Dynamo

Amazon’s Highly Available Key-value Store
SOSP ’07
Authors

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

Werner Vogels
Cornell → Amazon
Motivation

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

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

“Over 3 million checkouts in a single day” and “hundreds of thousands of concurrently active sessions.”

Reliability can be a problem: “data center being destroyed by tornados”.
Motivation

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

ALL customers have a good experience

Always writeable!
Consequence of “always writeable”

Always writeable $\Rightarrow$ no master! Decentralization; peer-to-peer.

Always writeable + failures $\Rightarrow$ conflicts

CAP theorem: A and P
Amazon’s solution

Sacrifice consistency!
System design: Overview

- Partitioning
- Replication
- Sloppy quorum
- Versioning
- Interface
- Handling permanent failures
- Membership and Failure Detection
System design: Overview

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

Consistent hashing

- The output range of the hash function is a fixed circular space
- Each node in the system is assigned a random position
- Lookup: find the first node with a position larger than the item’s position
- Node join/leave only affects its immediate neighbors
System design: Partitioning

Consistent hashing

- Advantages:
  - Naturally somewhat balanced
  - Decentralized (both lookup and join/leave)
System design: Partitioning

Consistent hashing

- **Problems:**
  - Not really balanced -- random position assignment leads to non-uniform data and load distribution
  - Solution: use virtual nodes

*Figure 2: Partitioning and replication of keys in Dynamo ring.*
**System design: Partitioning**

**Virtual nodes**

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

*Figure 2: Partitioning and replication of keys in Dynamo ring.*
System design: Partitioning

- Benefits
  - Incremental scalability
  - Load balance

Figure 2: Partitioning and replication of keys in Dynamo ring.
System design: Partitioning

- Up to now, we just redefined Chord

Figure 2: Partitioning and replication of keys in Dynamo ring.
System design: Overview

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

- Coordinator node
- Replicas at N - 1 successors
  - N: # of replicas
- Preference list
  - List of nodes that is responsible for storing a particular key
  - Contains more than N nodes to account for node failures

Figure 2: Partitioning and replication of keys in Dynamo ring.
System design: Replication

- Storage system built on top of Chord
- Like Cooperative File System (CFS)

Figure 2: Partitioning and replication of keys in Dynamo ring.
System design: Overview

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

- Temporary failure handling
- Goals:
  - Do not block waiting for unreachable nodes
  - Put should always succeed
  - Get should have high probability of seeing most recent put(s)
  - CAP
System design: Sloppy quorum

- Quorum: $R + W > N$
  - $N$ - first $N$ reachable nodes in the preference list
  - $R$ - minimum # of responses for get
  - $W$ - minimum # of responses for put
- Never wait for all $N$, but $R$ and $W$ will overlap
- “Sloppy” quorum means $R/W$ overlap is not guaranteed
Conflict!

Example:
N=3, R=2, W=2
Shopping cart, empty ""
preference list n1, n2, n3, n4
client1 wants to add item X
  get() from n1, n2 yields ""
  n1 and n2 fail
  put("X") goes to n3, n4
n1, n2 revive
client2 wants to add item Y
  get() from n1, n2 yields ""
  put("Y") to n1, n2
client3 wants to display cart
  get() from n1, n3 yields two values!
  "X" and "Y"
  neither supersedes the other -- conflict!
Eventual consistency

- Accept writes at any replica
- Allow divergent replica
- Allow reads to see stale or conflicting data
- Resolve multiple versions when failures go away (gossip!)
Conflict resolution

- When?
  - During reads
  - Always writeable: cannot reject updates
- Who?
  - Clients
  - Application can decide the best suited method
System design: Overview

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

- Eventual consistency $\Rightarrow$ conflicting versions
- Version number? No; it forces total ordering (Lamport clock)
- Vector clock
System design: Versioning

- Vector clock: version number per key per node.
- List of [node, counter] pairs

Figure 3: Version evolution of an object over time.
System design: Overview

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

- All objects are immutable
- Get(key)
  - may return multiple versions
- Put(key, context, object)
  - Creates a new version of key
System design: Overview

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

- Detect inconsistencies between replicas
- Synchronization

Figure 2: Partitioning and replication of keys in Dynamo ring.
System design: Handling permanent failures

- Anti-entropy replica synchronization protocol
- Merkle trees
  - A hash tree where leaves are hashes of the values of individual keys; nodes are hashes of their children
  - Minimize the amount of data that needs to be transferred for synchronization

\[
H_{\text{ABCD}} \leftarrow \text{Hash}(H_{\text{AB}} + H_{\text{CD}})
\]

\[
H_{\text{AB}} \leftarrow \text{Hash}(H_{\text{A}} + H_{\text{B}})
\]

\[
H_{\text{CD}} \leftarrow \text{Hash}(H_{\text{C}} + H_{\text{D}})
\]

\[
H_{\text{A}} \leftarrow \text{Hash}(A)
\]

\[
H_{\text{B}} \leftarrow \text{Hash}(B)
\]

\[
H_{\text{C}} \leftarrow \text{Hash}(C)
\]

\[
H_{\text{D}} \leftarrow \text{Hash}(D)
\]
System design: Overview

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

- Gossip-based protocol propagates membership changes
- External discovery of seed nodes to prevent logical partitions
- Temporary failures can be detected through timeout
### System design: Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>
Evaluation?

No real evaluation; only experiences
Experiences: Flexible N, R, W and impacts

- They claim “the main advantage of Dynamo” is flexible N, R, W
- What do you get by varying them?
  - (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
Experiences: Latency

- 99.9th percentile latency: ~200ms
- Avg latency: ~20ms
- “Always-on” experience!

Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages.
Experiences: Load balancing

- Out-of-balance: 15% away from average load
- High loads: many popular keys; load is evenly distributed; fewer out-of-balance nodes
- Low loads: fewer popular keys; more out-of-balance nodes

Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.
Conclusion

- Eventual consistency
- Always writeable despite failures
- Allow conflicting writes, client merges
Questions?