Cloud Scale Storage: Industry Distributed File Systems

L17
Theo Benson
Outline

• Distributed storage
  – Commodity server, limited resources,
  – Geodistribution, scalable, reliable

• DynamoDB
  • Write-always

• Cassandra [FB]
  – High reads, high writes

• Megastore [Google]
  – High reads, low writes

• Spanner [Google]
  – RDBMS
  – GFS [Google]
    – AppendOnly + sequential reads
Traditional Web Service: 3-Tiered Architecture

- Stateful Backend = SQL-DB
  - Provides ACID: easy to understand semantics (very little concurrency)
  - Provide SQL-query: easy to write application that read (in complex ways)

- Challenges at scale:
  - Commodity servers: failure happens frequently
  - Scale: need to add/remove server frequently
  - RDBMS requires DB-Administrator: $$$$$$
  - RDDMS is hard to scale
RDBMS V. NoSQL

• **RDBMS → ACID (Strong consistency – Raft-style)**
  – Transactions ensure that strong consistency is maintained
  – Hard to provide strong consistency and high availability
    • During failure (or network partition) if not majority, then can’t make progress

• **NoSQL → weak consistency – Gossip Style**
  – Eventually every one becomes consistent
  – During partition individual nodes can still make progress
  – There may be conflicts (use vector clocks): need conflict resolution
    • DynamoDB: Application specified conflict resolution + vector clocks
    • Cassandra: timestamps + last-writer-wins
General Data Center Characteristics

• Build on Commodity components
  – “in cluster first year, it’s typical that 1000 individual machines failures will occur”
  – “there’s about a 50 percent chacne that the cluster will overhead, taking down most of the servers in less than 5 minutes and taking 1 or 2 days to recover”

• BUGS in code

• Failures will happen a lot!!!!
  – But node will recover → Failures are transient.
  – Don’t want to react drastically because the node will come back up
How Do Request Get Processed in a Data Center

Worker: mappers
Aggregation: reducers
Other implication of Network Limits

Scalability

Scalability of Netflix like recommendation system is bottlenecked by communication.

Did not scale beyond 60 nodes
- Communication time increased faster than computation time decreased.

Application SLAs
- Cascading SLAs
  - SLAs for components at each level of the hierarchy
- Network SLAs
  - Deadlines on communications between components
Motivation

• Even the *slightest* outage has significant financial consequences and impacts customer trust.

• The platform is implemented on top of an infrastructure of *tens of thousands* of servers and network components located in *many datacenters* around the world.

• Persistent state is managed *in the face of these failures* - drives the *reliability and scalability* of the software systems
Shopping Cart
• Edit Cart add brush
• Close browser
• Simultaneously open cart on multiple browsers
  • Add Water in one browser
  • Add Book in another

• There’s a potential conflict
  • Two carts simultaneously modify
  • With Raft: eliminate conflict by editing cart once-at-a-time
  • With Dynamo: all simultaneous writes

• Deal with conflicts during reads
Motivation (Cont’d)

• Build a distributed storage system:
  – Scale
  – Simple: key-value
  – Highly available (sacrifice consistency)
  – Guarantee Service Level Agreements (SLA)
System Assumptions and Requirements

**Query Model**

- simple read and write operations to a data item that is uniquely identified by a key.

- Most of Amazon’s services can work with this simple query model and do not need any relational schema.

- targeted applications - store objects that are relatively small (usually less than 1 MB)
ACID Properties (Cont’d)

• Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability.

• Dynamo targets applications that operate with weaker consistency (the “C” in ACID) if this results in high availability.
Efficiency

- latency requirements which are in general measured at the 99.9th percentile of the distribution
- Average performance is not enough
Other Assumptions

- operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.
Service Level Agreements (SLA)

• Application can deliver its functionality in **bounded** time
  – Every dependency in the platform needs to deliver its functionality with even tighter bounds.

• Example
  – service guaranteeing that it will provide a response within 300ms for **99.9%** of its requests for a peak client load of 500 requests per second.
Service-oriented architecture
Design Consideration

• Sacrifice strong consistency for availability

• Conflict resolution is executed during read instead of write, i.e. “always writeable”.
  – Shopping cart example: must always be able to edit cart even if server is unavailable.
  – Two people can edit cart at the same time.
  – During read: conflict is fixed
    • Cart may have too many items
    • Cart may have old deleted items
Design Consideration (Cont’d)

- Incremental scalability.
- Symmetry
  - Every node in Dynamo should have the same set of responsibilities as its peers.
- Decentralization.
  - In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible.
- Heterogeneity.
  - This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.
- “always writeable” data store where no updates are rejected due to failures or concurrent writes.
- an infrastructure within a single administrative domain where all nodes are assumed to be trusted.
System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection
Partition Algorithm

- **Consistent hashing**: the output range of a hash function is treated as a fixed circular space or “ring”.

- **Load balancing**: "Virtual Nodes": Each node can be responsible for more than one virtual node.
Partition (Cont’d)

• Advantages of using virtual nodes:
  
  – If a node becomes unavailable, the load handled by this node is evenly dispersed across the remaining available nodes.

  – When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.

  – The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.
System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection
Replication

- Each data item is replicated at N hosts.

- "preference list": The list of nodes that is responsible for storing a particular key.
  - N-1 successors

- N = 3
  - K falls between A and B
  - K stored at B, C, D

Key K
Nodes B, C and D store keys in range (A, B) including K.
Data Versioning

• A put() call may return to its caller before the update has been applied at all the replicas

• A get() call may return many versions of the same object.

• Key Challenge: distinct version sub-histories - need to be reconciled.

• Solution: uses vector clocks in order to capture causality between different versions of the same object.
Vector Clock

• A vector clock is a list of (node, counter) pairs.

• Every version of every object is associated with one vector clock.

• If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.
Vector clock example

- **D1** ([Sx, 1])
  - **write**
  - handled by Sx

- **D2** ([Sx, 2])
  - **write**
  - handled by Sx

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

- **D4** ([Sx, 2], [Sz, 1])
  - **write**
  - handled by Sz

- **D5** ([Sx, 3], [Sy, 1], [Sz, 1])
  - reconciled and written by Sx
Client. \rightarrow \text{Get/Set} \rightarrow \text{Load Balancer (Stateless)} \rightarrow \text{Send request to random node}

Nodes B, C and D store keys in range (A,B) including K.
System architecture

• Partitioning
• High Availability for writes
• Handling temporary failures
• Recovering from permanent failures
• Membership and failure detection
Temporary Failures
Sloppy Quorum

• R/W is the minimum number of nodes that must participate in a successful read/write operation.

• Setting $R + W > N$ yields a quorum-like system.
  • $N$ is the number of nodes in the system

• In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas.
  • For this reason, R and W are usually configured to be less than $N$, to provide better latency.
Hinted handoff

- Assume $N = 3$. When $B$ is temporarily down or unreachable during a write, send replica to $F$.

- $F$ is hinted that the replica belongs to $B$ and it will deliver to $B$ when $B$ is recovered.
  - $F$ must constantly monitor the state of $B$

- Again: "always writeable"
System architecture

• Partitioning

• High Availability for writes

• Handling temporary failures

• Recovering from permanent failures

• Membership and failure detection
Client. Load Balancer (Stateless)

Send request to random node

Nodes gossip with each other:
- Allows nodes to build routing table
- Each node knows every other node
  - Allows for faster look up than: chord or tapestry
- Gossip messages allow detection of failure
  - Failure == node doesn’t gossip

Nodes B, C and D store keys in range (A,B) including K.
# Summary of techniques used in *Dynamo* and their advantages

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>
Challenge: When a new node startup, how does it join the ring

Solution: Configuration file
- Each node gets a config file with a list of a few nodes (seeds)
- Using these seeds the new node can bootstrap
Implementation

• Java

• Local persistence component allows for different storage engines to be plugged in:
  – Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  – MySQL: object of > tens of kilobytes
  – BDB Java Edition, etc.
Conclusion

• Dynamo is a highly available and scalable data store for Amazon.com’s e-commerce platform.

• Dynamo has been successful in handling server failures, data center failures and network partitions.

• Dynamo is incrementally scalable and allows service owners to scale up and down based on their current request load.

• Dynamo allows service owners to customize their storage system by allowing them to tune the parameters N, R, and W.
Outline

• Distributed storage
  – Commodity server, limited resources,
  – Geodistribution, scalable, reliable

• DynamoDB
  • Write-always

• Cassandra [FB]
  – High reads, high writes

• Megastore [Google]
  – High reads, low writes

• Spanner [Google]
  – RDBMS
  – GFS [Google]
    – AppendOnly + sequential reads
Summary of techniques used in *Dynamo* and their advantages

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique (Dynamo)</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Consistent hashing</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Virtual nodes</td>
<td>Reassigning Node IDs</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td></td>
</tr>
<tr>
<td>Recovering from permanent</td>
<td>Anti-entropy using Merkle trees</td>
<td>----</td>
</tr>
<tr>
<td>failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Membership and failure</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Gossip-based membership protocol and PHI failure detector</td>
</tr>
<tr>
<td>detection</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenges with Partition scheme

• Partition doesn’t account for
  – Load imbalance (some items are more popular)
  – Key imbalance (items aren’t evenly distributed to keys)

• How to solve this?
  – Have a node appear in multiple locations (dynamoDB)
  – Analyze the keyspace and Move lightly loaded nodes in the ring (Cassandra)
Replication

- Each data item is replicated at N (replication factor) nodes.

- **Different Replication Policies**
  - **Rack Unaware** – replicate data at N-1 successive nodes after its coordinator
  - **Rack Aware** – uses ‘Zookeeper’ to choose a leader which tells nodes the range they are replicas for
  - **Datacenter Aware** – similar to Rack Aware but leader is chosen at Datacenter level instead of Rack level.
Read Operation

* Figure taken from Avinash Lakshman and Prashant Malik (authors of the paper) slides.
Failure Detection

• Traditional approach
  – Heart-beats (Used by HDFS & Hadoop): binary (yes/no)
  – If you don’t get X number of heartbeats then assume failure

• Accural Failure approach
  – Based on the distribution of interarrival times of update messages
    – For all messages exchanged build distribution of latency
  – Based on latency distribution:
    – Returns a # representing probability of death
  – Modify this # to reflect N/W congestion & server load
Practical Experience/lessons Learned

• “Applications that have asked for transactional operations …
  Most developers with years of development experience working with RDBMS’s find this”
  – A need for two phase commit

• “although Cassandra is a completely decentralized systems we have learned that having some amount of coordinate is essential …”
  – Use Zookeeper for Coordination (Similar to Raft)
Outline

• Distributed storage
  – Commodity server, limited resources,
  – Geodistribution, scalable, reliable

• DynamoDB
  • Write-always

• Cassandra [FB]
  – High reads, high writes

• Megastore [Google]
  – High reads, low writes

• Spanner [Google]
  – RDBMS
  – GFS [Google]
    – AppendOnly + sequential reads