Reliability at Scale

A tale of Amazon Dynamo

Presented by Yunhe Liu @ CS6410 Fall’19
Slides referenced and borrowed from Max & Zhen “P2P Systems: Storage” [2017], VanHattum [2018]
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
Amazon.com
Authors

Giuseppe DeCandia (Cornell Alum: BS & MEng ’99)
Deniz Hastorun
Madan Jampani
Gunavardhan Kakulapati
Avinash Lakshman (Authored Cassandra)
Alex Pilchin
Swaminathan Sivasubramanian (Amazon AI VP)
Peter Vosshall

Werner Vogels (Cornell, Amazon VP, CTO)
Motivation: No Service Outage

- Amazon.com, one of the largest e-commerce operations
- Slightest outage has:
  - significant financial consequences
  - impacts customer trust

Image source: https://www.flickr.com/photos/memebinge/15740988434
Challenge: Reliability at Scale

A key-value storage system that provide an “always-on” experience at massive scale.
Challenge: Reliability at Scale

A key-value storage system that provide an “always-on” experience at massive scale.

- Tens of thousands of servers and network components
- Small and large components fail continuously (extreme case: tornadoes can striking data centers)
Challenge: Reliability at Scale

A key-value storage system that provide an “always-on” experience at massive scale.

- Service Level Agreements (SLA): e.g. 99.9th percentile of delay < 300ms
- Users must be able to buy -> always writable!
Challenge: Reliability at Scale

A key-value storage system that provide an “always-on” experience at massive scale.

- Given Partition tolerance: has to pick between A and C.
Solution: Sacrifice Consistency

- Eventually consistent
- **Always writeable**: allow conflicts
- Conflict resolution **on reads**
  - Defer to applications
  - Defaults to “last write wins”
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Design

- **Interface**
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Interface

\textbf{put} (key, context, object) \rightarrow \textbf{success} \rightarrow \textbf{failure}

\rightarrow \textbf{success} + \textbf{object} + \textbf{context}

\textbf{get} (key) \rightarrow \textbf{success} + \textbf{set of objects} + \textbf{context} \\
\rightarrow \textbf{failure}

- Key/value treated as opaque array of bytes.
- Context encodes system metadata, for conflict resolution.
Design

- Interface
- **Partitioning**
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Partitioning

- **Goal:** load balancing
- **Consistent hashing across ring**
- **Nodes responsible for regions**
  - Region: between the node and its processor.
- **Unlike Chord:** no fingers
Partitioning

- Advantages:
  - Decentralized find
  - Join, leave have minimum impact (incremental scalability)

- Disadvantages:
  - Random position assignment for \((k,v)\) instead of uniform
  - No server heterogeneity
Partitioning

- To address non-uniform distribution and node heterogeneity: **Virtual Nodes**!

- Nodes get several, smaller key ranges instead of a big one
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Replication

- Coordinator node replicates $k$ at $N - 1$ successors
  - $N$: # of replicas
  - Skip positions to avoid replicas on the same physical node

- Preference list
  - All nodes that store $k$
  - More than $N$ in node preference list for fault tolerance
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Sloppy Quorum

- Quorum-like System: $R + W > N$
  - $N$ - number of replicas
  - $R$ - minimum # of responses for get
  - $W$ - minimum # of responses for put

- Why require $R + W > N$?
  - What is the implication of having a larger $R$?
  - What is the implication of having a larger $W$?
Sloppy Quorum

● “Sloppy quorum”
  ○ Does not enforce strict quorum membership
  ○ Ask first N healthy nodes from preference list
  ○ R and W configurable

● Temporary failure handling
  ○ Do not block waiting for unreachable nodes
  ○ Put should always succeed (set W to 1). Again, always writable.
  ○ Get should have high probability of seeing most recent put(s)
Sloppy Quorum: Conflict Case

Can you come up with a conflict case with the following parameters:

1) \( N = 3, W = 2, W = 2 \)
2) Preference list: B, C, D, E
3) Client0 performs \( \text{put}(k, v) \)
4) Client1 performs \( \text{put}(k, v') \)
Sloppy Quorum: Eventual Consistency

- Allow divergent replica
  - Allow reads to see stale or conflicting data
  - Application can decide the best way to resolve conflict.
  - Resolve multiple versions when failures go away (gossip!)
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Versioning

- **Eventual Consistency**
  - Updates propagates to all replicas asynchronously
  - A put() can return before all replicas update
  - Subsequent get() may return data without latest updates.

- **Versioning**
  - Most recent state is not available, write to a state without latest updates.
  - Treat each modification as a new and immutable version of the data.
  - Uses vector clocks to capture causality between different versions of same data.
System design: Versioning

- Vector clock (node, counter)

- Syntactic Reconciliation:
  - Versions has causal order
  - Pick later version

- Semantic Reconciliation:
  - Versions does not have casual order
  - Client application perform reconciliation.
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership and Failure Detection
Handling permanent failures

- Replica becomes unavailable.
  - Replica synchronization is needed.
Handling permanent failures

- **Anti-entropy (replica synchronization) protocol**
  - Using Merkle trees.

- **Merkle Tree**
  - Leaf node: Hash of data (individual keys)
  - Parent node: hash of children nodes.
  - Efficient data transfer for comparison: Just the root!

\[
H_{AB} = \text{Hash}(H_A + H_B)
\]

\[
H_A = \text{Hash}(A)
\]

\[
H_B = \text{Hash}(B)
\]
Design

- Interface
- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Handling permanent failures
- Membership
Membership: Adding Node

- Explicit mechanism by admin
Membership: Adding Node

- Explicit mechanism by admin
- Propagated via gossip
  - Pull random peer every 1s
Membership: Adding Node

- Explicit mechanism by admin
- Propagated via gossip
  - Pull random peer every 1s
- “Seeds” to avoid partitions
Membership: Detect/Remove Failed Nodes

- Local failure detection
Membership: Detect/Remove Failed Nodes

- Local failure detection
- Use alternative nodes in preference list
Membership: Detect/Remove Failed Nodes

- Local failure detection
- Use alternative nodes in preference list
- Periodic retry
## Design Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent hashing</td>
<td>Scalability</td>
</tr>
<tr>
<td>High availability for writes</td>
<td>Vector clocks; fix conflicts on reads</td>
<td>Users can buy!</td>
</tr>
<tr>
<td>Temporary failures</td>
<td>Sloppy quorum</td>
<td>Users can buy!</td>
</tr>
<tr>
<td>Permanent failures</td>
<td>Merkle trees for anti-entropy</td>
<td>Sync in background</td>
</tr>
<tr>
<td>Membership/failure detection</td>
<td>Gossip</td>
<td>Decentralized</td>
</tr>
</tbody>
</table>
Evaluation

“Experiences & Lessons Learned”
Latency: “Always-on” Experience

- Common SLA goal: 99.9th percentile < 300ms
- Dynamo gets < 200ms!
- Average an order of magnitude less: < 20ms

Average and 99.9% Latency, December 2006
Flexible N, R, W

- The main advantage of Dynamo” is flexible N, R, W
- Many internal Amazon clients, varying parameters
  - (N-R-W)
  - (3-2-2) : default; reasonable R/W performance, durability, consistency
  - (3-3-1) : fast W, slow R, not very durable
  - (3-1-3) : fast R, slow W, durable
Balancing

- “Out-of-balance” if load > 15% off from average

Low loads

20% out-of-balance

High loads

10% out-of-balance
Conclusion

1. Combines well-known systems protocols into highly available database.
2. Achieved reliability at massive-scale.
4. Conflict resolution not an issue in practice.
Acknowledgement

1. Dynamo (DeCandia et al., 2007)