CS 695: Virtualization and Cloud Computing

Lecture 13: High performance key value stores: Dynamo

Mythili Vutukuru

IIT Bombay

Spring 2021

Amazon’s Dynamo

- Dynamo is a distributed key-value store: simple get/put interface
  - Map a key to a blob of instructured data, stored across multiple nodes
  - Example of No-SQL data store (unlike traditional RDBMS)
- Highly available (responsive even when nodes fail), high performance (high throughput, low average/tail latency), highly scalable (throughput scales with increasing nodes)
- Weak consistency (eventual consistency): a get may not always return the latest value put in the past
  - No atomicity, isolation, or consistency (ACID of RDBMS)
  - A get may also return multiple conflicting values
- Suitable for applications that can tolerate inconsistencies (e.g., shopping cart)
  - Building block for many Amazon services (S3, DynamoDB)
  - Traditional RDBMS is an overkill for such applications

Dynamo: Amazon’s Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

System architecture

- A service chain of web servers, application servers, data stores
- Aggregator services aggregate data from multiple applications
- Very important to keep even tail latency (e.g., 99.9 percetile latency) low

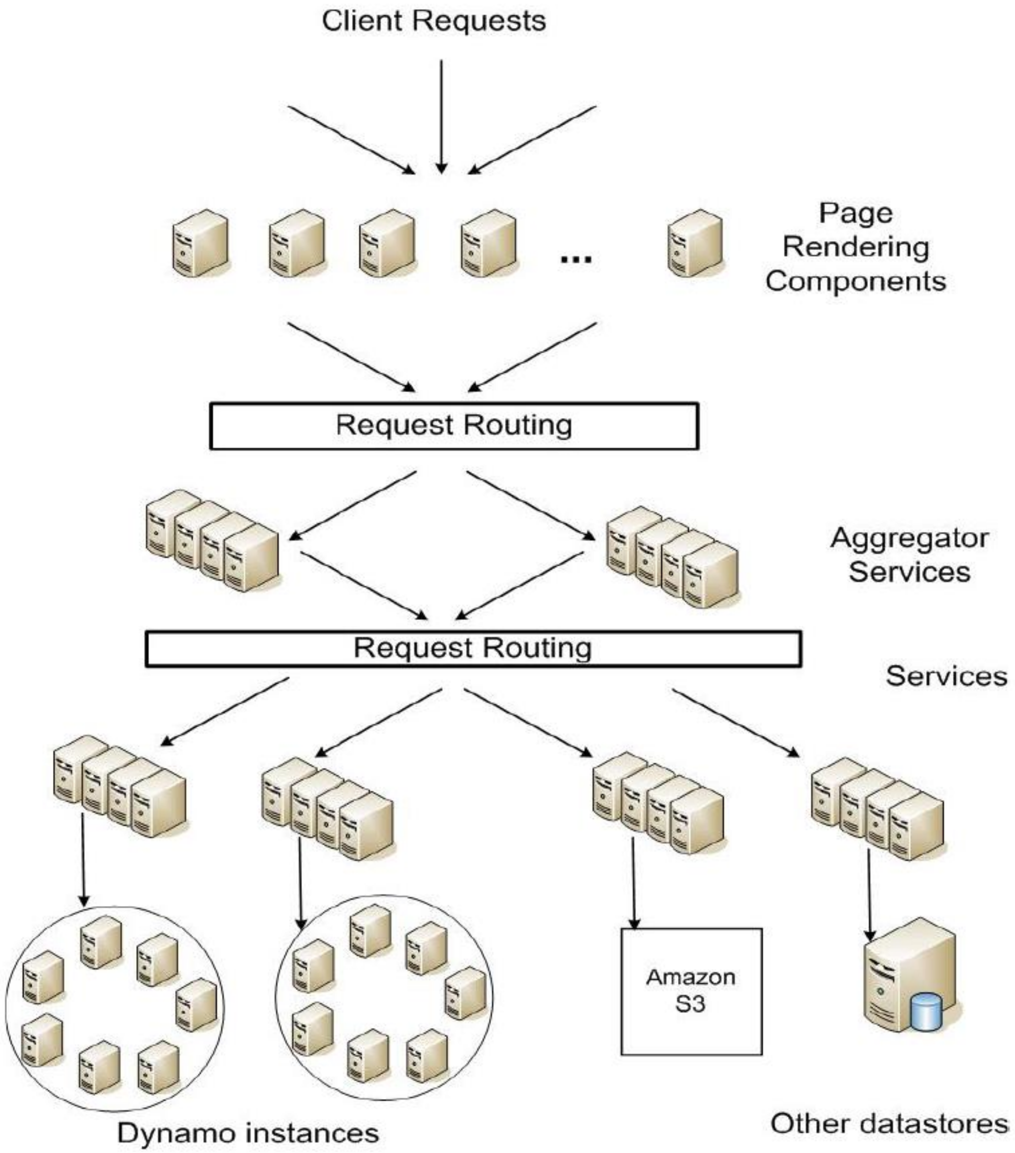


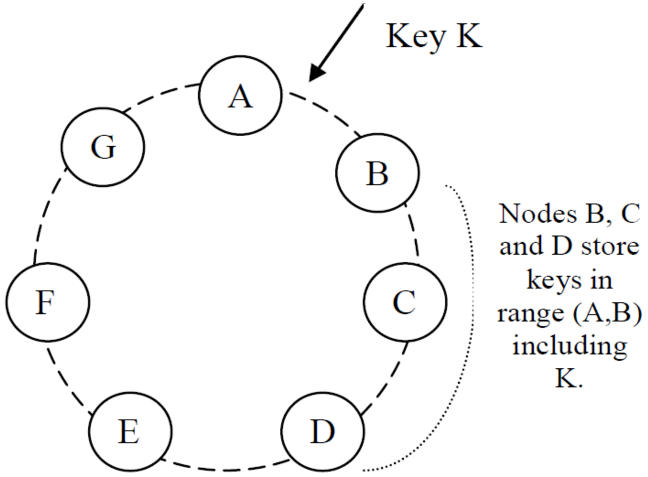
Figure 1: Service-oriented architecture of Amazon's platform

Key idea of Dynamo

- Dynamo partitions the keys over the set of nodes using consistent hashing
  - Every key is stored at a subset N of the total nodes (“preference list” of a key)
- Shared-nothing architecture: each replica independently stores state
  - System can scale by adding more nodes
- Put operation: the key is written to a subset W of the N nodes
  - Succeeds even if some subset of nodes are unavailable
- Get operation: the key is read back from some subset R of the N nodes
  - Eventual consistency: get may not return latest put
  - Multiple values can be returned, application has to reconcile
- Dynamo chooses R,W,N such that R+W > N, so that the latest value can be returned most of the times
  - Quorum protocol
  - R,W chosen to be less than N in order to achieve good latency

Assigning keys to nodes: Consistent Hashing

- Every key is hashed to generate a number in a circular range
- Every node/replica assigned an ID in the same space
- A key is stored at the first N nodes which succeed the hash of the key in the circular ring
    - Called "preference list" of the key
- First node on the list is the coordinator for the key
    - Get/put operations at all nodes managed by coordinator
- For better load balancing, every node is treated as multiple virtual nodes, assigned many positions on list
    - Preference list will contain N distinct physical nodes



Failures and eventual consistency

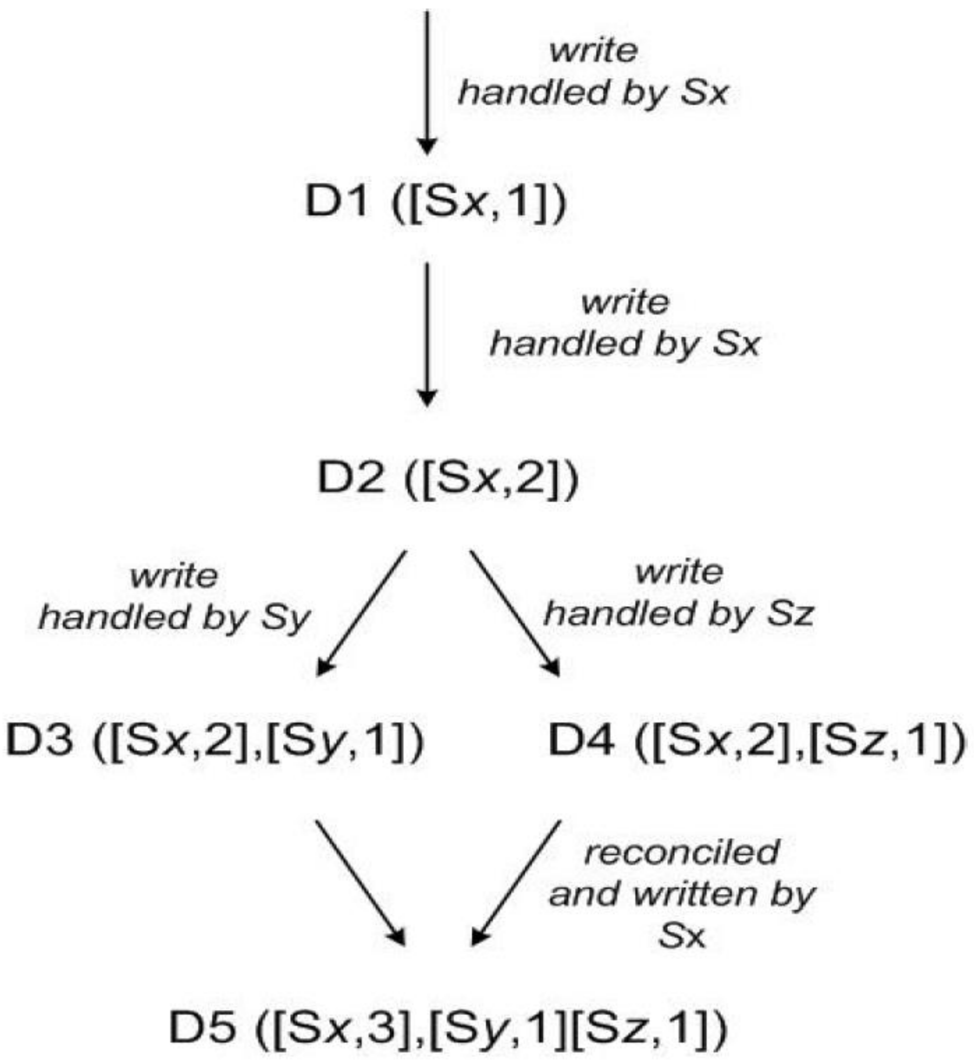
- In cases of node/network failures:
  - Preference list of first N nodes can change with failures, finds first N alive nodes ("sloppy" quorum)
  - Nodes in original preference list will be contacted and updated when they come back
- Put is asynchronous: coordinator does not wait for confirmation from all W nodes before sending a reply to the client
  - In case of failures, a put may not reach all W nodes
- A get after a put can find multiple versions of the key at different nodes
  - Consistency (get returning latest put) is only guaranteed "eventually"
- Why this design? Because one of the goals is to be always writeable
  - System should never turn down a write request from a client
  - Systems with strong consistency will turn down client requests in case of failures, and will only accept requests when they can guarantee consistency

Versioning: vector clocks

- Since multiple versions of a key-value pair can exist, need some version number to track values
- Dynamo uses the idea of vector clocks to version the key-value pairs
- Vector clock is a set of (node, count) pairs, where the count is incremented locally at every node
  - Every node that handles a key will add/increment its entry in the vector clock
- Vector clock version number associated with value (also called "context") is returned with every get to the client, and the client sends it along with its next put request
  - object, context = get(key)
  - put(key, context, object)
- Suppose there are three nodes X, Y, Z handling a key
  - Suppose client gets a value from X with vector clock [(X, nx), (Y, ny), (Z, nz)]
  - Next put at X will increment the vector clock to [(X, nx+1), (Y, ny), (Z, nz)]
  - If put done at Y instead, vector clock will be [(X, nx), (Y, ny+1), (Z, nz)]

Example of vector clocks

- A client gets D1, puts D2, Sx is the coodinator
  - D2 directly descends from D1, can overwrite D1
- Client reads D2, Sx is down, so client writes D3 at Sy. Another client reads D2 and writes D4 at Sz in parallel
  - Both D3 and D4 descend from D2
  - D3 and D4 can have conflicting changes, one does not overwrite the other
- On the next get, both D3 and D4 returned to client
  - Node performing get cannot decide which to return
  - Client must reconcile and arrive at new value (D5)
- Next put of D5 at Sx has combined vector clock, indicating reconciliation has happened
  - D5 can overwrite D3 and D4



Summary of key ideas

- Discussed in this lecture:
  - Consistent hashing to partition keys to nodes for scalability
  - “Shared nothing” architecture to scale over multiple nodes
  - Handle temporary failures via sloppy quorum, async writes
  - High availability by settling for weaker consistency guarantees
  - Leave it to application to reconcile inconsistencies using vector clocks
- More details in the paper:
  - Handling permanent failures
  - Membership changes

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.