Finding Data in the Cloud using Distributed Hash Tables (Chord)

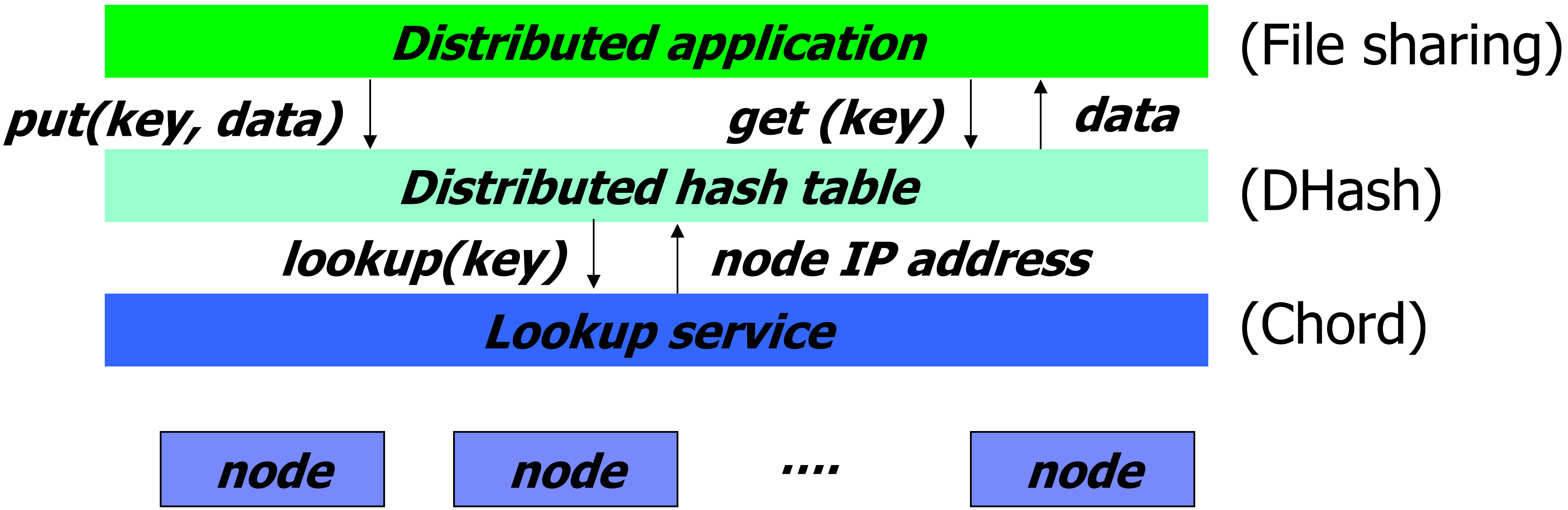
IBM Haifa Research

Storage Systems

In File Systems

- ***The App needs to know the path***
  - /home/user/my pictures/…
- ***The Filesystem looks up the directory structure to find the file’s inode, which reveals the physical location of the file’s data on disk.***
- ***At Cloud Scale the file system must get distributed***
  - Millions of different users uploading / downloading files
  - Billions of files
- ***This brings challenges that traditional file systems are not built for***
  - File system metadata is usually a single point of failure
  - File systems must adhere POSIX semantics requiring strong consistency of the file system metadata
- ***Something else is required here….***

The Abstraction: Distributed hash table (DHT)



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

Picture taken from the lecture slides "***Topics in Data-Intensive Computing Systems***", given at UBC (***http://www.ece.ubc.ca/~matei/EECE571.06/***)

The Requirements (Chord)

- ***Scalability. Billions of keys stored on hundreds or millions of nodes.***
  - Operations cannot take time that is substantially larger than logarithmic in the number of keys
- ***Availability.***
  - The lookup service must be able to function despite network partitions and node failures. Chord provides 'best effort' availability keeping 'r' replicas of each key.
  - Note: In the presence of many joins/failures data may be temporarily unavailable until the network stabilizes.
- ***Load Balancing.***
  - Resource usage is evenly distributed among the machines in the system. In a system with N nodes and K keys, each node holds approximately K/N of the keys.
- ***Dynamism.***
  - Nodes can join or leave the system without any downtime
- ***NOT Covered, but interesting in some cases:***
  - Authenticated Inserts / Access control.
  - Protection against malicious servers.
  - Stronger Consistency.

The API (Chord)

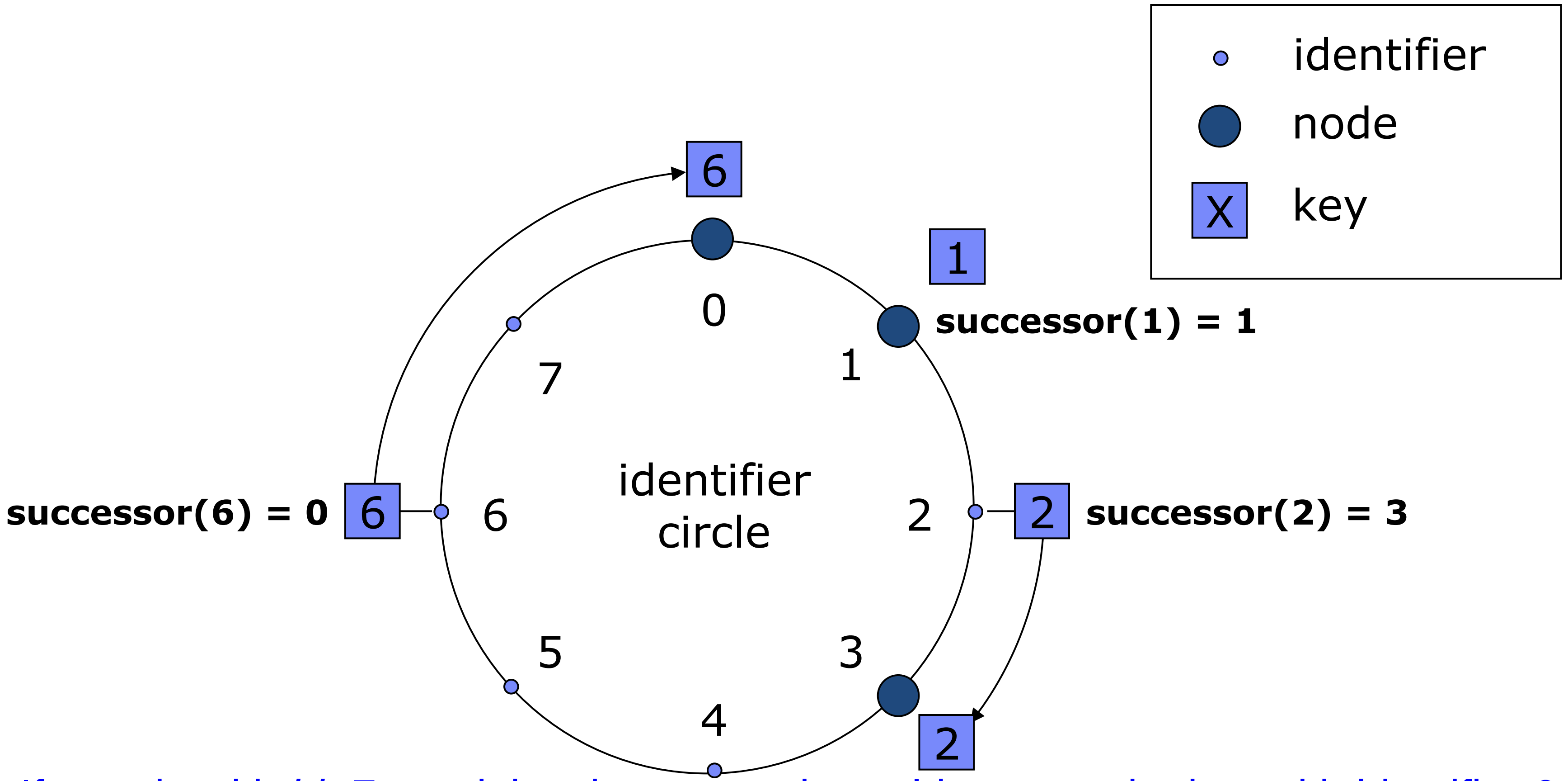
- ***Insert (key, value) – Inserts key/value at r distinct nodes***
- ***Lookup (key) – Returns the value associated with the key***
- ***Update(key, newval)***
- ***Join(n) – Causes a node to add itself as a Chord node, where n is an existing node***
- ***Leave() – Leave the Chord system***

The Chord Structure *(Based on Consistent Hashing)*

- ***Choose a good hash function mapping to some m bit domain. Defining a 2^m space.***
  - A popular choice is SHA1 mapping to 160 bit strings, defining a space of size 2^{160}
- ***Assign each key and node in the system an 'm bit identifier'***
  - For each node apply SHA1 on the node's IP
  - For each human/application readable key apply SHA1 on that key
- ***Given the identifiers, each key,* k*, is stored in a node whose identifier,* id*, is equal to or follows* k*, in the identifier space.***

The Chord Structure with m=3

Peer-to-Peer Systems, Anthony D. Joseph, John Kubiatowicz, Berkely



If a node with *id=7* now joins the system it would capture the key with identifier 6.

Picture taken from a presentation by Anthony D. Joseph and John Kubiatowicz (Berkely) on peer-to-peer systems (***http://www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/***)

The Fundamental Properties of Consistent Hashing

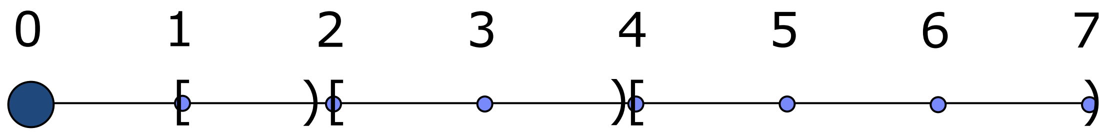
- ***For any set of N nodes and K keys, with high probability:***
  - Each machine is responsible for at most (1+ε)K/N keys
  - When an (N+1)st machine joins or leaves the network, O(K/N) keys are moved (and only to or from the joining or leaving machine)
- ***In practice a good hash function such as SHA1 should be sufficient to achieve the above***
- ***To achieve small ε, each physical node actually needs to run log N 'virtual' nodes each with independent identifier over the ring.***

Routing in Chord (how do I get to the value of K=6)

- ***Holding a complete table of the nodes and their ids should do the trick in O(1) time, but:***
  - This is a large table to hold.
  - This is a constantly changing table.
- ***Alternatively, let each node know only about its successor.***
  - A search can take up to O(N) messages.
- ***Turns out that holding a table of m entries (3 in our example, 160 in the SHA1 case) on each node suffices to locate any key with O(log N) messages.***

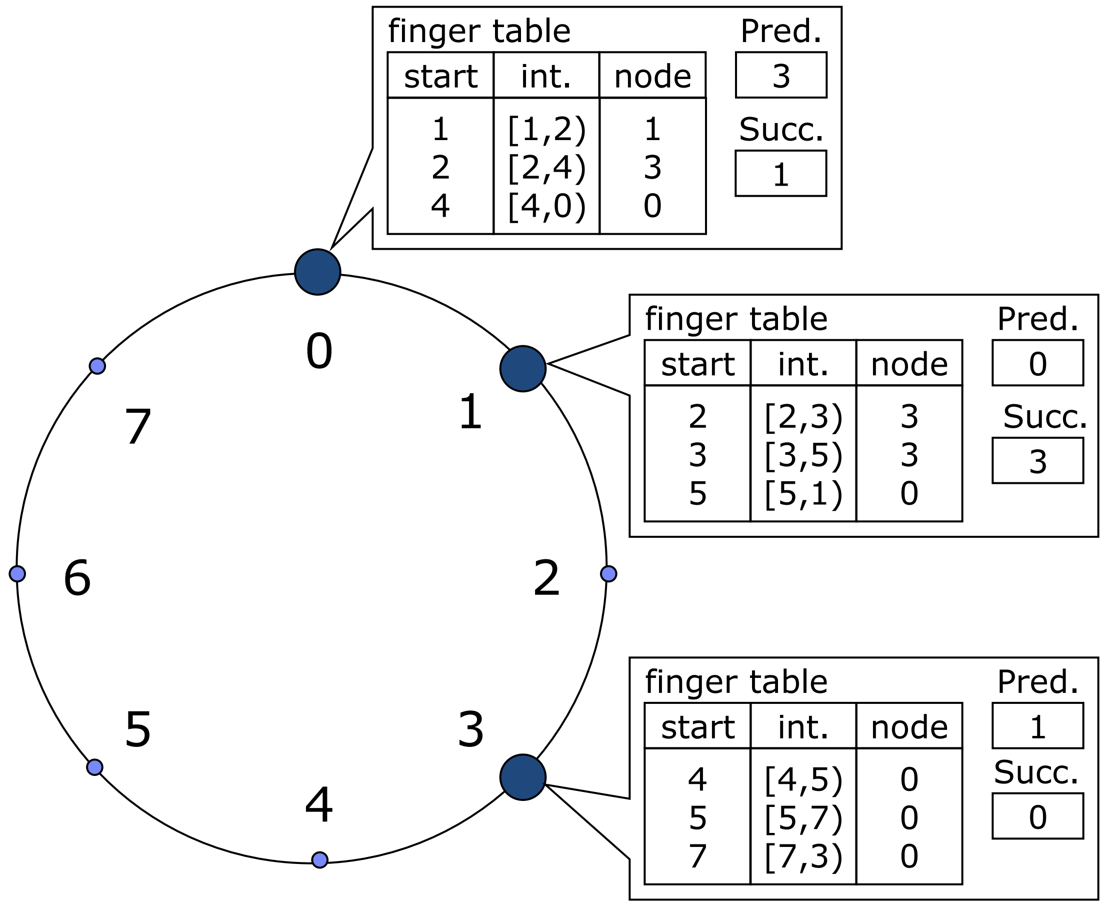
The Routing Information in Chord

- ***Each node holds a table that ‘divides’ the identifier circle into exponentially increasing intervals.***



- ***For each interval, keep the first node, whose identifier is equal to or follows the interval start point.***
- ***In addition, each node keeps a pointer to its immediate predecessor on the identifier circle.***

The Routing Information in Chord – Example



Picture taken from a presentation by Anthony D. Joseph and John Kubiatowicz (Berkely) on peer-to-peer systems
(*http://www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/*)

The Routing Information in Chord - Some notes

- ***Each node stores information about only a small number of other nodes.***
  - 120 in the case of SHA1.
- ***The amount of information maintained about other nodes falls exponentially with the distance between the two nodes.***

The Finger Table – Formal Notations

- ***Node n holds the following finger table:***

Notation	Definition
Finger[k].start	$(n+2^{k-1})$ mode 2^m, for $1 \le k \le m$
Finger[k].interval	[finger[k].start, finger[k+1].start), if $1 \le k<m$ [finger[k].start, n), for k=m
Finger[k].node	*The first node whose identifier is equal to of follows n.finger[k].start*
successor	*Imidiate successor of node n on the identifier circle*
predecessor	*Immidiate predecessor of node n on the identifier circle*

Routing in Chord – Pseudo Code

// ask node *n* to find the successor of *id*

n.find_successor(id)

n' = find_predecessor(id);

*return n'.*successor*;*

// search the local table for the closest predecessor of *id*

n.closest_preceding_node(id)

for i = m downto 1

*if (*finger*[i].node* $\in$ *(n, id))*

return finger*[i].node;*

return n;

Routing in Chord – Pseudo Code

//* ask node *n* to find the predecessor of *id

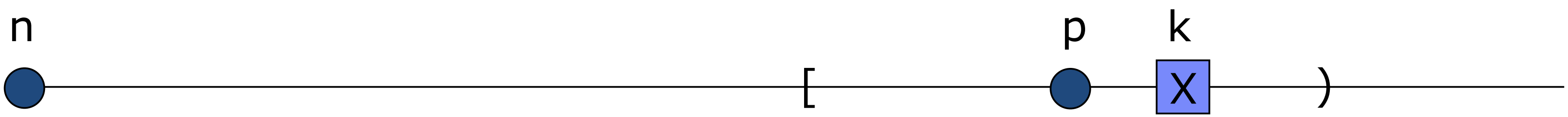
```
n.find_predecessor(id)
  if (n == successor)
          return n // n is the only node in the network
  n' = n
  while (id ∉ (n', n'.successor])
          n' = n'.closest_preceding_node(id)
  return n'
```

Routing in Chord – run time

- ***Theorem: With high probability, the number of nodes that must be contacted to resolve a successor query in an N-node network is O(log N)***
- ***Denote:***
  - ***n*** is the node wishing to resolve the successor of the key ***k***
  - Let ***p*** be the predecessor of ***k***. ***p*** is the actual node we are looking for.

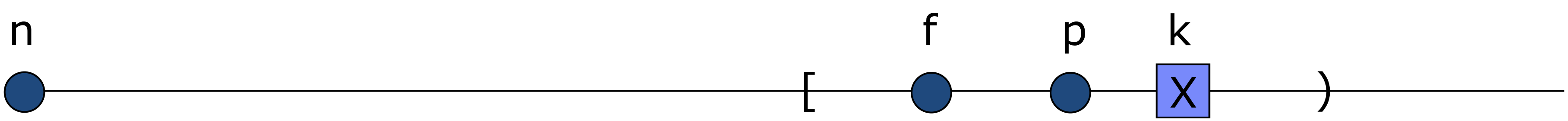
Routing in Chord – run time

- ***Theorem: With high probability, the number of nodes that must be contacted to resolve a successor query in an N-node network is O(log N)***
- ***Denote:***
  - ***n*** is the node wishing to resolve the successor of the key ***k***
  - Let ***p*** be the predecessor of ***k***. ***p*** is the actual node we are looking for.
  - Suppose that ***p*** is in the i'th interval of ***n***.
    - This does not mean that ***p*** actually appears in ***n****'s finger table.*
    - *It merely suggests that this interval is not empty*



Routing in Chord – run time

- ***Theorem: With high probability, the number of nodes that must be contacted to resolve a successor query in an N-node network is O(log N)***
- ***Denote:***
  - ***n*** is the node wishing to resolve the successor of the key ***k***
  - Let ***p*** be the predecessor of ***k***. ***p*** is the actual node we are looking for.
  - Suppose that ***p*** is in the i'th interval of ***n***.
    - This does not mean that ***p*** actually appears in ***n****'s finger table.*
    - *It merely suggests that this interval is not empty*



  - *Let **f*** be n.finger[i].node. That is, the first node in ***n***'s i'th interval.
  - The distance between n and f is at least 2^{i-1}. However, the distance between f and p is at most 2^{i-1} as both appear in the i^{th} interval.
  - Conclusion: By doing a single hop the distance to the key reduced by at least half.

Routing in Chord – run time continue

- **The maximum distance between a node and a key is 2^m**
- **At each hop, the distance is reduced by half**
  - *After log N steps the distance is at most $2^m/N$*
- **Since the nodes are evenly distributed (This is where the high probability comes from), the expected number of nodes in any interval of size $2^m/N$ is 1.**
- **With high probability, the number of nodes in any interval of size $2^m/N$ is bounded by O(log N).**
- **Even if we traverse those in single steps the total time would still be O(log N).**

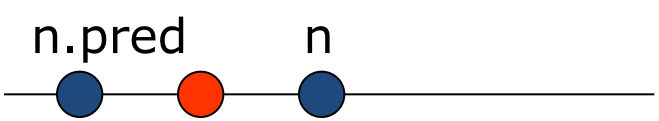
Maintaining the routing information

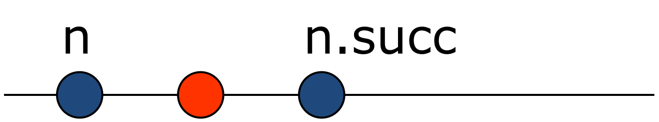
- ***Maintaining / Initializing the routing information in the presence of continuous joins and departures of nodes is hard.***
- ***Solution: Periodical update of the finger table and the successor / predecessor***
  - Turns out that maintaining the successor and predecessor in the presence of many changes is enough for queries to succeed, at the cost of the running time.

// Verify n's immediate pred/succ

// Called periodically.

```
n.stabilize()
  x = predecessor;
  x = x.successor;
  if (x ∈ (predecessor, n))
          predecessor = x
  x = successor;
  x = x.predecessor;
  if (x ∈ (n, successor))
    Finger[1].node = successor = x;
```





References

- ***Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications (2001) by Ion Stoica , Robert Morris , David Karger , M. Frans Kaashoek , Hari Balakrishnan***
- ***Topics in Data-Intensive Computing Systems UBC (http://www.ece.ubc.ca/~matei/EECE571.06/)***
- ***Peer-to-Peer Systems, Anthony D. Joseph, John Kubiatowicz, Berkely course slides (http://www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/)***