DHT: Distributed Hash Table

Day 20

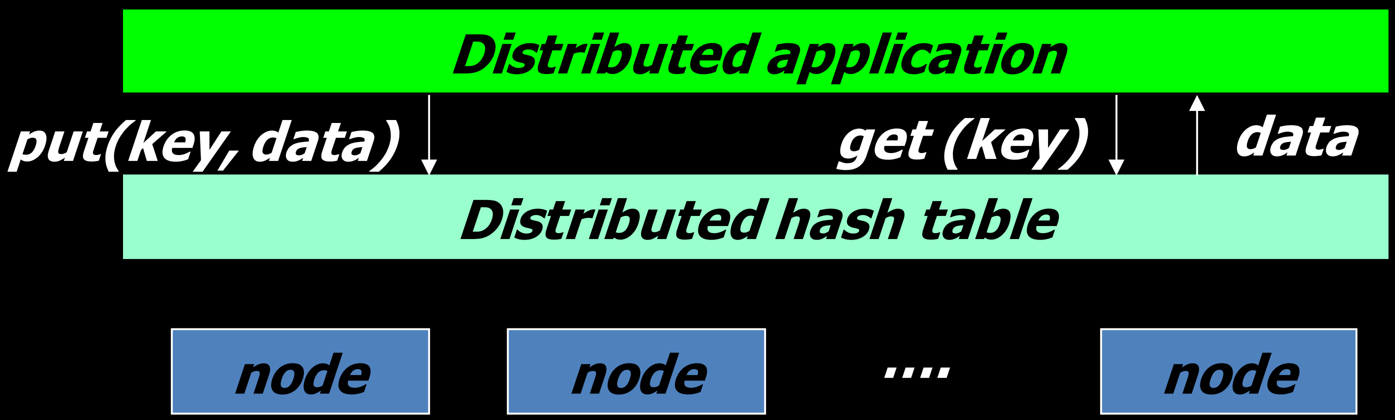
Applications

- Anything that requires a hash table
- Databases, FSes, storage, archival
- Web serving, caching
- Content distribution
- Query & indexing
- Naming systems
- Communication primitives
- Chat services
- Application-layer multi-casting
- Event notification services
- Publish/subscribe systems ?

Definition of a DHT

- Hash table → supports two operations
  - `insert(key, value)`
  - `value = lookup(key)`
- Distributed
  - Map hash-buckets to nodes
- Requirements
  - Uniform distribution of buckets
  - Cost of `insert` and `lookup` should *scale* well
  - Amount of local state (routing table size) should *scale* well

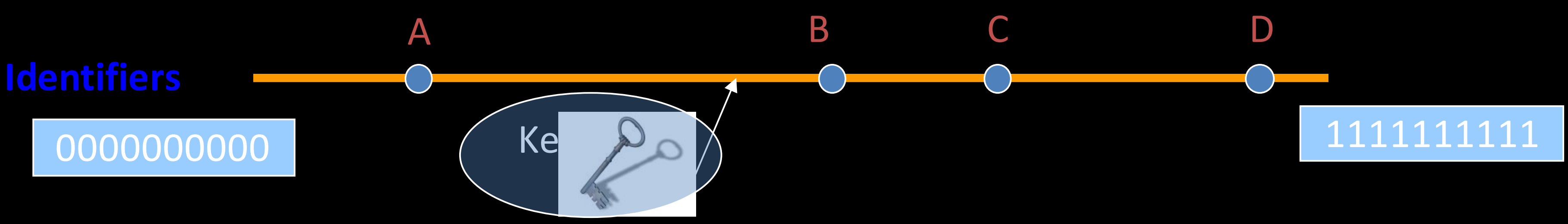
What is DHT?



(Figure adopted from Frans Kaashoek)

Fundamental Design Idea - I

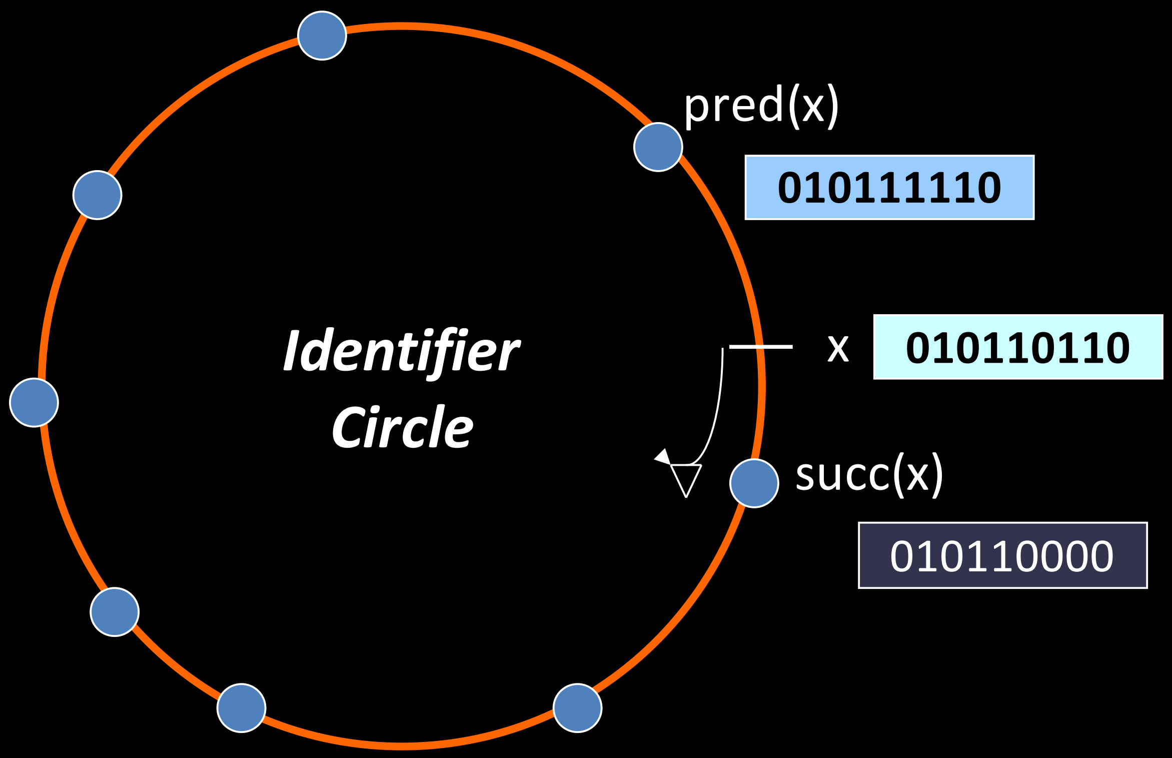
- Consistent Hashing
  - Map keys *and* nodes to an *identifier* space; implicit assignment of responsibility



- Mapping performed using hash functions (e.g., SHA-1)
  - Spread nodes and keys *uniformly* throughout

Chord [Karger, et al]

- Map nodes and keys to identifiers
  - Using randomizing hash functions
- Arrange them on a circle



Look-Up Performance V. Scalability

- Alternatives:
  - O(N) → Each node stores only successor
    - Look-ups are expensive but scales really well
  - O(1) → Each nodes store information for all nodes
    - Look-ups are really fast/cheap but does not scale

Performance -- Lookup

Purpose -- to locate a target node

- Each step, try to get closer to locating target node
    - Ask a closer neighbour
- Performance & scalability tied directly to lookup algorithm

2 Aspects to Performance

- Path latency
- Lookup path length (# hops)

2 Aspects to Scalability

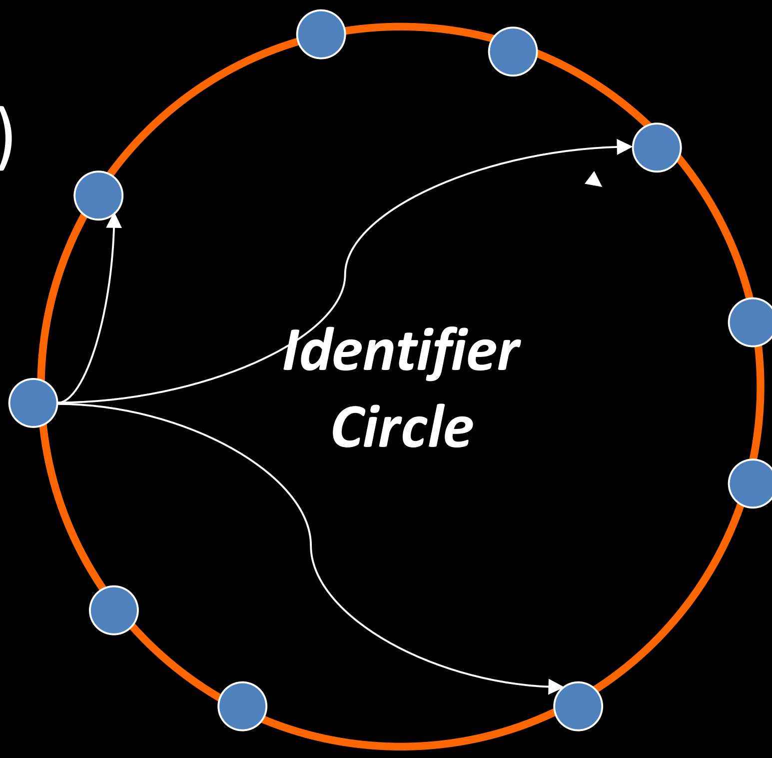
- size of routing table – O(log N)
- lookup path length – O(log N)

3 Techniques

- proximity lookup
- proximity neighbour selection
- geographic layout

Chord
Efficient routing

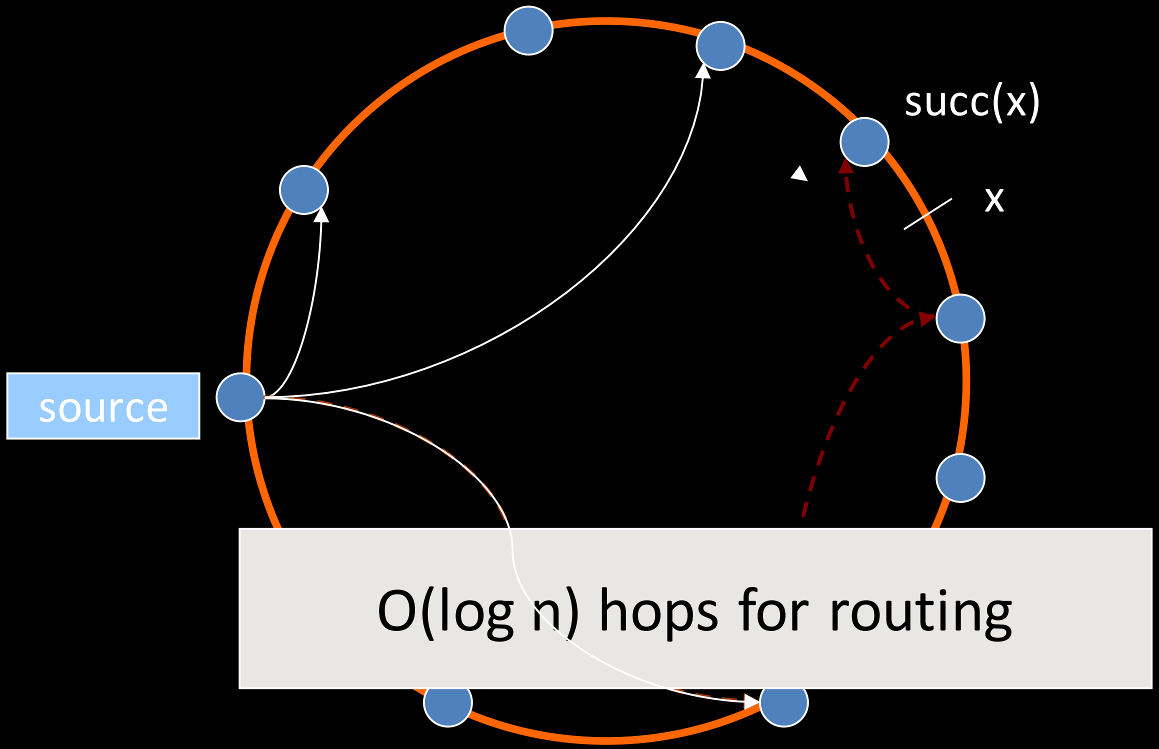
- Routing table
  - log(n) *finger pointers*
  - i^{th} entry = succ(n + 2^i)



Exponentially spaced pointers!

Chord
Key Insertion and Lookup

To insert or lookup a key ‘x’,
route to succ(x)

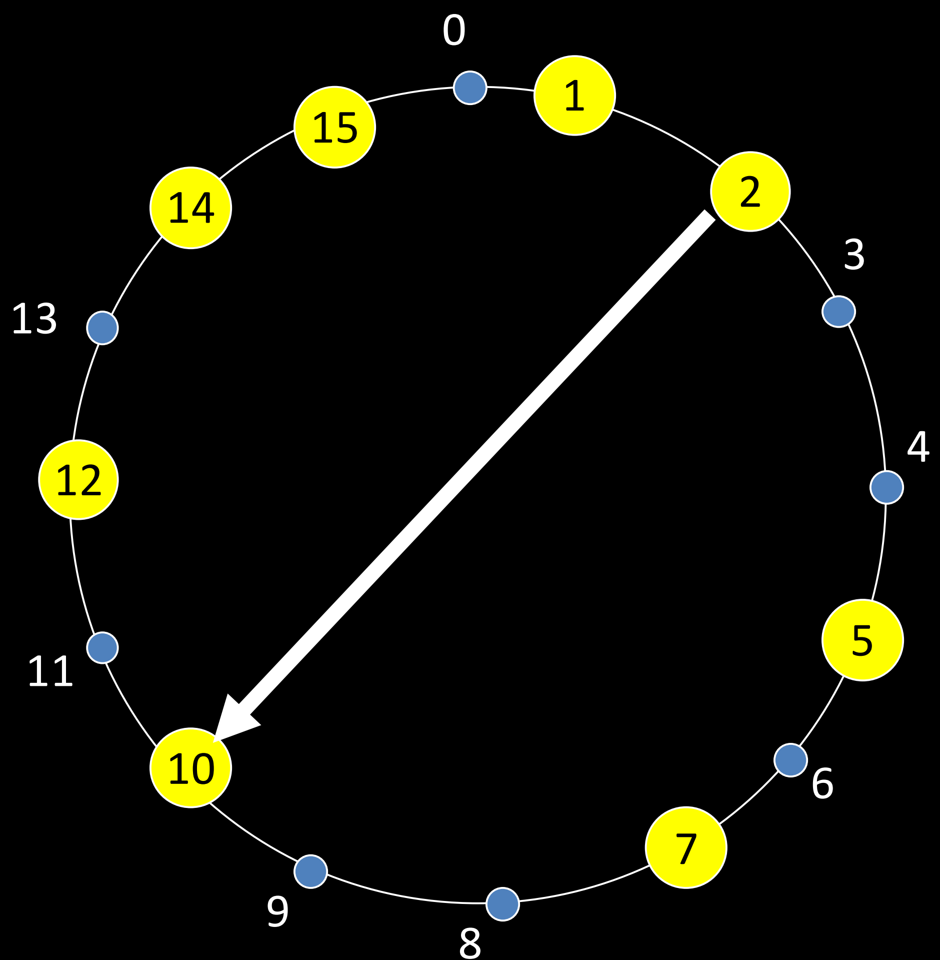


How lookup works? Try to find ‘0’

Example: Chord [Stoica et. al.]

Finger Table for Node 2

start	interval	succ.
3	[3,4)	5
4	[4,6)	5
6	[6,10)	7
10	**[10,2)**	**10**

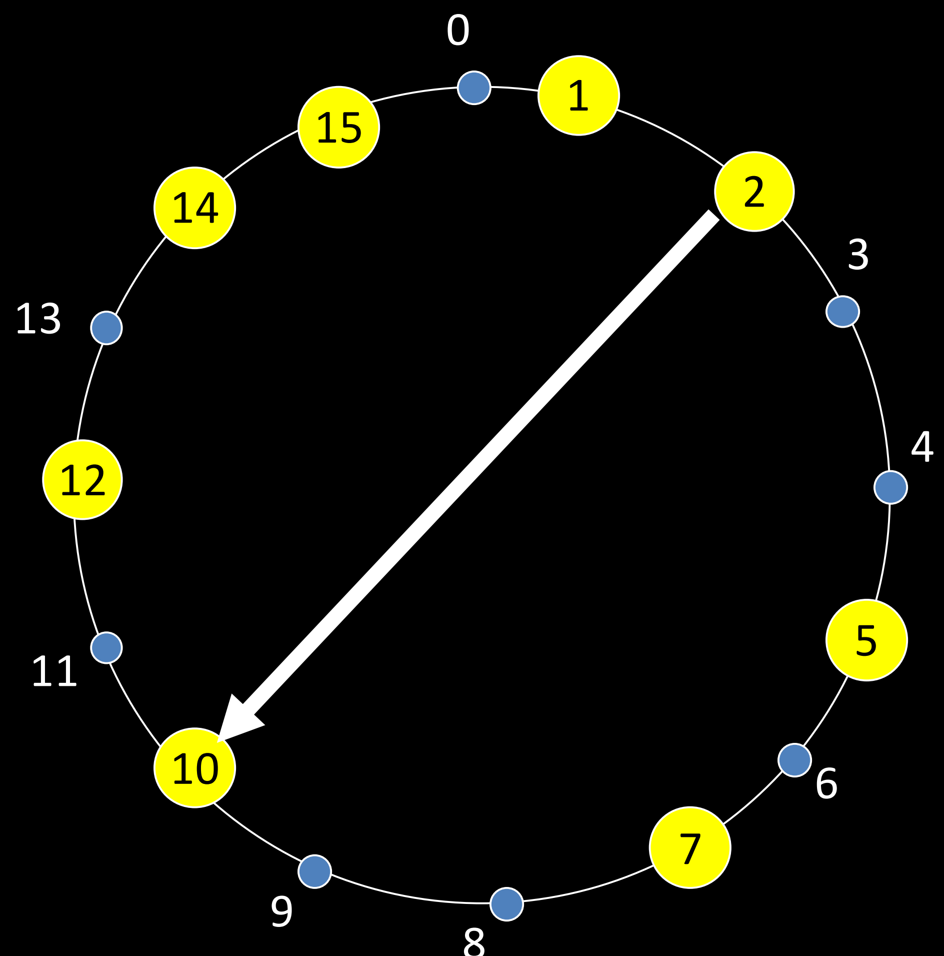


How lookup works? Try to find '0'

Example: Chord

Finger Table for Node 10

start	interval	succ.
11	[11,12)	12
12	[12,14)	12
14	**[14,2)**	**14**
2	[2,10)	2

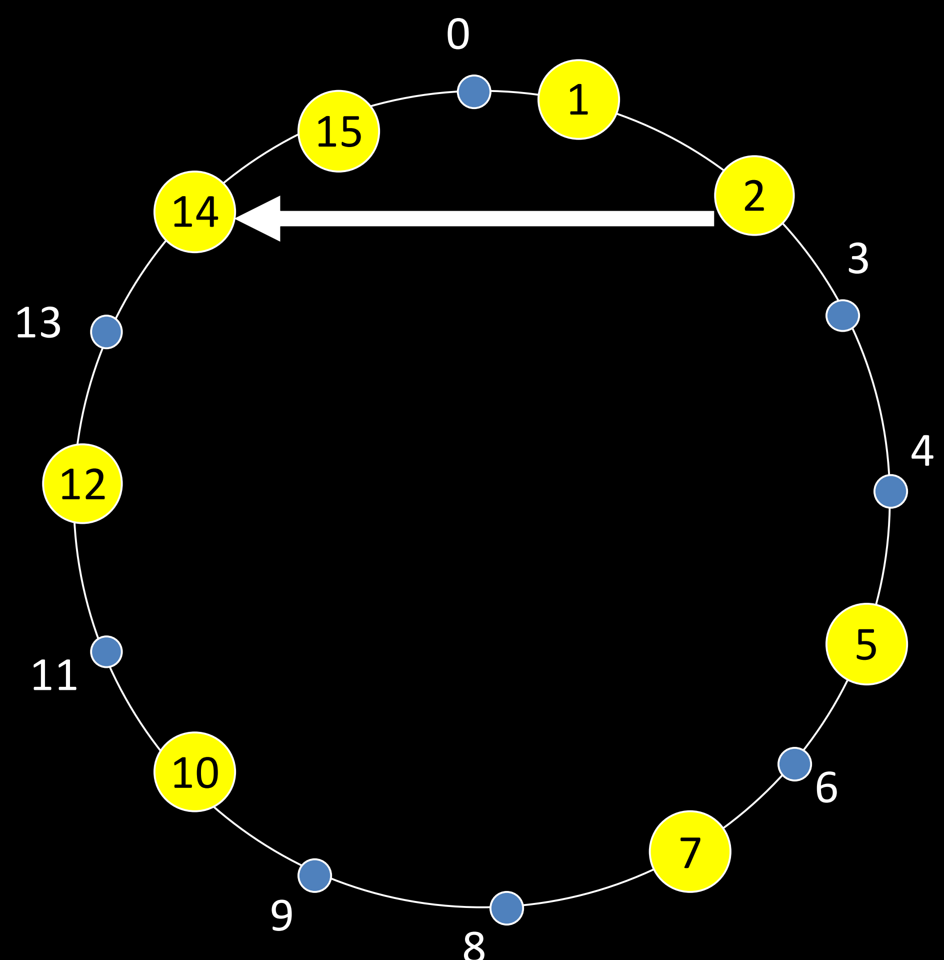


How lookup works? Try to find ‘0’

Example: Chord

Finger Table for Node 10

start	interval	succ.
11	[11,12)	12
12	[12,14)	12
14	**[14,2)**	**14**
2	[2,10)	2

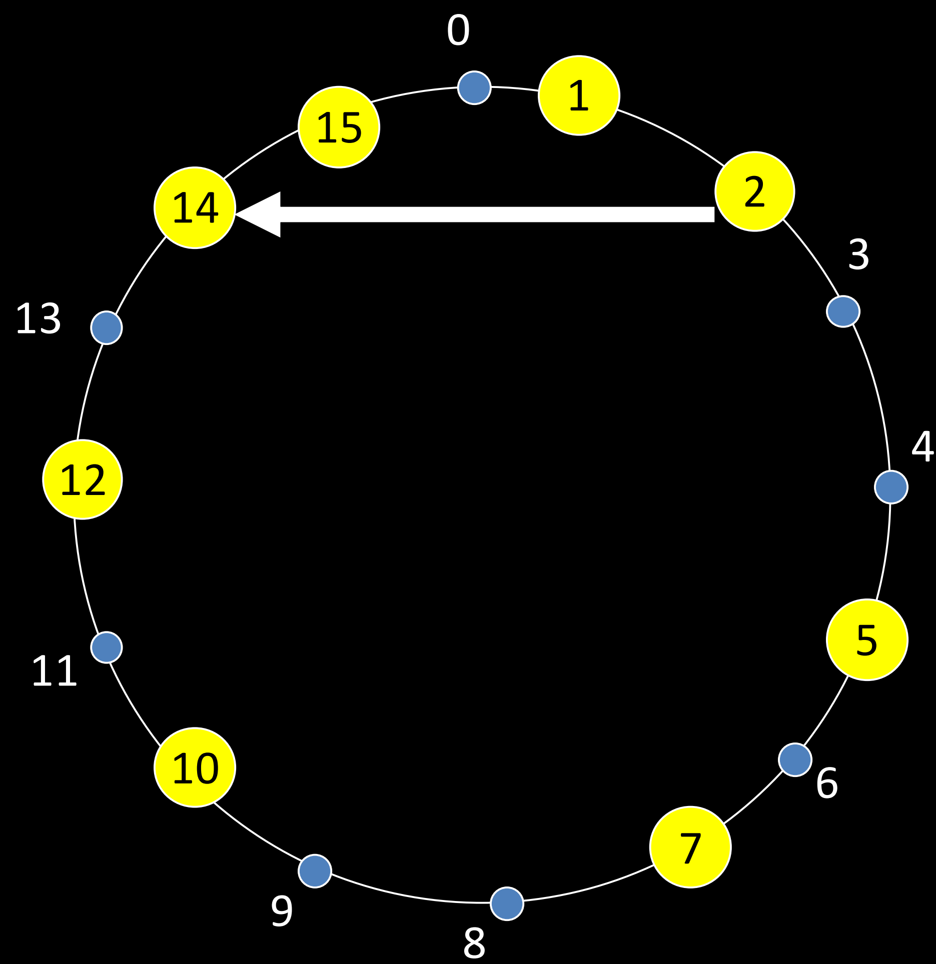


How lookup works? Try to find '0'

Example: Chord

Finger Table for Node 14

start	interval	succ.
15	[15,0)	15
0	**[0,2)**	**1**
2	[2,6)	2
6	[6,13)	7

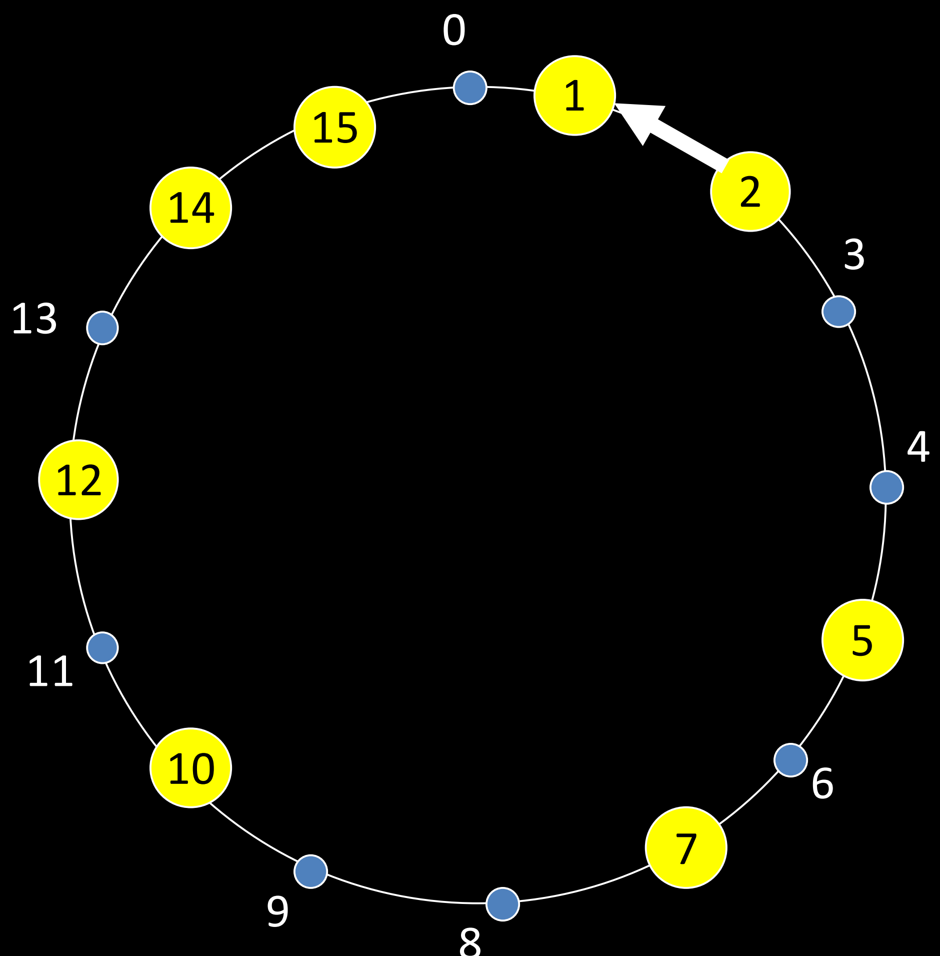


How lookup works? Try to find '0'

Example: Chord

Finger Table for Node 14

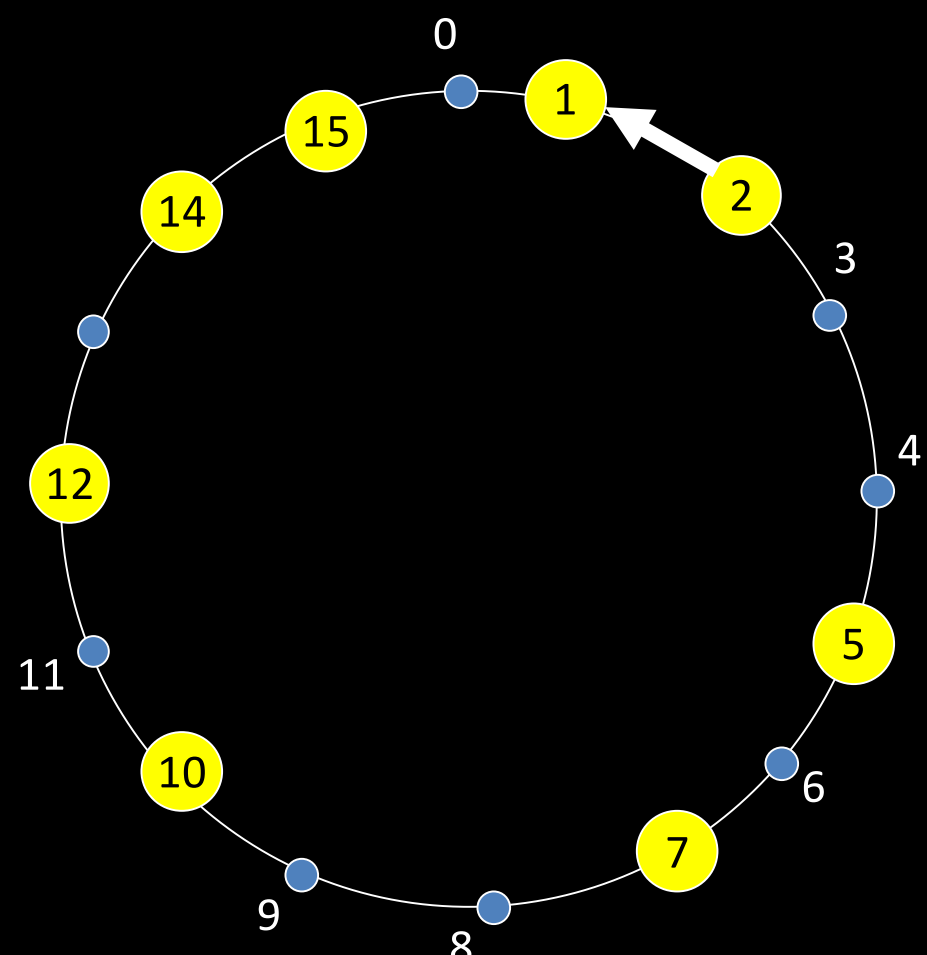
start	interval	succ.
15	[15,0)	15
0	**[0,2)**	**1**
2	[2,6)	2
6	[6,13)	7



How lookup works? Try to find ‘0’

Example: Chord

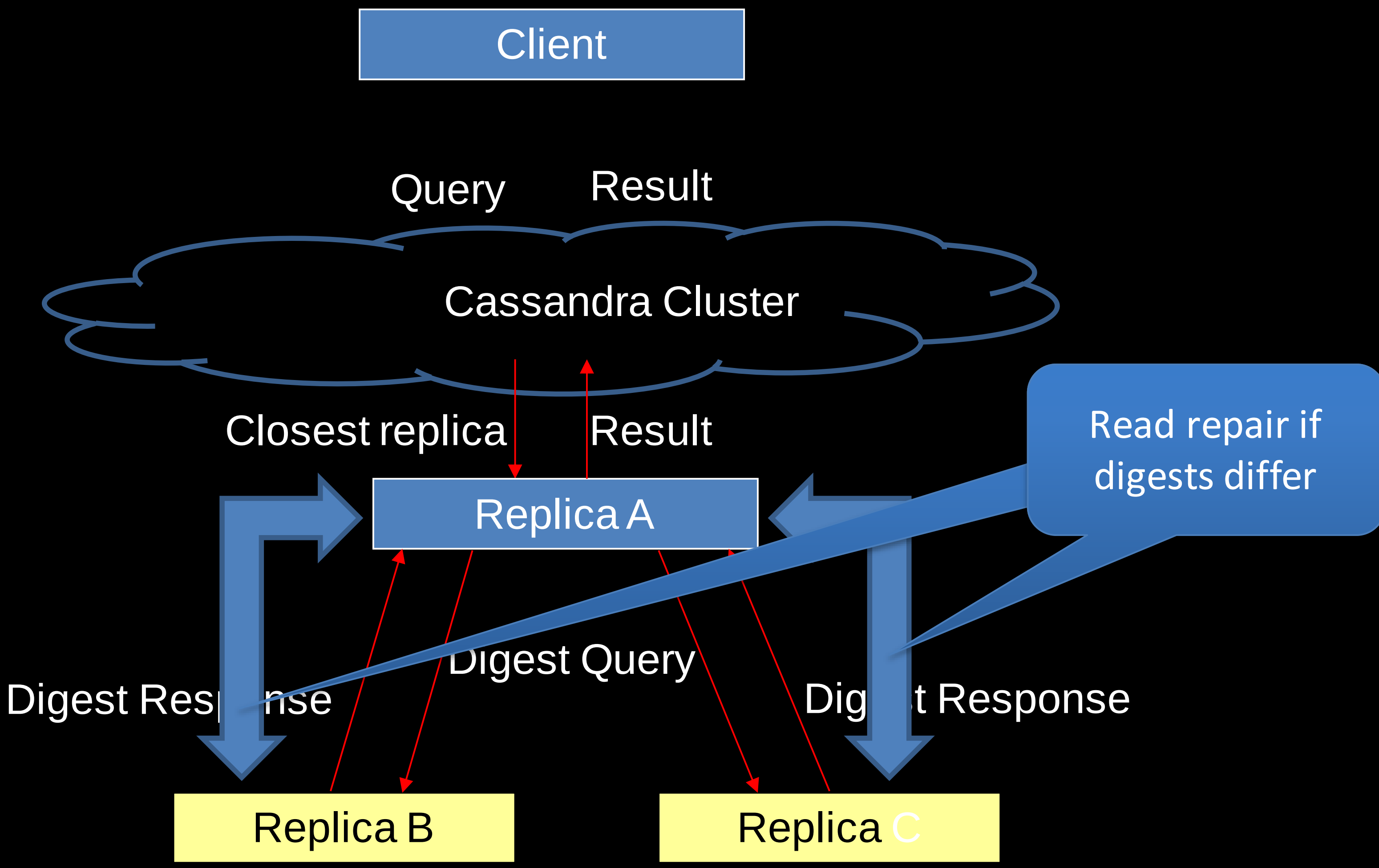
Now Node 2 can retrive information for key 0 from Node 1.



Chord
Self-organization

- Node join
  - Set up finger *i:* route to $succ(n + 2^i)$
  - log(n) fingers) $O(\log^2 n)$ cost
- Node leave
  - Maintain successor list for ring connectivity
  - Update successor list and finger pointers

Read Operation



* Figure taken from Avinash Lakshman and Prashant Malik (authors of the paper) slides.

FB's Cassandra

System Architecture

- **Partitioning: provides high throughput**

  How data is partitioned across nodes?

  What do we want from a good partition algorithm?

High Throughput

- Use a DHT like Chord

System Architecture

- **Partitioning: provides high throughput**

  How data is partitioned across nodes?

  What do we want from a good partition algorithm?

- **Replication: overcome failure**
  - How data is duplicated across nodes? Challenges:
    - Consistency issues
    - Overhead of replication

Replication

- **Each data item is replicated at N (replication factor) nodes.**

- **Different Replication Policies**
  - **Rack Unaware –** replicate data at N-1 successive nodes after its coordinator
  - **Rack Aware –** uses 'Zookeeper' to choose a leader which tells nodes the range they are replicas for
  - **Datacenter Aware –** similar to Rack Aware but leader is chosen at Datacenter level instead of Rack level.

- Why??

Local Persistence

- **Relies on local file system for data persistency.**
- **Write operations happens in 2 steps**
  - **Write to commit log in local disk of the node**
  - **Update in-memory data structure.**
- **Read operation**
  - **Looks up in-memory ds first before looking up files on disk.**
  - **Uses Bloom Filter (summarization of keys in file store in memory) to avoid looking up files that do not contain the key.**

Failure Detection

- Traditional approach
  - Heart-beats (Used by HDFS & Hadoop): binary (yes/no)
  - If you don't get X number of heart beats then assume failure
- Accrual failure approach
  - Returns a # representing probability of death
    - X of the last Y messages were received: (X/Y)*100%
  - Modify this # to reflect N/W congestion & server load
  - Based on the distribution of inter-arrival times of update messages
    - How would you do this?

Issues with DHT

Issues with DHT

- DHT distributes keys evenly but …
  - Some keys are more popular than others
  - Some keys have geographical properties
  - How do you deal with tail latency?

Are DHTs a panacea?

- Useful primitive
- Tension between network efficient construction and uniform key-value distribution
- Does every non-distributed application use only hash tables?
  - Many rich data structures which cannot be built on top of hash tables alone
  - Exact match lookups are not enough
  - Does any P2P file-sharing system use a DHT?

How can you build a MySQL atop DHT