Dynamo: Amazon’s Highly Available Key-value Store

Ashvin Goel
Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Many slides adapted from a talk by Peter Vosshall
Amazon’s eCommerce Platform Architecture

- Loosely coupled, service-oriented architecture
- Stateful services manage their own state
- Stringent latency requirements
  - Services must adhere to formal SLAs
  - Measured at 99.9 percentile
- Availability is paramount
- Large scale, keeps growing
  - 10,000s servers worldwide
How does Amazon use Dynamo?

• Shopping cart

• Session information
  • E.g., recently visited products

• Product list
  • Mostly read-only, replicated for high read throughput
Motivation

• Need a highly available, scalable storage system
• Key-value storage is prevalent, powerful pattern
  • Data is mostly accessed by primary key
  • Data served is often self-describing blobs (not structured)
• RDMS is not a good fit
  • Most features are unused, e.g., query optimizer, stored procedures, triggers, etc.
  • Scales up, not out so easily
  • Strongly consistent, limits availability
Key Requirements

• High “always writable” availability is critical
  • Accept writes during failure scenarios (so total ordering not possible)
  • Allow writes without prior context
• User-perceived consistency is also very important
  • Anomalies due to weak consistency should be rare
• Guaranteed latency, measured in 99.9 percentile
• Incremental scalability, reduces TCO
• Tunable latency, consistency, availability, durability
Design Overview

• Dynamo is a decentralized (peer-to-peer) replicated, distributed hash table

• Key design questions
  • How is data placed and replicated on nodes?
  • How to provide availability and consistency under failures?
  • How to route requests to nodes storing the data?

• Techniques
  • Consistent hashing for partitioning the key space
  • Sloppy quorum for high availability and consistency
  • Optimistic replication for eventual consistency
  • Gossip-based protocols for membership and mapping
The get(k) and put(k, v) API includes a context that contains version information (discussed later)

```
// context contains version information returned by previous get.
put(key, object, context)

// get returns one or more object versions.
object[], context = get(key)
```
Consistent Hashing
Why Consistent Hashing?

- Enables partitioning the key space across nodes
- Handles adding and deleting nodes
  - If you use standard hashing, why would this be a problem?
  - Enables incremental scalability
- Handles data replication
Hash ID

• Hash the key using MD5 to a 128 bit ID
• ID lies in a circular key space
Node and Key Assignment

- Key idea of consistent hashing:
  - Each node is assigned an ID in the key space
  - Each key is owned by first clockwise node
Nodes Store Key Ranges

- Each node owns keys in the range between its predecessor and itself
Node Addition/Deletion

- Adding or removing a node affects only a part of the key range
Replication

• A key is replicated at the first 3 clockwise nodes
• Each node stores key ranges between its 3\textsuperscript{rd} predecessor and itself
Key Load Imbalance

- Key range can be unbalanced

Diagram:
- Nodes A, B, C, D, E, F, G are connected in a circular manner.
- A and B are connected with an arrow indicating a large range.
- E and B are connected with an arrow indicating a small range.
Load Balancing via Virtual Nodes

- Map each physical node to multiple virtual nodes
  - Pros: reduces key range skew across physical nodes
  - Cons: increases membership size
Sloppy Quorum
Why Sloppy Quorum?

- Challenge is to ensure both high availability and user-perceived consistency, with two goals:
  - Data should be always writable
  - Avoid anomalies due to weak consistency with high probability
- Solution: Be available
  - Consistent during normal operation, sloppy during failures
Majority Quorum Protocol

• Sloppy quorum builds on majority quorum protocol

• Basic Majority Quorum protocol
  • Assume
    • N: Number of nodes (or replicas) storing a key
    • R: Successful read involves at least R nodes
    • W: Successful write involves at least W nodes
  • Choose: R + W > N
    • Since reads and writes overlap at least one replica, majority quorum ensures reads will read the latest data

• Example:
  • N = 3, R = 2, W = 2
Majority Quorum Example

• Assume $N = 3$, $R = 2$, $W = 2$

• $\text{put}(k, v)$
  • Coordinated by a node that stores key $k$
    • Typically, first replica is chosen
    • However, other replicas may also be chosen for load balancing
  • Returns when at least $W=2$ replicas update key and respond to the coordinator

• $\text{get}(k)$
  • Coordinated by any node (whether node stores $k$ or not)
  • Returns when at least $R=2$ replicas respond with the value of key to the coordinator
Majority Quorum Example

- $N = 3$, $R = 2$, $W = 2$
- Assume client performs $\text{put}(k1, v1)$

- $\text{put}(k1, v1)$ is performed by coordinator $A$
  - $A$ forwards $\text{put}$ to $E$ and $B$
  - $\text{put}$ returns when $A$ receives response from $E$ or $B$
- $E$'s response
- $B$'s response
- $B$ could have failed but $\text{put}$ returns successfully
Sloppy Quorum

1. When a node is not available, writes sent to a new node.
2. Reads and writes are performed on N healthy nodes.
   - So failed nodes are skipped.
   - Sloppy: R+W > N does not guarantee that reads, writes overlap.
3. However, reads still often read the latest data.
Sloppy Quorum

- Assume client performs put(k1, v2)
- If B fails, A forwards put(k1, v2) to D (temporary replica)
- Even if B restarts, get(k1) often returns latest version
Sloppy Quorum and Replica Divergence

• After node B fails, it will have a stale replica
Sloppy Quorum and Failure Recovery

- After node B fails, it will have a stale replica
- When temporary replica D finds that B has recovered:
  - D sends v2 to B, and may delete v2 from its store

Node B recovers from failure
Replica Synchronization

- Nodes may have stale replicas, leave or fail permanently
- Replicas of key ranges are synchronized with an efficient anti-entropy protocol that uses Merkle trees

Node B failed, A or E can send (F-A) range to D

A and E can compare and update (F-A) range
## Sloppy Quorum Configuration

<table>
<thead>
<tr>
<th>N</th>
<th>R</th>
<th>W</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Consistent, durable, user state (typical configuration)</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>N</td>
<td>High performance read engine</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Distributed web cache</td>
</tr>
</tbody>
</table>
Optimistic Replication
Why Optimistic Replication?

• With sloppy quorum, replicas may be stale or conflicting
  • Stale replica: replica has old version
  • Conflicting replica: process wrote to a stale replica

• Optimistic replication is used to
  • Detect stale and conflicting replicas
  • Synchronize them so replicas become eventually consistent

• Dynamo implements optimistic replication using immutable versions and version histories
  • put() creates new, immutable object version
  • Each node tracks version history, i.e., version information for each object version and how they are related
Optimistic Replication Example

- `put(k, v1)` writes to A, E, B

Assume `v1` is a version and a value associated with the version.
Example

• B and E fail
• put(k, v2) writes to A and D
  • D is a temporary replica
Example

- B and E fail
- put(k, v2) writes to A and D
  - D is a temporary replica
  - A removes v1, an ancestor (stale version) of v2
Example

- B and E recover
- A fails
- `get(k)` reads v1 from E and B
  - v1 is a stale version
Example

- A recovers
- put(k, v3) writes to E, A, B
  - Creates branch in history, since put() performed based on stale version v1
Example

- A recovers
- put(k, v3) writes to E, A, B
  - Creates branch in history, since put() performed based on stale version v1
  - E and B remove v1, ancestor of v3
  - A stores v2 and v3, since they conflict
Example

- `get(k)` reads `[v2, v3]` from A, E, B

- Dynamo provides all conflicting versions to client, since client knows best how to reconcile them
  - E.g., app can merge two conflicting shopping carts

- Dynamo expects applications to perform conflict resolution on reads
  - Enables conflicting writes to always proceed
Example

- put(k, v4) writes to A, E, B

- Dynamo expects v4 is a reconciled version, based on v2 and v3

- Version history has single head
  - Object is eventually consistent
Example

- put(k, v4) writes to A, E, B
- Dynamo expects v4 is a reconciled version based on v2 and v3
- Version history has single head
  - Object is eventually consistent
- A, E, B and D can remove stale versions v2 and v3
Implementing Version History With Vector Clocks

• Dynamo uses vector clocks to implement version history
• Efficiently capture causality
  • Stale versions can be forgotten
  • Concurrent versions are conflicting, require reconciliation
• Each object version stores a vector clock:
  
  $$[(\text{node1, } \#\text{updates1}), \ (\text{node2, } \#\text{updates2}), \ldots]$$
Dynamo API With Vector Clocks

- The get(k) and put(k, v) API includes a context that contains version information (vector clock)

```cpp
// context contains version information returned by previous get. 
// helps generate version information for new object version. 
put(key, object, context)

// get returns one or more object versions. 
// context provides version information for each version. 
object[], context = get(key)
```
Gossip-Based Protocols
Why Gossip-Based Protocols?

• Gossip protocols exchange information between nodes in a peer-to-peer (symmetric) manner
  • A<->B: A and B learn about each other’s state
  • B<->C: B and C learn about each other’s state, so C learns about A’s state as well

• In general, these protocols enable nodes to
  • Learn about the state of other nodes
  • Use version history of state to become eventually consistent

• Tradeoffs:
  • Pros: avoid need for a coordinator, provide higher availability
  • Cons: nodes may have stale information for a while
Membership and Mapping

- Dynamo uses gossiping to propagate membership, mapping information
- Administrator explicitly adds and remove nodes
- Membership: After that, nodes communicate with each other to eventually learn about an added/deleted node
- Mapping information: Nodes also learn about node mappings, i.e., the key ranges stored on a node
Routing Key Lookup

- With gossiping, each node knows about 1) all other nodes, and 2) the key ranges each node stores
- Allows one-hop routing (critical for low latency)
Failure Detection

• Initially implemented node failure detection via gossip

• Not needed due to explicit node add/remove
  • No need to distinguish between temporarily failed/recovering nodes versus removed/added nodes

• Simple failure detection
  • A detects B as failed if it doesn't respond to a ping message
  • A periodically checks if B is alive again
  • In the absence of requests, A doesn't need to know if B is alive
Evaluation

500 ms SLA for storage system for shopping cart application
Lessons Learned: Tail Latency

• 99.9 percentile is a high bar
  • Packet losses, waiting on disk, accessing large objects, JVM garbage collection, ...

• Techniques used to reduce tail latency
  • Use buffered writes to avoid waiting on disk
    • Need to deal with version consistency, e.g., if version number is increased on disk, but failure loses the object version
  • Lazy removal of stale versions
  • Adaptive throttling of background operations based on observed foreground operation latency
Lessons Learned: Repartitioning

• Slow repartitioning
  • Successor (C) splits key range to bootstrap new node (D)
  • Requires ordered key traversal, but it causes heavy random disk I/O at C, with throttling, takes hours/days to finish

C splits its key range to send to D

new node D
Lessons Learned: Repartitioning

- Use fixed arcs strategy
  - Divide hash ring into many fixed key ranges called segments
  - Coordinate assignment of segments to nodes
  - New node (D) steals entire existing segments from other nodes, allowing simple file transfer, sequential IO
- Scales better
- However, moves away from decentralized principle
Dynamo: Pros and Cons

• Pros
  • Highly available - 99.9995% request success over one year
  • Meets tight latency requirements
  • Incrementally scalable
  • Tunable consistency, durability

• Cons
  • No transactional semantics
  • More challenging programming model, e.g., handling conflicts
  • Doesn’t support ordered key operations, streaming operations
  • Not appropriate for large (> 1MB) objects
Conclusions

• Highly scalable, replicated, eventually consistent key-value store

• Decentralized (peer-to-peer) techniques can be used for building highly available system
  • High availability: provides an ”always-on” experience
  • Mostly consistent: clients rarely see conflicting versions

• Highly influential
  • Apache Cassandra builds on Dynamo’s design
Discussion
Q1

- What design constraints are imposed by the “always writable” requirement?
Q2

• How would you compare Dynamo against Bigtable in terms of:
  • API
  • Workloads
  • Availability
  • Consistency
Q3

- Say dynamo is heavily loaded, i.e., many of the nodes are loaded, and so the dynamo administrator decides to add a node. Would that help reduce load on all the nodes?
Q4

• Under what scenarios can a client read multiple conflicting versions of an object? Why is this unlikely in Dynamo?
Q5

- What are the scalability limitations of Dynamo?