Scaling KVS

CS6450: Distributed Systems
Lecture 13

Ryan Stutsman
Horizontal or vertical scalability?

Vertical Scaling

Horizontal Scaling
Horizontal scaling is chaotic

- Probability of any failure in given period = $1-(1-p)^n$
  - $p = \text{probability a machine fails in given period}$
  - $n = \text{number of machines}$

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

- For 100K machines, failures 30% of the time!
Today

1. Techniques for partitioning data
   • Metrics for success

2. Case study: Amazon Dynamo key-value store
Scaling out: Partition and place

• **Partition management**
  • Including how to recover from node failure
    • e.g., bringing another node into partition group
  • Changes in system size, *i.e.* nodes joining/leaving

• **Data placement**
  • **On which node(s) to place** a partition?
    • Maintain mapping from data object to responsible node(s)

• **Centralized:** Cluster manager
• **Decentralized:** Deterministic hashing
Partitioning/Mapping Keys: Modulo hash

• Consider problem of data partition:
  • Given object id X, choose one of k servers to use

• Suppose instead we use modulo hashing:
  • Place X on server \( i = \text{hash}(X) \mod k \)

• What happens if a server fails or joins \( (k \leftarrow k \pm 1) \)?
  • or different clients have different estimate of k?
Problem: Changing number of servers

\[ h(x) = x + 1 \pmod{4} \]

Add one machine: \[ h(x) = x + 1 \pmod{5} \]

All entries get \textit{remapped} to new nodes!
→ Need to \textit{move} objects over the network
Consistent hashing

- Assign \( n \) tokens to random points on \( \mod 2^k \) circle; hash key size = \( k \)
- Hash object to random circle position
- Put object in closest clockwise bucket
  - successor (key) \( \rightarrow \) bucket

• Desired features –
  - Balance: No bucket has “too many” objects
  - Smoothness: Addition/removal of token minimizes object movements for other buckets
Consistent hashing’s load balancing problem

• Each node owns $1/n^{th}$ of the ID space in expectation
  • Says nothing of request load per bucket

• If a node fails, its successor takes over bucket
  • Smoothness goal: Only localized shift, not O(n)

• But now successor owns two buckets: $2/n^{th}$ of key space
  • The failure has upset the load balance
Virtual nodes

• **Idea:** Each physical node now maintains $v > 1$ tokens
  • Each token corresponds to a virtual node

• Each virtual node owns an expected $\frac{1}{(vn)^{th}}$ of ID space

• **Upon a physical node’s failure,** $v$ successors take over, each now stores $\frac{(v+1)}{vn^{ths}}$ of ID space (compared to $\frac{v}{vn}$ before)

• **Result:** Better load balance with larger $v$
Today

1. Techniques for partitioning data

2. Case study: the Amazon Dynamo key-value store
Dynamo: The P2P context

• **Chord** and **DHash** intended for **wide-area P2P systems**
  • Individual nodes at Internet’s edge, file sharing

• **Central challenges:** low-latency key lookup with small forwarding state per node

• **Techniques:**
  • **Consistent hashing** to map keys to nodes
  • **Replication** at successors for availability under failure
Amazon’s workload (in 2007)

- **Tens of thousands** of servers in globally-distributed **data centers**
- **Peak load:** Tens of millions of customers
- **Tiered** service-oriented architecture
  - **Stateless** web page rendering servers, atop
  - **Stateless** aggregator servers, atop
  - **Stateful** data stores (e.g. **Dynamo**)
    - `put( )`, `get( )`: values “usually less than 1 MB”
How does Amazon use Dynamo?

- **Shopping cart**
- **Session info**
  - Maybe “recently visited products” etc.?
- **Product list**
  - Mostly read-only, replication for high read throughput
Dynamo requirements

• **Highly available writes** despite failures
  • Despite disks failing, network routes flapping, “data centers destroyed by tornadoes”
  • Always respond quickly, even during failures → replication

• **Low request-response latency**: focus on 99.9% SLA

• **Incrementally scalable** as servers grow to workload
  • Adding “nodes” should be seamless

• Comprehensible **conflict resolution**
  • High availability in above sense implies conflicts

**Non-requirement:** Security, authentication, authorization (used in a non-hostile environment)
Design questions

• How is data placed and replicated?

• How are requests routed and handled in a replicated system?

• How to cope with temporary and permanent node failures?
Dynamo’s system interface

• Basic interface is a key-value store
  • `get(k)` and `put(k, v)`
  • Keys and values opaque to Dynamo

• `get(key) → [value], context`
  • Returns one value or multiple conflicting values
  • Context describes version(s) of value(s)

• `put(key, context, value) → “OK”`
  • Context indicates which versions this version supersedes or merges
Dynamo’s techniques

• **Place** replicated data on nodes with **consistent hashing**

• Maintain consistency of replicated data with **vector clocks**
  - **Eventual consistency** for replicated data: prioritize success and low latency of writes over reads
    - And availability over consistency (unlike DBs)

• Efficiently **synchronize replicas** using **Merkle trees**

---

Key trade-offs: Response time vs. consistency vs. durability
Data placement

Key K

Each data item is **replicated** at $N$ virtual nodes (e.g., $N = 3$)
Data replication

- Much like in Chord: a key-value pair $\rightarrow$ key’s $N$ successors (*preference list*)
  - Coordinator receives a put for some key
  - Coordinator then replicates data onto nodes in the key’s *preference list*

- Preference list *size* $> N$ to account for node failures

- For robustness, the preference list *skips tokens* to ensure distinct physical nodes
Gossip and “lookup”

- **Gossip:** Once per second, each node contacts a randomly chosen other node
  - They exchange their lists of known nodes (including virtual node IDs)

- Each node **learns** which others handle all **key ranges**
  - **Result:** All nodes can send directly to any key’s coordinator (“zero-hop DHT”)
    - Reduces variability in response times
Varying N, R, W

- Allow configuration of quorums
  - Read from R replicas, wait for W writers
  - Tuning of consistency, availability, durability, performance
  - If R + W > N, then quorum-like, acks writes intersect with future reads

<table>
<thead>
<tr>
<th>N</th>
<th>R</th>
<th>W</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Parameters from paper:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Good durability, good R/W latency</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>Slow reads, weak durability, fast writes</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>Slow writes, strong durability, fast reads</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>More likely that reads see all prior writes?</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Read quorum doesn’t overlap write quorum</td>
</tr>
</tbody>
</table>
Partitions force a choice between availability and consistency

- Suppose **three** replicas are partitioned into **two and one**

- If one replica fixed as master, no client in other partition can write

- In Paxos-based primary-backup, no client in the partition of one can write

- Traditional distributed databases emphasize consistency over availability when there are partitions
Alternative: Eventual consistency

- Dynamo emphasizes **availability over consistency** when there are partitions

- Tell client write complete when only some replicas have stored it

- Propagate to other replicas in background

- **Allows writes in both partitions**...but risks:
  - Returning **stale data**
  - **Write conflicts** when partition heals:
Mechanism: Sloppy quorums

- If no failure, reap consistency benefits of single master
  - Else sacrifice consistency to allow progress

- Dynamo tries to store all values put() under a key on first $N$ live nodes of coordinator’s preference list

- BUT to speed up get() and put():
  - Coordinator returns “success” for put when $W < N$ replicas have completed write
  - Coordinator returns “success” for get when $R < N$ replicas have completed read
Suppose coordinator doesn’t receive $W$ replies when replicating a put()
  • Could return failure, but remember goal of high availability for writes...

**Hinted handoff**: Coordinator tries next successors in preference list (*beyond first $N$*) if necessary
  • Indicates the intended replica node to recipient
  • Recipient will periodically try to forward to the intended replica node
Hinted handoff: Example

• Suppose C fails
  • Node E is in preference list
    • Needs to receive replica of the data
  • Hinted Handoff: replica at E points to node C

• When C comes back
  • E forwards the replicated data back to C
Wide-area replication

• Last ¶, §4.6: Preference lists always contain nodes from more than one data center
  • Consequence: Data likely to survive failure of entire data center

• Blocking on writes to a remote data center would incur unacceptably high latency
  • Compromise: $W < N$, eventual consistency
Sloppy quorums and get()s

• Suppose coordinator doesn’t receive $R$ replies when processing a get()
  • Penultimate ¶, §4.5: “$R$ is the min. number of nodes that must participate in a successful read operation.”
    • Sounds like these get()s fail

• Why not return whatever data was found, though?
  • As we will see, consistency not guaranteed anyway...
Sloppy quorums and freshness

• Common case given in paper: \( N = 3; R = W = 2 \)
  • With these values, do sloppy quorums guarantee a `get()` sees all prior `put()`s?

• If no failures, yes:
  • Two writers saw each `put()`
  • Two readers responded to each `get()`
  • Write and read quorums must overlap!
Sloppy quorums and freshness

• Common case given in paper: $N = 3, R = W = 2$
  • With these values, do sloppy quorums guarantee a get() sees all prior put()s?

• With node failures, no:
  • Two nodes in preference list go down
    • put() replicated outside “beyond” $N$ in pref list
  • Two nodes in preference list come back up
    • get() occurs before they receive prior put()
Conflicts

• Suppose $N = 3$, $W = R = 2$, nodes A, B, C
  • 1$^{st}$ put($k$, ...) completes on A and B
  • 2$^{nd}$ put($k$, ...) completes on B and C
  • Now get($k$) arrives, completes first at A and C

• Conflicting results from A and C
  • Each has seen a different put($k$, ...)

• Dynamo returns both results; what does client do now?
Conflicts vs. applications

- Shopping cart:
  - **Could take union** of two shopping carts
  - What if second put() was result of user deleting item from cart stored in first put()?
    - Result: “resurrection” of deleted item

- Can we do better? Can Dynamo resolve cases when multiple values are found?
  - **Sometimes.** If it can’t, application must do so.
Version vectors (vector clocks)

- **Version vector**: List of (coordinator node, counter) pairs
  - e.g., [(A, 1), (B, 3), ...]

- Dynamo stores a version vector with each stored key-value pair

- **Idea**: track “ancestor-descendant” relationship between different versions of data stored under the same key $k$
Version vectors: Dynamo’s mechanism

- Rule: If vector clock comparison of v1 < v2, then the first is an ancestor of the second – **Dynamo can forget v1**

- Each time a put() occurs, Dynamo increments the counter in the V.V. for the coordinator node

- Each time a get() occurs, Dynamo returns the V.V. for the value(s) returned (in the “context”)
  - Then users **must supply that context** to put()s that modify the same key
Version vectors (auto-resolving case)

\[ v_1 \left[ (A,1) \right] \]

put handled by node A

\[ v_2 \left[ (A,1), (C,1) \right] \]

put handled by node C

\[ v_2 > v_1, \text{ so Dynamo nodes automatically drop } v_1, \text{ for } v_2 \]
Version vectors (app-resolving case)

\[ \text{put handled by node A} \]

\[ \text{v1 } [(A,1)] \]

\[ \text{put handled by node B} \]

\[ \text{v2 } [(A,1), (B,1)] \]

\[ \text{put handled by node C} \]

\[ \text{v3 } [(A,1), (C,1)] \]

Client reads v2, v3; context: \( [(A,1), (B,1), (C,1)] \)

\[ v2 \lor v3, \text{ so a client must perform semantic reconciliation} \]

\[ \text{v4 } [(A,2), (B,1), (C,1)] \]

Client reconciles v2 and v3; node A handles the put
Trimming version vectors

- **Many nodes** may process a series of put()s to same key
  - Version vectors **may get long** – do they grow forever?

- No, there is a **clock truncation scheme**
  - Dynamo stores physical timestamp with each entry

  - When V.V. > 10 nodes long, V.V. **drops** the timestamp of the **node that least recently processed** that key
Impact of deleting a VV entry?

\[ v_2 \parallel v_1 \], so looks like application resolution is required
Concurrent writes

• What if two clients concurrently write w/o failure?
  • e.g. add **different items** to **same cart** at **same time**
  • Each does get-modify-put
  • They both see the same initial version
    • And they both send put() to **same coordinator**

• Will coordinator create two versions with conflicting VVs?
  • We want that outcome, otherwise one was thrown away
  • Paper says, most conflicting versions arise from concurrent writes
Removing threats to durability

• Hinted handoff node **crashes before it can replicate data** to node in **preference list**
  • Need another way to **ensure** that each key-value pair is **replicated N times**

• **Mechanism:** **replica synchronization**
  • Nodes nearby on ring periodically **gossip**
    • Compare the (k, v) pairs they hold
    • Copy any missing keys the other has

How to compare and copy replica state quickly and efficiently?
Efficient synchronization with Merkle trees

- Merkle trees hierarchically summarize the key-value pairs a node holds

- One Merkle tree for each virtual node key range
  - Leaf node = hash of one key’s value
  - Internal node = hash of concatenation of children

- Compare roots; if match, values match
  - If they don’t match, compare children
    - Iterate this process down the tree
Merkle tree reconciliation

- A, B differ in two keys

- Exchange and compare hash nodes from root downwards, pruning when hashes match

Finds differing keys quickly and with minimum information exchange
Evolution of partitioning and placement

Strategy 1: Chord + virtual nodes partitioning and placement

- New nodes “steal” key ranges from other nodes
  - Scan of data store from “donor” node took a day
- Burdensome recalculation of Merkle trees on join/leave
Evolution of partitioning and placement

Strategy 2: Fixed-size partitions, random token placement

• **Q partitions**: fixed and equally sized

• **Placement**: **T** virtual nodes per physical node (random tokens)
  - Place the partition on **first** **N** nodes after its end
Evolution of partitioning and placement

Strategy 3: Fixed-size partitions, equal tokens per partition

- **Q partitions**: fixed and equally sized
- **S total nodes** in the system
- **Placement**: Q/S tokens per partition
Dynamo: Take-away ideas

• Consistent hashing broadly useful for replication—not only in P2P systems

• Extreme emphasis on availability and low latency, unusually, at the cost of some inconsistency

• Eventual consistency lets writes and reads return quickly, even when partitions and failures

• Version vectors allow some conflicts to be resolved automatically; others left to application