CSE 486/586 Distributed Systems
Case Study: Amazon Dynamo

Slides by Steve Ko
Computer Sciences and Engineering
University at Buffalo
Amazon Dynamo

• Distributed key-value storage
  – Objects are accessible only by their (unique) key
  – Provides put(key, value) and get(key) operations

• Used for many Amazon services
  – Shopping cart, best seller lists, customer preferences, product catalog, etc.
  – Now in AWS as well (DynamoDB) [1]

• With some Google systems (GFS & Bigtable), Dynamo marks one of the first non-relational storage systems (a.k.a. NoSQL)
Amazon Dynamo

• A synthesis of many techniques we’ve discussed in class
  – Very good example of developing a principled distributed system
  – Comprehensive picture of the design of a distributed storage system
• Main motivation: shopping cart service
  – 3 million checkouts in a single day
  – Hundreds of thousands of concurrent active sessions
• Properties (in the CAP theorem sense)
  – Eventual consistency
  – Partition tolerance
  – High Availability
Necessary Pieces?

- We want to design a storage service on a cluster of servers
- What might we need?
  - Membership maintenance
  - Object insert/lookup/delete
  - (Some) Consistency with replication
  - Partition tolerance
- Dynamo is a good example of a working system.
Overview of Key Design Techniques

- **Gossiping** for membership and failure detection
  - Eventually-consistent membership
- **Consistent hashing** for node & key distribution
  - Similar to Chord
  - But there’s no ring-based routing; everyone knows everyone else
- **Object versioning** for eventually-consistent data objects
  - A vector clock associated with each object
- **Quorums** for partition/failure tolerance
  - Called “sloppy” quorum
- **Merkel tree** for resynchronization after failures/partitions
  - (This has not yet been covered in class)
Membership

- Nodes are organized in a ring using **consistent hashing**
- **Everyone knows everyone else**, unlike Chord.
- **Node join/leave**
  - Manual operation; an operator uses a console to add/delete a node
  - Reason: it’s a well-maintained system; nodes come back pretty quickly and usually don’t depart permanently
- **Membership change propagation**
  - Each node maintains its own view of membership and a history of membership changes
  - Propagated using gossiping (every second, pick random targets)
- **Eventually-consistent membership protocol**
Failure Detection

- **Does not use a separate protocol**; each request serves as a ping for failure detection
  - Dynamo has enough requests for this at any given moment
- If a node doesn’t respond to a request, it is considered to be failed.
Node & Key Distribution

- Distribution by consistent hashing
- Load may become uneven
  - With a small number of nodes and/or as nodes come and go, partition sizes may become uneven.
Node & Key Distribution

• Uses consistent hashing with “virtual nodes” for better load balancing

• Starts with a static number of virtual nodes uniformly distributed over the ring
Node & Key Distribution

- One node joins and gets all virtual nodes

Node 1
Node & Key Distribution

- A second node joins and takes $\frac{1}{2}$ of the virtual nodes

• Node 1
• Node 2
Node & Key Distribution

• One more node joins and gets $1/3$ (roughly) of the virtual nodes from the first two

- Node 1
- Node 2
- Node 3
Replication

• Each object is stored on N replicas
• The first is stored normally with consistent hashing
• N-1 replicas are stored in the N-1 (physical) successor nodes (called preference list)

![Diagram showing nodes A, B, C, D, E, F, and G with nodes B, C, and D storing keys in range (A,B) including K.](image-url)
Replication

• **Any server** can handle read/write in the preference list for an object, but it **walks over the ring**
  – *E.g.*, try B first, then C, then D, *etc.*

• **Update propagation** is handled by the server that served the request

![Diagram of nodes and key K]

Nodes B, C and D store keys in range (A,B) including K.
Replication

• Dynamo replication is lazy.
  – A put() request is returned “right away” (more on this later); it does not wait until the update is propagated to the replicas.
  – As long as there’s one reachable server, a write completes.
  – This could lead to inconsistency

Key K

Nodes B, C and D store keys in range (A,B) including K.
Object Versioning

- Writes should always succeed
  - *E.g.*, “Add to Cart” as long as there’s at least one reachable server
- **Object versioning** is used to reconcile inconsistency.
- Each object has a vector clock
  - *E.g.*, D1 ([Sx, 1], [Sy, 1]): Object D (version 1) was written once by server Sx and Sy.
  - Each node keeps all versions until the data becomes consistent
  - *I.e.*, no overwrite; almost like each write creates a new object
- **Causally concurrent versions** lead to inconsistency
  - *I.e.*, there are writes that are not causally related.
- If an object is inconsistent, reconcile it later.
  - *E.g.*, deleted items might reappear in the shopping cart.
Object Versioning

• Example

```
D1 ([Sx,1])

write
handled by Sx

D2 ([Sx,2])

write
handled by Sy

write
handled by Sz

D3 ([Sx,2],[Sy,1])

D4 ([Sx,2],[Sz,1])

reconciled
and written by
Sx

D5 ([Sx,3],[Sy,1][Sz,1])
```
Conflict Detection & Resolution

- Object versioning gives clients the ability to detect write conflicts.
- Reconciliation
  - Simple resolution done by the system (last-write-wins policy)
  - Complex resolution done by individual applications: The system presents all conflicting versions of data to an application.
Object Versioning Experience

Over a 24-hour period:
• 99.94% of requests saw exactly one version
• 0.00057% saw 2 versions
• 0.00047% saw 3 versions
• 0.00009% saw 4 versions

Multiple versions were usually triggered by many concurrent requests issued by robots, not human clients.
Quorums

• Parameters
  – N replicas, R readers, W writers
• Static quorum approach: R + W > N, W > N / 2
• Typical Dynamo configuration: (N, R, W) == (3, 2, 2)
• But it depends
  – High performance read (e.g., write-once, read-many data):
    • R==1, W==N
  – Low R & W might lead to more inconsistency
• Dealing with failures
  – Another node in the preference list handles requests temporarily
  – Replicas are delivered to the failed node upon recovery
Replica Synchronization

• Key ranges are replicated.
• Suppose a node fails and recovers; it needs to quickly determine whether it needs to resynchronize or not.
  – Transferring all (key, value) pairs for comparison is not an option
• Merkel trees
  – Leaf values are hashes of the values of individual keys
  – Parent values are hashes of their (immediate) children
  – Comparing parents at the same level identifies differences in children
  – Does not require transferring entire (key, value) pairs
Replica Synchronization

• Comparing two nodes that are synchronized
  – Two <key, value> pairs: <k0, v0> & <k1, v1>

Node0

- h0 = hash(v0)
- h1 = hash(v1)
- h2 = hash(h0 + h1)

Node1

- h0 = hash(v0)
- h1 = hash(v1)
- h2 = hash(h0 + h1)

Equal
Replica Synchronization

• Comparing two nodes that are not synchronized
  – One has <k0, v2> and <k1, v1>
  – The other has <k0, v0> & <k1, v1>

\[
\begin{align*}
  h_3 &= \text{hash}(v_2) \\
  h_1 &= \text{hash}(v_1) \\
  h_4 &= \text{hash}(h_2 + h_1) \\
  h_2 &= \text{hash}(h_0 + h_1) \\
  h_0 &= \text{hash}(v_0) \\
  h_1 &= \text{hash}(v_1)
\end{align*}
\]

Node 0 and Node 1 are not equal.
Summary

- **Amazon Dynamo**
  - Distributed key-value storage with *eventual consistency*

- **Techniques**
  - *Gossiping* for membership and failure detection
  - *Consistent hashing* for node & key distribution
  - *Object versioning* for eventually-consistent data objects
  - *Quorums* for partition/failure tolerance
  - *Merkel tree* for resynchronization after failures/partitions

- Very good example of developing a principled distributed system
References

[1] Werner Vogels. *Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications*. From the All Things Distributed weblog. **Required Reading.**
https://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html
Acknowledgements

• These slides are by Steve Ko, lightly modified and used with permission by Ethan Blanton.
• These slides contain material developed and copyrighted by Indranil Gupta (UIUC).