Content-Addressable Network (CAN)

Reading:
A scalable content-addressable network. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker, ACM SIGCOMM Computer Communication Review.

A scalable content-addressable network

S Ratnasamy, P Francis, M Handley, R Karp… - Proceedings of the …, 2001 - dl.acm.org

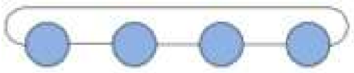
… In this paper, we introduce the concept of a Content-Addressable Network (CAN) as a … We call such indexing systems Content-Addressable Networks and, in this paper, propose a …

(as of 2023.03.16)

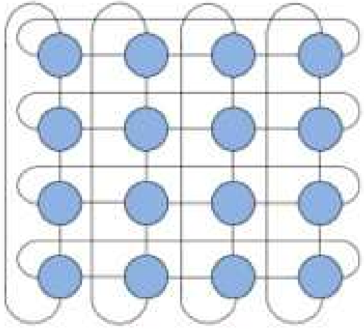
UC Berkeley

Content-Addressable Network (CAN)

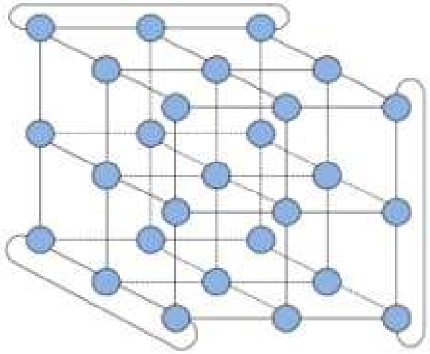
- Uses virtual *d*-dimensional coordinate space on a *d*-torus to store (*key*, *value*) pairs, and a uniform hash function to map keys to points in the *d*-torus
- The coordinate space is partitioned among peers to let each peer handle a zone.
  - (*key*, *value*) pairs are stored at the peer that owns the zone containing the corresponding point of the key.



1-D Torus (4-ary 1-cube)

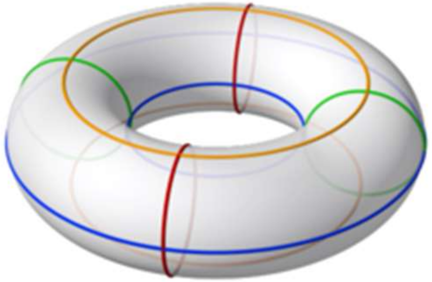


2D Torus (4-ary 2-cube)



3D Torus (3-ary 3-cube)

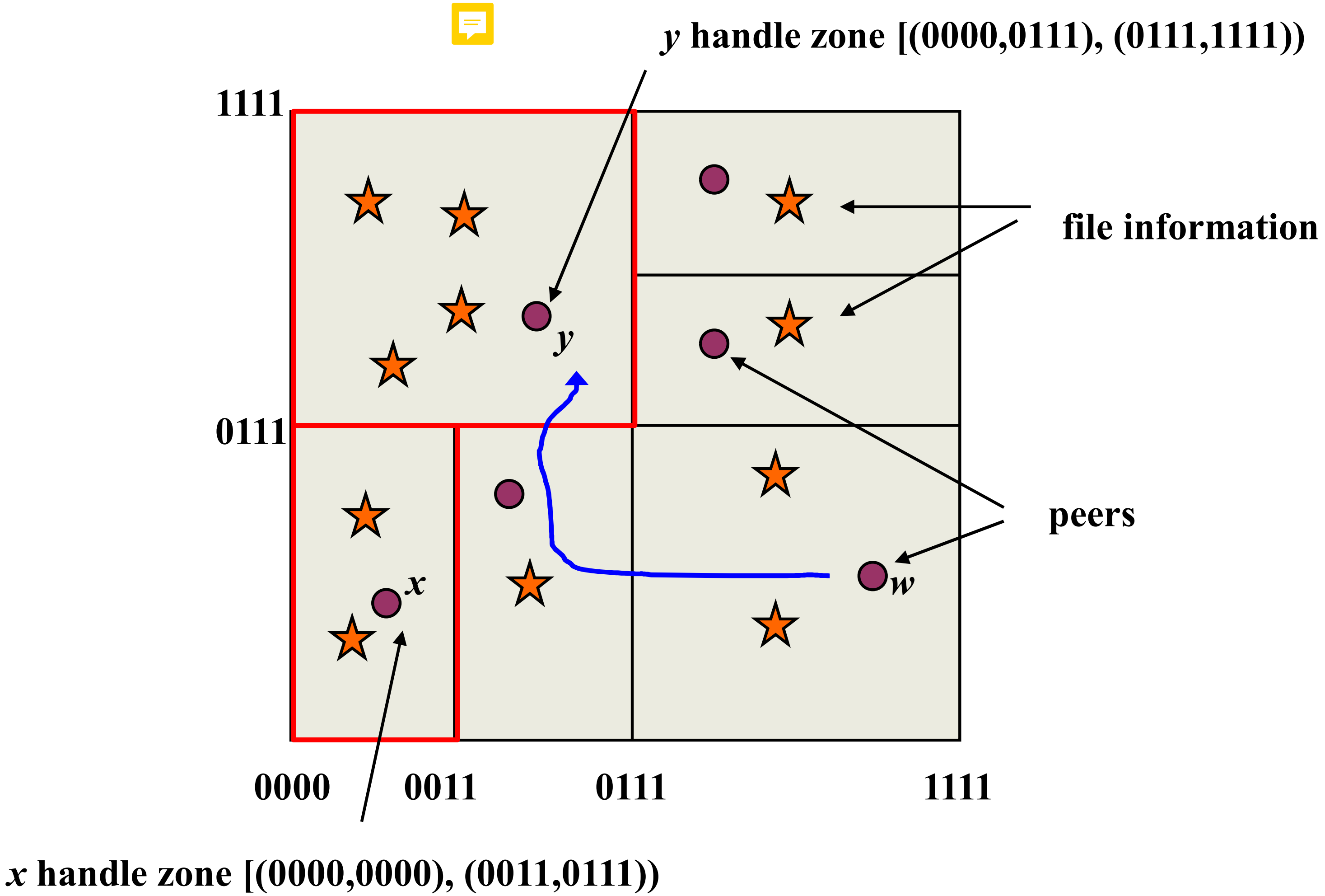
Torus



source: wiki

source: Ting Wang, et al. 2015

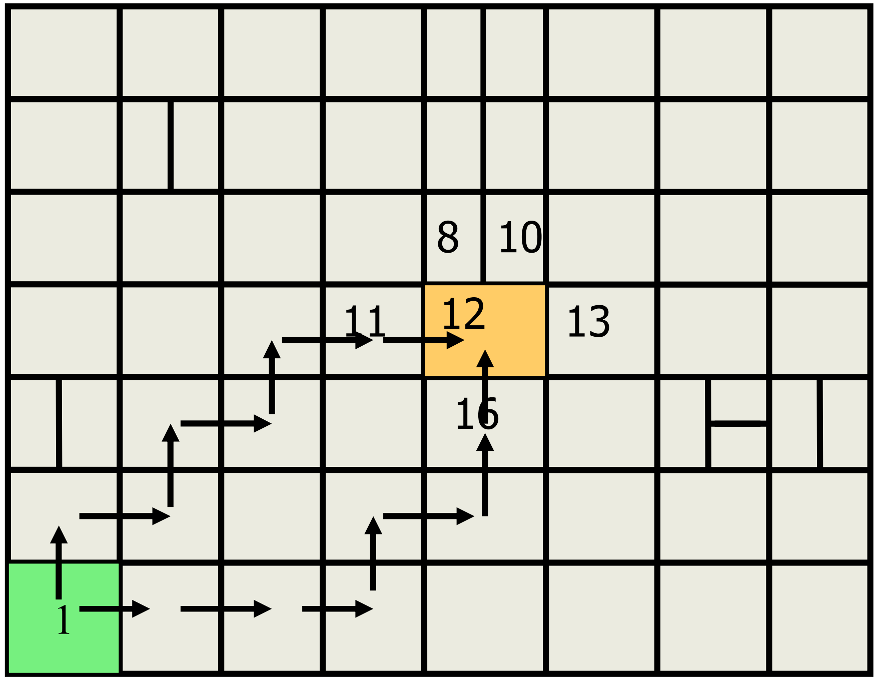
Example: 2-D CAN


y handle zone [(0000,0111), (0111,1111))
1111
0111
file information
y
peers
x
w
0000
0011
0111
1111
x handle zone [(0000,0000), (0011,0111))

Inserting and locating a key becomes a routing problem: how to find a peer handling a given key in the key space?

Routing in CAN

- Follow the straight path through the Cartesian space from source to destination
- Two nodes are neighbors if their responsible zones are adjacent
- Each node maintains a routing table that holds IP addresses and zones' coordinate of its neighbors in the space
- Routing can be done by greedily forwarding a message toward the destination.

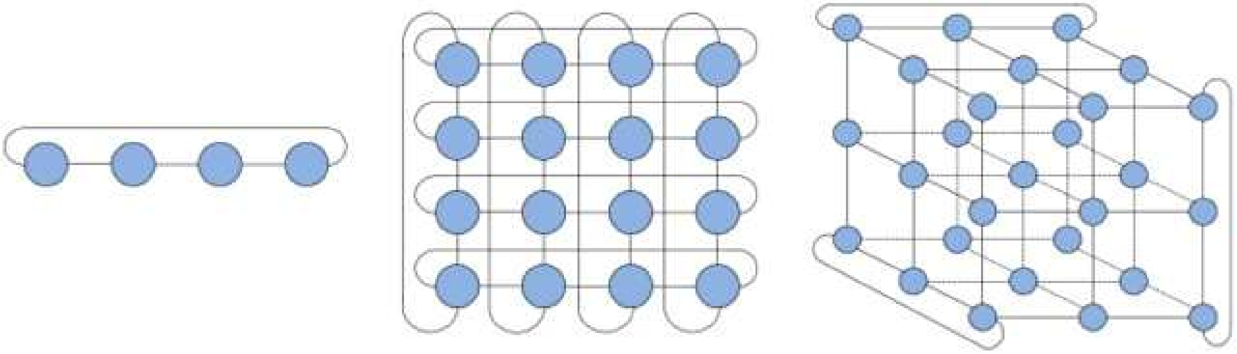


Neighbors of node 12: 8, 10, 11, 13, 16

Multiple routing paths exist between two nodes
What is the advantage?

Routing Complexity

- ❑ For the *d*-dimensional space equally partitioned into *n* nodes the average routing path is $d \times \frac{n^{1/d}}{4}$
- ❑ Individual nodes maintain 2*d* neighbors
- ❑ The path length growth proportionally to the $O(n^{1/d})$
- ❑ Many different routes between two points



1-D Torus (4-ary 1-cube)

2D Torus (4-ary 2-cube)

3D Torus (3-ary 3-cube)

source: Ting Wang, et al. 2015

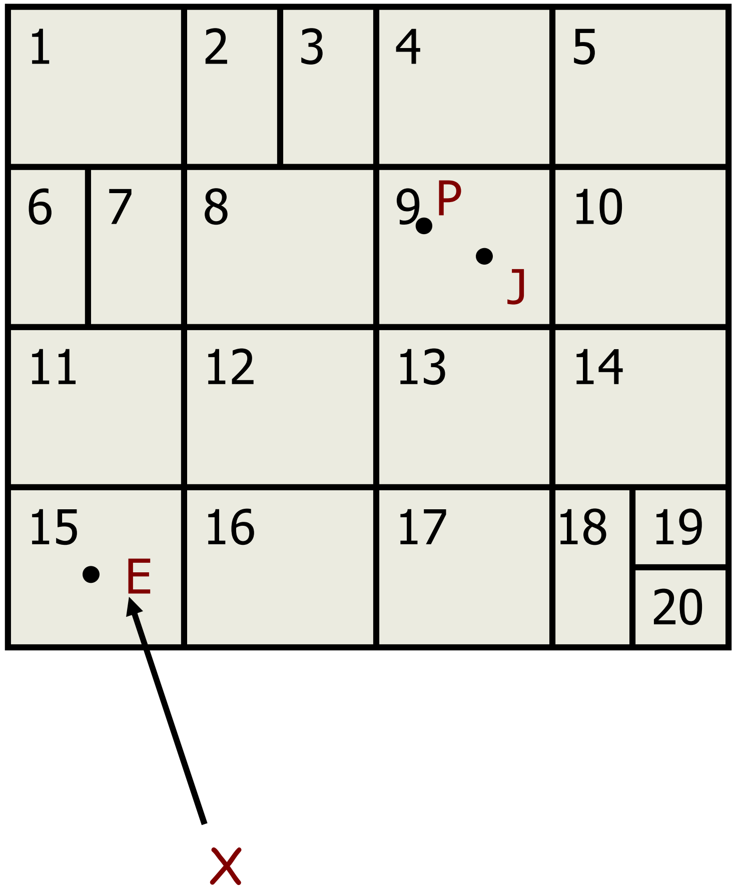
Basic Operations of CAN

- Inserting, updating, deleting of (key, value) pairs
  - easy
- Retrieving value associated with a given key
  - easy
- Adding new nodes to CAN
  - how?
- Handling departing nodes
  - same complexity as above
- Dealing with node failures
  - how?

CAN Overlay Construction

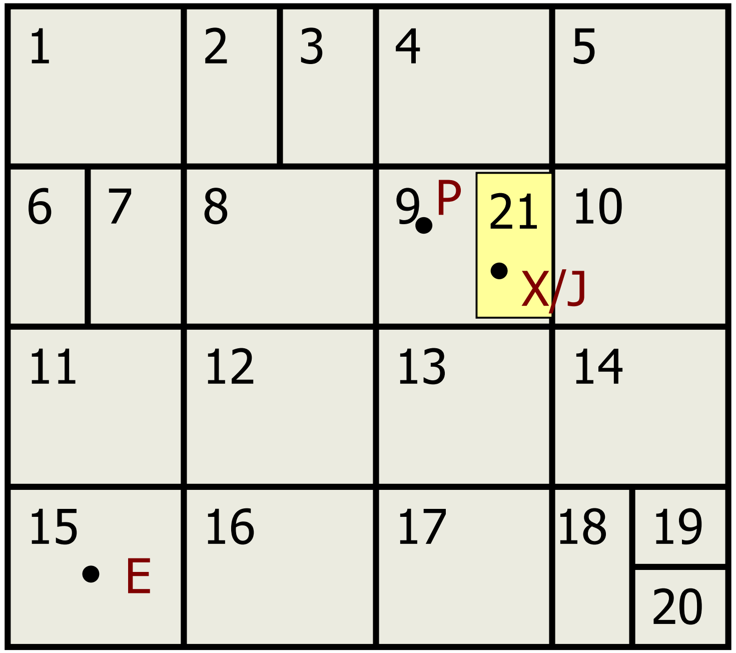
- New node is allocated its portion of the coordinate space in three steps:
  - Find a node already in the CAN – look up the CAN domain name in DNS
  - Pick a zone to join to and route request to its owner using CAN routing mechanisms
  - Split the zone between old and new node
  - The neighbors of the split zone must be notified so the routing can include the new node

Finding A Zone



- A new peer X finds a node E as a hookup node to the network
- Suppose X chooses the zone covering point **J** to split
- X sends a JOIN request message destined for point **J** (handled by P) via E using CAN routing mechanisms

Splitting A Zone

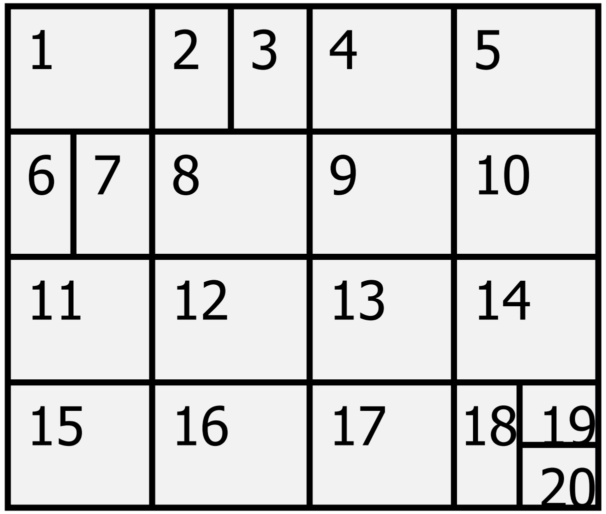


- P splits its zone in half and assigns one half to X (J), assuming some ordering of the dimensions, e.g. first dim-*x* then dim-y
- Transfer (key, value) pairs from the half of the zone to the new node

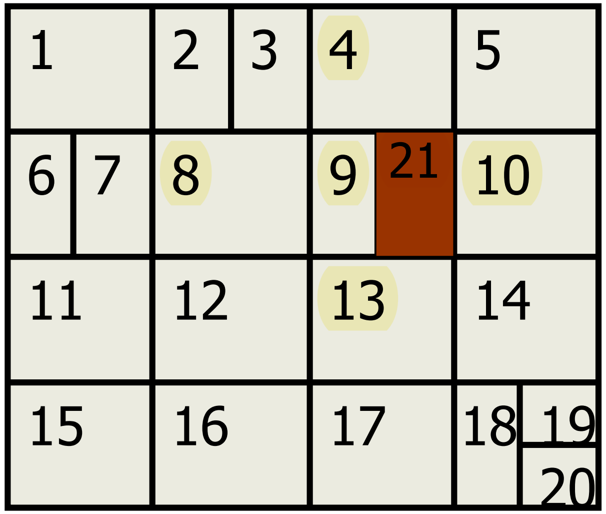
Joining the Routing

- The neighbors of the split zone must be notified so that routing can include the new node.
  - P sends its routing table to X (IPs and Zones of P’s neighbors)
  - Having P’s routing table, X may figure out its routing table.
  - Pre-computes its routing table.
  - Both P’s and X’s neighbors must be informed of this reallocation of space.
  - All of their neighbors update their own routing table.

Change of Routing Tables



Before 21 joins:
9: 4,8,10,13
4: 3,5,9,17
8: 2,3,7,9,12
10: 5,9,6,14
13: 9,12,14,17

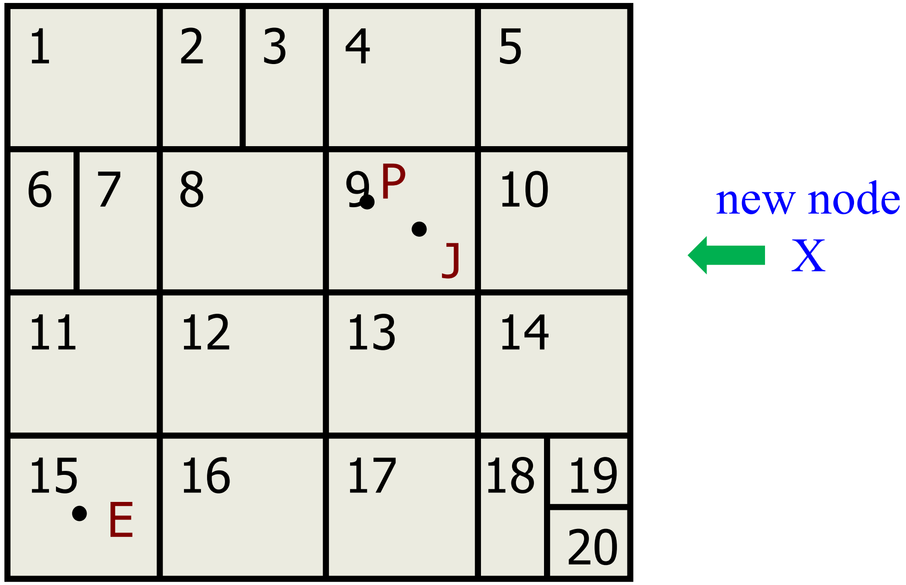


After 21 joins:
9: 4,8, 21,13
4: 3,5,9, 21,17
8: 2,3,7,9,12
10: 5, 21,6,14
13: 9,12,14,17, 21
21: 4,9,10,13

How many nodes need to change their routing tables? Why? Pro? Con?

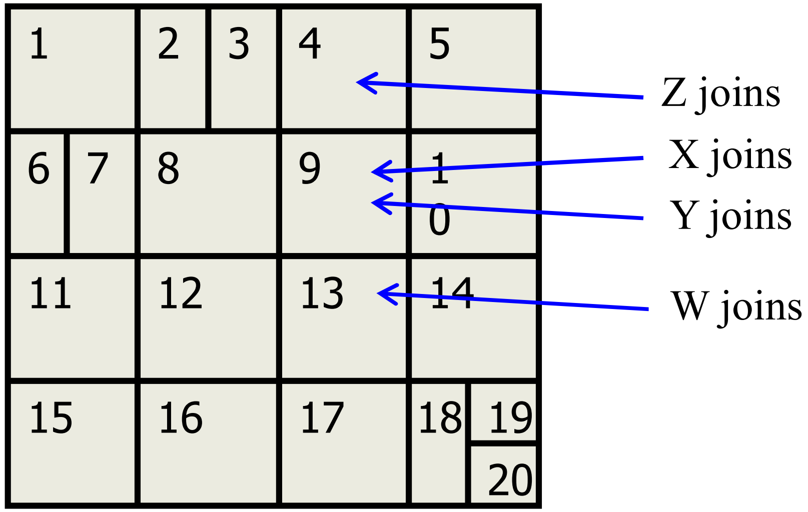
Zone Selection

- Can a new node select an arbitrary zone to split?
  - Flexibility allows load balance to be easily achieved
    - How?
      - ✓ Random selection



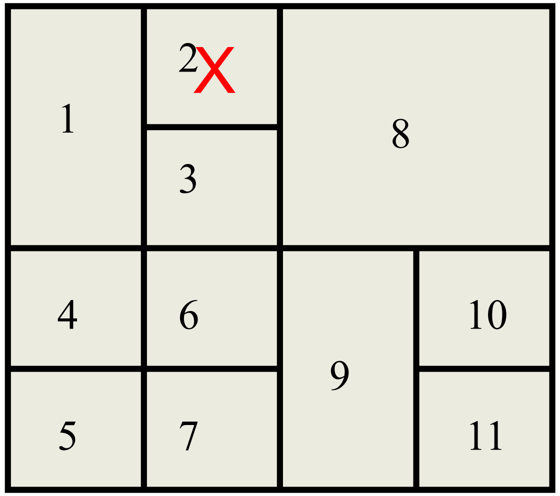
Concurrent Joining

- Can concurrent joining affect the correctness of the overlay structure?
- If so, how to deal with it?
  - The code can become very messy if one attempts to fix the routing tables in a single joining procedure.
  - Periodically maintenance



Normal Node Departure

- ❑ Explicit hand over of (key, value) pairs to one of the neighbors
- ❑ Zones merge and update of neighbor tables

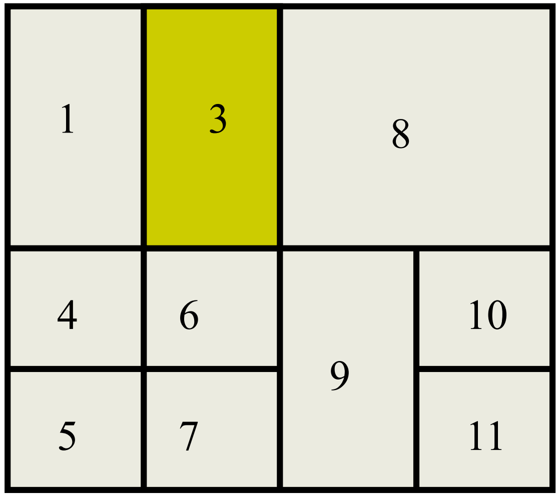


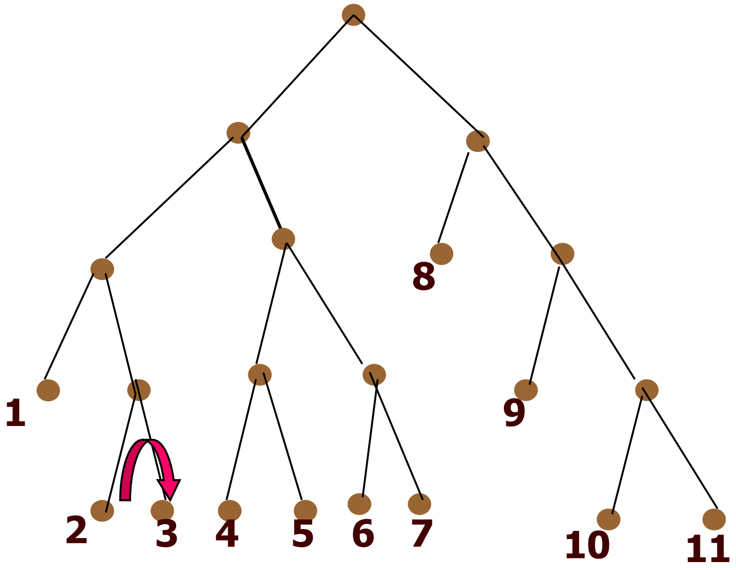
when zone 2 is to be merged, it should be merged into zone 3

What if zone 9 is to be merged?
Or zone 8?

Zone Reassignment

- Use of a binary partition tree to help zone reassignment
- Case 1: when zone 2 is to be merged
  - node 3 takes over zone 2.

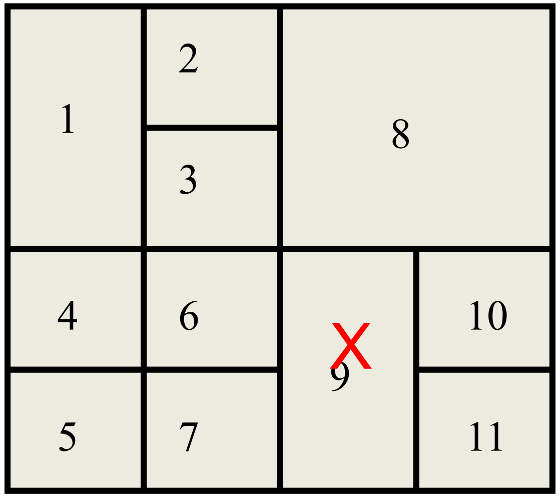


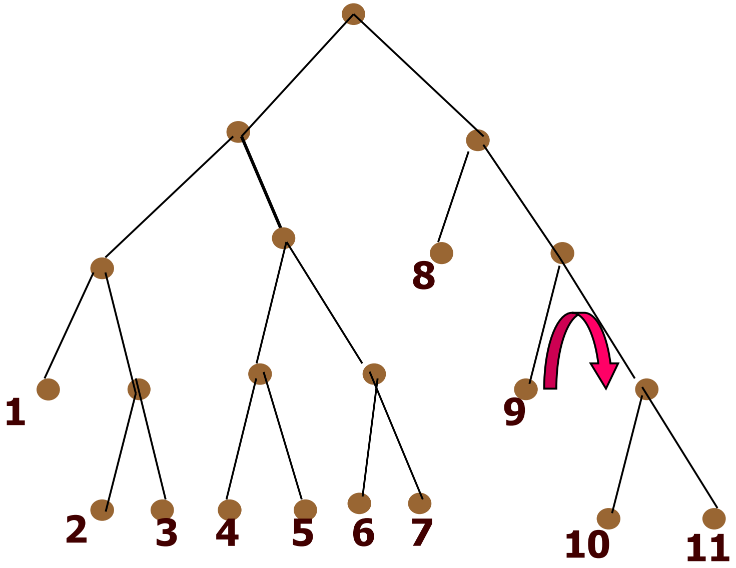


Binary Partition tree

Zone Reassignment (contd.)

- Case 2: when zone 9 is to be merged
  - Step1: Searching for sibling of 9, but fail
  - Step2: Use DFS until two sibling leaves are reached
  - Step3: Merge zone 10 with zone 11 and takeover by node 11

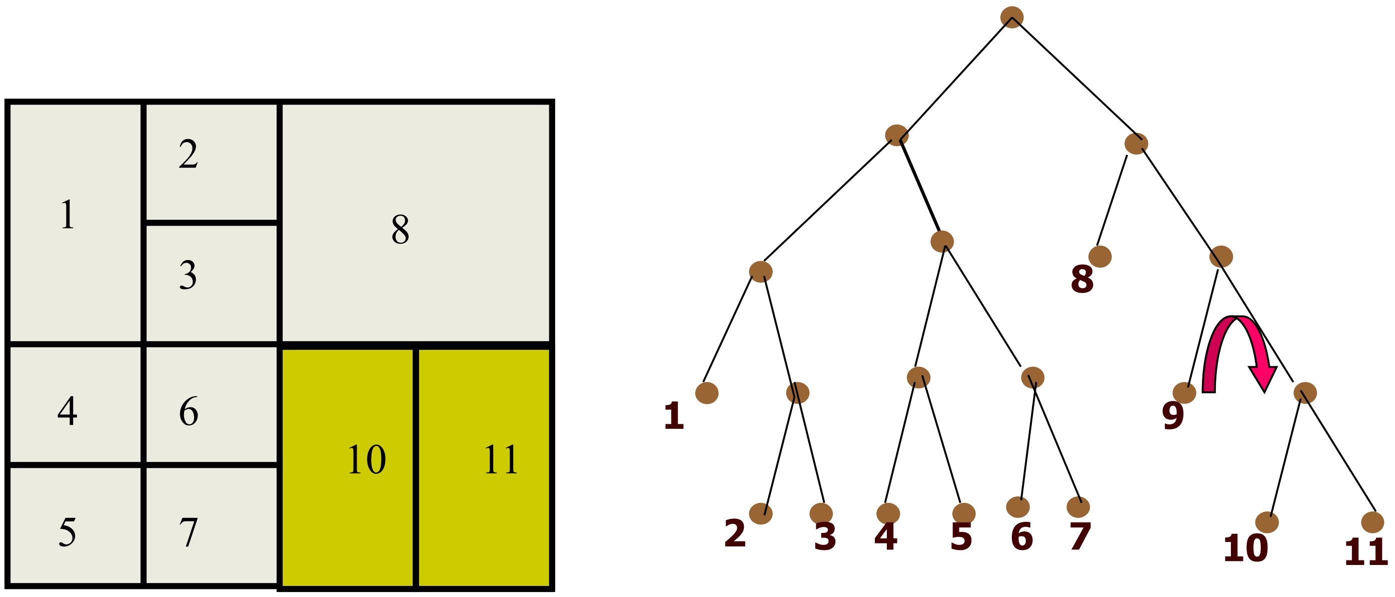




Binary Partition tree

Zone Reassignment (contd.)

- Case 2: when zone 9 is to be merged
  - Step4: node 11 now takes over zone x , DONE !!



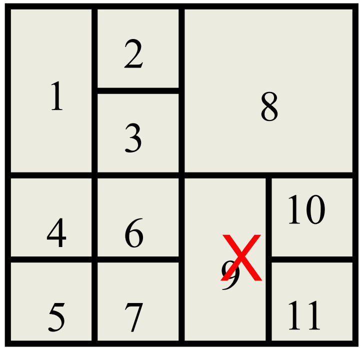
Binary Partition tree

How is this tree maintained in a decentralized manner?

Does each node need a complete knowledge of the tree?

Node Failures

- In P2P, it is unrealistic to assume that every node will leave normally, and thus failures must be taken care of.
- Failure detection
  - Periodical probe messages to neighboring nodes
  - If failure is detected, sends TAKEOVER message to all failed node's neighbors.
    - What if failure is detected simultaneously by more than one node?
      - Which node will be in charge of the failed node's zone?
- Failure handling can become very complex if multiple adjacent nodes fail simultaneously



Lost of Data

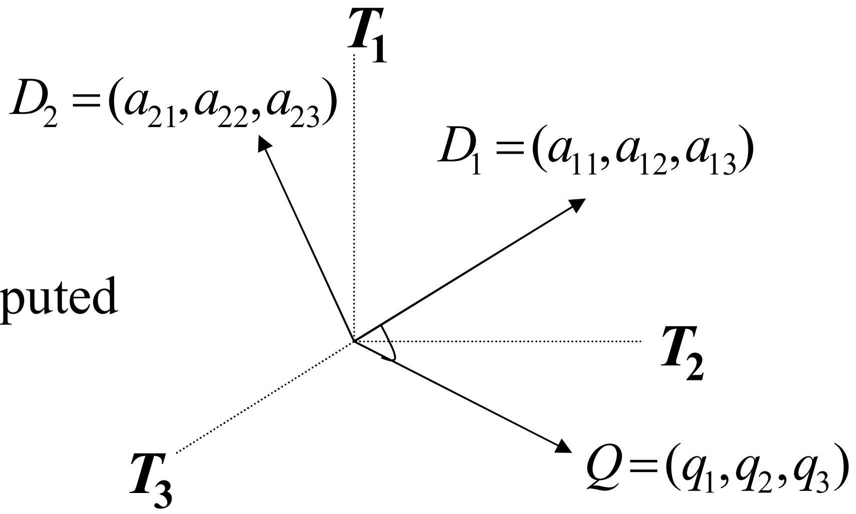
- Failures affect not only overlay structure, but also cause lost of data
- Solutions?
  - Redundancy (how?)

Application: Information Retrieval

- Vector Space Model (VSM)
  - Represents documents and queries as term vectors. Each element of the vector corresponds to the importance of a term in the document or query. The weight of an element is often computed using the statistical term frequency*inverse document frequency (TF*IDF) scheme.

$$D_i = (a_{i1}, a_{i2}, \ldots, a_{it})$$

$$Q_i = (q_{i1}, q_{i2}, \ldots, q_{it})$$

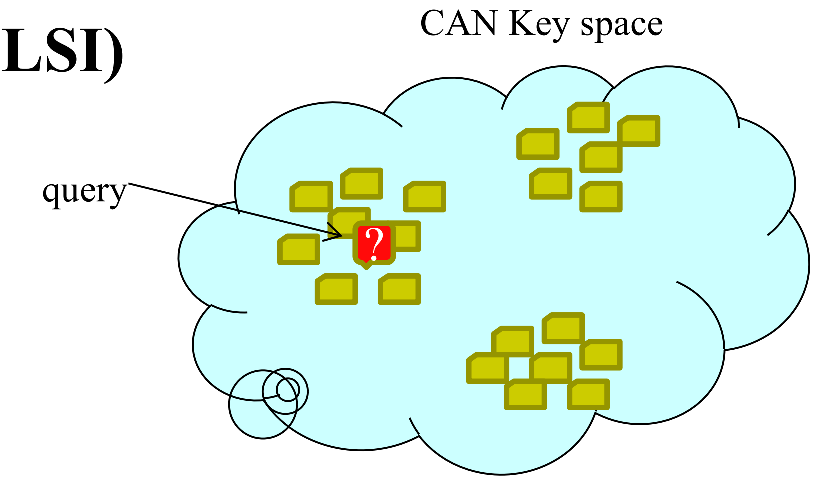

T_1
$D_2 = (a_{21}, a_{22}, a_{23})$
$D_1 = (a_{11}, a_{12}, a_{13})$
T_2
T_3
$Q = (q_1, q_2, q_3)$

  - Similarity between X and Y can be computed as inner product

$$COS(X, Y) = \frac{\sum_{i=1}^{t} x_i y_i}{\sqrt{\sum_{i=1}^{t} x_i^2} \cdot \sqrt{\sum_{i=1}^{t} y_i^2}}$$

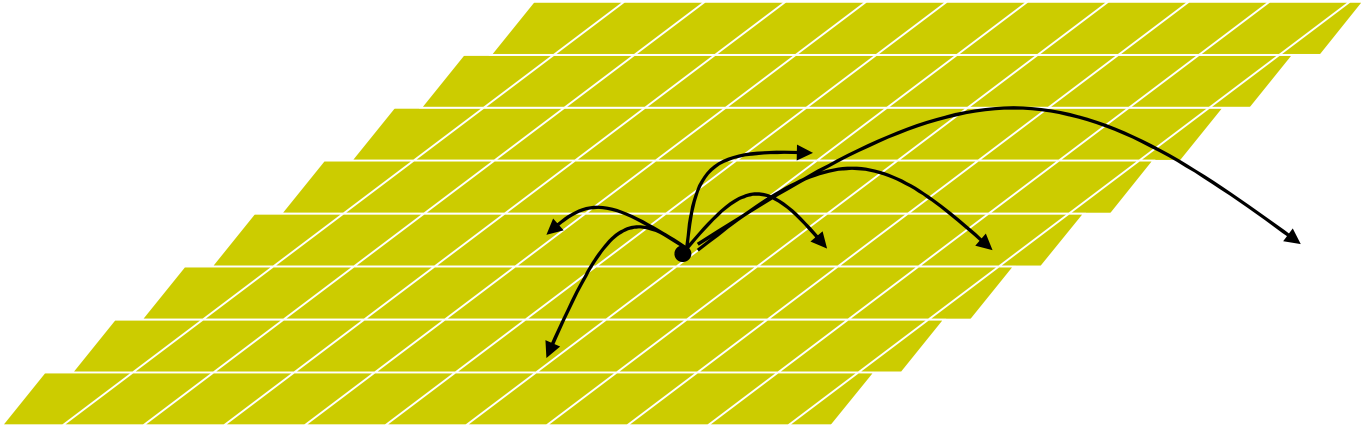
Information Retrieval in CAN

- ❑ CAN is especially suitable for IR because the vector space can be mapped directly to the CAN's key space such that two documents that are semantically close to each other will be mapped to two points that are geometrically close to each other in the key space.
  - ■ Documents can be easily retrieved by their similarity to a query.
- ❑ Potential problem: dimension explosion
  - ■ **Latent Semantic Indexing (LSI)**

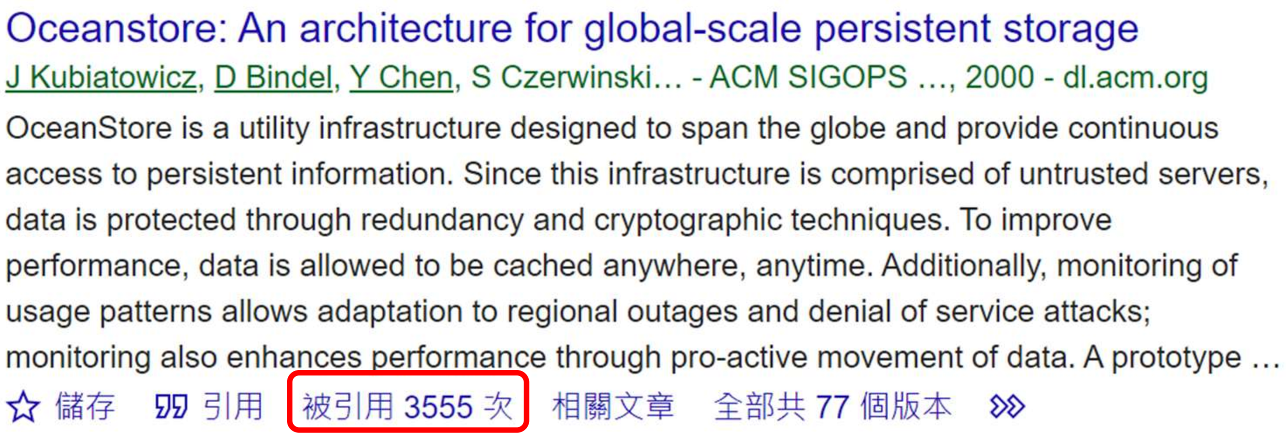


CAN Summary

- CAN has flexible routing selection.
- Provide tradeoff between routing state and routing path length by selecting the dimension *d*.
- Easy extension to information retrieval.
- Routing can be speeded up by building **hierarchical CAN**s.



OceanStore: A Global Scale Persistent Storage Utility Infrastructure



Oceanstore: An architecture for global-scale persistent storage

J Kubiatowicz, D Bindel, Y Chen, S Czerwinski… - ACM SIGOPS …, 2000 - dl.acm.org

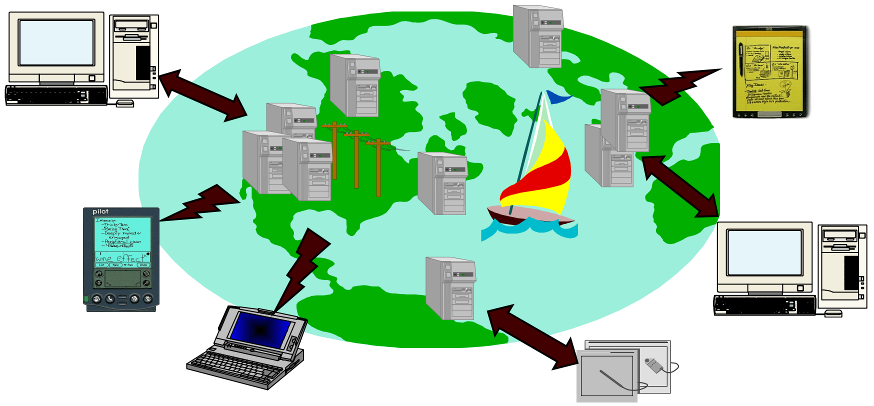
OceanStore is a utility infrastructure designed to span the globe and provide continuous access to persistent information. Since this infrastructure is comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere, anytime. Additionally, monitoring of usage patterns allows adaptation to regional outages and denial of service attacks; monitoring also enhances performance through pro-active movement of data. A prototype …

(as of 2023.04.26)

UC Berkeley

OceanStore: Motivation

- Ubiquitous Computing
  - Computing everywhere,
  - Connectivity everywhere
- But, are data just out there?
- OceanStore: An architecture for global-scale persistent storage



[Source: http://oceanstore.cs.berkeley.edu/]

Challenges

- Magnitude
  - Assume 10^{10} people in world, say 10,000 files/person (very conservative?), then 10^{14} files in total!
  - If 1MB/file, then 10^{20} size is needed
  - Surely, this must be maintained cooperatively by many ISPs.
- Persistent
  - Geographic independence for availability, durability, and freedom to adapt to circumstances

Challenges (cont.)

- Security
  - Encryption for privacy, signatures for authenticity, and Byzantine commitment for integrity
- Robust
  - Redundancy with continuous repair and redistribution for long-term durability
- Management
  - Automatic optimization, diagnosis and repair
- Anti-trust
  - Utility Infrastructure
  - Users pay monthly fee to access their data

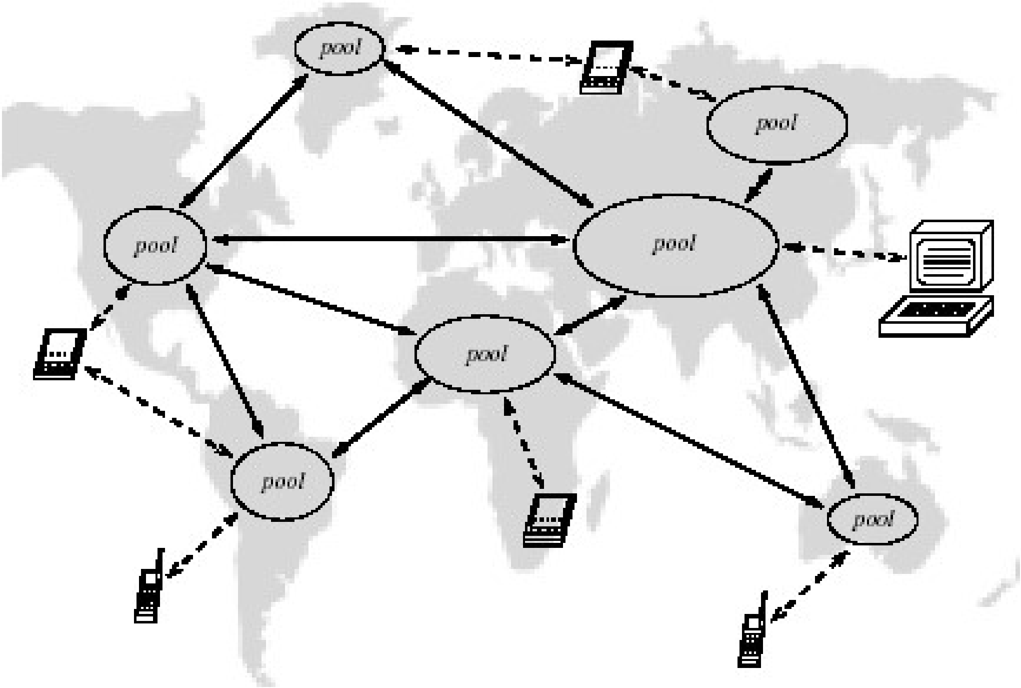
Design Goals

- Untrusted Infrastructure:
  - The OceanStore is comprised of untrusted components
  - Only ciphertext within the infrastructure
- Nomadic Data: data are allowed to flow freely
  - Promiscuous Caching: Data may be cached anywhere, anytime
    - continuous **introspective monitoring** is used to discover tacit relationships between objects.
  - Optimistic Concurrency via Conflict Resolution

Design Goals

- Untrusted Infrastructure:
  - The OceanStore is comprised of untrusted components
  - Only ciphertext within the infrastructure
- Nomadic Data: data are allowed to flow freely
  - Promiscuous Caching: Data may be cached anywhere, anytime
    - continuous **introspective monitoring** is used to discover tacit relationships between objects.
  - Optimistic Concurrency via Conflict Resolution

The OceanStore system



[Source: http://oceanstore.cs.berkeley.edu/]

- The core of the system is composed of a multitude of highly connected "**pools**", among which data is allowed to "flow" freely. Clients connect to one or more pools, perhaps intermittently.

Secure Naming

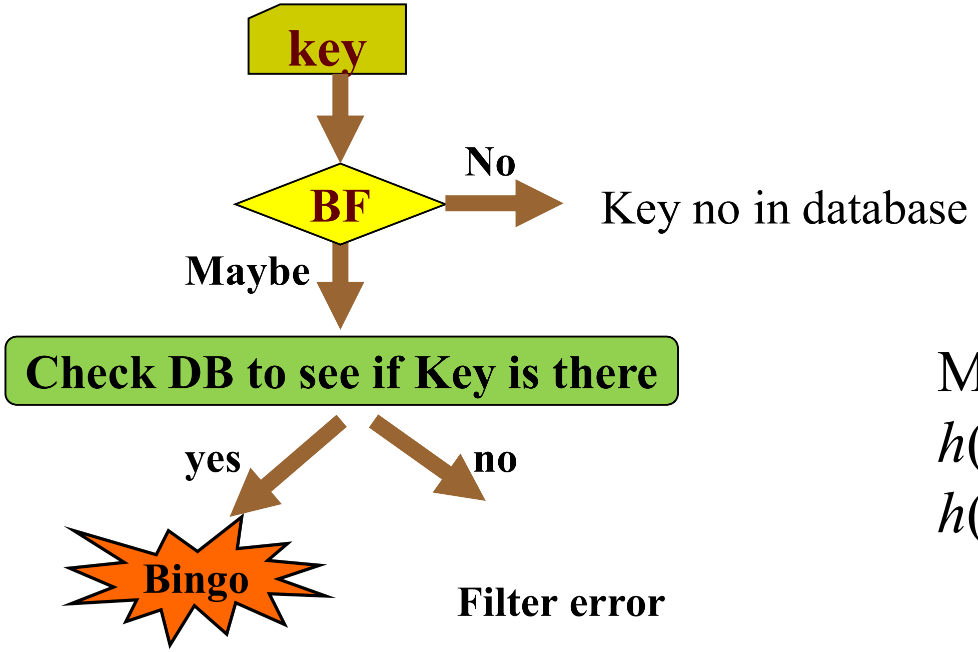
- Unique, location independent identifiers:
  - Every *version* of every unique entity has a permanent, ***Globally Unique ID (GUID)***
  - 160 bits SHA-1 hashes
    - 2^{80} names before name collision
- Naming hierarchy:
  - Users map from names to GUIDs via hierarchy of OceanStore objects (*ala SDSI)*
  - Require set of "root keys" to be acquired by user

Data Location and Routing

- Two-tiered approach
  - Attenuated bloom filters
    - fast, probabilistic
  - Wide-scale distributed data location
    - slower, reliable, hierarchical

Bloom Filter (BF)

A probabilistic algorithm to quickly test membership in a large set using multiple hash functions into a single array of bits.



Main Idea:

$$h(x) \neq h(y) \Rightarrow x \neq y$$

$$h(x) = h(y) \not\Rightarrow x = y$$

Bloom Filter Design

- Use an m bits vector, initialized to 0's, for the BF.
  - Larger m => fewer filter errors.
- Choose $h > 0$ hash functions: $f_1(), f_2(), \ldots, f_h()$.
- When key k is inserted into DB, set bits $f_1(k), f_2(k), \ldots$, and $f_h(k)$ in the BF to 1.
  - $f_1(k), f_2(k), \ldots, f_h(k)$ is the **signature** of key k.

EX: $m = 11$ (normally, m would be much much larger).
$h = 2$ (two hash functions).
$f_1(k) = k \bmod m$.
$f_2(k) = (2k) \bmod m$.

BF	0	0	1	0	1	0	1	0	0	1	0
	10	9	8	7	6	5	4	3	2	1	0

- k=15 inserted
- k=17 inserted

Example (contd.)

BF	0	0	1	0	1	0	1	0	0	1	0
	10	9	8	7	6	5	4	3	2	1	0

$f_1(k) = k \bmod m.$

$f_2(k) = (2k) \bmod m.$

DB has $k = 15$ and $k = 17$.

Search for k.

BF[$f_1(k)$] = 0 or BF[$f_2(k)$] = 0 => k not in DB.

BF[$f_1(k)$] = 1 and BF[$f_2(k)$] = 1 => k may be in DB.

$k = 25$ => not in DB

$k = 6$ => filter error.

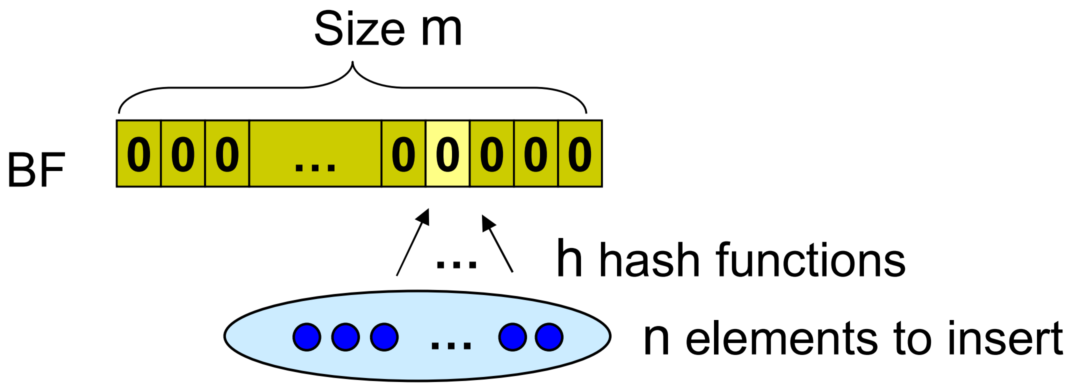
false positive

Note that Bloom filters can be aggregated (although the precision will be comprised)

Bloom Filter Design

- Choose *m* (filter size in bits).
  - *m* depends on the number of keys to be inserted.
- Pick *h* (number of hash functions).
  - *h* too small => probability of different keys having same signature is high.
  - *h* too large => filter filled with ones too soon.
- Select the *h* hash functions.
  - Hash functions should be relatively independent.

False Positive Rate of Bloom Filters



The probability that a certain bit is not set to one by a certain hash function in an insert operation is $(1-\frac{1}{m})$

The probability that a certain bit is not set by the h hash functions after n insert operations is $(1-\frac{1}{m})^{nh}$

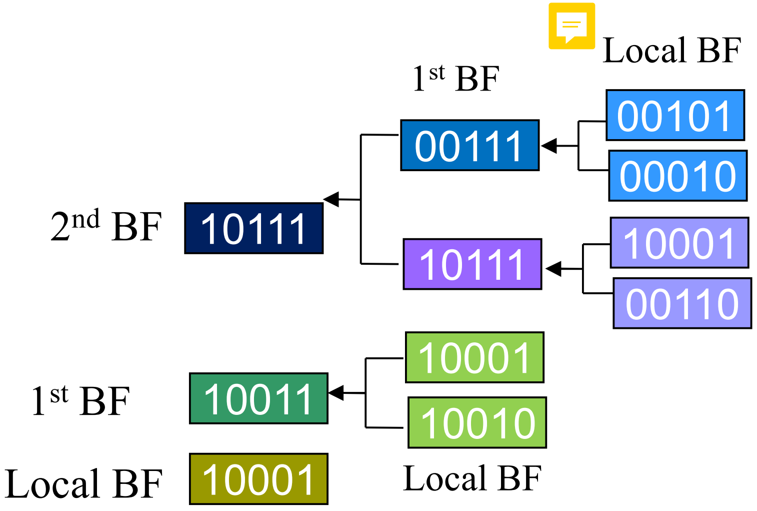
The probability that the bit is set is $1-(1-\frac{1}{m})^{nh}$

False positive occurs only when all corresponding bits set by the h functions are all 1, which has the probability

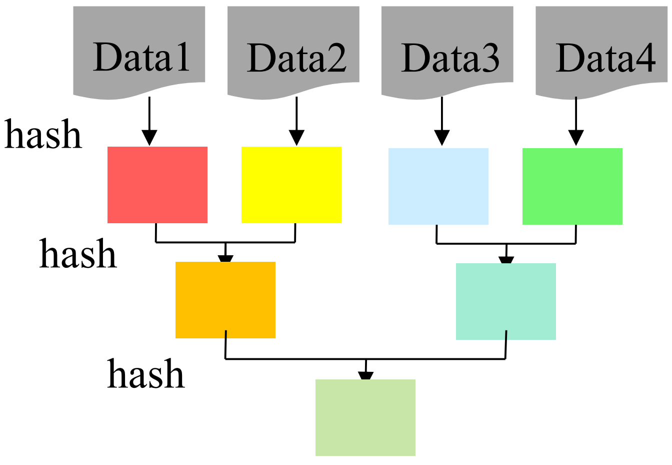
$$(1-(1-\frac{1}{m})^{nh})^{h} \approx 1-(1-e^{-nh/m})^{h}$$

Attenuated Bloom Filters

- An attenuated bloom filter of depth *D* can be viewed as an array of *D* normal Bloom filters
  - The first Bloom filter is a record of the objects contained locally on the current node.
  - The i^{th} Bloom filter is the union of all of the Bloom filters for all of the nodes at distance *i* through any path from the current node.



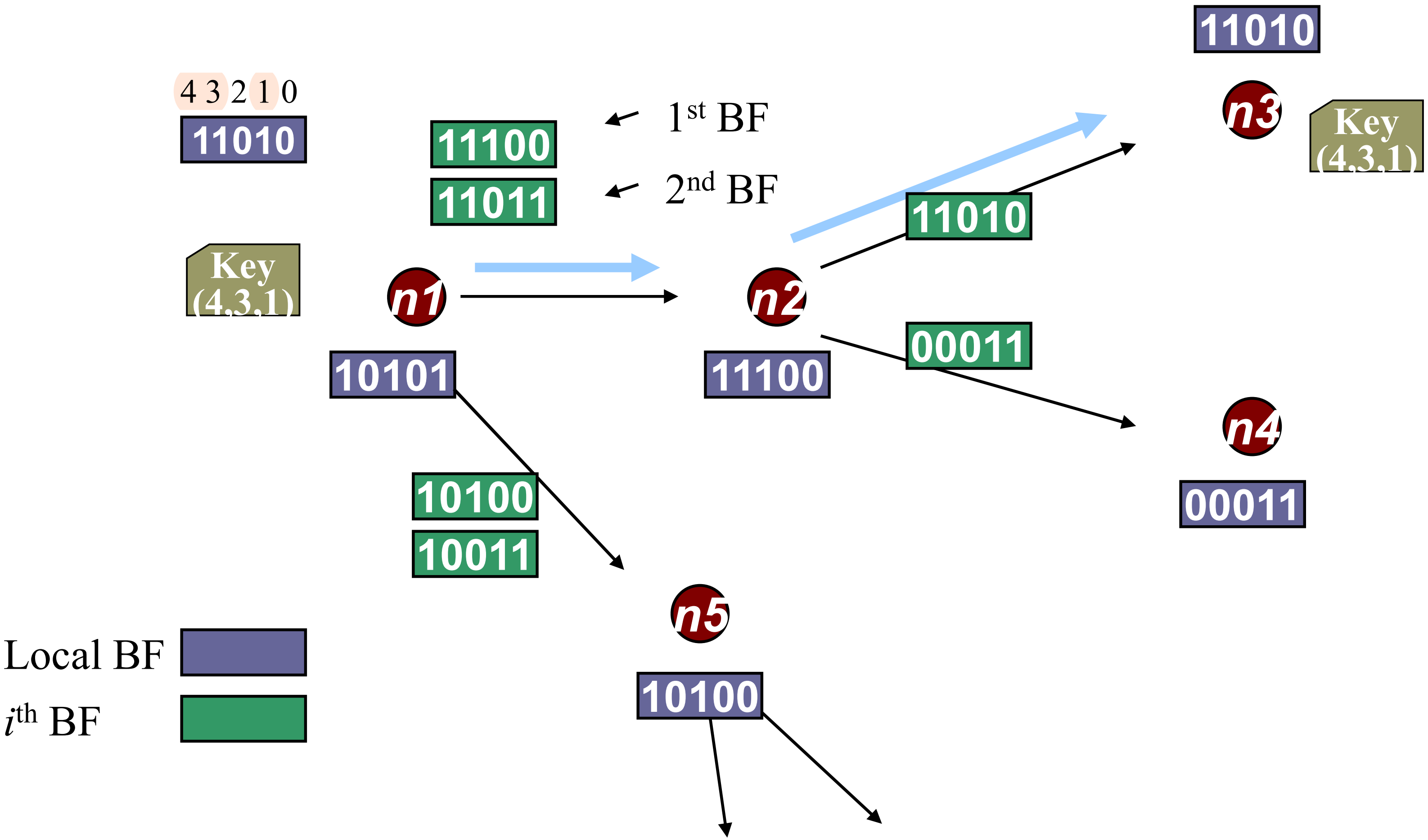
Cf. Merkle tree



Hash of the 4 data blocks

Probabilistic Query Process in OceanStore

Query is routed along the edges whose filters indicate the presence of the object at the smallest distance.



Tapestry

- A prototype of a decentralized, fault-tolerant, adaptive overlay infrastructure
- Network substrate of OceanStore (i.e., its lookup protocol)
  - Routing: Suffix-based hypercube
    Similar to Plaxton, Rajamaran, Richa (SPAA97)
  - Decentralized location:
    Virtual hierarchy per object with cached location references

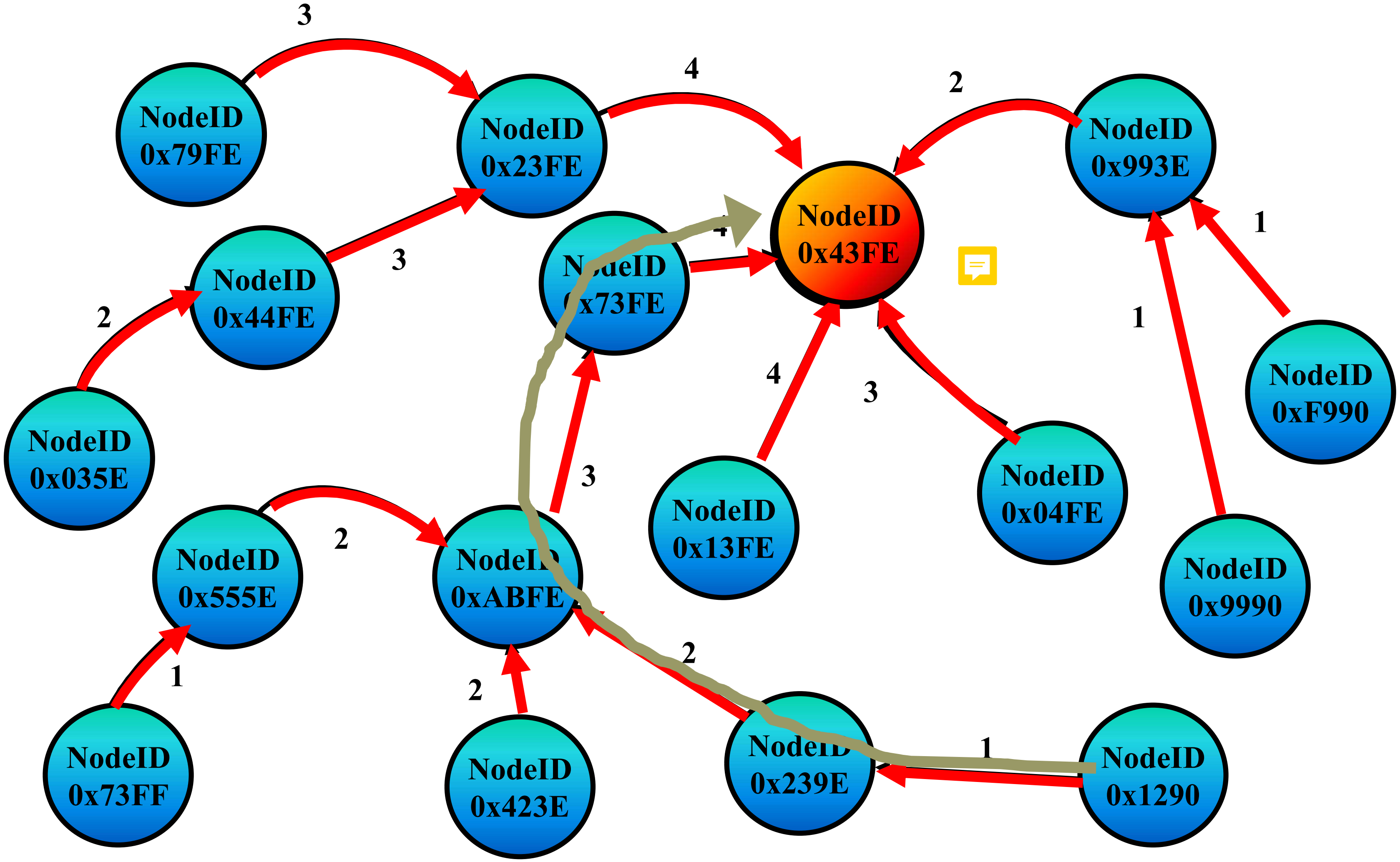
Ref.
Tapestry: An infrastructure for fault-tolerant wide-area location and routing. B.Y. Zhao, et al., Report No. UCB/CSD-01-1141, 2001.

(UC Berkeley)

Routing and Location in Tapestry

- Object location:
  - map GUIDs to root node Ids
    - Each object has its own hierarchy rooted at *Root*
    - Root responsible for storing object's location
- Suffix-match routing from A to B
  - At the h^{th} hop, arrive at the node that shares suffix with B of length *h* digits
  - Example: 5324 routes to **0729** via 5324 → 734**9** → 34**29** → 4**729** → **0729**
- What should a node's routing table be maintained?

Basic Plaxton Mesh
Incremental suffix-based routing



Neighbor Table

Neighbor table at $(3120)_4$

digit to match

level

	0	1	2	3
3	0120	1120	2120	3120
2	x020	x120	x220	x320
1	xx00	xx10	xx20	xx30
0	xxx0	xxx1	xxx2	xxx3

- How does a new node construct its neighbor table?

Surrogate Routing

- ❑ Neighbor table will have holes (can’t find a proper entry)
    - Must be able to find unique root for every object
    - Tapestry’s solution: try next highest.

Neighbor table at $(3120)_4$

	0	1	2	3
3	0120	1120	2120	3120
2	x020	x120	x220	x320
1	xx00	xx10	xx20	xx30
0	xxx0	xxx1	xxx2	xxx3

If there is no one to fill 2120, then use 3120 (or 1120) as substitute.

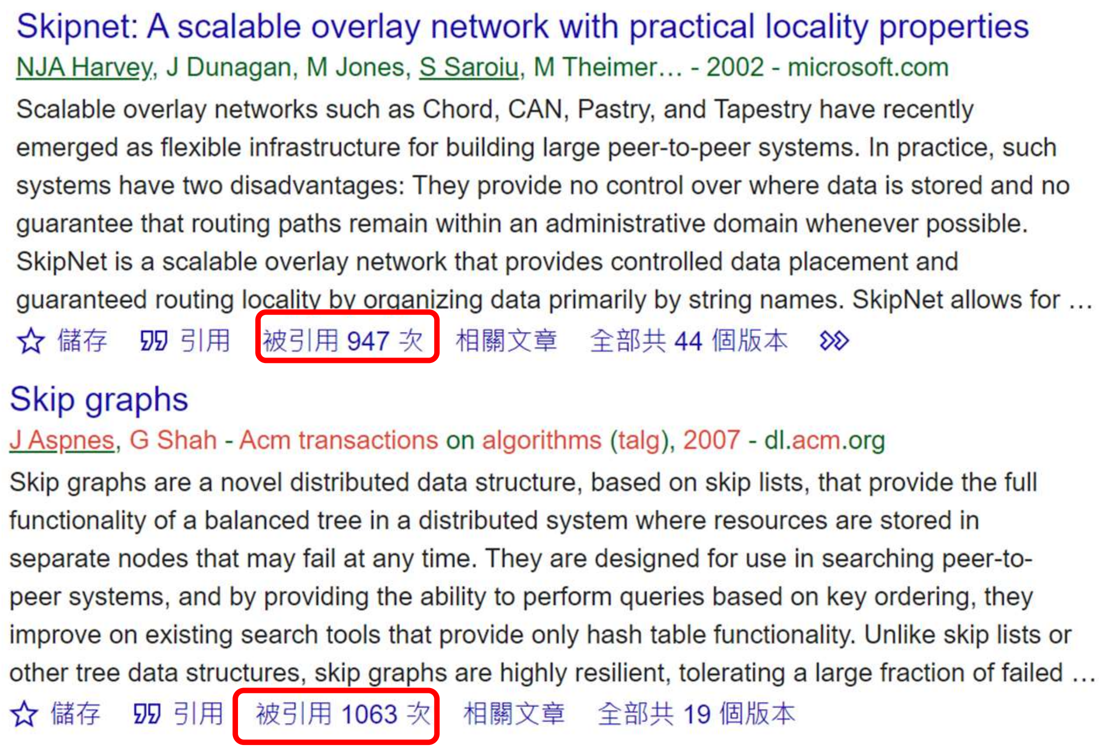
What if there is no x220?

Node Joins

- A little bit complicated
  - Try to think about the problem by yourself

Skip Graph & SkipNet: Decouple Object ID and Node ID

Structured, but not DHTs



Skipnet: A scalable overlay network with practical locality properties

NJA Harvey, J Dunagan, M Jones, S Saroiu, M Theimer… - 2002 - microsoft.com

Scalable overlay networks such as Chord, CAN, Pastry, and Tapestry have recently emerged as flexible infrastructure for building large peer-to-peer systems. In practice, such systems have two disadvantages: They provide no control over where data is stored and no guarantee that routing paths remain within an administrative domain whenever possible. SkipNet is a scalable overlay network that provides controlled data placement and guaranteed routing locality by organizing data primarily by string names. SkipNet allows for …

☆ 儲存 ❞ 引用 被引用 947 次 相關文章 全部共 44 個版本 »

Skip graphs

J Aspnes, G Shah - Acm transactions on algorithms (talg), 2007 - dl.acm.org

Skip graphs are a novel distributed data structure, based on skip lists, that provide the full functionality of a balanced tree in a distributed system where resources are stored in separate nodes that may fail at any time. They are designed for use in searching peer-to-peer systems, and by providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash table functionality. Unlike skip lists or other tree data structures, skip graphs are highly resilient, tolerating a large fraction of failed …

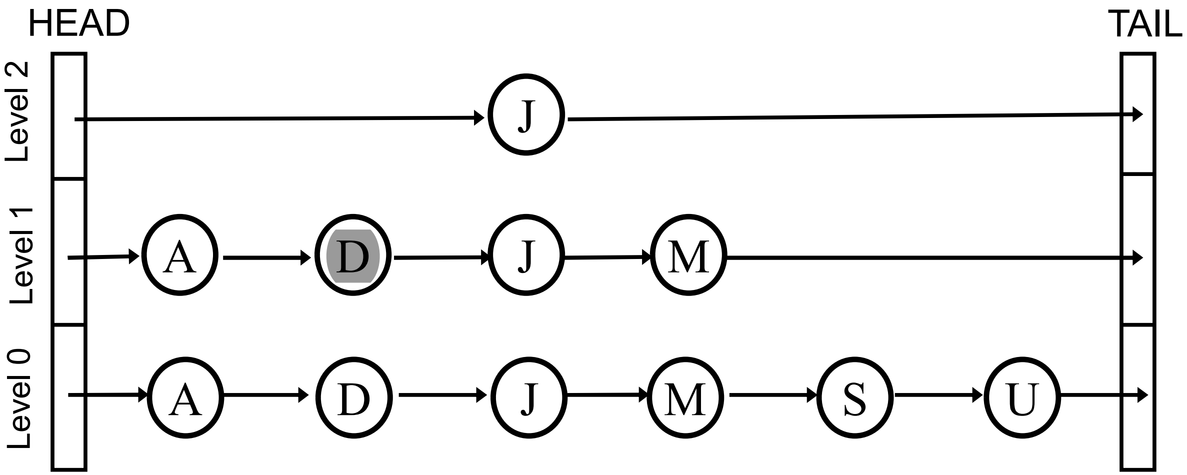
☆ 儲存 ❞ 引用 被引用 1063 次 相關文章 全部共 19 個版本

(as of 2023.04.26)

Motivations

- Problems with DHTs: no locality
  - nodes that are close to one another may be far apart in the overlays
  - adjacent data are indexed by "random" nodes in the overly
  - thus no support of complex searches like
    - search for objects with names "wild*.jpg"
    - keys < *x*,

Skip Lists [Pugh '90]



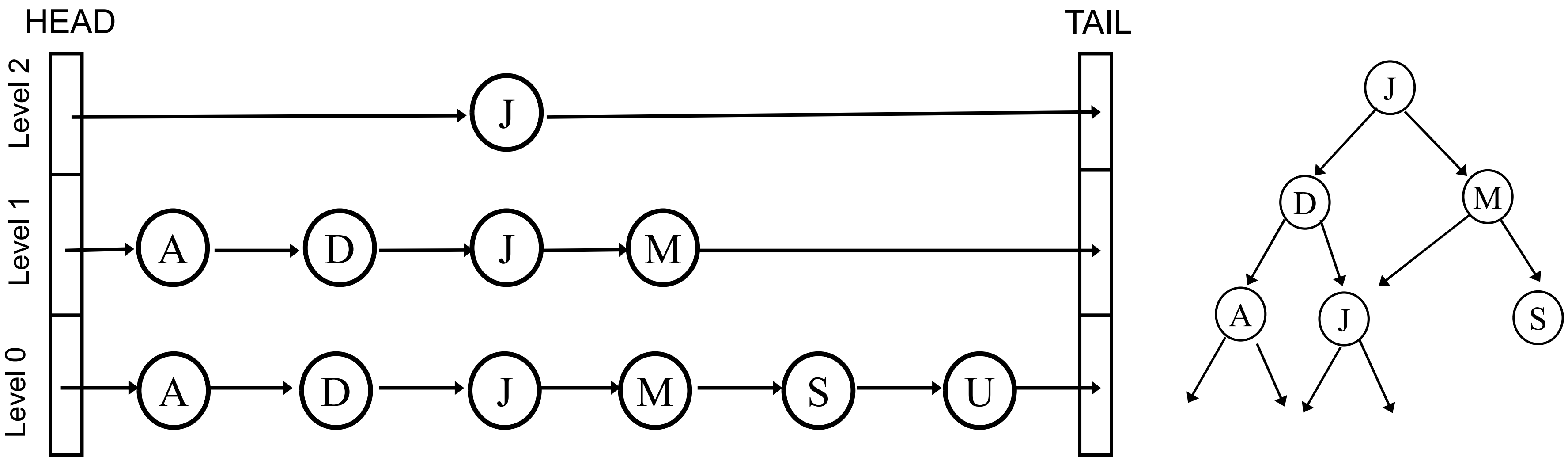
How to conduct search?

- ➢ Each level is a linked list
- ➢ Level 0 contains all nodes in increasing order by key
- ➢ For each *i* greater than 0, each node in level *i* – 1 appears in level *i* independently with some fixed probability *p*.
- ➢ When *p*=1/2, with probability $1/2^h$ a node has height *h*.
- ➢ The data structure supports an average search time of O(log *N*) on *N* elements.

Skip lists: a probabilistic alternative to balanced trees, William Pugh, Communications of the ACM, Vol. 33, Issue 6, June 1990

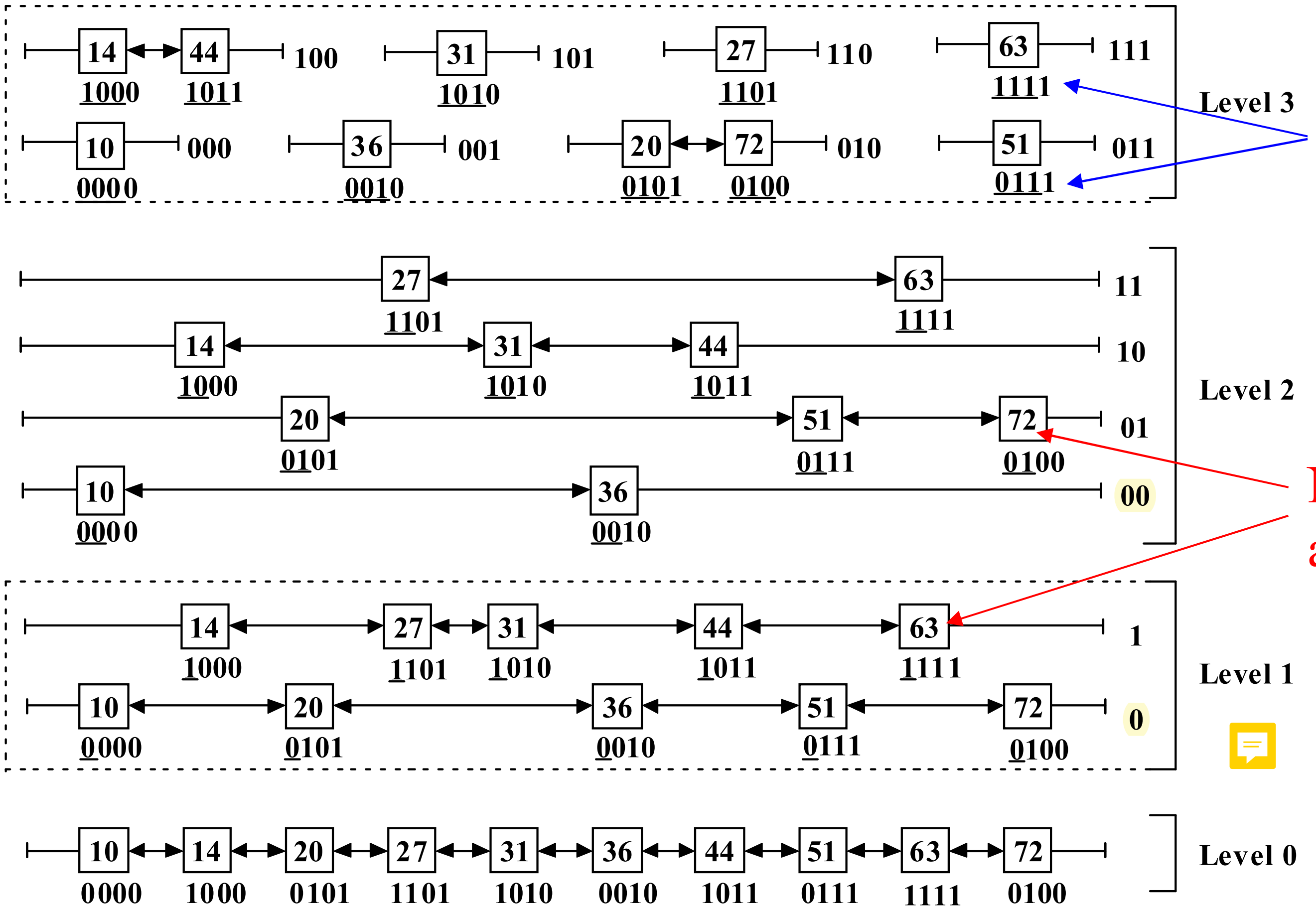
Skip Lists (2)

- How to adopt the structure in P2P?



- Skip List can be viewed as a randomized balanced tree
- Each node appears with some probability in higher levels.
- For P2P, nodes should all appear in the higher levels, as search can be initiated from everywhere.

Skip Graph



Each node obtains a random ID. Its links at level *i* pointing to nodes with matching prefix of length *i*.

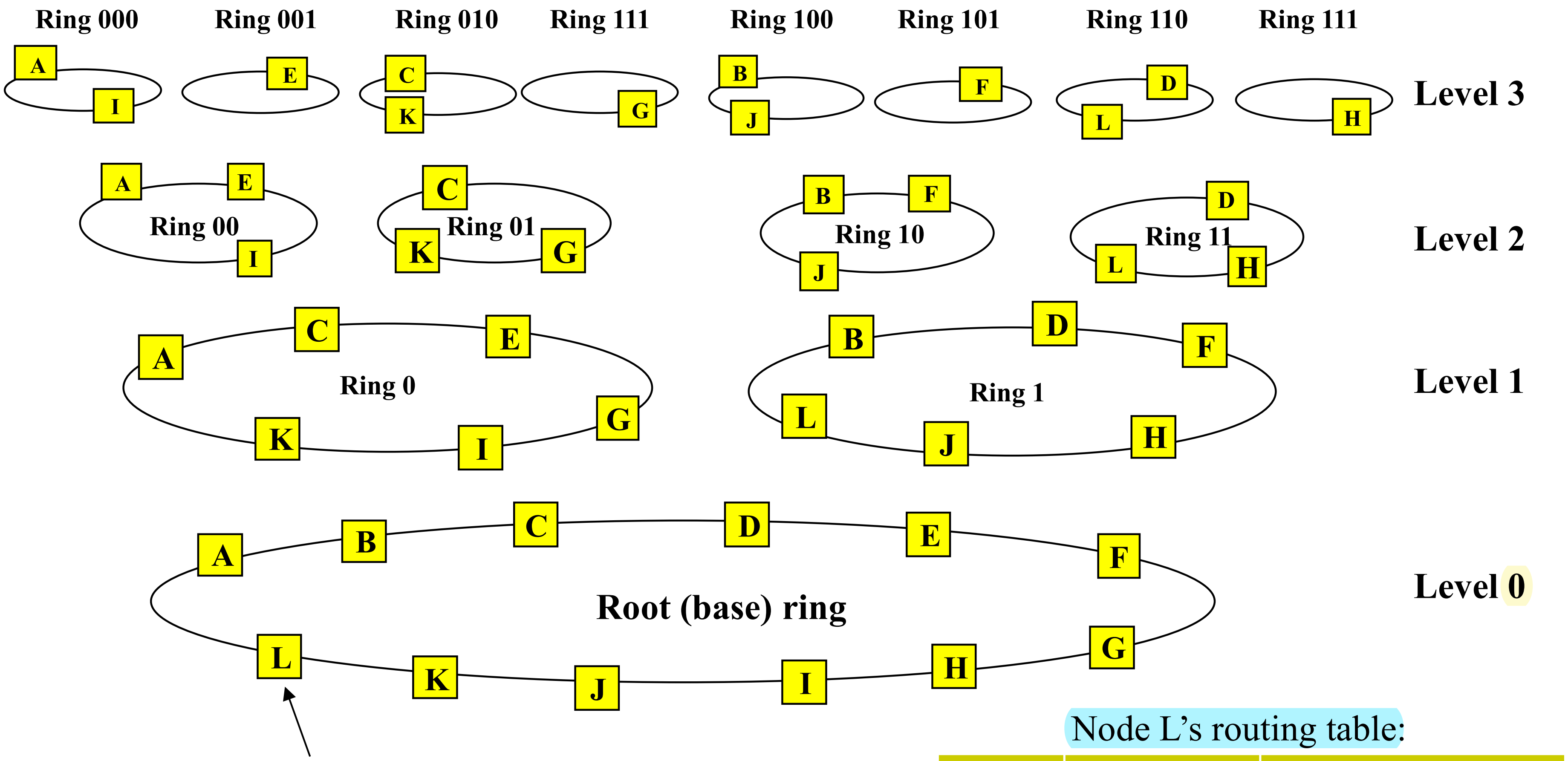
Node IDs and their keys are decoupled!

Skip Graph and SkipNet are structurally the same and independently developed based on Skip lists.

SkipNet Address Spaces

- Each node has a (string) name ID and a numeric ID:
  - Name IDs may bear some semantics (e.g., node's domain name or numeric keys) (and thus can be meanfully 'ordered')
  - Numeric IDs are simply random (but unique)
    - can be obtained by hashing a node's name
    - uniform distribution ensures routing to be done in O(log *N*) time and routing table size of O(log *N*) on average

SkipNet Structure



Node L (‘L’ is its node name, its numeric id is 110*)

nodes are ordered by name ID along each ring

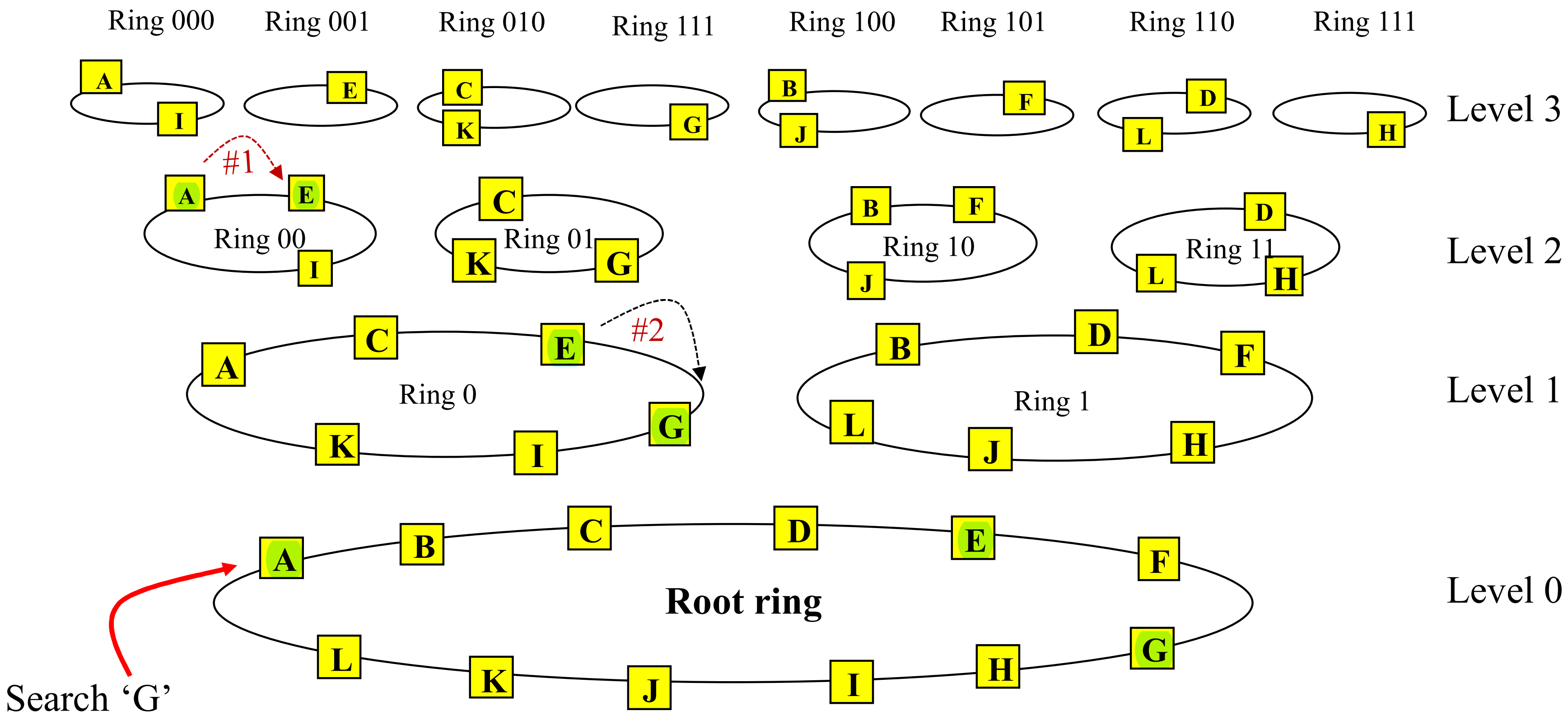
Node L’s routing table:

Level	clockwise	counterclockwise
3	D	D
2	D	H
1	B	J
0	A	K

Routing in SkipNet: by Name ID

- At each node, a message will be routed along (either direction of the rings) the highest-level pointer that does not point past the destination value until the node whose name ID is closest to the destination is reached.
  - If name IDs are strings with alphabetically order, then routing by name ID traverses only nodes whose name IDs share a non-decreasing prefix with the destination ID
    - aa* → aab* → aabcd* → aabcdgh* → …

Search by Name ID (Ex.)

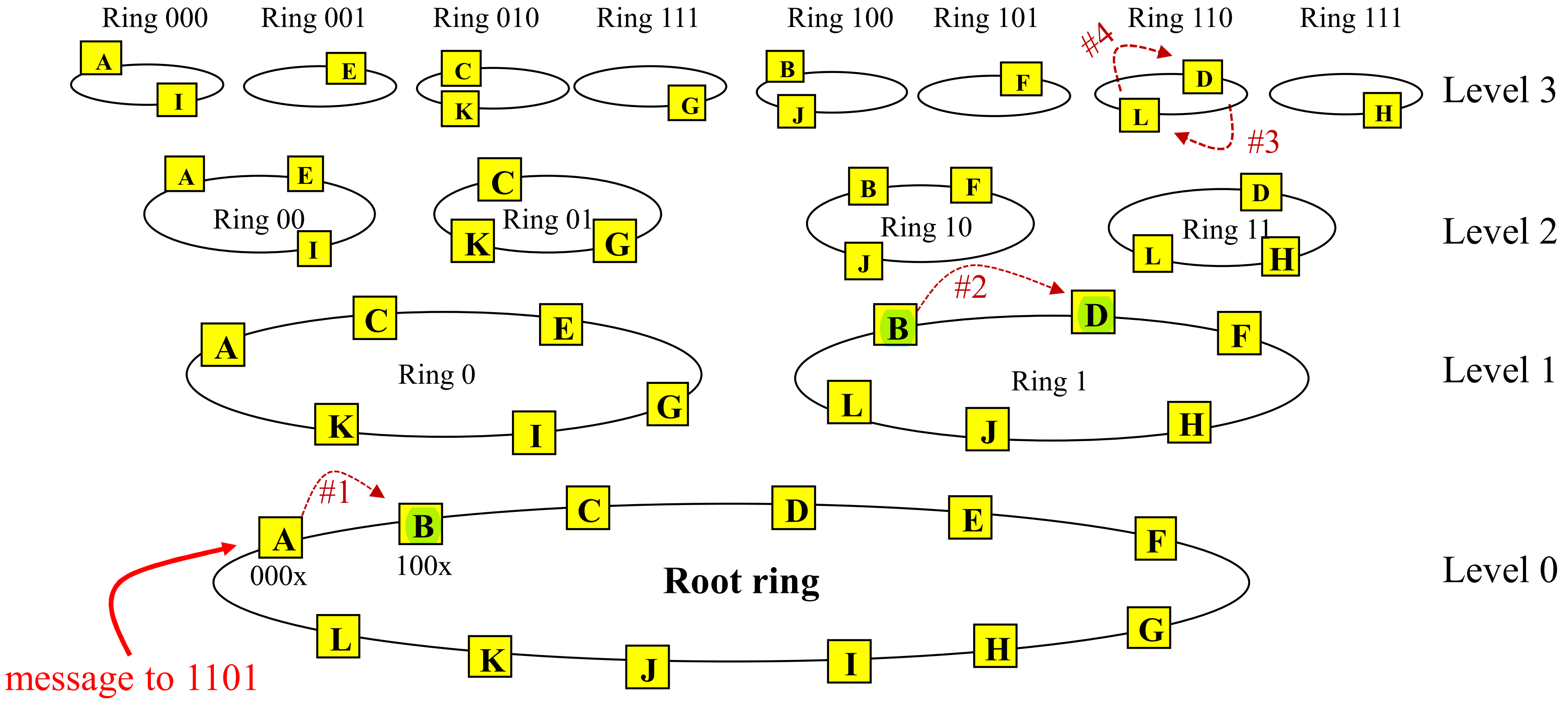


- Suppose A initiates a search of 'G'. Since A's highest neighbor that does not cause 'overshoot' is E (in ring 00), A forwards the message to E.
- Since E is neighbor to G in ring 0, it forwards the message to G, and the search ends

Routing in SkipNet: by Numeric ID

- begins by examining nodes in the level 0 (root) ring until a node X_1 is found whose numeric ID matches the destination numeric ID Y in the first digit.
- the routing operation jumps up to X_1's level 1 ring (which must which also contain the destination node)
- then examines nodes in this level 1 ring until a node X_2 is found whose numeric ID matches the destination numeric ID in the second digit (X_2's level 2 ring must also contain the destination node)
- This procedure repeats until no more progress can be made, i.e., we have reached a ring at some level *h* such that none of the nodes in that ring share $h + 1$ digits with the destination numeric ID
- the "destination" node is the node whose numeric ID is numerically closest to Y amongst all nodes in this highest ring

Routing by Numeric ID (Ex.)

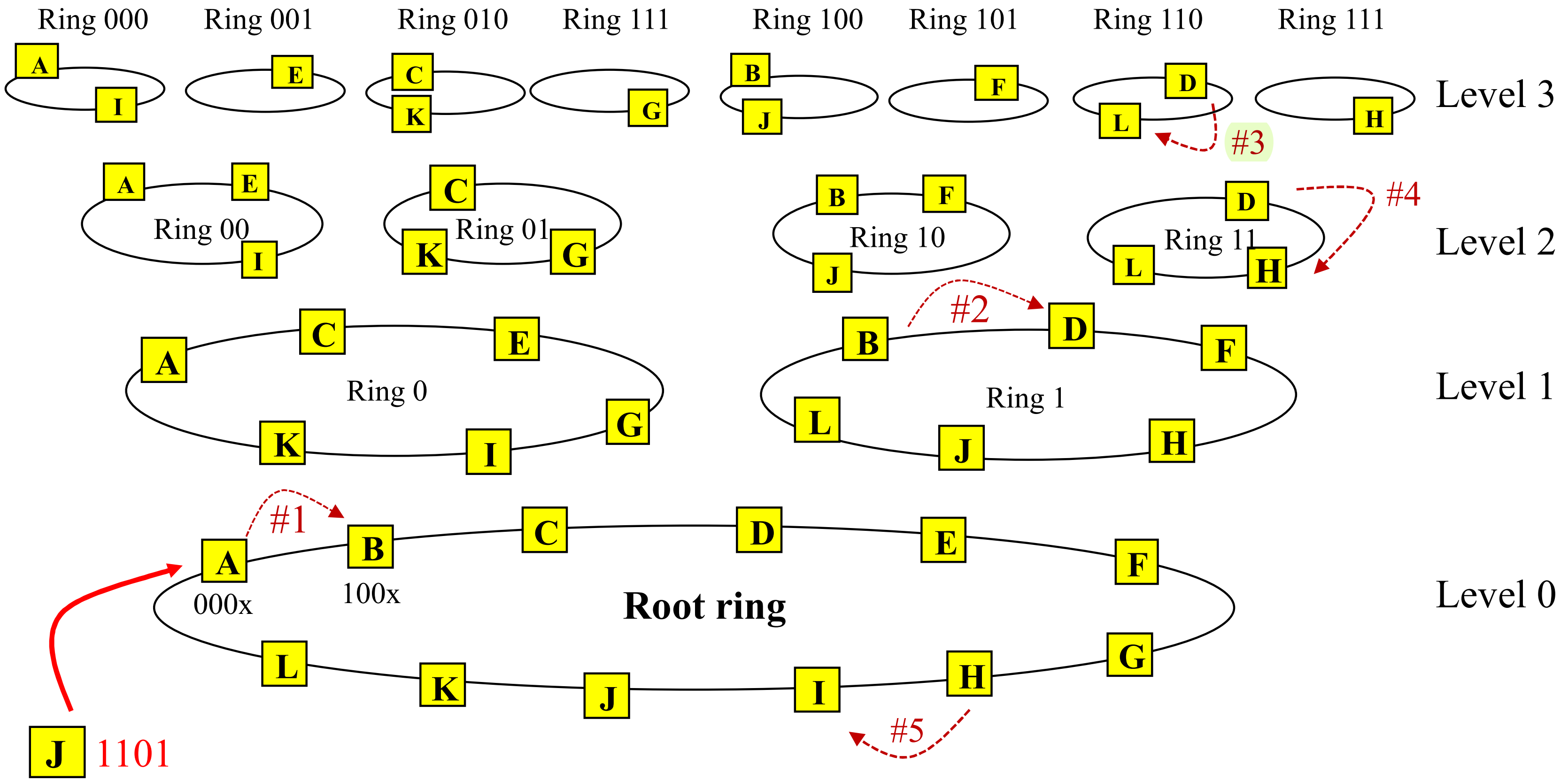


- Suppose A is to route a message to 1101. A first forwards the message to node B because A is not in ring 1.
- B then forwards the message to node D (or L) because B is in ring 1 but not in ring 11.
- If D is not 1101, it forwards the message to L. If L is not 1101 either, the message will return to D. The search ends as no further node can be forwarded.

Node Join

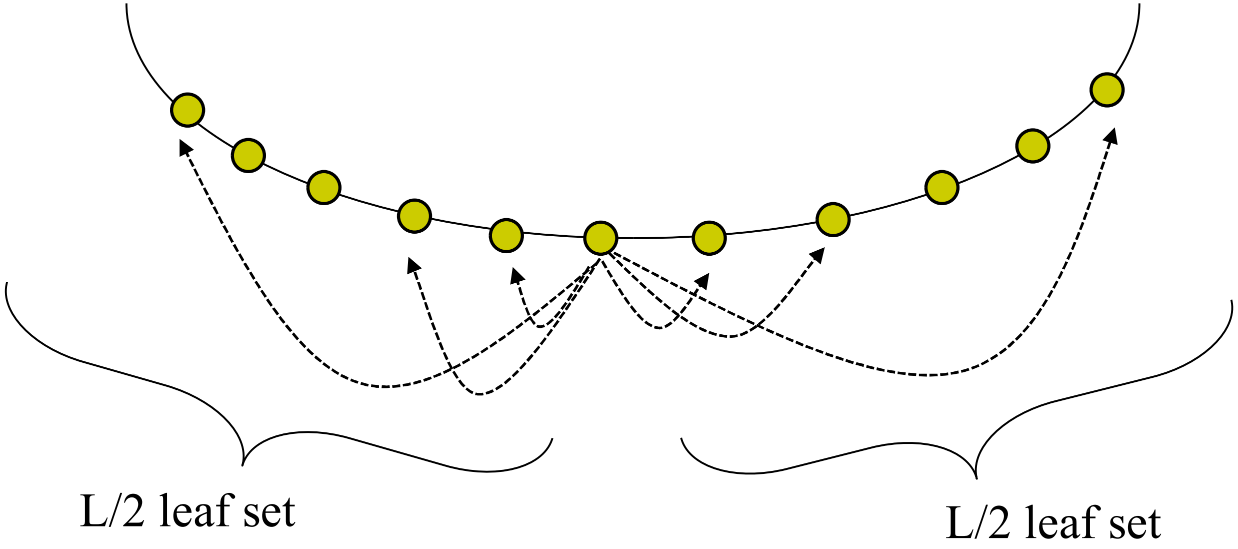
- A new node X first finds the top-level ring that corresponds to X’s numeric ID by routing to the ID.
- X then finds its neighbors in this top-level ring, using a search by name ID within this ring only. Starting from one of these neighbors, X searches for its name ID at the next lower level and thus finds its neighbors at this lower level, and so on until the root ring is reached.
- X then notifies its neighbors in each ring that it should be inserted next to them when it has joined the root ring.
  - Why can’t X neighbors update their neighbor tables once X has joined a level?

Node Join (Ex.)



- Suppose J with ID 1001 is joining via A. The search ID message routes to the highest level ring 110 (#1, 2, 3). J's neighbor in the ring is D & L.
- J then search its name ID at next lower level ring 11 (#4), and finds its position in between H & L. From H, J can learn that its position in ring 1 is also between H & L. From H, and via a search to I (#5), J learns its position in the root ring is between I & K.

Leaf Set and Structure Maintenance



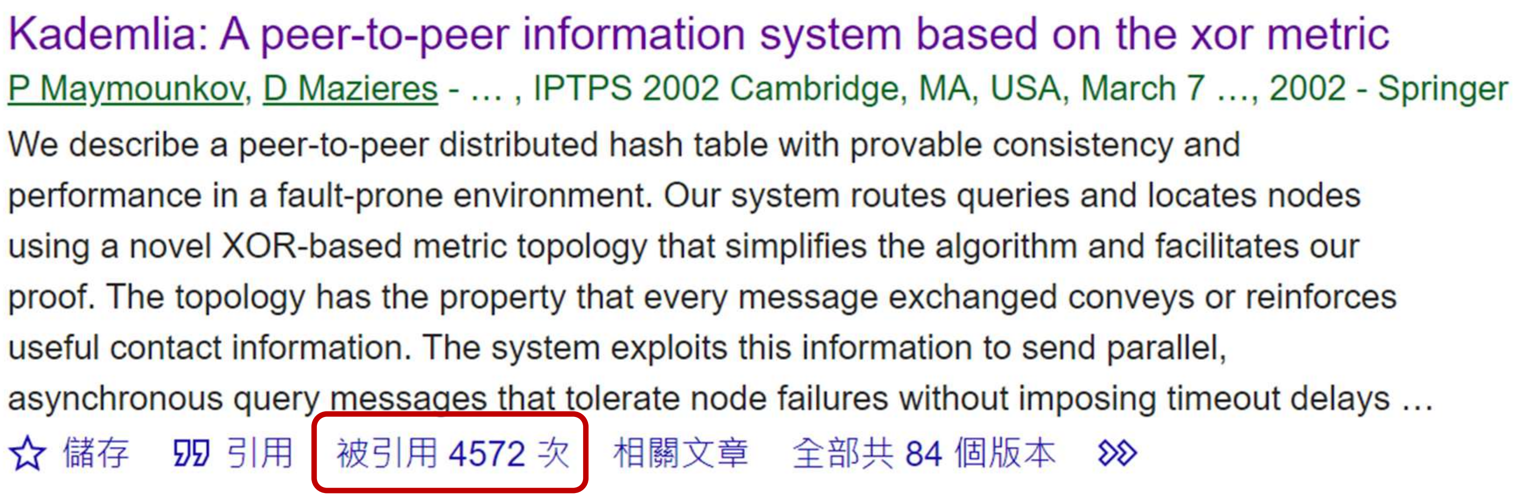
- ➢ SkipNet can route correctly so long as the root ring is maintained. Higher rings are for routing optimization.
- ➢ To increase fault tolerance, like Chord, a node can maintain a set of pointers to the L/2 nodes closest in name ID on the left side and similarly on the right side.
- ➢ Upper level ring memberships are maintained and repaired by a background repair process.

Support of Locality

- **Content locality**: data is placed on specific overlay nodes or distributed across nodes within a given organization.
- **Path locality:** message traffic between two overlay nodes within the same organization is routed within that organization only.
- SkipNet easily supports both locality
  - e.g., by reversing domain names, all nodes within *microsoft.com* are close to one another in the root ring
    - *com.microsoft.division1, com.microsoft.division2, …*
    - by naming documents like *com.microsoft.division1/doc-a, com.microsoft.division1/doc-b…,* documents are stored within organization boundary, and local search originating from an organization will not pass outside the organization.

Kademlia

- A Peer-to-peer Information System Based on the XOR Metric



Kademlia: A peer-to-peer information system based on the xor metric

P Maymounkov, D Mazieres - … , IPTPS 2002 Cambridge, MA, USA, March 7 …, 2002 - Springer

We describe a peer-to-peer distributed hash table with provable consistency and performance in a fault-prone environment. Our system routes queries and locates nodes using a novel XOR-based metric topology that simplifies the algorithm and facilitates our proof. The topology has the property that every message exchanged conveys or reinforces useful contact information. The system exploits this information to send parallel, asynchronous query messages that tolerate node failures without imposing timeout delays …

(as of 2023.04.26)

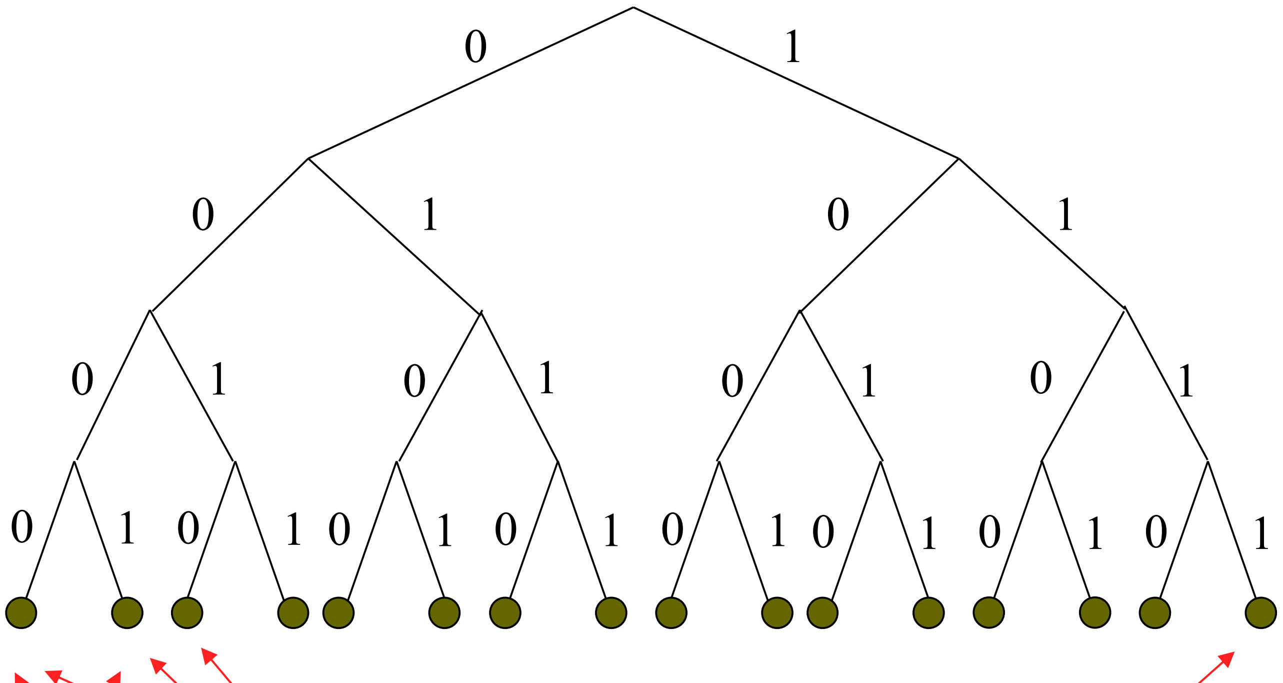
Characteristic

- Both keys and nodes have a uniformly distributed 160-bit ids (e.g., SHA-1 hash).
- Nodes are treated as leaves in a binary tree.
- Use XOR method as lookup algorithm.
- Lookup algorithm, which finds the node, whose ID is "nearest" to a given key.
- Store and retrieve a key/value pair at the node whose ID is "nearest" to the key.
- Widely used in BitTorrent, OverNet and eMule

XOR topology

- Distance between x, y is denote as : $d(x, y) = x \oplus y$ (interpreted as an unsigned integer)
  - $d(x, x) = 0$, $d(x, y) > 0$ if $x \neq y$
  - $\forall x, y : d(x, y) = d(y, x)$
  - $d(x, y) + d(y, z) \geq d(x, z)$ (triangle proper)
- Geometric intuition:
  - The closest leaf to an ID x is the leaf whose ID shares the longest common prefix of x.
  - If there are empty branches in the tree (then there might be more than one leaf with the longest common prefix), the closest leaf to x will be the closest leaf to ID $\tilde{x}$ produced by flipping the bits in x corresponding to the empty branches of the tree

Node Distance in Kademlia



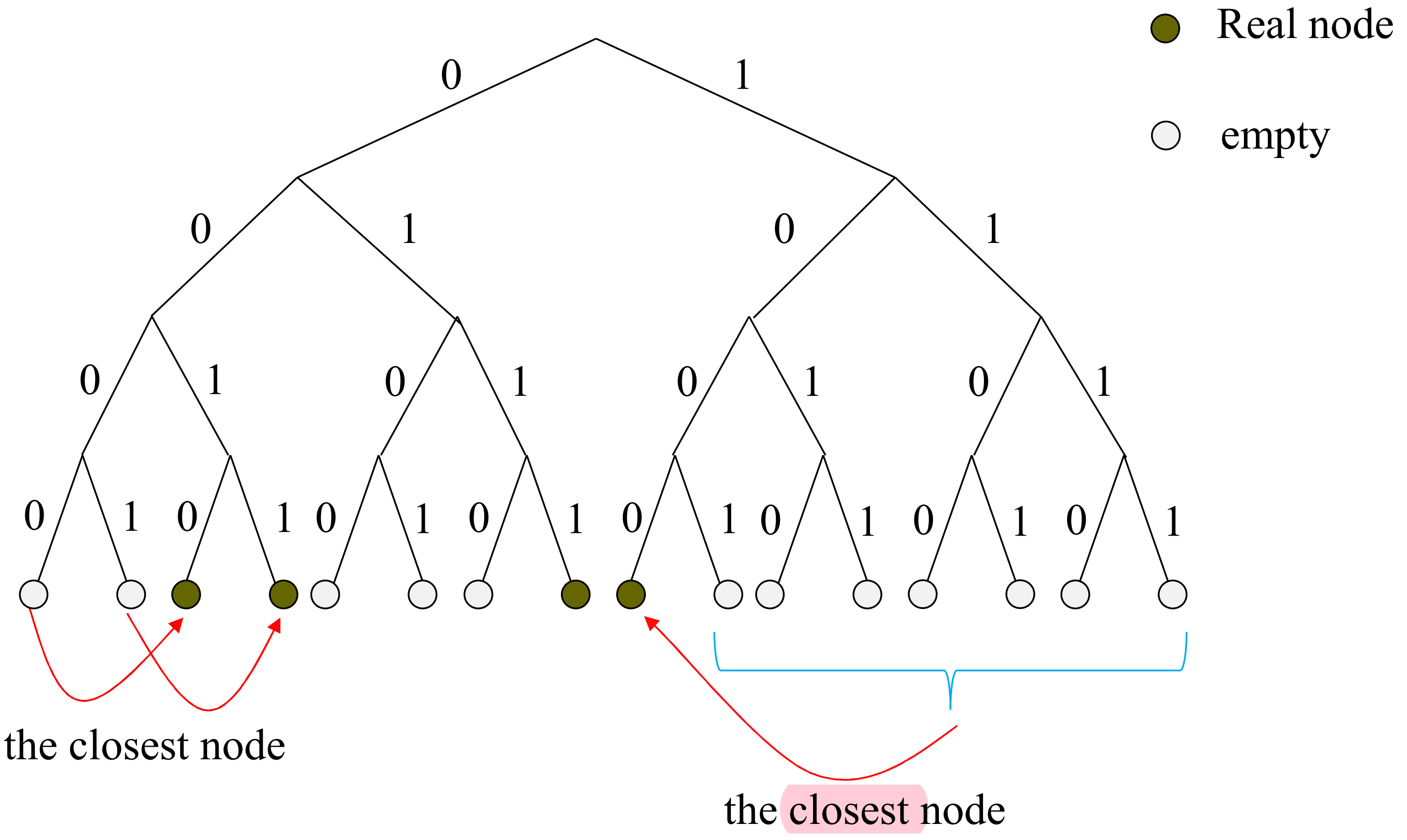
$d(0000, 0001) = 0000 \oplus 0001 = 0001$

$d(0001, 0010) = 0001 \oplus 0010 = 0011$

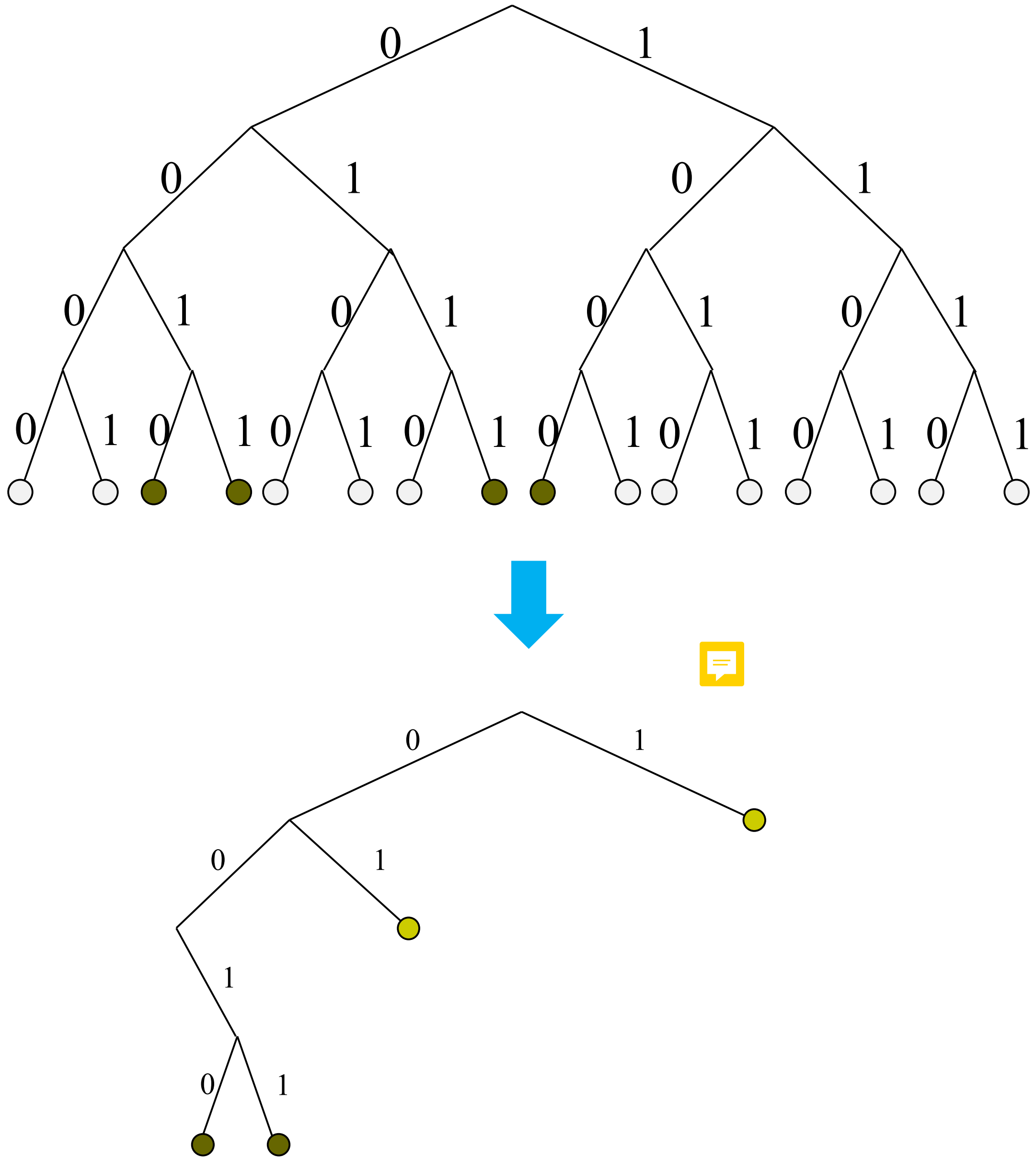
$d(0000, 1111) = 0000 \oplus 1111 = 1111$

IDs with longer matching prefixes are numerically closer.

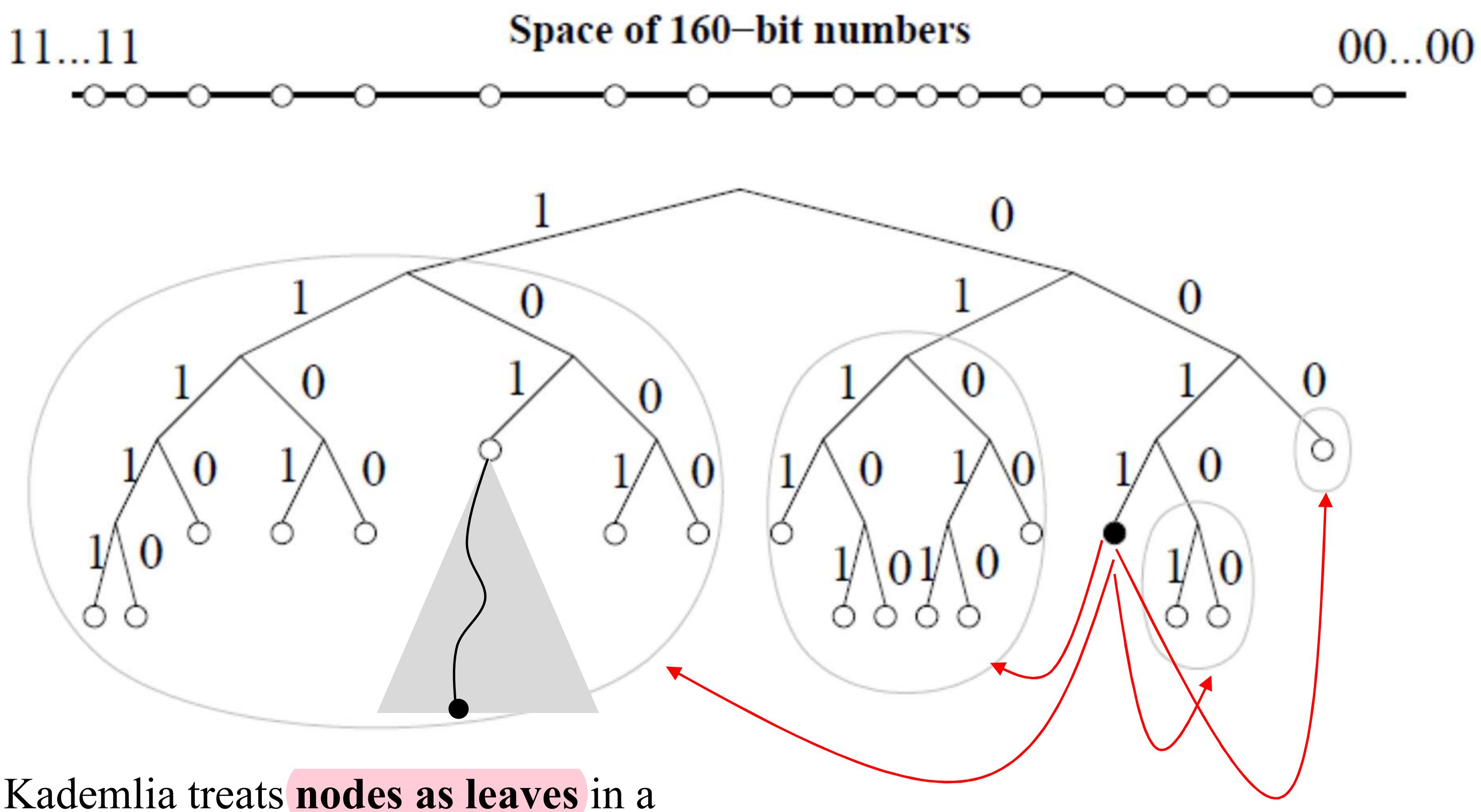
Dealing with Empty Nodes: Closest Node



Dealing with Empty Nodes (2)



Kademlia Binary Tree



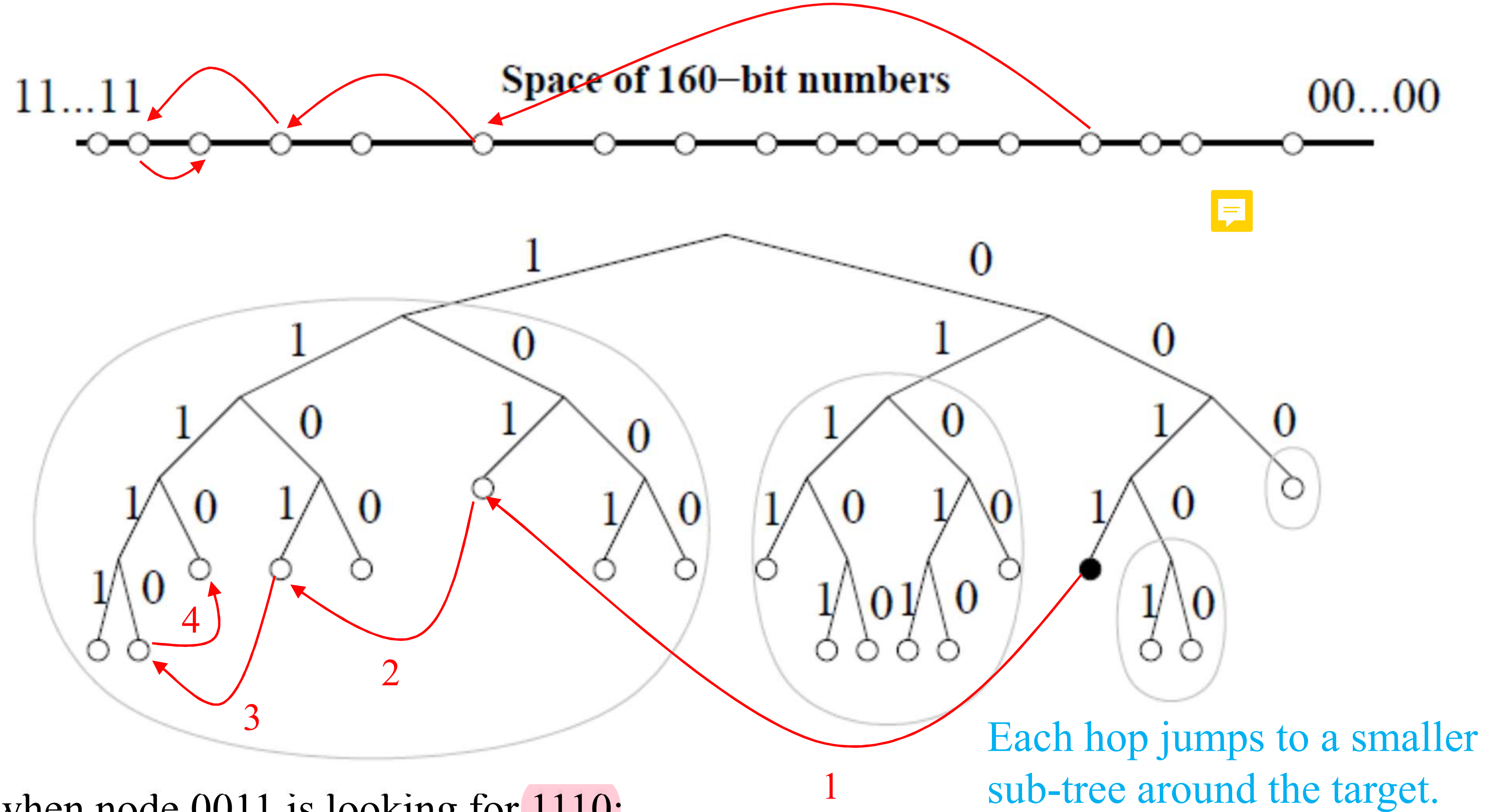
Kademlia treats **nodes as leaves** in a binary tree, with each node's position determined by the shortest unique prefix of its ID.

Since a node has 160-bit ID, what does this mean? Or what does a node 0011 mean?

node 0011 must have a contact in each of the group:

1*****…
01****…
000***…
0010**…

Routing in Kademlia



Each hop jumps to a smaller sub-tree around the target.

when node 0011 is looking for 1110:

1. node 0011 knows node 101 in 1***…, and sends the query
2. node 101 knows node 1101 in 11***…, and (**recursively**) forwards the query (or return the node to node 0011 for **iterative** resolution)
3. node 1101 knows node 11110 in 111**…, and forwards the query
4. node 11110 knows node 1110

Routing in Kademlia (2)

- If node 1100 0111 wishes to go to 1111 0111 then

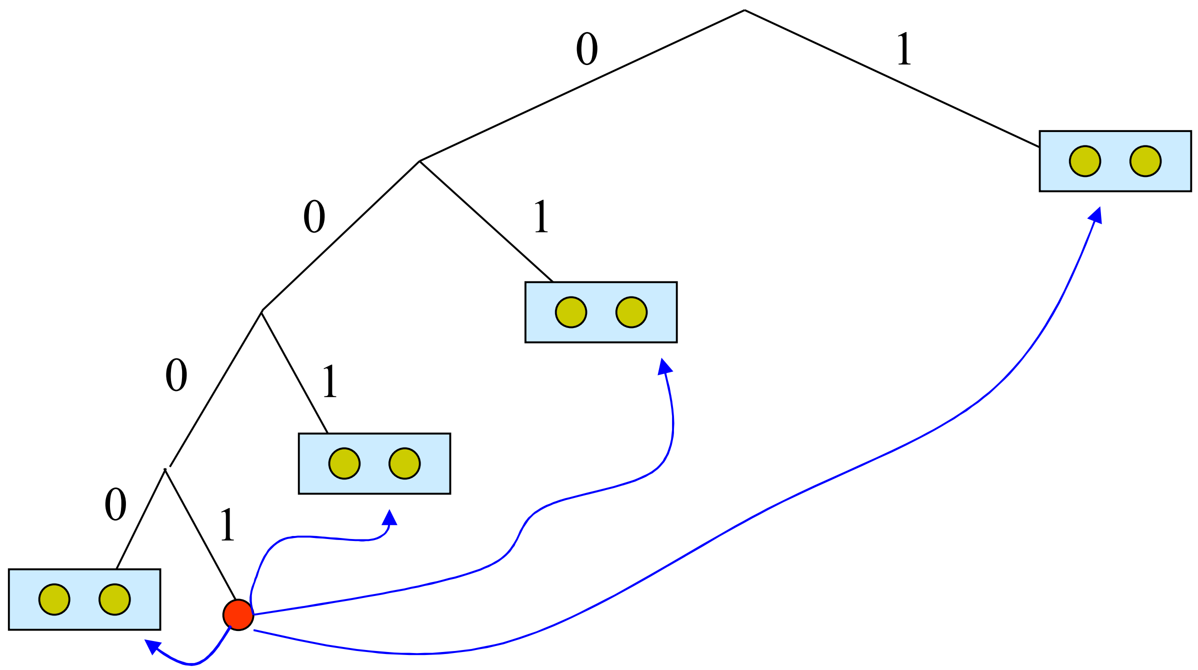
  1100 0111 → 111 * (say, 1110 1000)
  → 1111 * (say, 1111 0100)
  → 1111 011* (say, 1111 0110)
  → 1111 0111

- Much like Chord? Tapestry?
  - more flexible than Chord?

Node State

- Contact
  - A list of [IP, UDP port, node ID]
  - Node exchange contacts when exchange messages
- *K*-Bucket
  - Every node keeps a list of up to *k* nodes which are of distance between 2^i and 2^{i+1} for every $0 \leq i < 160$.
  - *K* is a system-wide parameter determining the fault-tolerance degree and routing flexibility (e.g., k=20)
    - So each node maintains a routing table consisting of up to 160 *k*-buckets.
  - replacement policy, e.g.,
    - keep the oldest live contacts around
      - ✓ The longer a node has been up, the more likely it is to remain up another hour.
    - least-recently seen eviction

2-Bucket



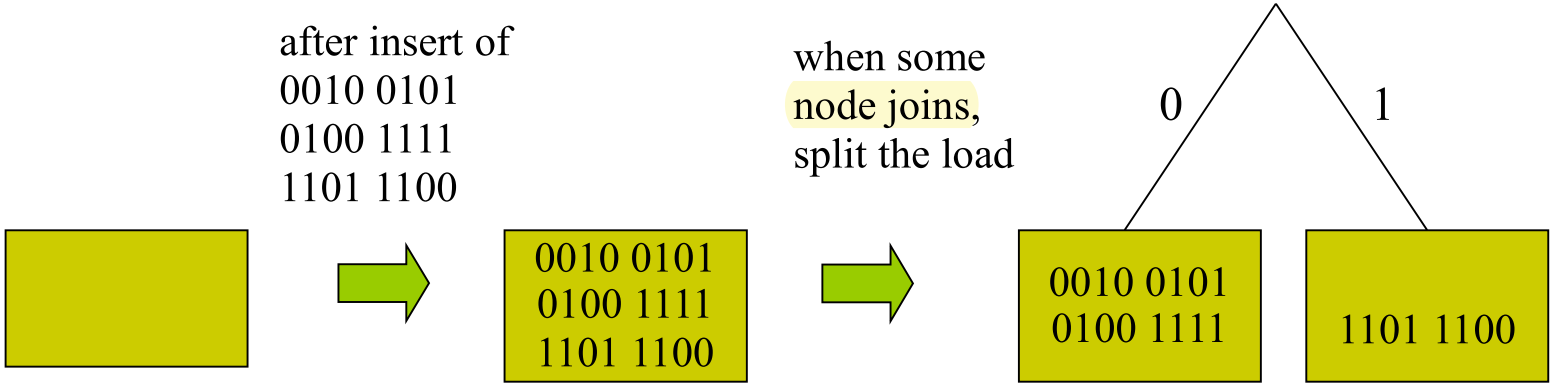
node 0001's routing table

Joining, Leaving and Refreshes

- Node join:
  - Borrow some contacts from an already online node
  - Lookup self
  - Cost of join is O(*log N*) messages
- Node leave: no action
  - Hourly bucket refreshes

How to use Kademlia to store objects

- Initially, one node holds no object



after insert of
0010 0101
0100 1111
1101 1100

0010 0101
0100 1111
1101 1100

when some
node joins,
split the load

0010 0101
0100 1111

1101 1100

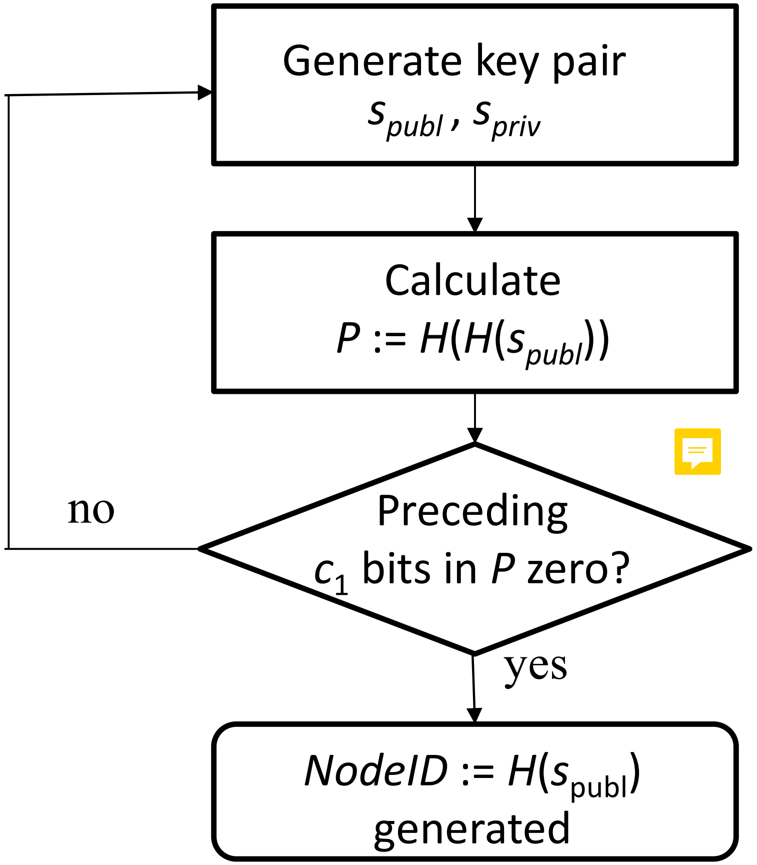
Enhance Security

- How to prevent
  - **Sybil attack** (generating a large number of node Ids)
  - **Eclipse attack** (choosing the node Id freely)
    - can we just use IP & Port to hash?
- S/kademlia's proposal:
  - use public key to hash
  - In the absence of a trustworthy authority to issue public/private keys, **crypto puzzles** can be used to allow a completely decentralized node Id generations.

Reading:
S/kademlia: A practicable approach towards secure key-based routing. I. Baumgart and S. Mies. International Conference on Parallel and Distributed Systems, 2007.

Static (left) and dynamic (right) crypto puzzles for node Id generation

difficult to choose an arbitrary node Id



Generate key pair s_{publ}, s_{priv}

Calculate $P := H(H(s_{publ}))$

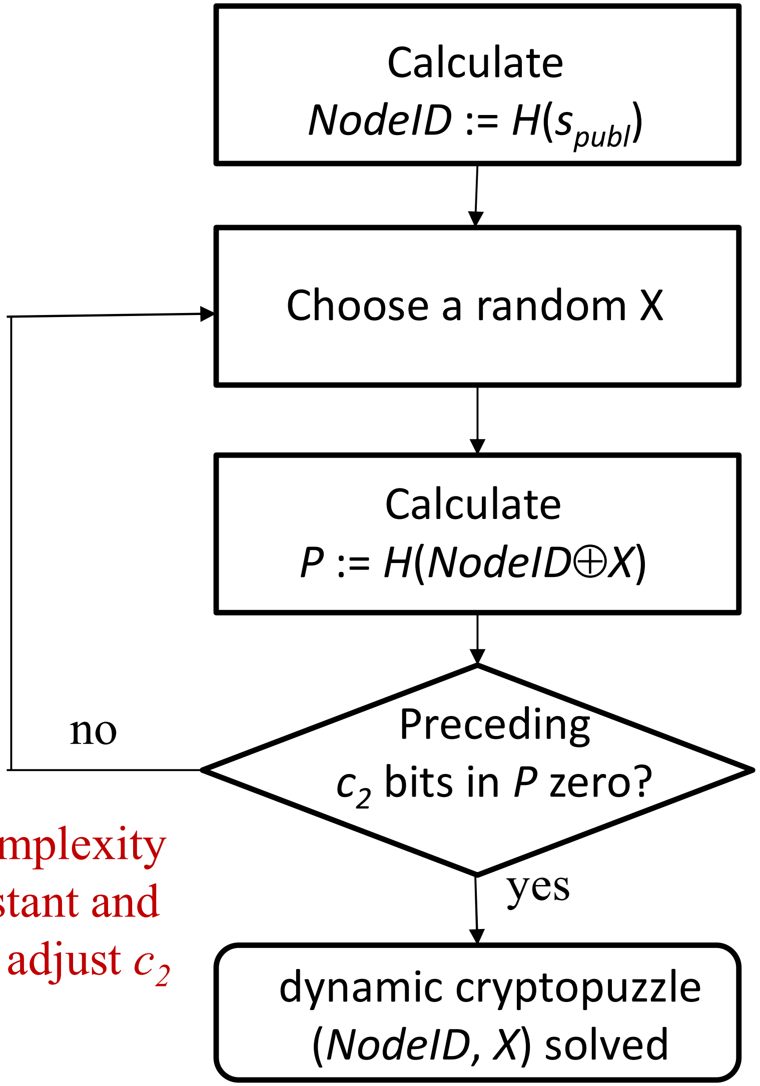
Preceding c_1 bits in P zero?

no

yes

$NodeID := H(s_{publ})$ generated

difficult to generate a huge amount of node Ids



Calculate $NodeID := H(s_{publ})$

Choose a random X

Calculate $P := H(NodeID \oplus X)$

Preceding c_2 bits in P zero?

no

yes

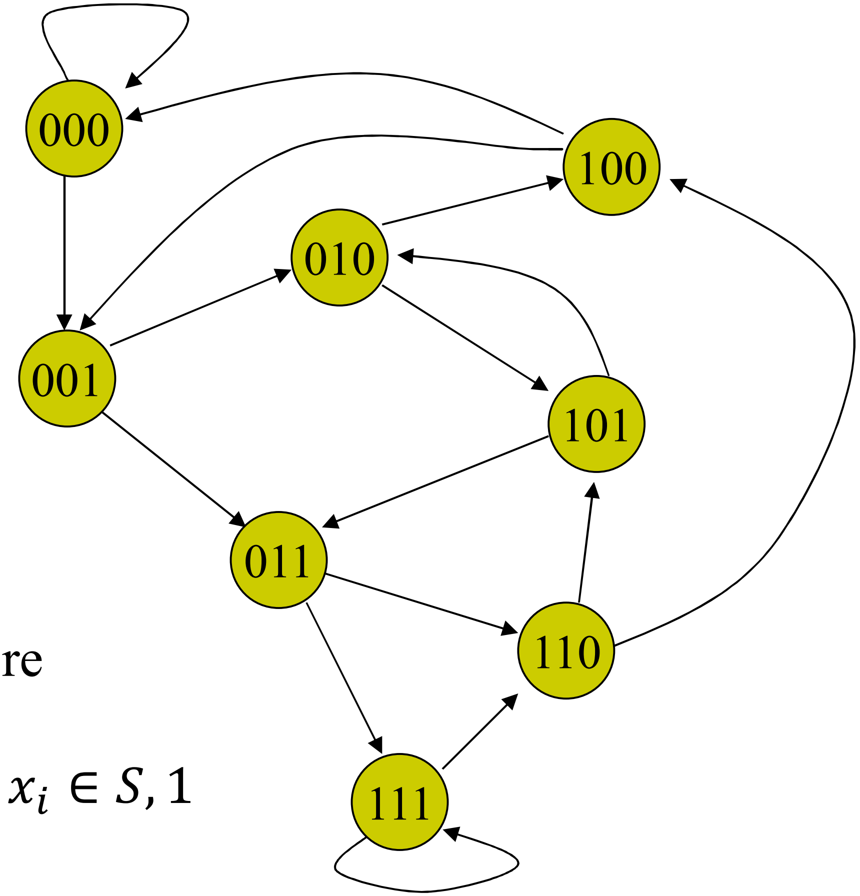
dynamic cryptopuzzle ($NodeID$, X) solved

c_i: puzzle complexity
use c_1 as constant and
dynamically adjust c_2

Koorde

O(*log N*) routing path with only O(1) node state

A node $x_n \ldots x_2x_1$ in $\{0,1\}^n$ connects to $x_{n-1} \ldots x_2x_1 1$ and $x_{n-1} \ldots x_2x_1 0$ (shifting left one symbol, connecting to those nodes $x_{n-1} \ldots x_2x_1 w$, where $w \in \{0, 1\}$)



can be extended to a graph G=(V, E), where
$V = S^n$, $S = \{s_1, s_2, \ldots s_m\}$,
$E = \{((x_1, x_2, \ldots, x_n), (x_2, x_3, \ldots, x_n, s_j)): x_i \in S, 1 \leq i \leq n, 1 \leq j \leq m\}$,

Why is this structure not practical?

A de Bruijn graph of $\{0, 1\}^3$

Coral & CoralCDN

- **CoralCDN**: a peer-to-peer content distribution network (CDN) built on top of a DHT-base key/value indexing infrastructure called **Coral**.
- **Coral** improve DHTs in
  - hot-spot prevention
  - locality
  - provides weaker consistency than traditional DHTs
  - it's indexing abstraction is called **distributed sloppy hash table (DSHT)**
  - built on Chord (or Kademlia)

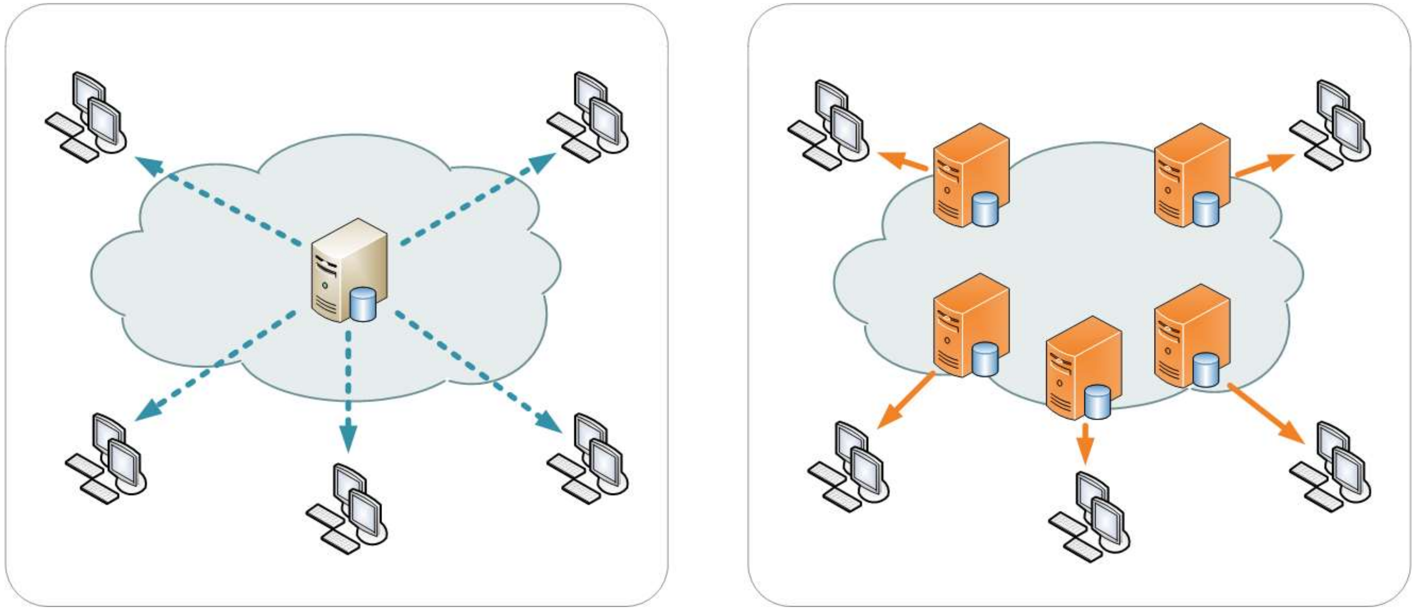
Reading:

Democratizing content publication with coral, M. J. Freedman, E. Freudenthal, and D. Mazieres. NSDI, 2004 (optional)

Sloppy Hashing and Self-Organizing Clusters, M. J. Freedman, E. Freudenthal, and D. Mazieres, IPTPS 2003

Content Distribution Network (CDN)

- CDN: a geographically distributed group of servers that work together to provide fast delivery of Internet content.



(Left) Single server distribution (Right) CDN scheme of distribution (source: wiki)

DHT vs. DSHT

- DHT functions: assuming a unique value for each key
  - *put(key, value)*
  - *get(key)*
- DSHTs functions: a key may have multiple values
  - *put(key, value):*
  - *get(key):* return some subset of values stored

Applications of Weakened Consistency

- For applications such as CDN, pointers to locations of cached/replicated data can be stored in DSHT
  - e.g., when a node cache an object referenced by URL, it inserts a pointer to that object into the DSHT by executing

    put (*key* = *hash* (URL), *nodeaddr*).
  - a *get*(*key*) suffices to return some pointers

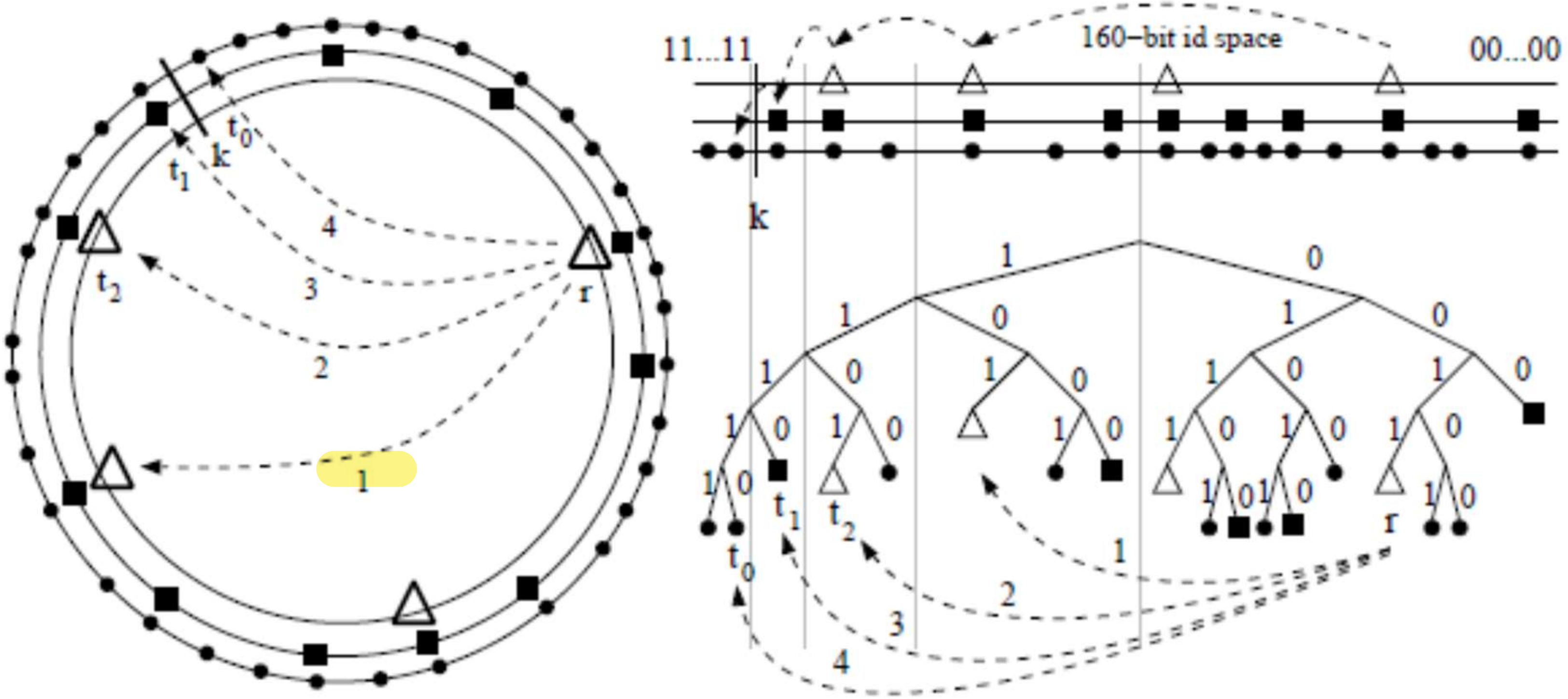
Hot-Spot Prevention

- *put*(*key*, *nodeaddr*):
  - The inserting node calls *find_closer_node*(*key*) until it locates the first node whose list stored under key is full, or it reaches the node closest to key.
    - "**spilling-over**": if this located node is full, backtrack one hop on the lookup path. This target node appends *nodeaddr* with a timestamp to the list stored under *key*.
- *get*(*key*):
  - upon hitting a node storing key, returns the key's corresponding contact list.
    - The requesting node can contact these nodes, in parallel or in some application-specific way, to download the stored data.

Locality by Hierarchical Lookup

- Coral uses several levels of DSHTs called *clusters*, of increasing network *diameter*.
  - The diameter of a cluster is the maximum desired round-trip time between any two nodes it contains.
  - Current implementation: 3 levels
    - level-0 (planet-wide) cluster: ∞ (there is only one level-0 cluster)
    - level-1 (“higher”) clusters: 100 msec (multiple level-1 clusters)
    - level-2 (“low-level”) clusters: 30 msec (multiple level-2 clusters)
  - Each Coral node joins one cluster in each level
- Data Insert & Retrieve:
  - To insert a key/value pair, a node performs a put on all levels of its clusters.
  - To retrieve a key, a node starts by performing a get on its level-2 cluster to try to take advantage of network locality.

Coral’s Hierarchical Lookup



Coral’s hierarchical lookup visualized on the Chord (left) and Kademlia (right) routing structures. Route RPCs are shown with sequential numbering.

source: Sloppy Hashing and Self-Organizing Clusters, M. J. Freedman, E. Freudenthal, and D. Mazieres, IPTPS 2003.

Nodes Join

- As in most peer-to-peer systems, a node *x* joins the system by learning some existing node *y* in Coral.
  - *x* can use RTT (Round-Trip Time) to find nearby nodes (candidates provided by *y*) to join an acceptable cluster, or creates a new cluster of itself.
    - A node can join a better cluster whenever it learns of one.
    - New nodes info can be obtained through regular operations of the system.

Discovering A Nearby Cluster

- Coral exploits the DSHT interface to let nodes find nearby clusters:
  - Upon joining a low-level cluster, a node inserts itself into its higher-level clusters, keyed under the IP addresses of its gateway routers, discovered by *traceroute:*
  - For each of the first five routers returned, it executes *put*(*hash*(*router.ip*), *nodeaddr*).
  - A new node, searching for a low-level acceptable cluster, can perform a get on each of its own gateway routers to learn some set of topologically-close nodes.

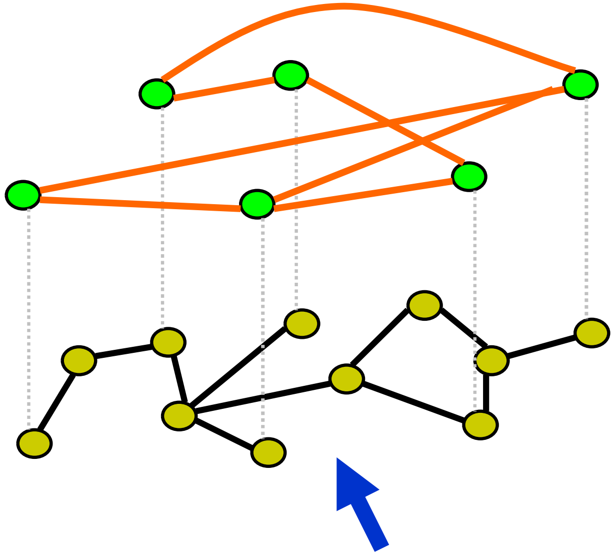
Clusters Merge & Split

See paper:
Sloppy Hashing and Self-Organizing Clusters, M. J. Freedman, E. Freudenthal, and D. Mazieres, IPTPS 2003

Proximity

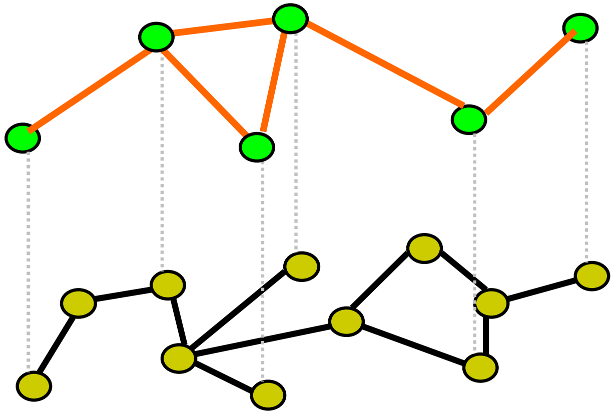
Proximity

Not consider proximity



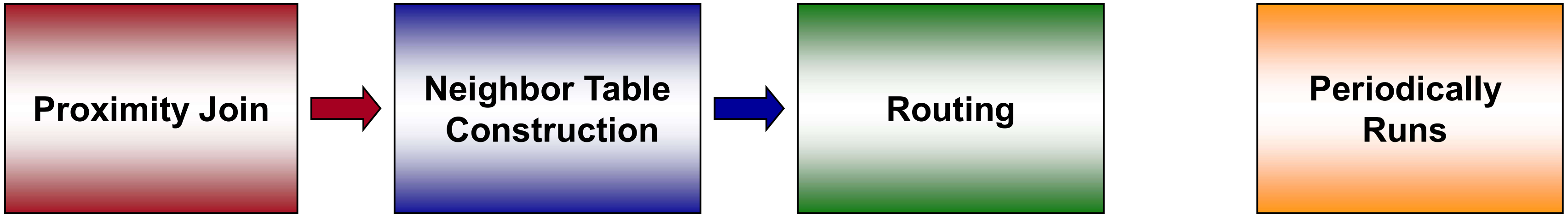
A hop in overlay could be several IP hops away in underlying network.

Consider physical topology



Proximity Approach

“When” the approach takes place:



Proximity Join → Neighbor Table Construction → Routing

Periodically Runs

Geometric Layout in CAN

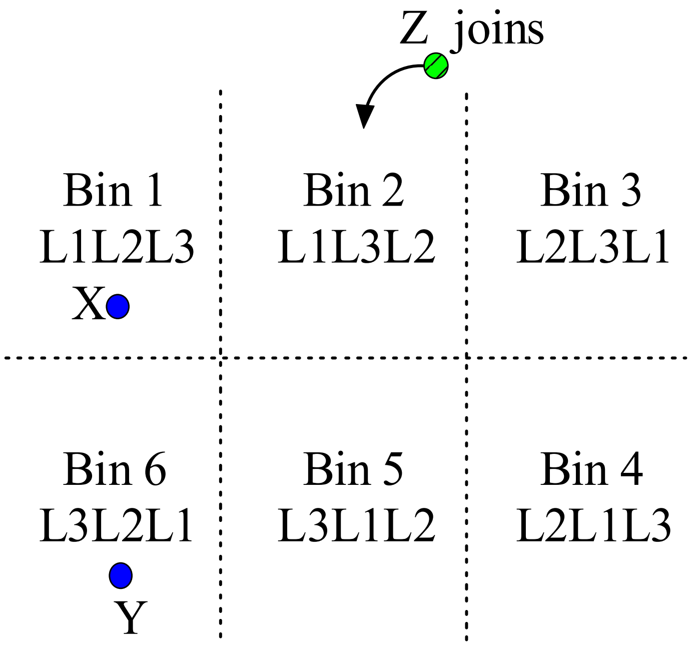
Proximity Neighbor Selection in Pastry and Chord

Proximity Route Selection in CAN and Chord

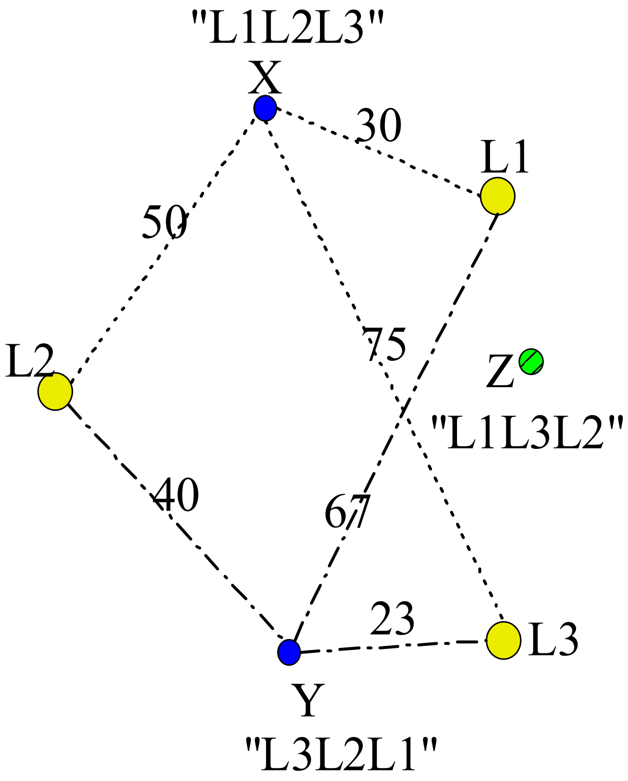
Position exchange

Proximity Join

- When a node joins, the first thing to do is to gather proximity information
  - Landmark (CAN)
  - Knowledge (Location-based ID)
- According to the proximity information, a node decides which position to join.



Bin 1 L1L2L3 X	Bin 2 L1L3L2	Bin 3 L2L3L1
Bin 6 L3L2L1 Y	Bin 5 L3L1L2	Bin 4 L2L1L3



Proximity Neighbor Table Construction

- Proximity is achieved by making nodes in neighbor table as close as possible
- Performance depends on neighbor selection flexibility.
  - High: Tapestry, Pastry, Chord
  - Low: CAN

Proximity Routing

- Proximity is achieved by selecting, at the routing stage, one neighbor that with the shortest latency to deliver a message.
- Routing flexibility.
  - High: CAN
  - Low: Tapestry, Pastry, Chord