Dynamo: Amazon’s Highly Available Key-value Store

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

Presenter Jiaxin Li
What is Dynamo?

A highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience.

- Available
- Scalable
- Eventual consistency
- Configurable
Why Dynamo?

➢ Customer Experience demands for reliability and performance

➢ Many Amazon services on platform (scalable)

➢ To satisfy different applications requirements (configurable)
Customer Obsession

Leaders start with the customer and work backwards. They work vigorously to earn and keep customer trust. Although leaders pay attention to competitors, they obsess over customers.

-From https://www.amazon.jobs/en/principles

Customer experience determines that reliability is more important than consistency
Needs to be highly scalable to support continuous service growth on Amazon platform
Needs ability to configure system to satisfy different services requirements
How is Dynamo designed?

1. Consistent hashing
   - Partition
   - Replication
2. Configurable quorums
3. Data versioning
4. Get() and put() interface
5. Handling failures
Why Consistent hashing

- Suppose we want to store customers information across N servers
- Usually we use a hash function and then do modulo
  - Hash function 1: $h_1 = \text{hash(First Name)} \mod N$
  - Hash function 2: $h_2 = \text{hash(Year of Birth)} \mod N$
- What is the problem? (Hint: think about scalability)
Why Consistent hashing

- What if a server is added/removed?
- Lots of key remapping and data transfer
  - Suppose $N = 6$ and now we add 1 more server
  - Hash value of 7 will move from server 1 to 0
  - Hash value of 8 will move from server 2 to 1
  - ...
  - TOO MUCH
- We need to reduce key remapping and data transfer!
What is Consistent hashing

- Represent the hash space as a ring
- Hash everything
  - Hash server ID and put server on the ring
  - Hash key value and put key on the ring
- Run the hashed key clockwise and let the first successor process
- Solves the scalability problem (partition)!
- New problem: skewed load
Virtual nodes

- Represent the hash space as a ring
- Hash everything
  - Each server has multiple IDs
  - Hash server IDs and put all nodes on the ring
  - Hash key value and put key on the ring
- Run the hashed key clockwise and let the first successor process
  - More virtual nodes means better load balancing
Replication

• To achieve high availability and durability, Dynamo replicates its data on multiple hosts.

• The first successor on the ring is the coordinator.

• Coordinator forwards to its successors.

• Suppose # of replications is 3.
Configurable System

- Recall that every key is replicated on N servers
  - Higher durability with greater N
  - But latency also increases!

- Handling reads and writes
  - Coordinator forwards requests to N-1 successors
  - Waits for response from R or W replicas (R W configurable)
If we prefer availability...

\[ N = 3 \quad R = 1 \quad W = 1 \]

A
\( (k, 2) \)

\[ \text{Put}(k, 2) \]

\( \text{Client 1} \)

B
\( (k, 1) \)

\[ \text{Get}(k) \]

\( \text{Client 2} \)

This is just a extreme case...
If we prefer consistency...

$N = 3 \quad R = 2 \quad W = 2$

Client 1

Put($k$, 2)  (k, 2)

Put($k$, 2)  (k, 2)

Client 2

Put($k$, 2)  (k, 1)
If we prefer consistency...

$N = 3 \quad R = 2 \quad W = 2$

- How to know which value is the most recent?
- Data versioning with vector clocks!
Data Versioning

- Dynamo treats the result of each modification as a new and immutable version of the data.
- Allows for multiple versions of an object to be present in the system at the same time.
- Uses vector clocks to capture causality between different versions of the same object.
- In case of conflicts, may apply different reconciliation mechanism.
Get() and put() interface (Part I)

Get(\textit{key})

- Returns list of \langle value, context \rangle pairs
- Context describes version of value
- In case of conflicting versions, write the reconciled version back

Put(\textit{key, context, object})

- Determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk
- Context indicates which version this request supersedes or merges
Get() and put() interface (Part II)

Sloppy quorum

Get(key)

- Setting a value $R$
- The coordinator waits for $R-1$ responses before returning to the client
- Possible to return multiple versions of data and then reconcile

Put(key, context, object)

- Setting a value $W$
- The coordinator sends the new version (and the vector clock) to the $N$ reachable nodes
- If at least $W-1$ nodes respond, then the write is considered successful
Handling temporary failures

- Sloppy quorums
  - All read and write operations are performed on the first $N$ healthy nodes

- What if less than $R$ or $W$ of $N$ nodes are up?

- How does a server catch up when it is recovered?
Hinted Handoff

- Suppose coordinator does not receive $W-1$ replies from $N$ successors
  - Return failure?
  - But want availability
- Coordinator writes to subsequent servers beyond $N$ successors
Hinted Handoff

- Let $N = 3$ and $W = 3$
- B as the coordinator forwards K to C and D
  - C is down
  - B forwards to E
  - E forwards replicated data to C when it is up
Handling permanent failures

- Permanent failures are only identified manually
  - Example: hinted server crashes before sending back data
- Need to detect inconsistencies between replicas faster
- Use Merkle tree
  - Each node maintains a separate Merkle tree for each key range it hosts
Summary

➢ Pros

- Highly scalable by consistent hashing
- Highly available by sloppy quorums and hinted handoff
- Configurable W and R, and reconciliation

➢ Cons

- Eventual consistency by vector clocks, reconciliation and Merkle tree
- Reconciliation logic design depends on the developers