Peer-to-Peer in the Datacenter: Amazon Dynamo

Mike Freedman
COS 461: Computer Networks

http://www.cs.princeton.edu/courses/archive/spr14/cos461/

This Lecture...

Amazon’s “Big Data” Problem

• Too many (paying) users!
  — Lots of data
• Performance matters
  — Higher latency = lower “conversion rate”
• Scalability: retaining performance when large
For 50K machines, with online time of 99.99966%:

- 16% of the time, data center experiences failures
- For 100K machines, 30% of the time!

**Dynamo Requirements**

- **High Availability**
  - Always respond quickly, even during failures
  - Replication!

- **Incremental Scalability**
  - Adding “nodes” should be seamless

- **Comprehensible Conflict Resolution**
  - High availability in above sense implies conflicts
### Dynamo Design

- **Key-Value Store via DHT over data nodes**
  - get(k) and put(k, v)
- **Questions:**
  - Replication of Data
  - Handling Requests in Replicated System
  - Temporary and Permanent Failures
  - Membership Changes

### Data Partitioning and Data Replication

- **Familiar?**
  - Nodes are virtual!
- **Replication:**
  - Coordinator Node
  - N-I successors also
  - Nodes keep preference list

### Handling Requests

- **Request coordinator consults replicas**
  - How many?
- **Forward to N replicas from preference list**
  - R or W responses form a read/write quorum
- **Any of top N in pref list can handle req**
  - Load balancing & fault tolerance

### Detecting Failures

- **Purely Local Decision**
  - Node A may decide independently that B has failed
  - In response, requests go further in preference list
- **A request hits an unsuspecting node**
  - “temporary failure” handling occur
**Handling Temporary Failures**

- **E is in replica set**
  - Needs to receive replica
  - Hinted Handoff: replica contains “original” node
- **When C comes back**
  - E forwards the replica back to C

**Managing Membership**

- **Peers randomly tell another their known membership history – “gossiping”**
- **Also called epidemic algorithm**
  - Knowledge spreads like a disease through system
  - Great for ad hoc systems, self-configuration, etc.
  - Does this make sense in Amazon’s environment?

**Gossip could partition the ring**

- **Possible Logical Partitions**
  - A and B choose to join ring at about same time: Unaware of one another, may take long time to converge to one another
- **Solution:**
  - Use seed nodes to reconcile membership views: Well-known peers that are contacted frequently

**Why is Dynamo Different?**

- So far, looks a lot like normal p2p
- Amazon wants to use this for application data!
- Lots of potential synchronization problems
- Uses versioning to provide eventual consistency.
## Consistency Problems

- **Shopping Cart Example:**
  - Object is a history of “adds” and “removes”
  - *All adds* are important (trying to make money)

<table>
<thead>
<tr>
<th>Client:</th>
<th>Expected Data at Server:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put(k, [+1 Banana])</td>
<td>[+1 Banana]</td>
</tr>
<tr>
<td>Z = get(k)</td>
<td></td>
</tr>
<tr>
<td>Put(k, Z + [+1 Banana])</td>
<td>[+1 Banana, +1 Banana]</td>
</tr>
<tr>
<td>Z = get(k)</td>
<td></td>
</tr>
<tr>
<td>Put(k, Z + [-1 Banana])</td>
<td>[+1 Banana, +1 Banana, -1 Banana]</td>
</tr>
</tbody>
</table>

## What if a failure occurs?

<table>
<thead>
<tr>
<th>Client:</th>
<th>Data on Dynamo:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put(k, [+1 Banana])</td>
<td>[+1 Banana] at A</td>
</tr>
<tr>
<td>Z = get(k)</td>
<td>A Crashes</td>
</tr>
<tr>
<td>Put(k, Z + [+1 Banana])</td>
<td>B not in first Put’s quorum</td>
</tr>
<tr>
<td>Z = get(k)</td>
<td>[+1 Banana] at B</td>
</tr>
<tr>
<td>Put(k, Z + [-1 Banana])</td>
<td>[+1 Banana, -1 Banana] at B</td>
</tr>
</tbody>
</table>

Node A Crashes

At this point, Node A and B disagree about object state

- How is this resolved?
- Can we even tell a conflict exists?

## “Time” is largely a human construct

- **What about time-stamping objects?**
  - Could authoritatively say whether object newer or older?
  - But, *all events are not necessarily witnessed*

- **If system’s notion of time corresponds to “real-time”...**
  - New object always blasts away older versions
  - Even though those versions may have important updates (as in bananas example).

- **Requires a new notion of time (causal in nature)**

- **Anyhow, real-time is impossible in any case**

## Causality

- **Objects are causally related if value of one object depends on (or witnessed) the previous**

- **Conflicts can be detected when replicas contain causally independent objects for a given key**

- **Notion of time which captures causality?**
Versioning

- Key Idea: Every PUT includes a version, indicating most recently witnessed version of updated object
- Problem: replicas may have diverged
  - No single authoritative version number (or "clock" number)
  - Notion of time must use a partial ordering of events

Vector Clocks

- Every replica has its own logical clock
  - Incremented before it sends a message
- Every message attached with vector version
  - Includes originator's clock
  - Highest seen logical clocks for each replica
- If $M_1$ is causally dependent on $M_0$:
  - Replica sending $M_1$ will have seen $M_0$
  - Replica will have seen clocks ≥ all clocks in $M_0$

Vector Clocks in Dynamo

- Vector clock per object
- get() returns obj's vector clock
- put() has most recent clock
  - Coordinator is "originator"
- Serious conflicts are resolved by app / client

Vector Clocks in Banana Example

Client:

- Put(k, [+1 Banana])
- Z = get(k)
- Put(k, Z + [-1 Banana])

Data on Dynamo:

- $v = [(B,1)]$ at B
  - $B$ not in first Put's quorum
- $v = [(B,2)]$ at B
  - $A$ comes online

$[(A,1)]$ and $[(B,2)]$ are a conflict!
**Eventual Consistency**

- Versioning, by itself, does not guarantee consistency
  - If you don’t require a majority quorum, you need to periodically check that peers aren’t in conflict
  - How often do you check that events are not in conflict?

- In Dynamo:
  - Nodes consult with one another using a tree hashing (Merkel tree) scheme
  - Quickly identify whether they hold different versions of particular objects and enter conflict resolution mode

---

**NoSQL**

- Notice that Eventual Consistency and Partial Orderings do not give you ACID!

- Rise of NoSQL (outside of academia)
  - Memcache
  - Cassandra
  - Redis
  - Big Table
  - MongoDB