Structured P2P-overlays

- Motivation
  - Locate content efficiently
- Solution – DHT (Distributed Hash Table)
  - Particular nodes hold particular keys
  - Locate a key: *route search request to a particular node that holds the key*
  - Representative solutions
    - *CAN, Pastry/Tapestry, Chord, etc.*

Challenges to Structured P2P-overlays

- Load balance
  - spreading keys evenly over the nodes.
- Decentralization
  - no node is more important than any other.
- Scalability
  - Lookup must be efficient even with large systems
- Peer dynamics
  - Nodes may come and go, may fail
  - Make sure that “the” node responsible for a key can always be found

Key Issue: Distributed Indexing

Tree-Based

- Pros:
  - Easy and well-known techniques
  - O(log n) operations
  - Equality & Inequality Queries
    search for d =val, search for d < val, search for 'D*',···
- Cons:
  - Node hierarchy / Specialization of nodes (root, sub*-root, leaves, etc.)
  - Bottleneck
  - Fault-prone, lack of dynamic...

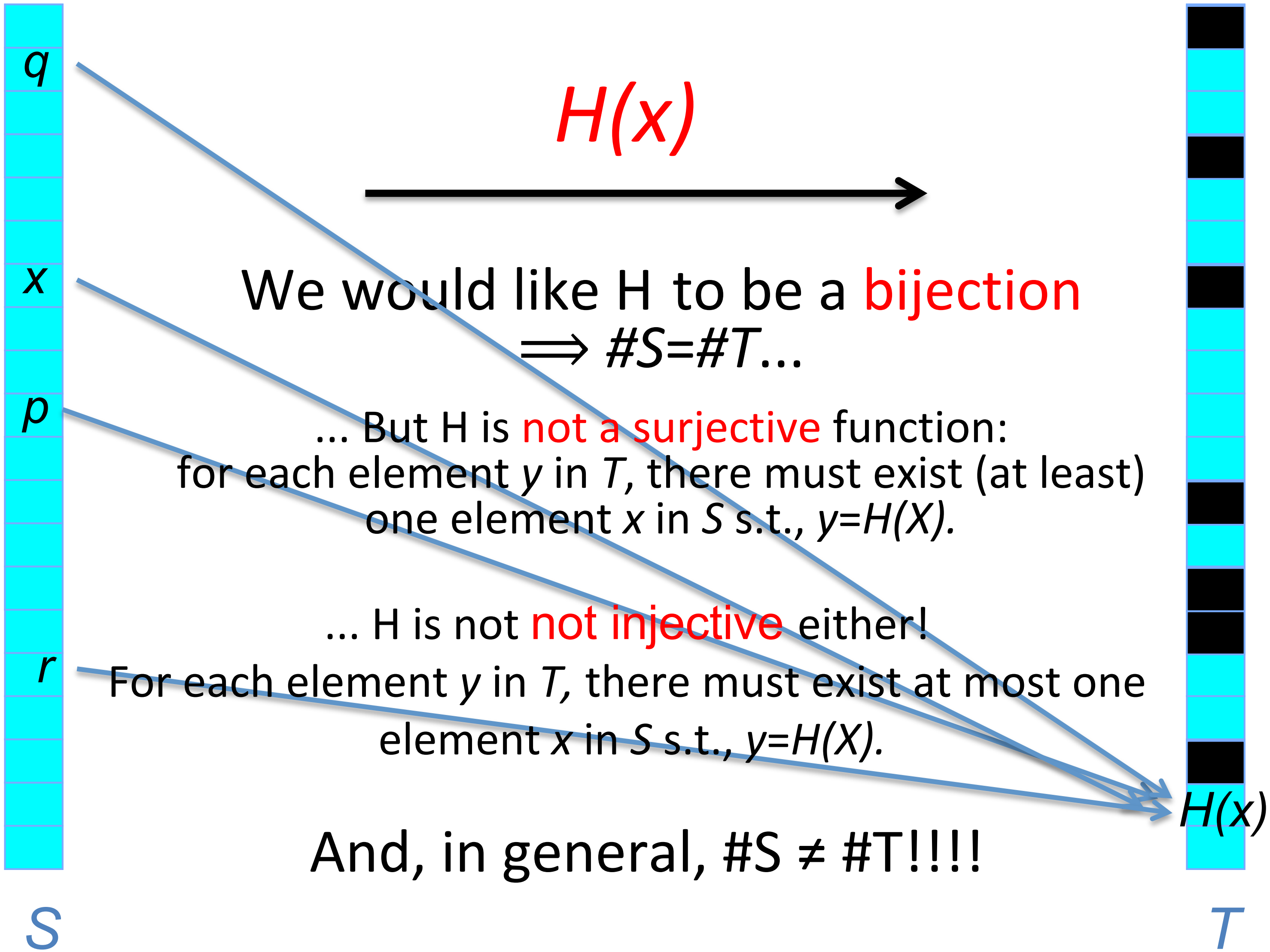
Key Issue: Distributed Indexing

Hash Function

- Pros:
  - Easy and well-known techniques
  - (Theoretically) O(1) operation
  - Fault-tolerance and dynamic-maintenance
- Cons:
  - Worst-Case: O(n) operations
  - Useless for Inequality Queries

Key Issue: Distributed Indexing

Hash Function



We would like H to be a bijection

$\Longrightarrow$ *#S=#T*...

... But H is not a surjective function:
for each element *y* in *T*, there must exist (at least) one element *x* in *S* s.t., *y=H(X).*

... H is not not injective either!
For each element *y* in *T,* there must exist at most one element *x* in *S* s.t., *y=H(X).*

And, in general, #S ≠ #T!!!!

Key Issue: Distributed Indexing

Hash Function

$$\xrightarrow{\textit{SHA-1(x)}}$$

(Secure Hash Algorithm)

Distributed Hash Table

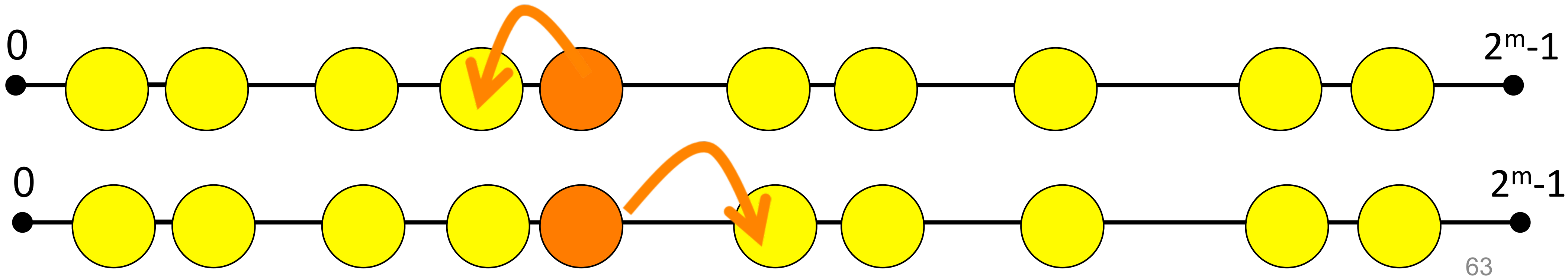
Keys & Nodes to IDs

- Keys and nodes are represented by identifiers taken from an ID space of 2^m values (*m*-bits space) e.g., $[0..2^{160}-1]$, all of them computed through the same hash function, e.g., SHA-1.
- For instance:
  - *ID("Let it be") = SHA1("Let it be")*
  - *ID(n) = SHA1("IP address of n")*

Distributed Hash Table

Keys & Nodes to IDs

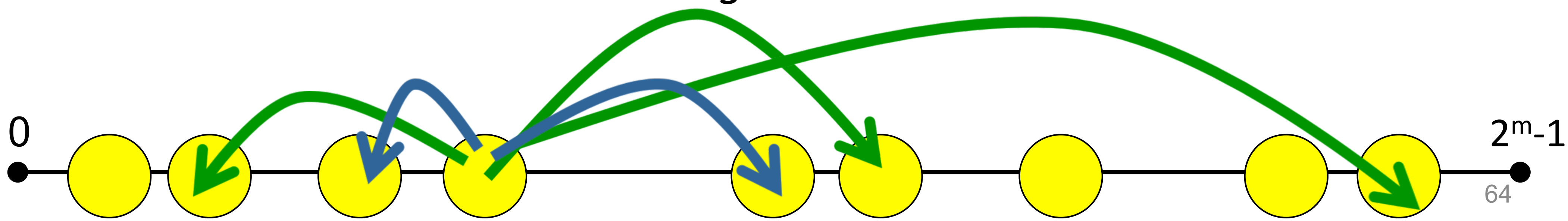
- Each node that participates to the DHT stores some keys
- A key *k* is stored at the node *n* such that, for examples:
  - *ID(n)* is the 'closest' to *ID(k) (w.r.t.*, a given distance*)*;
  - ID(n) is the smallest node id (strictly) greater than ID(k)



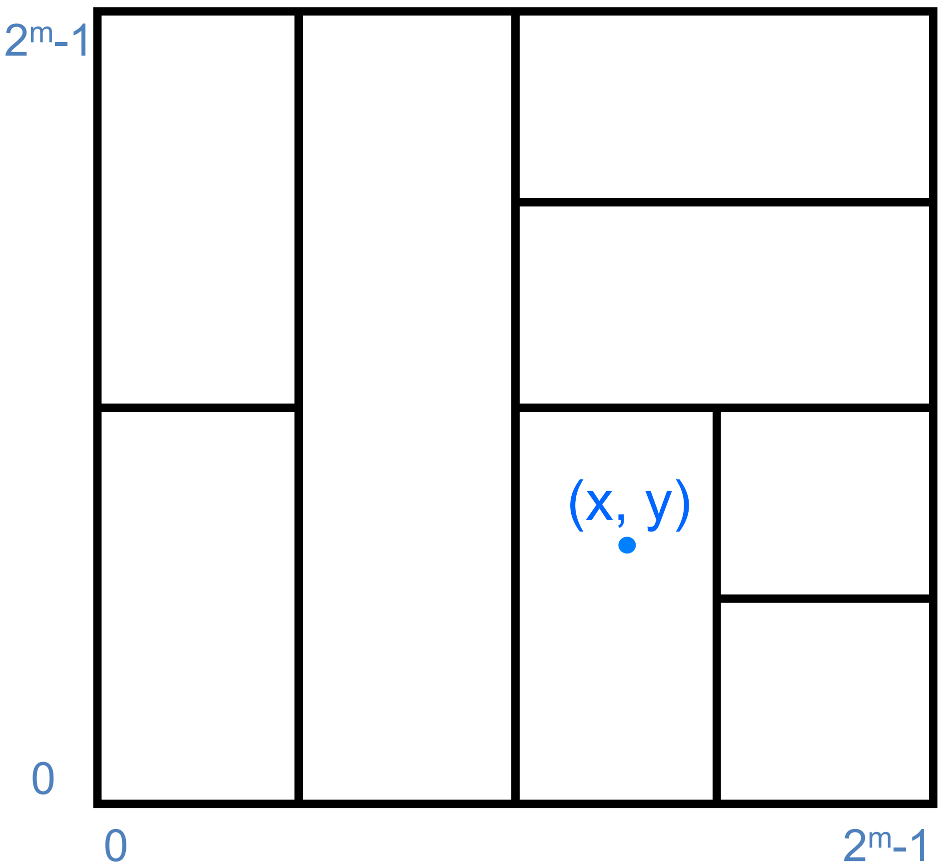
Distributed Hash Table

Build Overlay Network

- Each node has two sets of neighbors in the key space:
  - Immediate neighbors:
    - *Must avoid partition of the DHT*
    - *Must guarantee linear searching/routing*
    - *Dynamic/Fault-tolerance, self-adaptation*
  - Long-range neighbors:
    - *Allow sub-linear routing*

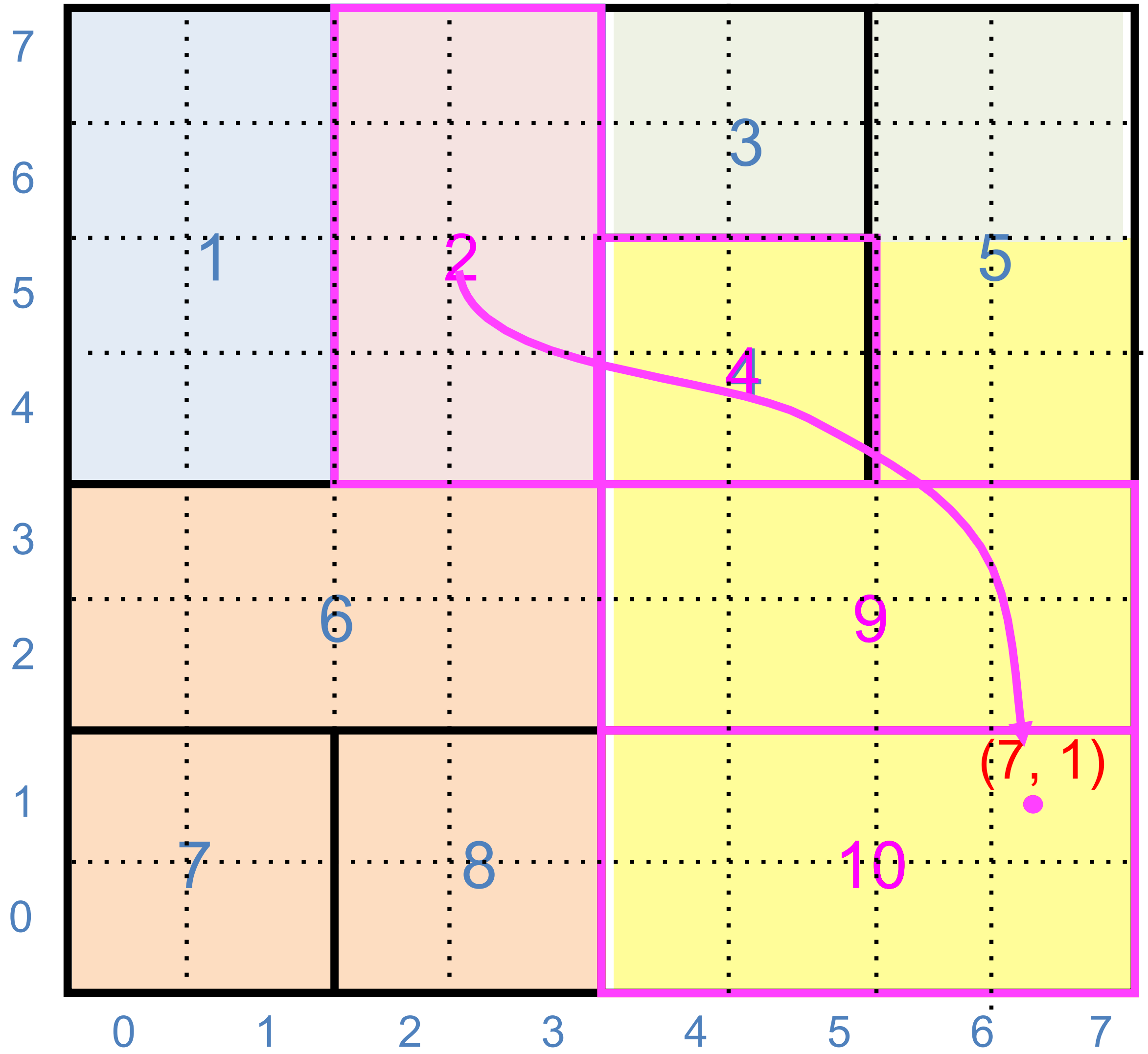


Content-Addressable Network (CAN)



- Maps a 2-dimensional coordinate space
- Each node is responsible for a fraction of the space
- A hash pair is provided using two hash functions on the value to be stored
- Routing is done by forwarding to the neighbor that is closest in Cartesian distance

CAN example



Routing table for node 2

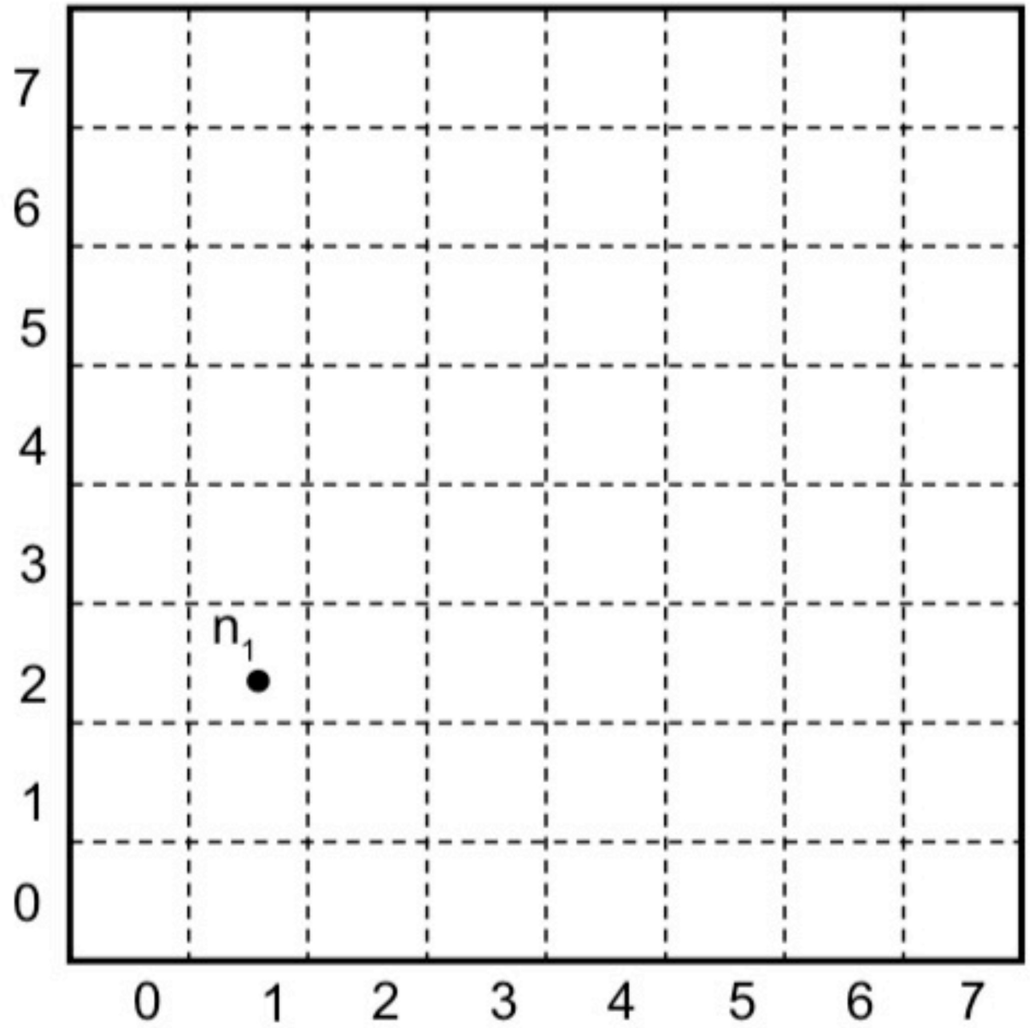
Node	Range
2	(2, 4) to (3, 7)
1	(0, 4) to (1, 7)
3	(4, 6) to (7,7)
4	(4, 0) to (7, 5)
6	(0, 0) to (3, 3)

Entry hashed to (7, 1) is stored at node 10

Search path from 2 to (7, 1):
2 → 4 → 9 → 10

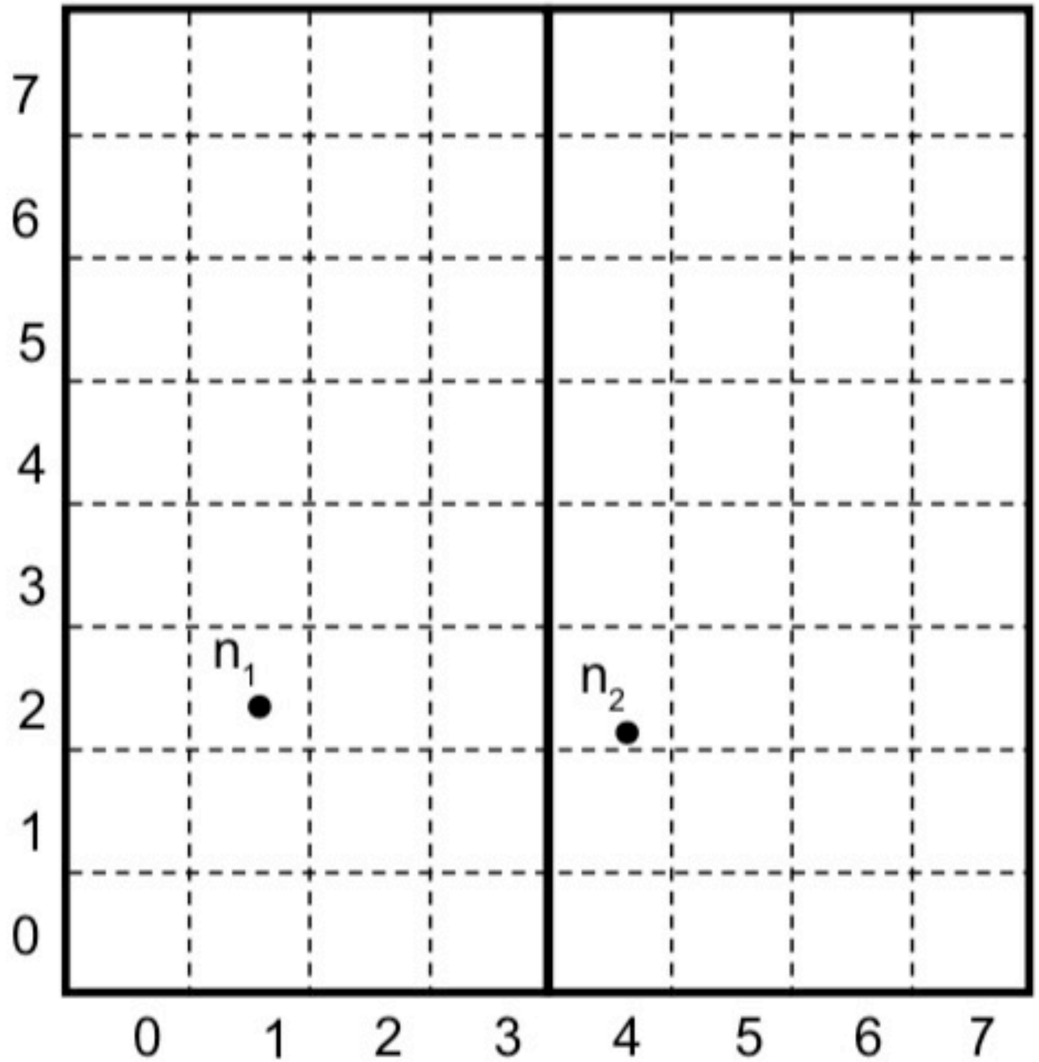
Building CAN

- The first node n_1 covers the entire space.



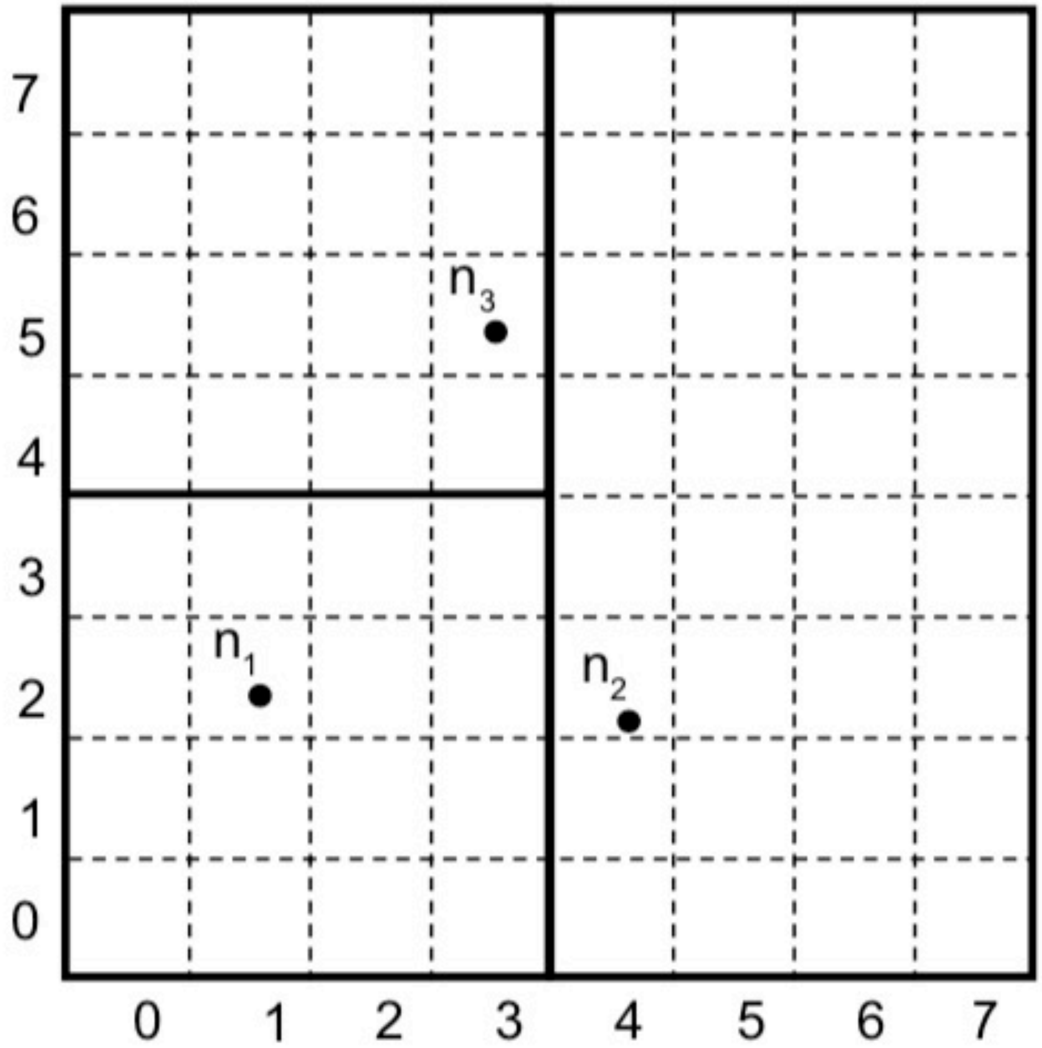
Building CAN

- Node n_2 joins and the space is divided into 2, according to n_2' coordinates.



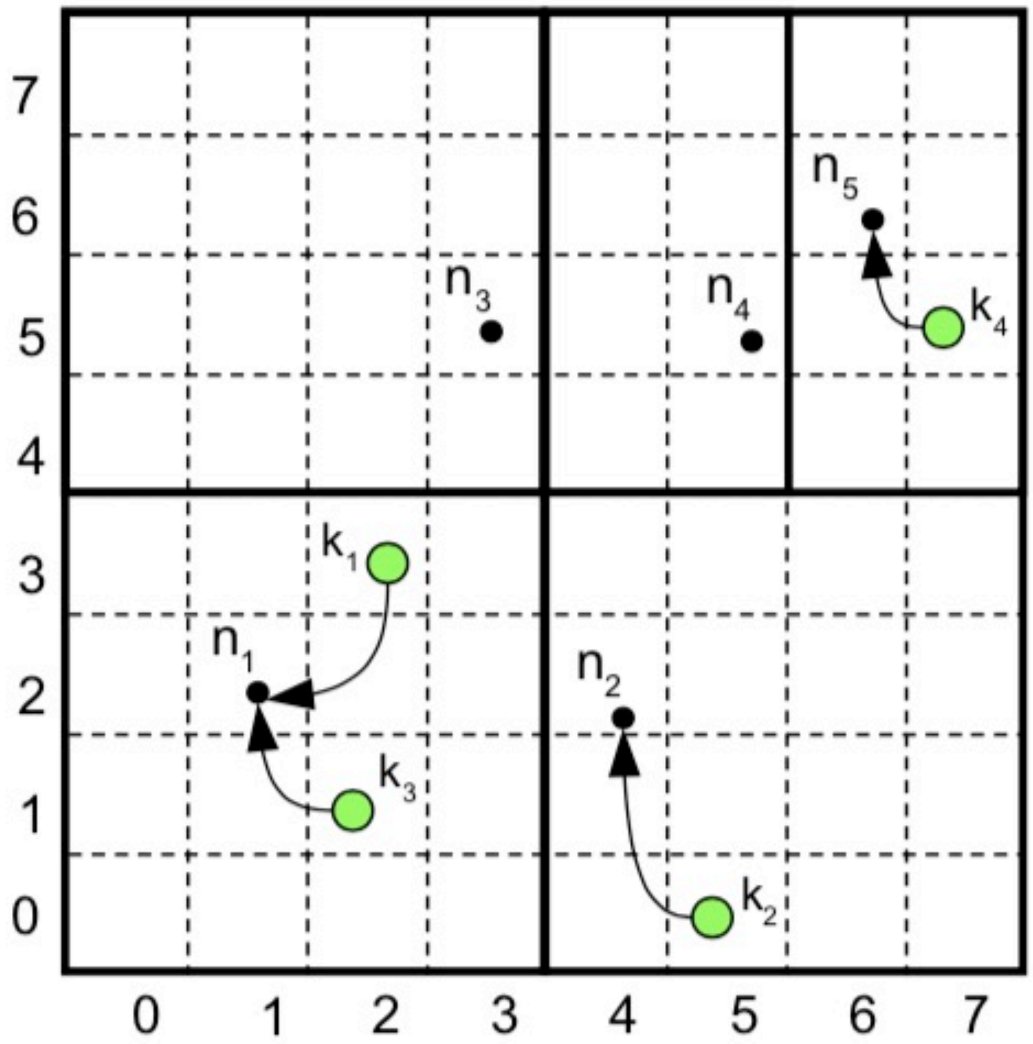
Building CAN

- Node n_3 joins with coordinates (3,5).



Building CAN

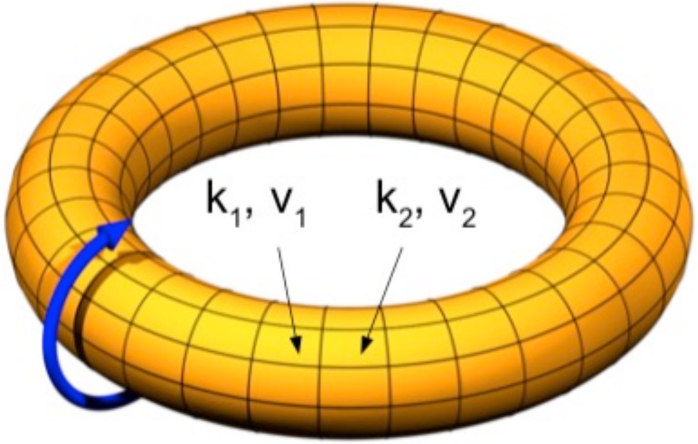
- Items are stored on nodes according to their coordinates.



CAN joins and departures

- Joining node picks a random pair (X, Y) and contacts node A
  - A sends a join message to B, the node responsible for the pair
  - B divides is its space with A and shares its routing information appropriately
  - A and B contact all its neighbors with updated info
- Departing node contacts its neighbors and finds one that can manage its space
  - The node sends updated routing info to its neighbors

CAN features



- Actually, a torus...
- Simple to understand and thus
- simple to add features to:
  – D-dimensional space (not just 2)
  – Dynamic division of space for load balancing
- Hop #: $O(Dn^{1/D})$
- Routing Table: $O(D)$
- Possible overload

Tapestry/Pastry

- Both based on **Plaxton Mesh**
  - Nodes – Act as routers, clients and servers simultaneously.
  - Objects – Stored on servers, searched by clients.
- Objects and nodes have **unique, location independent names**
  - Random fixed-length bit sequence in some base
  - Typically, hash of hostname or of object name
  - A hashing function *h* assigns each node and key a *m*-bit identifier

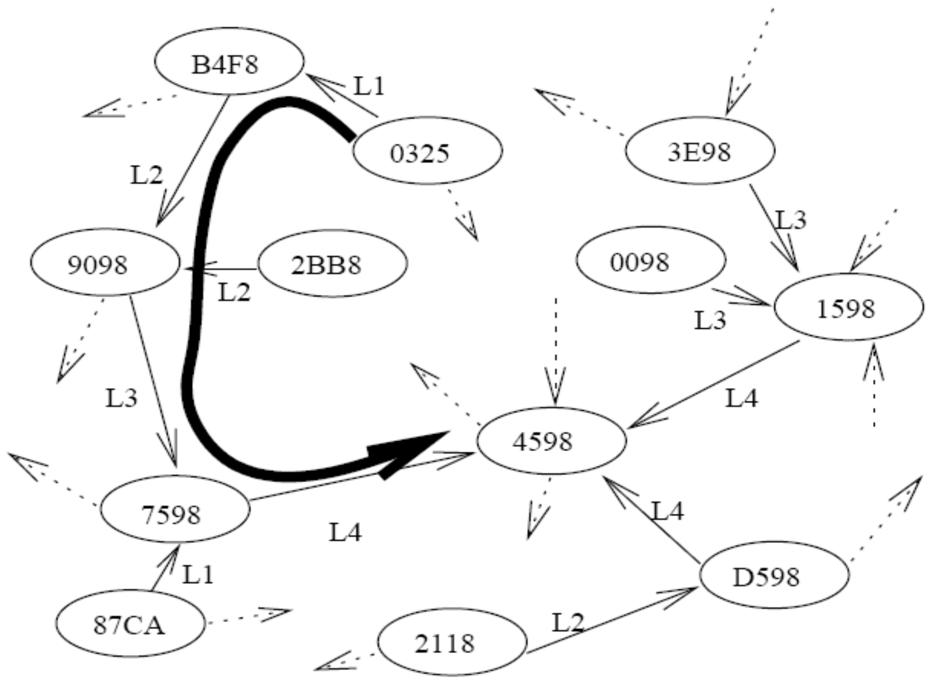
Key=“LetItBe” $\xrightarrow{h}$ ID=60

IP=“157.25.10.1” $\xrightarrow{h}$ ID=101

- What Plaxton does:
  - Given **message M** and **destination D**, a Plaxton mesh **routes M** to the node whose name is **numerically closest to D**

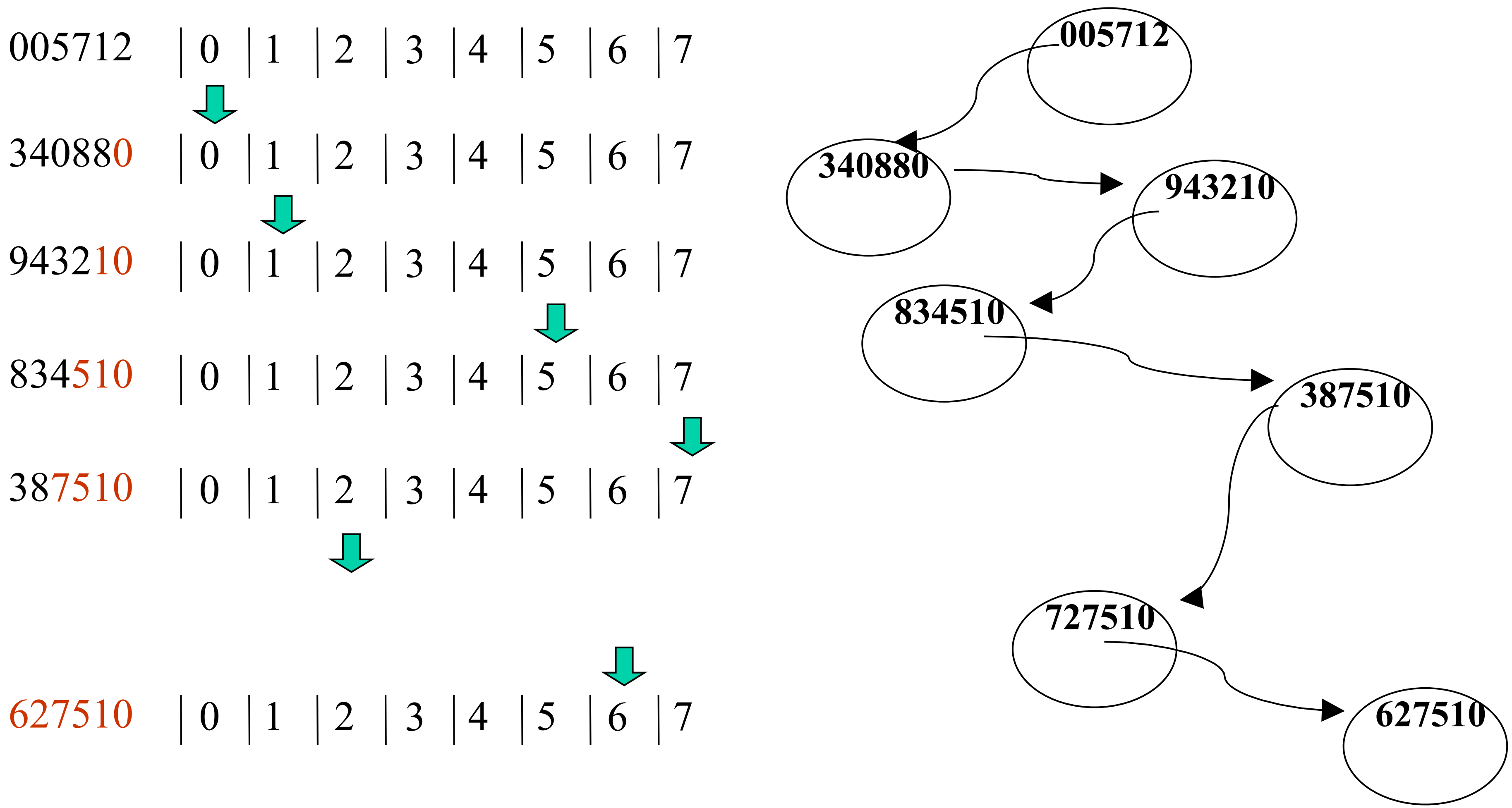
Plaxton Mesh Routing

- Routing done incrementally, digit by digit
  - In each step, message sent to node with address one digit closer to destination.
- Example - node 0325 sends M to node 4598
  - Possible route:
    - 0325
    - B4F**8**
    - 90**98**
    - 7**598**
    - **4598**



Plaxton Mesh
Routing : Another Example

Consider 2^{18} namespace, 005712 → 627510



Plaxton Mesh
Routing Tables

- Each node has a neighbor map
  - Used to find a node whose address has one more digit in common with the destination
- Size of the map: b*$\log_b$N, where b is the base used for the address

Plaxton Mesh
Routing Tables

- x are wildcards. Can be filled with any address.
  - Each slot can have several addresses. Redundancy.
- Example: node 3642 receives message for node 2342
  - Common prefix: XX**42**
    - Look into second column.
  - Must send M to a node one digit "closer" to the destination.
    - Any host with an address like X342.

Node 3642

0642	x042	xx02	xxx0
1642	x142	xx12	xxx1
2642	x242	xx22	xxx2
3642	x342	xx32	xxx3
4642	x442	xx42	xxx4
5642	x542	xx52	xxx5
6642	x642	xx62	xxx6
7642	x742	xx72	xxx7

Plaxton Mesh
Routing Algorithm

Assume that the destination node is a tree root. To route to node (xyz)

- Let shared suffix = n so far
- Look at level n+1
- Match the next digit of the destination id
- Send the message to that node

Node 3642

0642	x042	xx02	xxx0
1642	x142	xx12	xxx1
2642	x242	xx22	xxx2
3642	x342	xx32	xxx3
4642	x442	xx42	xxx4
5642	x542	xx52	xxx5
6642	x642	xx62	xxx6
7642	x742	xx72	xxx7

Pastry: Routing procedure

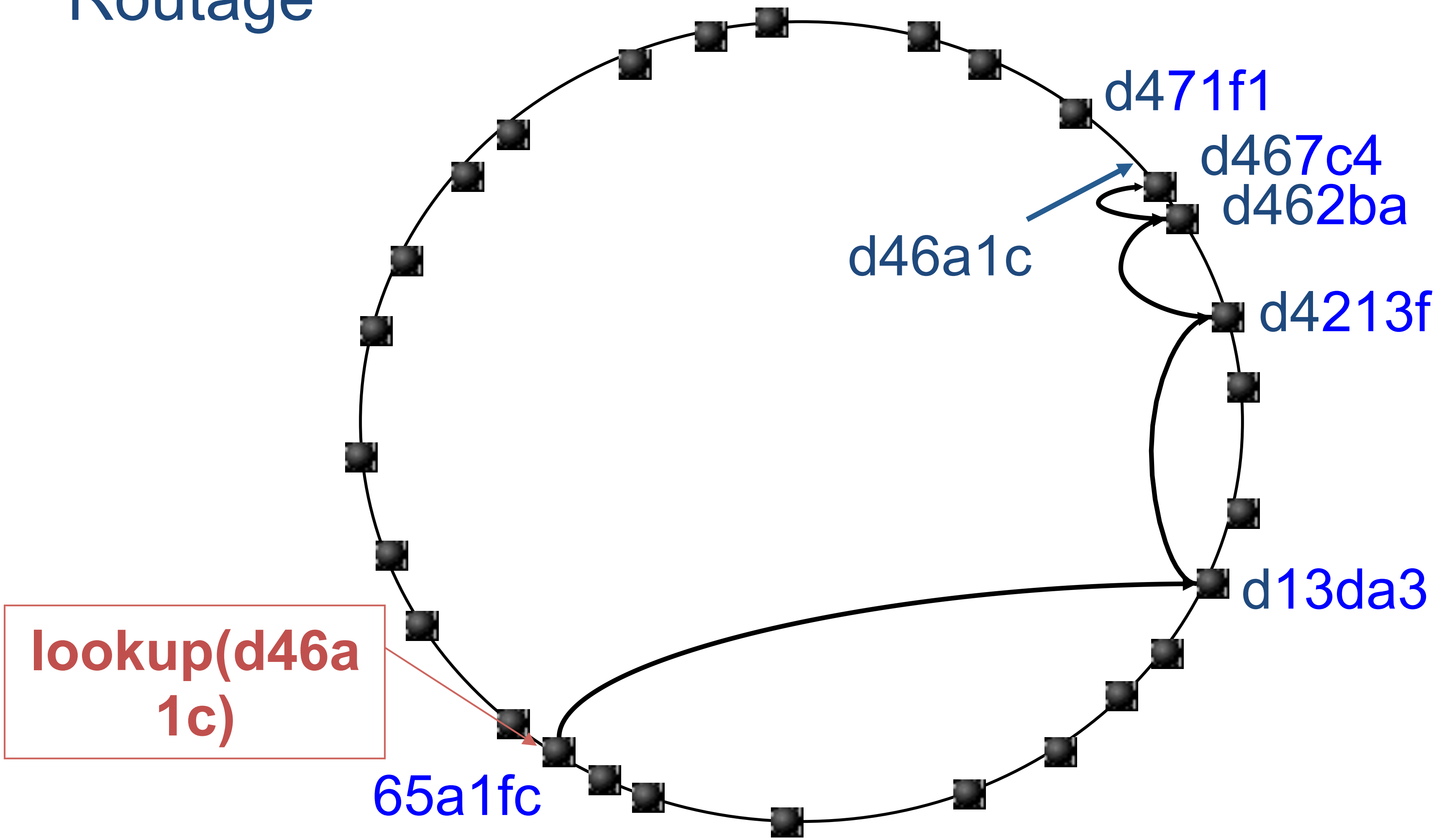
If (destination is within range of our leaf set)
forward to numerically closest member
else
let *l* = length of shared prefix
let *d* = value of *l*-th digit in *D*' s address
if (R_l^d exists)
forward to R_l^d
else
forward to a known node that
(a) shares at least as long a prefix
(b) is numerically closer than this node

Plaxton Mesh
Main Characteristics

- Memory per node : O(log(N))
- Self-adaptative
- Rooting in *O(log(N)*) hops
- Fault-tolerant
- Pastry, Patestry : Very similar to *Chord*

Plaxton Mesh
Main Characteristics

Routage



Chord Protocol

- A consistent hashing function (SHA-1) assigns each node and key a *m*-bit identifier

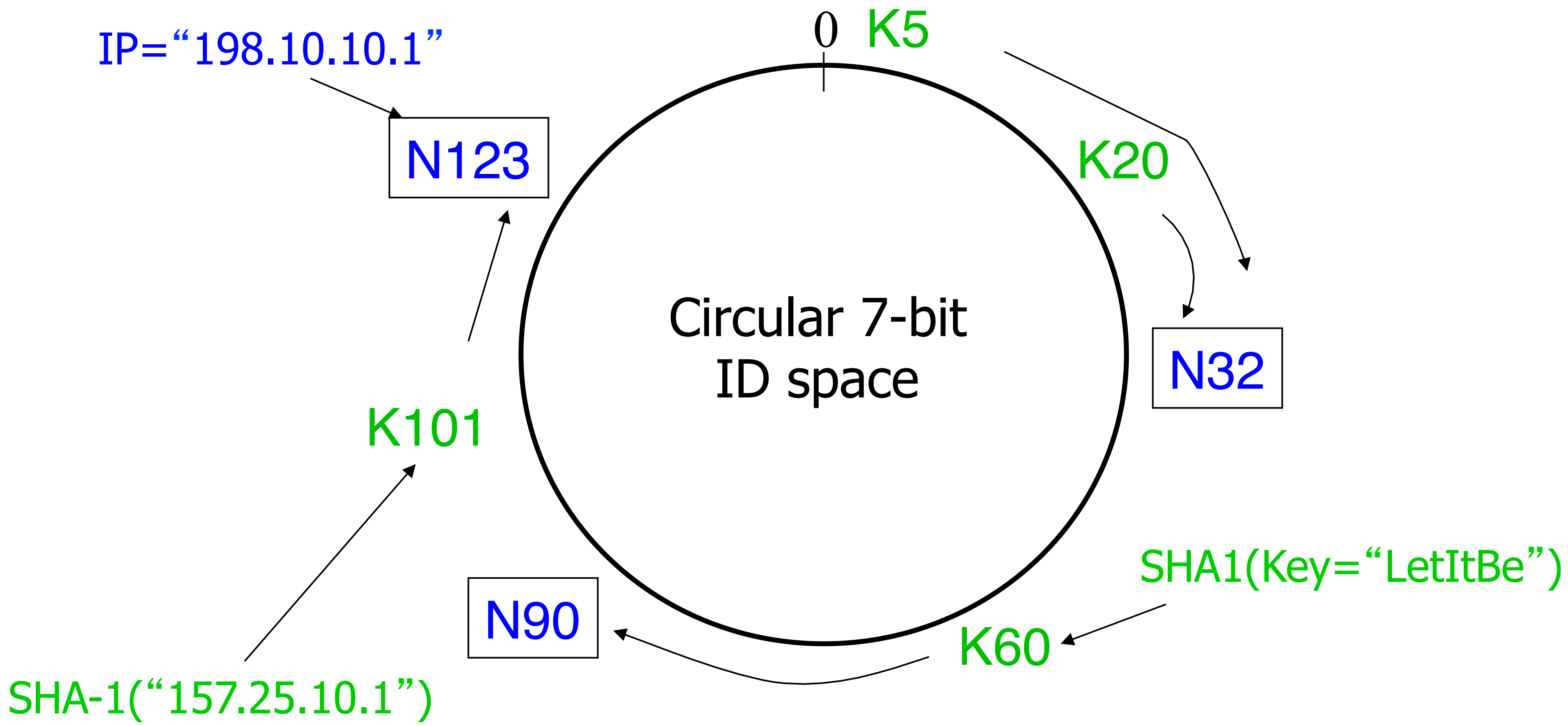
Key="LetItBe" $\xrightarrow{\text{SHA-1}}$ ID=60

IP="157.25.10.1" $\xrightarrow{\text{SHA-1}}$ ID=101

- Identifiers are ordered on a identifier circle modulo 2^m called a chord ring.
- *successor(k)* = first node whose identifier is >= identifier of k in identifier space.

Consistent Hashing (cont.)

- A key is stored at its **successor**: node with next higher ID

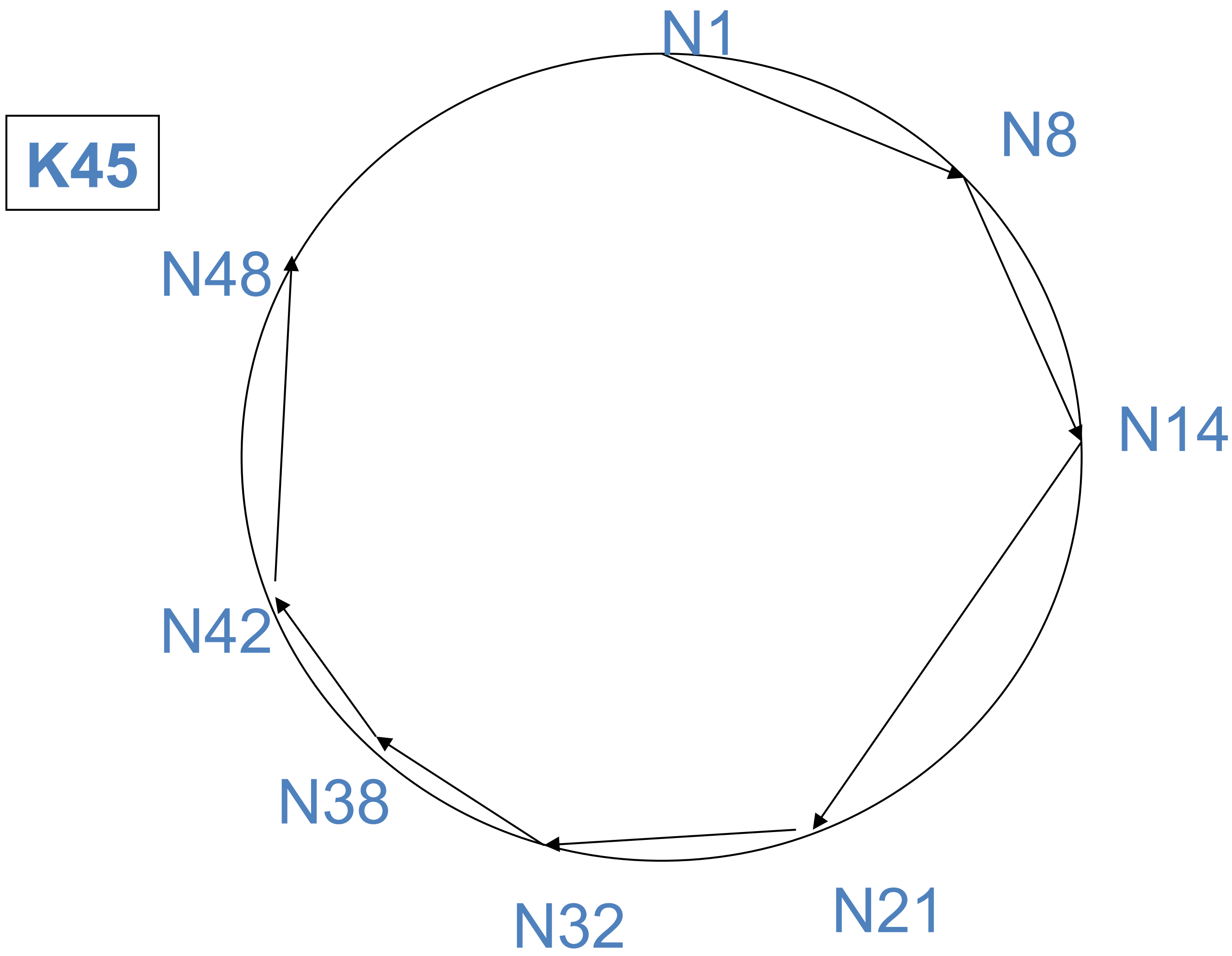


Theorem

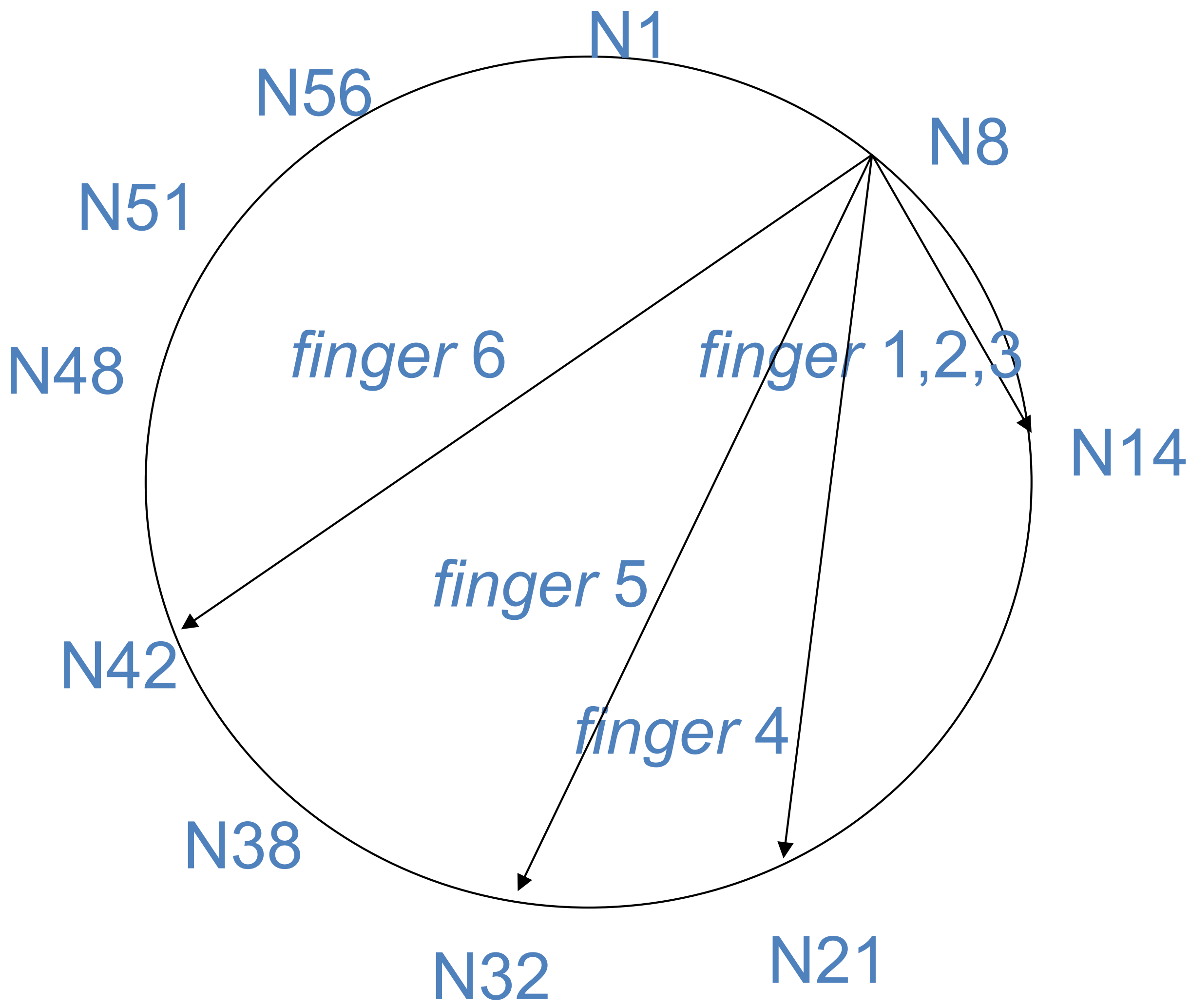
- For any set of N nodes and K keys, with *high probability*:
  1. Each node is responsible for at most (1+e)K/N keys.
  2. When an $(N+1)^{st}$ node joins or leaves the network, responsibility for O(K/N) keys changes hands.

  e = O(log N)

Simple Key Location Scheme



Scalable Lookup Scheme

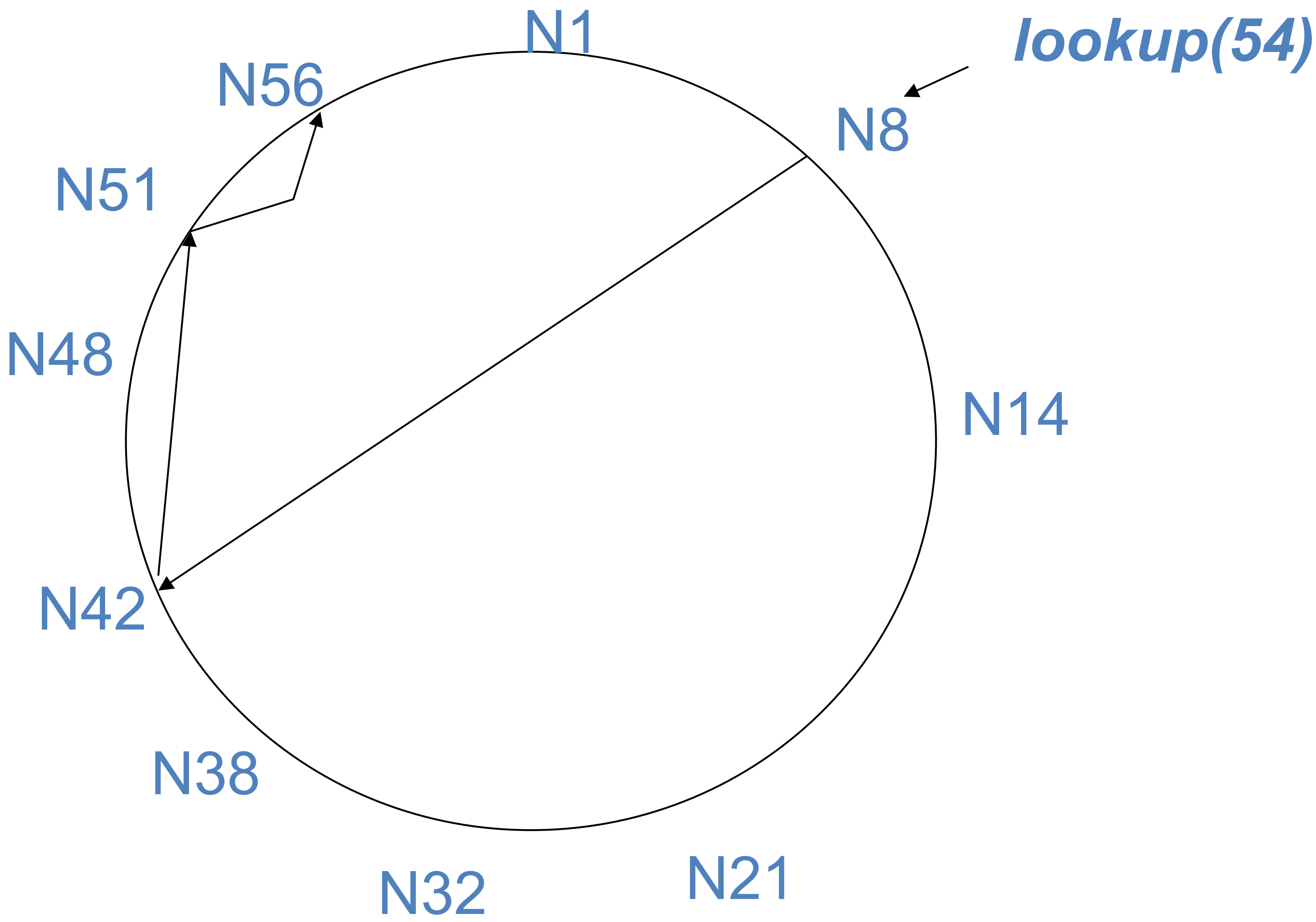


Finger Table for N8

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

finger [k] = first node that succeeds $(n+2^{k-1}) \bmod 2^m$

Lookup Using Finger Table



Scalable Lookup Scheme

// ask node n *to find the successor of* id
n.find_successor(id)
    if (id belongs to (n, *successor*])
      return *successor*;
    else
      n0 = closest preceding node(id);
      return n0.*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] belongs to (n, id))
          return *finger*[i];
    return n;

Scalable Lookup Scheme

- Each node forwards query at least halfway along distance remaining to the target
- Theorem: With high probability, the number of nodes that must be contacted to find a successor in a N-node network is O(log N)

Chord joins and departures

- Joining node gets an ID N and contacts node A
  - A searches for N's predecessor B
  - N becomes B's successor
  - N contacts B and gets its successor pointer and finger table, and updates the finger table values
  - N contacts its successor and inherits keys
- Departing node contacts its predecessor, notifies it of its departure, and sends it the new successor pointer

Chord features

- Correct in the face of routing failures since can correctly route with successor pointers
- Replication can easily reduce the possibility of loosing data
- Caching can easily place data along a likely search path for future queries
- Simple to understand

Classical & Future Issues

Classical Issues

- Large Scaling Search (Unstructured)
  – Gossip
- Data Replication and Maintenance
- Data Integrity
- Data Persistence
- Security / Anonymity
- Attacks
  – Sybil / Byzantin attack

Open Problems

- CLOUD Computing
  - Shared/Cooperative Work
- Deployment
- Resource Allocation
- Observation / Performance evaluation
- Security concerns / Building trust systems