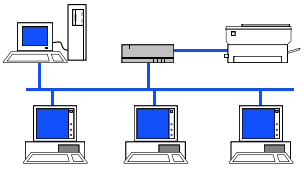


Distributed Systems 600.437 Large-Scale Data Stores & Probabilistic Protocols

Department of Computer Science
The Johns Hopkins University

Large-Scale Data Stores and Probabilistic Protocols

Lecture 11

Further reading:
* Reliable Distributed Systems by Ken Birman - Chapter 25.
* Cassandra - A Decentralized Structured Storage System – Lakshman, Malik
* HyperDex: A Distributed, Searchable Key-Value Store – Escriva, Wong, Sirer

Distributed Indexing

- Distributed (and peer to peer) file systems have two parts:
  - A lookup mechanism that tracks down the node holding the object.
  - A superimposed file system application that actually retrieves and stores the files.
- Distributed indexing refers to the lookup part.
  - The **Internet DNS** is the most successful distributed indexing mechanism to date, mapping machine names to IP addresses.
  - Peer to peer indexing tries to generalize the concept to **(key, value)** pairs.
  - Also called Distributed Hash Table (**DHT**).

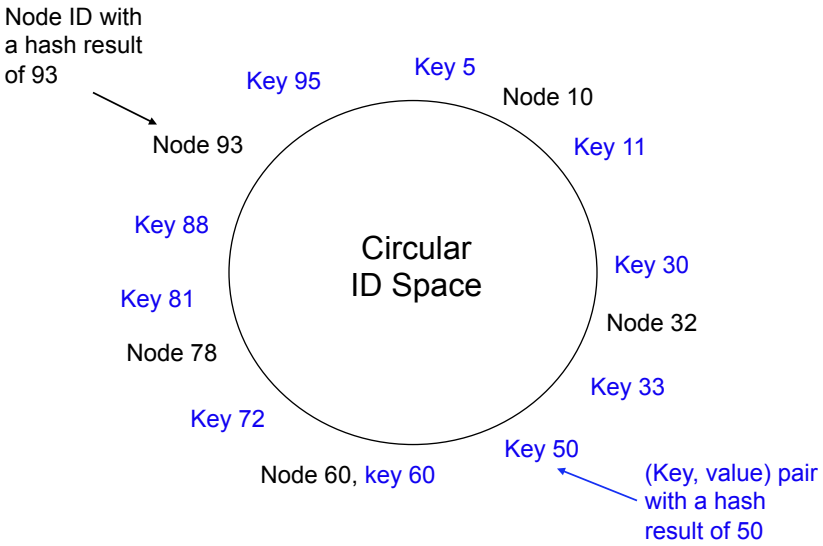
Distributed Indexing (cont.)

- Let us say we want to store a very large number of objects and access them based on their key.
- How would you implement a (key, value) distributed data store that provides good performance for lookup and scales to hundreds or thousands of nodes?
- …
- Now, think about what would you do to ensure robustness in the presence of participants coming and going.

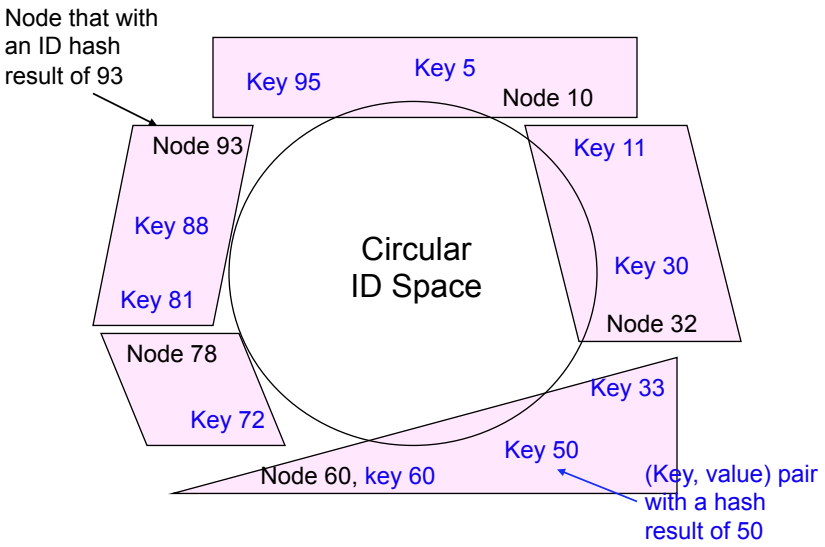
Chord

- Developed at MIT (2001).
- Main idea: forming a massive virtual ring where every node is responsible for a portion of the periphery.
- Node IDs and data keys are hashed using the same function into a non-negative space.
- Each node is responsible for all the (**key,value**) pairs for which the hash result is less or equal to the node ID hash result, but greater then the next smaller hashed node ID.

Chord Indexing



Chord Indexing (cont.)



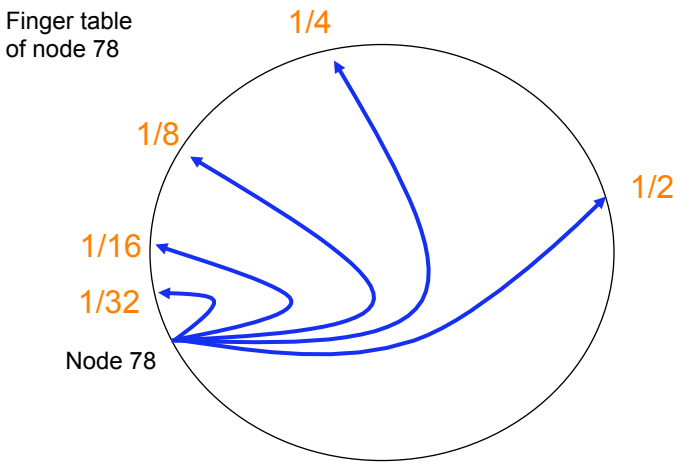
Chord Indexing (cont.)

- Each node maintains a pointer to the node after it and another pointer to the node before it.
- A new node contacts an existing node (startup issue) and traverses the ring until it finds the node before and after it.
- A state transfer is performed from the next node on the ring in order to accommodate the newly joined node.
- Lookup can be performed by traversing the ring, going one node at a time. Can we do better?

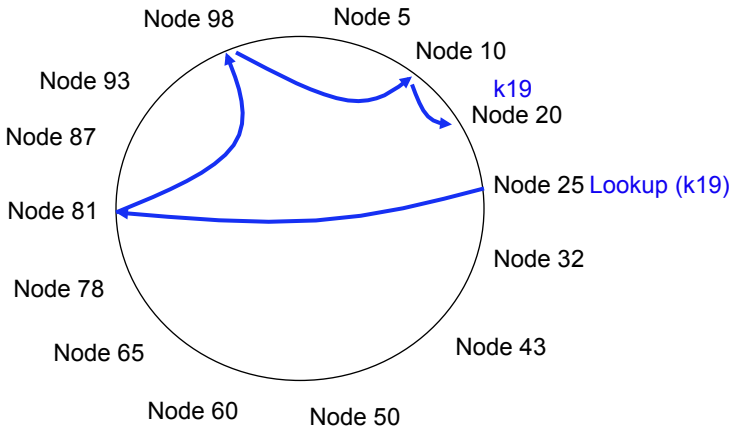
Chord Lookup

- Each node maintains a "finger table" that serves as short-cuts to nodes at various distances within the hash key space.
- Question:
  - How would you construct the "finger table" to allow logarithmic search steps?

Chord Lookup (cont.)



Chord Lookup (cont.)



Chord – Issues to Consider

- Overall, log(n) hops for lookup in the worst case! – very good.
- What is a hop? Where are the nodes? Is log(n) really good?
- What about churn?
- Is it really log(n) worst case over time?
- How to maintain robustness?

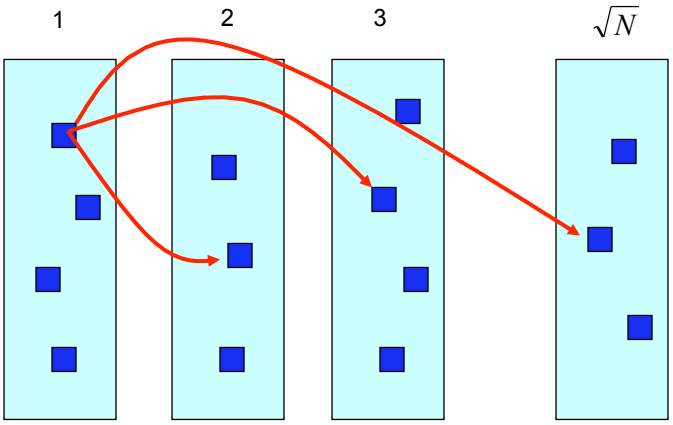
Kelips

- Developed at Cornell (2003).
- Uses more storage (sqrt(n) instead of log(n)) at each node.
  - Replicating each item at sqrt(n) nodes.
- Aims to achieve O(1) for lookups.
- Copes with churn by imposing a constant communication overhead.
  - Although data quality may lag if updates occur too rapidly.
- How would you do that?

Kelips Lookup

- N is approximate number for the number of nodes.
- Each node id is hashed into one of sqrt(N) affinity groups.
- Each key from (**key,value**) pair is hashed into one of the sqrt(N) groups.
- Approximately sqrt(N) replicas in each affinity group.
- Pointers are maintained to a small number of members of each affinity group.
- Lookup is O(1).
- Weak consistency between the replicas is maintained using a reliable multicast protocol based on gossip.

Kelips Lookup (cont.)



Probabilistic Broadcast Protocols

- A class game demonstrating the probabilistic broadcast (pbcast) protocol:
  - At least n >= 20 logical participants.
  - Each participant randomly picks 4 numbers 1-n, noting the order of their selection.
  - Playing the game with the first number, then the first 2 numbers, then 3 and 4 numbers and looking at coverage for a message generated by one participant.

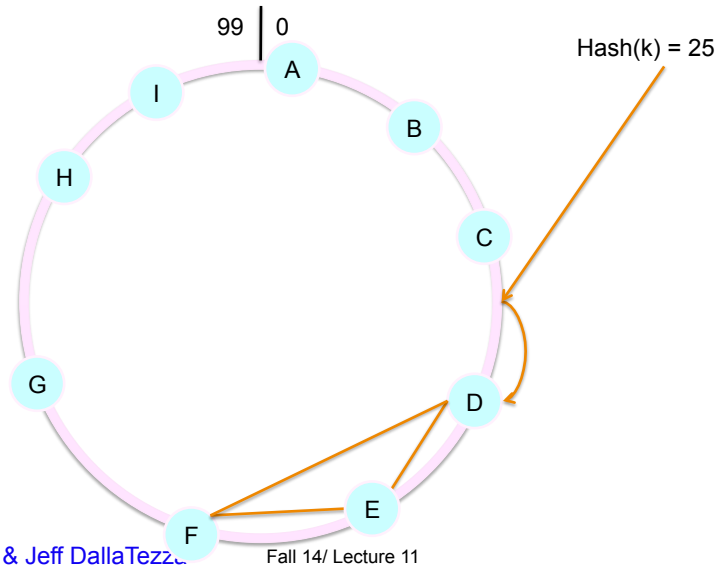
Cassandra: A Distributed Data Store cassandra.apache.org

- Key-value data store, supporting get(key) and put(key, value)
- Designed to scale to hundreds of servers
- Initially developed by Facebook, made open source in 2008
- Heavily influenced by Amazon's Dynamo storage system (which was influenced by Chord)

Cassandra: Partitioning

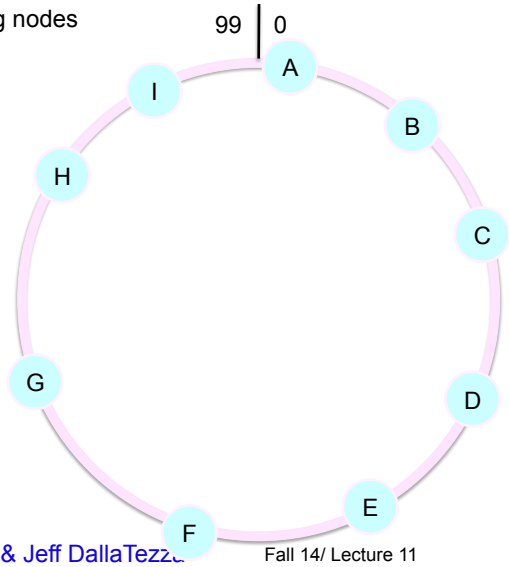
- Keys are hashed into a bounded space, which forms a logical ring (like Chord)
- Servers are placed at different locations on this ring
- Every server maintains information about the position of every other server
- The nodes responsible for a key are found by searching clock-wise from the hashed key
- Data is replicated on the first N servers found

Partitioning, N=3



Partitioning

Load can be rebalanced by moving nodes



Cassandra: Partitioning (cont)

- Changes in node positions (from membership changes or rebalancing) are handled by one node designated as the leader
- The leader stores these updates in Zookeeper (separate system using Paxos) for fault tolerance
- In case of a leader failure, a new leader is elected using Zookeeper
- Changes are disseminated probabilistically
- Every second, each node exchanges information with two random nodes

Cassandra: Replication
Read/Write Quorums

- If data is stored on N replicas, users can configure two values, R and W
- R is the minimum number of replicas that must participate for a read operation
- W is the minimum number of replicas that must participate for a read operation
- As long as R + W > N, every read quorum will contain a node with the latest write

Executing Put(key, value)

- Send put request to any node, which will act as the coordinator for that request
- Coordinator forwards the update to the relevant N replicas
- After W of those replicas have acknowledged the update, the coordinator can tell the client that the write was successful

Executing Get(key)

- Send get request to any node, which will act as the coordinator
- Coordinator requests the object from the relevant N nodes
- After R of those replicas respond, the coordinator returns the most recent version held by any of the replicas

Executing Get(key)

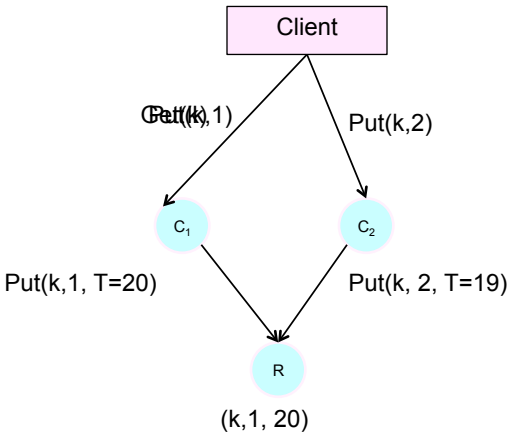
How is the most recent version determined?

Coordinators give each write update a timestamp, based on its local clock

Clock-based Timestamps

- Each put is given a timestamp by the coordinator responsible for that update
- If a replica receives an update with a lower timestamp than its current version, it ignores the update, but acknowledges that the write was successful
- If the clocks on different coordinators drift, this can cause unexpected behavior

Example: Losing an update



Let's say C_2 has a slower clock

Hyperdex: A Distributed Data Store hyperdex.org

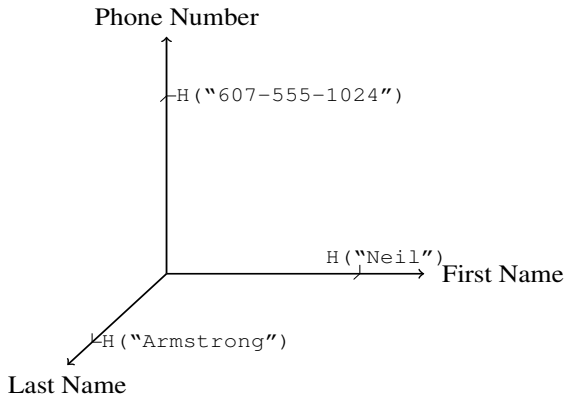
- A large-scale sharded key-value data store (2012)
- Supports get, put, and atomic operations such as conditional puts and atomic addition
- All operations on a single key are linearizable
- Supports efficient search on secondary attributes

Search

- In addition to partitioning based on the key, each object is stored on additional servers based on its secondary attributes
- Combining the hashes of a set of secondary attributes forms a hyperspace which can be partitioned
- This enables efficient search by limiting the number of servers that need to be contacted
- This can be done for multiple sets of attributes

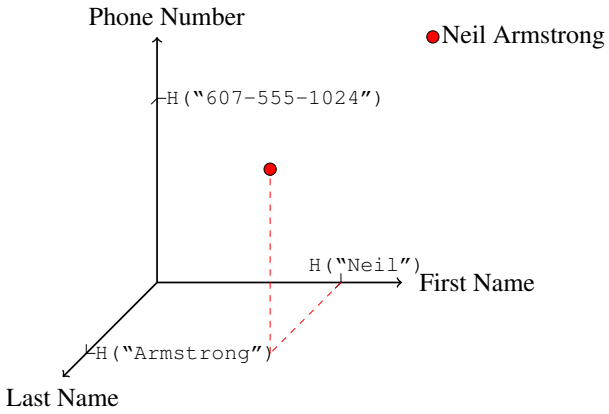
Hyperspace Hashing

Attribute values are hashed independently
Any hash function may be used



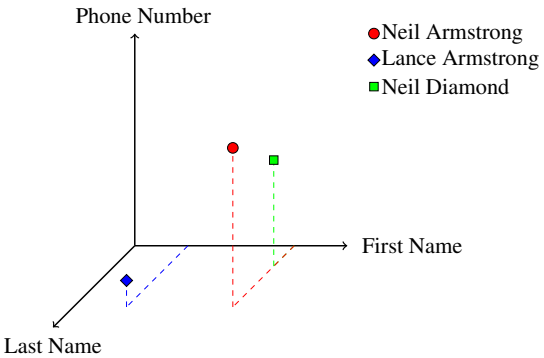
Hyperspace Hashing

Objects reside at the coordinate specified by the hashes



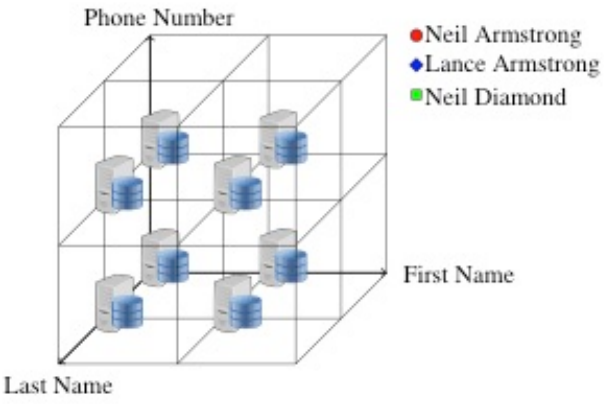
Hyperspace Hashing

Different objects reside at different coordinates



Hyperspace Hashing

Each server is responsible for a region of the hyperspace



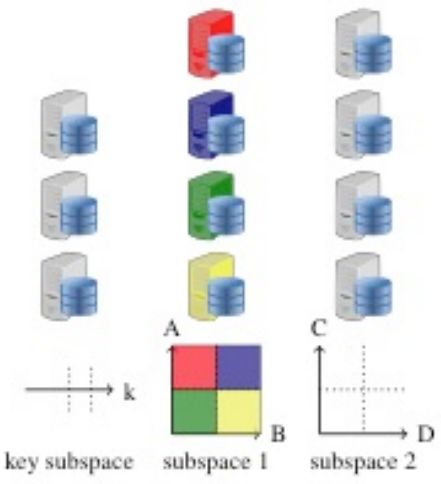
Hyperspace Hashing

- By specifying more of the secondary attributes, we can reduce the number of servers that need to be searched
- If all of the subspace attributes are specified, the search is equally efficient as searching by key
- What if the secondary attributes are updated?

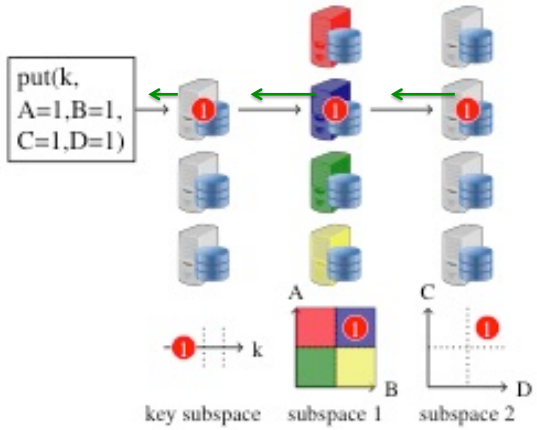
Value-Dependent Chaining

- To perform an update, all of the servers involved are organized in a chain
  - The server responsible for the primary key is at the head of the chain
  - Any server holding the current version of the object is in the chain
  - Any server that will hold the updated version of the object is also in the chain
- The update is ordered at the head and passed through the chain
- Once it reaches the end of the chain, the tail server can commit the update, and pass an acknowledgment back through the chain
- Updates are not committed until an acknowledgement is received from the next server in the chain

Value-Dependent Chaining



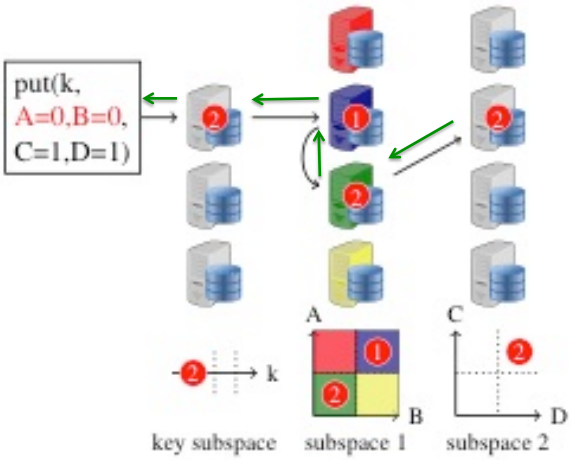
Value-Dependent Chaining



A `put` includes one node from each subspace

Each put takes a forward pass down the chain and is committed during the backward pass

Value-Dependent Chaining



When updating an object, the value-dependent chain includes the servers which hold the old and new versions of the object

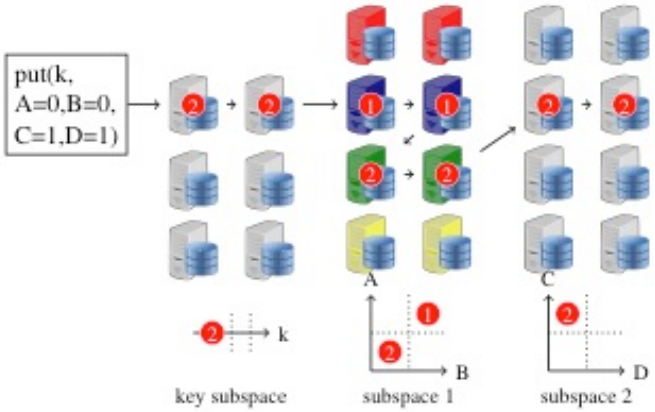
Consistency Guarantees

- Any operation that was committed before a search will be reflected in its results
- In the presence of concurrent updates, either version may be returned, but at least one version of every object will be seen
- Because an update can be reflected in a search before it is committed, search results may be inconsistent with get calls

Fault Tolerance

- Each server in the chain can be replicated
- Hyperdex uses chain replication, but any consistent replication protocol could be used
- If every block of replicas remains available, the system remains available

Fault Tolerance



The value-dependent chain includes all replicas