Recap

- CAP Theorem?
  - Consistency, Availability, Partition Tolerance
  - P then C? A?
- Eventual consistency?
  - Availability and partition tolerance over consistency
- Lazy replication?
  - Replicate lazily in the background
- Gossiping?
  - Contact random targets, infect, and repeat in the next round

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 [link](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)
- A synthesis of techniques we discuss in class
  - Well, not all but mostly
  - 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
  - "Sloppy" quorum similar to the available copies replication strategy
- Merkle tree for resynchronization after failures/partitions
  - (This was not covered in class)
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

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

![Diagram](image1.png)

CSE 486/586 Administrivia

• PA4 will be released soon.

![Diagram](image2.png)

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)

![Diagram](image3.png)

Replication

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

![Diagram](image4.png)

Object Versioning

• Writes should succeed all the time
  – E.g., “Add to Cart”
• Used to reconcile inconsistent data
• Each object has a vector clock
  – E.g., D1 ([Sx, 1], [Sy, 1]): Object D1 has written once by server Sx and Sy.
  – Each node keeps all versions until the data becomes consistent
• Causally concurrent versions: inconsistency
• If inconsistent, reconcile later.
  – E.g., deleted items might reappear in the shopping cart.
Object Versioning

- Example

\begin{center}
\begin{tikzpicture}
  \node (D1) [circle, fill=gray!20] at (0,0) {$D_1$};
  \node (D2) [circle, fill=gray!20] at (0,-1) {$D_2$};
  \node (D3) [circle, fill=gray!20] at (0,-2) {$D_3$};
  \node (D4) [circle, fill=gray!20] at (0,-3) {$D_4$};
  \node (D5) [circle, fill=gray!20] at (0,-4) {$D_5$};

  \draw[->] (D1) to node [above] {write, handled by $S_1$} (D2);
  \draw[->] (D2) to node [above] {write, handled by $S_2$} (D3);
  \draw[->] (D3) to node [above] {write, handled by $S_2$} (D4);
  \draw[->] (D4) to node [above] {write, handled by $S_3$} (D5);
  \draw[->] (D5) to node [above] {reconciled and written by $S_3$} (D1);
\end{tikzpicture}
\end{center}

Object Versioning

- Consistency revisited
  - Linearizability: any read operation reads the latest write.
  - Sequential consistency: for each client, any read operation reads
    the latest write.
  - Eventual consistency: a read operation might not read the
    latest write & sometimes inconsistent versions need to be
    reconciled.

- Conflict detection & resolution required
  - Dynamo uses vector clocks to detect conflicts
  - Simple resolution done by the system (last-write-wins
    policy)
  - Complex resolution done by each application
    - System presents all conflicting versions of data

Conflict detection & resolution required

- Dynamo uses vector clocks to detect conflicts
- Simple resolution done by the system (last-write-wins policy)
- Complex resolution done by each application
  - System presents all conflicting versions of data

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 busy 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

Comparing two nodes that are synchronized

- Two (key, value) pairs: $(k_0, v_0)$ & $(k_1, v_1)$

- $h_0 = \text{hash}(v_0)$
- $h_1 = \text{hash}(v_1)$

- $h_2 = \text{hash}(h_0 + h_1)$

- $h_0 = \text{hash}(v_0)$
- $h_1 = \text{hash}(v_1)$

- $h_2 = \text{hash}(h_0 + h_1)$

- Equal

Node0

Node1
Replica Synchronization

• Comparing two nodes that are not synchronized
  – One: (k0, v2) & (k1, v1)
  – The other: (k0, v0) & (k1, v1)

\[
\begin{align*}
h_3 &= \text{hash}(v2) \\
h_1 &= \text{hash}(v1) \\
h_4 &= \text{hash}(h_2 + h_1) \\
h_0 &= \text{hash}(v0) \\
h_2 &= \text{hash}(h_0 + h_1)
\end{align*}
\]

Not equal

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).