Dynamo: Amazon’s Highly Available Key-Value Store

By Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels.
ACM Symposium on Operating System Principles, 2007

Zachary Breitbart, Erich Drawdy, David Keefe, Dylan Keskinyan, Michel Maalouli

Georgia Tech

Introduction

Paper

- **Title**: Dynamo: amazon’s highly available key-value store
- **Authors**: Giueseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels
- **Journal**: ACM SIGOPS Operating Systems Review, Volume 41, Issue 6
- **Published**: October 14th, 2007

Motivation

- **Network traffic and distribution**
  - Amazon “serves tens of millions [of] customers using tens of thousands of servers”
- **Need for high availability**
  - Customers need their actions to execute properly regardless of network state
- **Primary-key access**
  - Many Amazon services do not need full capabilities of relational databases, only primary-key access
-

Background

System Assumptions and Requirements

- **Query Model**
  - Simple read and write operations to a single data item identified by a key, no relational schema
- **ACID Properties (Atomicity, Consistency, Isolation, Durability**
  - Dynamo sacrifices some consistency for high availability, no isolation, only single key updates
- **Efficiency**
  - Must support 99.9th percentile latency and throughput requirements
- **Other Assumptions**
  - No security requirements, designed for systems up to hundreds of hosts

Service Level Agreements

- **SLA**
  - Contract between client and service on system characters
  - Typically define client's API request rate and service's latency
  - Ex: responses within 300 ms for 99.9% of requests up to 500 per second
- **99.9th percentile latency**
  - All Amazon SLAs measured at rates that must be achieved for 99.9% of requests
  - Percentile decided based on internal cost-benefit analysis
  - Chosen over other measured to align with Amazon's focus on customer experience

Design Considerations

- Incremental scalability
- Symmetry
- Decentralization
- Heterogeneity

Related Work

Peer to Peer Systems

- **Unstructured P2P networks**
  - Queries spread across network to find many peers with the data
  - Ex: Freenet, Gnutella
- **Structured P2P networks**
  - Use globally consistent protocol to limit hops necessary to answer query
  - Ex: Pastry, Chord
- **Storage systems**
  - Built on top of P2P networks for globally persistent storage
  - Ex: Oceanstore, PAST

Distributed File Systems and Databases

- Support hierarchical namespaces
- Typically sacrifice consistency for higher availability by replicating files
  - Guarantee eventual consistency
- Some are resilient to network partitions and outages
- Use a variety of mechanisms for conflict resolution and fault tolerance
- Ex: Ficus, Coda, Bayou, Google File System, FAB, Antiquity, Bigtable

Discussion

- Dynamo built as an “always writeable” storage service
- Designed for internal use, all nodes assumed to be trusted
  - P2P networks typically cannot trust all clients
- Dynamo applications do not require a relational schema nor a hierarchical namespace
  - Necessary for most distributed file systems
- Support 99.9th percentile latency
  - Routing protocols of P2P networks create higher latency
  - Each Dynamo node stores all routing info to create a “zero-hop DHT”

System Architecture

Overview

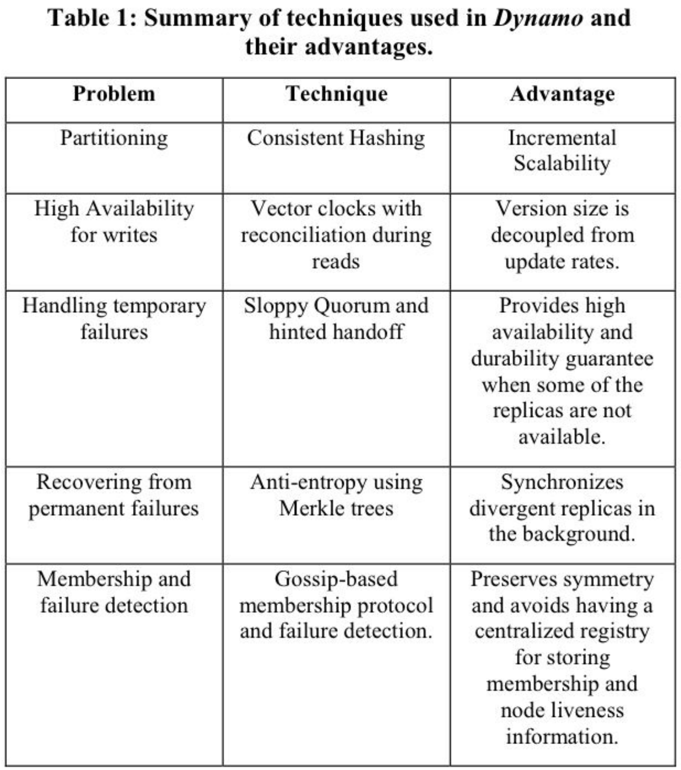


Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

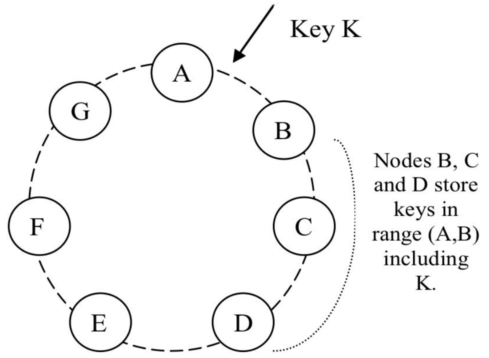
System Interface

- Manages objects associated with a key through two operations
  - **get(key)**
    - Returns single object associated with the key or list of objects with conflicting versions.
  - **put(key, context, object)**
    - **Key:** Determines where (which nodes) the replicas of the object should be placed.
    - **Context:** system metadata about the object (version of the object).

- Key & object treated as an array of bytes
  - No restrictions on structure or format.
  - MD5(key) => 128-bit identifier used to determine storage nodes.

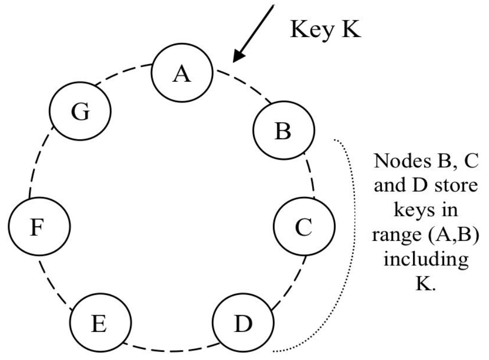
Partitioning Algorithm: Consistent Hashing

- "Ring" output space for hash function
  - Largest hash value wraps around to the smallest hash value
  - Each node is assigned a random value in this space - its position
- MD5(key) => position of data in the ring
  - Data items are mapped to the closest node on the ring in a clockwise direction.
- **Problem:** Random position assignment of nodes leads to non-uniform data and load distribution
- **Solution:** To balance load, each physical node may have multiple virtual nodes
  - Node unavailable => evenly distribute load across remaining nodes
  - Node available => accept roughly equivalent load from other nodes



Replication

- Each data item is replicated at ***N*** hosts
  - *N* is a parameter configured “per-instance”
- Each key is assigned to a coordinator node
  - Locally stores & replicates keys at N-1 clockwise successors.
  - Each node responsible for region of the ring between it and its Nth predecessor.
- **Preference list:** list of nodes responsible for storing a key
  - Every node can determine which nodes are in the list for a given key.
  - To account for failure, contains more than N nodes.
  - Contains only distinct physical nodes.

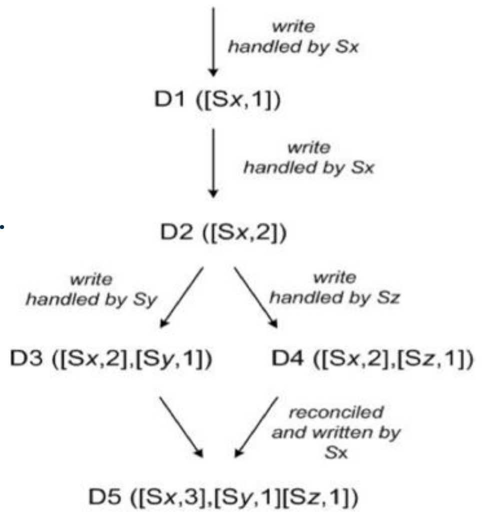


Data Versioning

- **Guarantee:** Eventual consistency
  - Updates propagated to all replicas asynchronously.
  - Each update is a new and immutable version of the object.
  - Multiple versions of an object are present at the same time
    - New versions subsume previous versions (syntactic reconciliation).
    - Version branching (conflicting versions).

- **Key concept:** Vector clocks
  - List of (node, counter) pairs, one per object version.
  - Capture causality between different versions of the same object.

- **Challenge:** Increased storage overhead
  - Growing vector clock if many servers coordinate writes to an object.
  - **Clock truncation scheme:** remove oldest pair once threshold reached.
  - Not likely in practice: writes handled by one of the top N nodes in the preference list.



Execution of get () and put () operations

- get(key) - fetches the value(s) associated with a key
  - May return **multiple versions** (due to Dynamo's eventual consistency) and clients are responsible for resolving version conflicts themselves
- put(key, context, object) - stores object with a key accounting for version context

- **Coordinator Node** - node handling a read/write operation
  - Typically first node within preference list for key (or a random one when load balanced) hat serves as a proxy between client and the N nodes that own the data.

- **Quorum Consistency Mechanism**
  - R (read quorum) - minimum number of nodes that must participate in a successful read
  - W (write quorum) - minimum number of nodes that must acknowledge a successful write
  - Eventual data consistency by R + W > N, with >=1 node overlap in read and write operations

Handling Failures: Hinted Handoff

- **Hinted Handoff**
  - When a designated node for a key-value pair is unreachable, **another node temporarily holds the data with a “hint” indicating its intended destination**
  - Unreachable target node? The coordinator selects an available node for storing the hint (intended original destination) so hinted data can be pushed to it once it is available again

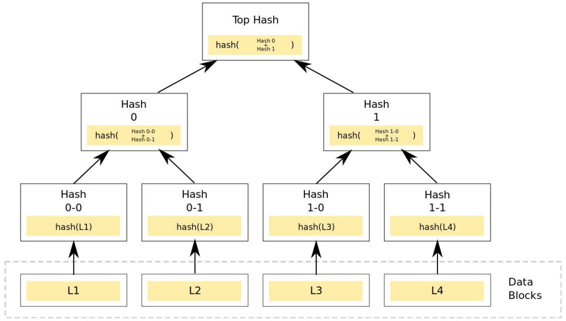
- Helps Dynamo offer a high availability experience, reducing potential for write failures due to temporary node failures while ensuring consistency and durability

Handling Permanent Failures: Replica Synchronization

- **Merkle Tree** - a tree in which every node is labeled by the hash of its child nodes, and leaf nodes are labeled by the hash of their data

- Dynamo uses Merkle Trees to efficiently **identify and reconcile divergent replicas after a permanent failure**
  - Each node maintains a Merkle Tree per key range it stores which allows inconsistencies between nodes to be easily resolved by exchanging sub-branches of their trees
  - Enables targeted synchronization between nodes, only transferring misaligned data

- Reconciliation of replicas at granular levels ensures all nodes eventually converge to the same data state (checks performed during periods of low activity)



Membership Detection

- **Ring Membership**
  - Outages don’t signify permanent node departures; system design avoids unnecessary rebalancing that could disrupt stability
  - **Administrators manually manage the Dynamo rings** to ensure careful control with addition and removal of nodes logged in a record of membership changes that simplifies auditing and rollbacks
  - Gossip-based protocol used for propagation of membership changes, with nodes communicating membership changes peer-to-peer
- **External Discovery**
  - Sequential node additions could cause logical partitions within the Ring, risking data inconsistency
  - Dynamo designated **“seed nodes,”** known to all nodes in the system, that act as anchors and ensure any node joining the system is quickly integrated into the existing structure without isolation
    - Seed information can be static or retrieved from a configuration service

Failure Detection

- Used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas

- Dynamo initially used a global failure detector but shifted to a model where nodes manage failure detection locally to improve efficiency and reduce complexity
  - Traditionally, global failure detection uses a decentralized gossip-style protocol to ensure nodes are informed about system membership changes

- The **local notion of failure detection** focuses on direct interactions rather than maintaining a global state of node health
  - Node A may consider Node B failed if it doesn't respond to A's messages, even if Node B responds to C's messages
    - A will periodically retry B to check if it has recovered, but send messages to other nodes in the meantime

Adding / Removing Storage Nodes

- **Node Addition Process**
  - New nodes are assigned a **set of tokens which determine the node's key range responsibility**
    - Existing nodes need to reevaluate their stored key ranges, with keys that are now within the new node's range having their responsibility transferred to the new node
  - A "confirmation round" is conducted to prevent duplicate key transfers and ensure smooth data migration to the new node

- **Node Removal Process**
  - The node's key ranges are reassigned to remaining nodes in the opposite way described above

- This dynamic approach ensures that load distribution remains uniform across the network, facilitating better system scaling and adaptability for Dynamo

Implementation

Software Components

- Software was required to implement logic for 3 components:
  - **Request Coordination:**
    - Executes read and write operations on behalf of clients by:
      - Collecting data from one or more nodes (reading)
      - Storing data at one or more nodes (writing)
  - **Membership and Failure Detection:**
    - Timeouts regarding message response thresholds
  - **Local Persistence Engine:**
    - Ensuring system maintains data without loss between sessions
    - Long term storage
- All implemented in Java

Local Persistence Engine

- Allows for different storage engines based on different needs
  - **Berkeley Database (BDB):**
    - Embedded key value database library (open source)
      - Simple function API calls for data access and management
      - Development of custom solutions without traditional overhead
      - Used for objects around or lower than 10 kilobytes in size
  - **MYSQL:**
    - Relational database technology and querying language
      - Used for larger object storage
- Applications select which storage solution to use via distribution of their objects' sizes

Georgia Tech

Request Coordination

- Built on top of message processing pipeline
  - **Java NIO Channels**
    - Transport system from entity to byte buffers
      - Asynchronous/non-blocking
      - Utilization of buffers
  - **Request Specific State Machines**
    - Logic for transitioning from states based on I/O
      - Identifying key
      - Sending Request
      - Wait for respondes
      - Handle failures (membership and failure detection)

Read State Machine

- 1. Send read requests to associated nodes
- 2. Wait for minimum number of required responses
- 3. If responses received < required responses (timeout):
  - 3.1. Send failure message
  - 3.2. Handle failure
  - 3.3. Retry
- 4. Gather data versions and determine return
  - 4.1. If versioning is on, generate a vector clock containing all versions
  - 4.2. Read repair

Handling Writes

- Writes are coordinated by one of the top n nodes in the preference list to allow for a more even load distribution.
  - Selects a node by selecting the node that replied fasted to the previous read operation
  - Increases the chance of getting 'read-your writes' consistency, while reducing variability in performance

Experiences and Lessons Learned

Service Configurations

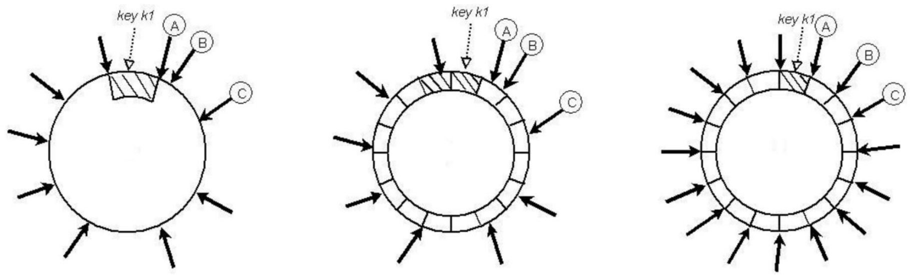
- Main Patterns
  - **Business Logic Specific Reconciliation:**
    - Data objects replicated over multiple nodes
      - In the case of divergent versions of data:
      - Application specific reconciliation logic performs (for example merging two data versions)
  - **Timestamp Based Reconciliation:**
    - “Last write wins”
      - Most recent version of data is considered to be correct
  - **High Performance Read Engine:**
    - Setting R to 1 and W to N
      - Able to partition and replicate data with multiple nodes, increasing scalability
- Ability to fine tune parameters (R, W, N) to customize levels of performance and reliability

Optimizations for Performance vs. Durability

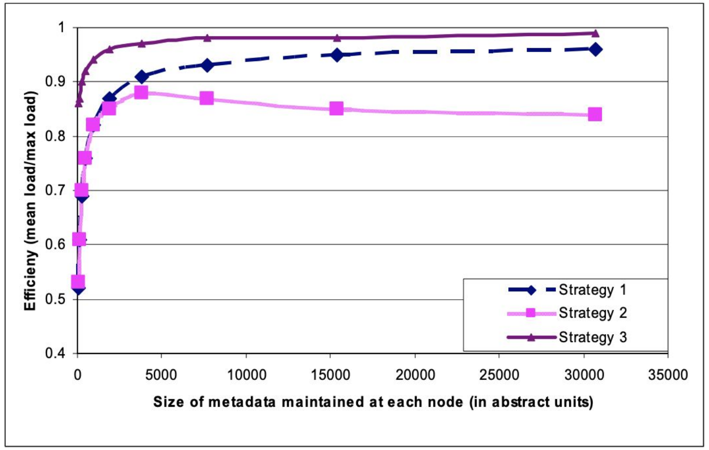
- High performance for read and write operations is a non-trivial task
  - **Using multiple Storage Nodes:**
    - limited by the slowest of the R or W replicas:
      - latencies are around 200 ms
      - Some services required higher performance
  - **Trade-off durability for performance**
    - Storage nodes maintain an object buffer
      - Write operations are stored in buffers
      - gets periodically written to storage by a writer thread
    - Read operations first check if the requested key is present in the buffer
- Lowered latency at the cost of lower durability

Ensuring Uniform Load Balancing

- uniform key distribution -> uniform load distribution:
  - **1. T random tokens per node and partition by token value**
  - **2. T random tokens per node and equal sized partitions**
  - **3. Q/S tokens per node, equal-sized partitions**
    - Q equally sized partitions
    - S: Number of nodes in the system



Strategy 1 Strategy 2 Strategy 3



Client-driven or Server-driven Coordination

- Should the client or a Dynamo server determine which node to go to with a new request?

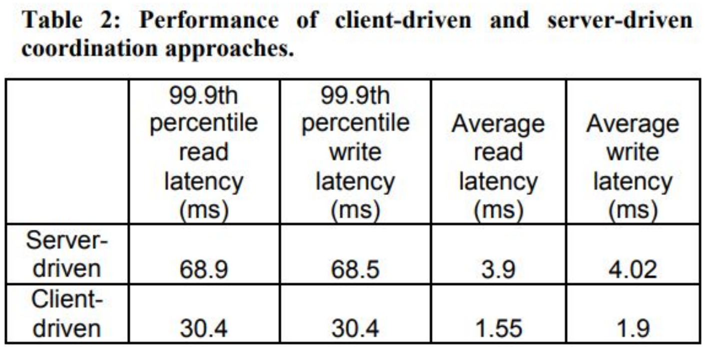


Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

- Why?:
  - The client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node

Balancing Background vs. Foreground Tasks

- Each node performs background tasks for replica synchronization and data handoff
  - These background tasks triggered the problem of resource contention and affected the performance of put() and get() operations

- Solution:
  - Add an admission control mechanism that adjusts background task compute resources based current on put and get performance

Conclusions

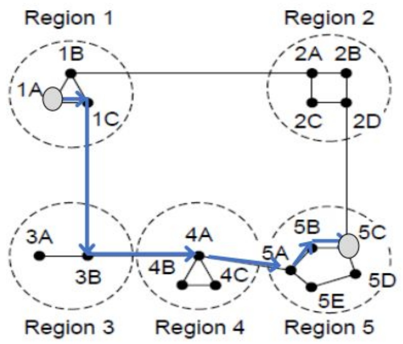
Conclusion

- Widespread use by Amazon to store state on e-commerce platform
- After 2 years of internal deployment at Amazon they noted it:
  - Provided the desired levels of availability and performance
  - Successfully handles server outages, data center failures, and network partitions
  - Successfully scales according to load
  - Has provided adequate customizability for different storage systems
  - Applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date

Conclusion

Negatives?:

- Leaves data consistency and reconciliation logic issues to the developers
  - Many applications already account for different modes of failure
- Adopts a full membership model where each node is aware of the data hosted by its peers
  - Each node actively gossips the full routing table with other nodes in the system
  - Not scalable to 10,000+ nodes because of routing tables
  - Mention future work by incorporating hierarchical extensions



Full table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2A	1B	2
2B	1B	3
2C	1B	3
2D	1B	4
3A	1C	3
3B	1C	2
4A	1C	3
4B	1C	4
4C	1C	4
5A	1C	4
5B	1C	5
5C	1B	5
5D	1C	6
5E	1C	5

Hierarchical table for 1A

Dest.	Line	Hops
1A	–	–
1B	1B	1
1C	1C	1
2	1B	2
3	1C	2
4	1C	3
5	1C	4

CSE 461 University of Washington 22

Network layer (routing) - University of Washington ... (n.d.). https://courses.cs.washington.edu/courses/cse461/17au/lectures/06-1-routing.pdf

Study Questions

1. Explain the tradeoff between different values of N, R, and W in terms of latency, availability, and consistency.

1. How do consistent hashing and the concept of virtual nodes enable incremental scalability in Dynamo? Discuss the challenges Dynamo faces with load distribution and how the strategies evolved to address these challenges.