Big Data Platforms (5 ECTS)
DATA14003
Lecture 9

Keijo Heljanko

Department of Computer Science
University of Helsinki
keijo.heljanko@helsinki.fi

12.10-2021

Amazon Dynamo

- The “mother of all eventually consistent datastores”, aims to be always writable
- Is an Available and Partition tolerant (AP) design
- A key-value store, binary large object (BLOB) data can be looked up by a primary key
- Based on a distributed hash table (DHT), used also by many peer-to-peer systems for data distribution and lookup
- Uses consistent hashing to enable easy elasticity to add or remove servers from a datastore system based on load

Amazon Dynamo

- Published in the paper: “Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: Dynamo: Amazon’s highly available key-value store. SOSP 2007: 205-220”.
- Conflicting data versions are resolved at data read time
- Uses vector clocks for automatically resolving some of the conflicts caused by network partitions
- Handled the Amazon shopping cart system

Consistent Hashing

- Consistent hashing (see e.g., `http://en.wikipedia.org/wiki/Consistent_hashing`) is a way to distribute the contents of a hash table over a distributed hash table (DHT)
- The main advantage of the method is its elasticity. Namely if hash table contains K keys, and is distributed over n servers, either adding or removing a server will only require the relocation of $O(K/n)$ keys
- Note that $O(K/n)$ keys need to be moved for any hashing scheme that maintains an even load balance. Thus the method is essentially optimal

Consistent Hashing

- Basic idea: Assume that the key space is 128 bits (use e.g., SHA-1 to help to compress the key to 128 bits)
- Each compute node selects a random unique node identifier n_i between 0 and $2^{128} - 1$
- All computer nodes are totally clockwise ordered to a virtual ring of servers in increasing node identifier order
- Now to store a data item with key k_j, the node with the smallest node i with node id n_i such that $k_j \leq n_i$ is selected as the server to host it
- If a new node k needs to join the virtual ring with identifier n_k, it will be allocated all data items k_l for which n_k is the smallest identifier such that $k_l \leq n_k$

Chord: P2P use of Consistent Hashing

- "Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. 11(1): 17-32 (2003)."

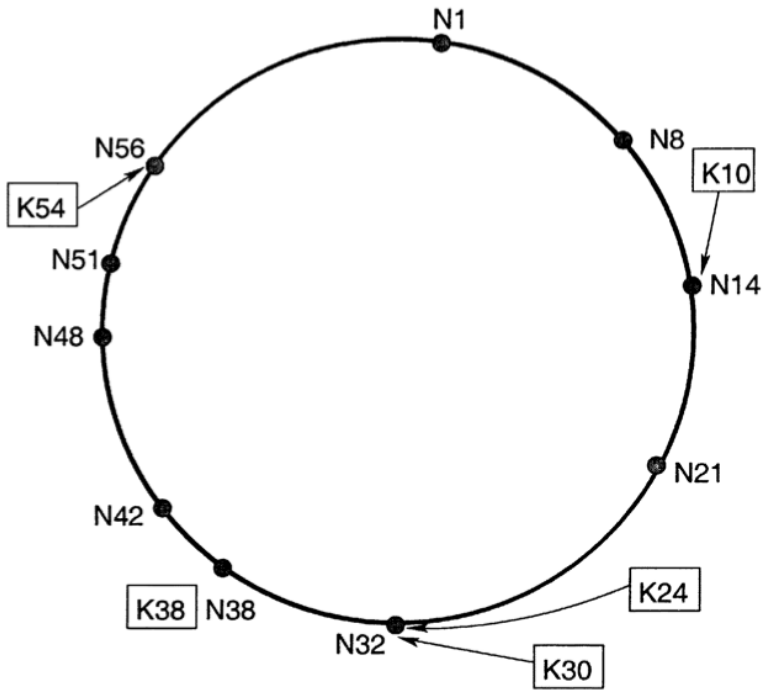


Fig. 2. Identifier circle (ring) consisting of ten nodes storing five keys.

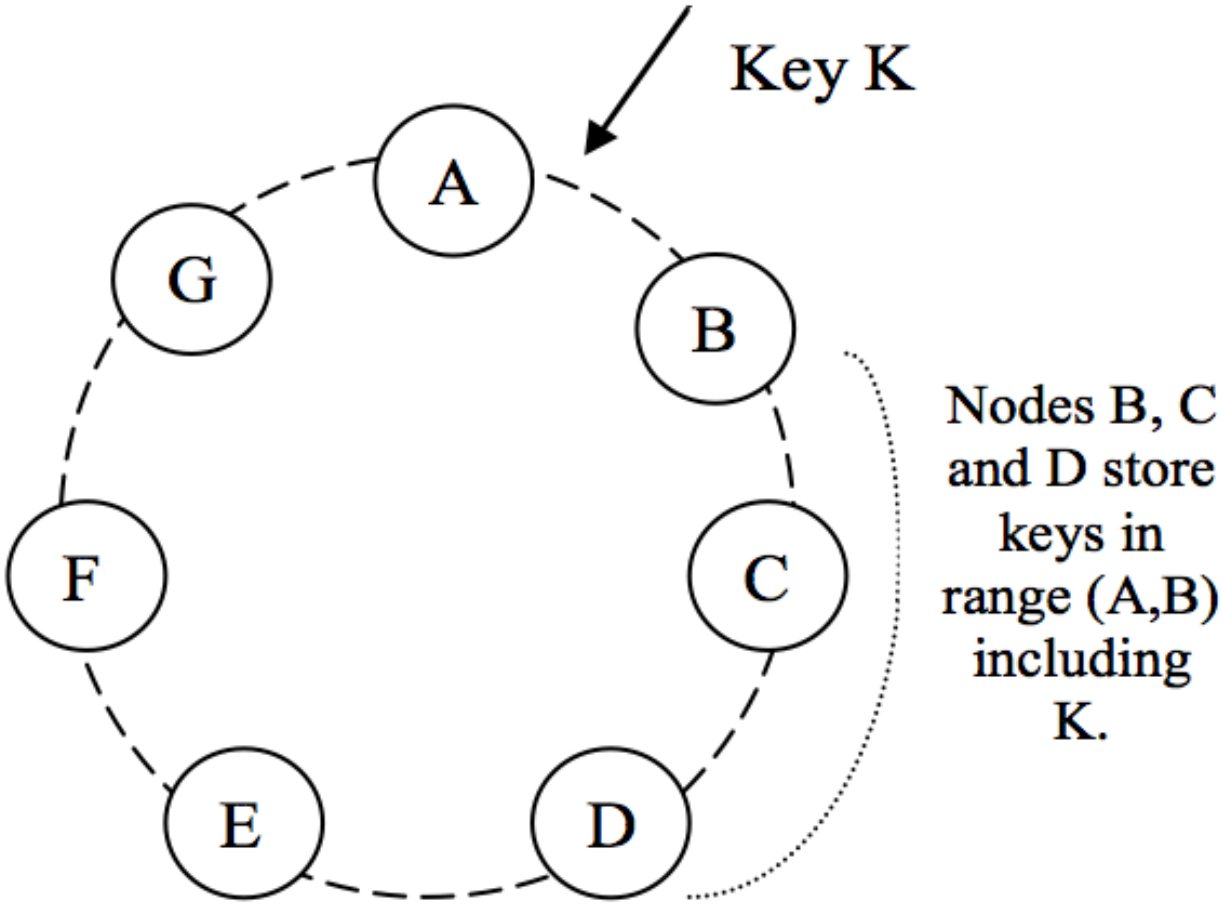
Chord: P2P use of Consistent Hashing (cnt.)

- The Chord protocol itself is buggy under node failures, so don't copy its design directly for other systems:
  Pamela Zave: Using lightweight modeling to understand Chord. Computer Communication Review 42(2): 49-57 (2012)
- However, the consistent hashing ring is still a solid idea

Consistent Hashing Implementation Details

- Dynamo distributes the node id information to all nodes to achieve $O(1)$ lookup of the correct node the data is stored on
- In order to do 3-way replication, Dynamo stores a copy of the data in the node itself and in its two successors clockwise in the ring
- In order to better balance the load (storage requirements), each node of the network simulates 10+ “virtual nodes”, each with their own id. In addition to load balancing, this helps with performance when nodes are added or removed from the system
- Careful with virtual nodes and replication: replicating data on three virtual nodes in the same physical node gives no hardware redundancy. Dynamo handles this problem

Consistent Hashing with 3-Way Replication



Dynamo: Data versioning

- In order to implement the eventual consistency model, Dynamo has the notion of data versioning: Each data item stored in Dynamo has a version number
- During network partition a Dynamo data `put()` method will return before all replicas are updated with the data value
- A technique called hinted handoff is used to store data items destined for nodes that are unreachable due to network partition
- This allows writes to proceed on both sides of a network partition
- When network connectivity returns, data versioning can often be used to recover which versions of the data items are obsolete and which versions are the most current ones

Dynamo: Vector clocks

- Dynamo allows several versions of the data to exists at the same time in the datastore
- Dynamo uses vector clocks to track the modifications made by servers to data items
- One vector clock is a list of [*server*, *clock*]-pairs, where *clock* is a monotonically increasing update counter for *server*
- The vector clock basically lists the versions from which the data item to be `put()` has been derived from
- On read: If Dynamo finds multiple versions of the same data, it will return all the most recent ones to the application
- If multiple versions are returned, the client must merge them to reconcile the conflicting versions

First Example: Vector Clock Auto Merge

- Assume three servers x,y,z and data item D
- Assume we first write data item D1 on server x
- Data set to D1 with vector clock ([Sx,1]), where Sx means server x has modified this data item once: D1, ([Sx,1])
- Assume network connectivity is lost after replicating
- Now server y modifies locally the new value to be D2, we get: D2, ([Sx,1],[Sy,1]), meaning server y has seen the value written by x, and has modified it once since
- Assume network connectivity resumes
- Because server x and y have different versions of the data we need to do a merge
- Because the counters in D2, ([Sx,1],[Sy,1]) are at least as big if not larger than in D1, ([Sx,1]), we can safely use server y version as the final value of the database

Second Example: Vector Clock Conflict

- Server x writes the initial value D1: D1, ([Sx,1])
- Server x writes the second value D2: D2, ([Sx,2])
- After replicating, we loose network
- Now server y modifies locally the new value to be D3, we get: D3, ([Sx,2],[Sy,1])
- Also server z modifies locally the new value to be D4, we get: D4, ([Sx,2],[Sz,1])
- Assume network connectivity resumes
- Because server x, y, and z have different versions of the data we need to do a merge
- Server x has older value than both y and z, and thus can be discarded
- However, there is no automatic way to merge the concurrent updates D3, ([Sx,2],[Sy,1]) and D4, ([Sx,2],[Sz,1])! Database will ask the application to do the merge to reconcile by x into D5, ([Sx,3],[Sx,1],[Sz,1])!

Second Example: Merging by Application

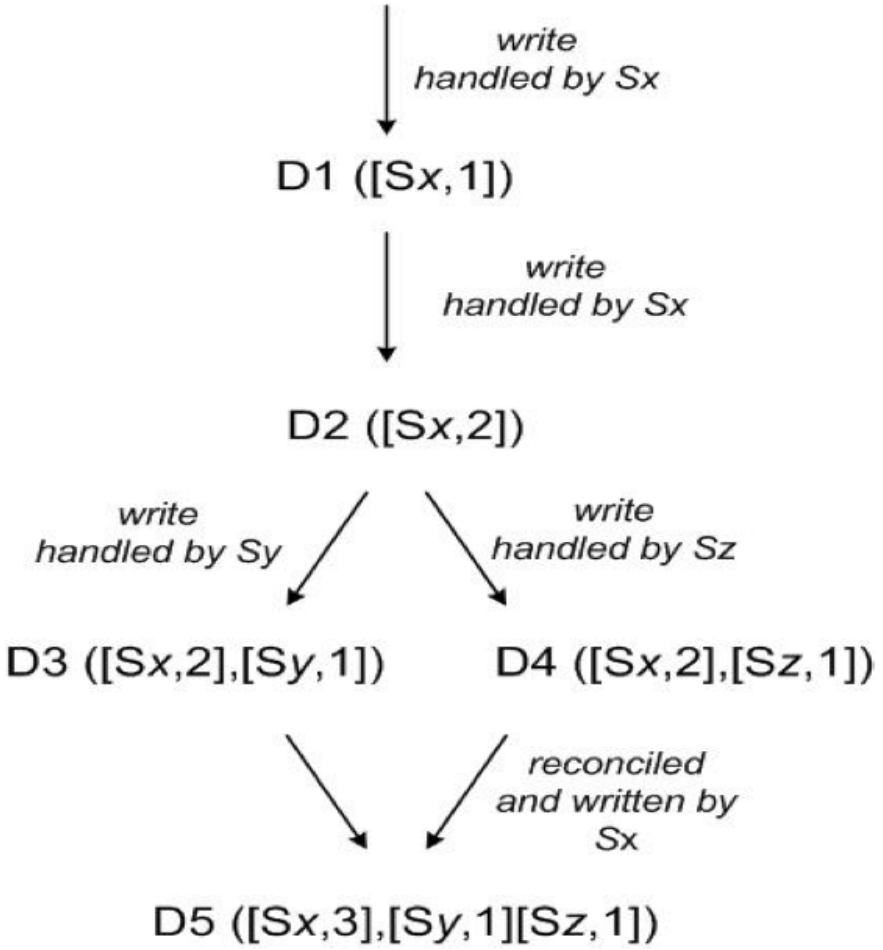


Figure 3: Version evolution of an object over time.

Conflicts and Merge Routines

- Vector clocks are expensive to implement, and thus also other cheaper mechanisms (allowing less automated merging) are used in practice
- AP systems allowing concurrent writes will eventually have conflicts that can not be automatically merged
- What happens if the application has not provided a merge routine?
- Many databases default to “Last Write Wins (LWW)”- using real time timestamp to figure out which update was the most recent chronologically
- LWW can result in data loss! It throws away one of the conflicting versions instead of merging its contents
- Be careful when using AP datastores and check how they perform merges

Dynamo: Read and Write Quorums

- With N-way replication, Dynamo has two additional configurable values: W and R
- On the write path, writes are allowed to return to the client when at least W replicas has acked the write
- On the read path, reads are allowed to return data to the client when at least R replicas have returned data
- When $W + R > N$, the Dynamo system can still be fully consistent
- There is a hidden catch: This is true only if no partitions or other node failures occur!
- For example, consider three way replication N=3, W=1, R=3. If the only node with data fails before having replicated the data to two other replicas, the data is lost forever
- The latency improvement of not having to wait for all replicas to ack writes/reads can be useful

Dynamo: Other Features

- Hinted handoff: If a replica is not available for writes, an arbitrary node can take its place and store data for it until the replica in question comes available
- A hierarchical hashing algorithm called Merkle trees is used to update data to a replica node that has been offline for a while and comes back to join the ring
- A gossip based protocol is used to find failed nodes in the ring and to eventually replace them
- Local storage engine based on MySQL

Amazon Dynamo: Advanced Techniques

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Apache Cassandra

- Apache Cassandra is a cloud datastore that values low latency and availability over consistency
- Developed originally by Facebook, now also commercially supported by Datastax
- Runs behind many massive Websites: Example - Netflix
- It uses a BigTable-like data model and a write optimized storage engine based on SSTables

Apache Cassandra

- Apache Cassandra is maybe the closest open source project in spirit to Amazon Dynamo
- Low latency and availability over consistency
- Its design resembles that of Dynamo:
  - Quorum reads and writes with configurable N, W, and R. Also new Paxos based “light transactions” for atomic updates inside a single partition.
  - No vector clocks available! Automatic merging of conflicting versions during reads using “read repair”, where the data item with the largest timestamp value wins!!! (mutating data stored in Cassandra can be dangerous without proper care!)
- ScyllaDB is a Cassandra clone written in C++

Riak

- Another widely used AP datastore
- Just key-values, not wide rows like in Bigtable or Cassandra
- Uses consistent hashing
- Last write wins (LWW) by default, this is dangerous for mutating data!
- Implements vector clocks, which makes updating data easier by making full custom merge functions easier to write
- Implements Commutative Replicated Data Types - CRDTs, allowing fully automatic merging for certain data types with limited operations
- Also runs large systems in production, is commercially supported by Basho

Commutative Replicated Data Types (CRDT)

- CRDTs are data types for which automated merging of concurrent updates is always possible in a safe manner
- Thus for CRDT datatypes it is safe to do updates even when network connections between database replicas are down
- When network connectivity resumes, the database can automatically do the data merging for CRDTs without any data loss
- Note, however, that updates are not immediately visible under network outage but will only become visible when network connectivity is resumed
- Using CRDTs provides strong eventual consistency and using it in in some applications a CP datastore can be replaced by an AP datastore

Commutative Replicated Data Types (CRDT)

- High level intuition behind CRDTs: If the merged state of the data can be computed by performing (merging) all the operations made by the servers in any order, then the datatype is a CRDT
- Consider an integer counter i with initial value 0 that can only be incremented: If server x does an increment, server y does an increment, and server z does an increment, then final value of i is 3 irregardless of the order the increment updates to the database are deployed
- Thus integer counter with increment operation can be implemented as a CRDT

Merging counters - State based CRDT

- The merge can be implemented by each one of the servers doing their local counts, and then broadcasting their latest local counter value. Thus each server keeps triplet (i_x, i_y, i_z) of counters, and updates each of the counters as the largest values the counters broadcasted so far
- When a database client asks for the value of i, these counters are summed up to provide the latest known estimate of i
- This state based CRDT implementation can batch counter updates locally, and use a gossip based algorithm to update the counters to maximum value heard from each server in a best-effort manner

Merging counters - Operation based CRDT

- Alternatively, each server could just broadcast all of the operations it does locally to all other replicas in a reliable manner, and each replica could then apply them in arbitrary order
- This requires that operations are not dropped or duplicated when transmitted to the other replicas, otherwise the counts will not match!
- The operation based CRDTs require reliable communications, but can be more efficient when the amount of data needed to send over the network in a state based CRDT is too large

Positive / Negative Counter CRDT

- What if we want to allow decrement operations in addition to increment operations?
- We can implement this by having one CRDT counting the number of increments, and another counting the number of decrements
- The final value is then the increment counter value minus the decrement counter value
- Thus Positive / Negative counter can also be implemented as a CRDT

Set can not be implemented as CRDT!

- Consider the set datatype with add(item) and remove(item) operations
- Set can not be implemented as a CRDT!
  - Example assume set S is initially empty: Concurrently server x performs add(a), server y performs add(a), and server z performs remove(a)
  - If we perform (merge) the operations in the order: add(a), add(a), remove(a), we obtain an empty set
  - If we perform the operations in the order: add(a), remove(a), add(a), we get the set with element a in it
  - Thus the final value of the datatype depends on the order of performing the datatype operations, and thus a set datatype can not be implemented as a CRDT

Commutative Replicated Data Types (CRDT)

- Shapiro, M., Preguica, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. app. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), 2011
- More CRDTs listed at:
  `https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type`

Programmer Hints for AP Datastores

1. Hint: Write once-read many: If data is written only once, there can not be two conflicting versions of the data
2. Hint: If you mutate data, do not use “last-write-wins” - it is an unsafe default, and can result in data loss
3. Hint: If you have to mutate data, use Commutative Replicated Data Types CRDTs, either supported by the datastore, or roll your own, if the datastore does not support CRDTs
4. Hint: Not all data types can be turned into CRDTs (example: set with both add and remove operation can not!), for these data types use a CP datastore instead!

Testing Cloud Datastores

- Many cloud datastores can lose data, see: `http://jepsen.io`
- Examples:
  - Broken protocol design: Redis, NuoDB, Elasticsearch, MongoDB (version 4.2.6)
  - Protocol bugs that have been fixed: `etcd`
  - Trying to be AP and CP at the same time: RabbitMQ
  - Users naively using last-write-wins that is on by default: Riak, Cassandra
  - Default options and bugs that can result in data loss: MongoDB
  - Missing a CP configuration option: Kafka, added in newer versions

Marketing vs. Reality

- MongoDB marketing materials used to claim “Full ACID Transactions”:
  `https://web.archive.org/web/20200510151604/https://www.mongodb.com/`
- The jepsen tool found many bugs in MongoDB 4.2.6 implementation, including misleading use of the term ACID:
  `http://jepsen.io/analyses/mongodb-4.2.6`
- The updated claim is now “Distributed multi-document ACID transactions with snapshot isolation”:
  `https://www.mongodb.com/`
- It is not yet clear if the bugs in MongoDB snapshot isolation found by Jepsen tool have been fixed or not