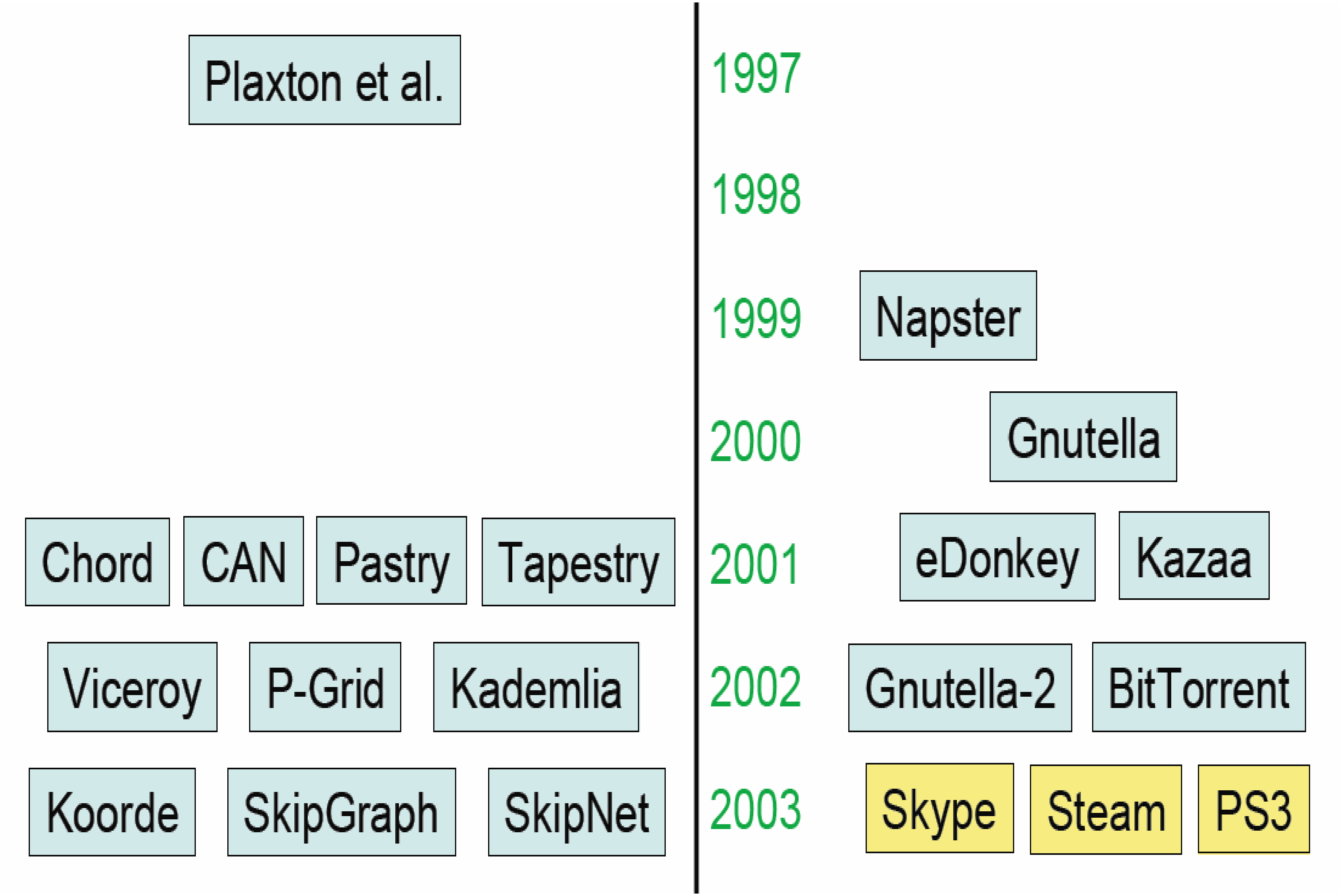
DHTs Support Many Applications

- File sharing [CFS, OceanStore, PAST, …]
- Web cache [Squirrel, …]
- Censor-resistant stores [Eternity, FreeNet, …]
- Application-layer multicast [Narada, …]
- Event notification [Scribe]
- Naming systems [ChordDNS, INS, …]
- Query and indexing [Kademlia, …]
- Communication primitives [I3, …]
- Backup store [HiveNet]
- Web archive [Herodotus]

Origins of Structured overlays

- Accessing Nearby Copies of Replicated Objects in a Distributed Environment“, by Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa, at SPAA 1997
- The paper proposes an efficient search routine (similar to the evangelist papers). In particular search, insert, delete, storage costs are all logarithmic, the base of the logarithm is a parameter.
- Prefix routing, distance and coordinates !
- Theory paper

Evolution


Plaxton et al.
1997
1998
1999
Napster
2000
Gnutella
Chord
CAN
Pastry
Tapestry
2001
eDonkey
Kazaa
Viceroy
P-Grid
Kademlia
2002
Gnutella-2
BitTorrent
Koorde
SkipGraph
SkipNet
2003
Skype
Steam
PS3

Hypercubic topologies

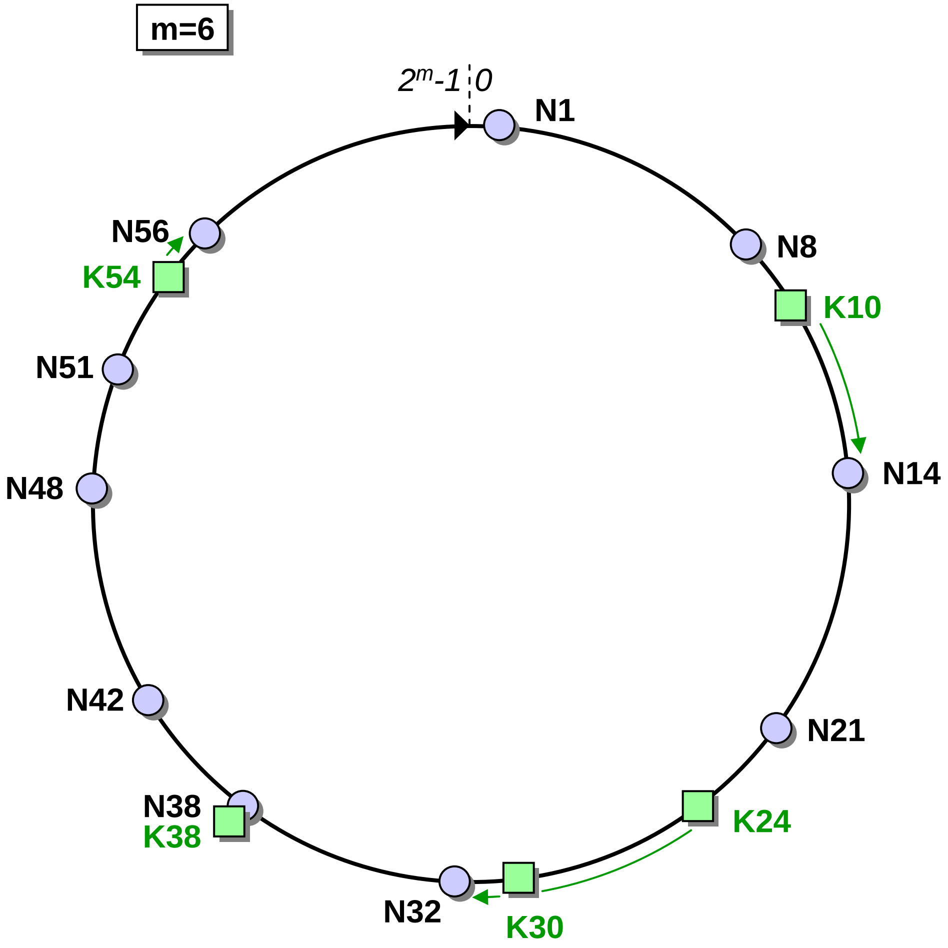
- Hypercube:
  - Plaxton, Chord, Kademlia, Pastry, Tapestry
- Butterfly /Benes:
  - Viceroy, Mariposa
- De Bruijn Graph:
  - Koorde
- Skip List:
  - Skip Graph, SkipNet
- Pancake Graph
- Cube Connected Cycles

DHT Case Studies

- Case Studies
  - Chord
  - Pastry
  - CAN
- Questions
  - How is the hash space divided evenly among nodes?
  - How do we locate a node?
  - How does we maintain routing tables?
  - How does we cope with (rapid) changes in membership?

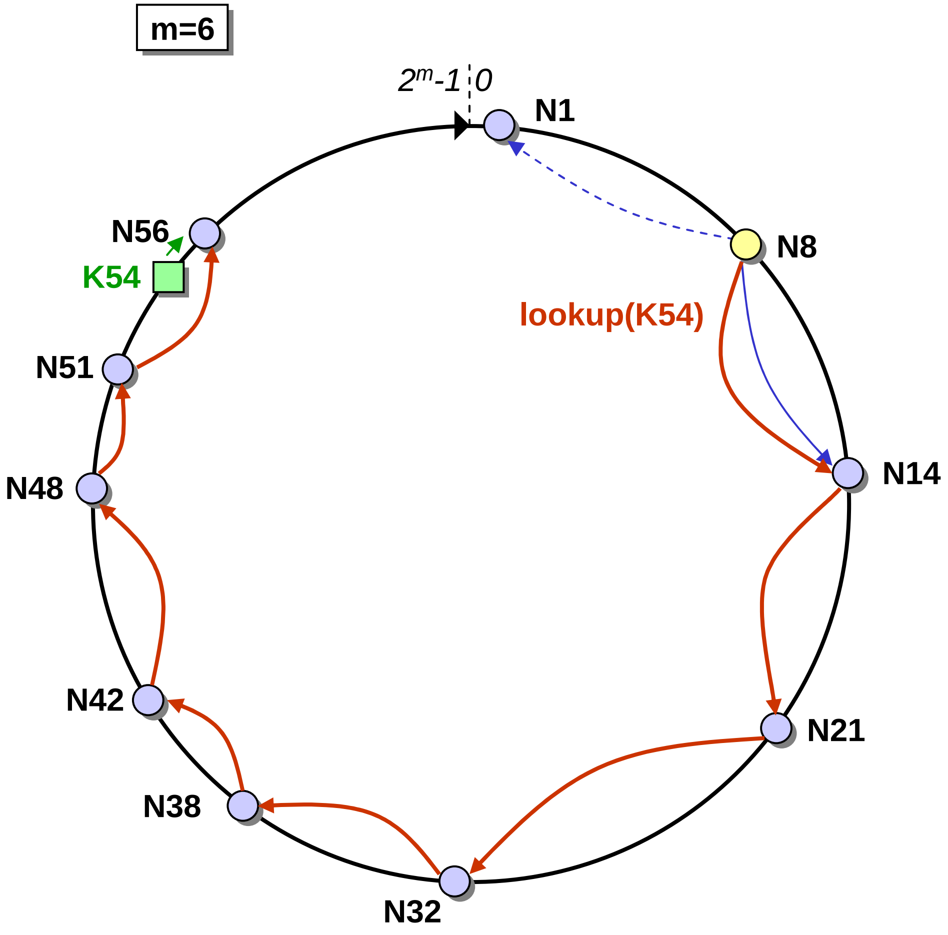
Chord (MIT)

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



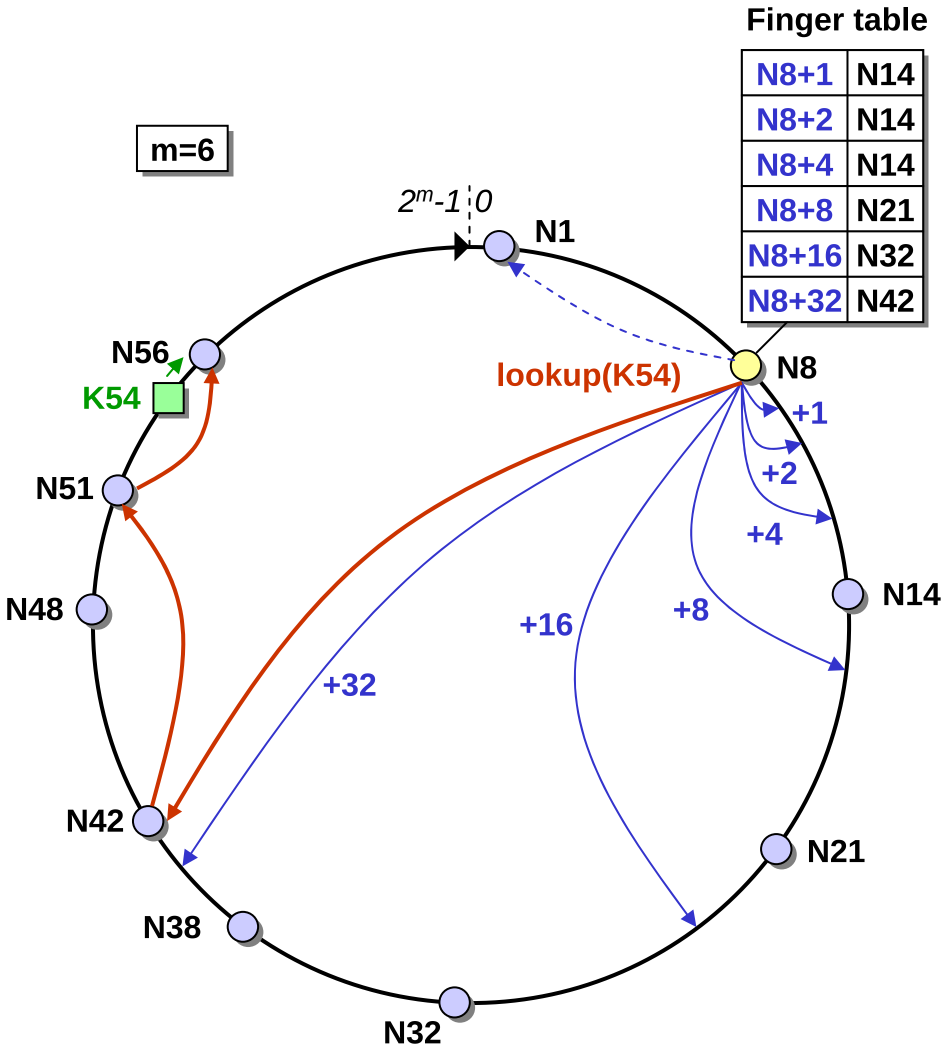
Chord State and Lookup (1)

- Basic Chord: each node knows only 2 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



Chord State and Lookup (2)

- Each node knows *m* other nodes on the ring
  - Successors: finger *i* of *n* points to node at $n+2^i$ (or successor)
  - Predecessor (for ring management)
  - *O(log N)* state per node
- Lookup is achieved by following closest preceding fingers, then 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

Chord Ring Management

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

Joining the Ring

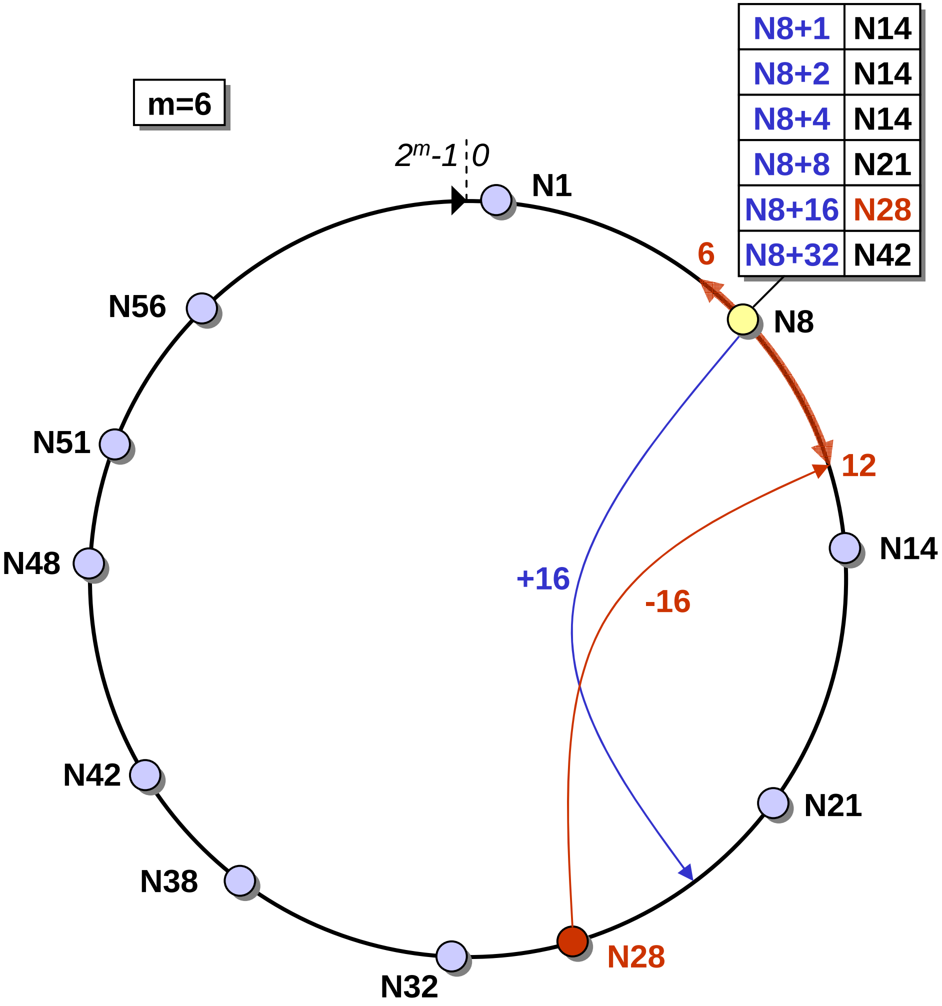
- Three step process:
  - Initialize all fingers of new node
  - Update fingers of existing nodes
  - Transfer keys from successor to new node

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

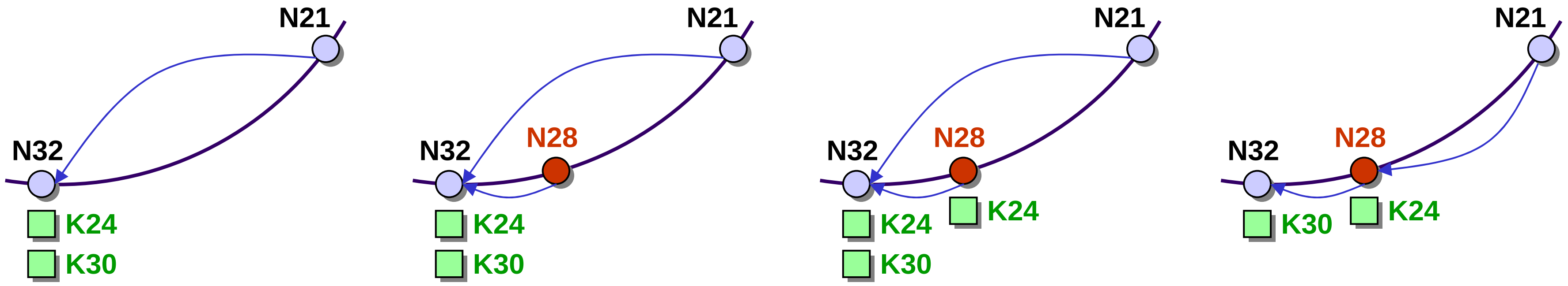
- Updating fingers of existing nodes
  - New node *j* calls update function on existing nodes that must point to *j*
    - Nodes in the ranges $[j-2^i\ ,\ pred(j)-2^i+1]$
  - *O(log N)* nodes need to be updated



N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N28
N8+32	N42

Joining the Ring — Step 3

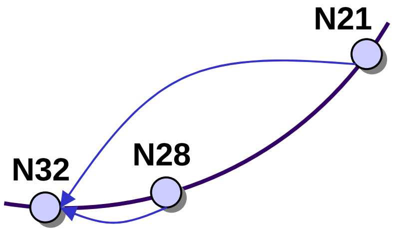
- Transfer key responsibility
  - Connect to successor
  - Copy keys from successor to new node
  - Update successor pointer and remove keys
- Only keys in the range are transferred



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)
  - Basic principle (at node *n*):

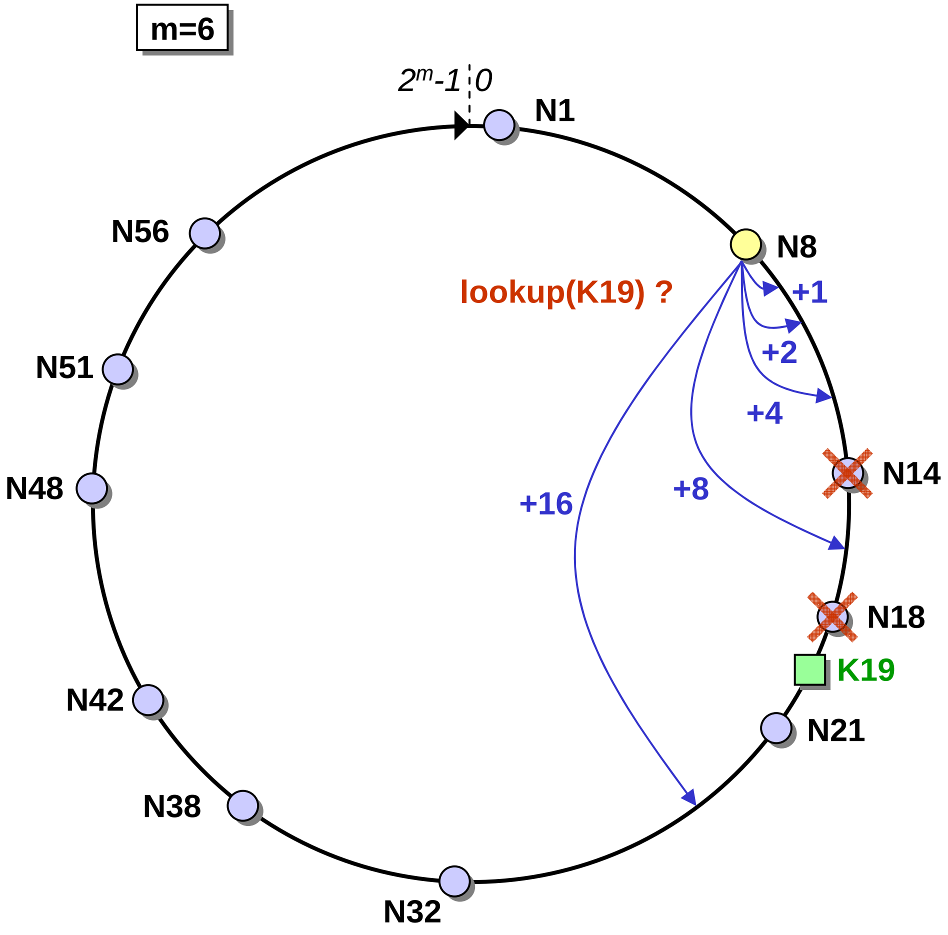
```
$x = n.succ.pred$
if $x \in (n, n.succ)$
    $n = n.succ$
    notify $n.succ$
```



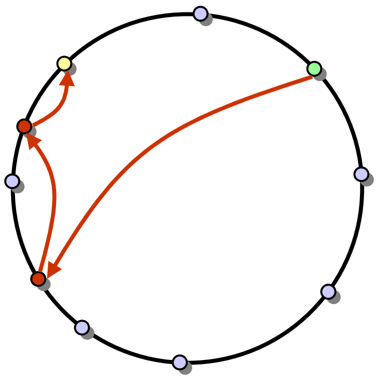
  - Eventually stabilizes the system when no node joins or fails

Dealing With Failures

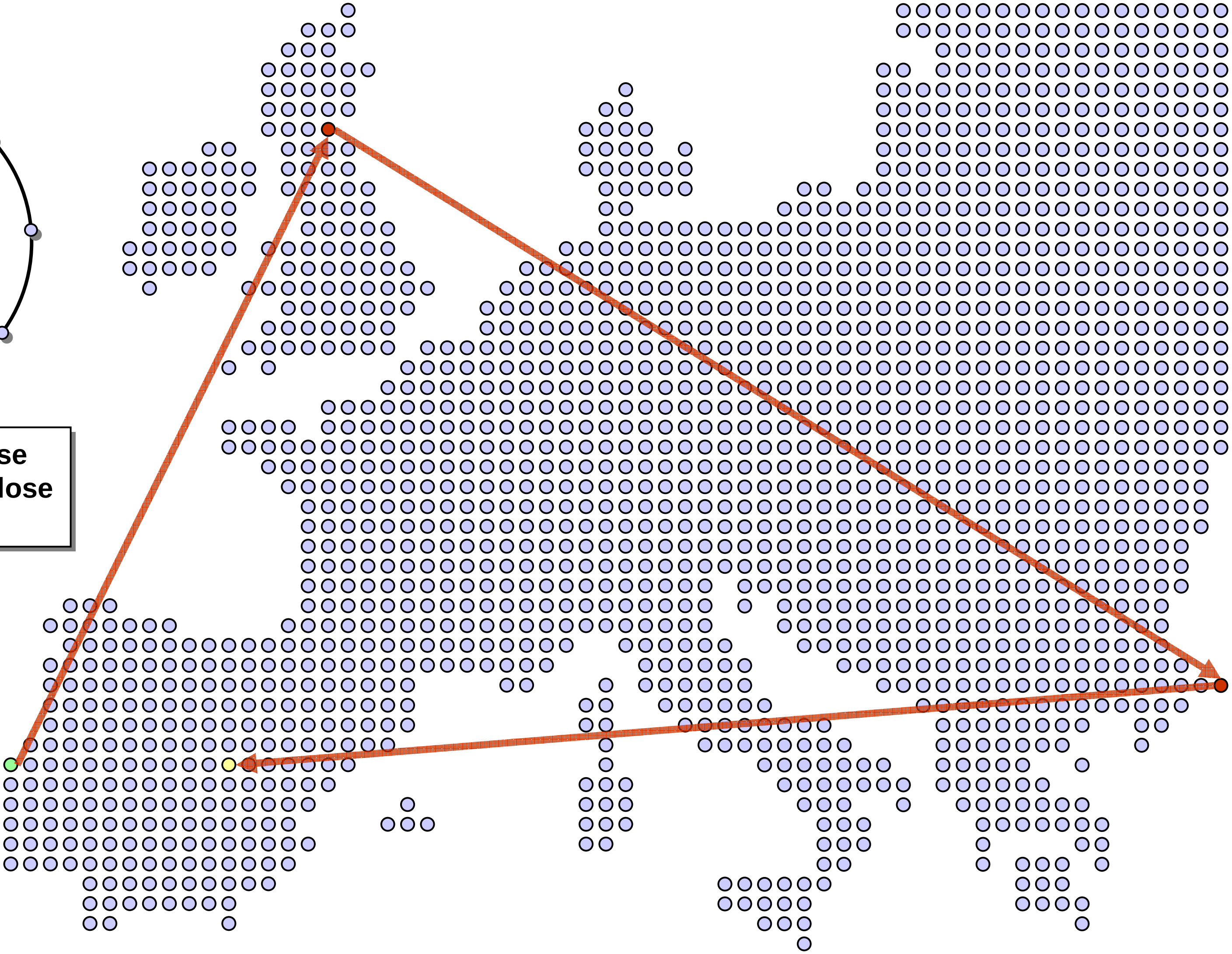
- Failure of nodes might cause incorrect lookup
  - N8 doesn't know correct successor, so lookup of K19 fails
- Solution: successor list
  - Each node *n* knows *r* immediate successors
  - After failure, *n* knows first live successor and updates successor list
  - Correct successors guarantee correct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



Chord and Network Topology

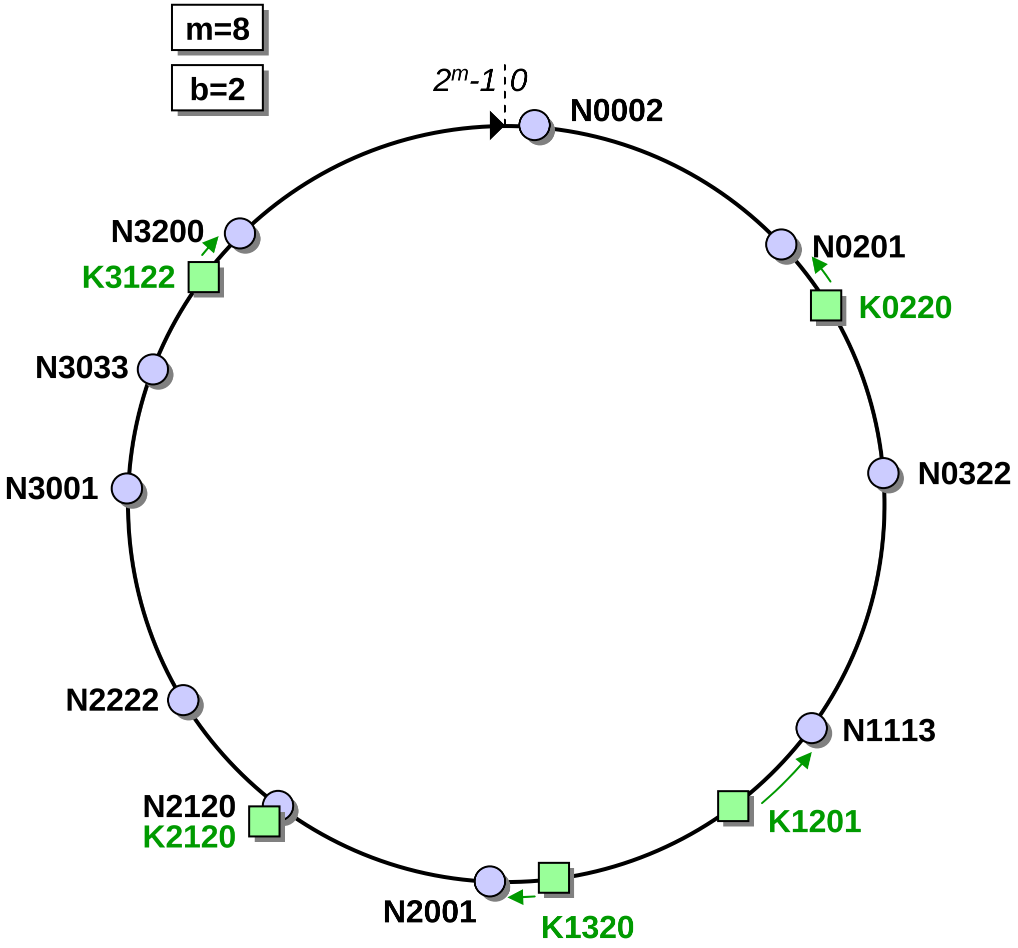


Nodes numerically-close are not topologically-close (1M nodes = 10+ hops)



Pastry (MSR)

- Circular *m*-bit ID space for both keys and nodes
  - Addresses in base 2^b with 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 the key ID

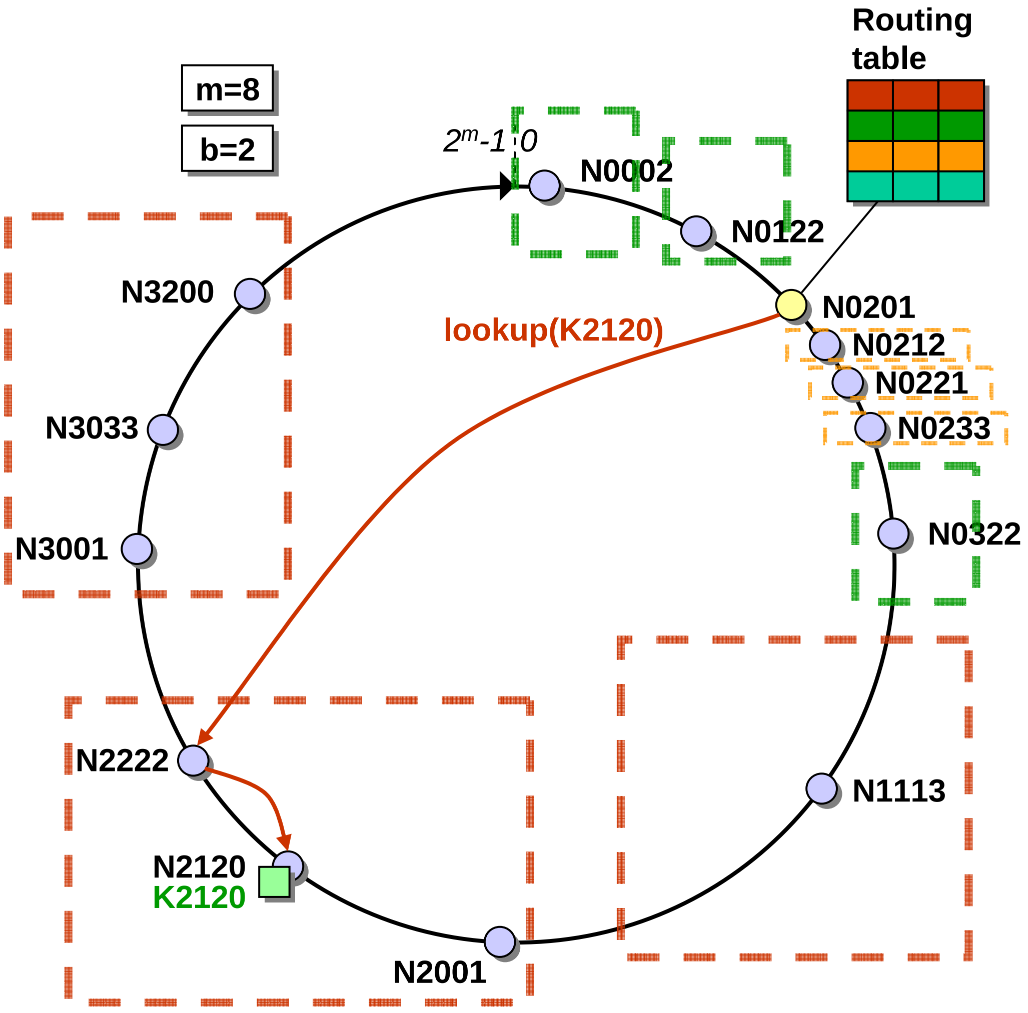


Pastry Lookup

- Prefix routing from A to B
  - At h^{th} hop, arrive at node that shares prefix with B of length at least h digits
  - Example: 5324 routes to 0629 via
    5324 → 0748 → 0605 → 0620 → 0629
  - If there is no such node, forward message to neighbor numerically-closer to destination (successor)
    5324 → 0748 → 0605 → 0609 → 0620 → 0629
  - $O(\log_{2^b} N)$ hops

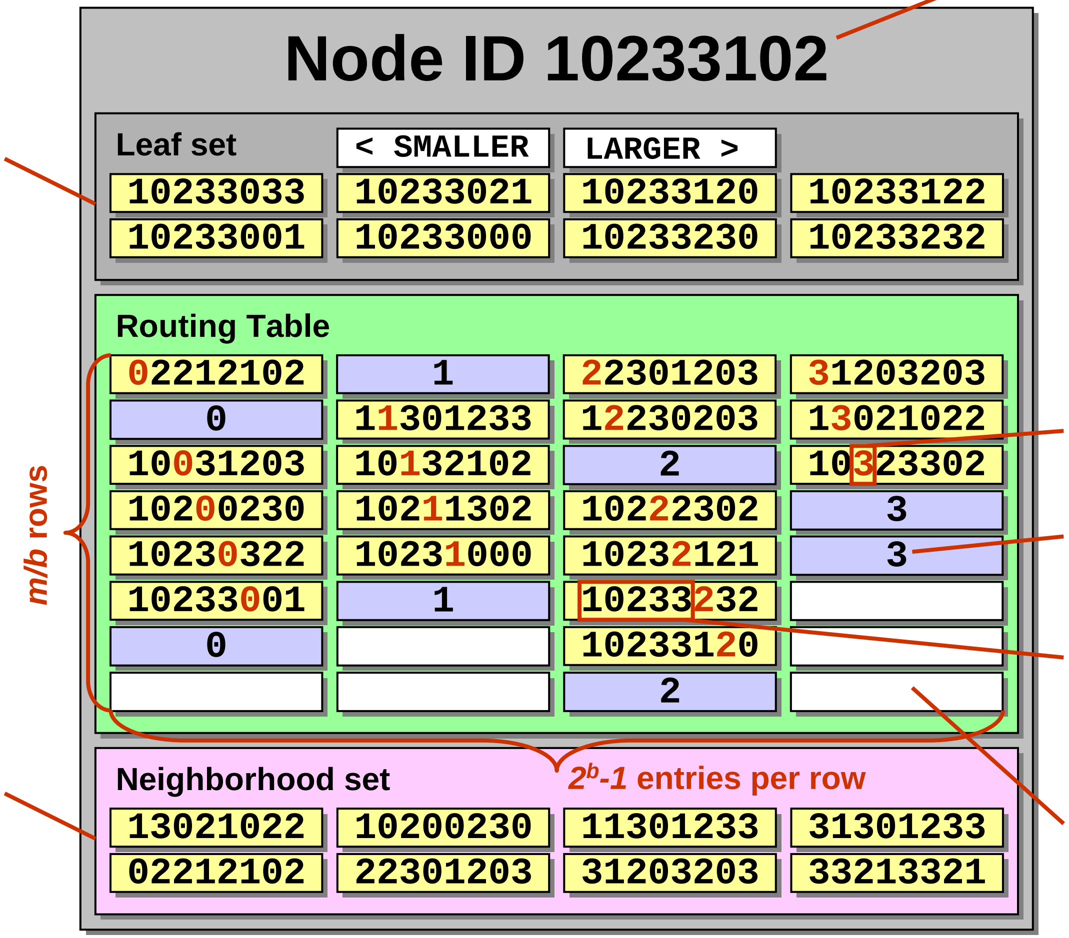
Pastry State and Lookup

- 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**



Node ID 10233102

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

Routing Table

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	

m/b **rows**

2^b-1 **entries per row**

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

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

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

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

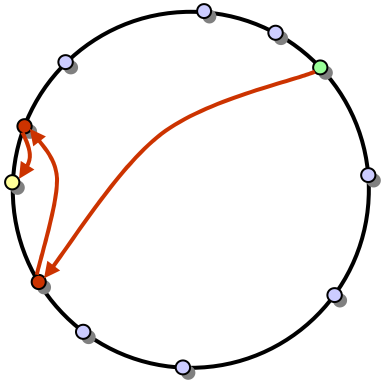
Entries in the m^{th} column have *m* as next digit

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]

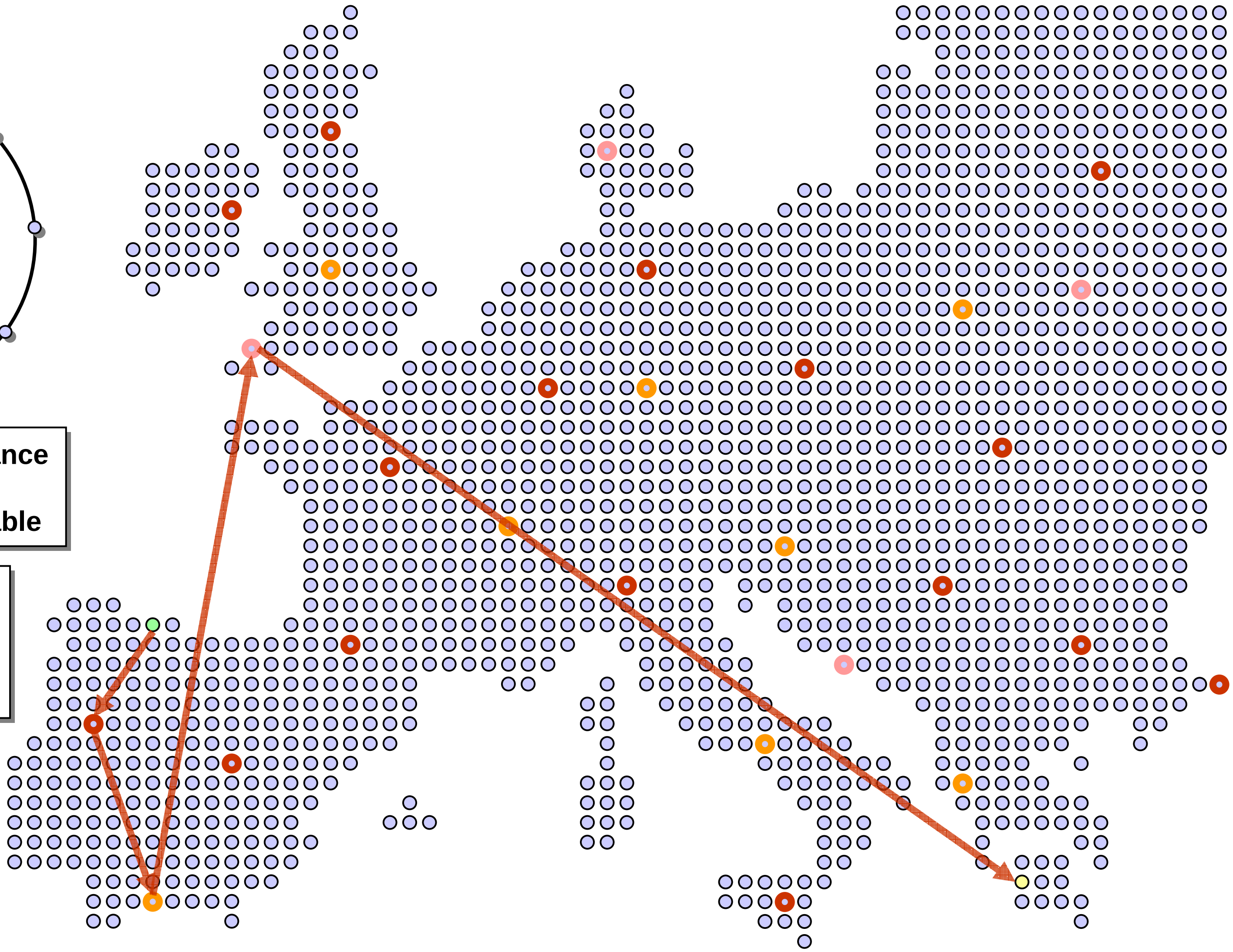
Entries with no suitable node ID are left empty

Pastry and Network Topology

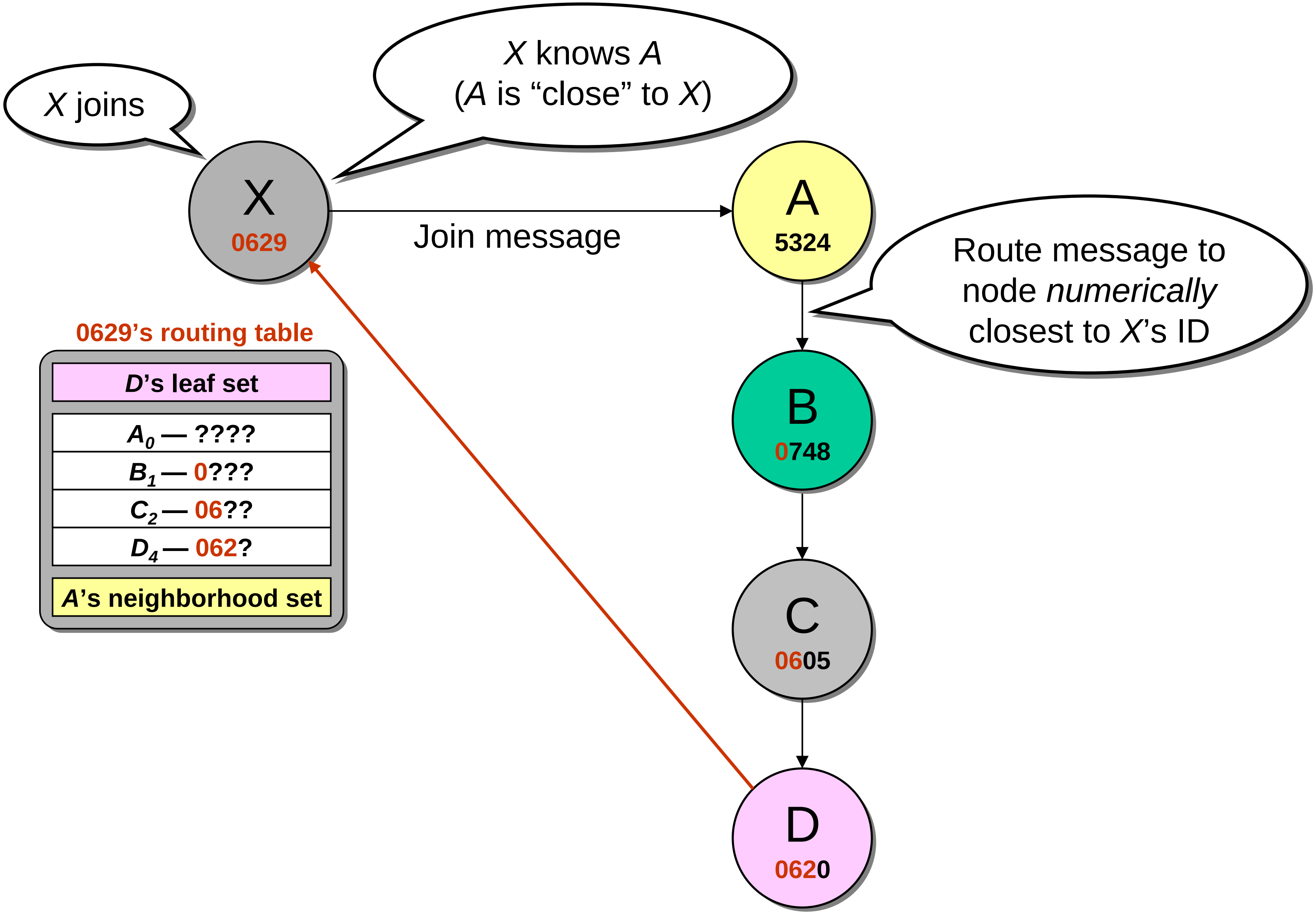


Expected node distance increases with row number in routing table

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



Joining



X joins

X knows *A*
(*A* is “close” to *X*)

Join message

Route message to node *numerically* closest to *X*’s ID

0629’s routing table

***D*’s leaf set**

A_0 — ????

B_1 — 0???

C_2 — 06??

D_4 — 062?

***A*’s neighborhood set**

Locality

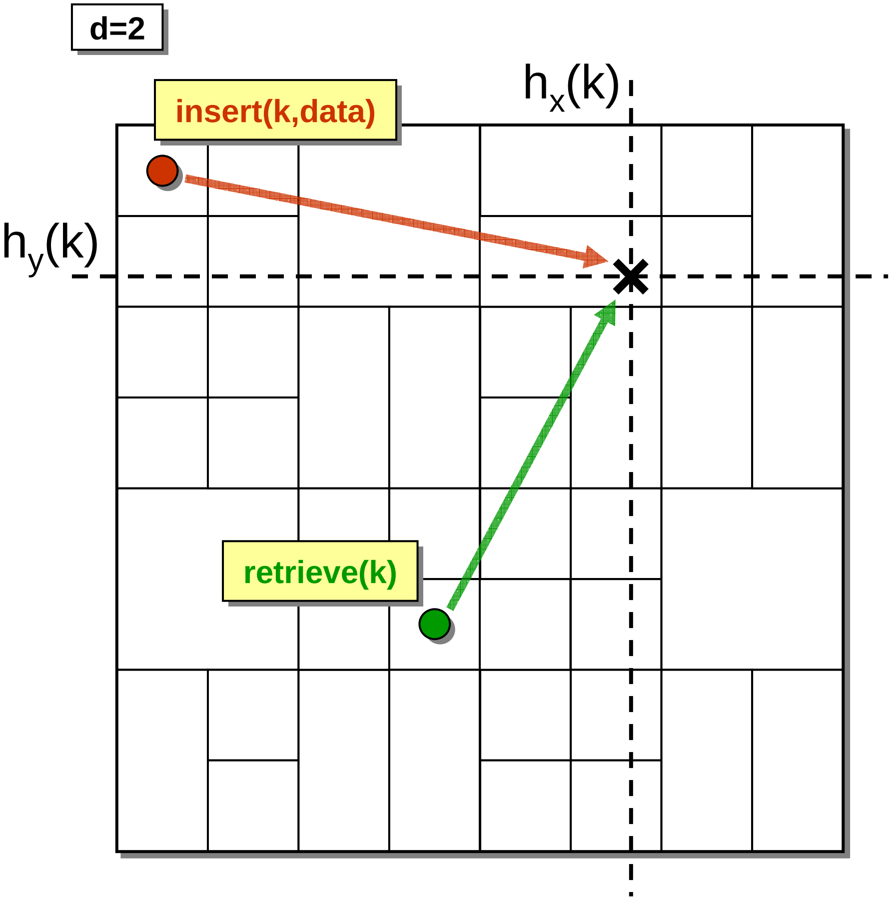
- The joining phase preserves the locality property
  - First: *A* must be near *X*
  - Entries in row zero of *A*'s routing table are close to *A*, *A* is close to *X* $\Rightarrow$ X_0 can be A_0
  - The distance from *B* to nodes from B_1 is much larger than distance from *A* to *B* (*B* is in A_0) $\Rightarrow$ B_1 can be reasonable choice for X_1, C_2 for X_2, etc.
  - To avoid cascading errors, *X* requests the state from each of the node in its routing table and updates its own with any closer node
- This scheme works “pretty well” in practice
  - Minimize the distance of the next routing step with no sense of global direction
  - Stretch around 2-3

Node Departure

- Node is considered failed when its immediate neighbors in the node ID space cannot communicate with it
  - To replace a failed node in the leaf set, the node contacts the live node with the largest index on the side of failed node, and asks for its leaf set
  - To repair a failed routing table entry R^d_l, node contacts first the node referred to by another entry R^i_l, $i \neq d$ of the same row, and ask for that node's entry for R^d_l
  - If a member in the *M* table, is not responding, node asks other members for their *M* table, check the distance of each of the newly discovered nodes, and update its own *M* table

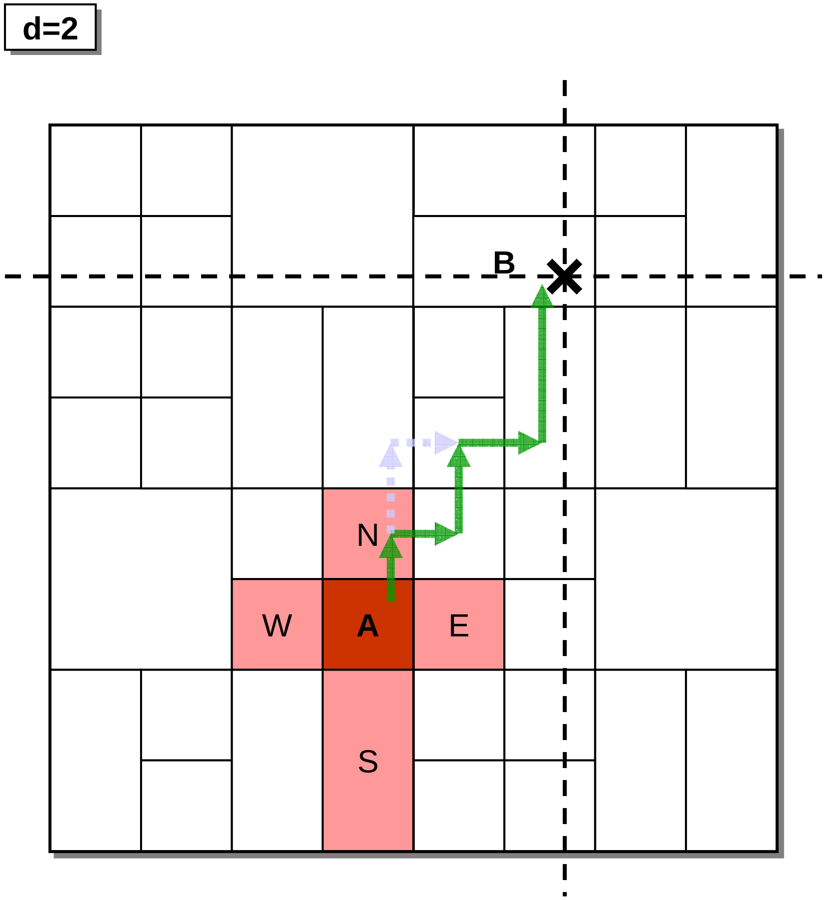
CAN (Berkeley)

- Cartesian space (*d*-dimensional)
  - Space wraps up: *d*-torus
- Incrementally split space between nodes that join
- Node (cell) responsible for key *k* is determined by hashing *k* for each dimension



CAN State and Lookup

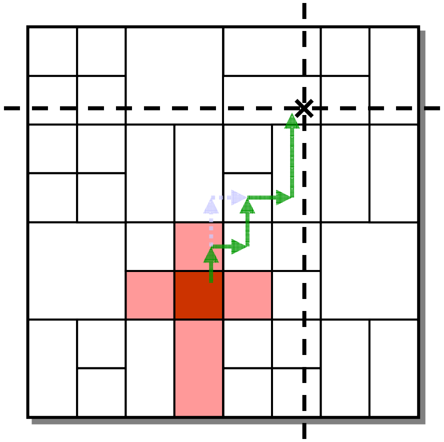
- A node *A* only maintains state for its immediate neighbors (*N*, *S*, *E*, *W*)
  - $2d$ neighbors per node
- Messages are routed to neighbor that minimizes Cartesian distance
  - More dimensions means faster the routing but also more state
  - $(dN^{1/d})/4$ hops on average
- Multiple choices: we can route around failures

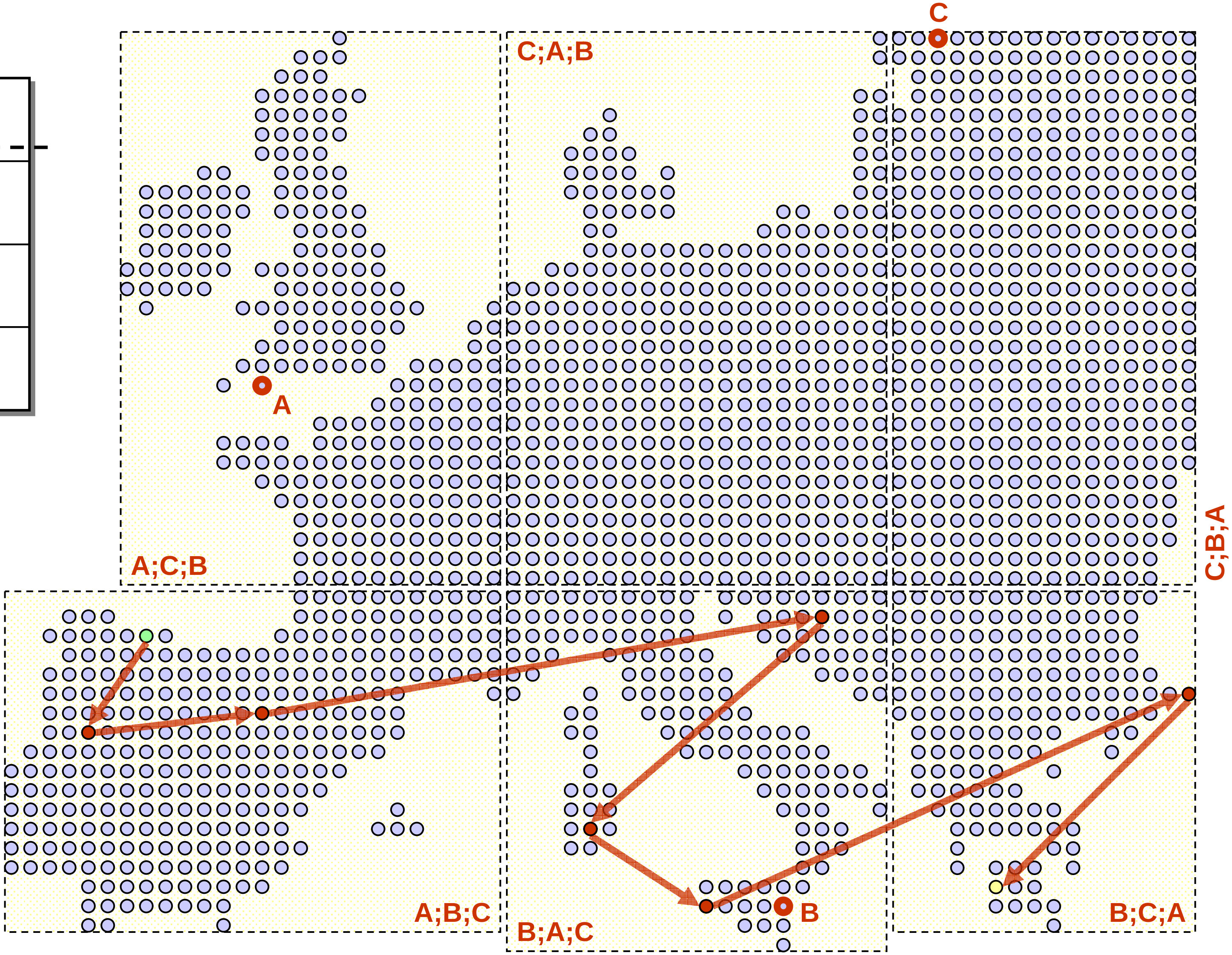


CAN Landmark Routing

- CAN nodes do not have a pre-defined ID
- Nodes can be placed according to locality
  - Use well known set of *m* landmark machines (e.g., root DNS servers)
  - Each CAN node measures its RTT to each landmark
  - Orders the landmarks in order of increasing RTT: *m!* possible orderings
- CAN construction
  - Place nodes with same ordering close together in the CAN
  - To do so, partition the space into *m!* zones: *m* zones on *x*, *m-1* on *y*, etc.
  - A node interprets its ordering as the coordinate of its zone

CAN and Network Topology





Use *m* landmarks to split space in *m!* zones

Nodes get random zone in their zone

Topologically-close nodes tend to be in the same zone

Conclusion

- DHT is 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 network is difficult to analyze
- Many open issues worth exploring