Key-Value Stores: Chord & Dynamo
“Chord: A Scalable Peer-to-peer Lookup Service for Internet Application”

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan 2001

- **Ion Stoica** (UC Berkeley) – ACM Dissertation Award, ACM Fellow, Mark Weiser Award
- **Robert Morris** (MIT) – NAE, ACM Fellow, Mark Weiser Award
- **David Karger** (MIT) – ACM Dissertation Award, ACM Fellow
- **Frans Kaashoek** (MIT) – NAE, NAAS, ACM Fellow, Mark Weiser Award
- **Hari Balakrishnan** (MIT) – ACM Dissertation Award, ACM Fellow
This paper introduced a novel protocol that enables efficient key lookup in a large-scale and dynamic environment; the paper shows how to utilize consistent hashing to achieve provable correctness and performance properties while maintaining a simplicity and elegance of design. The core ideas within this paper have had a tremendous impact both upon subsequent academic work as well as upon industry, where numerous popular key-value storage systems employ similar techniques. The ability to scale while gracefully handling node addition and deletion remains an essential property required by many systems today.
Distributed Key-Value Store using Chord

Sole operation: given a key, map the key onto a node

- Assign each node an $m$-bit identifier by hashing the node’s IP addr

- Assign each key $k$ an $m$-bit identifier $\text{keyid} = \text{hash}(k)$
  Map the key $k$ to the first node (clockwise) after its keyid
  - called successor($\text{keyid}$)

Called a Distributed Hash Table (DHT)
Chord: Node Joins

• When node joins, gets keys only from successor

Before

After node 6 joins

Node leaves/fails: symmetric

Only $O(\log^2 N)$ messages required
Chord: Lookups

- Each node maintains its successor in the ring

Correct but slow:
- $O(N)$ lookup for $N$ nodes

Add $m$ fingers:
- $O(\log N)$ lookup for $N$ nodes

“Dynamo: Amazon’s Highly Available Key-value Store”
DeCandia et al. 2007

- Giuseppe DeCandia (Elytra)
- Deniz Hastonrun (Facebook)
- Madan Jampani (Amazon)
- Gunavardhan Kakulapati (CureSkin)
- Avinash Lakshman (Commvault)
- Alex Pilchin (Deloitte)
- Swami Sivasubramanian (VP@Amazon)
- Peter Vosshall (ret. VP@Amazon)
- Werner Vogels (CTO@Amazon)
Dynamo is a scalable and highly reliable distributed key-value store. The paper describes how Dynamo manages the tradeoffs between availability, consistency, cost-effectiveness, and performance, and explains how the system combines a variety of techniques: consistent hashing, vector clocks, sloppy quorums, Merkle trees, and gossip-based membership and failure detection protocols. In particular, the paper emphasizes the value of supporting eventual consistency in order to provide high availability in a distributed system. Dynamo evolved within Amazon to become the basis of a popular cloud service, and also inspired open-source systems such as Cassandra.
Contributions

“The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system.”

Demonstrates that an eventually-consistent storage system can be used in production with demanding applications.

Provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.
System Assumptions & Requirements

• **Query Model & ACID Properties**
  – Key-value queries. State stored as blobs. No schema.
  – Ops on single data item. Objects < 1 MB
  – Sacrifice consistency & isolation

• **Efficiency & Platform**
  – Stringent latency SLOs (e.g., 99.9% within 300 millisecs)
  – Commodity HW

• **Trust**
  – Non-hostile environment, no authentication

• **Scale**
  – 100s hosts
“The choice for 99.9% [SLA] over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much.”
Design Considerations

“One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.”

- Conflict resolution after disconnection: When? Who?
  - When: During reads, in order to ensure writes never rejected
  - Who: Application, fall back to “last write wins” at data store

- Incremental scalability in storage hosts

- Exploit heterogeneity in hosts

- Symmetry: Each node has same responsibilities as its peers

- Decentralization of control
Use Zero-Hop Variant of Chord DHT

- **Zero-hop**: Each node maintains enough routing info locally to route directly to destination node

- **Replication**: A la Chord, replicates each key on $N$ successor nodes, across multiple data centers

$N=3$
Partitioning & Versioning

- Each storage node assigned multiple positions on ring. Why?
  - Node becomes unavailable: added load is more dispersed
  - Node joins / becomes available: assumed load is more balanced
  - Can match #positions assigned to node’s processing power

- Data versioning
  - Vector clocks capture version causality
  - Timestamp (node, ctr) pairs & Drop oldest pair if clock gets too long
Execution of `get()` & `put()` Operations

- Client can route request directly to coordinator OR to node based on load info, who will send to coordinator

- Quorum system: need $R$ nodes to read, $W$ to write
  - $R+W > N$, where $N$ is replication factor
  - In practice, use $R < N$ & $W < N$

- Return all versions that are causally unrelated (by VCs)
  - Also do read repair of stale versions

- For availability, use “sloppy quorum”
  - Send to first $N$ healthy nodes
  - Hinted handoff: If $B$ is down, send to $E$ instead with hint that belongs to $B$
  - $E$ sends to $B$ when $B$ recovers
Divergent Versions

• Good metric for consistency: number of divergent versions seen by the application in production environment
  – From failures
  – From concurrent writes to a single data item

• Shopping cart service: 99.94% of requests saw 1 version
  – Source of many versions? Bots
• State the 3 most important things the paper says. These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Popular Configurations

- Business-logic-specific reconciliation
  - E.g., Shopping cart

- Timestamp-based reconciliation: “last write wins”
  - E.g., Customer session information

- High performance read engine
  - E.g., Product catalog, Promotional items

(N,R,W) provide availability vs. performance trade-off. Common settings?
  - N=3, R=2, W=2
Read/Write Latency

Couple hundred nodes with (3,2,2) configuration
Benefits of Buffering Writes

Also: For durability, one replica does a durable write (but don’t wait)
Fraction of Nodes Out-of-Balance

load deviation threshold = 15%
Discussion

- **Availability (in 2 years of production runs)**
  - Applications have received successful responses (without timing out) for 99.9995% of requests
  - No data loss event

- **Key feature: Tunable (N,R,W)**
  - Requires tuning to get right

- **Scalability challenge**
  - Each node has a full routing table for all data
  - Could introduce hierarchical extensions to Dynamo
Discussion: Summary Question #2

• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Partitioning & Placement Strategies

Also:
• Faster bootstrapping/recovery
• Ease of archival

But:
• Limited scalability because need Q >> S machines
• Hash collisions. Nonintegral Q/S
Replica Synchronization

- Node maintains Merkle tree for each key range it hosts
Membership & Failure Detection

• Explicit command for adding or removing nodes from ring

• Gossip-based protocol propagates membership changes
  – Each node contacts a random peer every second
  – Seed nodes help avoid logical partitions

• Local view of failures suffices
Latency Optimization

• Coordinator for a write is the node that replied fastest to the previous read operation

• Also, increases chance of “read-your-writes” consistency
Server-driven vs. Client-driven Coordination

- **Server-driven**: Load balancer assigns each client read request to a random node that acts as coordinator.

- **Client-driven**: Client caches membership state (refresh by polling random node every 10 secs). Coordinates reads locally. Sends write requests to preference list. Avoids overheads of load balancer & extra network hop of going to random node.

<table>
<thead>
<tr>
<th></th>
<th>99.9th percentile read latency (ms)</th>
<th>99.9th percentile write latency (ms)</th>
<th>Average read latency (ms)</th>
<th>Average write latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-driven</td>
<td>68.9</td>
<td>68.5</td>
<td>3.9</td>
<td>4.02</td>
</tr>
<tr>
<td>Client-driven</td>
<td>30.4</td>
<td>30.4</td>
<td>1.55</td>
<td>1.9</td>
</tr>
</tbody>
</table>
Background vs. Foreground Tasks

- Use admission control on background tasks
- Feedback mechanism determines admitting rate
<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>
Friday: No Class

Monday: Day of Project Meetings

Wednesday’s Class
Big Data Systems (II)

“Spanner: Google’s Globally-Distributed Database”