Peer-to-peer distributed file system

Lab assignment D7024E

These slides introduce concepts to be used in the lab.
Detailed instructions are in the lab assignment doc.

Goal

- Practical experiences building distributed systems
  - e.g. how to handle scalability & resilience
- Learn more about underlying technologies used in today's cloud platforms

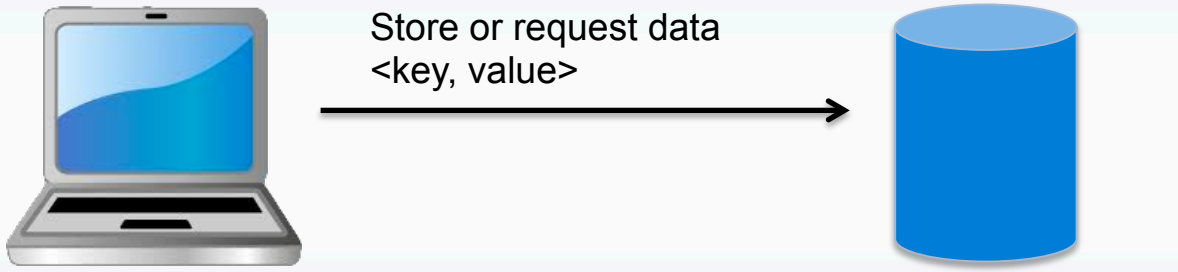
Background

- Cloud storage
  - Data stored in digital pools spawning multiple servers (and location)
  - Made up of many distributed resources
  - Highly fault tolerant through redundancy and distribution of data
- Example of services
  - Dropbox, Amazon S3, Google Cloud Storage, IPFS etc.

Develop your own cloud storage service!

- Flexibility and Openness
  - Anyone should be able to set up a server and provide storage resource to the network
- Scalability
  - Stored data should be distributed among servers
  - New servers can be added during runtime
- High availability
  - Replication so that data is not lost even if a server disconnects
- Security
  - Data encryption should be supported

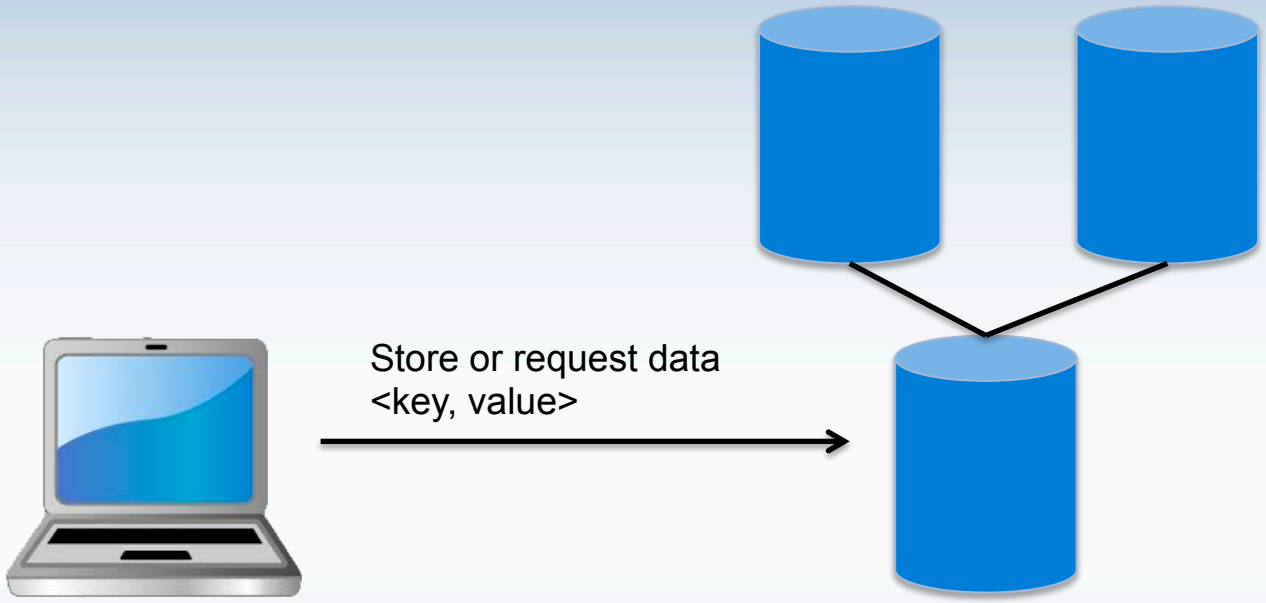
Problem



Database server

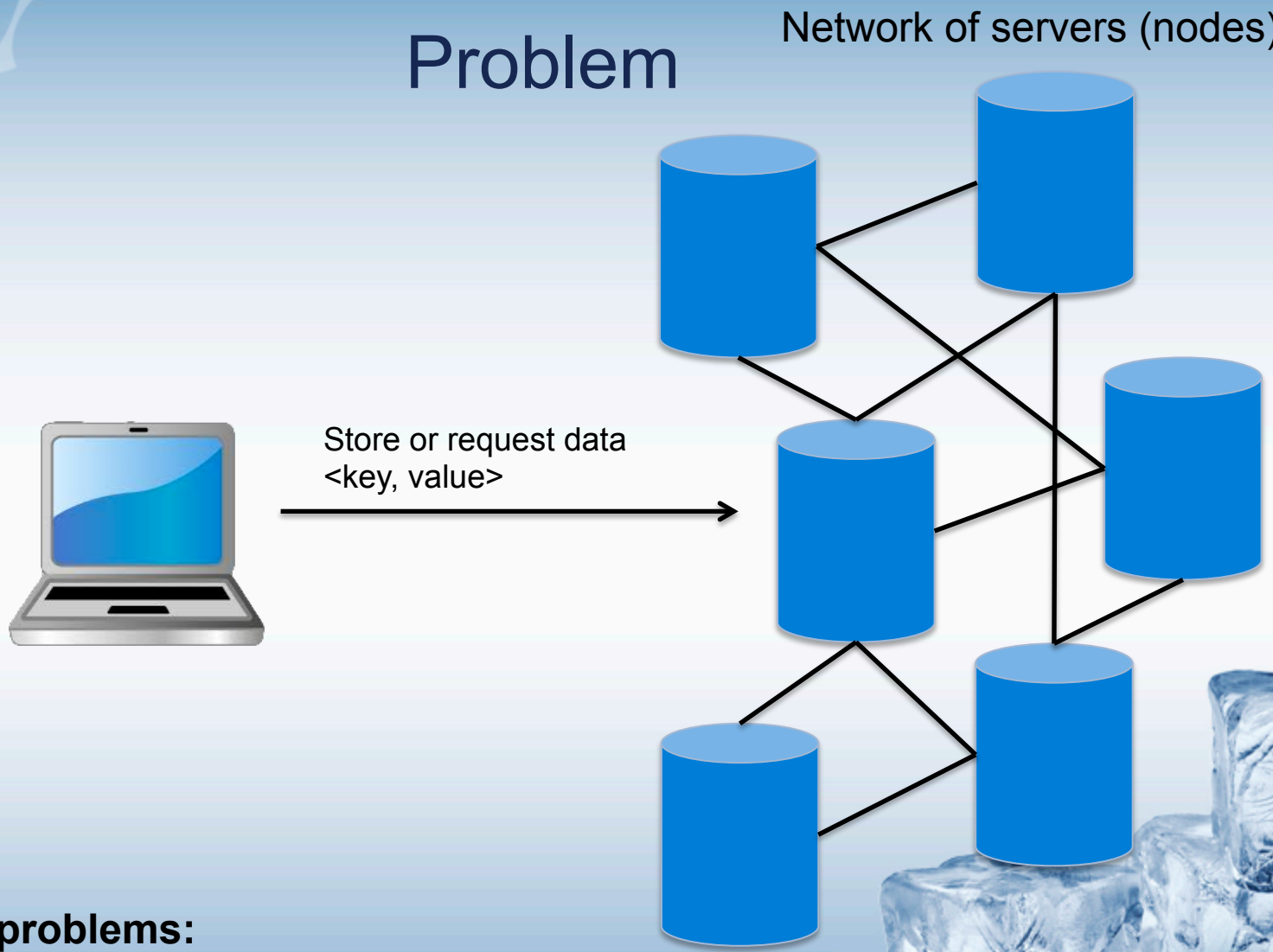
How to “backup” the database server?
How to make sure it doesn’t crash?
How to scale it up?

Problem



Solved: How to “backup” the database server?
Solved: How to make sure it doesn’t crash?
How to scale it up?

Problem

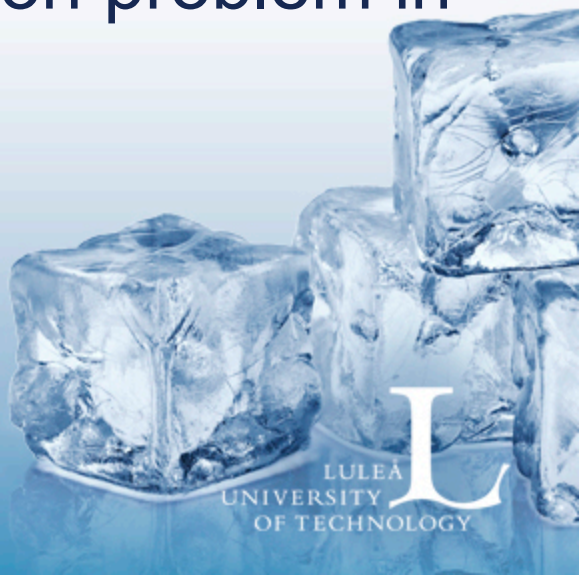


Two new problems:

- How do we select which node to store data?
- What happen if some node disconnects?
- How to mange new nodes?

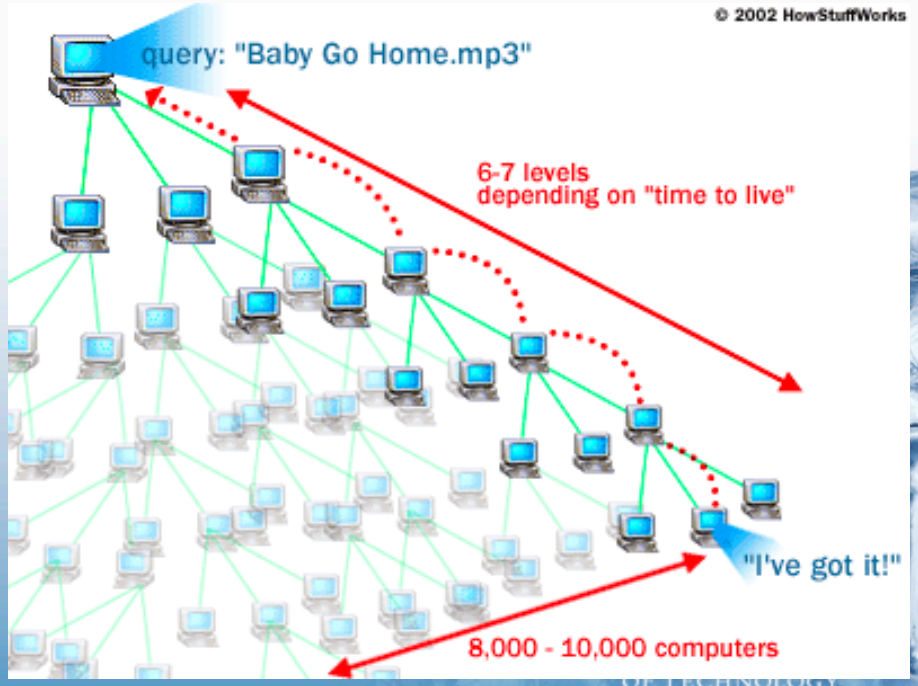
Distributed Lookup Problem

- Given a data item X stored at some dynamic set of nodes in the system, find it.
  - The problem of determining where a new data item shall be stored is the same
- This problem is important in many distributed systems, and it is the critical common problem in P2P systems.

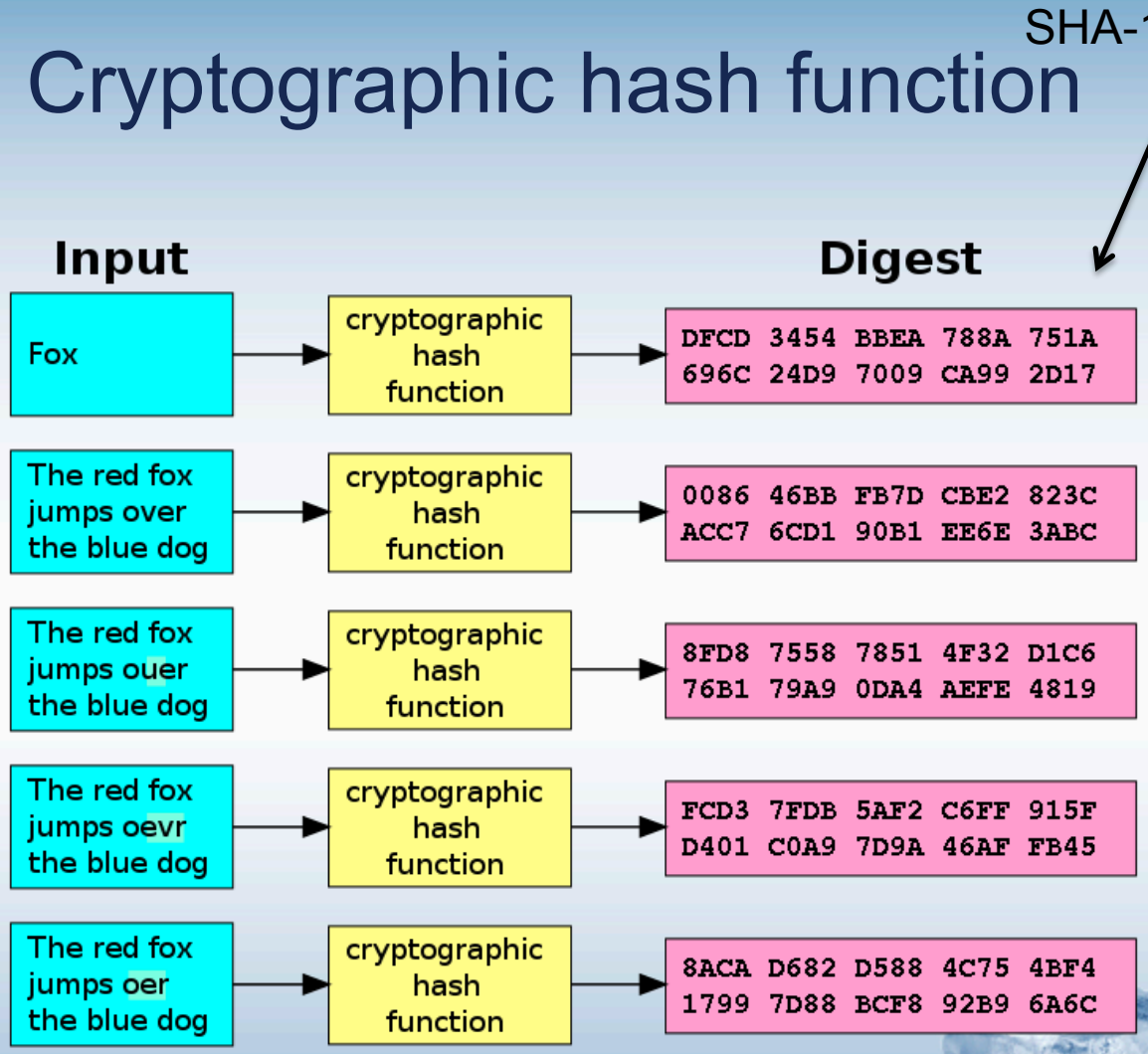


History & related problems

- First-generation P2P
  - Gnutella, Kazaa, Gosip, Napster
- Problems
  - Index server
  - Scalability problem
  - Flooding
  - Can not search entire network (TTL)
- How to efficiently find the owner of file?



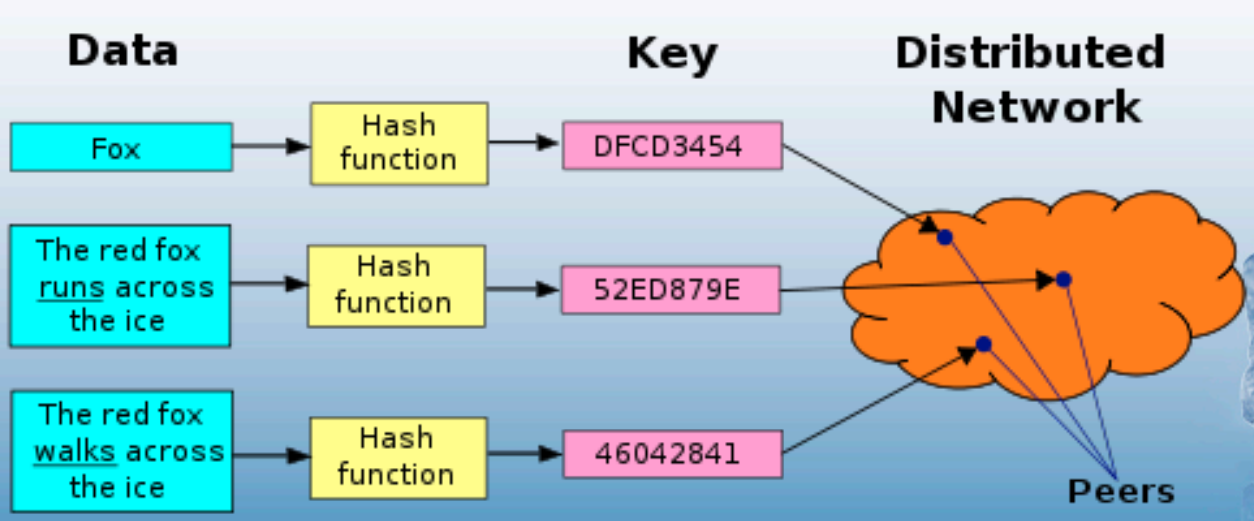
Cryptographic hash function



- Example of algorithms
  - SHA-1, MD5

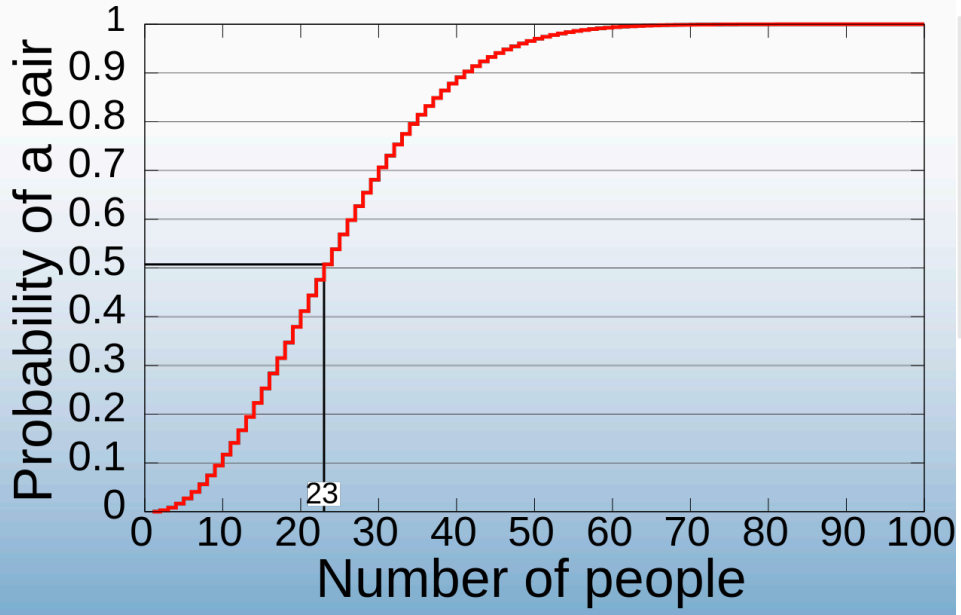
Distributed Hash tables (basic idea)

- A cryptographic hash function to calculated a digest for each key (keyspace)
  - Globally known
- Partitioning scheme to split ownership of the keyspace among participating nodes
  - Globally known strategy
- The participating nodes cooperate to find “owner” of a key



Birthday problem & hash collisions

- Probability randomly chosen people, some pair of them will have the same birthday
  - 99.9% probability is reached with just 70 people, and 50% probability with 23 people



Number of **32-bit** hash values	Number of **64-bit** hash values	Number of **160-bit** hash values	Odds of a hash collision	
77163	5.06 billion	1.42×10^{24}	1 in 2	
30084	1.97 billion	5.55×10^{23}	1 in 10	Odds of a full house in poker 1 in 693
9292	609 million	1.71×10^{23}	1 in 100	
2932	192 million	5.41×10^{22}	1 in 1000	Odds of four-of-a-kind in poker 1 in 4164
927	60.7 million	1.71×10^{22}	1 in 10000	
294	19.2 million	5.41×10^{21}	1 in 100000	Odds of being struck by lightning 1 in 576000
93	6.07 million	1.71×10^{21}	1 in a million	Odds of winning a 6/49 lottery 1 in 13.9 million
30	1.92 million	5.41×10^{20}	1 in 10 million	
10	607401	1.71×10^{20}	1 in 100 million	Odds of dying in a shark attack 1 in 300 million
	192077	5.41×10^{19}	1 in a billion	
	60740	1.71×10^{19}	1 in 10 billion	
	19208	5.41×10^{18}	1 in 100 billion	
	6074	1.71×10^{18}	1 in a trillion	
	1921	5.41×10^{17}	1 in 10 trillion	Odds of a meteor landing on your house 1 in 182 trillion
	608	1.71×10^{17}	1 in 100 trillion	
	193	5.41×10^{16}	1 in 10^{15}	
	61	1.71×10^{16}	1 in 10^{16}	
	20	5.41×10^{15}	1 in 10^{17}	
	7	1.71×10^{15}	1 in 10^{18}	

Distributed Hash Table (DHT)

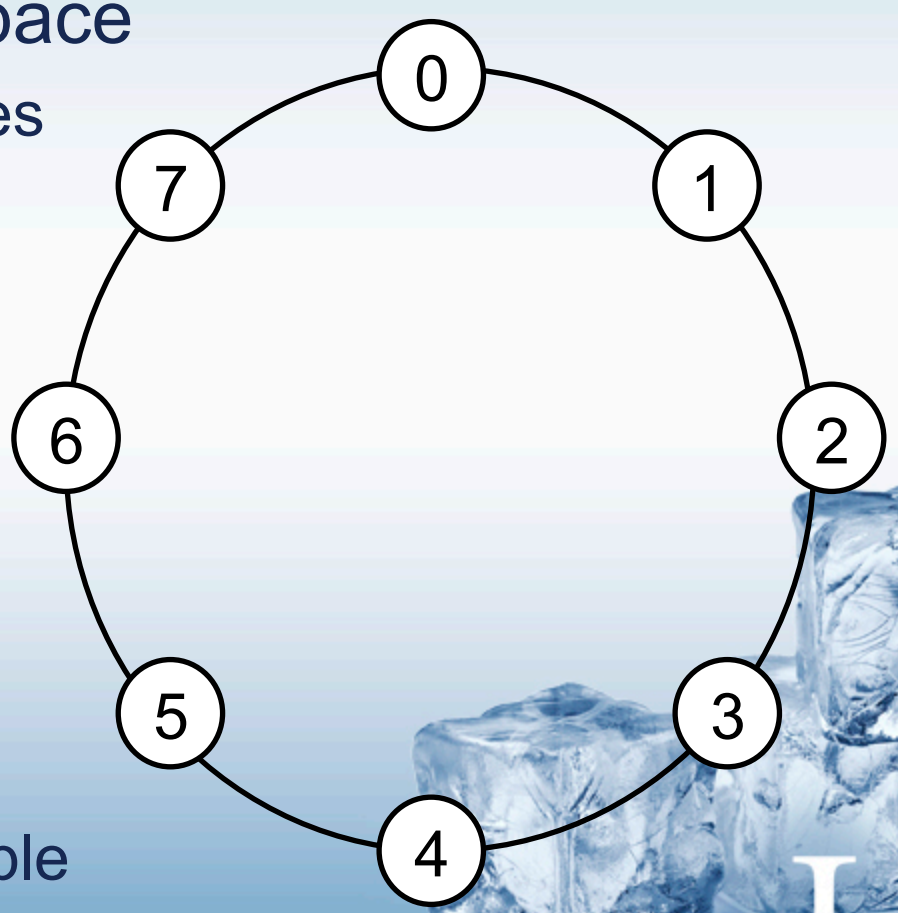
- Typical lookup strategies
  - Direct routing: Cassandra
    - Requires that all nodes know the identity of all other nodes
  - Skip-list like routing: Chord
  - Multiple Dimensions routing: CAN
  - Tree like routing: Kademlia
- DHTs contacts only O(log(n)) nodes during lookup out of a total of n nodes in the system

Consistent hashing

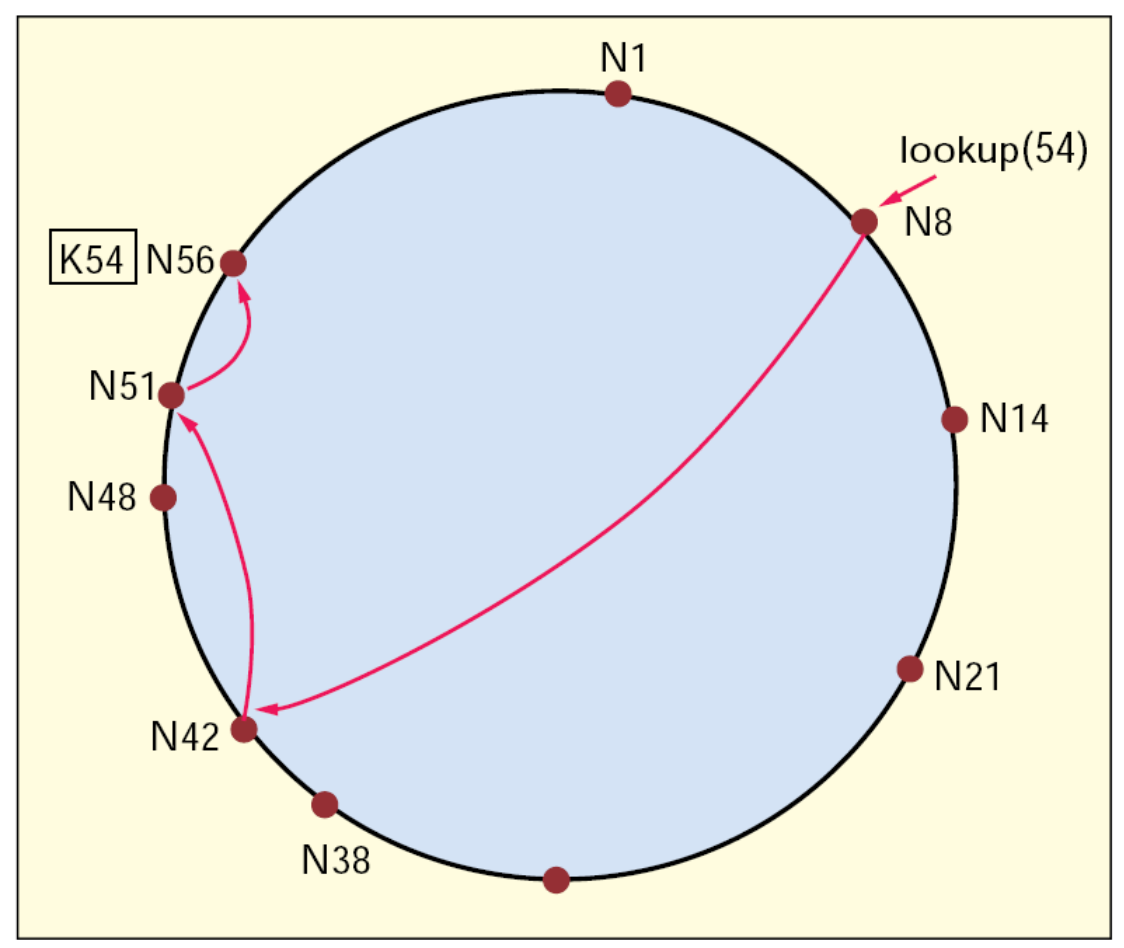
- Hash function assigns each node *and* key an m-bit *identifier* using a base hash function such as SHA-1
  - ID(node) = hash(IP, Port)
  - ID(key) = hash(key)
- Properties of consistent hashing
  - Function balances load: all nodes receive roughly the same number of keys – good?
  - When an Nth node joins (or leaves) the network, only an O(1/N) fraction of the keys are moved to a different location

DHT based on consistent hashing

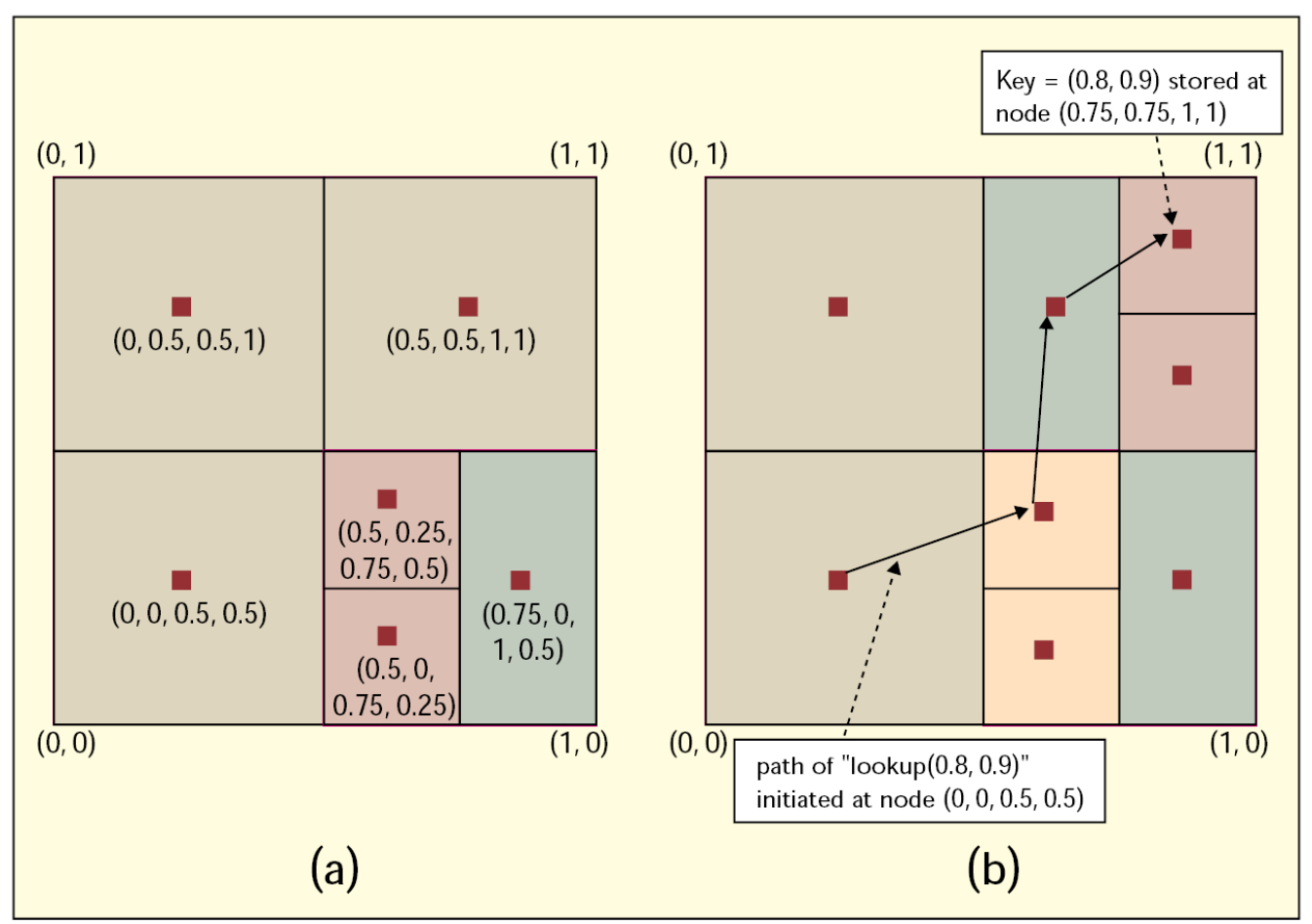
- Let assume 3 bit keyspace
  - Thus, max 8 hash values and nodes
- Organize all node in a ring and assign ID from keyspace
- The ID determines if a node is responsible for a hash key
  - E.g. node 6 is responsible for hash key 6



Chord



CAN



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

Source: Petar Mayamounkov

David Mazières

https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

Adapted by: Olov Schelén

Kademlia properties

- Each node ID and each data key corresponds to an SHA-1 hash into a 160 bits space.
  - The number of nodes is very small compared to the available key space
- Every node maintains information about keys at different levels of “closeness to itself”.
- The closeness (i.e., distance D) between two objects is measured as their bitwise XOR interpreted as an integer.
- D(a, b) = a XOR b

The XOR distance d is well defined

- the distance between a node and itself is zero
- it is symmetric: the "distances" calculated from A to B and from B to A are the same
- it follows the triangle inequality: given A, B and C are vertices (points) of a triangle, then the distance from A to B is shorter than (or equal to) the sum of the distance from A to C and the distance from C to B.

Properties of d formally defined

- $d(x,x) = 0$
- $d(x,y) > 0$ if $x \neq y$
- $d(x,y) = d(y,x)$
- $d(x,y) + d(y,z) \geq d(x, z)$
- For each x and t, there is exactly one value y for which $d(x,y) = t$

Note

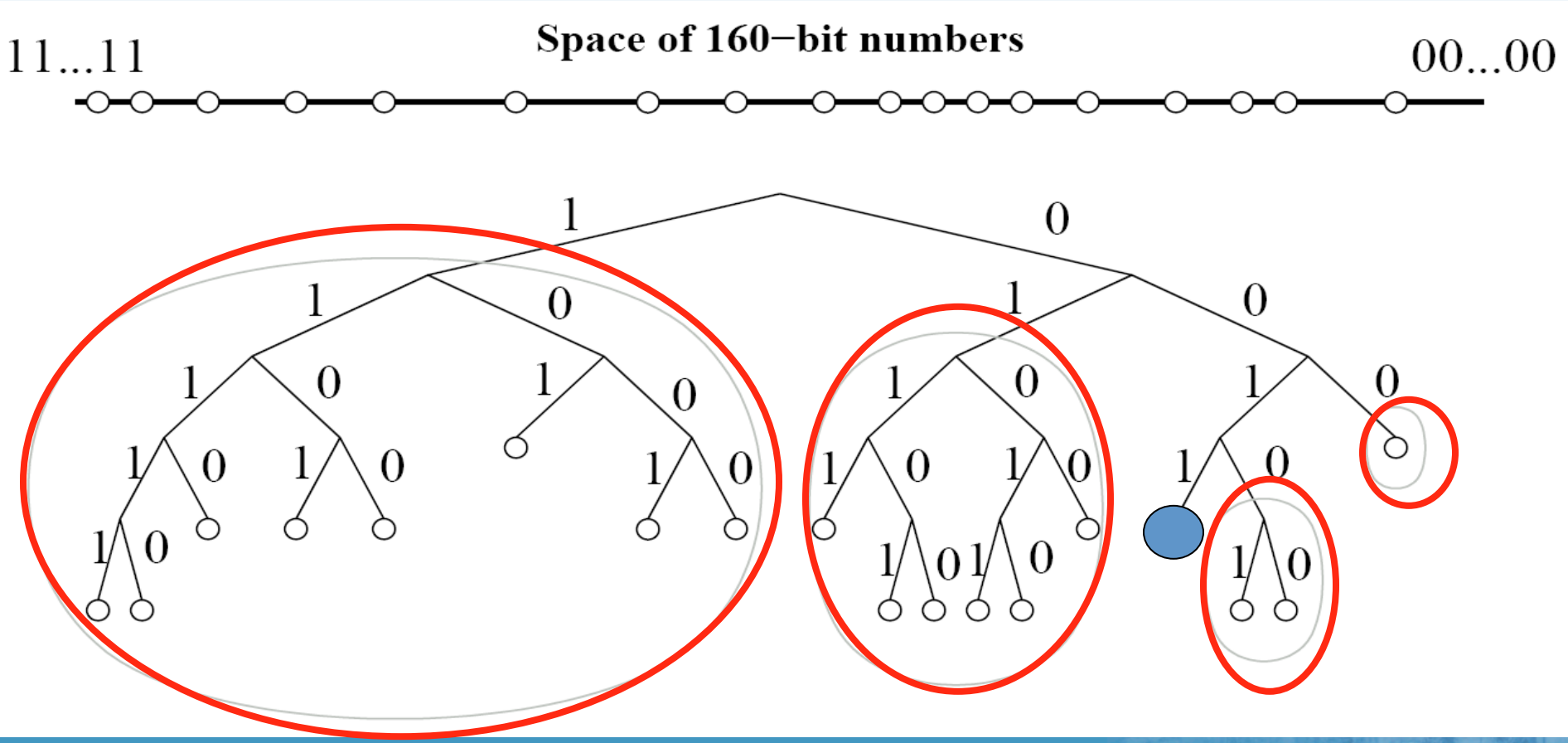
- The distance is logical and is not related to geographical closeness
- The node IDs are quite randomly spread (e.g., by hash)
  - Nodes are very few compared to the hash space
  - i.e., no central planning for assigning node IDs
    - although it would be possible to assign IDs by plan so that they are better spread over the key space

Kademlia Binary Tree

- Treat nodes as leaves of a binary tree.
- A tree of Start from root, for any given node, dividing the binary tree into a series of successively lower subtrees that don't contain the node.

Kademlia Binary Tree

Subtrees of interest for a node 0011……



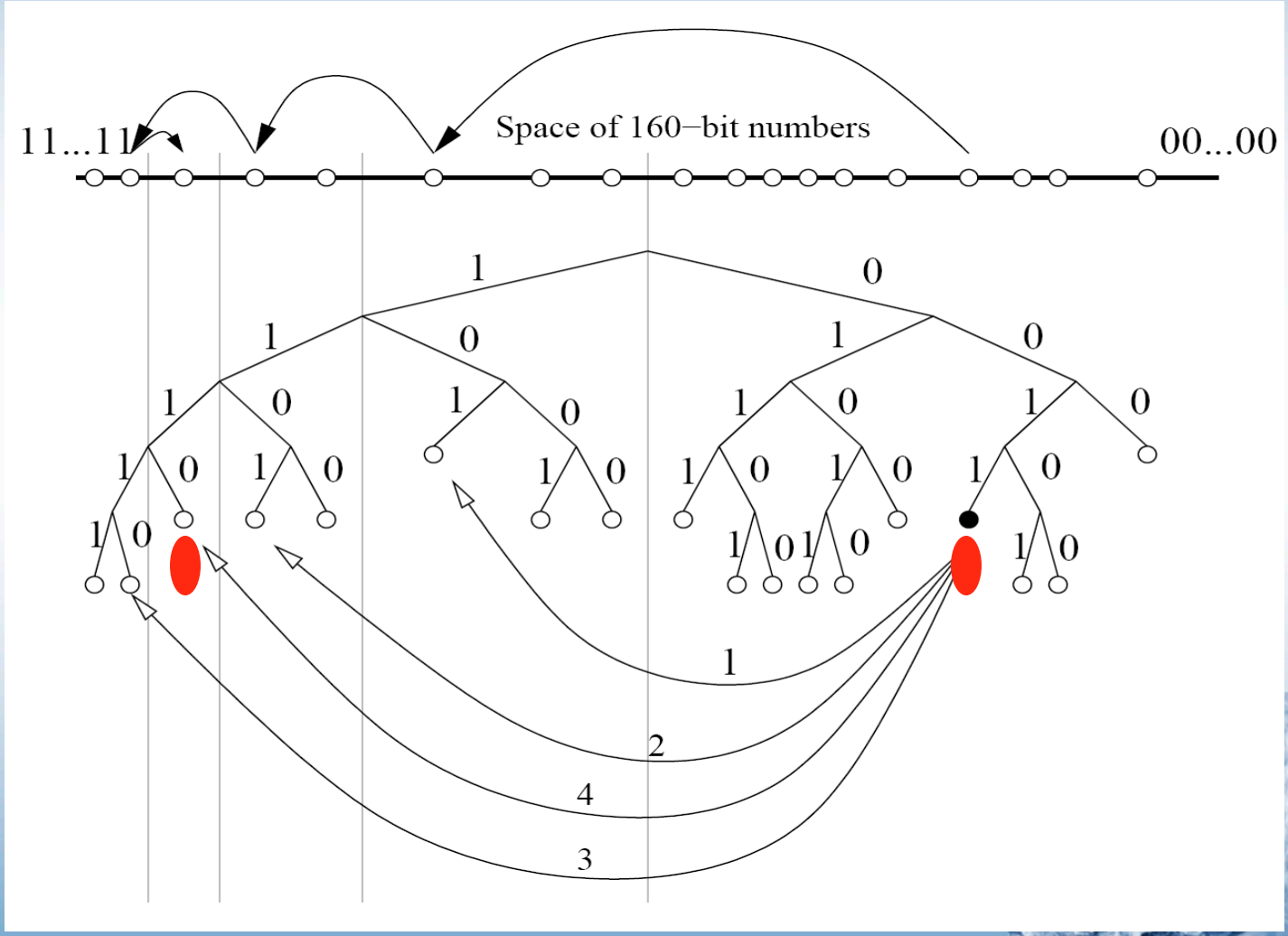
Principle

- The same hash space is used for both nodes (i.e., node IDs) and data
  - Relatively few of the leaves will correspond to nodes
  - When the number of data items is super large, there may be leaves that correspond to many data items, but that's ok
- Data that hashes to a certain value will be stored at the node(s) that are "closest" in the tree
- But, still the number of nodes is dynamic and we do not want to require that all nodes know about all other nodes
  - Instead we like each node to know about a few nodes in other subtrees
  - So when we do not know exactly, we can certainly contact some node that is "closer"

Kademlia Routing tables

- The routing table in each node contains a list (k-bucket), for each neighboring subtree.
  - i.e., routing tables consist of a *list* for each bit of the node ID
  - e.g., if node ID is 128 bits long, a node will keep 128 such *lists*.
  - Each list has up to k entries. System wide, e.g., 20.
  - Every entry in a *list* holds the necessary data to locate a node (IP-address, Port, Node id).
- So, every *list* corresponds to a distance range from the node. Nodes that can go in the n^{th} *list* must have a differing n^{th} bit from the node's ID; the first n-1 bits of the candidate ID must match those of the node's ID.
  - This means that it is easy to populate the first *list* as 1/2 of the nodes in the network are far away candidates. The next *list* can use only 1/4 of the nodes in the network (one bit closer than the first), etc.

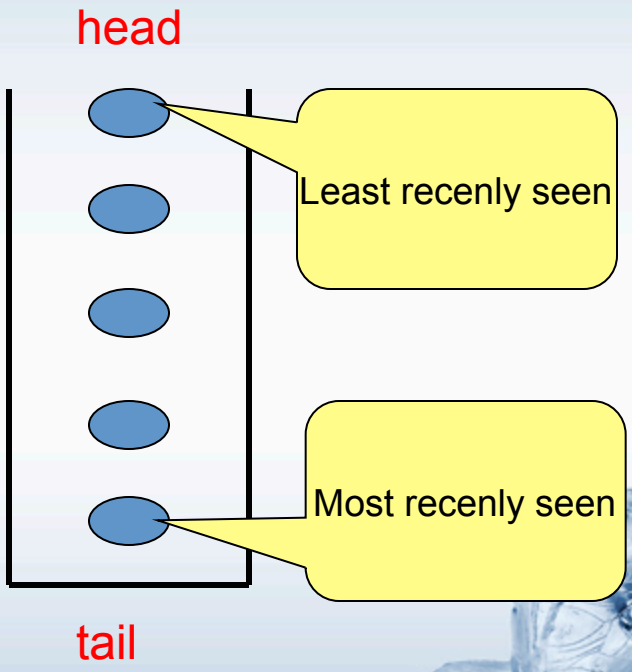
Kademlia Search



An example of lookup: node 0011 is searching for 1110……in the network

Node state

For each i ($0 \leq i < 160$) every node keeps a list of nodes of distance between 2^i and $2^{(i+1)}$ from itself. Call each list a k-bucket. The list is sorted by time last seen. The value of k is chosen so that any give set of **k nodes** is unlikely to fail within an hour. The list is updated whenever a node receives a message.



k = system-wide replication parameter (typically a small number)

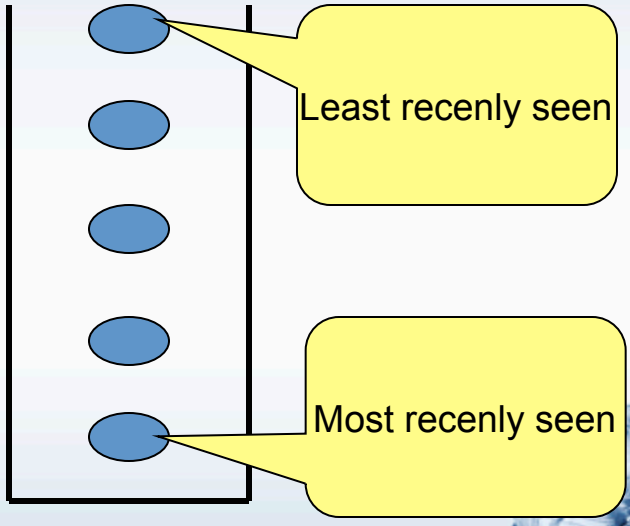
Gnutella showed that the longer a node is up, the more likely it is to remain up for one more hour

Node state

The nodes in the k-buckets are the stepping stones of routing.

By relying on the oldest nodes, k-buckets promise the probability that they will remain online.

DoS attack is prevented since the new nodes find it difficult to get into the k-bucket



How is the bucket updated?

Kademlia RPC

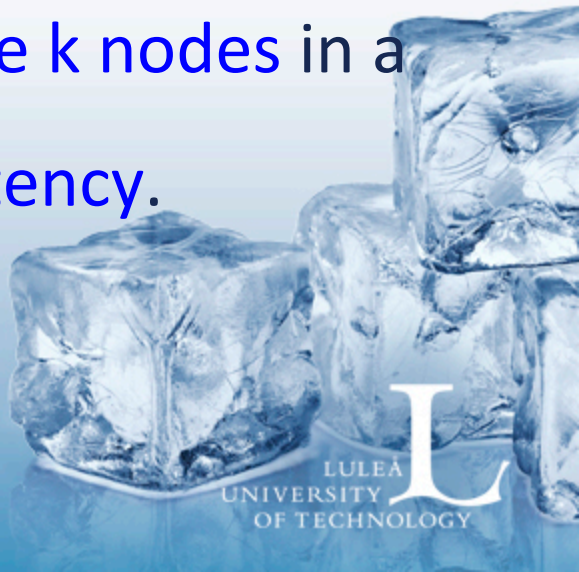
- PING: to test whether a node is online
- STORE: instruct a node to store a key
- FIND_NODE: takes an ID as an argument, a recipient returns (IP address, UDP port, node id) of the k nodes that it knows from the set of nodes closest to ID (node lookup)
- FIND_VALUE: behaves like FIND_NODE, unless the recipient received a STORE for that key, it just returns the stored value.

Kademlia Lookup

- The most important task is to locate the k closest nodes to some given node ID.
- Kademlia employs a recursive algorithm for node lookups. The lookup initiator starts by picking α nodes from its closest non-empty k-bucket.
- The initiator then sends parallel, asynchronous FIND_NODE to the α nodes it has chosen.
- α is a system-wide concurrency parameter, such as 3.

Kademlia Lookup

When $\alpha = 1$, the lookup resembles that in Chord in terms of message cost, and the latency of detecting the failed nodes. However, unlike Chord, Kademlia has the flexibility of choosing any one of the k nodes in a bucket, so it can forward with lower latency.



Kademlia Lookup

- The initiator resends the FIND_NODE to nodes it has learned about from previous RPCs.
- If a round of FIND_NODES fails to return a node any closer than the closest already seen, the initiator resends the FIND_NODE to all of the k closest nodes it has not already queried.
- The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen.

Kademlia Keys Store

- To store a (key,value) pair, a participant locates the k closest nodes to the key and sends them STORE RPCs.
- Additionally, each node re-publishes (key,value) pairs as necessary to keep them alive.
- For Kademlia's file sharing application, the original publisher of a (key,value) pair is required to republish it every 24 hours. Otherwise, (key,value) pairs expire 24 hours after publication.

New node join

- Each node bootstraps by looking for its own ID

  Search recursively until no closer nodes can be found
- The nodes passed on the way are stored in the routing table
- The nodes passed on the way may learn about the existence of the new node (if they have room in their k-bucket)

Conclusion

- Operation cost
  - As low as other popular protocols
  - Look up: O(logN), Join or leave: $O(\log^2 N)$
- Fault tolerance and concurrent change
  - Handles well via the use of k-buckets
- Proximity routing -- chooses nodes that has low latency
- Handles DoS attacks by using nodes that are up for a long time
- The architecture works with various base values. A common choice is b=5.

References

- http://en.wikipedia.org/wiki/Skiplist 20070409
- Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica. Looking up data in P2P systems. Communications of the ACM, Volume 46 , Issue 2 (February 2003), SPECIAL ISSUE: Technical and social components of peer-to-peer computing Pages: 43 – 48
- Maymounkov, P., and Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Springer-Verlag version, Cambridge, MA (Mar. 2002); kademlia.scs.cs.nyu.edu.

Other DHT algorithms and implementation

- Chord
- Apache Cassandra
- CAN (Content Addressable Networks)
- Pastry
- Tapestry

You will learn more about these during the course!

Now, back to the lab assignment ...

- The labs should be done in groups of 2-3 students. Enter your group membership to Canvas. There are n groups allocated to which you can add yourselves.
- A key ability is to search, find and re-use existing knowledge, software and other information that can bring you forwards effectively. Obviously, you cannot use software that directly solves the lab assignment. Don't forget to make references.
- In your reports you must name the people you have cooperated closely with and which persons you have discussed with (if it has had impact on your solution). Obviously you must reference all external information (web-pages, papers etc.) that you have used. The assignment is generally defined, so we expect all solutions to be different to some extent.
- The lab should preferable be implement in Golang. However, if you decide to use another language, you are own your own and get not expect as much help.
- Consist of 5 objectives + 1 optional objective

Examination

Sprint 1

- Upload to Canvas
  - sprint planning (backlog) for each objective
  - draft documentation (see lab instructions) explaining key principles for the top priority items in the backlog
- Book time for sprint review with your instructor

Sprint 2

- Upload to Canvas
  - Final version of documentation (see lab instructions) & backlog
  - Code or link to code
  - Demo outline (may have screenshots)
- Book time for final review with your instructor

Note: demo is very important, aim for small demo already at sprint 1 review.

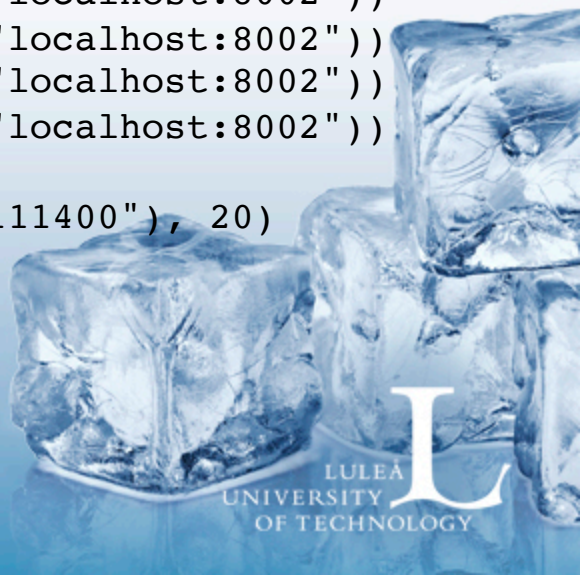
Objective 1 – Understand Kademlia

- The purpose is to get you familiar with the Kademlia algorithm.
  - Read the paper
  - Learn to Go
  - Develop a simple simulator
  - Some source code will be provided

Objective 1 - Kademlia DHT playground

- Create a routing table
- Find nearest nodes

```
rt := NewRoutingTable(NewContact(NewKademliaID("FFFFFFFF00"),"localhost:8000"))

rt.AddContact(NewContact(NewKademliaID("FFFFFFFF00"), "localhost:8001"))
rt.AddContact(NewContact(NewKademliaID("1111111100"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111200"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111300"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111400"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("2111111400"), "localhost:8002"))

contacts := rt.FindClosestContacts(NewKademliaID("2111111400"), 20)
for i := range contacts {
    fmt.Println(contacts[i].String())
}
```



Objective 2 – Fully working Kademlia DHT

- Add network support
  - Marshaling (serialization) and RPC
- Implement functions
  - *LookupContact*, *LookupData* and *Store*
- Make initial solution with minimal functionality
  - then add features and optimizations
- Make unit testing

Objective 3 – File system

$ dfs store myFile.txt
eb9d5f1f1874cfa48c5442e6172d45d307267eb9

The *dfs store* command should return the hash of the file, i.e. the address of the file in the Kademlia network.

The *cat* command should print the content of a file given the address as argument, e.g.
$ dfs cat eb9d5f1f1874cfa48c5442e6172d45d307267eb9
… content of my file …

The *pin* command should make sure important data is not deleted.
$ dfs pin eb9d5f1f1874cfa48c5442e6172d45d307267eb9

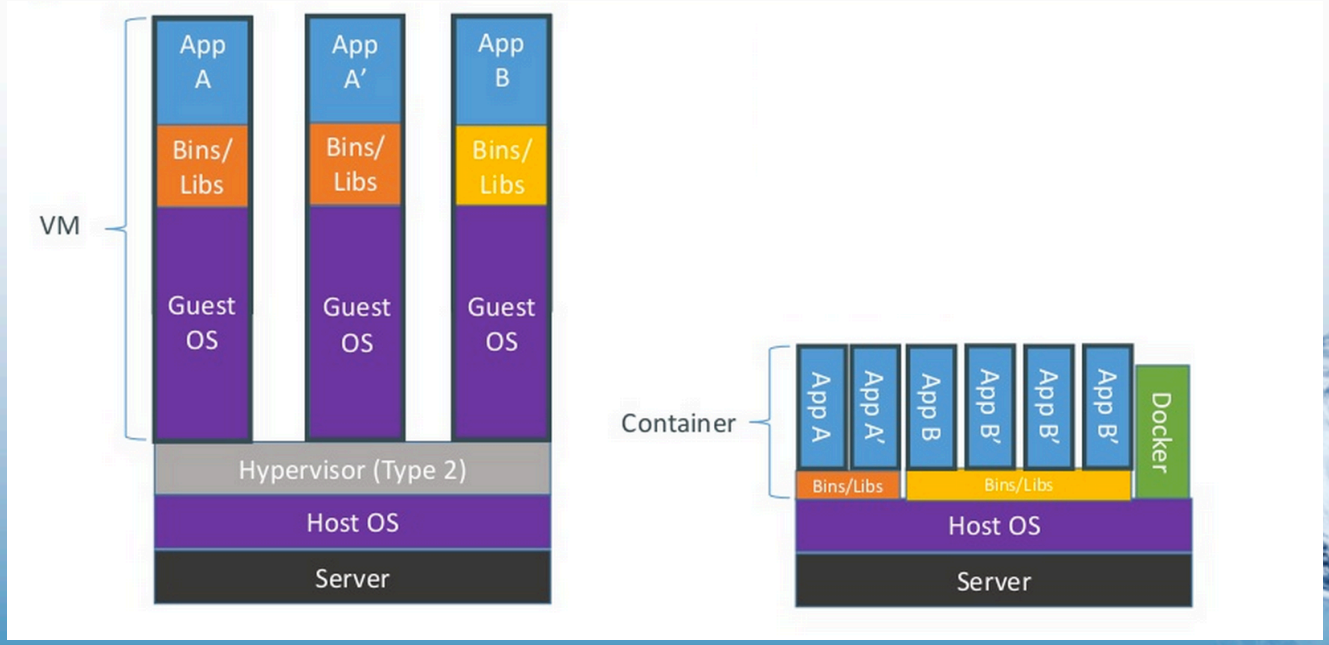
Finally, the *unpin* command should remove pinned data.
$ dfs unpin eb9d5f1f1874cfa48c5442e6172d45d307267eb9

Virtualization

- Learning virtualization technologies
  - Docker (possibly virtualbox and vagrant)
- Each node should run in a Docker container on a VM (Virtual Machine)
  - Other outlines are ok if you prefer
- Consider ansible for orchestration
  - But not required

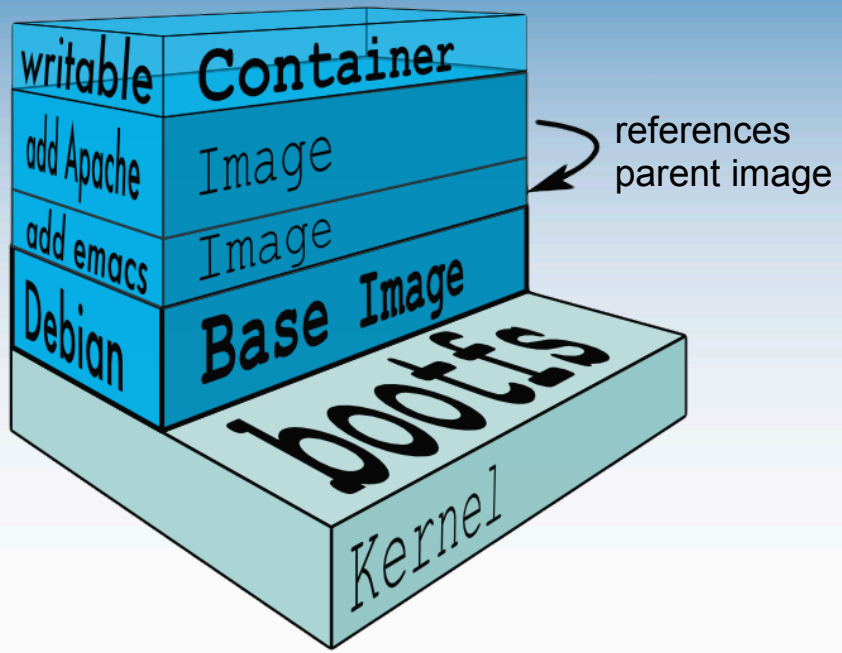
Docker

- Container = Lightweight Linux Environment
- Open source project to manage containers
  - Makes it possible package application along with dependencies such as binaries and libraries



DockerFile

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y xvfb
RUN apt-get install -y firefox
RUN apt-get install -y nodejs
RUN apt-get install -y npm
RUN apt-get install -y mercurial
RUN apt-get install -y git
RUN apt-get install -y golang

RUN npm config set registry http://registry.npmjs.org/
RUN npm install connect
RUN npm install serve-static
RUN ln -s /usr/bin/nodejs /usr/bin/node

ENV GOPATH /greenhouse
RUN go get github.com/nu7hatch/gouuid
RUN go get code.google.com/p/go.net/websocket

ADD conf/supernode.conf /.bitverse/supernode.conf
ADD . /greenhouse/src/bitverse

CMD sh -c "cd /greenhouse/src/bitverse/systemtests; xvfb-
run ./setmap_system_test.sh"
```



> docker build –t myTest .

> docker run myTest

> docker push myTest …

> docker pull myTest …

Why use Golang?

- Go compiles very quickly
- Go supports concurrency at the language level
- Functions are first class objects in Go
- Go has garbage collection
- Strings and maps are built into the language
- Go is very strongly typed
- Go is not object oriented in the traditional sense

Some Golang highlights

- Classes/methods
  - http://tour.golang.org/#52
- Goroutines
  - http://tour.golang.org/#65
- Channels
  - http://tour.golang.org/#66
- Full tutorial at
  - http://tour.golang.org/#1

Go testing

- Go has built in testing framework
  - http://golang.org/pkg/testing/

```
package math

import "testing"

func TestAverage(t *testing.T) {
    var v float64
    v = Average([]float64{1,2})
    if v != 1.5 {
        t.Error("Expected 1.5, got ", v)
    }
}
```

Now run this command:

```
go test
```