CSE 486/586 Distributed Systems
Case Study: Amazon Dynamo

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap

• CAP Theorem?
  – Consistency, Availability, Partition Tolerance
  – P then C? A?
• Eventual consistency?
  – Availability and partition tolerance over consistency

Amazon Dynamo

• Distributed key-value storage
  – Only accessible with the primary key
  – put(key, value) & get(key)
• Used for many Amazon services (“applications”)
  – Shopping cart, best seller lists, customer preferences, product catalog, etc.
  – Now in AWS as well (DynamoDB) (if interested, read
    http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html)
• With other Google systems (GFS & Bigtable), Dynamo marks one of the first non-relational storage systems (a.k.a. NoSQL)

Amazon Dynamo

• A synthesis of techniques we discuss in class
  – Very good example of developing a principled distributed system
  – Comprehensive picture of what it means to design 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
  – Availability ("always-on" experience)

Necessary Pieces?

• We want to design a storage service on a cluster of servers
  • What do we need?
    – Membership maintenance
    – Object insert/lookup/delete
    – (Some) Consistency with replication
    – Partition tolerance
  • Dynamo is a good example as 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
• Merkle tree for resynchronization after failures/partitions
  – (This was not covered in class yet)
Membership

- Nodes are organized as a ring just like Chord using consistent hashing.
- But everyone knows everyone else.
- Node join/leave
  - Manually done
  - An operator uses a console to add/delete a node
  - Reason: it's a well-maintained system; nodes come back pretty quickly and don't depart permanently most of the time
- Membership change propagation
  - Each node maintains its own view of the membership & the history of the 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
  - Dynamo has enough requests at any moment anyway
- If a node doesn’t respond to a request, it is considered to be failed.

Node & Key Distribution

- Original consistent hashing
- Load becomes uneven
  - With a small number of nodes and/or as nodes come and go, each partition size becomes uneven.

Node & Key Distribution

- Consistent hashing with “virtual nodes” for better load balancing
- Start with a static number of virtual nodes uniformly distributed over the ring

Node & Key Distribution

- One node joins and gets all virtual nodes

Node & Key Distribution

- One more node joins and gets 1/2
Node & Key Distribution
- One more node joins and gets 1/3 (roughly) from the other two

Replication
- N: # of replicas; configurable
- The first is stored regularly with consistent hashing
- N-1 replicas are stored in the N-1 (physical) successor nodes (called preference list)

Replication
- Any server can handle read/write in the preference list, but it walks over the ring
  - E.g., try B first, then C, then D, etc.
  - Update propagation: by the server that handled the request

Replication
- Dynamo's 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 is done.
  - This could lead to inconsistency

Object Versioning
- Writes should succeed all the time
  - 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) has written once by server Sx and Sy.
    - Each node keeps all versions until the data becomes consistent
    - I.e., no overwrite, each write creates a new object
  - Causally concurrent versions: inconsistency
    - I.e., there are writes not causally related.
  - If inconsistent, reconcile later.
    - E.g., deleted items might reappear in the shopping cart.

CSE 486/586 Administrivia
- Final: 5/18/2017, Thursday, 6 pm – 8 pm, Knox 110
Object Versioning

- Example

```
D1 ([Sx, 1])
  write
  handled by Sx

D2 ([Sx, 1])
  write
  handled by Sx

D3 ([Sx, 2], [Sy, 1])
  write
  handled by Sx

D4 ([Sx, 2], [Sy, 1])
  read

D5 ([Sx, 3], [Sy, 1], [Sz, 1])
  read

D6 ([Sx, 3], [Sy, 1], [Sz, 1])
  write
  handled by Sz
```

Conflict Detection & Resolution

- Object versioning gives the ability to detect write conflicts.
- Reconciliation
  - Simple resolution done by the system (last-write-wins policy)
  - Complex resolution done by each application: 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
- 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
- Typical Dynamo configuration: (N, R, W) == (3, 2, 2)
- But it depends
  - High performance read (e.g., write-once, read-many): R==1, W==N
  - Low R & W might lead to more inconsistency
- Dealing with failures
  - Another node in the preference list handles the requests temporarily
  - Delivers the replicas to the original node upon recovery

Replica Synchronization

- Key ranges are replicated.
- Say, a node fails and recovers, a node needs to quickly determine whether it needs to resynchronize or not.
  - Transferring entire (key, value) pairs for comparison is not an option
- Merkel trees
  - Leaves are hashes of values of individual keys
  - Parents are hashes of (immediate) children
  - Comparison of parents at the same level tells the difference 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)

\[ h_0 = \text{hash}(v_0) \]
\[ h_1 = \text{hash}(v_1) \]
\[ h_2 = \text{hash}(h_0 + h_1) \]

Node0

\[ h_0 = \text{hash}(v_0) \]
\[ h_1 = \text{hash}(v_1) \]
\[ h_2 = \text{hash}(h_0 + h_1) \]

Node1

Equal
Replica Synchronization

- Comparing two nodes that are not synchronized
  - One: $(k_0, v_2)$ & $(k_1, v_1)$
  - The other: $(k_0, v_0)$ & $(k_1, v_1)$

```

h4 = hash(h2 + h1)
Not equal
h2 = hash(h0 + h1)

h3 = hash(v2)
h1 = hash(v1)

h0 = hash(v0)
h1 = hash(v1)

Node0               Node1
```

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
  - Merkle tree for resynchronization after failures/partitions
- Very good example of developing a principled distributed system

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).