Scaling Out Key-Value Storage and Dynamo

COS 418: Distributed Systems
Lecture 10
Mike Freedman

Availability: vital for web applications

- Web applications are expected to be “always on”
  - Down time \(\rightarrow\) pisses off customers, costs $\$

- System design considerations relevant to availability
  - **Scalability**: always on under growing demand
  - **Reliability**: always on despite failures
  - **Performance**: 10 sec latency considered available?
    - “an availability event can be modeled as a long-lasting performance variation” (Amazon Aurora SIGMOD ’17)

Scalability: up or out?

- **Scale-up** (vertical scaling)
  - Upgrade hardware
  - E.g., Macbook Air \(\rightarrow\) Macbook Pro
  - Down time during upgrade; stops working quickly

- **Scale-out** (horizontal scaling)
  - Add machines, divide the work
  - E.g., a supermarket adds more checkout lines
  - No disruption; works great with careful design

Reliability: available under failures

- More machines, more likely to fail
  - \(p = \text{probability one machine fails}; n = \text{# of machines}\)
  - Failures happen with a probability of \(1 - (1 - p)^n\)

- For 50K machines, each with 99.99966% available
  - 16% of the time, data center experiences failures

- For 100K machines, failures happen 30% of the time!
Two questions (challenges)

• How is data partitioned across machines so the system scales?
• How are failures handled so the system is always on?

Today: Amazon Dynamo

1. Background and system model
2. Data partitioning
3. Failure handling

Amazon in 2007

• 10⁶ s of servers in multiple datacenters
  – 10⁶ s of servers, 80+ DCs (as of now)
• 10⁷ s of customers at peak times
  – 20M+ purchases in US. (Prime Day 2020)
• Tiered architecture (similar today)
  – Stateless web servers & aggregators
  – Stateful storage servers

Basics in Dynamo

• A key-value store (vs. relational DB)
  – get(key) and put(key, value)
  – Nodes are symmetric
  – Remember DHT?
• Service-Level Agreement (SLA)
  – E.g., “provide a response within 300ms for 99.9% of its requests for peak client load of 500 requests/sec”
Today: Amazon Dynamo

1. Background and system model

2. Data partitioning
   - Incremental scalability
   - Load balancing

3. Failure handling

Consistent hashing recap

Identifiers have $m = 3$ bits
Key space: $[0, 2^3 - 1]$

- Identifiers/key space
- Node
- Stores key 1
- Stores keys 6, 5
- Stores keys 4, 3
- Stores keys 2, 3

Key is stored at its successor: node with next-higher ID

Incremental scalability (why consistent hashing)

- Minimum data is moved around when nodes join and leave
- Please try modular hashing and see the difference

Challenge: unbalanced load

- Nodes are assigned different # of keys
Challenge: unbalanced load
- Nodes are assigned different # of keys
- Unbalanced with nodes join/leave

Solution: virtual nodes
- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

- Some keys are more popular
Solution: virtual nodes

- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

3 physical nodes (servers)
2 vnodes / server

3-bit ID space

Identifiers/key space

Virtual node: same color → same physical node

Gold server leaves
Keys moved to blue and red

Solution: virtual nodes

- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

3-bit ID space

Identifiers/key space

Virtual node: same color → same physical node

Today: Amazon Dynamo

1. Background and system model
2. Data partitioning
3. Failure handling
   - Data replication
• Key replicated on M vnodes
  – Remember “r-successor” in DHT?
• All M vnodes on distinct servers across different datacenters

Preference list (data replication)

Read and write requests
• Received by the coordinator
  – Either the client (web server) knows the mapping or re-routed. (This is not Chord)
• Sent to the first N “healthy” servers in the preference list (coordinator included)
  – Durable writes: my updates recorded on multiple servers
  – Fast reads: possible to avoid straggler
• A write creates a new immutable version of the key instead of overwriting it
  – Multi-versioned data store
• Quorum-based protocol: \( W + R > N \)
  – A write succeeds if \( W \) out of \( N \) servers reply (write quorum)
  – A read succeeds if \( R \) out of \( N \) servers reply (read quorum)

Quorum implications (\( W, R, \) and \( N \))
• \( N \) determines the durability of data (Dynamo \( N = 3 \))
• \( W \) and \( R \) plays around with the availability-consistency tradeoff
  – \( W = 1 \) (\( R = 3 \)): fast write, weak durability, slow read (read availability)
  – \( R = 1 \) (\( W = 3 \)): slow write (write availability), good durability, fast read
  – Dynamo: \( W = R = 2 \)
• Why \( W + R > N \)?
  – Read and write quorums overlap when there are no failures!
  – Reads see all updates without failures
    – What if there are failures?
Failure handing: sloppy quorum + hinted handoff

- Sloppy: not always the same servers used in N
  - First N servers in the preference list without failures
  - Later servers in the list take over if some in the first N fail

- Consequences
  - Good performance: no need to wait for failed servers in N to recover
  - Eventual (weak) consistency: conflicts are possible, versions diverge
  - Another decision on availability-consistency tradeoff!

An example of conflicting writes (versions)

1. Shopping cart:
   - CL1: Add Item x
   - A and B fail

2. Time

3. Preference list (M = 5, N = 3)
   - [A B C D E]
   - [x x]

An example of conflicting writes (versions)

1. Shopping cart:
   - CL1: Add Item x
   - A and B fail
   - CL2: Add Item y

2. Time

3. Preference list (M = 5, N = 3)
   - [A B C D E]
   - [x x]
An example of conflicting writes (versions)

Preference list (M = 5, N = 3)

Shopping cart: [A, B, C, D, E]

CL1: Add Item x
x x
A and B fail

CL2: Add Item y
y y
A and B recover

CL1: Read cart
read read

Conflicting versions only possible under failures.

Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Shopping cart: [A, B, C, D, E]

CL1: Add Item x
A.1 A.1
A and B fail

CL2: Add Item y
y y
A.1 A.1
A and B recover

CL1: Read cart
read read

Can we use Lamport clocks?

Read returns x(A.1) and y(C.1)
A.1 and C.1 are not causally related: conflicts!

Conflict resolution (reconciliation)

- If vector clocks show causally related (not really conflicting)
  - System overwrites with the later version

- For conflicting versions
  - System handles it automatically, e.g., last-writer-wins, limited use case
  - Application specific resolution (most common)
    - Clients resolve the conflict via reads, e.g., merge shopping cart

Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Shopping cart: [A, B, C, D, E]

CL1: Add Item x
x x
A.1 A.1
A and B fail

CL2: Add Item y
y y
A.1 A.1
A and B recover

CL1: Read cart
read read

Read returns x(A.1) and y(C.1)
A.1 and C.1 are not causally related: conflicts!
Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Shopping cart:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL1: Add Item x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL2: Add Item y</td>
<td></td>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>CL1: Read cart</td>
<td></td>
<td></td>
<td>C.1</td>
<td>C.1</td>
<td></td>
</tr>
<tr>
<td>CL1: Add Item z</td>
<td>x, y, z (A.1, C.1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

33

Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Shopping cart:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL1: Add Item x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL2: Add Item y</td>
<td></td>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>CL1: Read cart</td>
<td></td>
<td></td>
<td>C.1</td>
<td>C.1</td>
<td></td>
</tr>
<tr>
<td>CL1: Add Item z</td>
<td>x, y, z (A.1, C.1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

34

Anti-entropy (replica synchronization)

- Each server keeps one Merkle tree per virtual node (a range of keys)
  - A leaf is the hash of a key's value: # of leaves = # keys on the virtual node
  - An internal node is the hash of its children
- Replicas exchange trees from top down, depth by depth
  - If root nodes match, then identical replicas, stop
  - Else, go to next level, compare nodes pair-wise

Failure detection and ring membership

- Server A considers B has failed if B does not reply to A's message
  - Even if B replies to C
  - A then tries alternative nodes
  - With servers join and permanently leave
- Servers periodically send gossip messages to their neighbors to sync who are in the ring
  - Some servers are chosen as seeds, i.e., common neighbors to all nodes
Conclusion

- Availability is important
  - Systems need to be scalable and reliable

- Dynamo is eventually consistent
  - Many design decisions trade consistency for availability

- Core techniques
  - Consistent hashing: data partitioning
  - Preference list, sloppy quorum, hinted handoff: handling transient failures
  - Vector clocks: conflict resolution
  - Anti-entropy: synchronize replicas
  - Gossip: synchronize ring membership