Distributed Computer Systems Engineering

CIS 508: Lecture 11
Peer-to-peer Systems

Lecturer: Sid C.K. Chau
Email: ckchau@masdar.ac.ae

What is Peer-to-peer

Desirable architecture for distributed systems

- ***Equality***: Systems are equally privileged, equipotent participants in the application
- ***Sharing***: Systems contribute a portion of their resources (e.g. processing power, disk storage, network bandwidth) to other participating systems
  - There is minimal centralized coordination by other servers
- ***Robust***: No single point of failure
- ***Self-organizing***: Adaptable to changing dynamic of participating systems and requirement

What is Peer-to-peer

- A holy-grail for distributed system architecture
  - No centralized control
  - Nodes are symmetric in function
- Large number of (perhaps) server-quality nodes
- Plug-and-interact, with dynamic and adaptive system designs

Good about Peer-to-peer

- Resistant to DoS and failures
  - Resilience increases in numbers, no single point of attack or failure
- Self-organizing and adaptive
  - Nodes insert themselves into structure
  - Need no manual configuration or oversight
- Flexibility and heterogeneity
  - Can be distributed or co-located
  - Can comprise of powerful hosts or low-end devices
  - Mixed of trusted or unknown peers
- Plenty of killer applications (e.g. Naptser, Gnutella)
  - Share the content, storage, bandwidth of individual home users

Bad about Peer-to-peer

- Difficult to configure (or uncontrollable)
  - Machines may not grant privilege to be configured by external parties
- Unpredictable
  - Machines can come and go
  - Machines can be down at any time
- Security problem
  - Open to external machines
  - Can comprise of powerful hosts or low-end devices
- Incentive problem
  - Individual may not motivated to contribute resource
  - Every machine is selfish

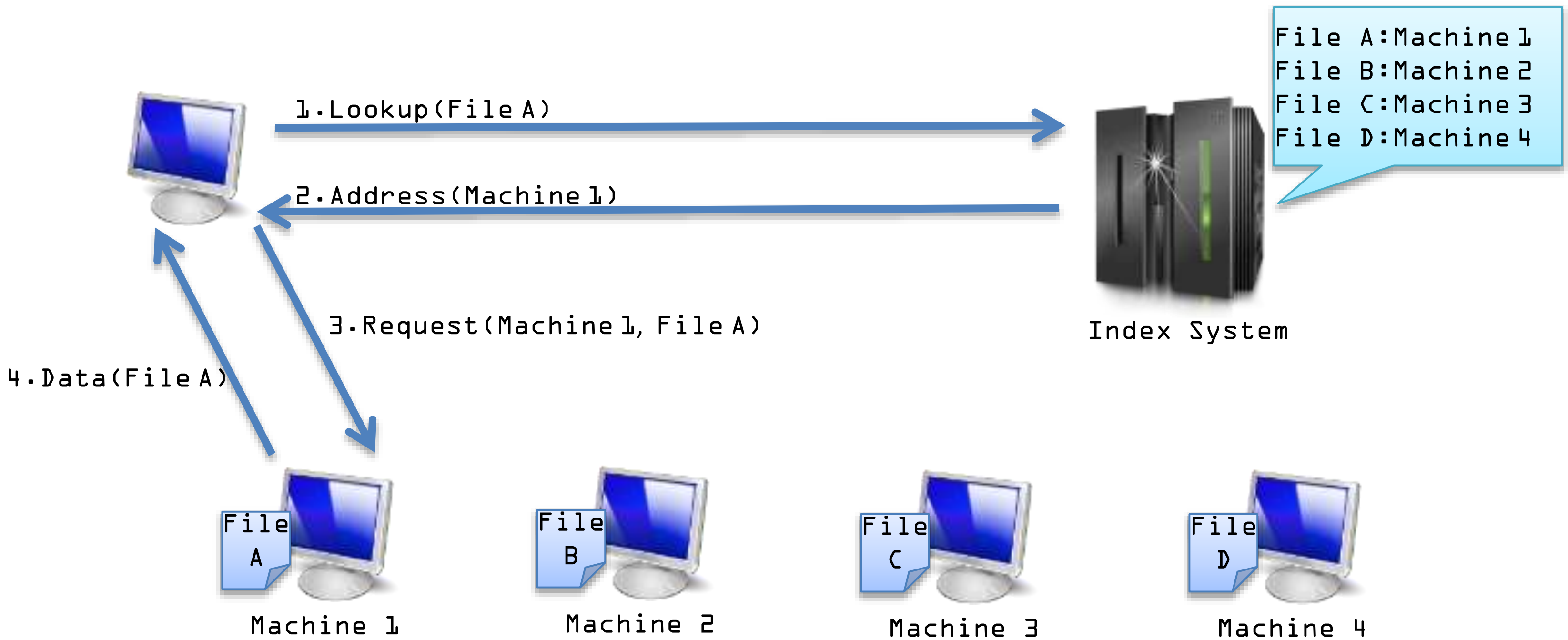
Application Example

- *Distributed peer-to-peer file storage system*
  - Each user stores a subset of files
  - Each user has access (and can download) files from all users in the system
  - Challenges
    - How to find where a particular file is stored?
    - How to achieve scalability?
      - Up to hundred of thousands or millions of machines
    - How to handle dynamicity?
      - Machines can come and go any time
      - Data can be lost, if machines are down

Napster

- Assume a centralized index system that maps files (songs) to machines that are present
- How to find a file (song)
  - Query the index system
  - Return a machine that stores the required file
    - Ideally this is the closest/least-loaded machine
  - Get the file from the machine
- Advantage
  - Simplicity, easy to implement search engines on index system
- Disadvantage
  - Robustness, scalability

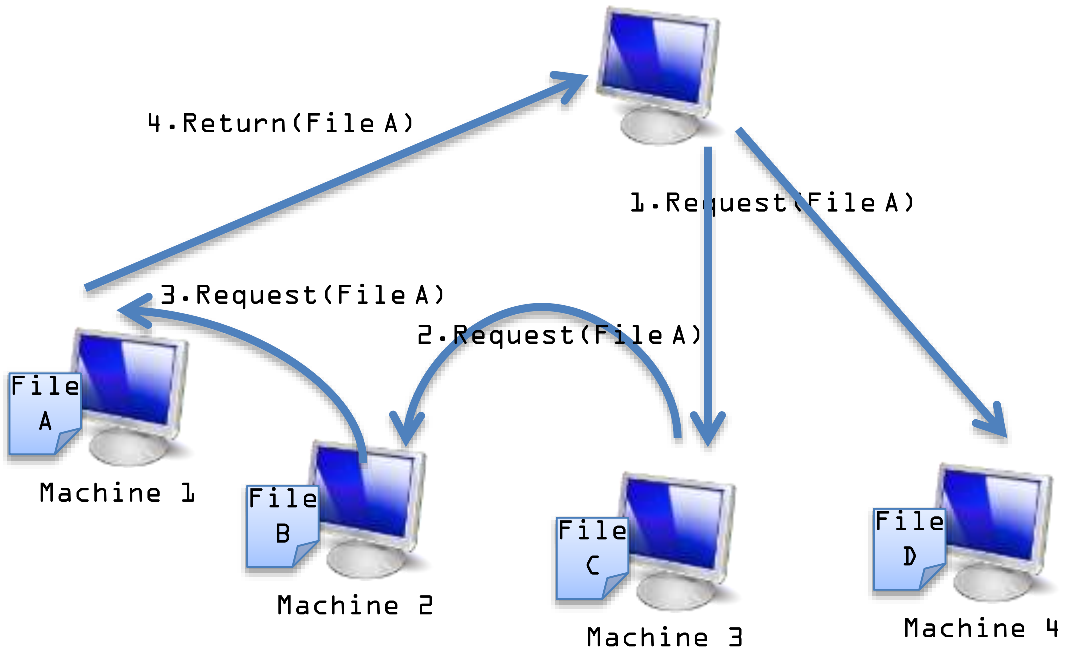
Napster


File A:Machine 1
File B:Machine 2
File C:Machine 3
File D:Machine 4
1.Lookup(File A)
2.Address(Machine 1)
3.Request(Machine 1, File A)
4.Data(File A)
Index System
File A
Machine 1
File B
Machine 2
File C
Machine 3
File D
Machine 4

Gnutella

- Locating file in distributed fashion
- Flood the request until reaching the machine has the file
- How to find a file (song)
    - Send request to all neighbors
    - Neighbors recursively multicast the request
    - Eventually a machine that has the file receives the request, and it sends back the answer
- Advantage
    - Totally decentralized, highly robust
- Disadvantage
    - Not scalable; the entire network can be swamped with requests

Gnutella


4.Return(File A)
1.Request(File A)
3.Request(File A)
2.Request(File A)
File A
Machine 1
File B
Machine 2
File C
Machine 3
File D
Machine 4

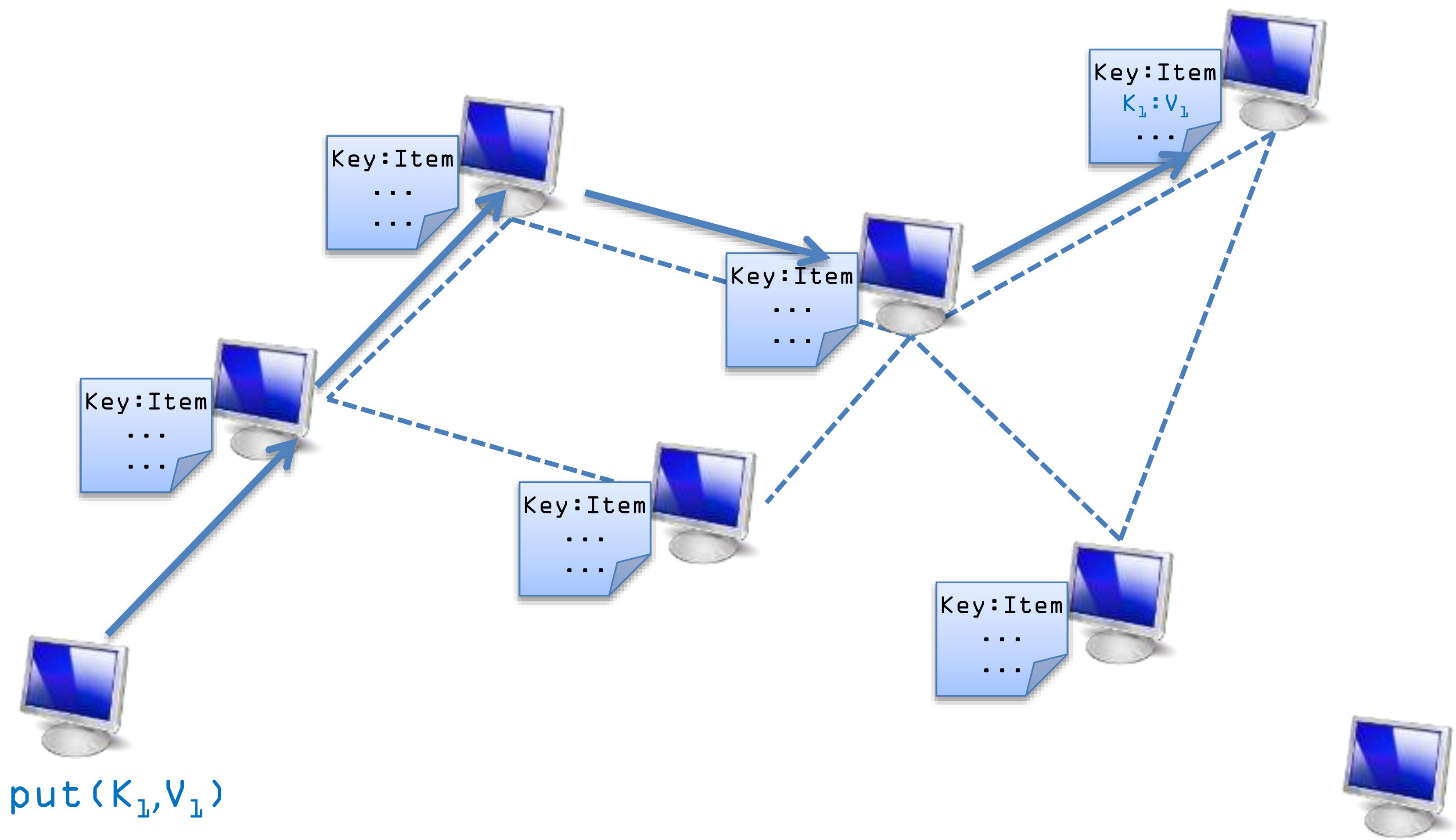
Classification

- *Unstructured peer-to-peer*
  - Do not impose any structure on topology of peers
  - Topology formalization in ad hoc fashion
  - E.g. Gnutella, eDonkey
- *Structured peer-to-peer*
  - Employ a globally consistent protocol to ensure topology conforming to certain structure
  - Optimize the performance of data query and retrieval
  - E.g. Distributed hash tables, CAN, Chord, Pastry

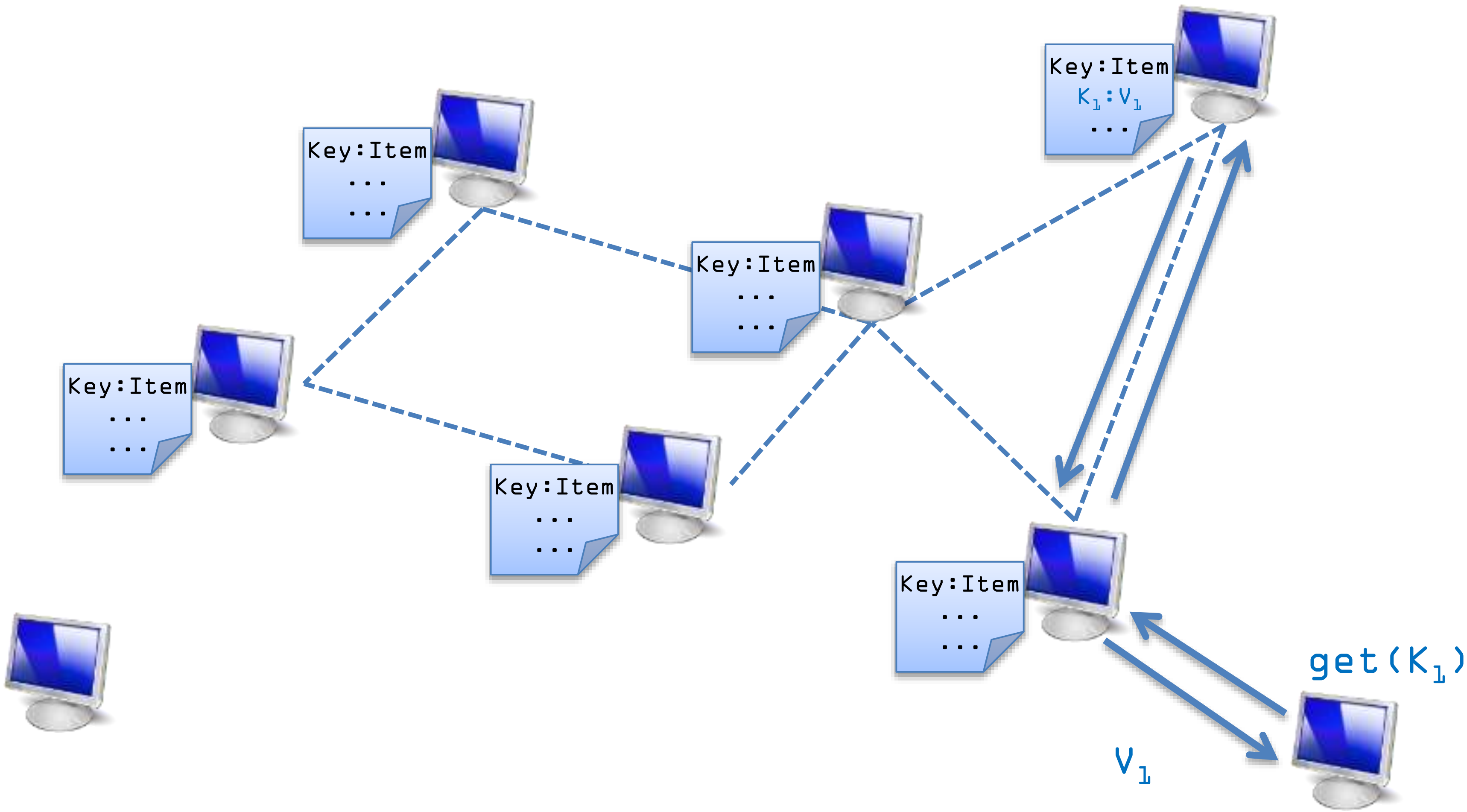
Distributed Hash Tables (DHTs)

- Structured peer-to-peer systems
- Abstract data structure model: Hash-table implemented by distributed systems
  - `put(key,item)`
  - `item = get(key)`
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Make sure that an item (file) identified is always found
- Scales to hundreds of thousands of nodes
- Handles rapid arrival and failure of nodes

DHT: put()


Key:Item
...
...
Key:Item
...
...
Key:Item
...
...
Key:Item
K_1:V_1
...
Key:Item
...
...
Key:Item
...
...
put(K_1,V_1)

DHT: get()


Key:Item
...
...
Key:Item
...
...
Key:Item
...
...
Key:Item
K_1:V_1
...
Key:Item
...
...
Key:Item
...
...
get(K_1)
V_1

DHT Applications

- Databases, file system, storage, archival
- Web serving, caching
- Content distribution
- Query & indexing
- Naming systems
- Chat services
- Application-layer multi-casting
- Event notification services
- Publish/subscribe systems
- Communication primitives

Challenges

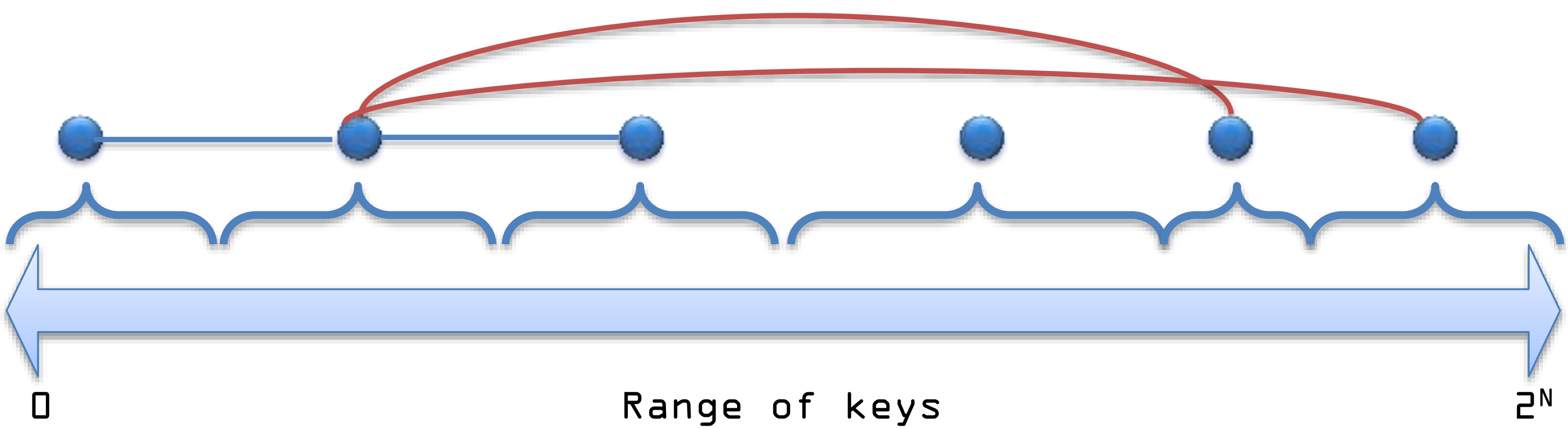
- How to ensure that all puts and gets for a particular key must end up at the same machine?
  - Even in the presence of failures and new nodes (*churn*)
- How to ensure fast lookup in a large peer-to-peer systems?
- How to ensure even distributed resource among peers?
- How to ensure scalability and minimal coordination among peers?
- How to support more sophisticated query and data structures?
- How to ensure security, interoperability, reliability?
- How to deal with free-riding (better incentive mechanism)?

Simplest DHT

- Step 1: Partition Key Space
- Each node in DHT will store some (k,v) pairs
- Given a key space K, e.g. $[0, 2^N)$:
  - Choose an identifier for each node, $id_i \in K$, uniformly at random
  - A pair (k,v) is stored at the node whose id is closest to k

Simplest DHT

- Step 2: Build Overlay Network
- Each node has two sets of neighbors
- *Immediate neighbors in the key space*
  - Step-by-step lookup; important for correctness
- *Long-hop neighbors*
  - Fast lookup; reachable in O(log n) hops



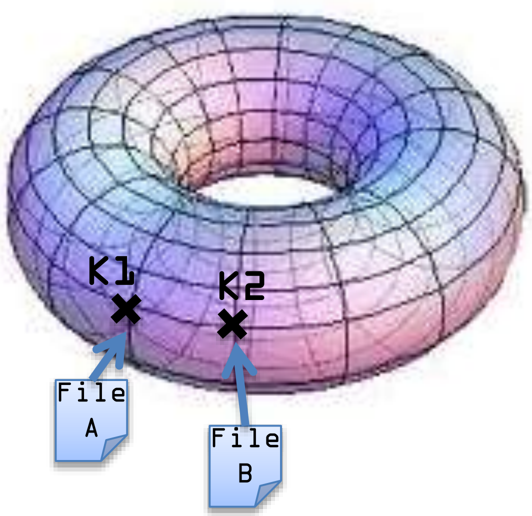
Node Departure

- Node explicitly hands over its zone and the associated (k,v) database to one of its neighbors
- In case of network failure this is handled by a take-over algorithm
- Problem
  - Take over mechanism does not provide regeneration of data
- Solution
  - Every node has a backup of its neighbours

DHT Systems

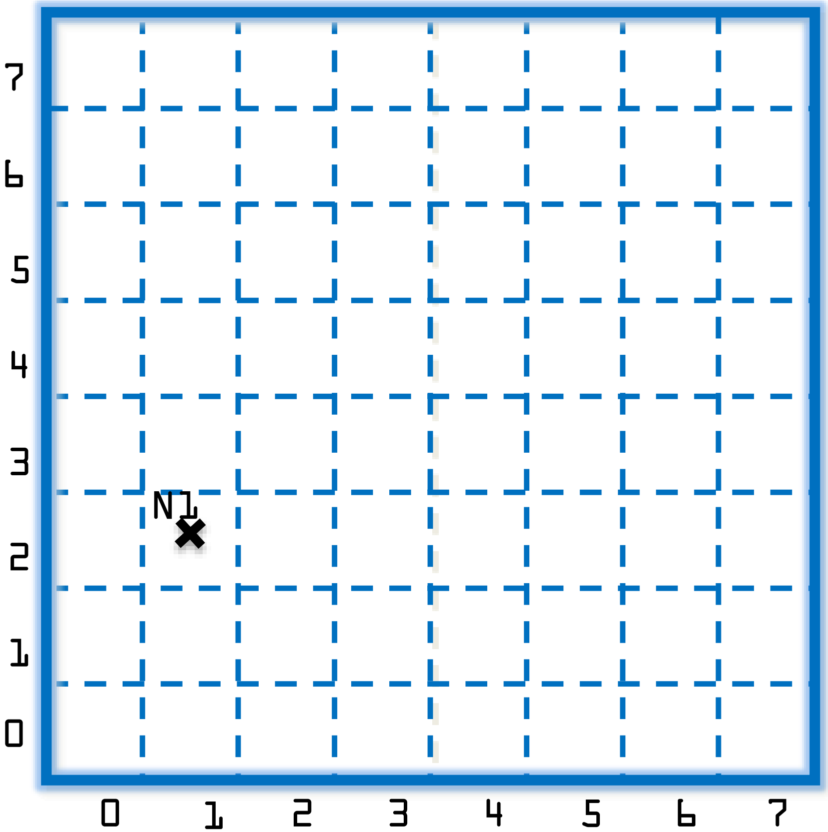
- All began in 2001…
- *CAN*
  - UC Berkeley
- *Chord*
  - MIT
- *Pastry*
  - Microsoft Research Cambridge
- *Tapestry*
  - UCSB/UC Berkeley

Content Addressable Network (CAN)



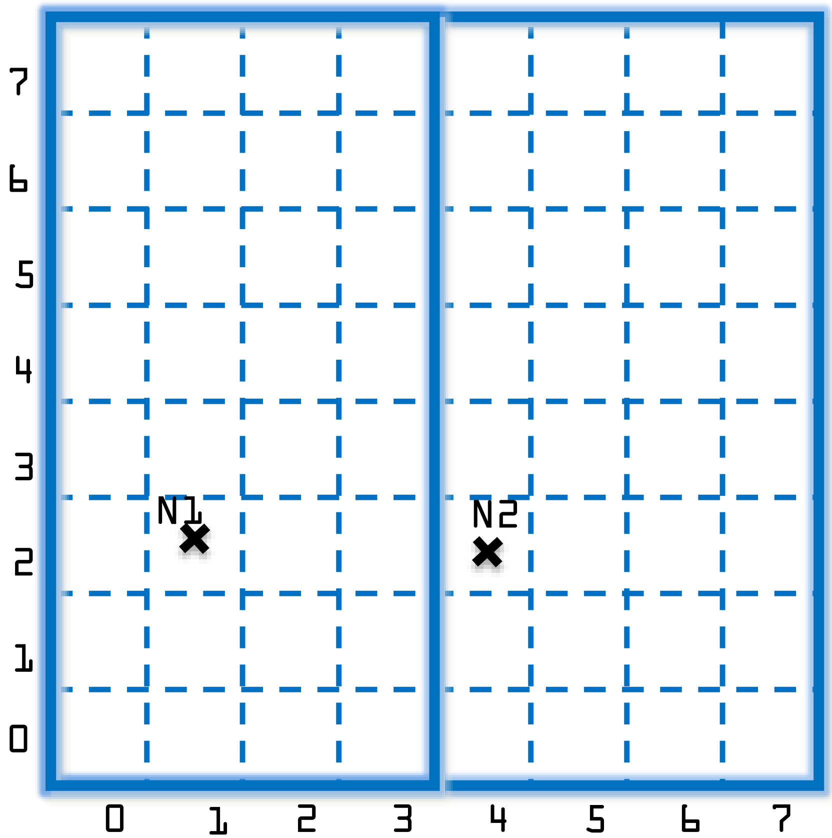
- Associate to each node and item a unique *id* in an d-dimensional Cartesian space on a *d*-torus
- Properties
  - Routing table size $O(d)$
  - Guarantees that a file is found in at most $d \times n^{1/d}$ steps, where n is the total number of nodes

CAN: Example



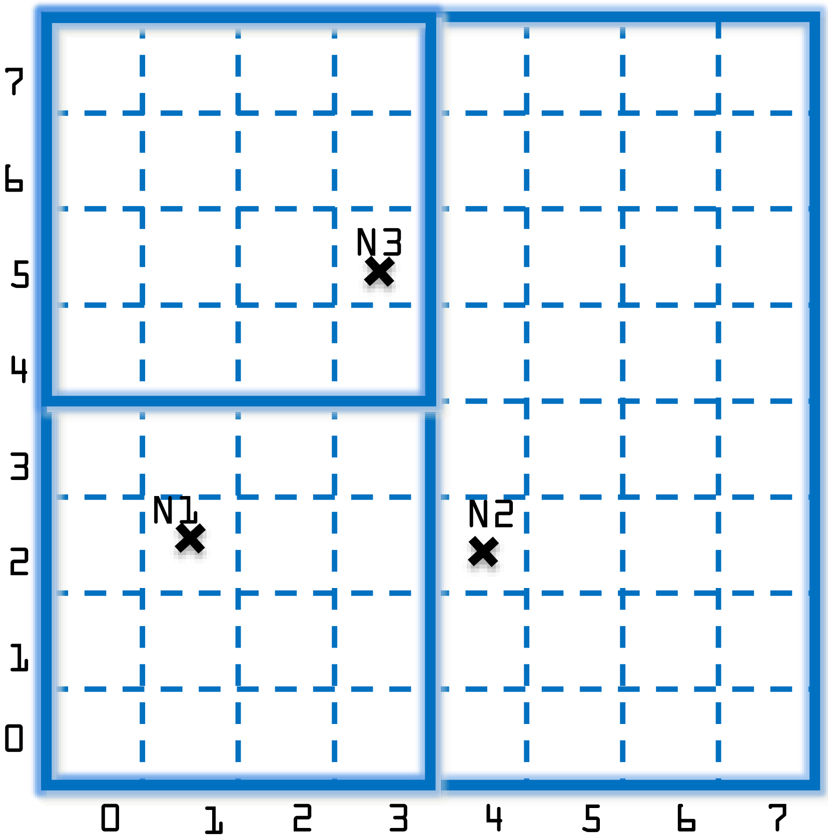
- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example
  - Node `N1` with coordinate (1, 2) first node that joins
  - `N1` covers the entire space

CAN: Example



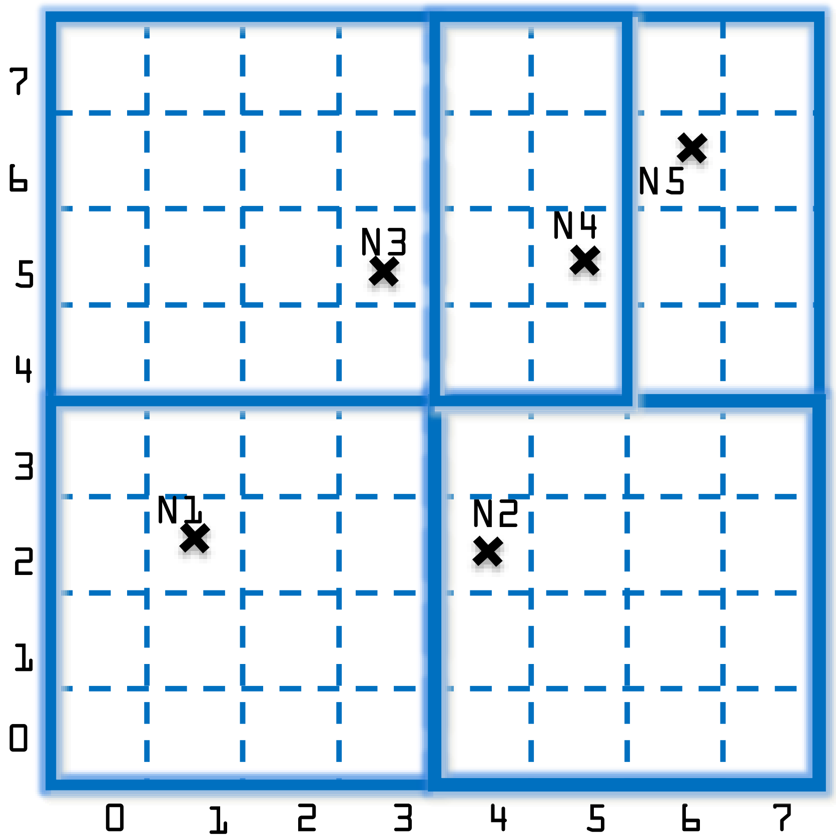
- Node N2 with coordinate (4, 2) joins
- Space is divided between N1 and N2

CAN: Example



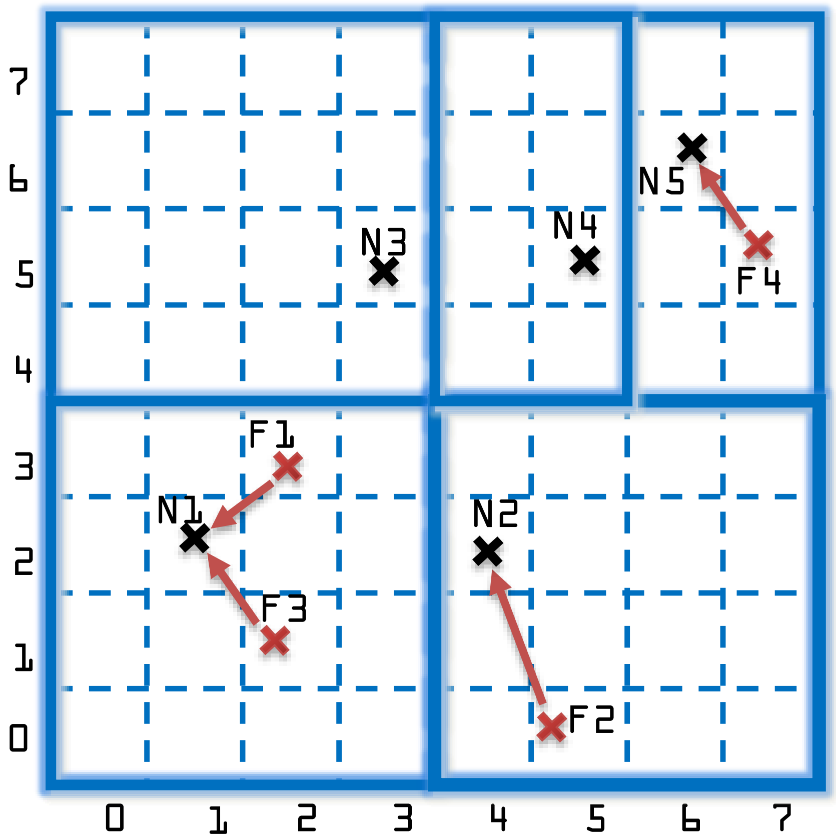
- Node N3 with coordinate (3, 5) joins
- Space is divided between N1 and N3

CAN: Example



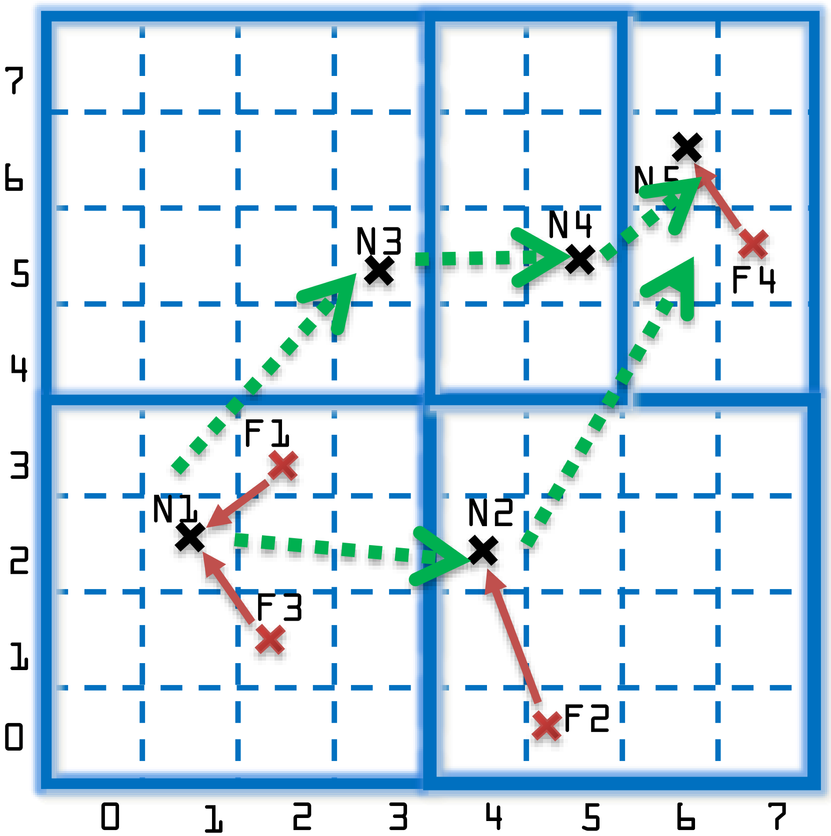
- Node N4 with coordinate (5, 5) joins
- Node N5 with coordinate (6, 6) joins

CAN: Example



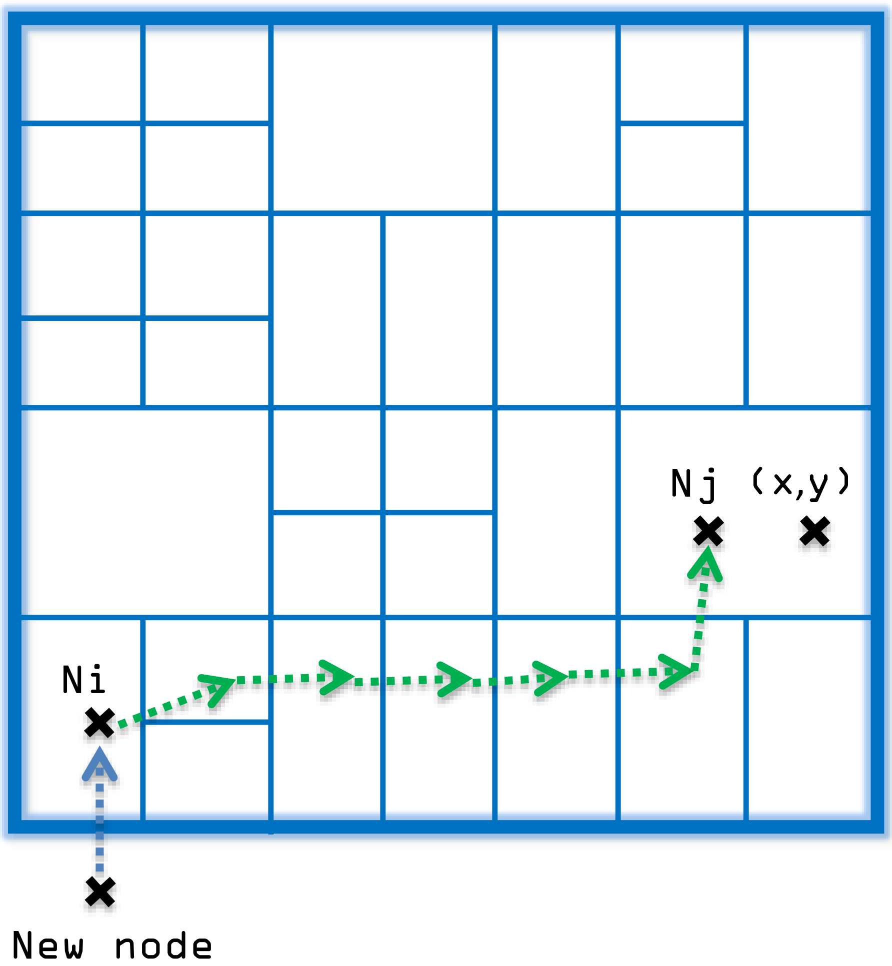
- Nodes
  - N1=(1, 2); N2=(4,2); N3=(3, 5); N4=(5,5); N5=(6,6)
- Items
  - F1=(2,3); F2=(5,1); F3=(2,1); F4=(7,5);
- Each item is stored by the node who owns its mapping in the space

CAN: Query



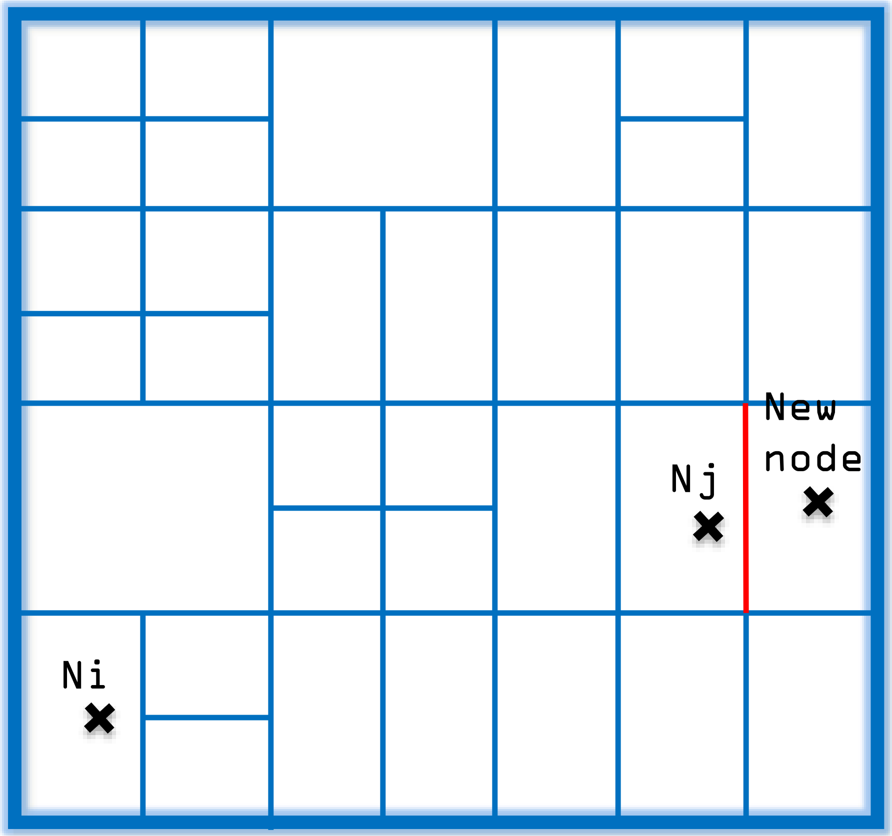
- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: N1 wants to query F4
- Can route around some failures

CAN: Joining



1. Discover some node `Ni` already in CAN
2. Pick random point `(x,y)` in space
3. routes to `(x,y) and` discovers node `Nj`

CAN: Joining



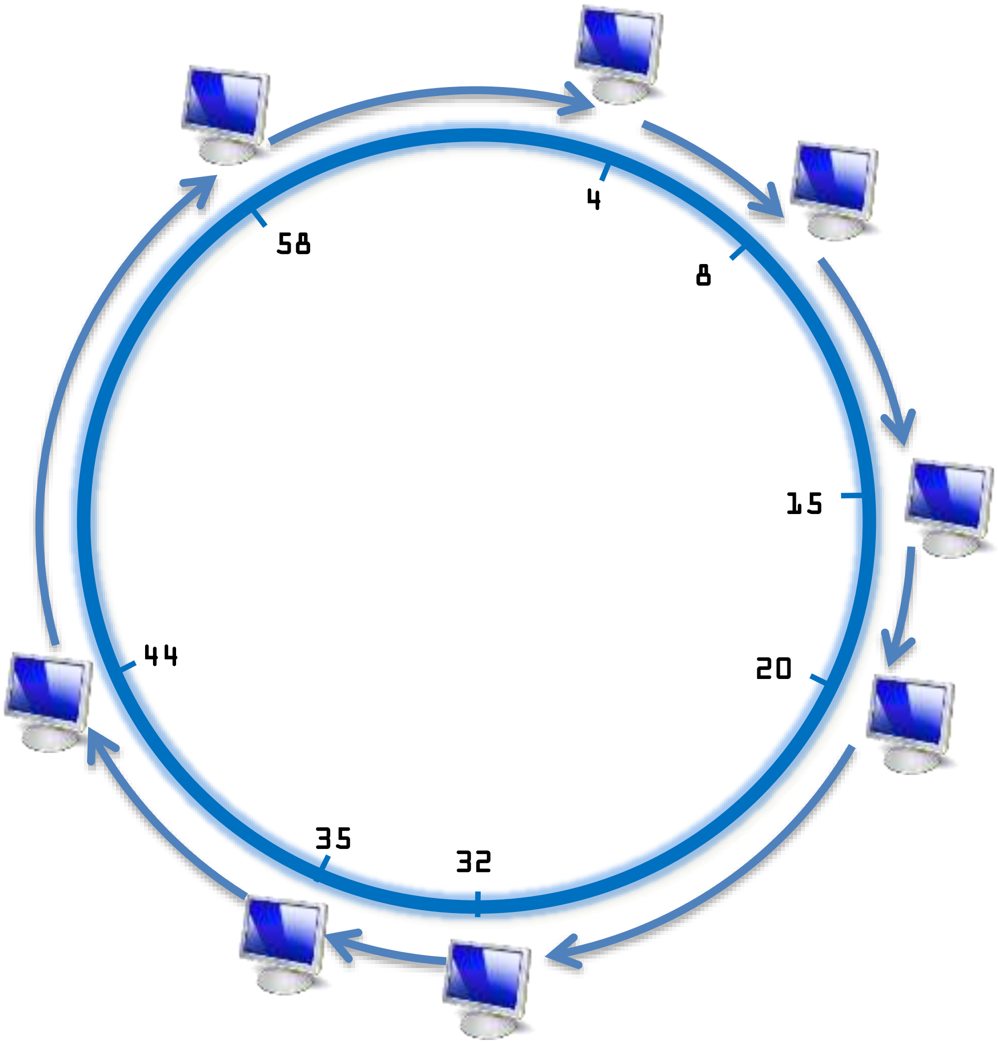
1. Discover some node `Ni` already in CAN
2. Pick random point `(x,y)` in space
3. Route to `(x,y)` and discovers node `Nj`
4. Split `Nj`'s zone in half

   new node owns one half

Chord

- Associate to each node and item a unique *id* in an *uni-dimensional* space $[0,\ldots, 2^m-1]$
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size $O(\log(N))$, where N is the total number of nodes
  - Guarantees that a file is found in $O(\log(N))$ steps

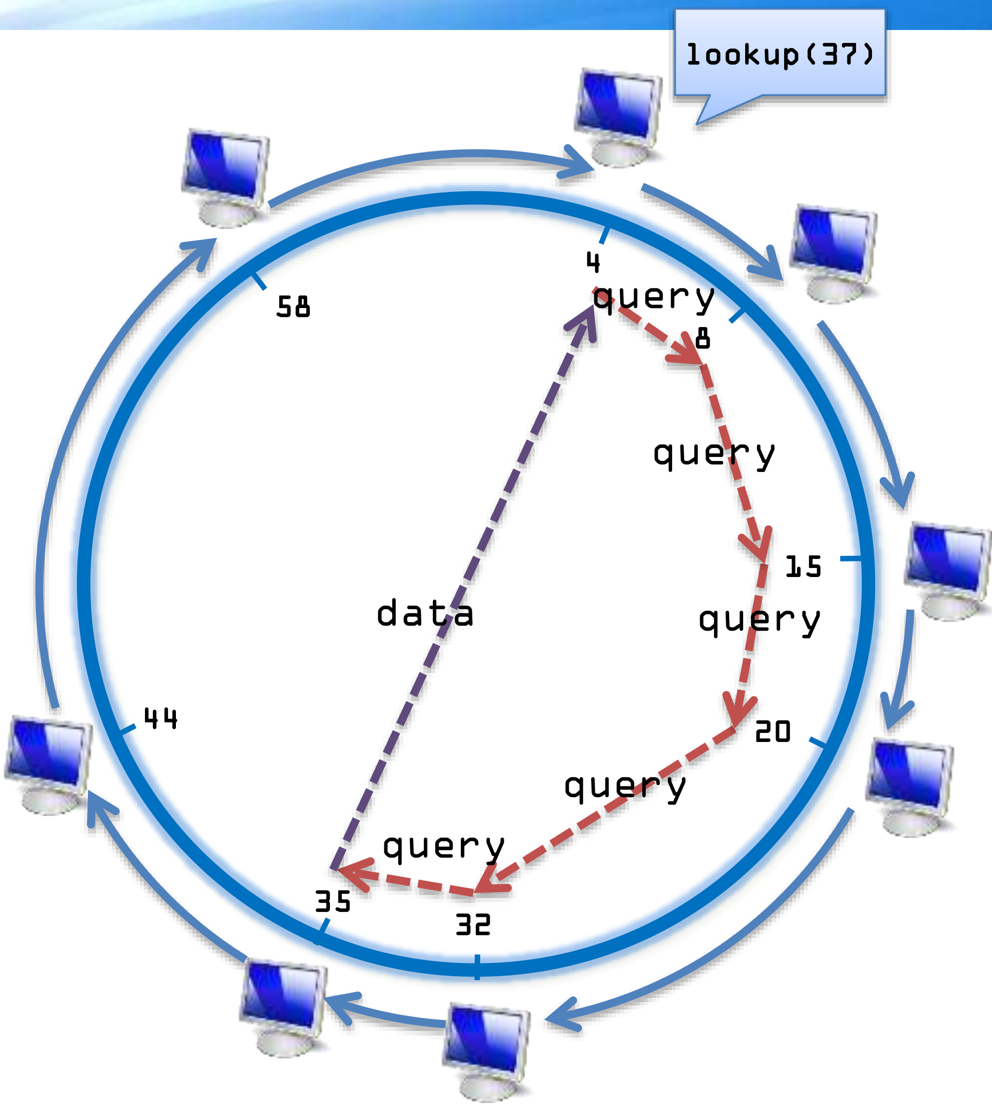
Chord: Mapping



- Node 8 maps [5,8]
- Node 15 maps [9,15]
- Node 20 maps [16, 20]
- …
- Node 4 maps [59, 4]

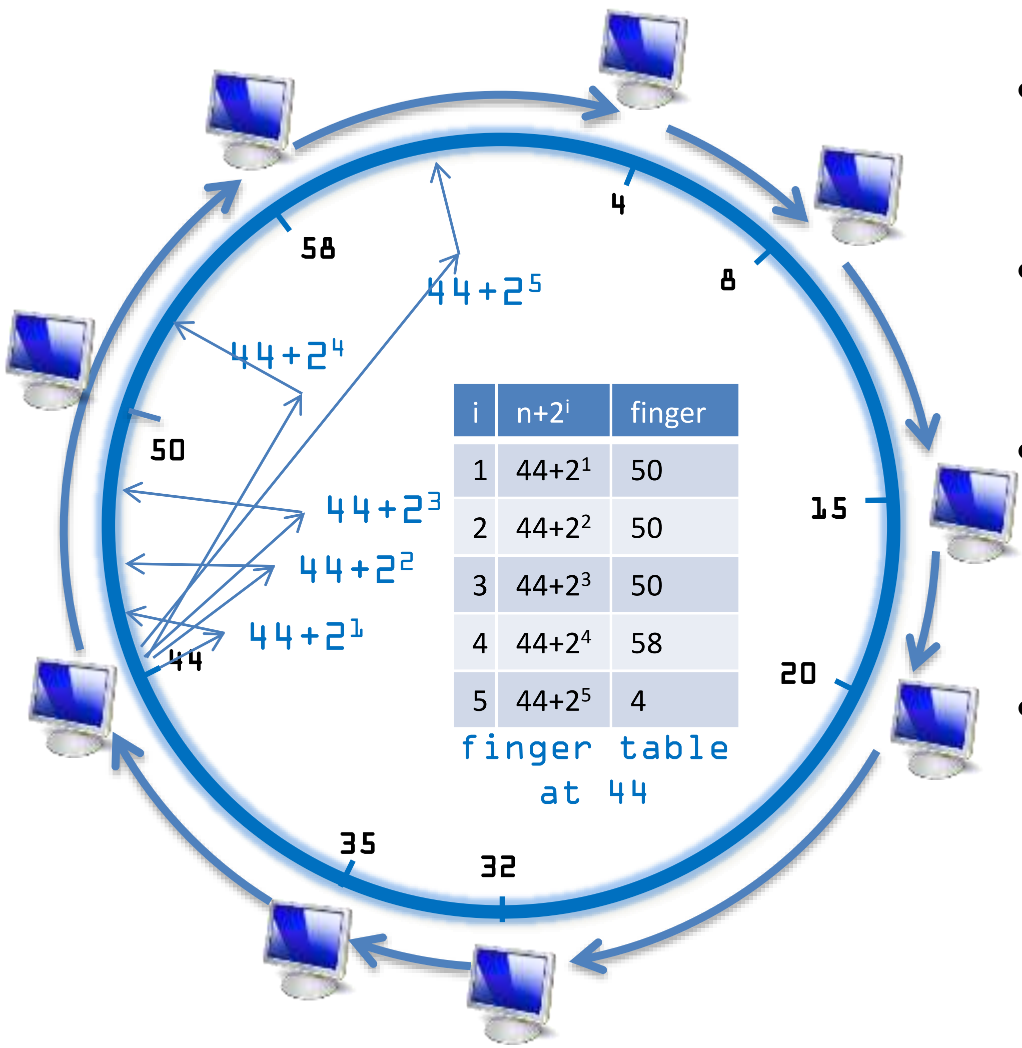
- Each node maintains a pointer to its successor

Chord: Mapping



- Each node maintains its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers

Chord: Lookup



i	$n+2^i$	finger
1	$44+2^1$	50
2	$44+2^2$	50
3	$44+2^3$	50
4	$44+2^4$	58
5	$44+2^5$	4

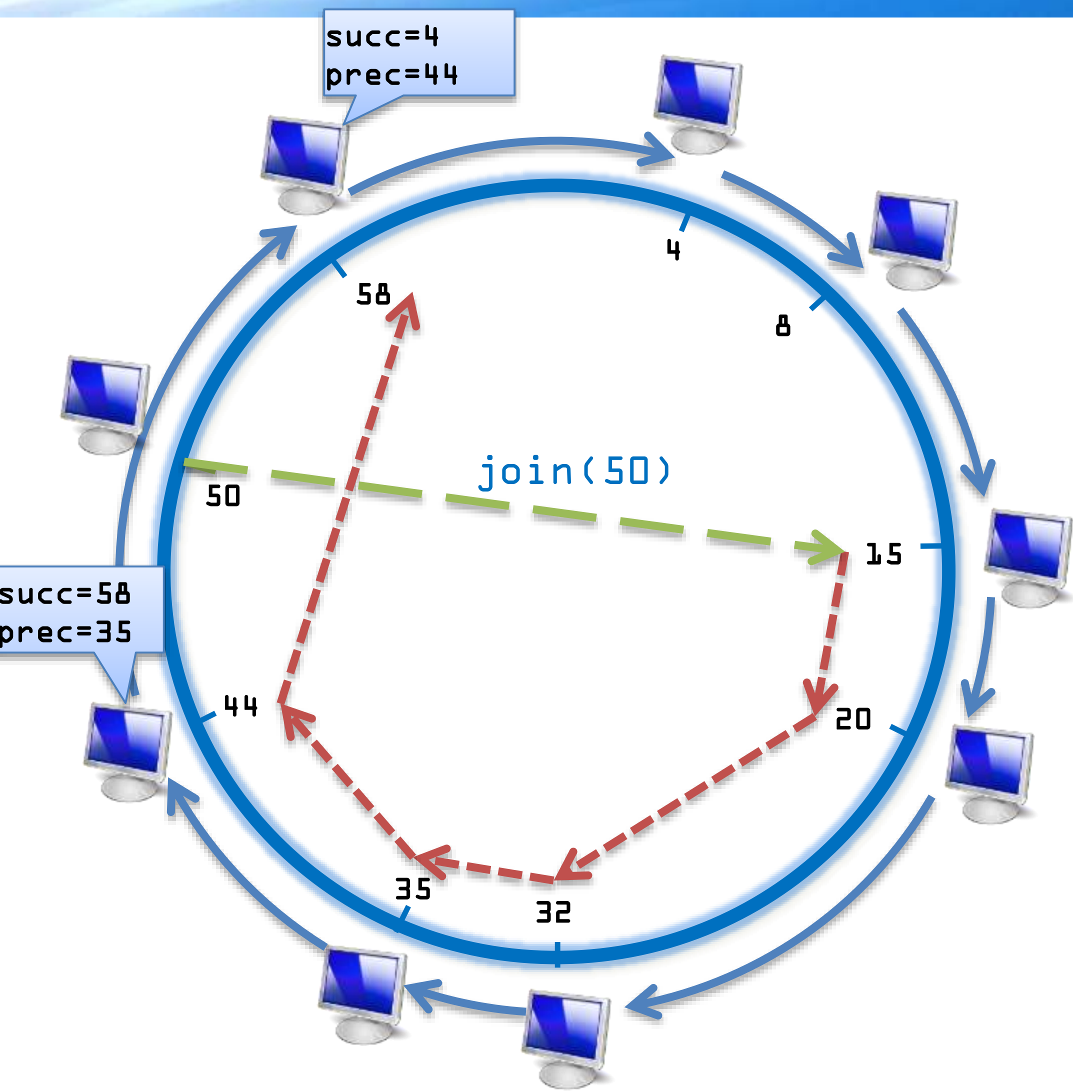
finger table at 44

- Locally storing neighbor ids of further hops away
- i-th entry at peer with id *n* is first peer with id >= $n+2^i$
- Use greedy routing to forward to the closet node in the finger table
- Shorten the lookup time to O(log n), rather O(n)

Chord: Joining

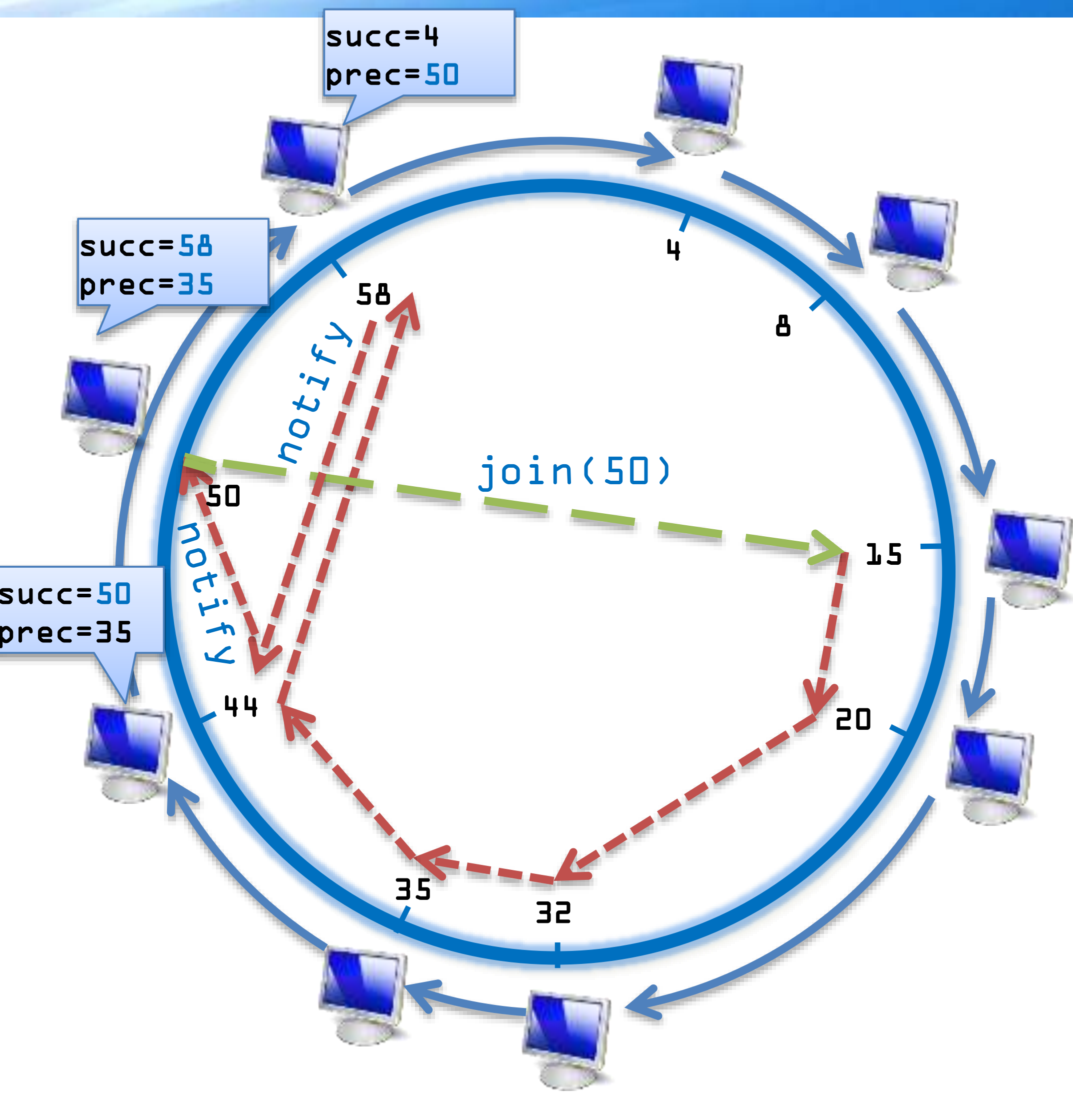
- Each node A periodically sends a stabilize() message to its successor B
- Upon receiving a stabilize() message node B
  - returns its predecessor B’=pred(B) to A by sending a notify(B’) message
- Upon receiving notify(B’) from B,
  - if B’ is between A and B, A updates its successor to B’
  - A does nothing, otherwise

Chord: Joining



- Node 50 asks node 15 to forward join message
- When join(50) reaches the destination (i.e., node 58), node 58
  1) updates its predecessor to 50,
  2) returns a notify message to node 50
- Node 50 updates its successor to 58

Chord: Joining



- Node 44 sends a stabilize message to its successor, node 58
- Node 58 reply with a notify message
- Node 44 updates its successor to 50
- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44

Achieving Robustness

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- In the `notify()` message, node A can send its k-1 successors to its predecessor B
- Upon receiving `notify()` message, B can update its successor list by concatenating the successor list received from A with A itself

DHTs vs Unstructured P2P

- DHTs good at:
  - Exact match for “rare” items
- DHTs bad at:
  - Keyword search, etc. (Cannot construct DHT-based Google)
  - Tolerating extreme churn
- Gnutella etc. good at:
  - General search
  - Finding common objects
  - Very dynamic environments
- Gnutella etc. bad at:
  - Finding “rare” items

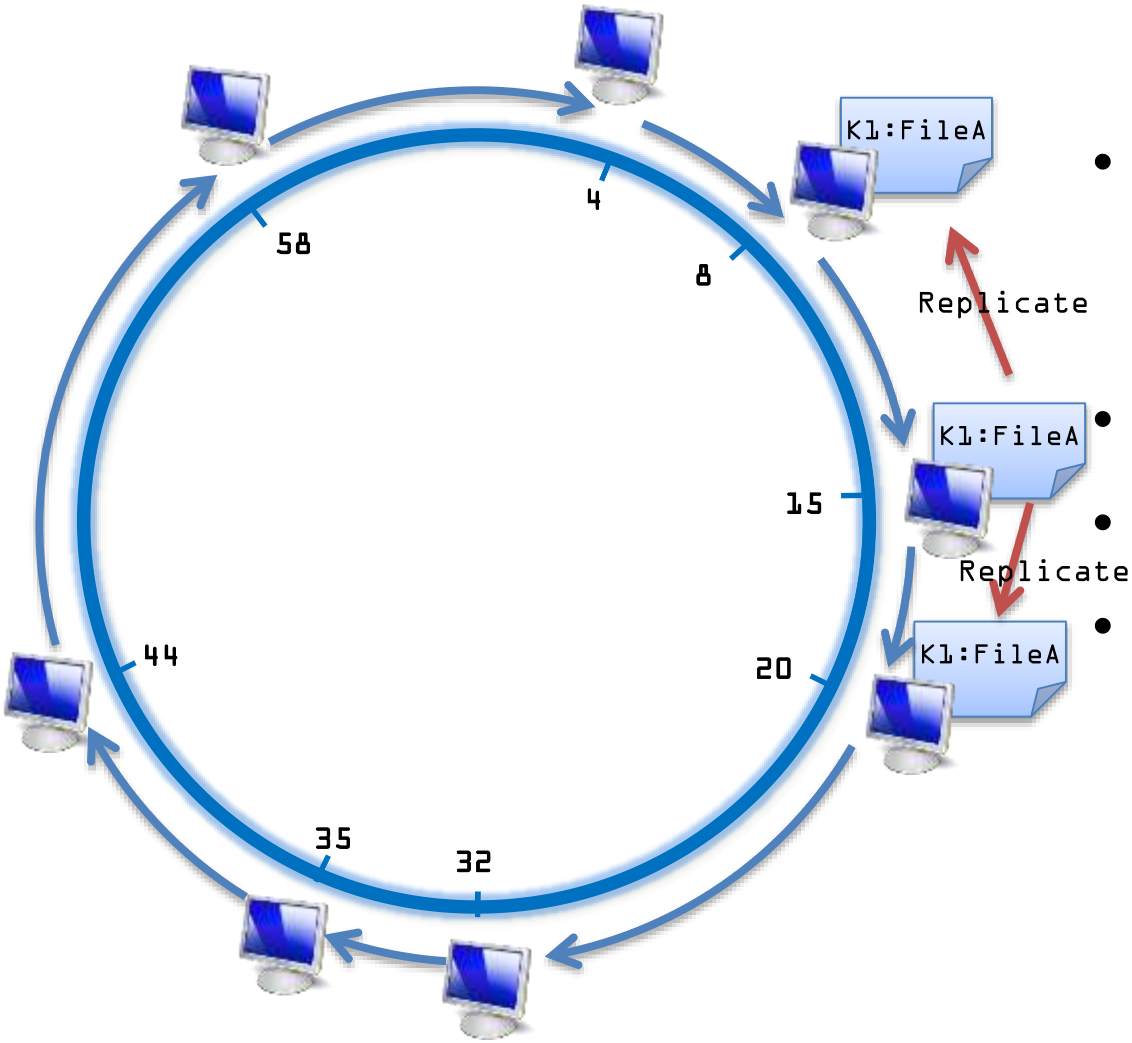
Key Applications of DHT

- Rendezvous applications
  - Phone book, lookup for global id to permanent id
  - E.g. lookup the phone number from passport id
- Storage applications
  - Storing larger file than id
- Routing and multicasting
  - Topology structure

Storage Applications

- Rendezvous applications use the DHT only to store small pointers (IP addresses, etc.)
- What about using DHTs for more serious storage, such as file systems
- Examples
  - File Systems, backup, archiving, electronic mail
  - Content Distribution Networks
- Why store data in DHT?
  - High storage capacity: many disks
  - High serving capacity: many access links
  - High availability by replication
  - Simple application model

Data Availability via Replication



- DHT replicates each key/value pair at the nodes after it on the circle
- It is easy to find replicas
- Put(k,v) to all
- Get(k) from closest

Problems with Centralized Servers

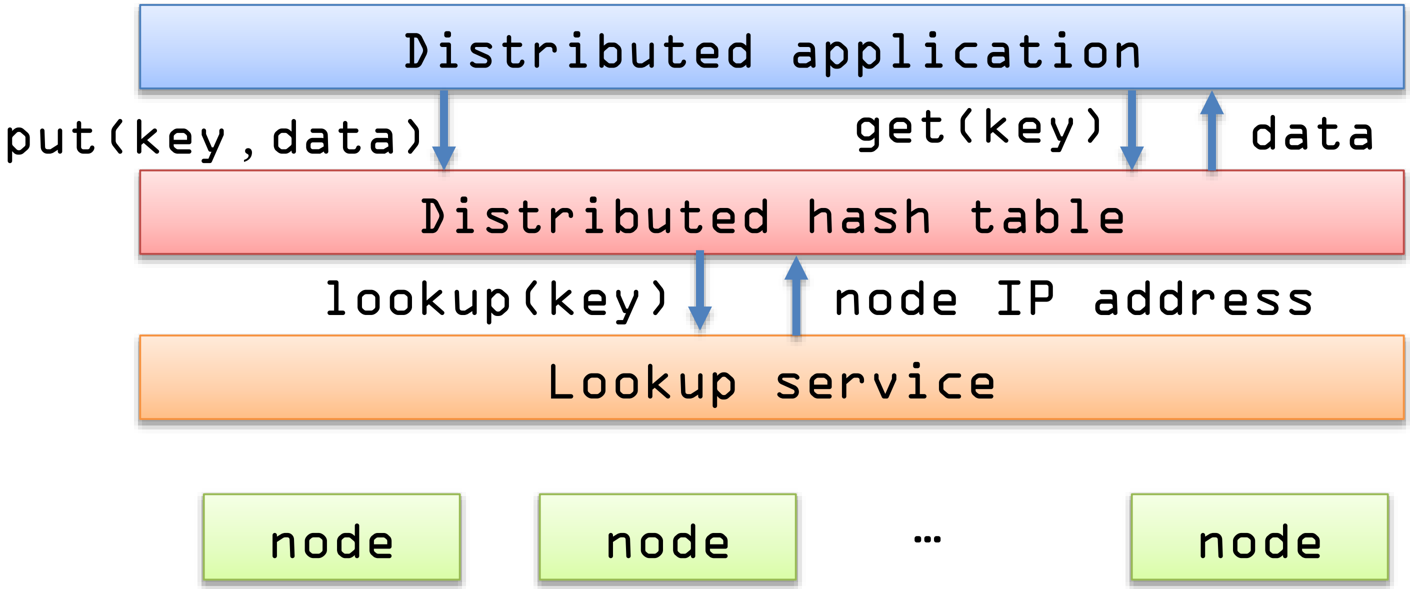
- Weak availability:
  - Susceptible to point failures and DoS attacks
- Management overhead
  - Data often manually partitioned to cope with scale
  - Management and maintenance give large fraction of cost
- Per-application design
  - High hurdle for new applications
- Do not leverage the advent of powerful clients
  - Limits scalability and availability

The DHT Community's Goal

Produce a common infrastructure that will help solve these problems by being:

- Robust in the face of failures and attacks
  - Availability solved
- Self-configuring and self-managing
  - Management overhead reduced
- Usable for a wide variety of applications
  - No per-application design
- Able to support very large scales, with no assumptions about locality, etc.
  - No scaling limits, few restrictive assumptions

DHT Layering



- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

Universal Interface of DHT

- Challenge for P2P systems: finding content
  - Many machines, must find one that holds file
- Essential task: Lookup(key)
  - Given key, find host (IP) that has file with that key
- Higher-level interface: Put()/Get()
  - Easy to layer on top of lookup()
  - Allows application to ignore details of storage
    - System looks like one hard disk
  - Good for some apps, not for others