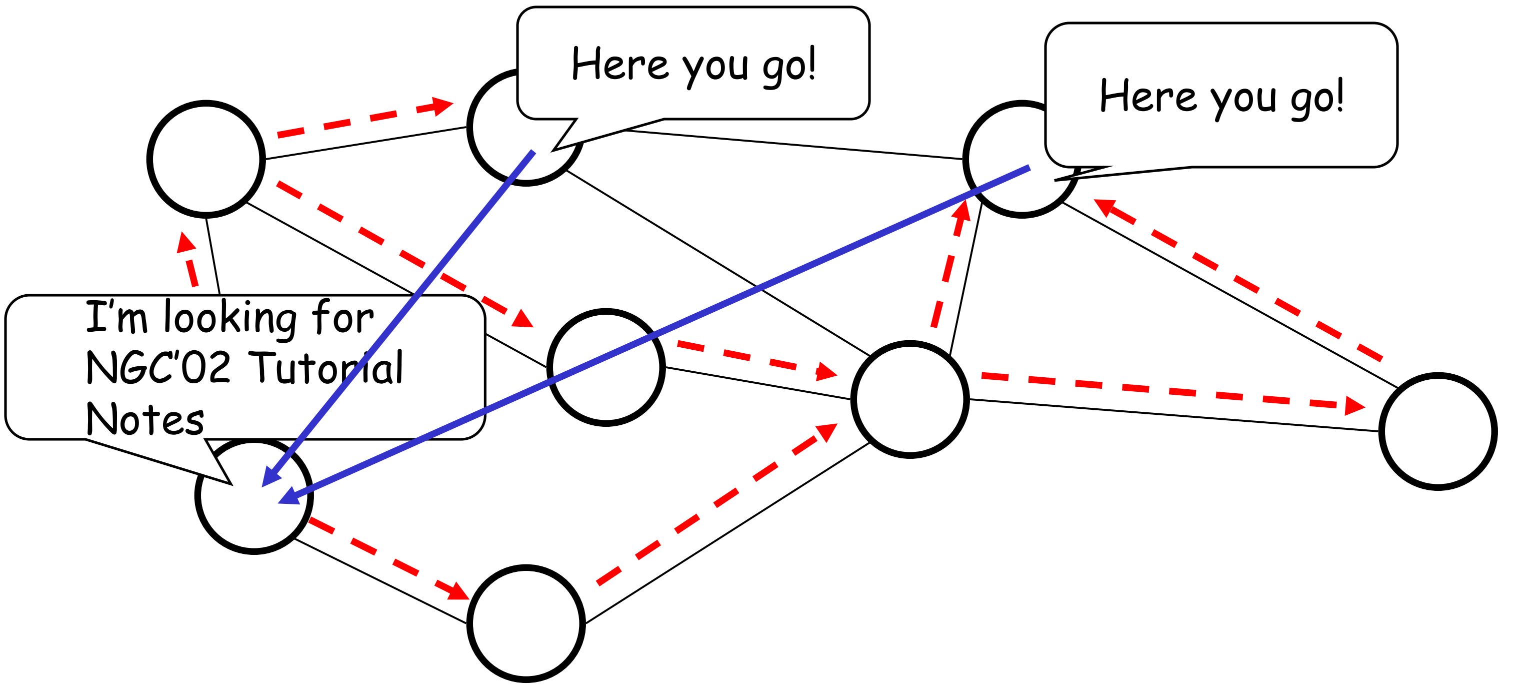
Structured P2P: DHT Approaches

- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services

Notes based on notes by K.W. Ross, J. Kurose, D. Rubenstein, and others

Challenge: Locating Content



- Simplest strategy: expanding ring search
- If K of N nodes have copy, expected search cost *at least* N/K, i.e., O(N)
- Need many cached copies to keep search overhead small

Directed Searches

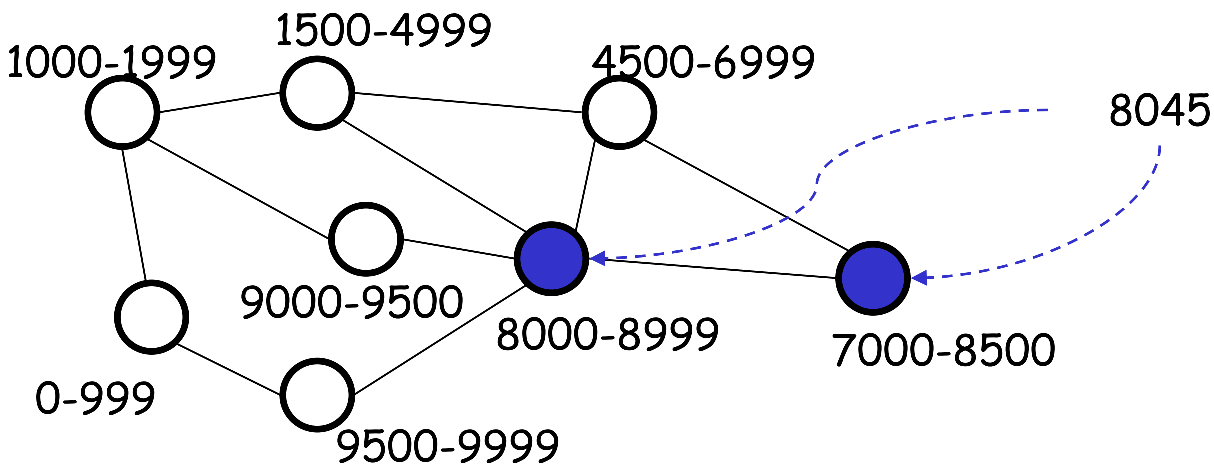
- Idea:
  - assign particular nodes to hold particular content (or pointers to it, like an information booth)
  - when a node wants that content, go to the node that is supposed to have or know about it
- Challenges:
  - Distributed: want to distribute responsibilities among existing nodes in the overlay
  - Adaptive: nodes join and leave the P2P overlay
    - distribute knowledge responsibility to joining nodes
    - redistribute responsibility knowledge from leaving nodes

Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
  - key: ss number; value: human name
  - key: content type; value: IP address
- Peers query DB with key
  - DB returns values that match the key
- Peers can also insert (key, value) peers

DHT Step 1: The Hash

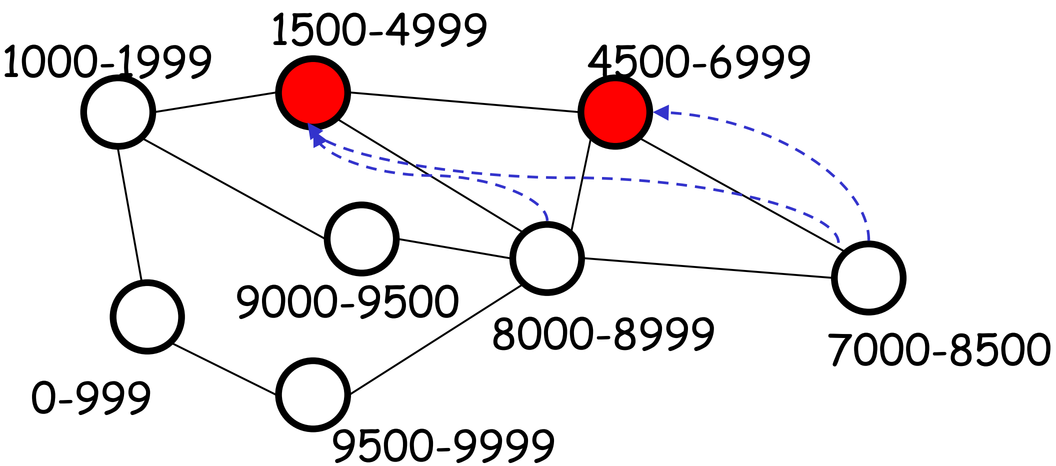
- Introduce a hash function to map the object being searched for to a unique identifier:
  - e.g., h("NGC'02 Tutorial Notes") → 8045
- Distribute the range of the hash function among all nodes in the network



- Each node must "know about" at least one copy of each object that hashes within its range (when one exists)

"Knowing about objects"

- Two alternatives
  - Node can cache each (existing) object that hashes within its range
  - Pointer-based: level of indirection - node caches pointer to location(s) of object



DHT Step 2: Routing

- For each object, node(s) whose range(s) cover that object must be reachable via a "short" path
  - by the querier node (assumed can be chosen arbitrarily)
  - by nodes that have copies of the object (when pointer-based approach is used)
- The different approaches (CAN,Chord,Pastry,Tapestry) differ fundamentally only in the routing approach
  - any "good" random hash function will suffice

DHT Routing: Other Challenges

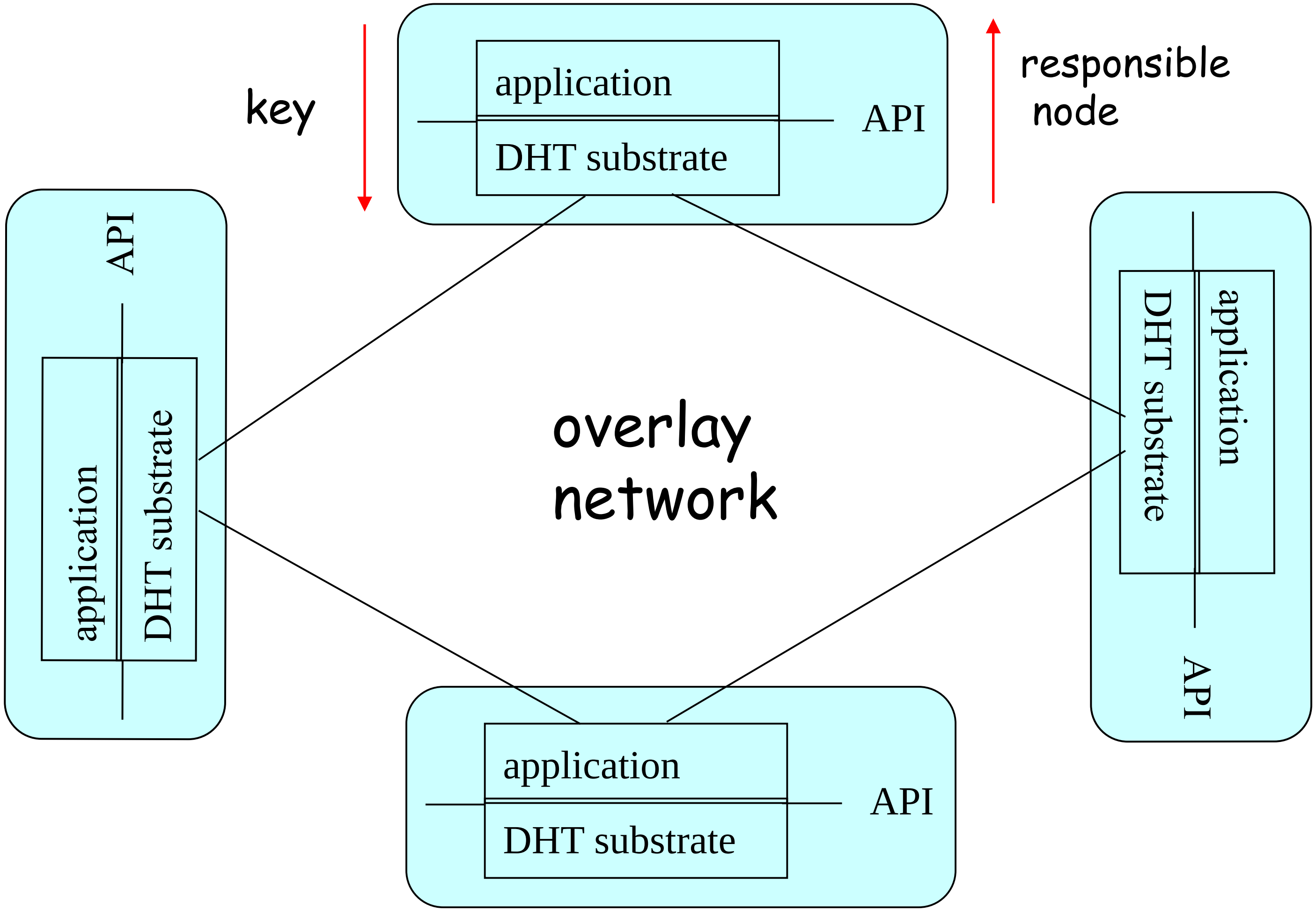
- # neighbors for each node should scale with growth in overlay participation (e.g., should not be O(N))
- DHT mechanism should be fully distributed (no centralized point that bottlenecks throughput or can act as single point of failure)
- DHT mechanism should gracefully handle nodes joining/leaving the overlay
  - need to repartition the range space over existing nodes
  - need to reorganize neighbor set
  - need bootstrap mechanism to connect new nodes into the existing DHT infrastructure

DHT API

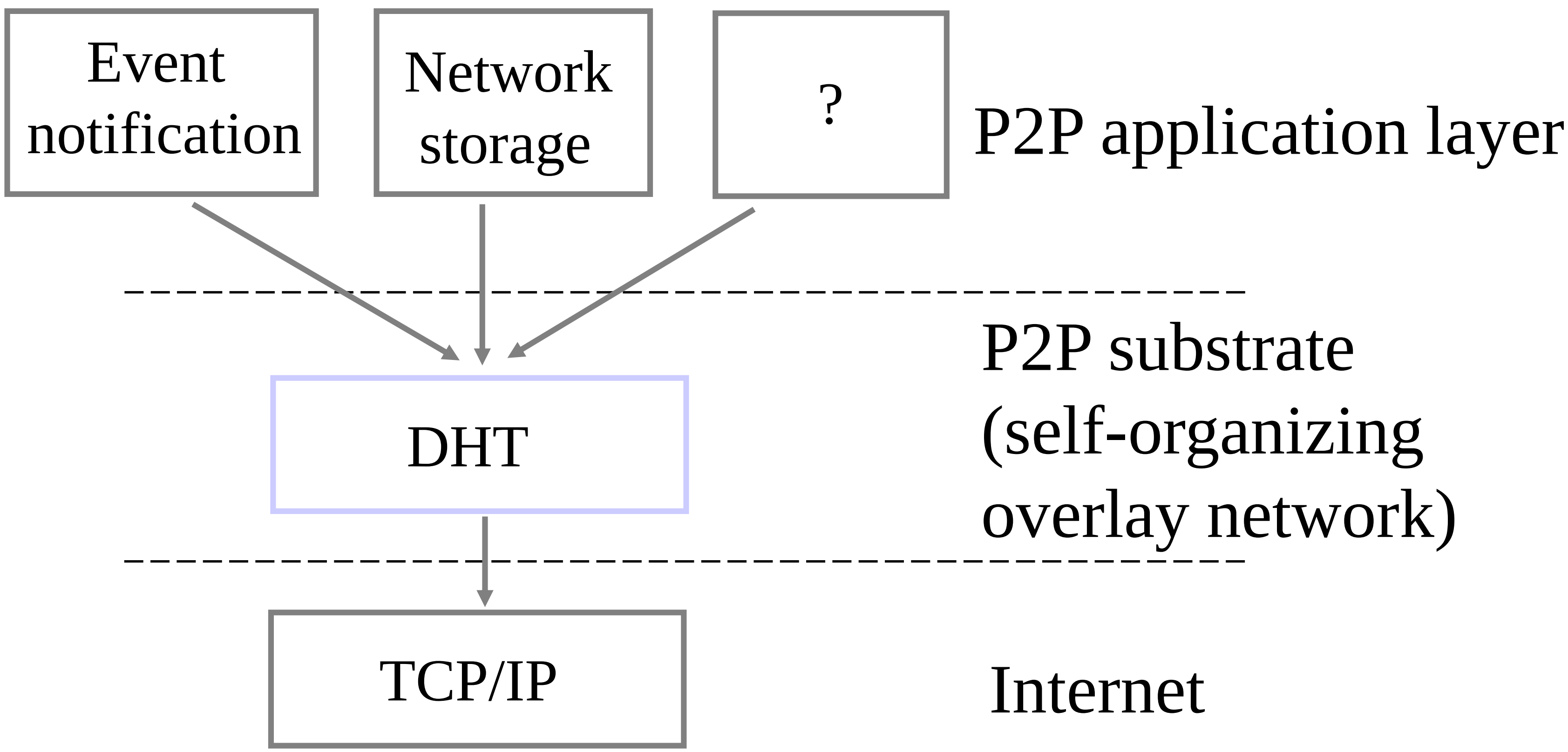
- each data item (e.g., file or metadata containing pointers) has a key in some ID space
- In each node, DHT software provides API:
  - Application gives API key k
  - API returns IP address of node that is responsible for k
- API is implemented with an underlying DHT overlay and distributed algorithms

DHT API

each data item (e.g., file or metadata pointing to file copies) has a key



DHT Layered Architecture



3. Structured P2P: DHT Approaches

- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services
- Topology-centric lookup service

CARP

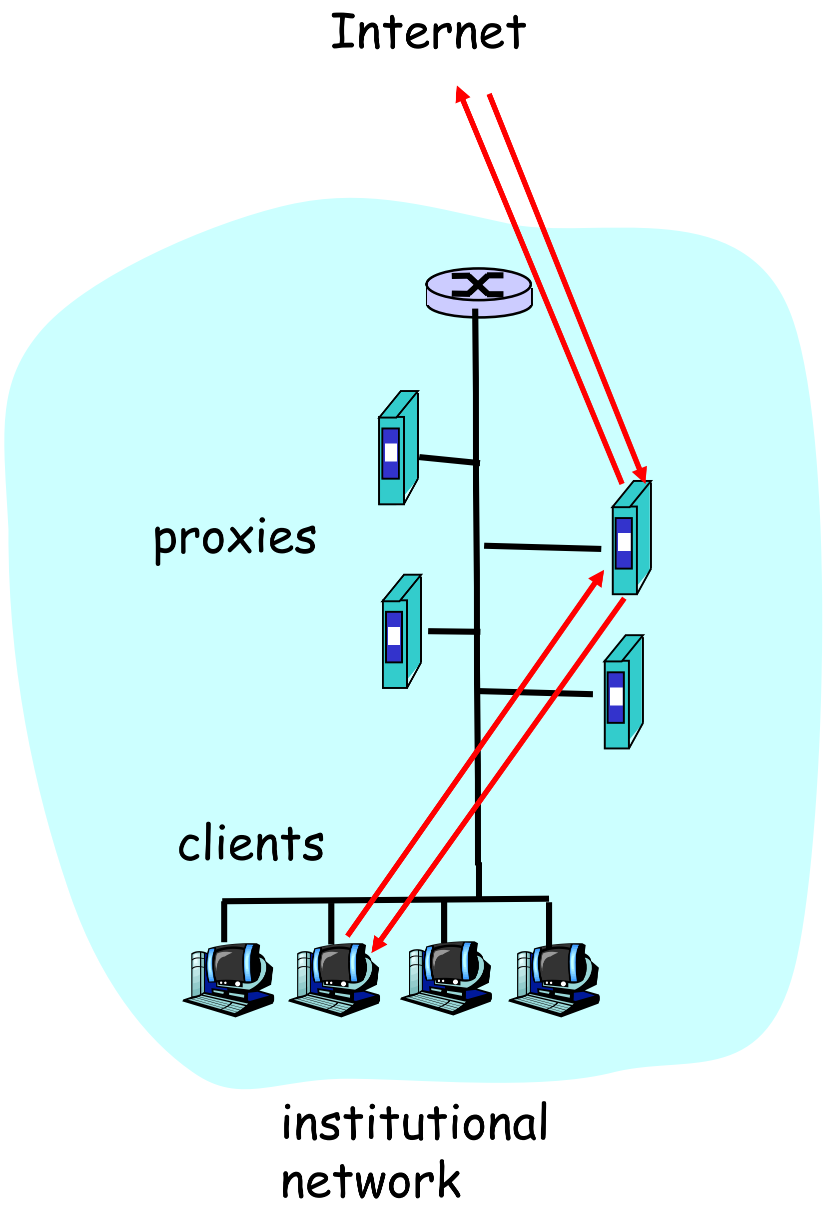
DHT for cache clusters

- Each proxy has unique name

key = URL = u

- calc $h(\text{proxy}_n, u)$ for all proxies
- assign u to proxy with highest $h(\text{proxy}_n, u)$

if proxy added or removed, u is likely still in correct proxy



CARP (2)

- circa 1997
  - Internet draft: Valloppillil and Ross
- Implemented in Microsoft & Netscape products
- Browsers obtain script for hashing from proxy automatic configuration file (loads automatically)

Not good for P2P:

- Each node needs to know name of all other up nodes
- i.e., need to know O(N) neighbors
- But only O(1) hops in lookup

3. Structured P2P: DHT Approaches

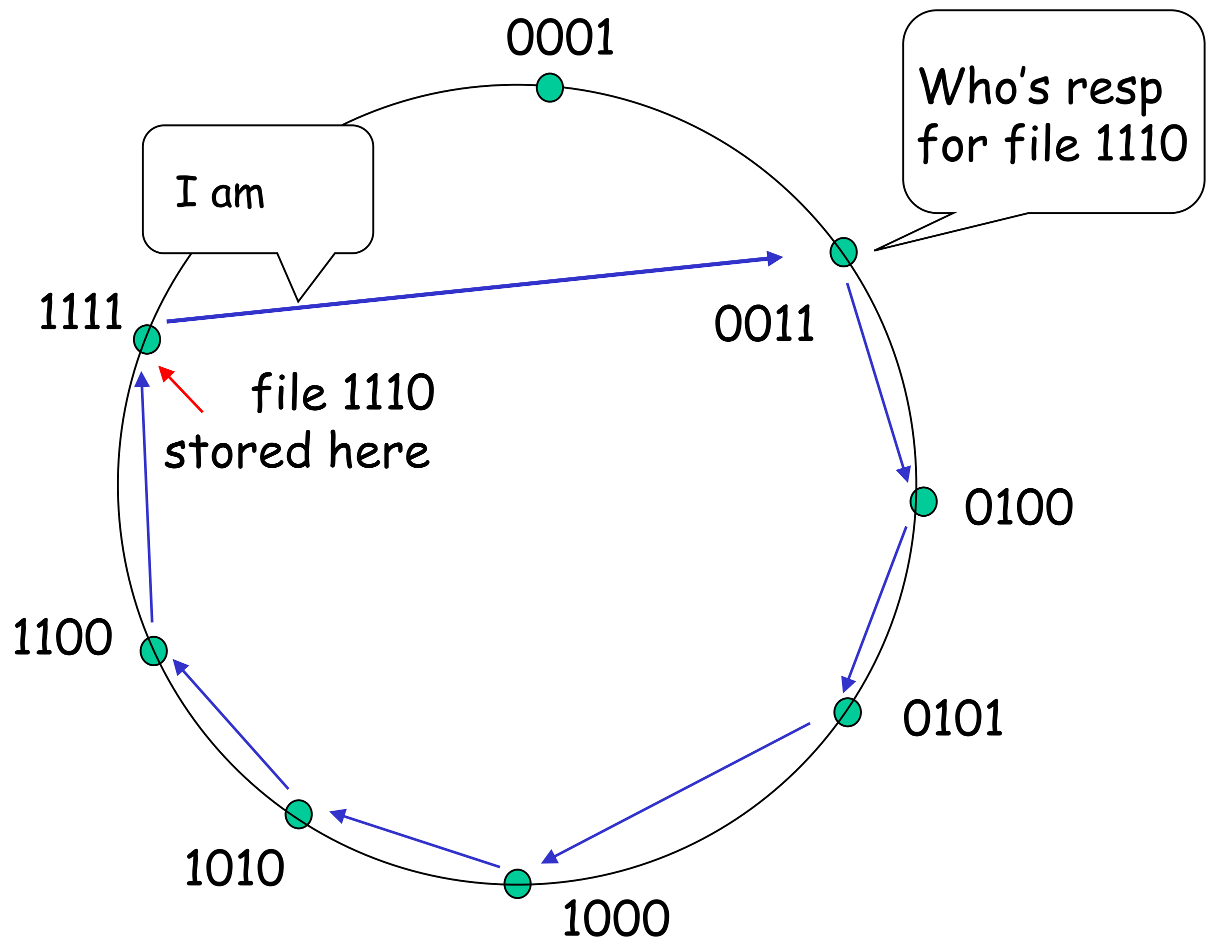
- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services
- Topology-centric lookup service

Consistent hashing (1)

- Overlay network is a circle
- Each node has randomly chosen id
  - Keys in same id space
- Node's successor in circle is node with next largest id
  - Each node knows IP address of its successor
- Key is stored in closest successor

Consistent hashing (2)

O(N) messages on avg to resolve query



Note: no locality among neighbors

Consistent hashing (3)

Node departures

- Each node must track $s \geq 2$ successors
- If your successor leaves, take next one
- Ask your new successor for list of its successors; update your s successors

Node joins

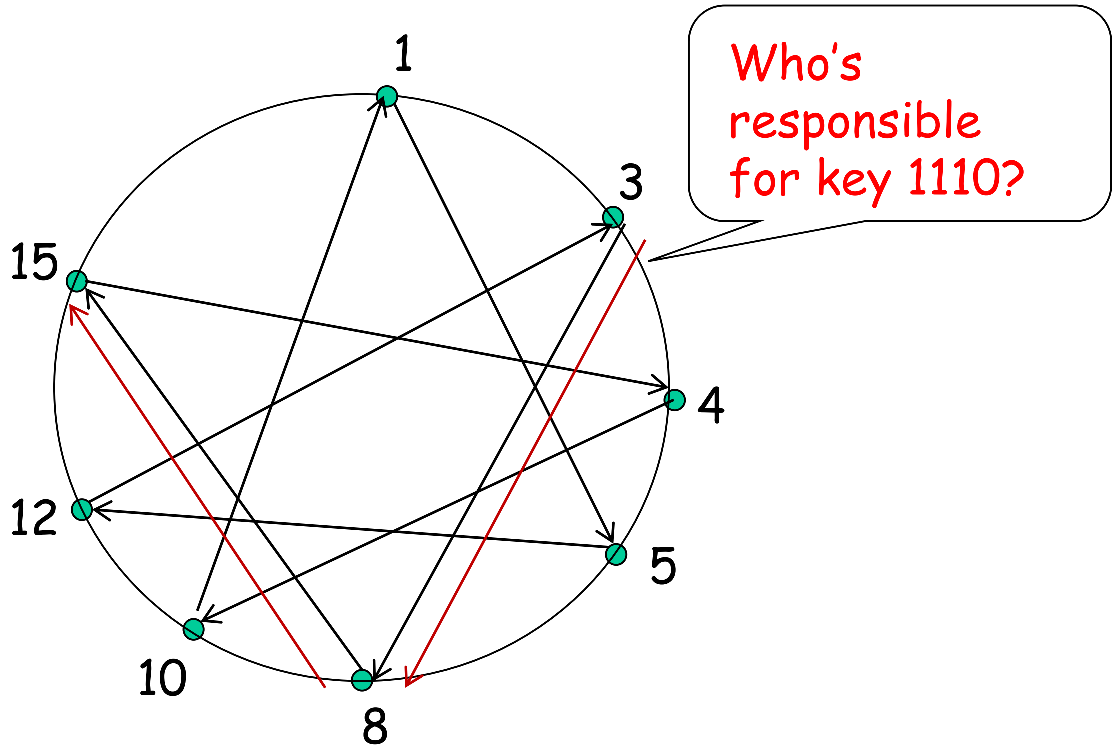
- You're new, node id k
- ask any node n to find the node n' that is the successor for id k
- Get successor list from n'
- Tell your predecessors to update their successor lists
- Thus, each node must track its predecessor

Consistent hashing (4)

- Overlay is actually a circle with small chords for tracking predecessor and k successors
- # of neighbors = s+1: O(1)
  - The ids of your neighbors along with their IP addresses is your “routing table”
- average # of messages to find key is O(N)

Can we do better?

Circular DHT with Shortcuts



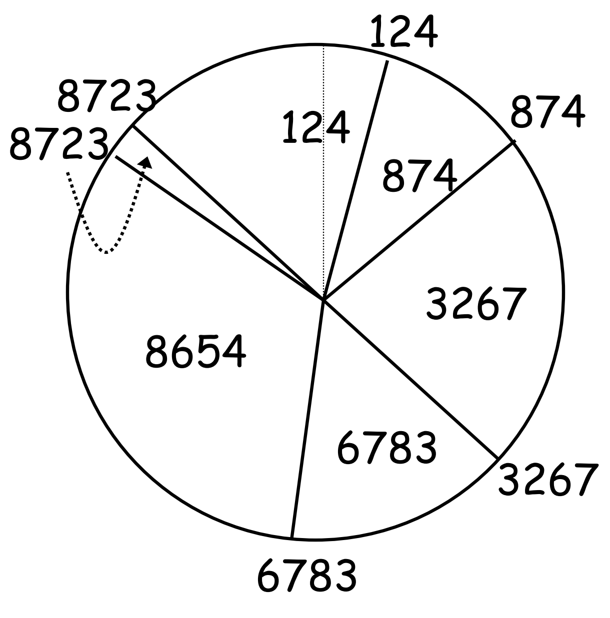
- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 2 messages.
- Possible to design shortcuts so O(log N) neighbors, O(log N) messages in query

3. Structured P2P: DHT Approaches

- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services
- Topology-centric lookup service

Chord

- Nodes assigned 1-dimensional IDs in hash space at random (e.g., hash on IP address)
- Consistent hashing: Range covered by node is from previous ID up to its own ID (modulo the ID space)



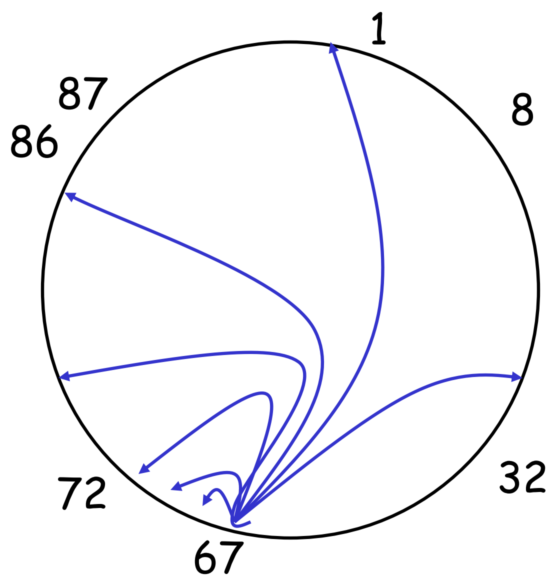
Chord Routing

- A node s's i^{th} neighbor has the ID that is equal to $s+2^i$ or is the next largest ID (mod ID space), $i \geq 0$
- To reach the node handling ID t, send the message to neighbor $\#\log_2(t-s)$
- Requirement: each node s must know about the next node that exists clockwise on the Chord (0^{th} neighbor)
- Set of known neighbors called a finger table

Chord Routing (cont'd)

- A node s is node t's neighbor if s is the closest node to $t+2^i$ mod H for some i. Thus,
  - each node has at most $\log_2 N$ neighbors
  - for any object, the node whose range contains the object is reachable from any node in no more than $\log_2 N$ overlay hops

    (each step can always traverse at least half the distance to the ID)
- Given K objects, with high probability each node has at most $(1 + \log_2 N)\, K / N$ in its range
- When a new node joins or leaves the overlay, O(K / N) objects move between nodes



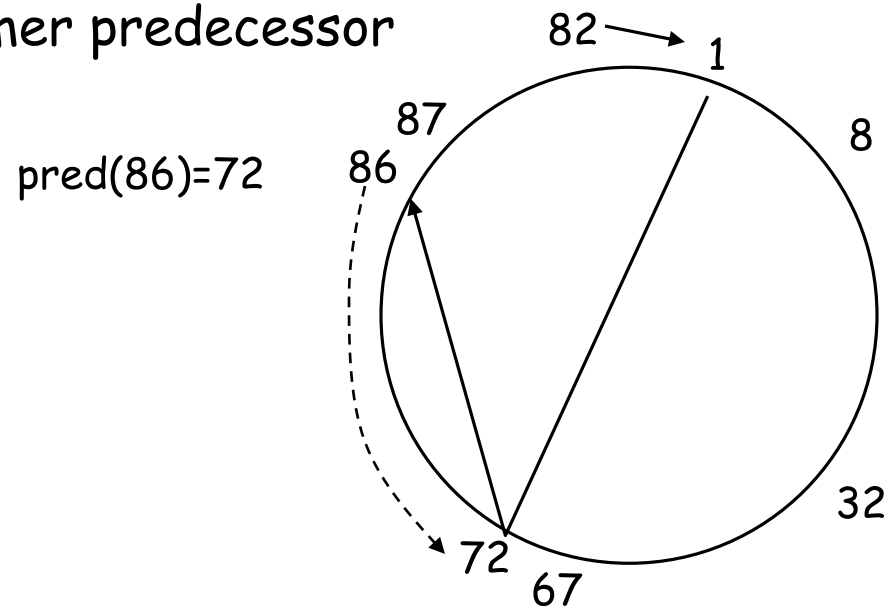
i	Finger table for node 67
0	72
1	72
2	72
3	86
4	86
5	1
6	32

Closest node clockwise to

$67+2^i$ mod 100

Chord Node Insertion

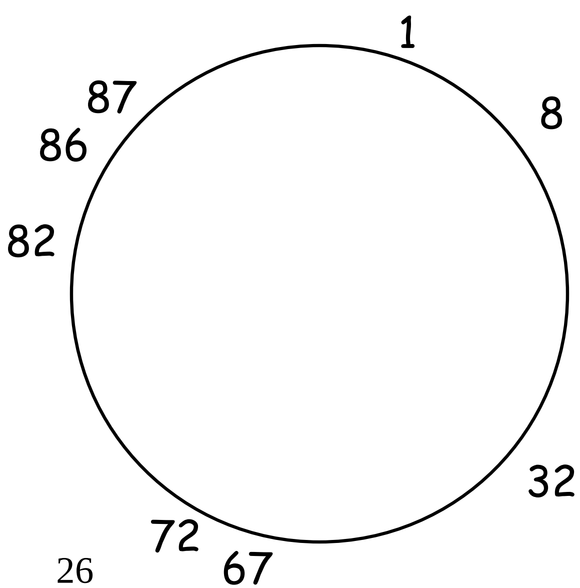
- One protocol addition: each node knows its closest counter-clockwise neighbor
- A node selects its unique (pseudo-random) ID and uses a bootstrapping process to find some node in the Chord
- Using Chord, the node identifies its successor in the clockwise direction
- An newly inserted node's predecessor is its successor's former predecessor



Example: Insert 82

Chord Node Insertion (cont'd)

- First: set added node s's fingers correctly
  - s's predecessor t does the lookup for each distance of 2^i from s



Lookups from node 72

Lookup(83) = 86

Lookup(84) = 86

Lookup(86) = 86

Lookup(90) = 1

Lookup(98) = 1

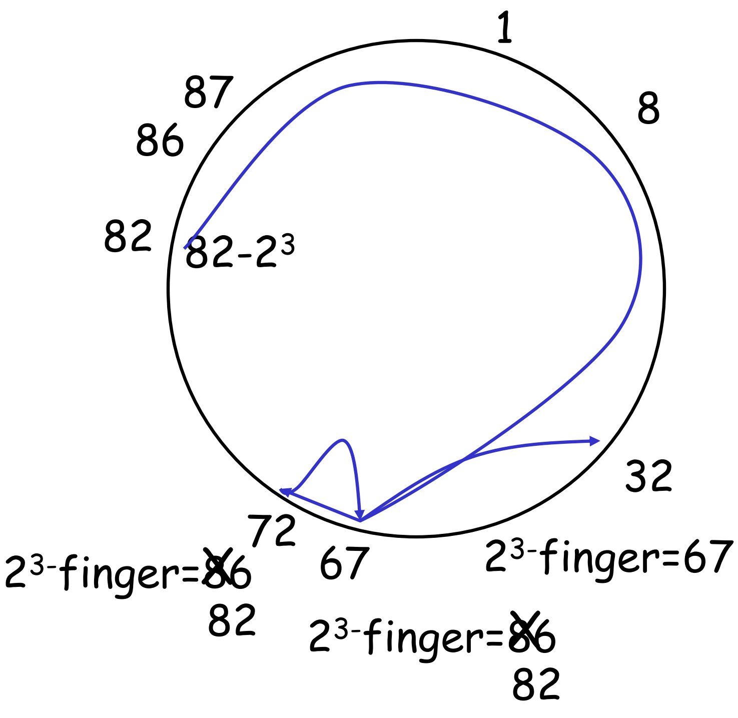
Lookup(14) = 32

Lookup(46) = 67

i	Finger table for node 82
0	86
1	86
2	86
3	1
4	1
5	32
6	67

Chord Node Insertion (cont'd)

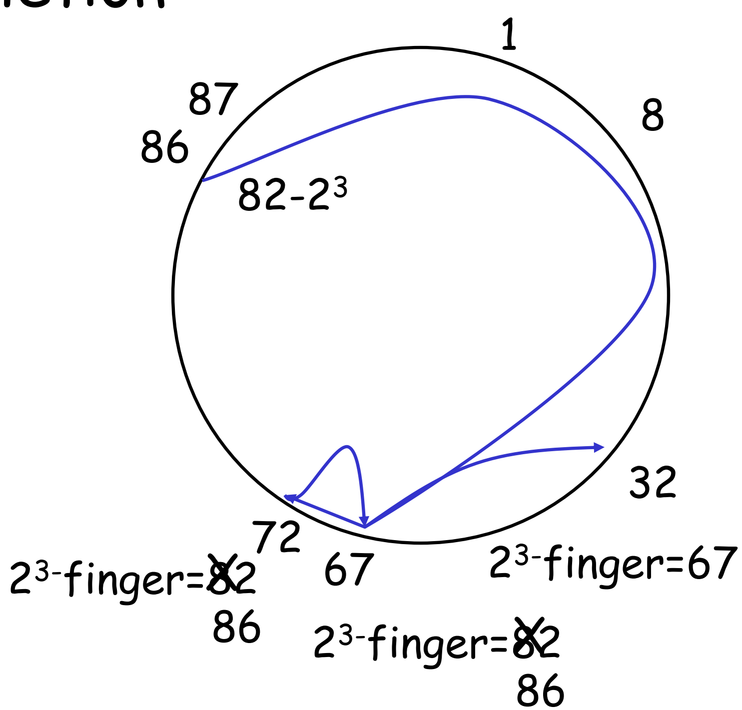
- Next, update other nodes' fingers about the entrance of s (when relevant). For each i:
  - Locate the closest node to s (counter-clockwise) whose 2^i-finger can point to s: largest possible is $s - 2^i$
  - Use Chord to go (clockwise) to largest node t before or at $s - 2^i$
    - route to $s - 2^i$, if arrived at a larger node, select its predecessor as t
  - If t's 2^i-finger routes to a node larger than s
    - change t's 2^i-finger to s
    - set t = predecessor of t and repeat
  - Else i++, repeat from top
- $O(\log^2 N)$ time to find and update nodes



e.g., for i=3

Chord Node Deletion

- Similar process can perform deletion



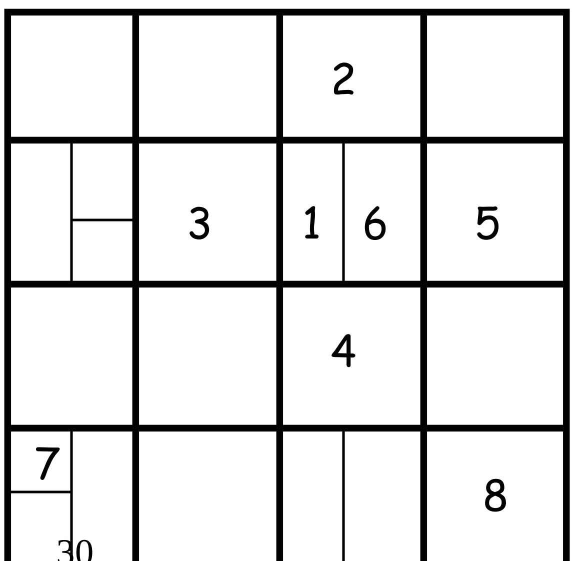
e.g., for i=3

3. Structured P2P: DHT Approaches

- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services
- Topology-centric lookup service

CAN

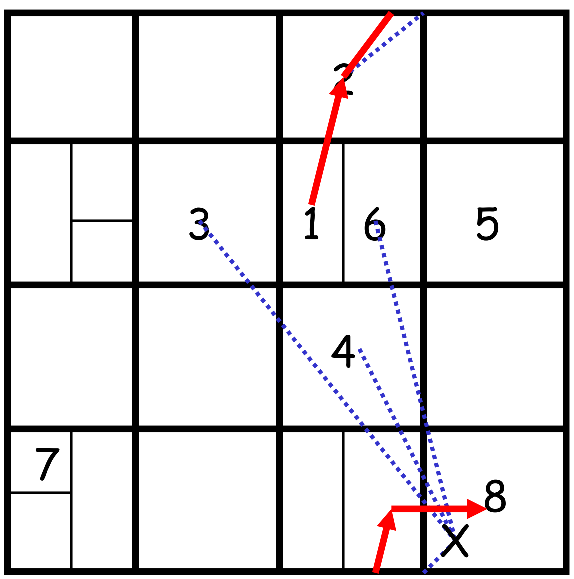
- hash value is viewed as a point in a D-dimensional cartesian space
- each node responsible for a D-dimensional "cube" in the space
- nodes are neighbors if their cubes "touch" at more than just a point (more formally, nodes s & t are neighbors when
  - s contains some
    $[\langle n_1, n_2, \ldots, n_i, \ldots, n_j, \ldots, n_D\rangle, \langle n_1, n_2, \ldots, m_i, \ldots, n_j, \ldots n_D\rangle]$
  - and t contains
    $[\langle n_1, n_2, \ldots, n_i, \ldots, n_j+\delta, \ldots, n_D\rangle, \langle n_1, n_2, \ldots, m_i, \ldots, n_j+\delta, \ldots n_D\rangle]$)



- Example: D=2
- 1's neighbors: 2,3,4,6
- 6's neighbors: 1,2,4,5
- Squares "wrap around", e.g., 7 and 8 are neighbors
- expected # neighbors: O(D)

CAN routing

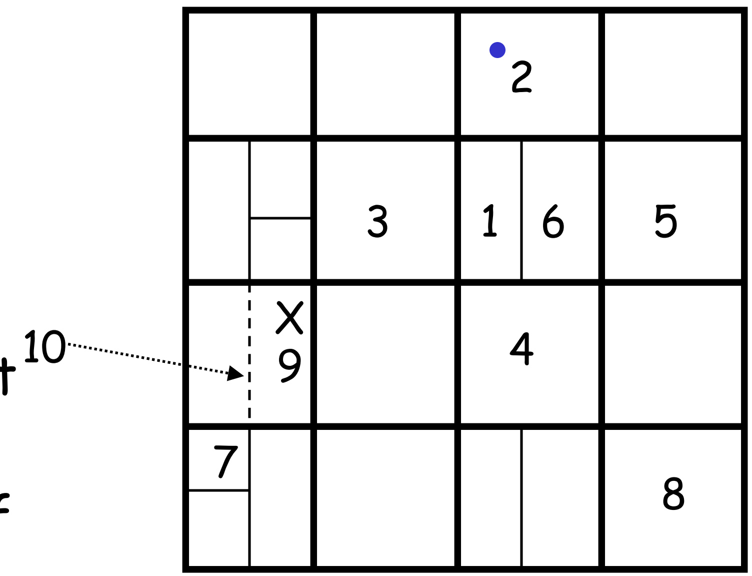
- To get to $\langle n_1, n_2, \ldots, n_D \rangle$ from $\langle m_1, m_2, \ldots, m_D \rangle$
  - choose a neighbor with smallest cartesian distance from $\langle m_1, m_2, \ldots, m_D \rangle$ (e.g., measured from neighbor's center)



- e.g., region 1 needs to send to node covering X
- checks all neighbors, node 2 is closest
- forwards message to node 2
- Cartesian distance monotonically decreases with each transmission
- expected # overlay hops: $(DN^{1/D})/4$

CAN node insertion

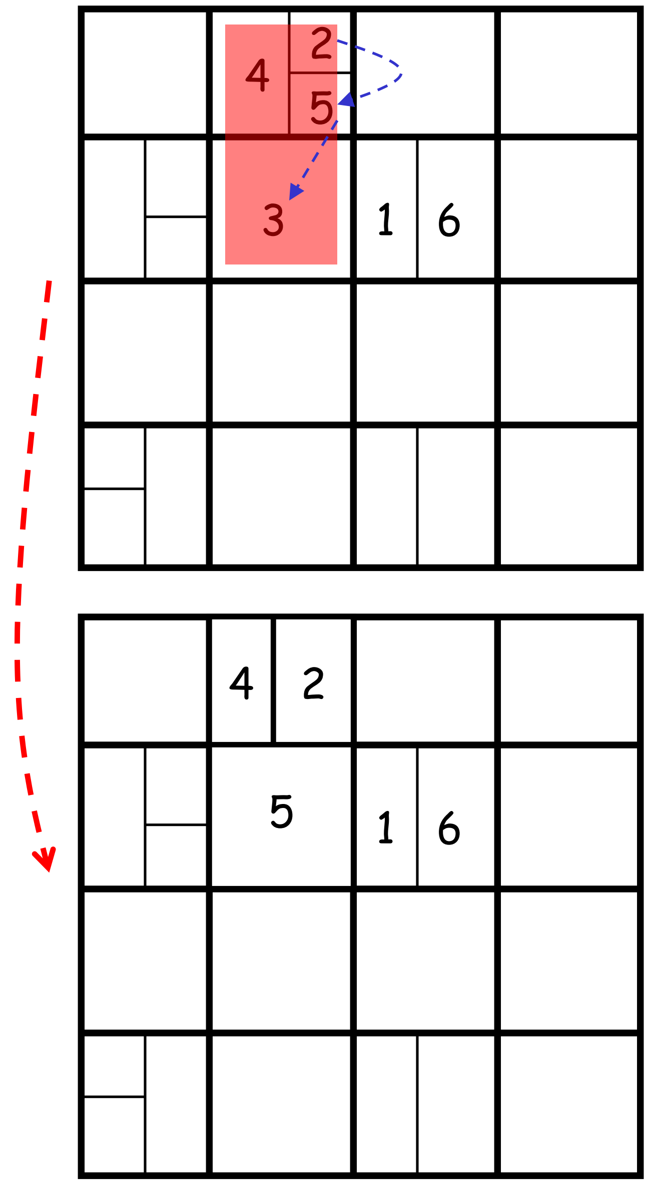
- To join the CAN:
  - find some node in the CAN (via bootstrap process)
  - choose a point in the space uniformly at random
  - using CAN, inform the node that currently covers the space
  - that node splits its space in half
    - 1st split along 1st dimension
    - if last split along dimension i < D, next split along i+1st dimension
    - e.g., for 2-d case, split on x-axis, then y-axis
  - keeps half the space and gives other half to joining node



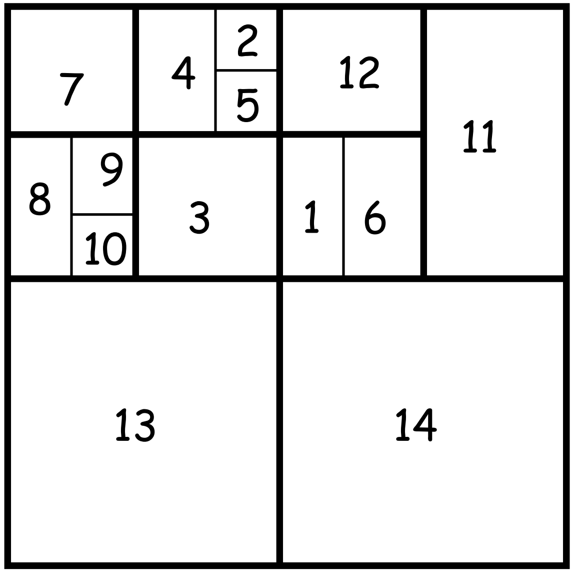
Observation: the likelihood of a rectangle being selected is proportional to it's size, i.e., big rectangles chosen more frequently

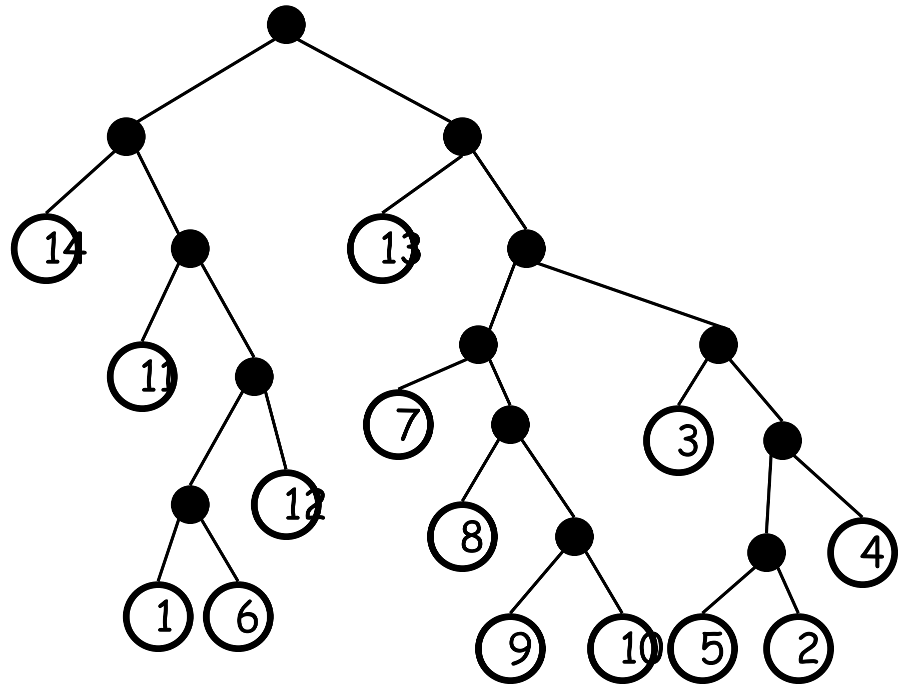
CAN node removal

- Underlying cube structure should remain intact
  - i.e., if the spaces covered by s & t were not formed by splitting a cube, then they should not be merged together
- Sometimes, can simply collapse removed node's portion to form bigger rectangle
  - e.g., if 6 leaves, its portion goes back to 1
- Other times, requires juxtaposition of nodes' areas of coverage
  - e.g., if 3 leaves, should merge back into square formed by 2,4,5
  - cannot simply collapse 3's space into 4 and/or 5
  - one solution: 5's old space collapses into 2's space, 5 takes over 3's space



CAN (recovery from) removal process





- View partitioning as a binary tree of
  - leaves represent regions covered by overlay nodes (labeled by node that covers the region)
  - intermediate nodes represent "split" regions that could be "reformed", i.e., a leaf can appear at that position
  - siblings are regions that can be merged together (forming the region that is covered by their parent)

3. Structured P2P: DHT Approaches

- DHT service and issues
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry
- Hierarchical lookup services
- Topology-centric lookup service

Routing table (node: 65a1fc04)

Row 0	***0x***	***1x***	***2x***	***3x***	***4x***	***5x***		***7x***	***8x***	***9x***	***ax***	***bx***	***cx***	***dx***	***ex***	***fx***
Row 1	***60x***	***61x***	***62x***	***63x***	***64x***		***66x***	***67x***	***68x***	***69x***	***6ax***	***6bx***	***6cx***	***6dx***	***6ex***	***6fx***
Row 2	***650x***	***651x***	***652x***	***653x***	***654x***	***655x***	***656x***	***657x***	***658x***	***659x***		***65bx***	***65cx***	***65dx***	***65ex***	***65fx***
Row 3	***65a0x***		***65a2x***	***65a3x***	***65a4x***	***65a5x***	***65a6x***	***65a7x***	***65a8x***	***65a9x***	***65aax***	***65abx***	***65acx***	***65adx***	***65aex***	***65afx***

$\log_{16} N$ rows

Pastry: Routing procedure

if (destination is within range of our leaf set)
    forward to numerically closest member
else
    if (there's a longer prefix match in table)
      forward to node with longest match
    else
        forward to node in table
        (a) shares at least as long a prefix
        (b) is numerically closer than this node

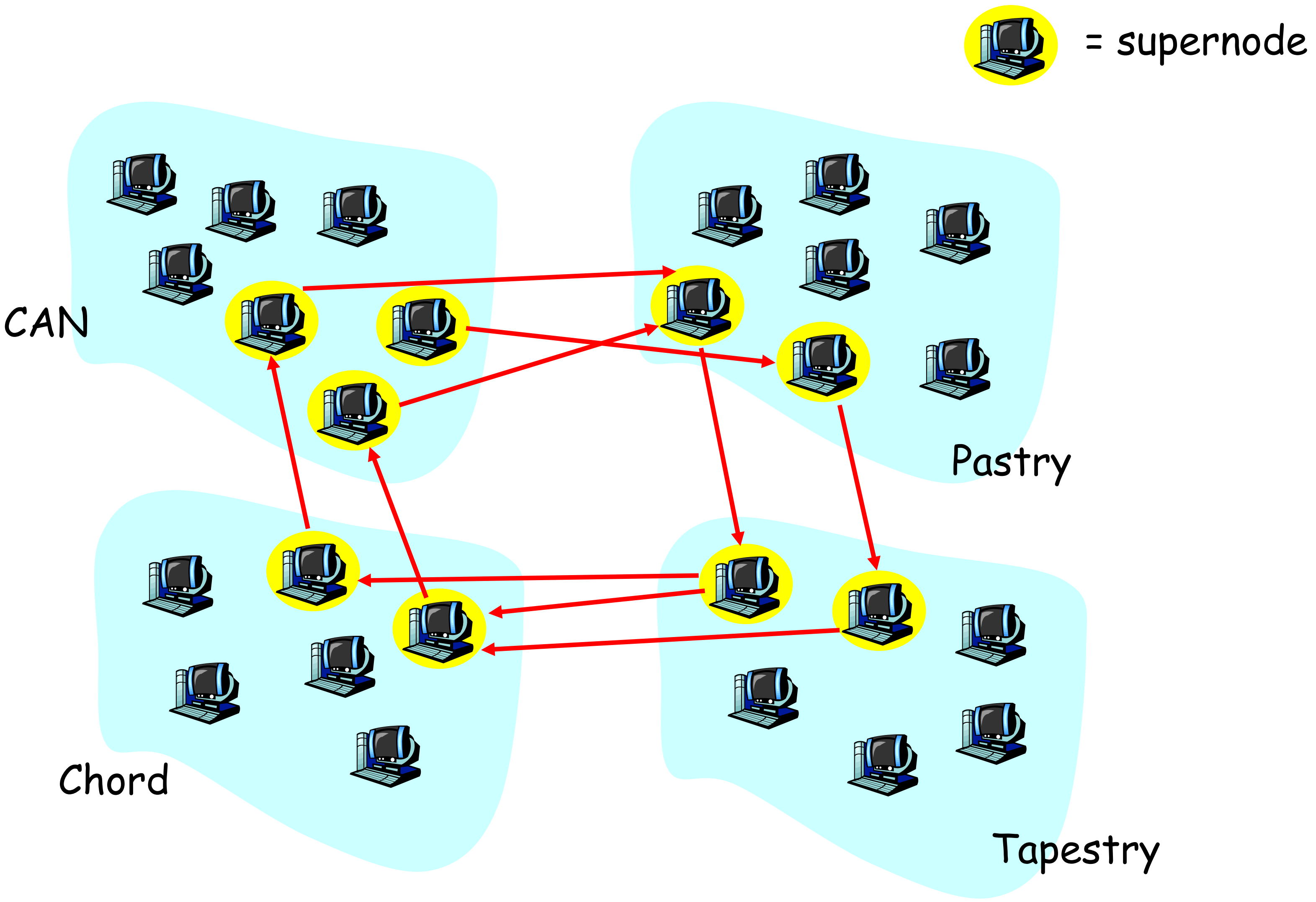
3. Structured P2P: DHT Approaches

- The DHT service and API
- CARP
- Consistent Hashing
- Chord
- CAN
- Pastry/Tapestry
- Hierarchical lookup services

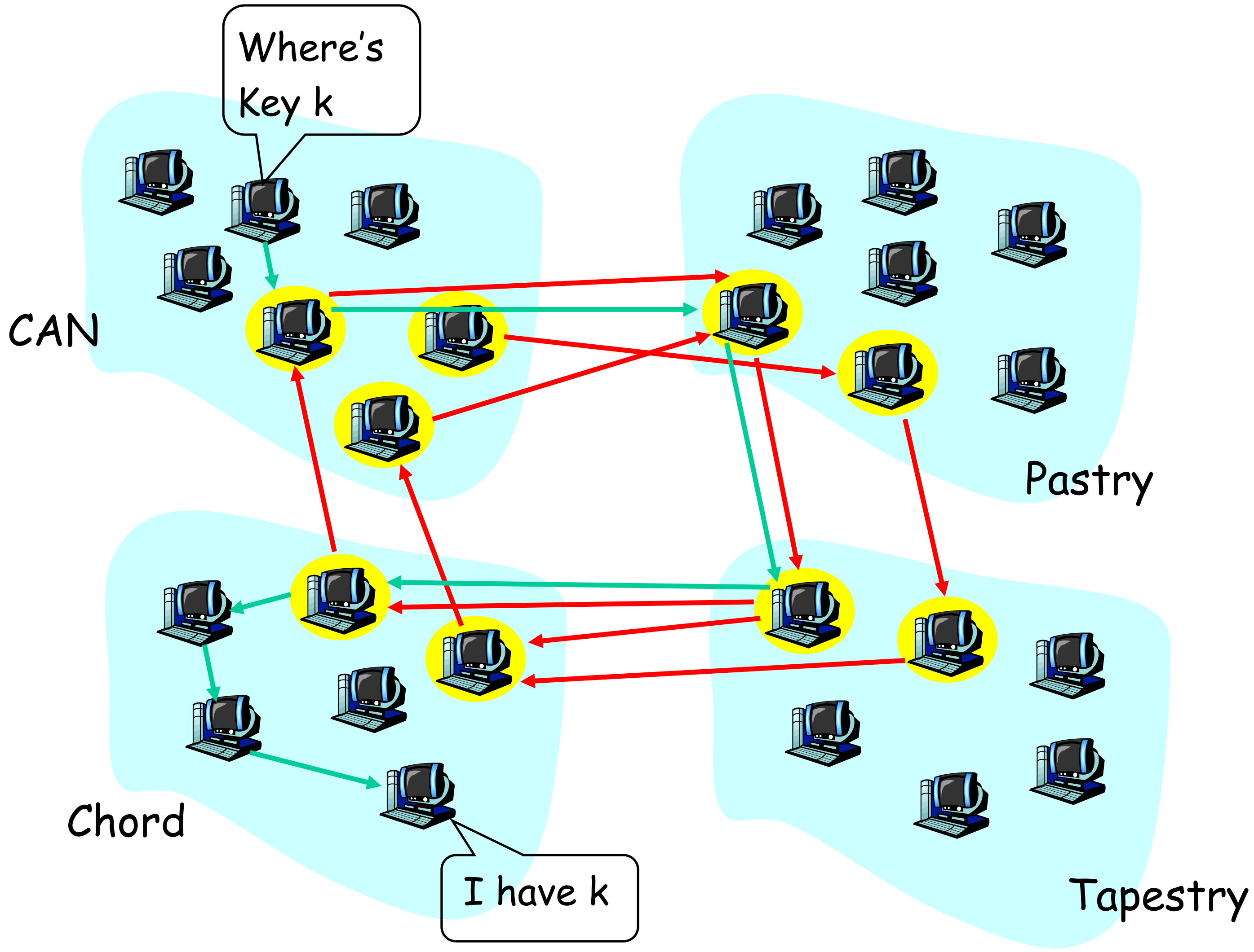
Hierarchical Lookup Service

- KaZaA is hierarchical but unstructured
- Routing in Internet is hierarchical
- Perhaps DHTs can benefit from hierarchies too?
  - Peers are organized into groups
  - Inter-group lookup, then intra-group lookup

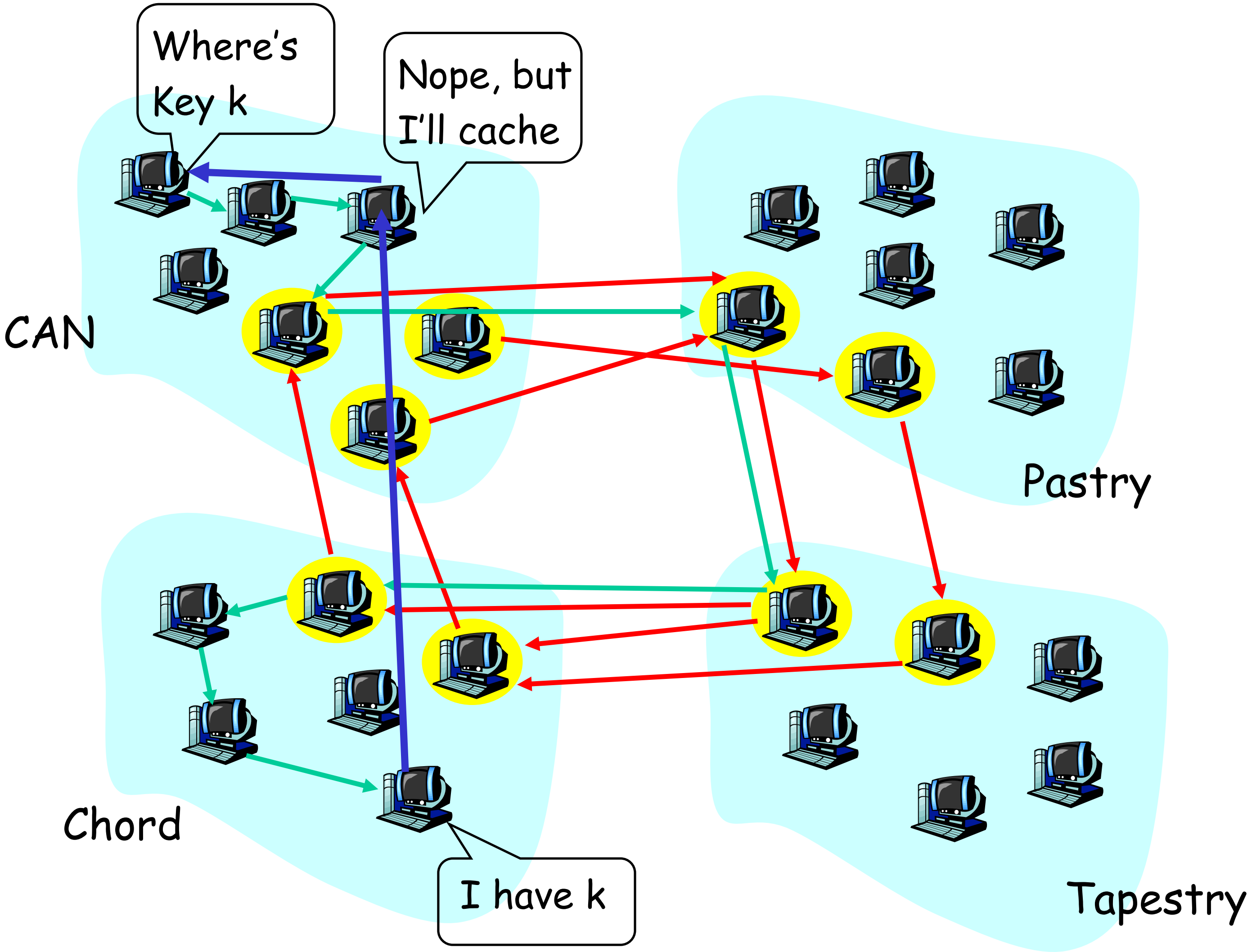
Hierarchical framework



Hierarchical Lookup



Cooperative group caching



Hierarchical Lookup (2)

Benefits

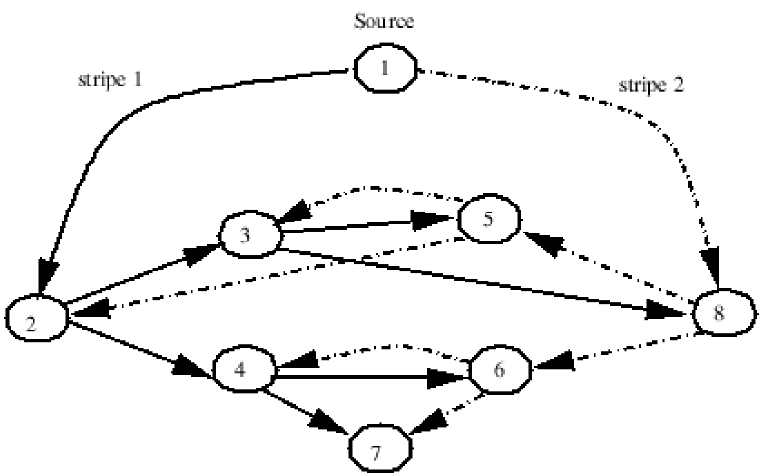
- Can reduce number of hops, particularly when nodes have heterogeneous availabilities
- Groups can cooperatively cache popular files, reducing average latency
- Facilitates large scale deployment by providing administrative autonomy to groups
  - Each ISP can use it is own DHT protocol
  - Similar to intra-AS routing

SplitStream Multicast

- Balance load over peers
- Accommodate different limitations
  - Each node has a desired indegree and a forwarding capacity (max outdegree)
- Be robust to failures

The SplitStream approach

- Split data into stripes, each over its own tree
- Each node is internal to only one tree
- Built on Pastry and Scribe
  - Recall that Pastry uses prefix routing



Scribe background

- Built on top of Pastry
- Any Scribe node may create a group
  - Other nodes may join group or send multicast
- Node with nodeId numerically closest to groupId is the *rendezvous point*
  - Root of multicast tree for the group
  - Joins handled locally
- But it's only a single tree

Stripes

- SplitStream divides data into stripes
- Each stripe uses one Scribe multicast tree
- Prefix routing ensures property that each node is internal to only one tree
  - Inbound bandwidth: can achieve desired indegree while this property holds
  - Outbound bandwidth: this is harder—we'll have to look at the node join algorithm to see how this works

Respecting forwarding capacity

- The tree structure described may not respect maximum capacities
- Scribe's push-down fails to resolve the problem because a leaf node in one tree may have children in another tree