CS-E4640 - Big Data Platforms Lecture 9

Keijo Heljanko

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

21.11-2018

Amazon Dynamo

- The “mother of all eventually consistent datastores”, aims to be always available for database writes
- 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”.
- Allows for conflicting versions of data to be written, 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 hash 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)."

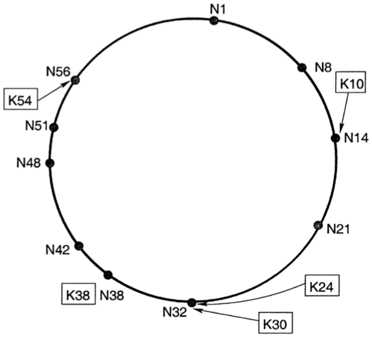


Fig. 2. Identifier circle (ring) consisting of ten nodes storing five keys.

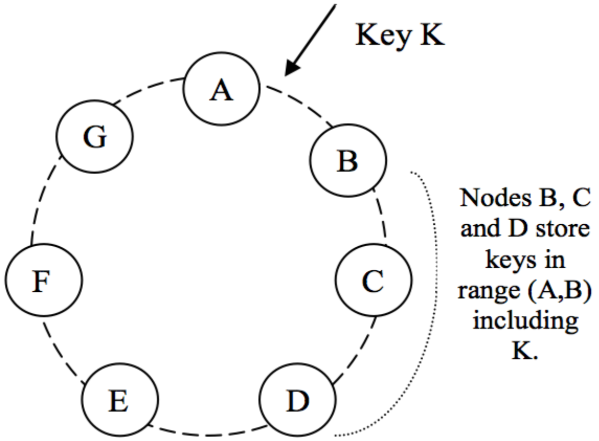
Chord: P2P use of Consistent Hashing (cnt.)

- Note that the Chord protocol itself is buggy under node failures and needs to be fixed:
  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 (*node*, *counter*)-pairs, where *counter* is a monotonically increasing update counter for *node*
- 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

Example: Vector Clocks - Merging by Client

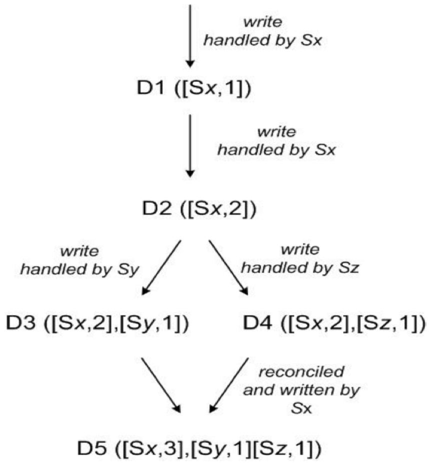


Figure 3: Version evolution of an object over time.

Dynamo: Read and Write Quorums

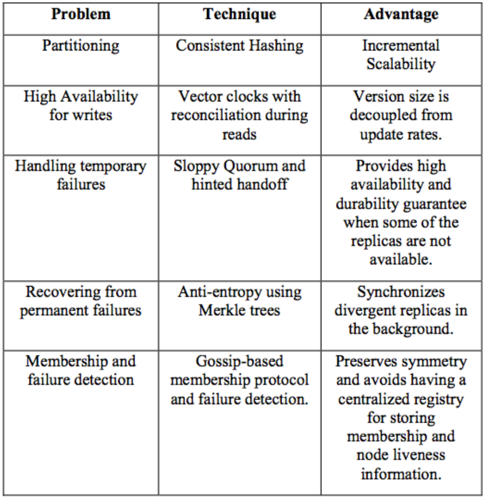
- With N-way replication, Dynamo has two additional configurable values: W and R
- On the write path, writes are allowed to return to the client when at least W replicas has acked the write
- On the read path, reads are allowed to return data to the client when at least R replicas have returned data
- 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!)
- The latency improvement of not having to wait for all replicas to ack writes/reads can be useful

Dynamo: Other Features

- Hinted handoff for improving fault tolerance: 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 maybe the closest open source project in spirit to Amazon Dynamo
- 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
- Otherwise its design resembles that of Dynamo:
  - Quorum reads and writes with configurable N, W, and R
  - 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!)

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 merge functions easier to write
- Also runs large systems in production, is commercially supported by Basho

Programmer Hints for AP Datastores

1. Hint: Write once-read many: If data is written only once, there can not be 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 vector clocks, and ensure that the merge function you use is idempotent, associative, and commutative (i.e., use Conflict-Free Replicated Data Types CRDTs)
4. Hint: Not all data types can be turned into CRDTs (example: set with both add and delete operation!), for these data types use a CP datastore instead!

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

Testing Cloud Datastores

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

Transactions in Distributed Databases

- The most widely used protocol to coordinate transactions in distributed databases is “two phase commit” (`http://en.wikipedia.org/wiki/Two-phase_commit_protocol`)
- In a two phase commit algorithm you have two kinds of servers: A transaction “coordinator”, and database servers called “cohorts”
- The protocol consists of two main phases:
  - A voting phase, where the coordinator asks all cohorts to prepare for a transaction
  - A completion phase, where the coordinator asks all cohorts to either commit (if all cohorts voted for the transaction to proceed) or rollback the transaction (in all other cases)

Transactions in Distributed Databases

- If a cohort crashes during transaction, the system will automatically either rollback or commit the transaction that was in progress for that cohort
- If a coordinator crashes permanently in the middle of the completion phase, some cohorts might be blocked forever waiting for either a commit or a rollback message to arrive
- The problem can not be easily solved without changing the used protocol
- Thus two phase commit can block indefinitely: Two phase commit in distributed databases is not fault tolerant!

The Asynchronous Consensus Problem

- Classic problem in distributed systems, can be used e.g., to decide whether to commit a transaction or not
- System has N processes
- Each process i starts with an initial vote $v_i \in \{0, 1\}$
- Asynchronous (no upper bound on message delivery) but reliable network (all messages are eventually delivered). Note: In particular there is no access to a common clock
- At most one process fails by halting (becoming permanently unresponsive)
- The protocol should terminate in finite time by deciding 0 or 1, and the decided value should be an initial vote of one of the processes (otherwise we could trivially always e.g., decide 0)

The FLP Theorem

- Published in the paper: “Michael J. Fischer, Nancy A. Lynch, Mike Paterson: Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2): 374-382 (1985)”
- Impossibility result: There is no algorithm that will solve the asynchronous consensus problem!
- Key intuition: In an asynchronous model it is impossible to distinguish between a failed process and a slow process that has all its messages delayed
- Thus a consensus protocol will have very long (actually infinite) runs where it is assuming some slow process is dead, making no progress in achieving consensus
- The problem becomes decidable if messages always have an upper bound transmission delay

Detailed Look at FLP Proof

- A detailed look at the proof: FLP proves that any sound fault-tolerant protocol for consensus in an asynchronous setting has runs that never terminate in consensus
- However, in practice these runs consist of an infinite sequence of cases where many different process are repeatedly assumed to have failed because their messages are delayed

FLP and Practical Approaches to Consensus

- In a practical computer networks the probability of infinite runs where the correctly functioning processes repeatedly keep suspecting each other to be faulty due to excessive message delays is diminishingly small
- Thus a practical consensus algorithm can be made that with very high probability quickly solves the asynchronous consensus problem

Paxos Algorithm

- The Paxos algorithm was designed to be a sound algorithm for the asynchronous consensus problem
- It is presented in the paper: “Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)”
- The guarantee is the following: If the Paxos algorithm terminates, then its output is a correct outcome of a consensus algorithm
- Paxos does not always terminate (FLP Theorem still stands!), but instead it terminates in practice with very high probability

Paxos Algorithm

- One can use Paxos to implement a highly fault tolerant database
- The Paxos algorithm uses $2f + 1$ servers to tolerate f concurrent failures, and still make progress
- Thus if one needs to tolerate two failures, five servers are needed
- Key intuition: If modifications are saved on at least $f + 1$ servers (a majority / a quorum), at least one of them will survive f failures
- For efficiency reasons, the algorithm uses a centralized leader through which all modifications are done
- If the leader is expected to have failed, a new leader can be (with high probability) quickly elected, and modifications sent to it instead