Lecture Coverage

1. Characteristics, Properties and Requirements
2. Survey of Approaches
3. Napster/Gnutella
4. Distributed Hash Table Based Systems
    1. Chord
    2. Can
5. Similarity Metric Based Systems
    1. Freenet

Pure P2P Systems- Properties

Fully distributed, fully symmetric control

Common knowledge – Initialization and by Discovery

Discovery is essential

Discovery is via peer interaction protocols

Functionality is composed from independent components

Self-organizing behaviors

Protocols:

Join , Leave, Insertion, Discovery, Replication

Search/Discovery

- "Location Resolution"

  Given an object (might be name, attribute, or even content)

  Return a channel to a node (peer) that has that object

  Or the object itself

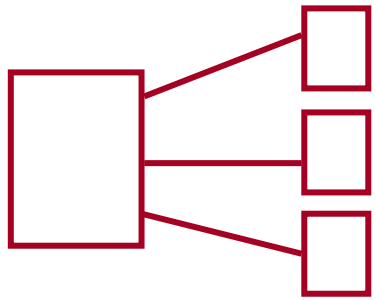
- Approaches:
  - Centralized Index (Napster)
  - Broadcast information to be resolved (Gnutella)
  - Similarity metric based searches
  - Distributed Hashing

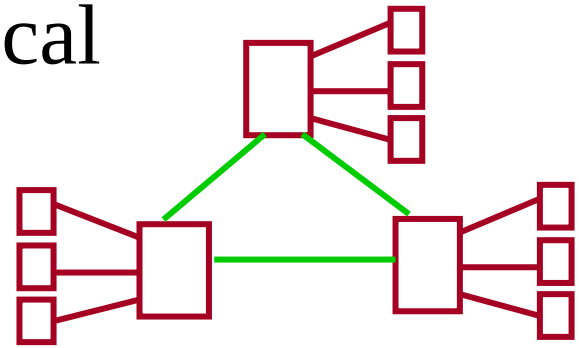
Design Goals

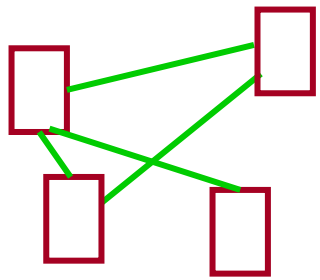
- Scalability
- Low latency (efficient resolution)
- Load balancing
- Completely distributed/self-organizing
- Robust – fault-tolerance
- Deployable
- Simple

Spectrum of "Purity"

- Hybrid
  - Centralized index, P2P file storage and transfer
- Super-peer or Hierarchical
  - A "pure" network of "hybrid" clusters
- Pure
  - functionality completely distributed







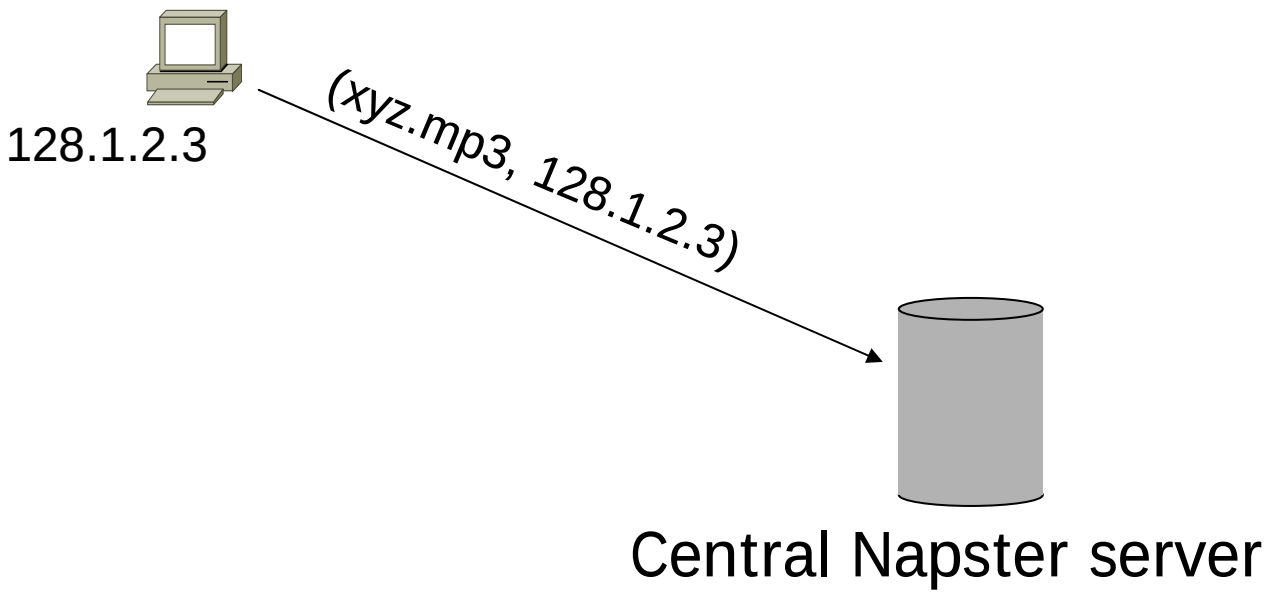
Metrics for Discovery/Insertion/Join

- Cost (aggregate)
  - Number of Messages or Interactions
  - Bandwidth
  - Processing Power
- Quality of Results
  - Number of results
  - Satisfaction (true if # results >= X, false otherwise)
  - Time to satisfaction

P2P File-sharing

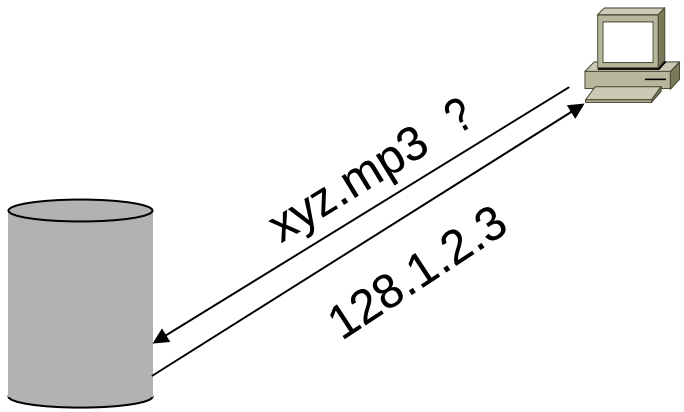
- Napster
  - Decentralized storage of actual content
    - transfer content directly from one peer (client) to another
  - Centralized index and search
- Gnutella
  - Like Napster, with decentralized indexing
  - Search via flooding
  - Direct download

Napster



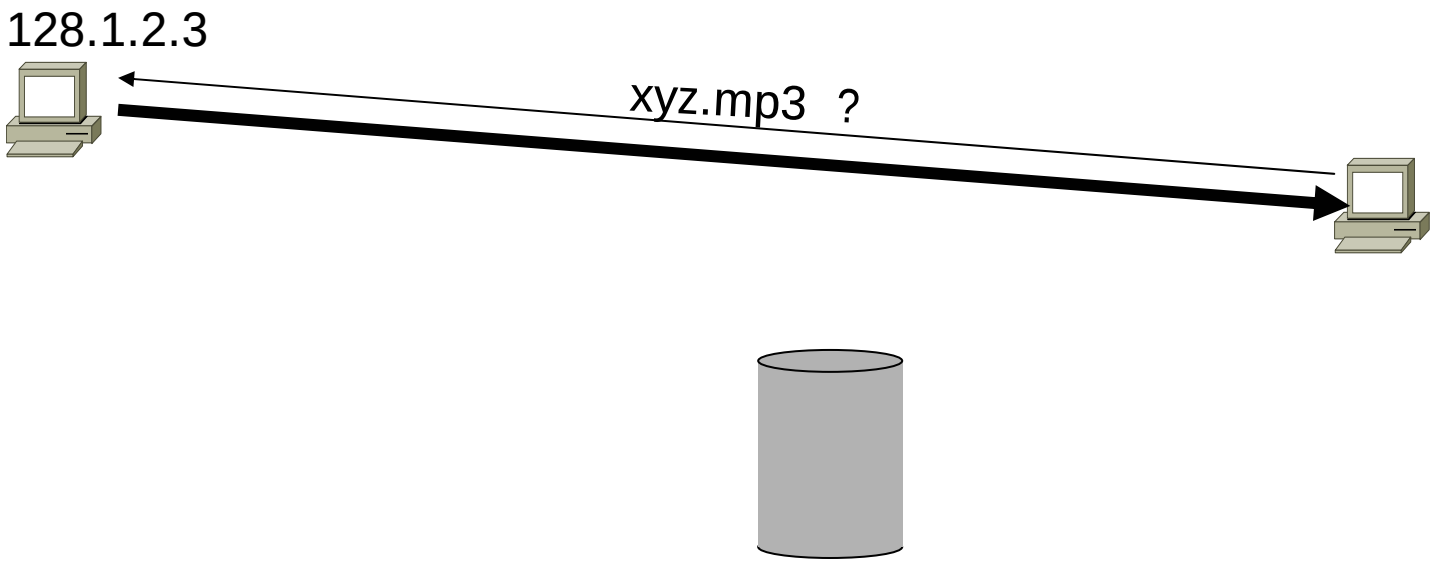
Napster

128.1.2.3

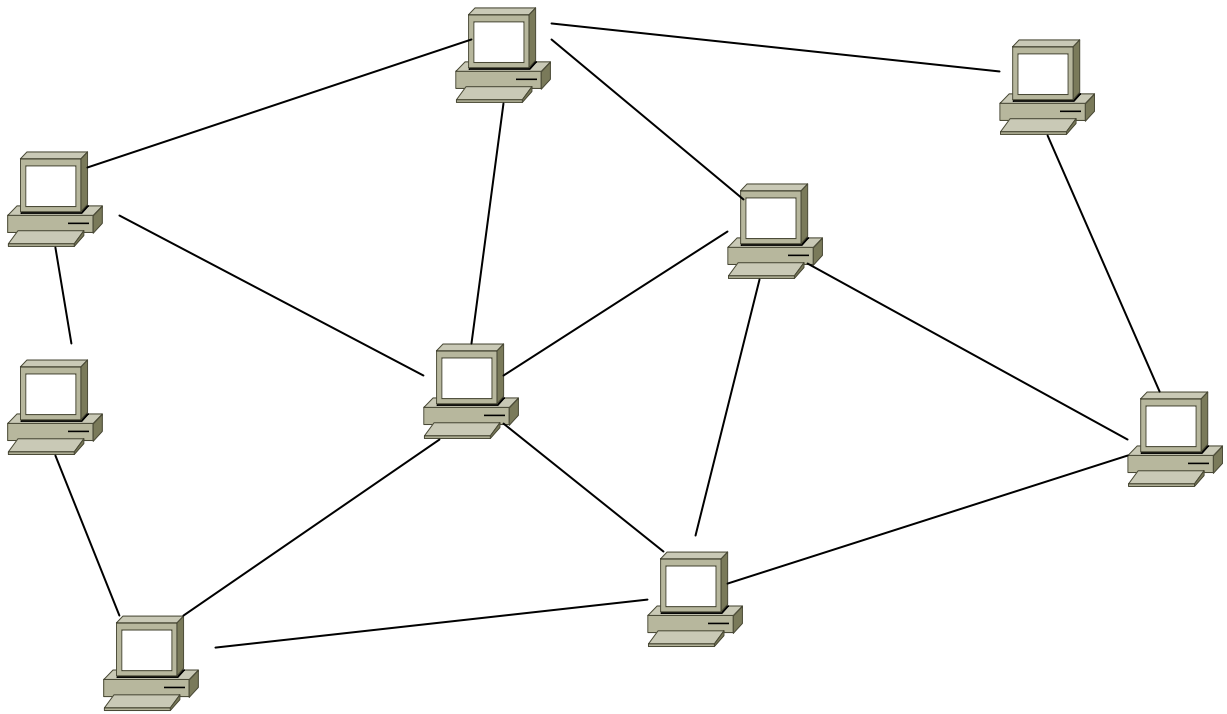


Central Napster server

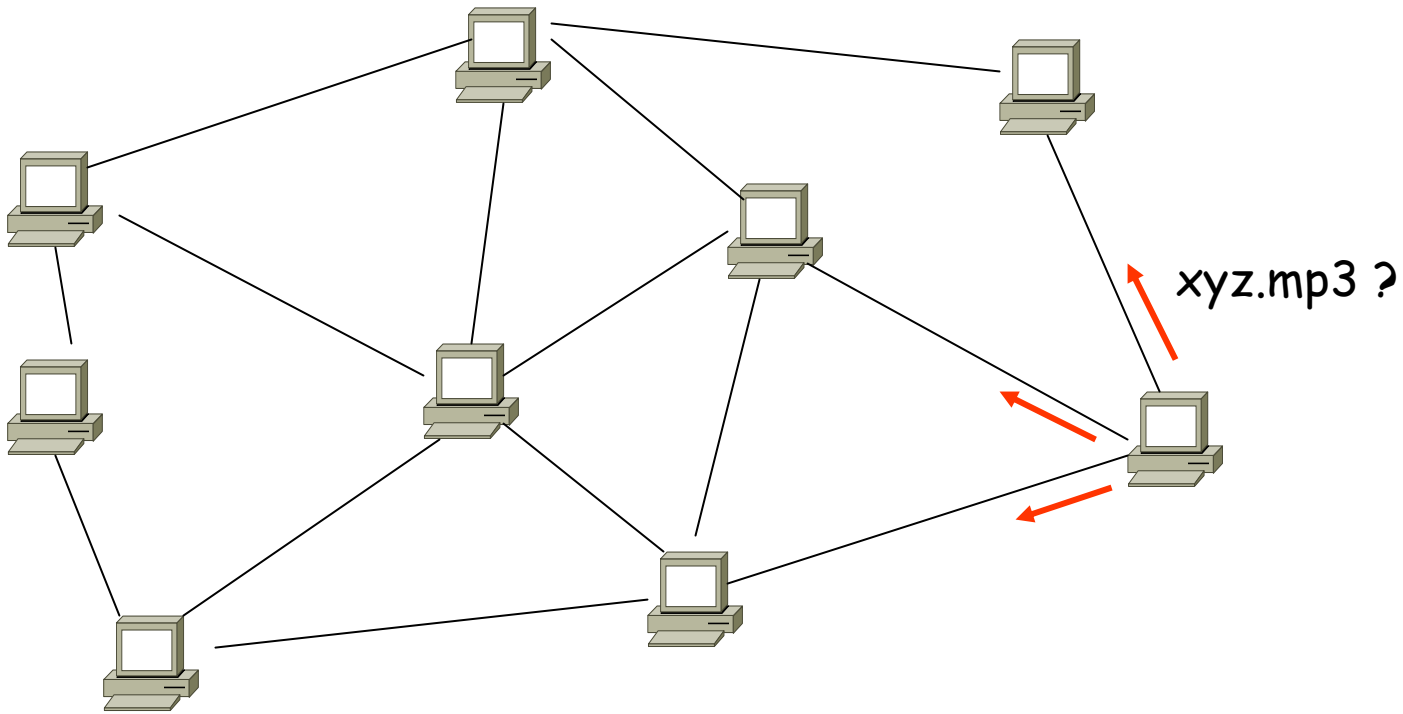
Napster



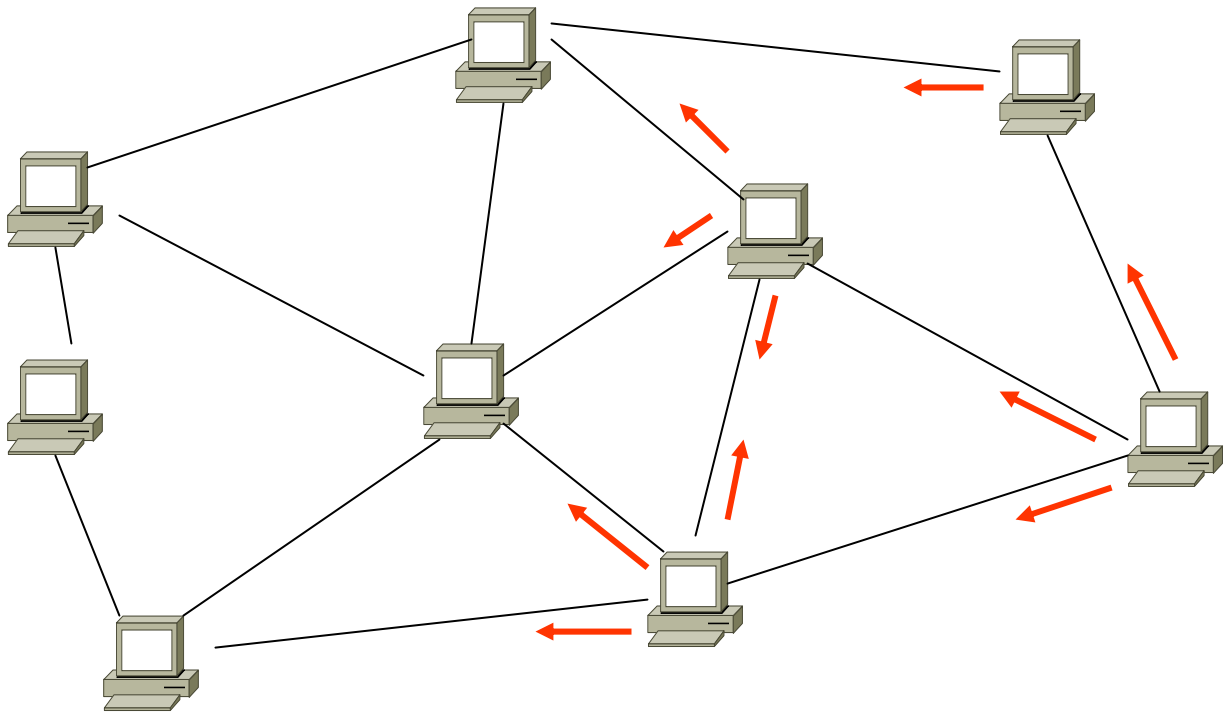
Gnutella



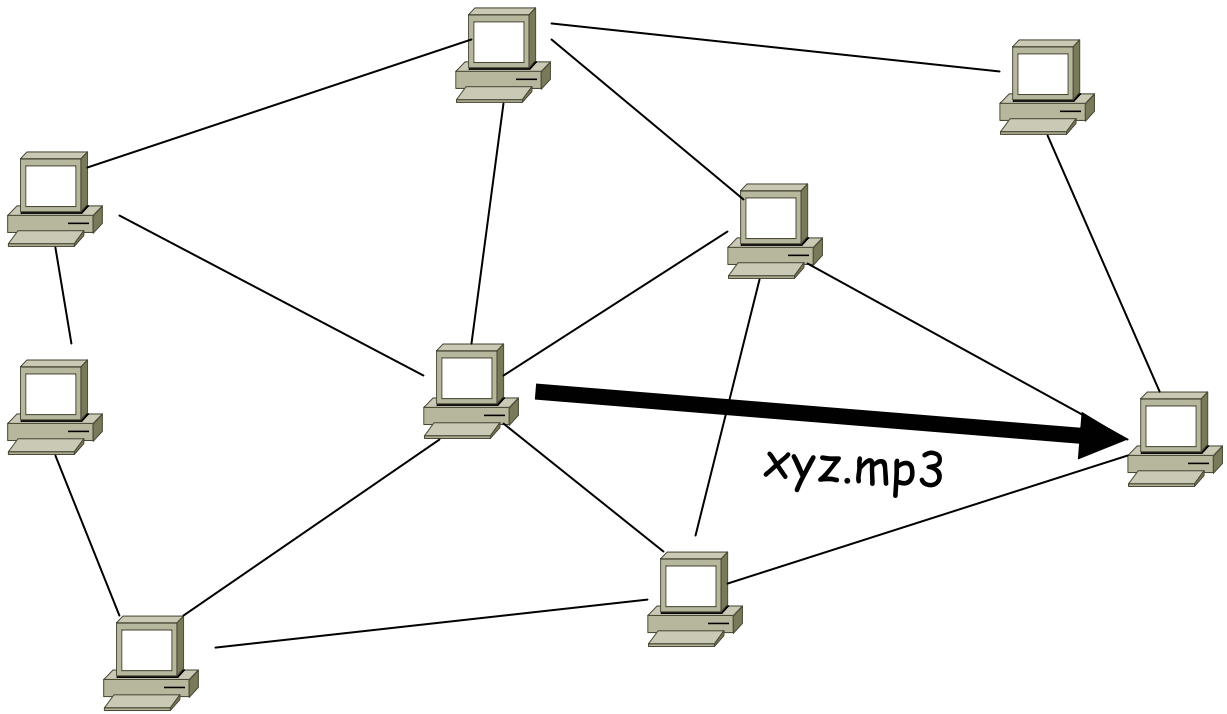
Gnutella



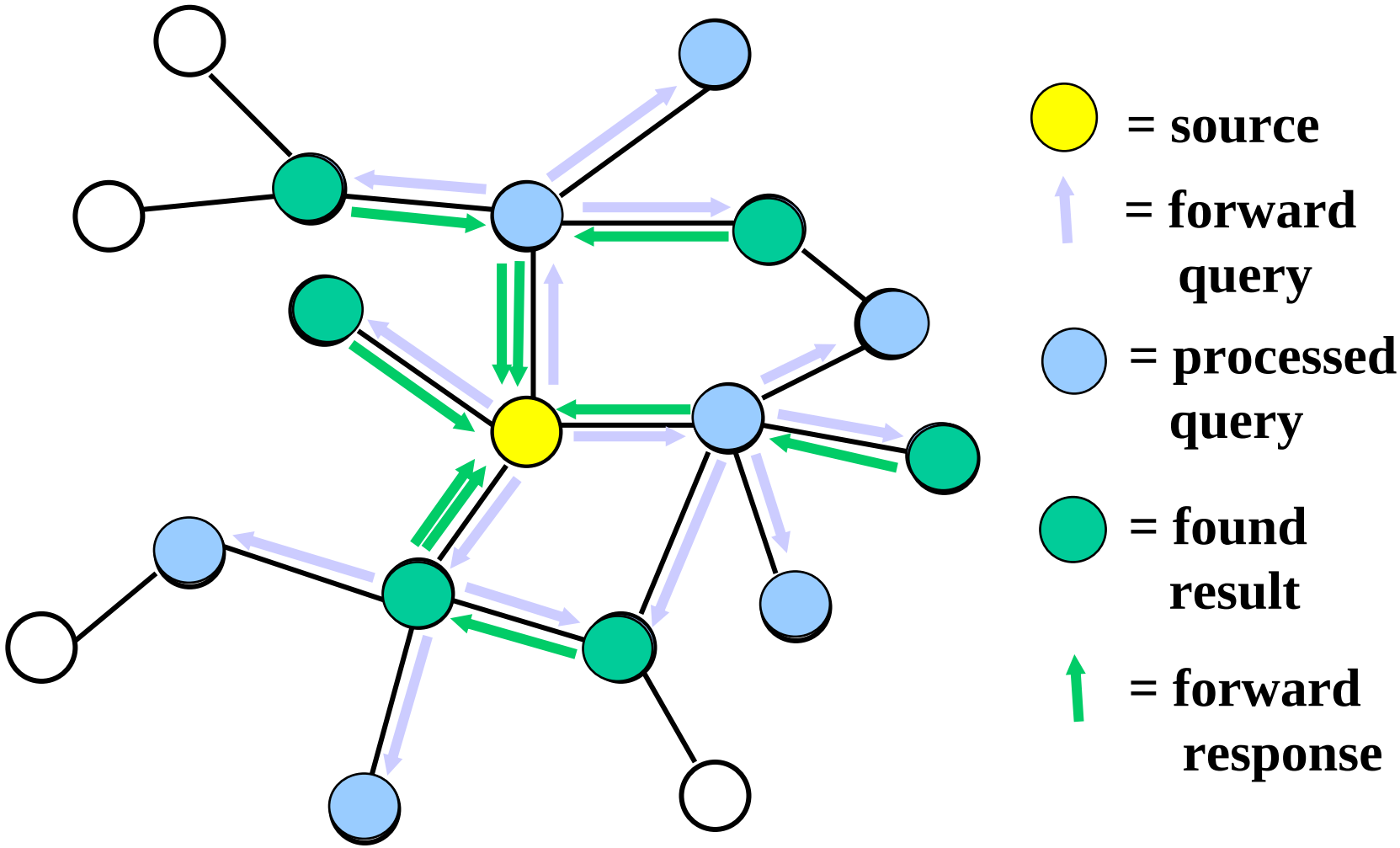
Gnutella



Gnutella

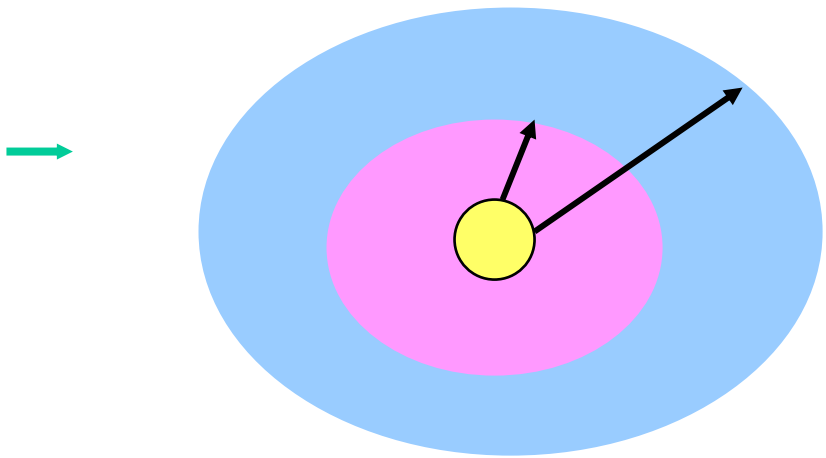


Current Techniques: Gnutella: Breadth-First Search (BFS)

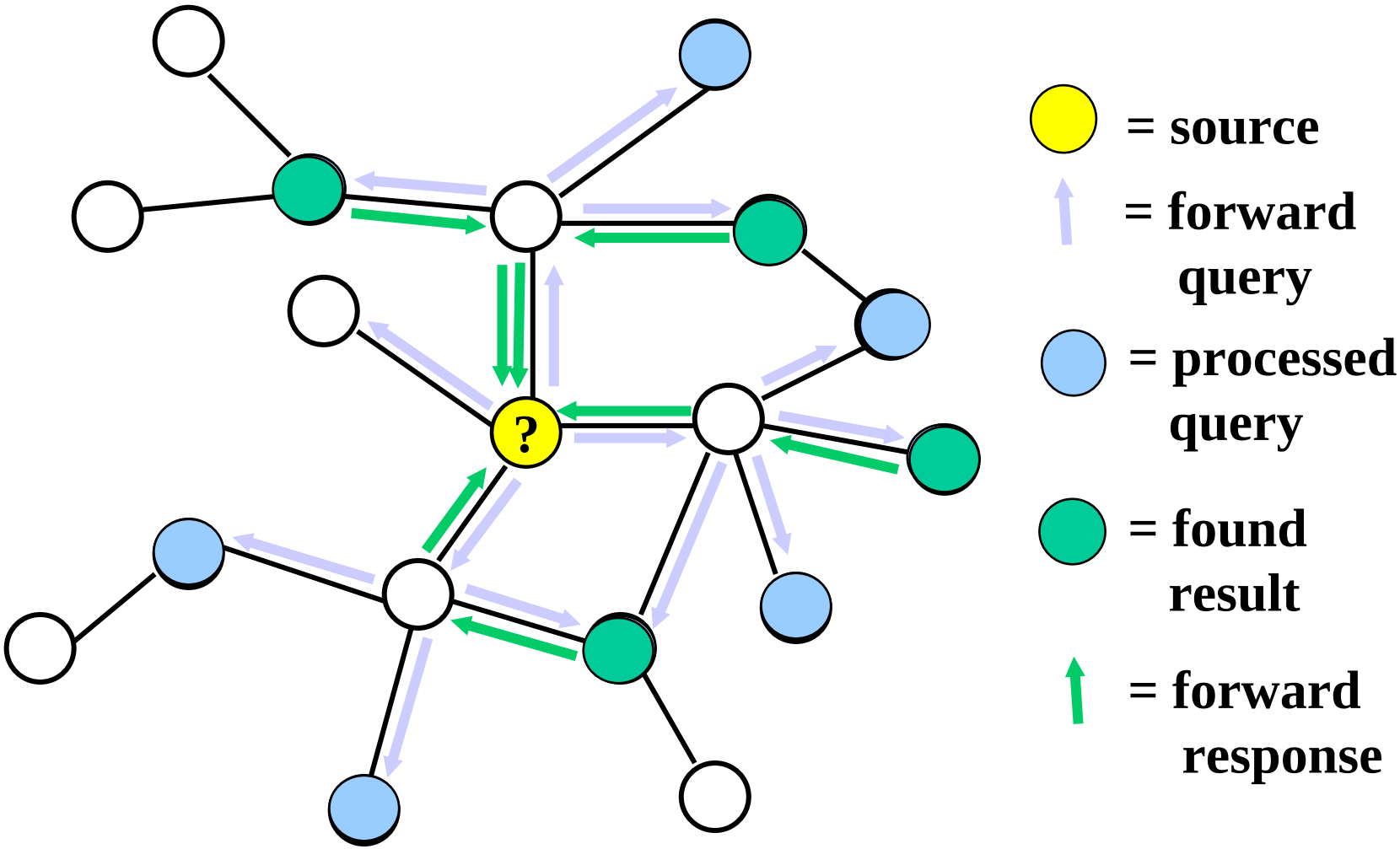


Iterative Deepening

- Interested in **satisfaction**, not # of results
- BFS returns “too many” results expensive
- Iterative Deepening: common technique to reduce the cost of BFS
  - Intuition: A search at a small depth is much cheaper than at a larger depth

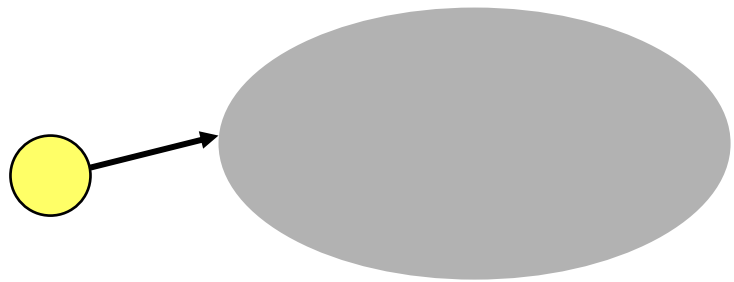


Iterative Deepening

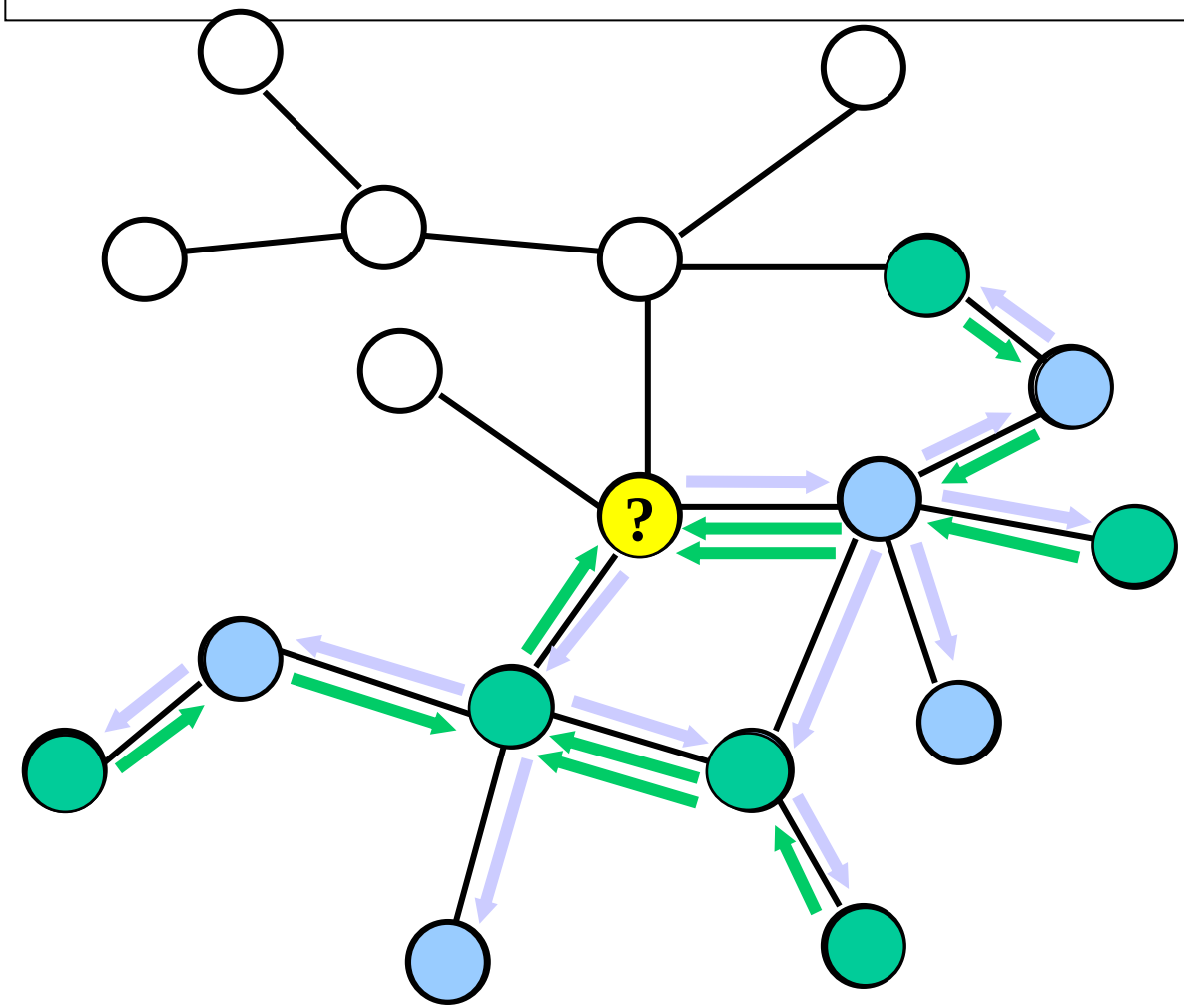


Directed BFS

- Sends query to a subset of neighbors
- Maintains statistics on neighbors
  - E.g., ping latency, history of number of results
- Chooses subset intelligently (via heuristics), to maximize quality of results
  - E.g., Neighbors with shortest message queue, since long message queue implies neighbor is saturated/dead



Directed BFS



= source

= forward query

= processed query

= found result

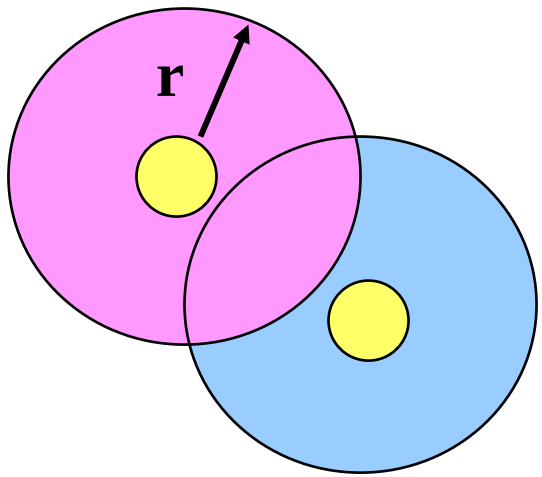
= forward response

Directed BFS: Heuristics

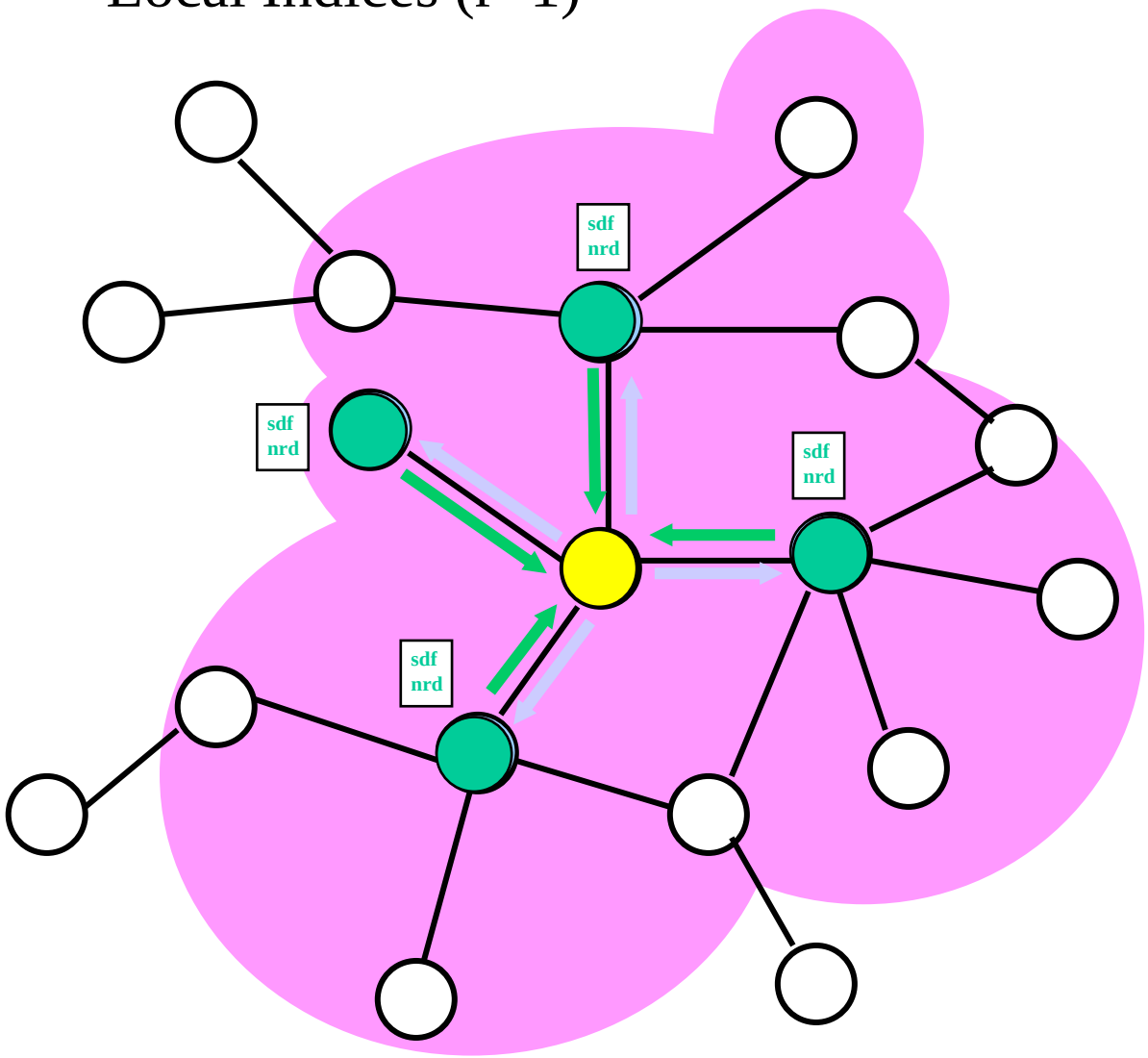
RAND	(Random)
RES	Returned greatest # results in past
TIME	Had shorted avg. time to satisfaction in past
HOPS	Had smallest avg. # hops for response messages in past
MSG	Sent our client greatest # of messages
QLEN	Shortest message queue
DEG	Highest degree

Local Indices

- Each node maintains index over other nodes' collections
  - *r* is the **radius** of the index
  - Index covers all nodes within *r* hops away
- Can process query at fewer nodes, but get just as many results back



Local Indices (r=1)



= source

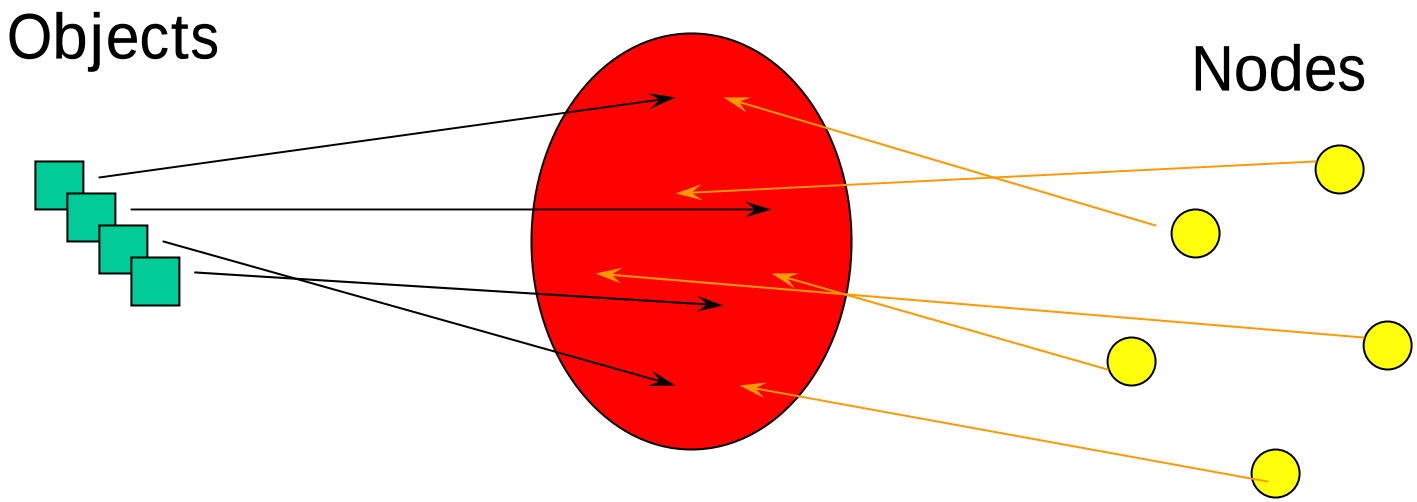
= forward query

= processed query

= found result

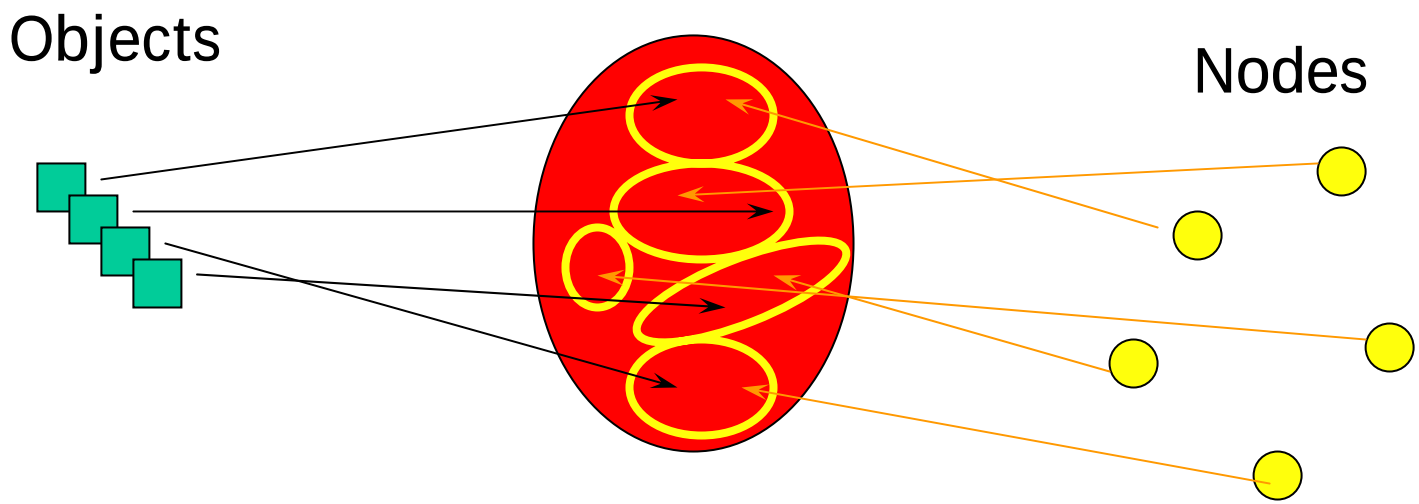
= forward response

Distributed Hashing — General Approach



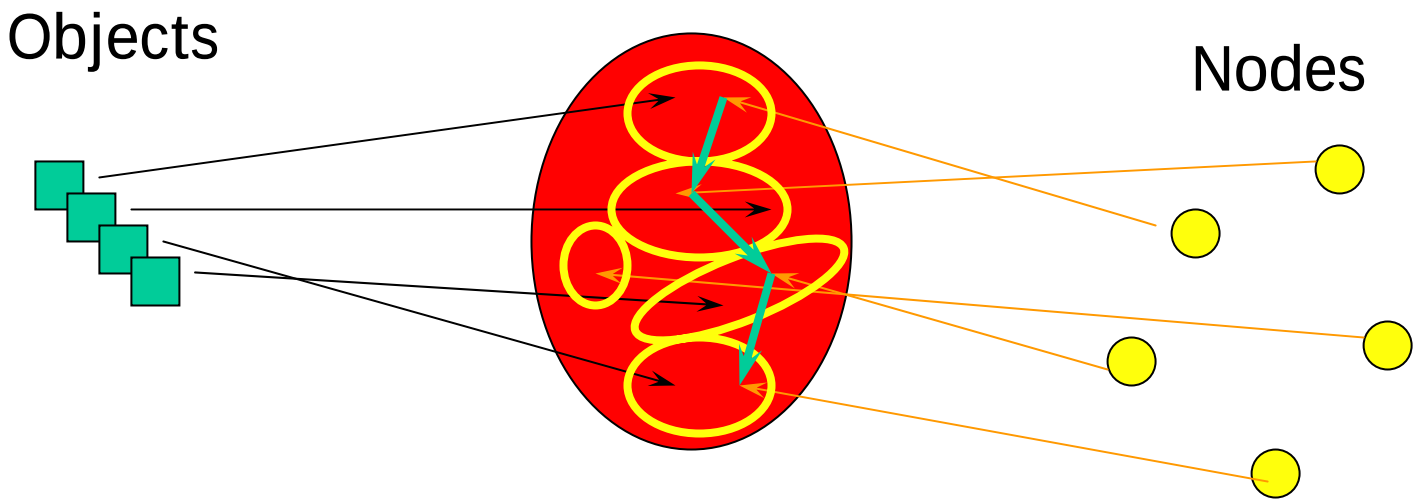
1. Map both objects and nodes into some topology (“id space”)

Distributed Hashing — General Approach



1. Map both objects and nodes into some topology (“id space”)
2. Each node “owns” some neighborhood in the topology, has channel to some neighbors

Distributed Hashing — General Approach

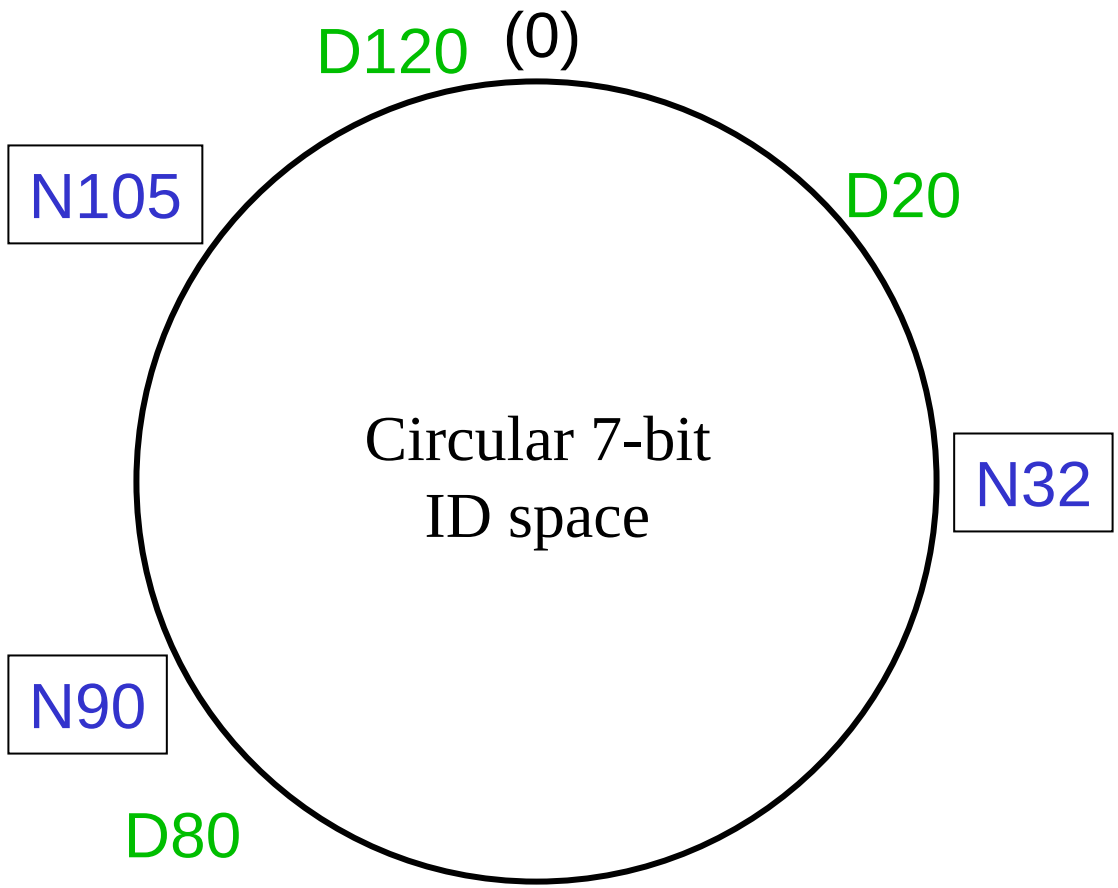


1. Map both objects and nodes into some topology (“id space”)
2. Each node “owns” some neighborhood in the topology, has channel to some neighbors
3. Topological structure lets query be routed to the “owner” of a given point

Chord Architecture

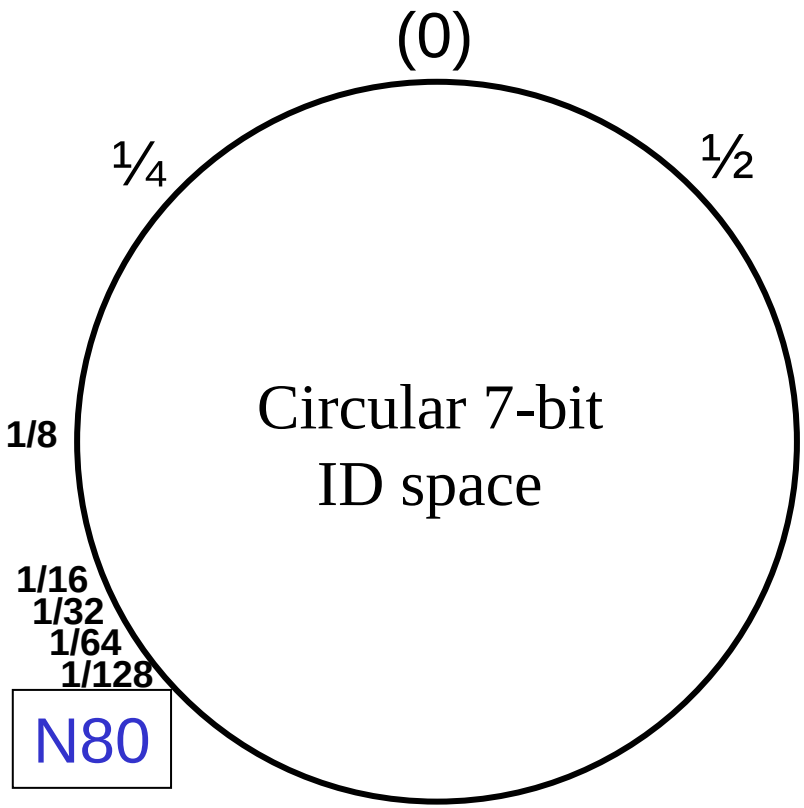
- Interface:
  - lookup(DocumentID) → NodeID, IP-Address
- Chord consists of
  - Consistent Hashing
  - Small routing (finger) tables: log(n)
  - Fast join/leave protocol

Consistent Hashing



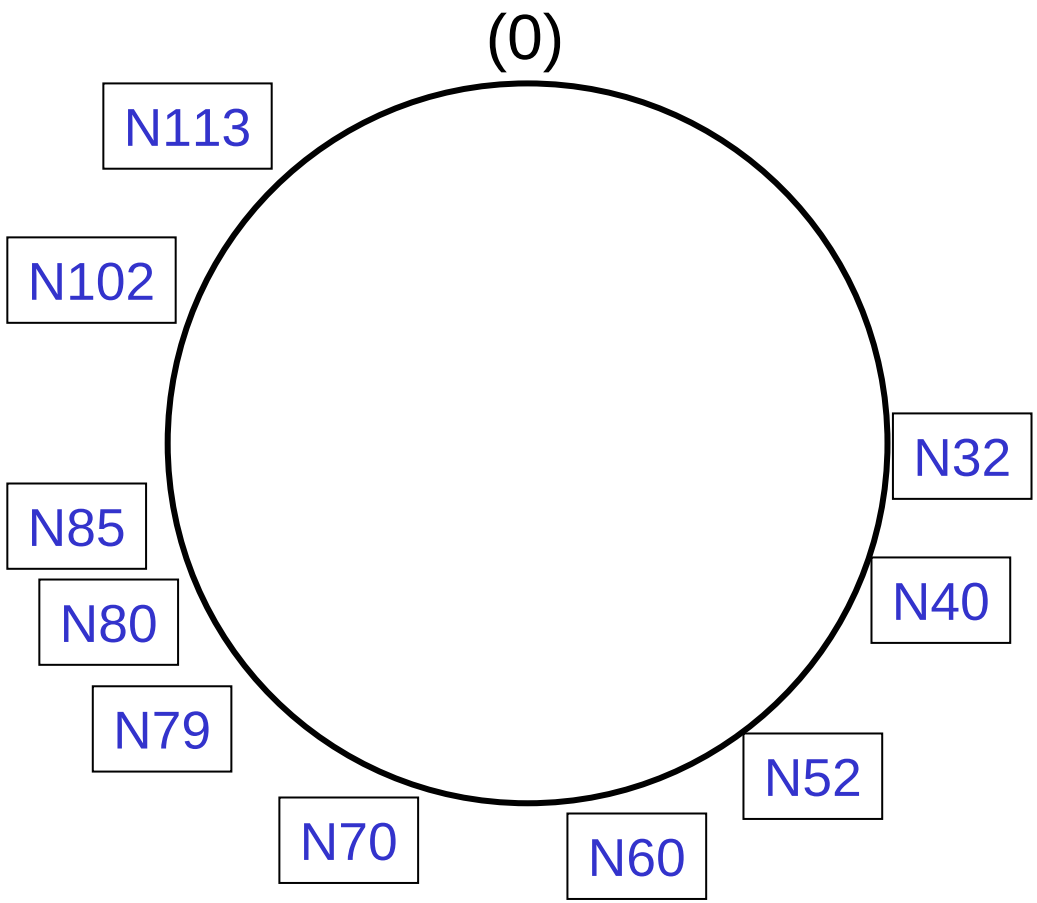
Example: Node 90 is the “successor” of document 80.

Chord Uses log(N) "Fingers"



N80 knows of only seven other nodes.

Chord Finger Table

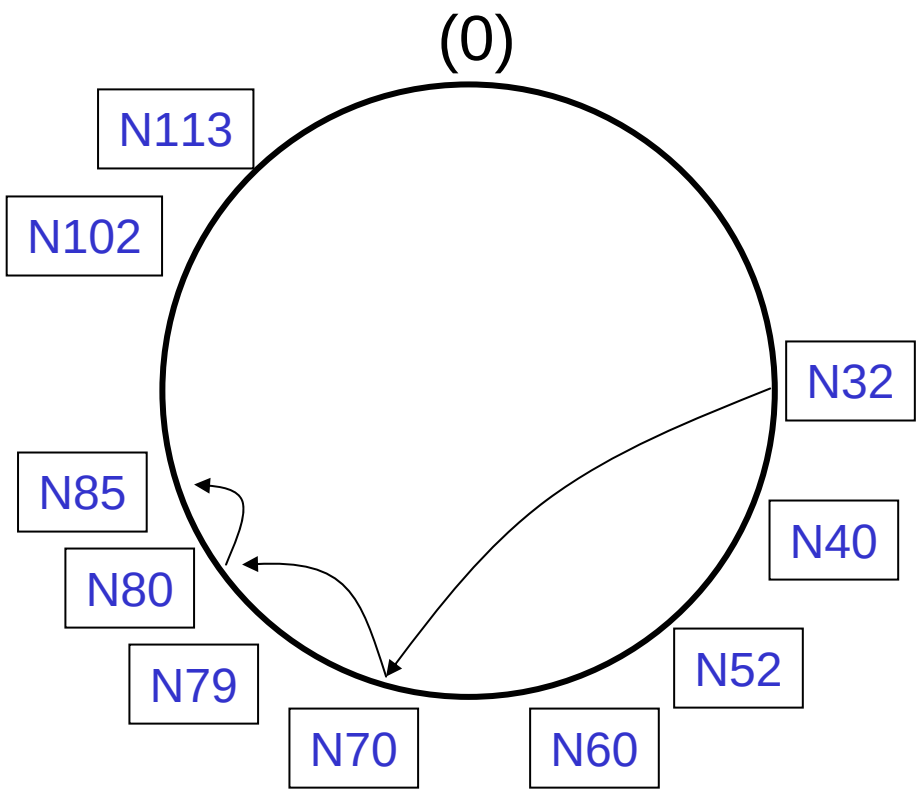


N32's Finger Table

33..33	N40
34..35	N40
36..39	N40
40..47	N40
48..63	N52
64..95	N70
96..31	N102

Node n's i-th entry: first node $\geq n + 2^{i-1}$

Chord Lookup



N32's Finger Table

33..33	N40
34..35	N40
36..39	N40
40..47	N40
48..63	N52
64..95	N70
96..31	N102

N70's Finger Table

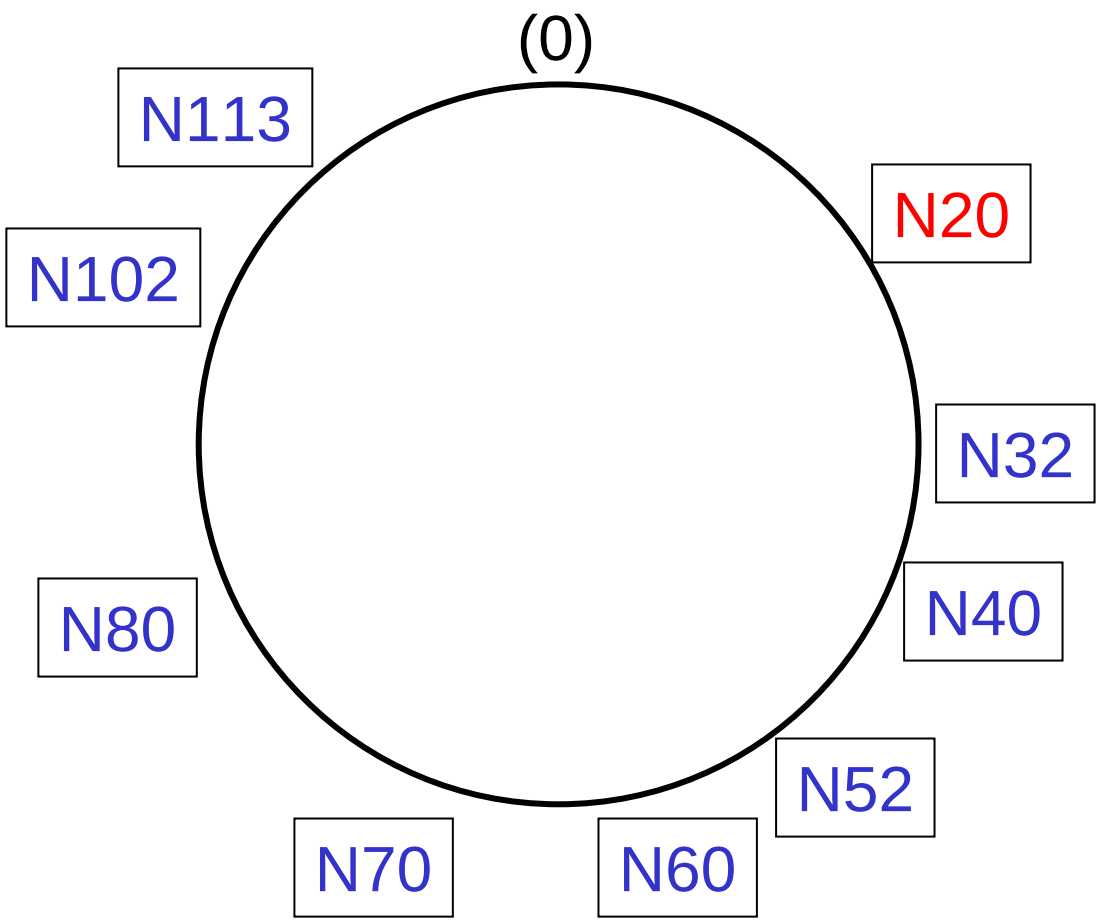
71..71	N79
72..73	N79
74..77	N79
78..85	N80
86..101	N102
102..5	N102
6..69	N32

N80's Finger Table

81..81	N85
82..83	N85
84..87	N85
88..95	N102
96..111	N102
112..15	N113
16..79	N32

Node 32, lookup(82): 32 → 70 → 80 → 85.

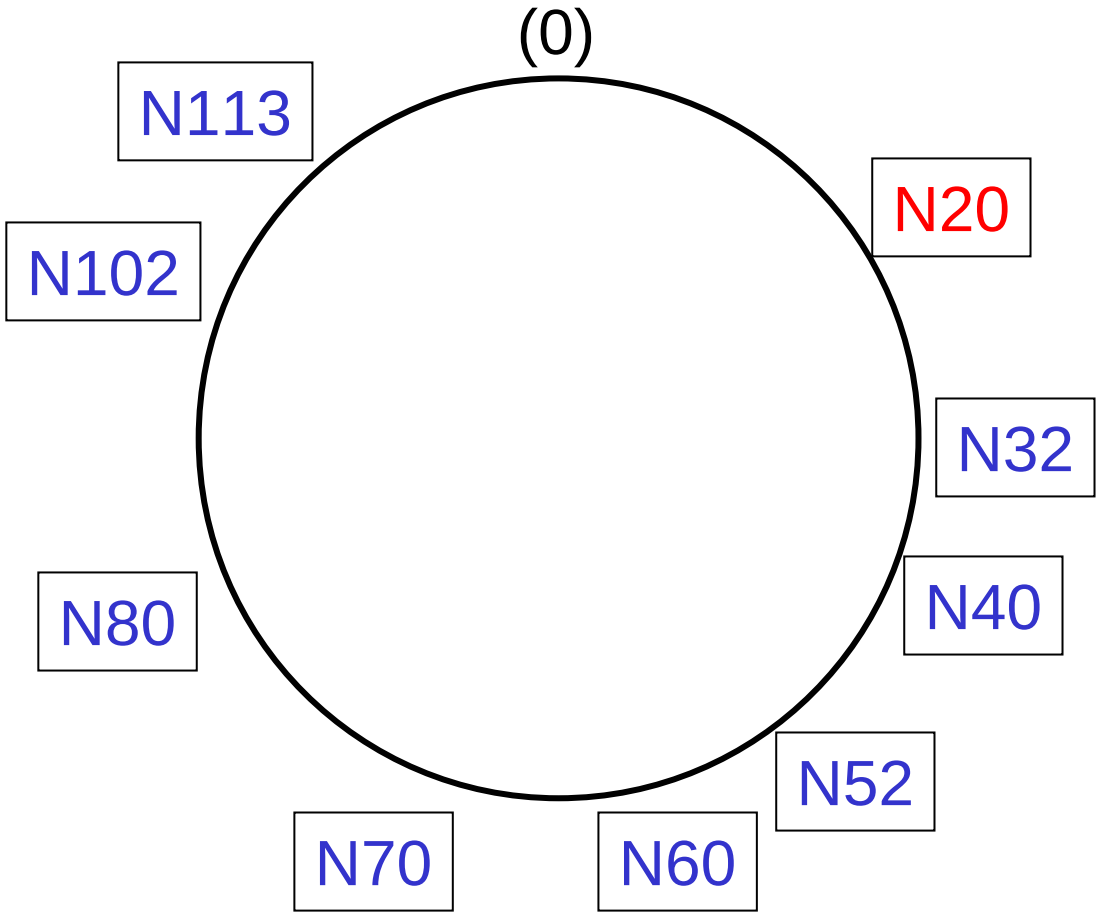
New Node Join Procedure



N20's Finger Table

21..21
22..23
24..27
28..35
36..51
52..83
84..19

New Node Join Procedure (2)

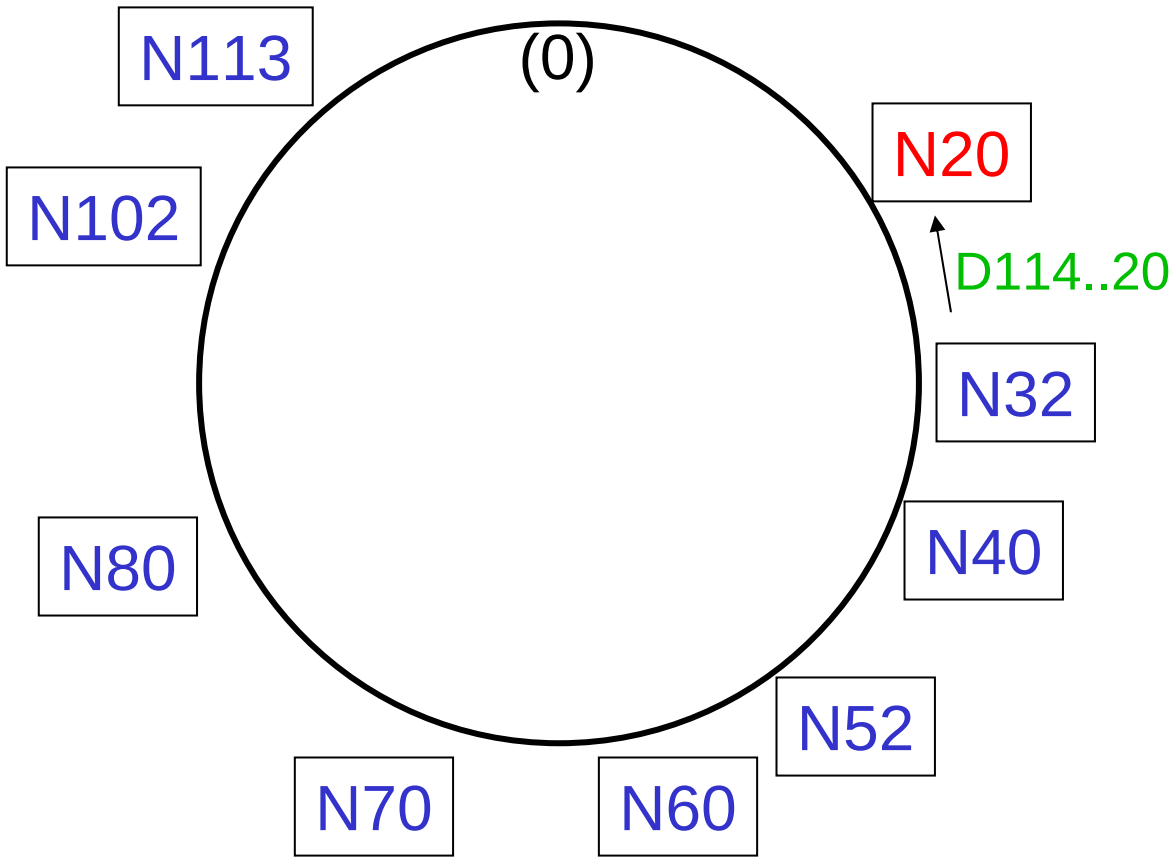


N20's Finger Table

21..21	N32
22..23	N32
24..27	N32
28..35	N32
36..51	N40
52..83	N52
84..19	N102

Node 20 asks *any* node for successor to 21, 22, …, 52, 84.

New Node Join Procedure (3)



N20's Finger Table

21..21	N32
22..23	N32
24..27	N32
28..35	N32
36..51	N40
52..83	N52
84..19	N102

Node 20 moves documents from node 32.

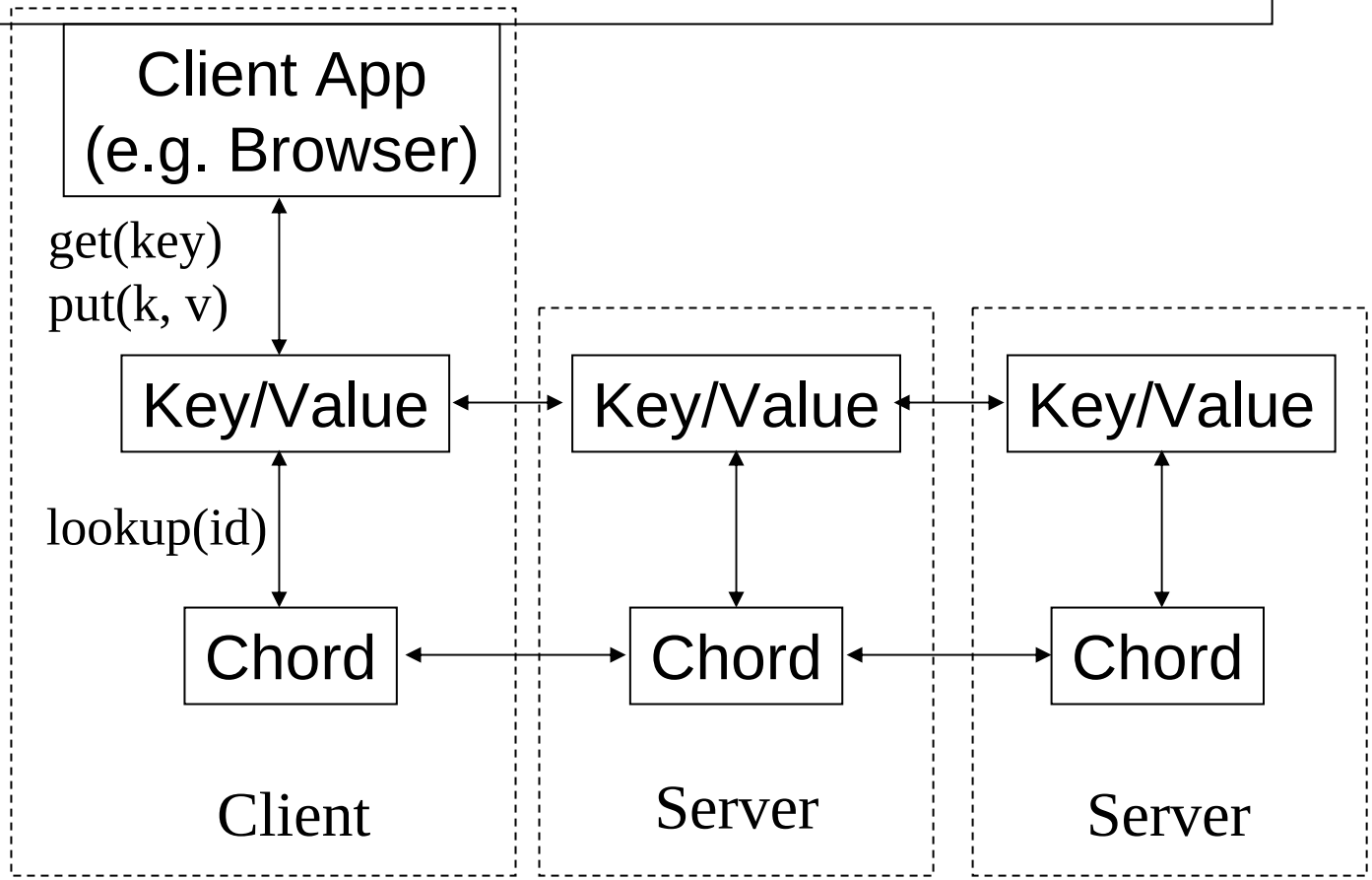
Chord Properties

- Log(n) lookup messages and table space.
- Well-defined location for each ID.
  - No search required.
- Natural load balance.
- No name structure imposed.
- Minimal join/leave disruption.
- Does not store documents…

Building Systems with Chord

Data auth.

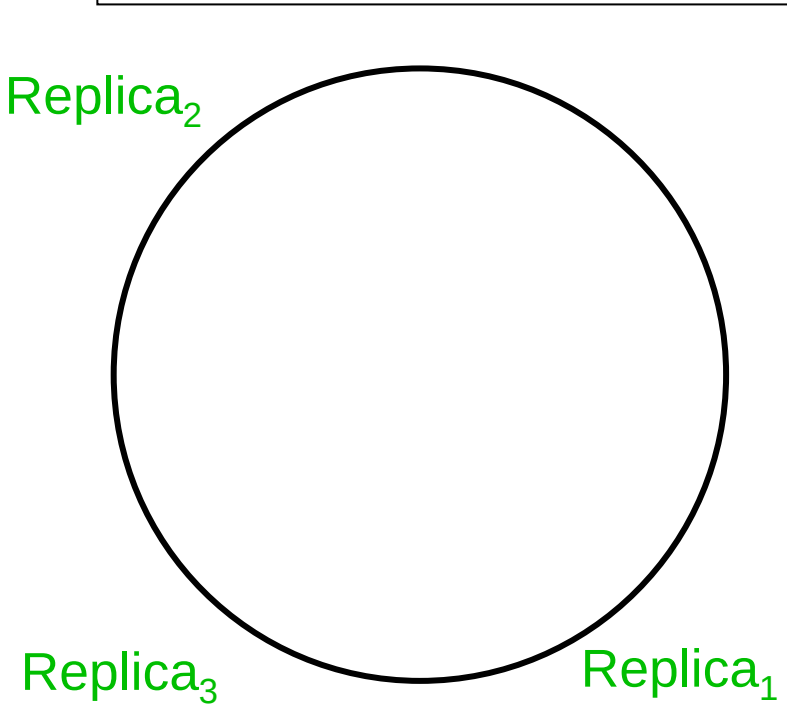
Storage
Update auth.
Fault tolerance
Load balance

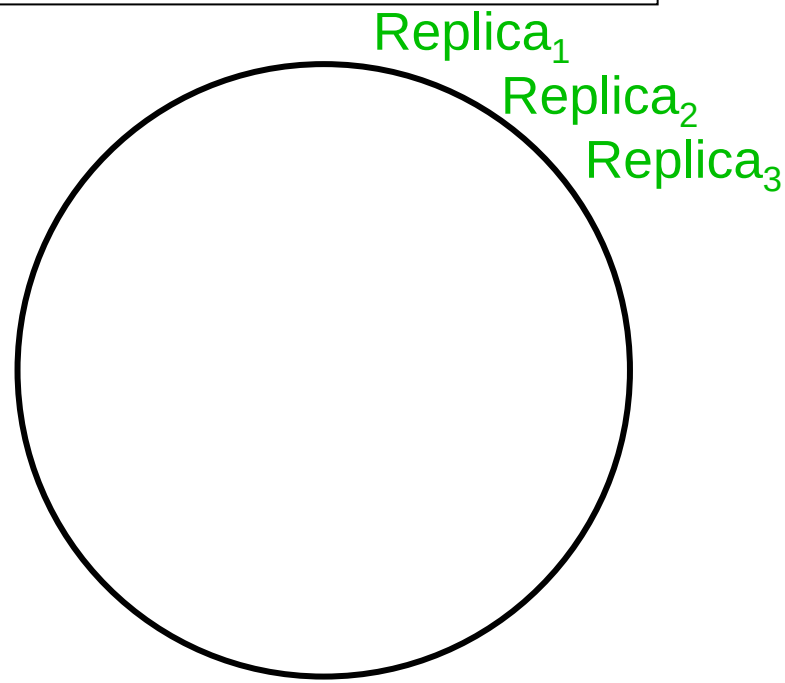


Naming and Authentication

1. Name could be hash of file content
   - Easy for client to verify
   - But update requires new file name
2. Name could be a public key
   - Document contains digital signature
   - Allows verified updates w/ same name

Naming and Fault Tolerance

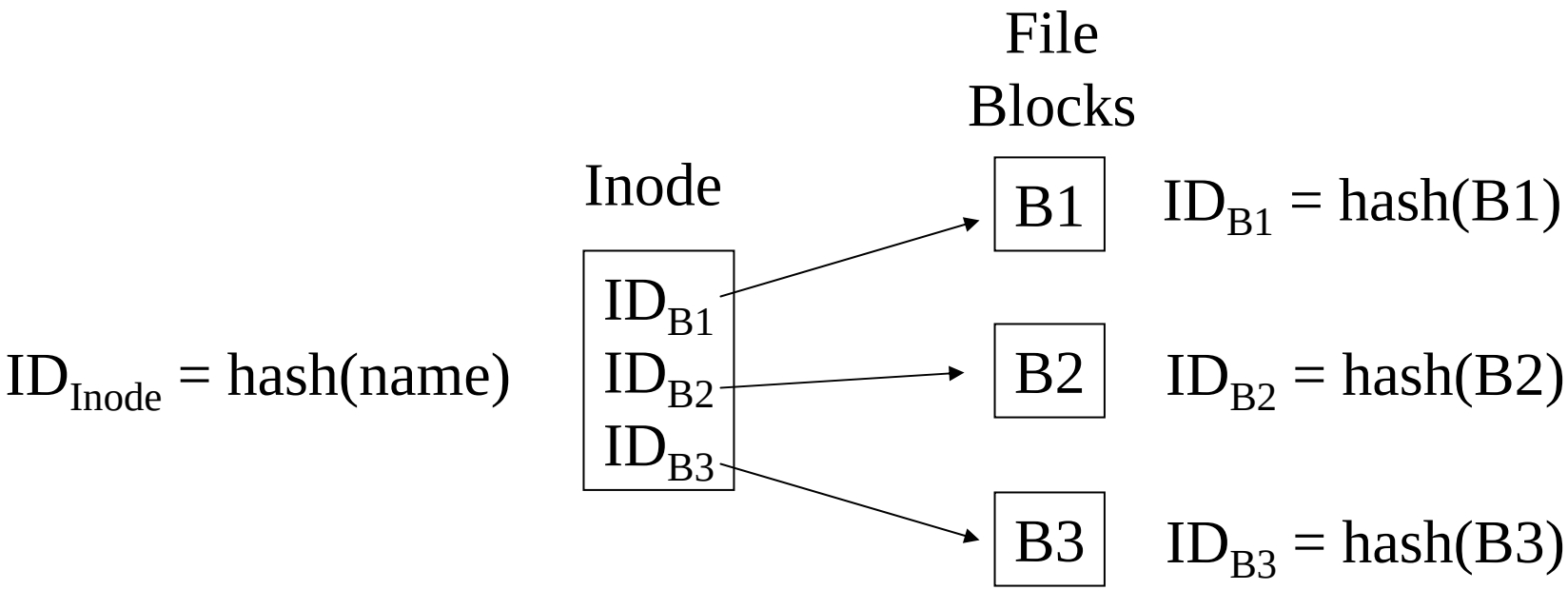




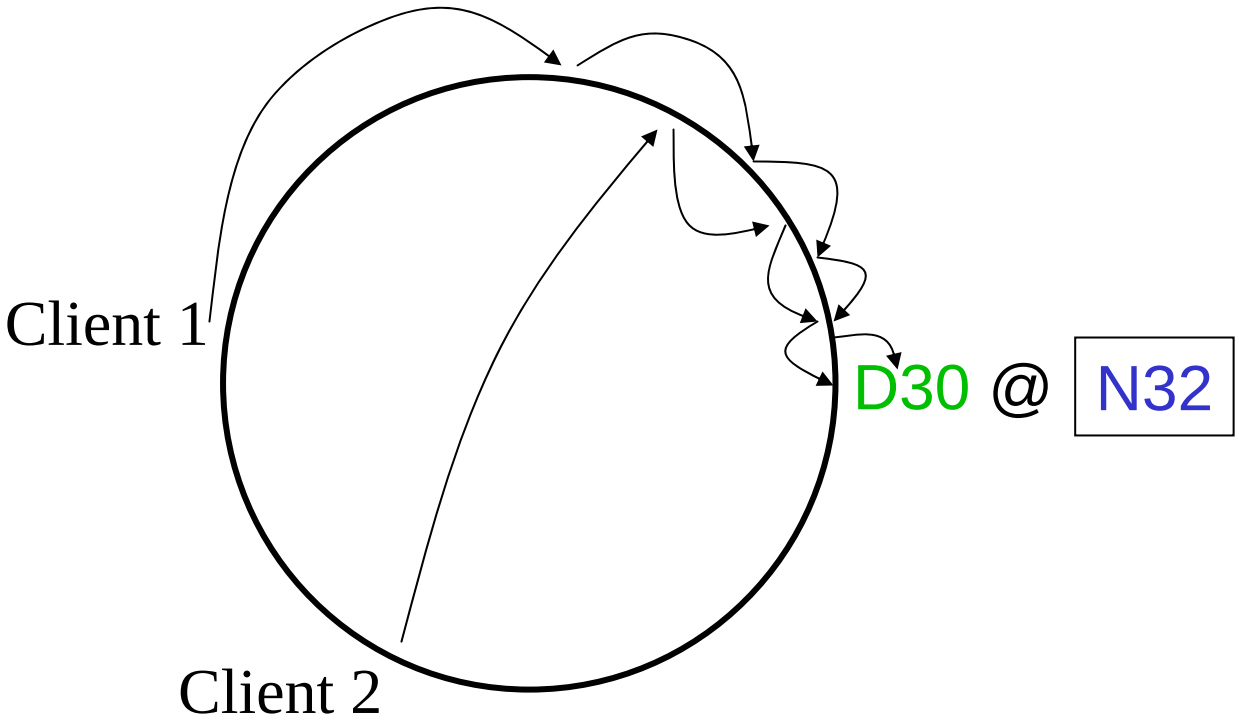
$ID_i = hash(name + i)$

$ID_i = successor^i(hash(name))$

Naming and Load Balance



Naming and Caching



Overview

Chord:

- **Maps keys onto nodes** in a 1D circular space
- Uses consistent hashing –D.Karger, E.Lehman
- Aimed at large-scale peer-to-peer applications

Talk

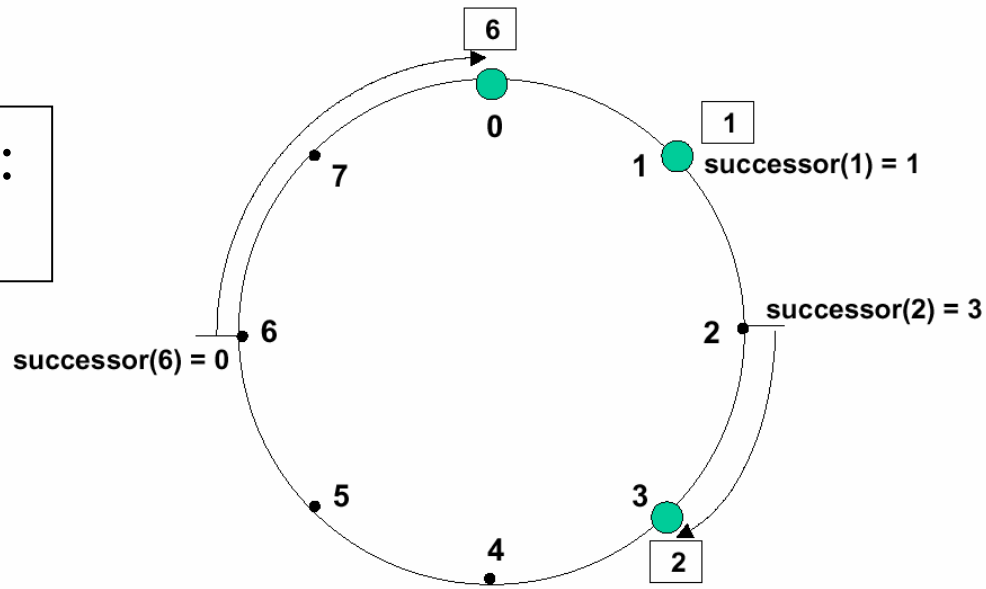
- Consistent hashing
- Algorithm for key location
- Algorithm for node joining
- Algorithm for stabilization
- Failures and replication

Consistent hashing

- Distributed caches to relieve hotspots on the web
- Node identifier hash = hash(IP address)
- Key identifier hash = hash(key)
- Designed to let nodes enter and leave the network with minimal disruption

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

In Chord hash function is Secure Hash SHA-1



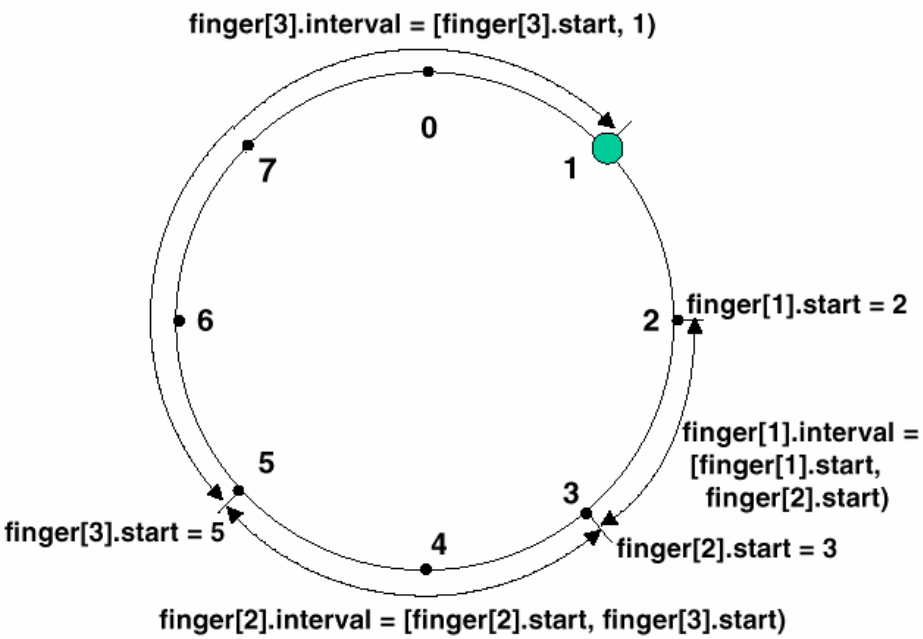
Key Location

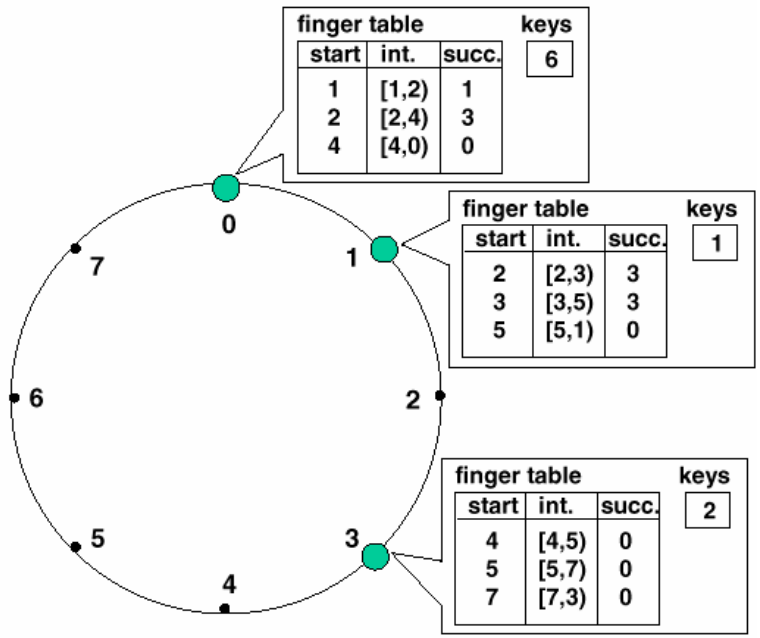
- Finger tables allow faster location by providing additional routing information than simply successor node

Notation	Definition
finger[k].start	$(n+2^{k-1}) \bmod 2^m$, 1=<k=<m
.interval	[finger[k].start,finger[k+1].start)
.node	first node>=n.finger[k].start
successor	the next node on the identifier circle; finger[1].node
predecessor	the previous node on the identifier circle

k is the finger table index

Lookup(id)





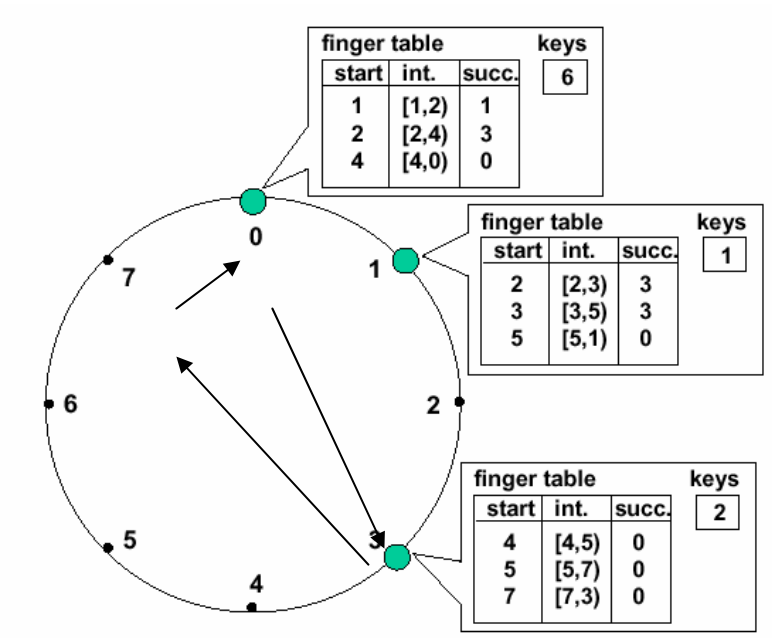
Finger table for Node 1

Finger tables and key locations with nodes 0,1,3 and keys 1,2 and 6

Lookup PseudoCode

To find the successor of an id :

Chord returns the successor of the closest preceding finger to this id.




```
// ask node n to find id's successor
n.find_successor(id)
   n' = find_predecessor(id);
   return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
   n' = n;
   while (id ∉ (n', n'.successor])
      n' = n'.closest_preceding_finger(id);
   return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
   for i = m downto 1
      if (finger[i].node ∈ (n, id))
         return finger[i].node;
   return n;
```

Finding successor of identifier 1

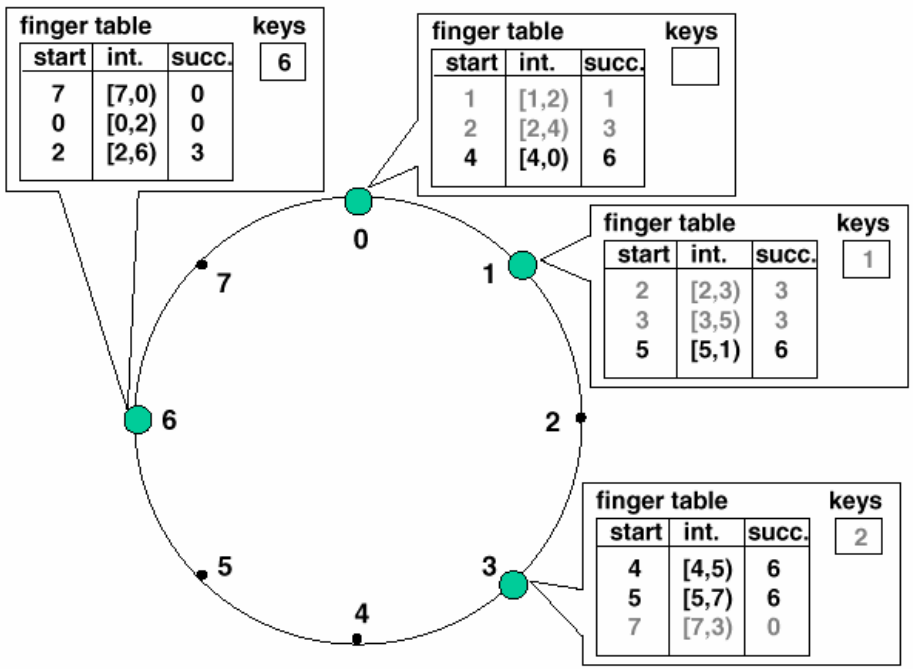
Lookup cost

- The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in *find_predecessor* to halve the distance to the target identifier.

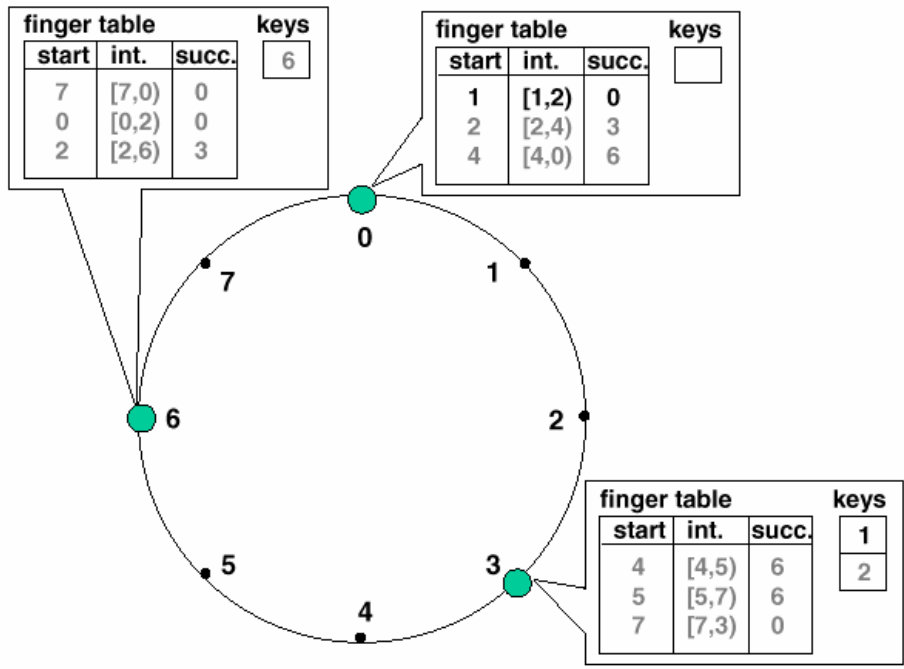
  In an N node Network the number of messages is of

  O(Log N)

Node Join/Leave



Finger Tables and key locations after Node 6 joins



After Node 3 leaves

Changed values are in black, unchanged in gray

Join PseudoCode

Three steps:

1- **Initialize** finger and predecessor of new node n

2- **Update** finger and predecessor of existing nodes to reflect the addition of n

n becomes i^{th} finger of node p if:

- p precedes n by at least 2^{i-1}
- i^{th} finger of node p succeeds n

3- **Transfer** state associated with keys that node n is now responsible for

New node n only needs to contact node that immediately forwards it to transfer responsibility for all relevant keys

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);

// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Join/leave cost

Number of nodes that need to be updated when a node joins is

$O(Log\ N)$

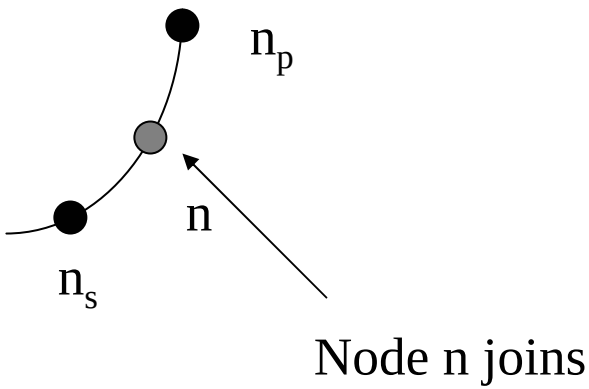
Finding and updating those nodes takes

$O(Log^2\ N)$

Stabilization

- If nodes join and stabilization not completed 3 cases are possible
  - finger tables are current → lookup successful
  - successors valid, fingers not → lookup successful (because find_successor succeeds) but slower
  - successors are invalid or data hasn't migrated → lookup fails

Stabilization cont'd



n acquires n_s as successor

n_p runs stabilize:

- asks n_s for its predecessor (n)
- n_p acquires n as its successor
- n_p notifies n which acquires n_p as predecessor

Predecessors and successors are correct

Failures and replication

- Key step in failure recovery is correct successor pointers
- Each node maintains a successor-list of *r* nearest successors
- Knowing *r* allows Chord to inform the higher layer software when successors come and go → when it should propagate new replicas

Algorithms

find_successor()

return find_predecessor().successor

find_predecessor()

while(id not in [n,n.successor])

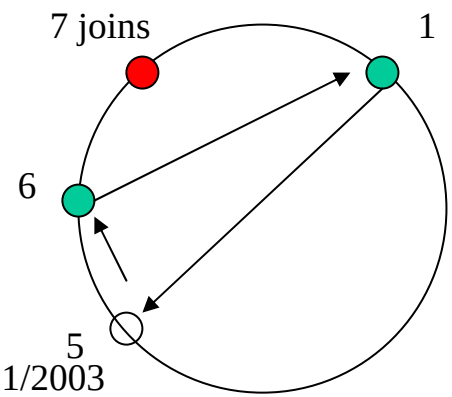
n=n.closest_preceding_finger()

closest_preceding_finger()

from last level in FT to first:

if(succAtLevel in (n,id))

return succAtLevel



join()

init_finger_tables()

update_others()

init_finger_tables()

successor=node.find_successor()

predecessor=successor.predecessor

predecessor.successor=new

everything btw new and successor gets assigned this successor as succ

update_others()

for each level in FT

l=find_predecessor()

l.update_finger_table

update_finger_table()

if(new in [this, successor])

p=getPredecessor

if(p!new) p.update_finger_table

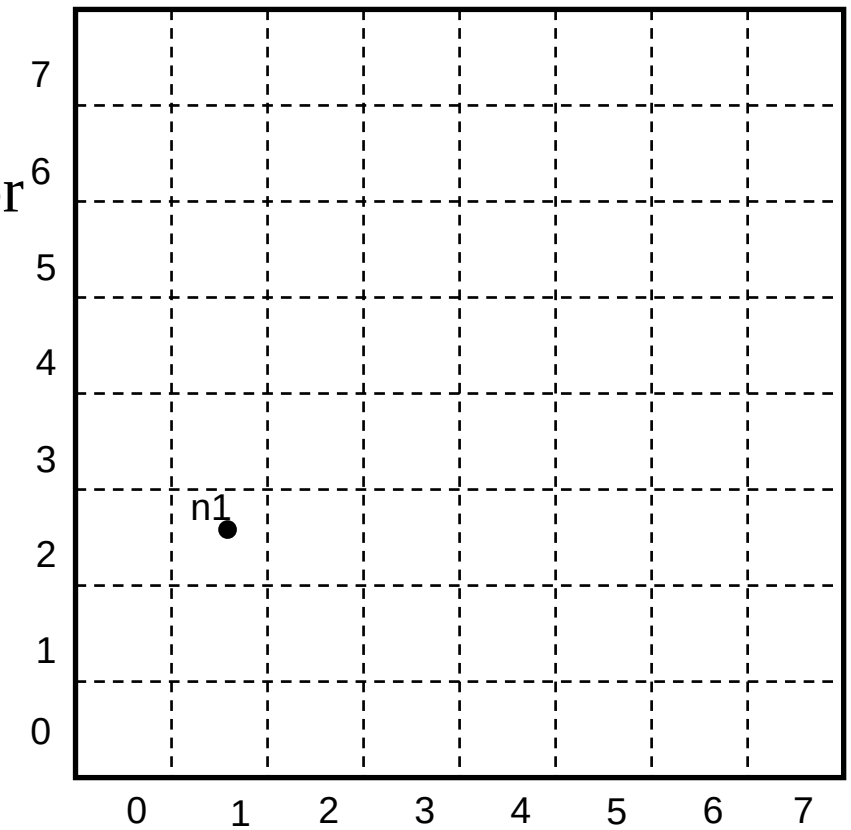
Document Routing – CAN

- Associate to each node and item a unique *id* in an *d*-dimensional space
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(*d*)
  - Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes

Slide modified from another presentation

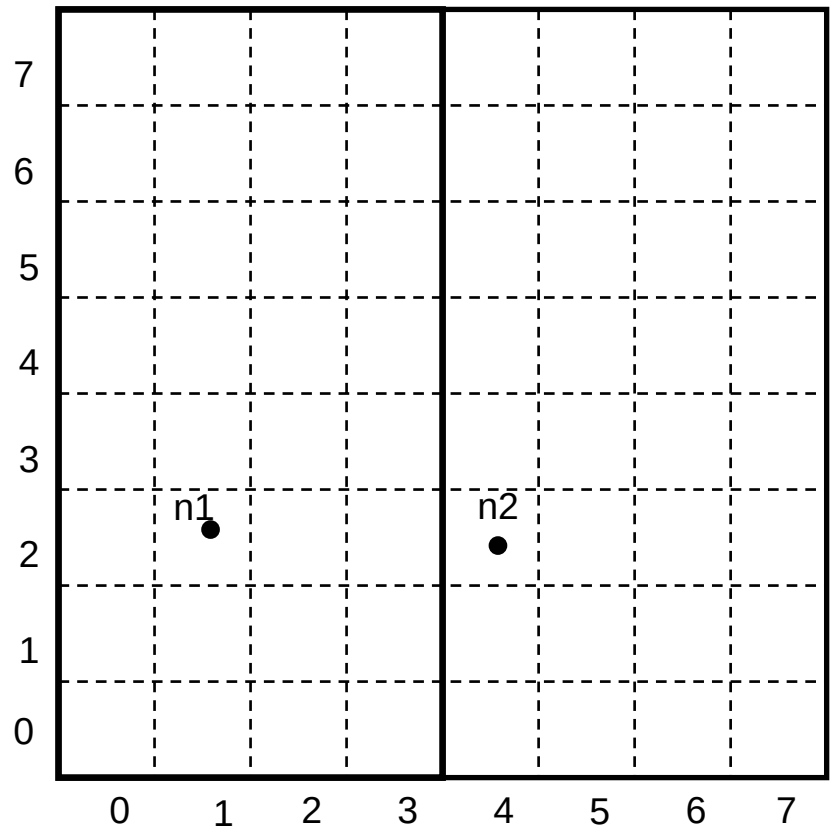
CAN Example: Two Dimensional Space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1:(1, 2) first node that joins → cover the entire space



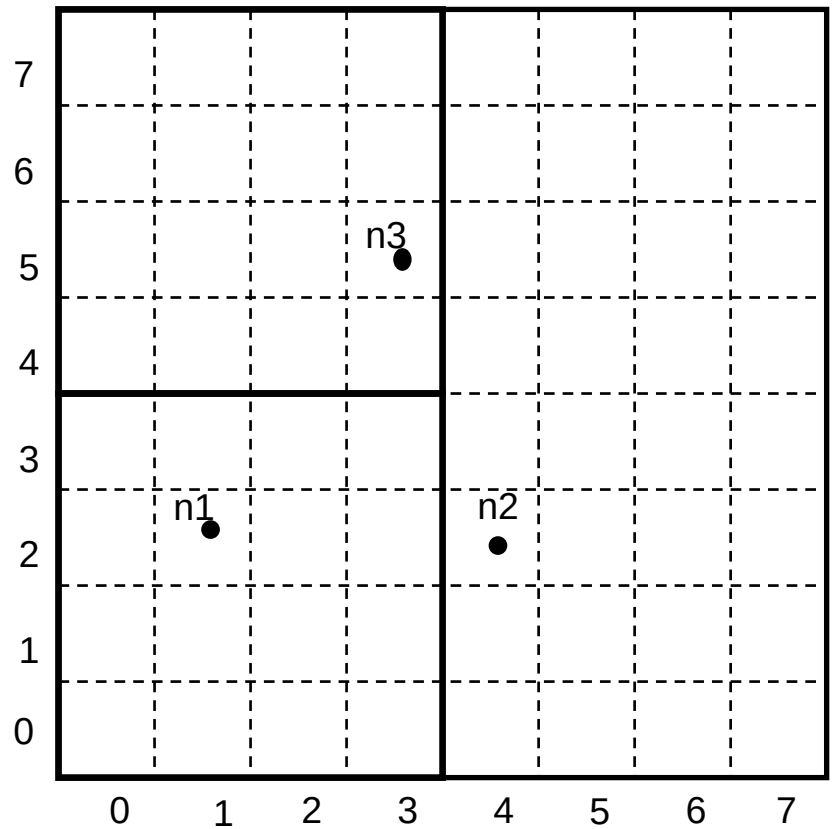
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



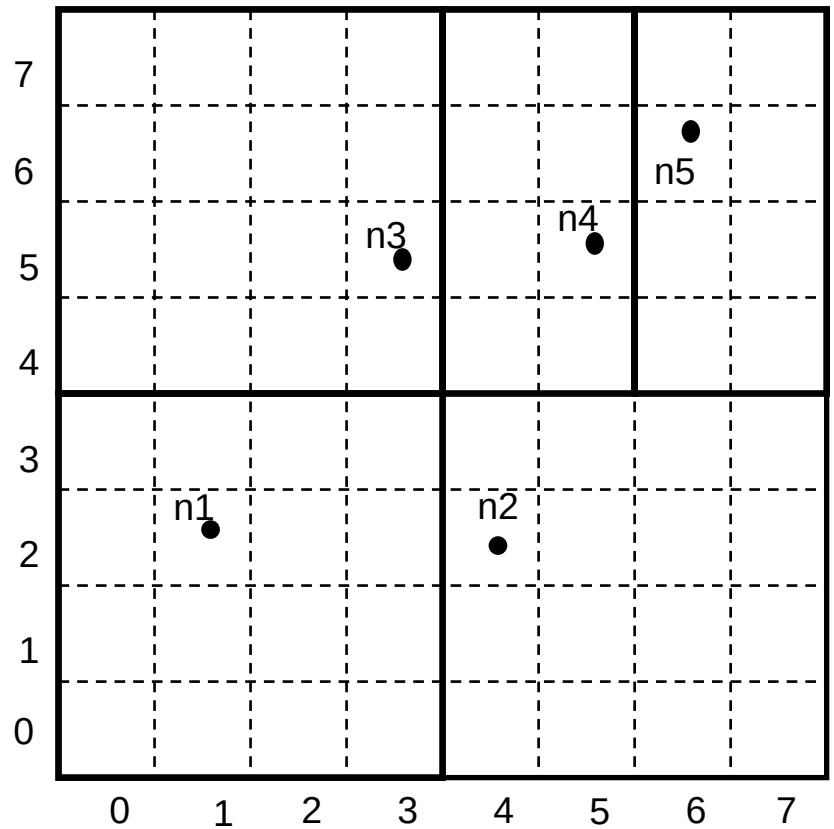
CAN Example: Two Dimensional Space

- Node n3:(3, 5) joins → space is divided between n1 and n3



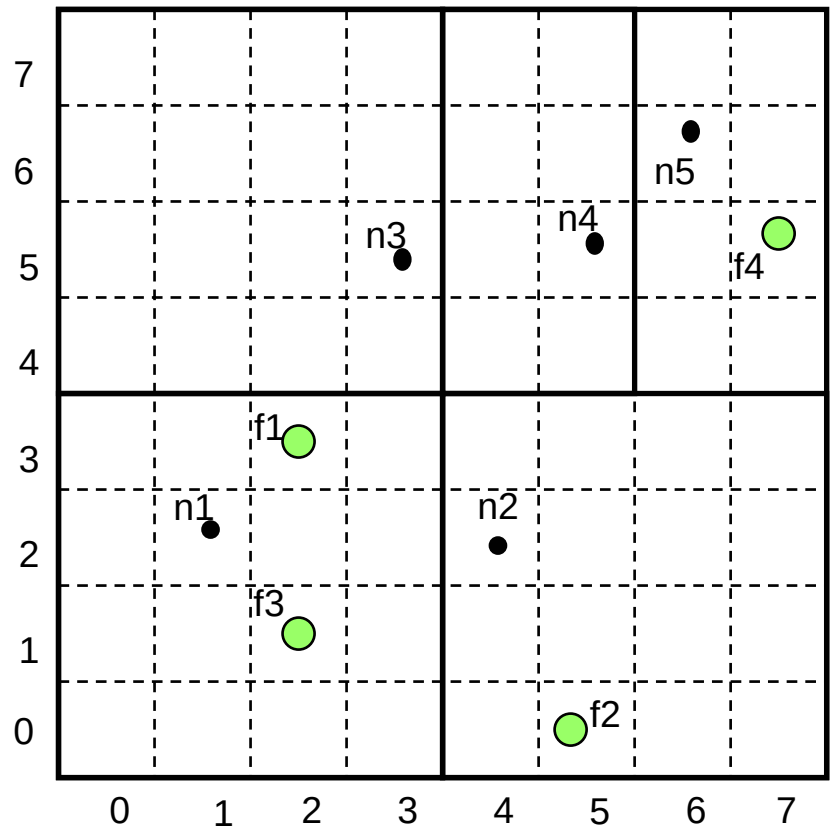
CAN Example: Two Dimensional Space

- Nodes n4:(5, 5) and n5:(6,6) join



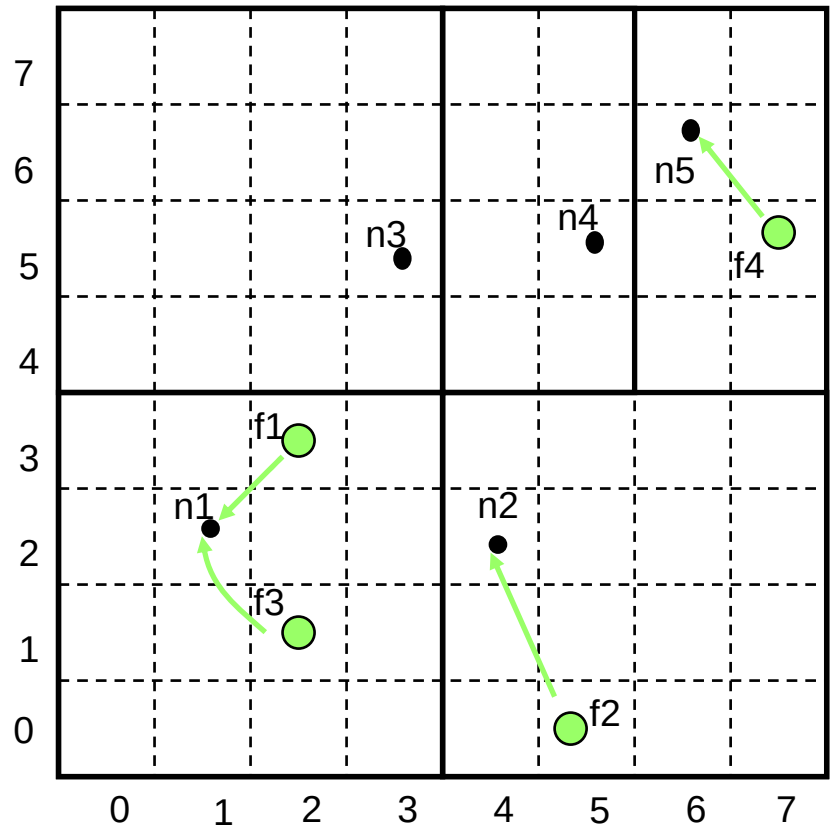
CAN Example: Two Dimensional Space

- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



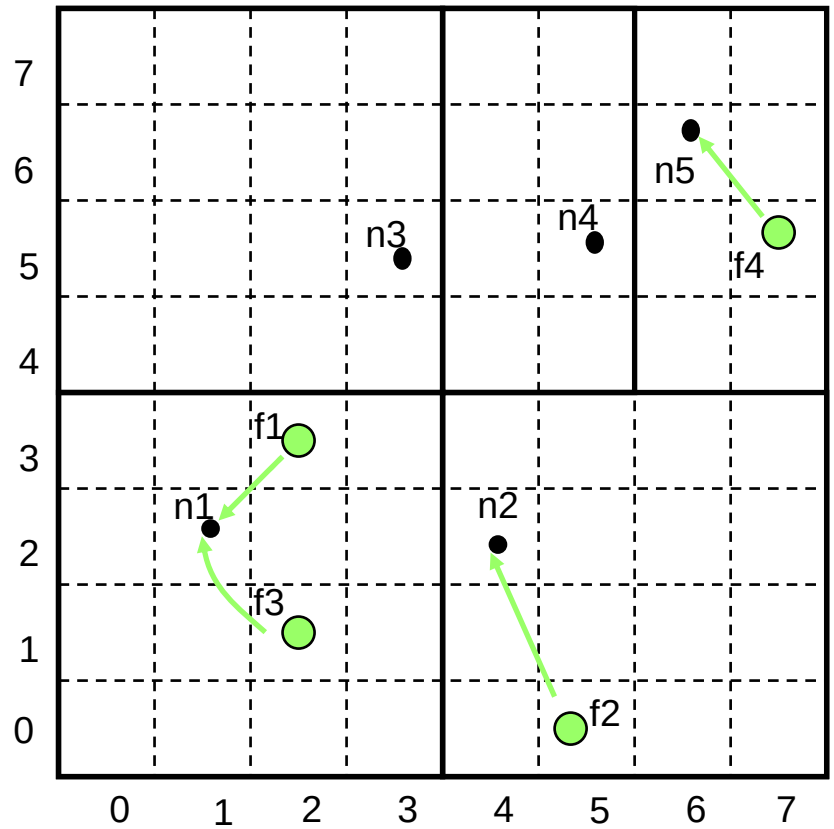
CAN Example: Two Dimensional Space

- Each item is stored by the node who owns its mapping in the space



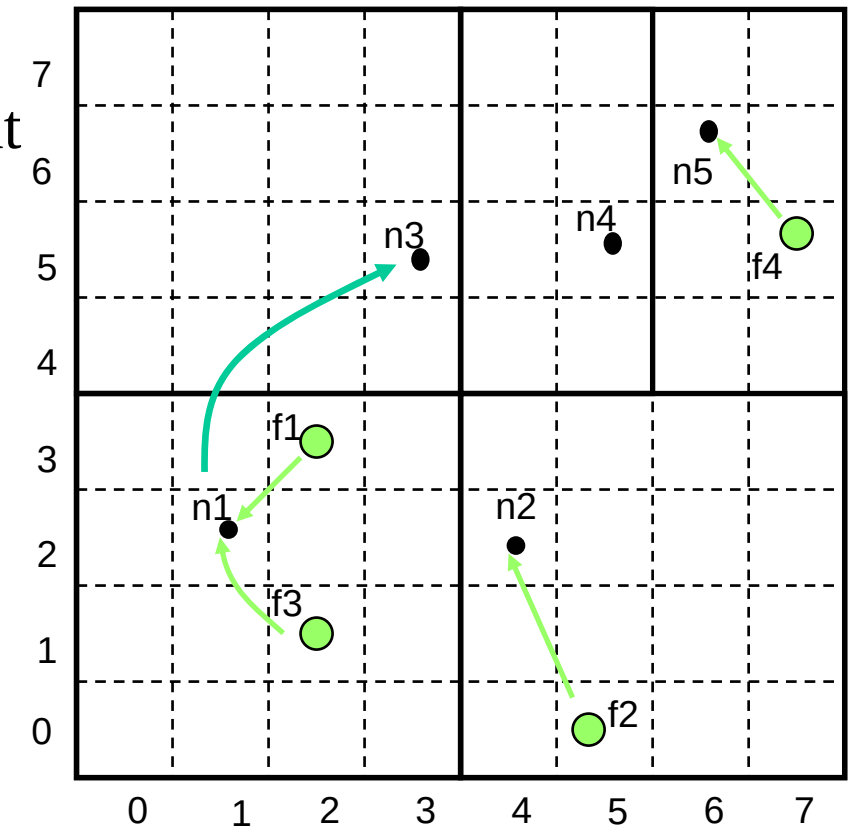
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



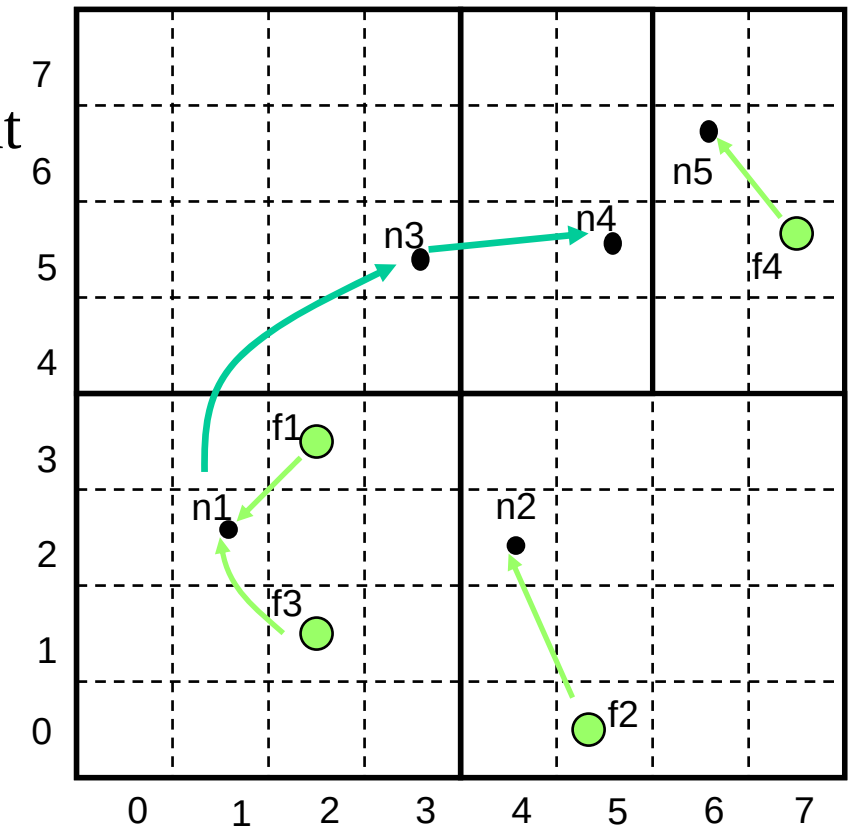
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



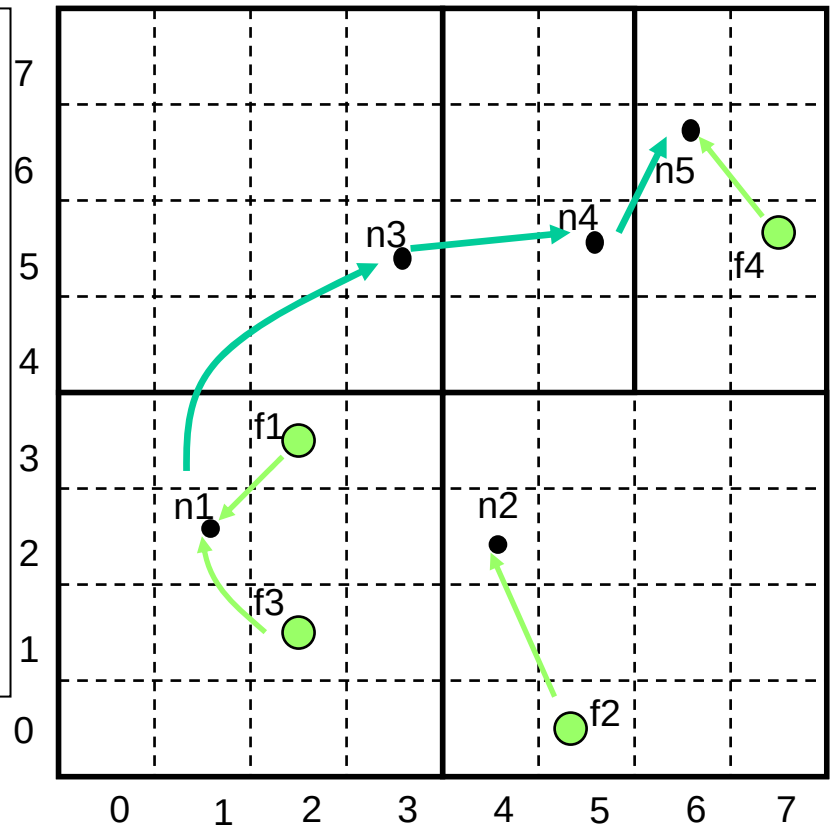
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding

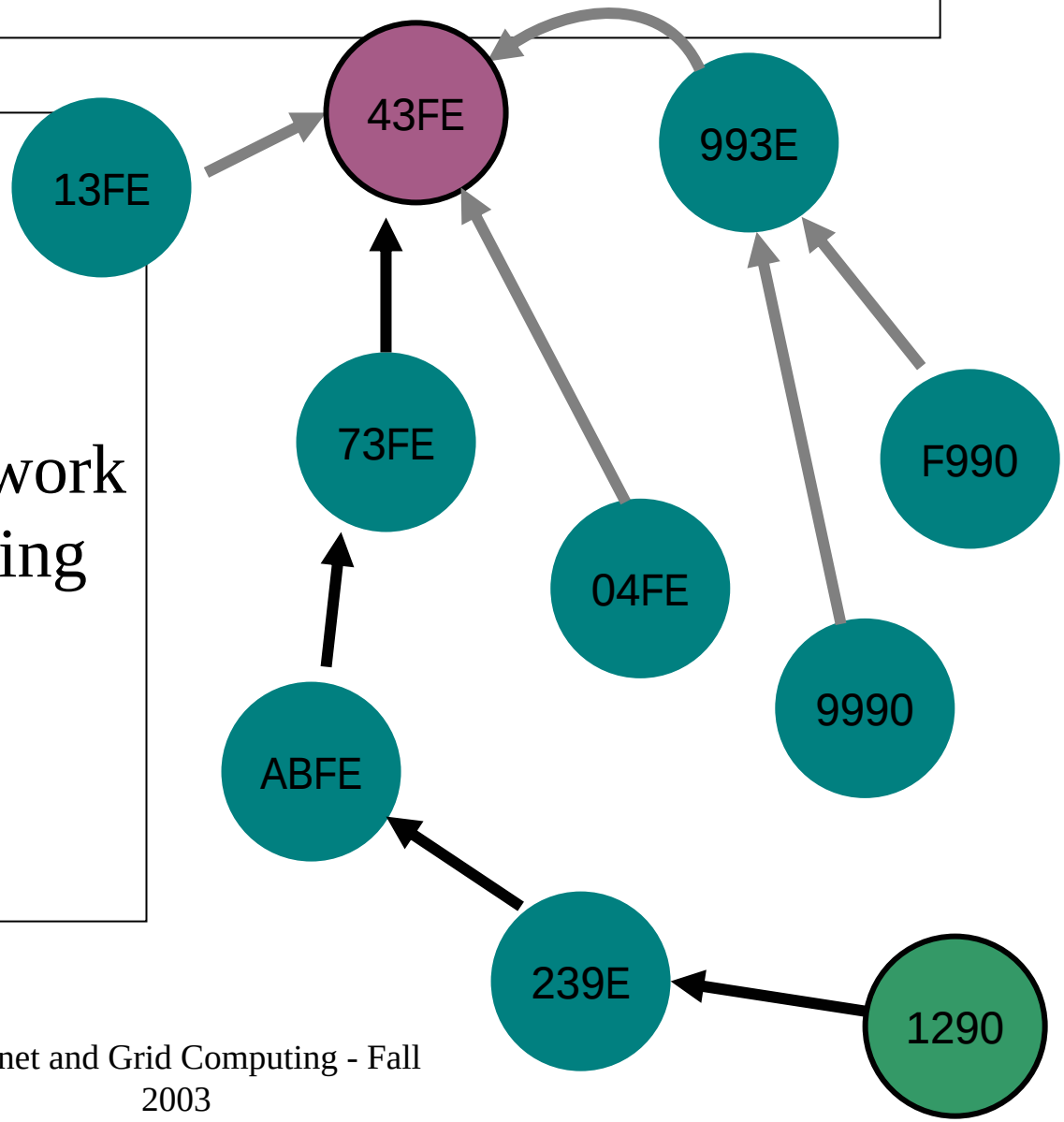


Node Failure Recovery

- Simple failures
  - know your neighbor's neighbors
  - when a node fails, one of its neighbors takes over its zone
- More complex failure modes
  - simultaneous failure of multiple adjacent nodes
  - scoped flooding to discover neighbors
  - hopefully, a rare event

Doc Routing – Tapestry/Pastry

- Global mesh
- Suffix-based routing
- Uses underlying network distance in constructing mesh



CAN: node failures

- Simple failures
  - know your neighbor's neighbors
  - when a node fails, one of its neighbors takes over its zone
- More complex failure modes
  - simultaneous failure of multiple adjacent nodes
  - scoped flooding to discover neighbors
  - hopefully, a rare event
- Background zone reassignment algorithm

CAN: scalability

- For a uniformly partitioned space with **n** nodes and **d** dimensions
  - per node, number of neighbors is 2d
  - average routing path is $(dn^{1/d})/4$ hops
    - A hop here is an application-level hop
    - 1 app-level hop = (possibly) multiple IP-level hops
  - simulations show that the above bounds hold for imperfectly partitioned spaces

Can scale the network without increasing per-node state

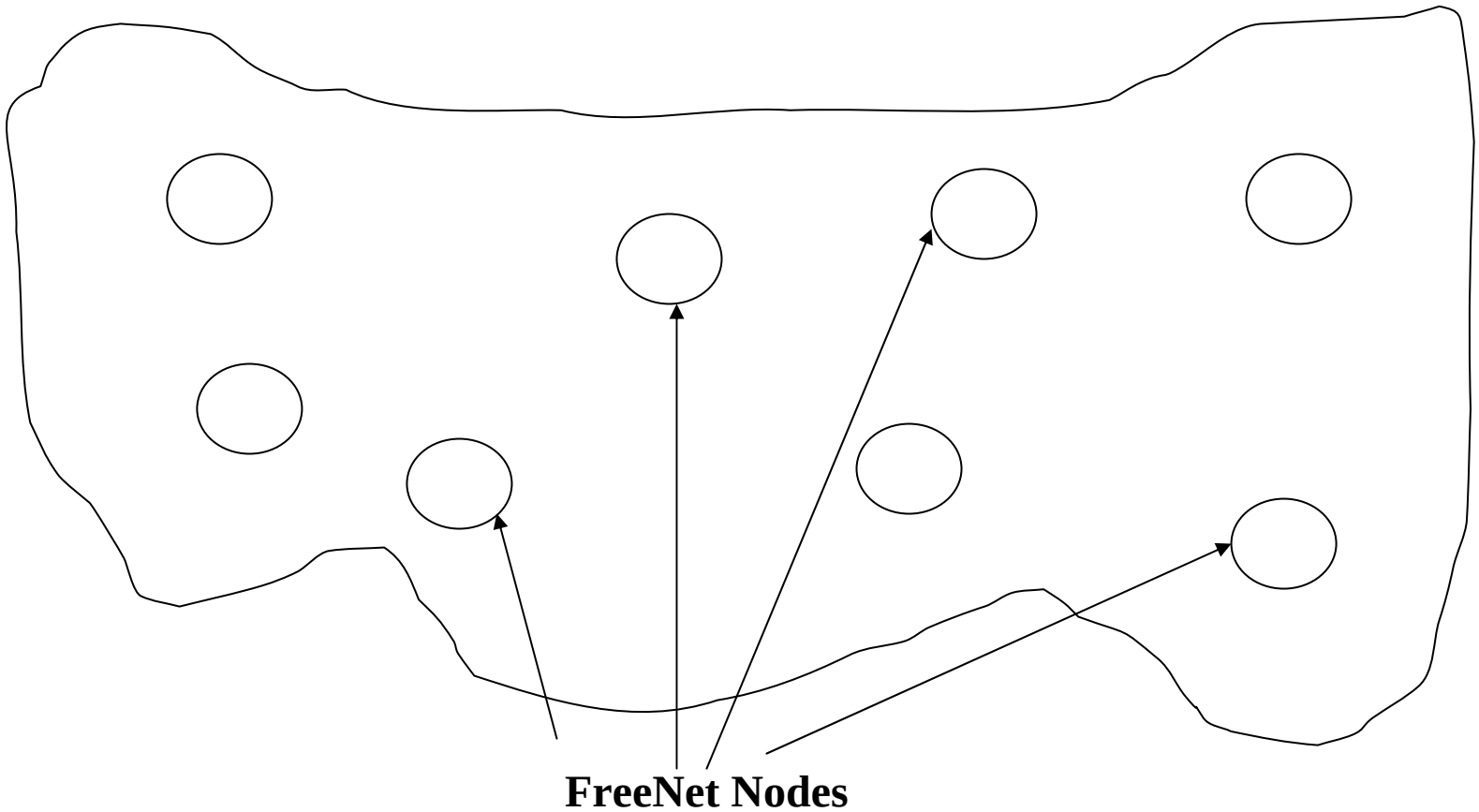
Comparing Guarantees

	Model	**Search**	**State**
Chord	Uni-dimensional	log N	log N
CAN	Multi-dimensional	$dN^{1/d}$	2d
Tapestry	Global Mesh	$\log_b N$	$b \log_b N$
Pastry	Neighbor map	$\log_b N$	$b \log_b N + b$

Summary

- Two similar approaches to locating objects by “computed routing”
  - Similar to Manhattan Street Networks
- Both are scalabe, reasonably robust
- All these P2P networks ignore underlying topology!

FreeNet - Serverless, Symmetric, Secure, Parallel Internet File System



Introduction to P2P Systems

Content Summary

- **Conceptual Elements**
- **File Insertion**
- **File Retrieval**
- **File Update**
- **Large File Management**
- **Node Join**
- **Communication Protocols**

Conceptual Elements of FreeNet

- **Network Nodes**
  - **Storage**
  - **Files**
  - **Routing Tables**
- **Files**
  - **Byte String**
  - **Path Name**
  - **Search Key**
    - **Keyword Signed Key**
    - **Content Key**
  - **File Signature**
- **Directories - Personal Name Spaces**
  - **Signed SubSpace Key**
- **Interactions/Transactions**
  - **Join Network**
  - **Insert File**
  - **Request File**
- **Storage Management Algorithm**
  - **File Storage, Retrieval and Retention**
  - **Routing Tables**

Files

File Content = string of bytes

File Name/ File Description - Unix text string

/text/philosophy/sun-tzu/art-of-war

/peertopeer/books/oram

Keys associated with each file.

{Public Key, Private Key} = PPKP-Generator(File Descriptor)

File Keys

Keyword Signed Key = Secure Hash Algorithm(Public Key)

Content Hash Key = SHA(File Content)

File Encryption Key - Use file descriptor as an encryption key.

File Signature = Private Key

Definitions:

1. Node Locality Set

*** Each node has a locality set of neighbors.**

2. Node Routing Table

*** A Node Routing Table is a set of Search Key - Node pairs.**

Node Resource Management

1. Each node allocates the amount of storage to be assigned to FreeNet Functions.

2. Storage is partitioned for file data storage and routing table storage.

3. Node data storage is managed as an LRU Cache. Files which are not accessed are eventually deleted from a node.

4. Routing table entries are also managed as an LRU cache but entries of the routing tables persist after eviction of file data.

Personal Name Spaces/Directories

File keys may be coupled to a unique identifier for a personal name space The personal name space is created as follows.

1. **Randomly generate a public/private key pair (The signed subspace key.)**

   {Public Key, Private Key} = PPKP(Random())

2. **Hash the public key**

   HPK = SHA(Public Key)

3. **Hash the file path name/file descriptor**

   HFPN = SHA(Path Name)

4. **File Key = XOR(HFPN, HPK)**

5. **File Signature = Private Key**

6. **Publish the file descriptor, the public key and the File Encryption Key.**

7. **This file is a "directory file" to which only the owner of the private key can add files.**

Name Space "Directory Structure"

Create a file using the signed-subspace key process with hypertext links to other files, perhaps signed-subspace key files.

The linked files may themselves contain links, etc.

Used to implement file update.

3. File Insertion Algorithm

a) Create a path name and a binary file key for the file.

b) The creating node sends a message to itself including a "hops to live" counter for this insertion.

c) The originating node then checks to see if that key is already in use.

d) If a match is found then the file associated with the previously defined key is returned as though a request had been made and the node knows this key has already been used. The node generates a new key for this file and again attempts the insertion process.

e) If no match is found then the originating node finds the node with the nearest key in its routing table and sends it an "insert" message with the key and the "hops to live" counter.

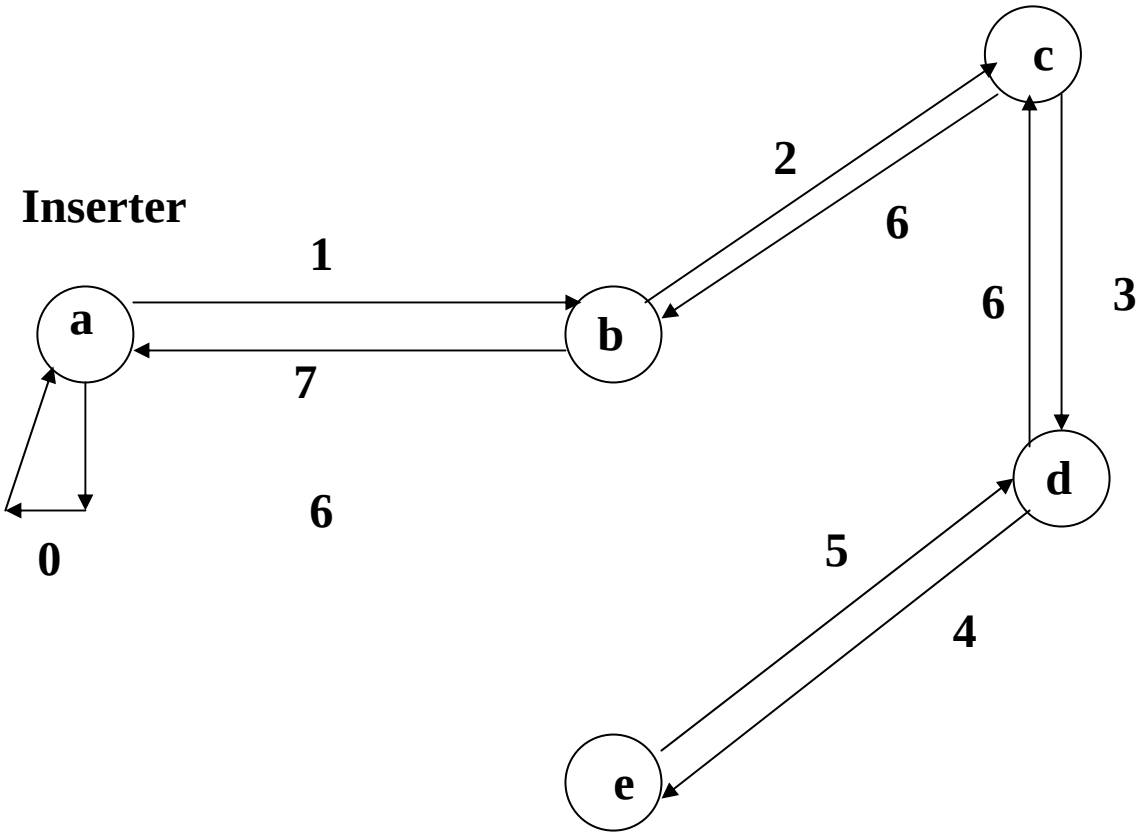
f) If this insert message causes a collision then the file is returned to the upstream inserter and again behave as though a request had been made. (Cache the file locally and create a routing table entry for the data source.

3. File Insertion Algorithm - Continued

g) If the "hops to live" limit is reached without a collision being detected then the insertion is successful and the "all clear" message is sent back to the originator of the file insertion.

h) The originator of the file insertion then sends the file itself which is propagated along the path of the key collision search. Each node will create a local copy of the file and establish a routing table entry matching the inserting node and the key. (A forwarding node may choose to arbitrarily change the supposed source when it forwards the insert message.)

i) The descriptive string (path name) is published by some out of band mechanism or in the case of name space encoded files, the descriptive string name and the subspace public key.



Successful insertion of file key in five nodes. File data will follow the same path.

Properties of Insertion Algorithm

1. Key-space locality

Files are cached on nodes with similar keys.

2. New nodes can use inserts to extend their network locality space.

3. Discourages fake file propagation. (A fake file is a file with “junk” or malicious content with a key identical to some file with actual data.)

The original files are propagated upon collision.

File Search and Retrieval Algorithm

a) Obtain or calculate the binary file search key.

b) Send a message to yourself with the key and a "hops to live" counter.

c) If the data is stored locally the request is satisfied.

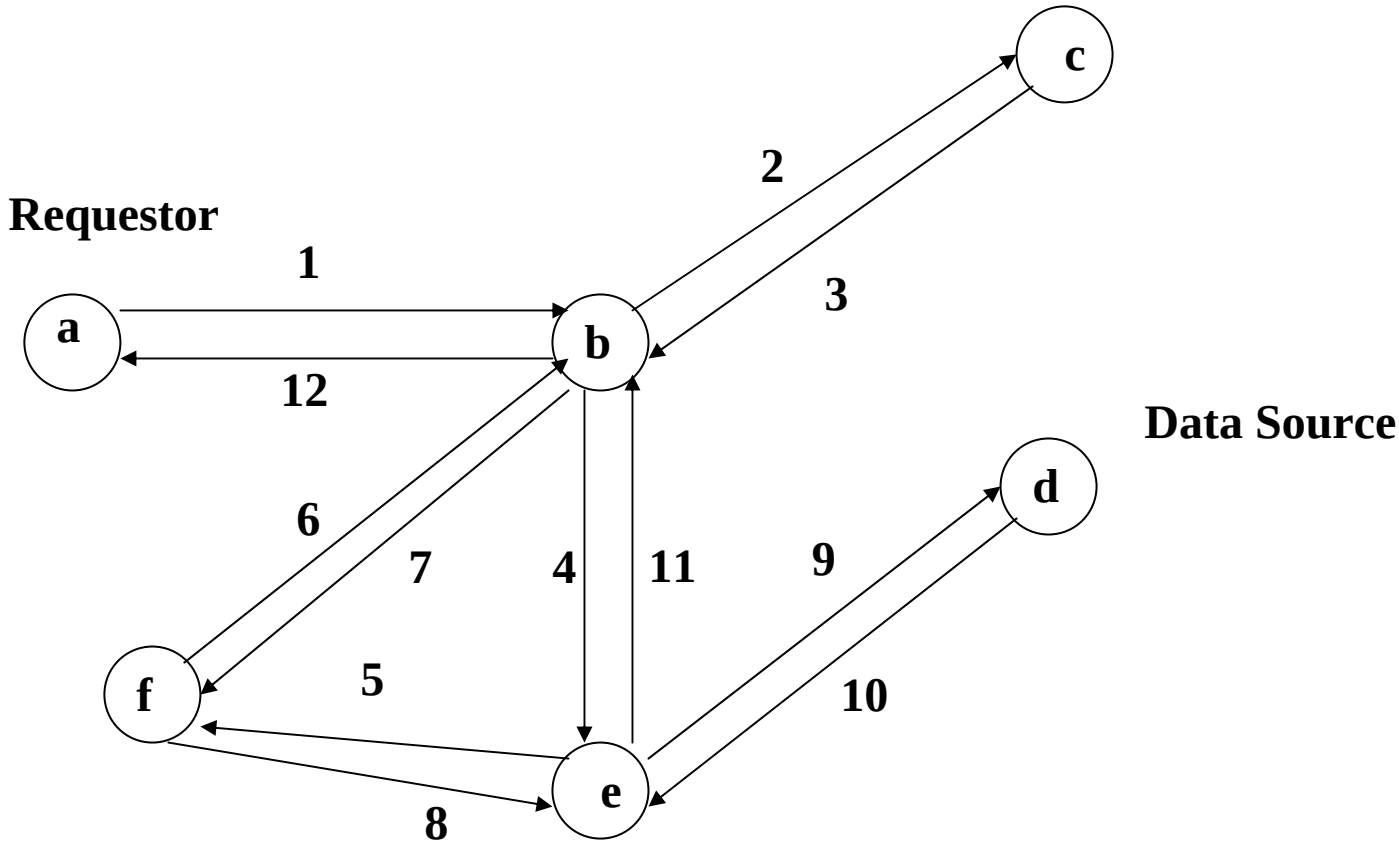
d) If the data is not found then the search is continued at the node in the local routing table associated with the key "nearest" to the search key.

e) If the search is successful then the data is returned to the upstream requestor which will cache the data and create a new entry in its routing table associating the actual data source with the key.

File Search and Retrieval Algorithm- Continued

f) If a node in the search path cannot forward a request to its preferred downstream node then the node with the second nearest key is selected as the downstream target, etc. If a node runs out of possible downstream paths without finding the file then it reports failure to the immediately upstream requestor. This upstream requestor then chooses its second choice target. This process is recursively followed until the originator runs out of downstream targets at which time failure is reported.

g) If the "hops to live" limit is exceeded at any time in this search process then the failure result is propagated back to the original requestor.


Requestor
Data Source
a
b
c
d
e
f
1
2
3
4
5
6
7
8
9
10
11
12

Properties of Search Algorithm

1. Nodes will tend to accumulate similar keys and similar files.

2. This should lead to better routing as well as better caching performance.

3. The greater the number of requests the more copies of the file which will exist.

4. The more localized the requests the more localized the number of copies of the file will become.

5. Network connectivity is increased as routing tables are built by requests and inserts.

6. Nodes with popular files will preferentially appear in routing tables of other nodes.

7. While files are clustered by key they are dispersed with respect to subject.

File Search and Retrieval for Up-datable Files

a) An up-datable file should be stored under a “content hash” key.

b) A file with the content hash key is inserted in a personal name space (signed-subspace key)

c) The file is encrypted with a random key which is published with the file key.

d) Retrieval is by first retrieving the file containing the content hash key and using this key to search for the file.

File Update Algorithm

a) A new version of an up-datable file is inserted under its content hash key.

B) The insert algorithm is executed for the new indirect file under the original signed subspace key.

C) When the insert reaches a node with the old version of the file a key collision will occur.

D) The node will check the signature on the new version, verify that it is correct and replace the old version of the file with the new version. Then the signed subspace key will always lead to the new version while a content hash search will still lead to the old version.

Management of Large Files

1. Partition a large file.
2. Create a content hash key for each partition.
3. Create a file to serve as the indirect access file for the partitions of the large file.
3. Insert the content hash key for each partition separately as an entry in the indirect file.
5. Insert the indirect file.

Publication of Names and Search Mechanisms

1. Create a search engine specific to FreeNet.

2. Create for each actual file a family of “lightweight indirect files” each named by a search keyword relating to the actual file and containing a pointer to the actual file.

3. Encourage users to create directories of “favorites”

Node Join Protocol

1. **The joining node must obtain the address of at least one node which is already a member of FreeNet by some out of band means.**
2. **The joining node chooses a random seed and sends an announcement message containing the hash of that seed, its address and a "hops to live" counter to the existing nodes for which it knows addresses.**
3. **When a node receives an announcement message it generates a random seed, XORs that with the hash it received and hashes the result again to generated a "commitment."**
4. **The node which received the announcement message forwards the new hash to some node chosen randomly from its routing table and decrements the hops to live counter.**
5. **The last node to receive the announcement message just generates a seed.**
6. **There all the nodes in the announce chain share their seeds and the key for the new node is assigned as the XOR of all of the seeds.**
7. **Then each of the nodes in the announce chain add the new node to their routing table under that key.**

FreeNet Communication Protocols

1. Each request, insert or join action is a transaction which has a (probably) unique ID associated with it.

2. Each message originated as a result of a transaction has a transaction ID which is carried in each message resulting from a transaction.

3.Node addresses are {transport identifier, transport specific identifier}, for example {tcp/192.168.1.1:19114}

4. Nodes which change addresses frequently may wish to used address resolution keys which are signed subspace keys updated to contain the current actual address of the node.

5. All transactions begin with a Request.Handshake message specifying the return address of the sending node.

6. The receiver of a Request.Handshake message may (or may not) respond with a Reply.Handshake message specifying the protocol it understands.

FreeNet Communication Protocols - Continued

7. All messages have a 64 bit randomly generated transaction ID, a hops to live counter, and a depth counter.

8. Hops to live and depth are set by the originator of a transaction is decremented by each receiver.

9. The propagation chain is continued with hops to live = 1 with some finite probability.

10. Depth is incremented at each hop and is used by a replying node to set hops to live high enough to reach a requestor. A depth of 1 is not automatically incremented but may be passed unchanged with some finite probability.

11. A time-out is superimposed on transactions.

12. Nodes in a chain may send back Reply.Restart messages based on knowledge of network delays not accessible to the originator.

Summary and Conclusions

1. Design achieves goals of symmetry, fully distributed control, parallelism, high security and use of "unused resources."
2. Performance properties are generally unknown except for simulations.
3. Algorithm domain for fully distributed control is largely unexplored.
4. State of the art is about where client-server systems were 10 years ago.
5. Next step - Integration of distributed computation and distributed file system.
6. Problem - Application design