# Dynamo Presentation

An Exploration of Amazon's Dynamo
A Distributed Data Store System

Yehuda Bogomilsky

Intro to Distributed Systems
Yeshiva University Fall 2021

# The Problem

# Amazon's Needs

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world ...

... In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing .... or data centers are being destroyed by tornadoes.

Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Note: Most (if not all) of the info in this presentation is from the original paper presenting Dynamo - "Dynamo: Amazon's Highly Available Key-value Store" Decandia et al. October 2007.

# So... just use a good old fashioned RDBMS!

# Well, first let's talk about ACID:

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

However -

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

# Hence, The Problem

For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures.

# Enter Dynamo

# Dynamo’s attempt to solve this problem

Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance.

There are many services on Amazon’s platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

# Basic Idea Behind Dynamo: Eventual Consistency

Dynamo is designed to be an eventually consistent data store - that is all updates reach all replicas eventually.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated.

# What Dynamo Is Not

- Dynamo permits only single key updates.
- No operations span multiple data items and there is no need for relational schema
- Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).
- Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization.

# Dynamo Architecture

# How much time do you have?

The architecture of a storage system that needs to operate in a production setting is complex.

In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management.

# Key Pieces Behind Replication and Consistency

- Consistent Hashing for evenly partitioning the load
- Vector Clocks to determine object versions and causal relatedness
- Preference Lists to manage Replication
- Merkle Trees to enforce Consistency internally

# Consistent Hashing

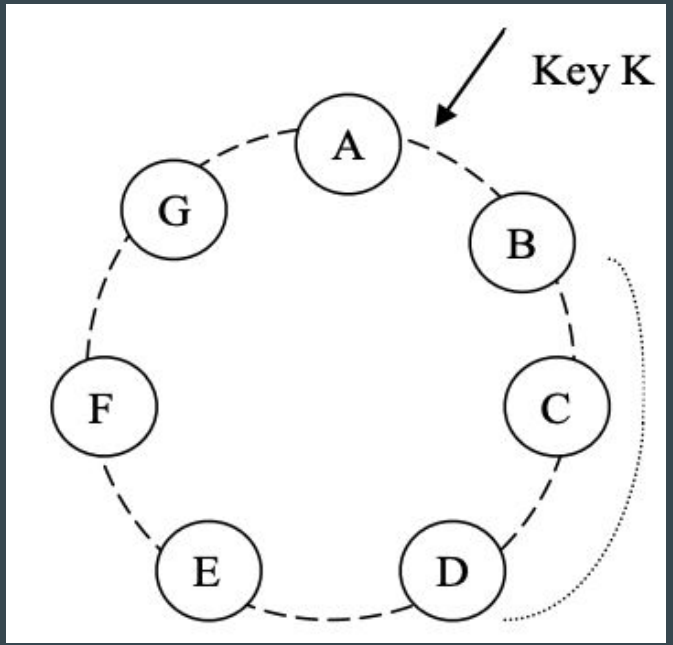
Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

In consistent hashing, the output range of a hash function is treated as a fixed circular space or “ring” (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its “position” on the ring.

Each data item identified by a key is assigned to a node by hashing the data item’s key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item’s position.

# The Ring of Nodes with Consistent Hashing

What's Missing?




Partitioning keys in Dynamo Ring

# Replication using Preference Lists and Coordinators

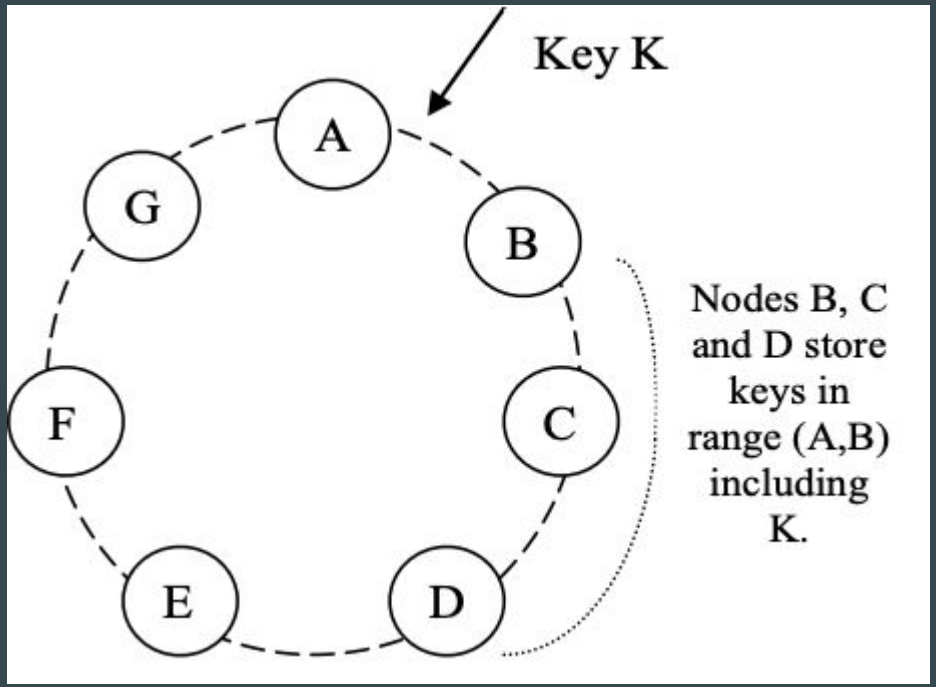
To achieve high availability and durability, Dynamo replicates its data on multiple hosts.

The list of nodes that is responsible for storing a particular key is called the preference list.

Each data item is replicated at N hosts, where N is a parameter configured “per-instance”.

Each key, K, is assigned to a Coordinator Node, who is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the Coordinator replicates these keys at the N-1 clockwise successor nodes in the ring.

# One more time (with Replication)



# How Preference Lists Enable Us to Handle Failure

In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers.

These data centers are connected through high speed network links.

This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

# How Dynamo Handles (Eventual) Consistency - N, R, W.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W.

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

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

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.

# Nothing May Be Forgotten

When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time.

# Vector Clocks to Track Object Versions

Dynamo uses vector clocks in order to capture causality between different versions of the same object.

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

One vector clock is associated with every version of every object.

One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examining their vector clocks.

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 (essentially a newer update of older data).

# When should we Reconcile?

For a number of Amazon services, rejecting customer updates could result in a poor customer experience.

For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures.

This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

# Handling a Write - As simple as 1, 2, W.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally.

The coordinator then sends the new version (along with the new vector clock) to the N highest-ranked reachable nodes.

If at least W-1 nodes respond then the write is considered successful.

# Who Reconciles? .... It Depends

- Syntactic Reconciliation
  - 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
- Semantic Reconciliation
  - " ... However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to collapse multiple branches of data evolution back into one"

# Handling A Read Can Get a Little More Messy

For a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client.

If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated (Syntactic Reconciliation). The divergent versions are then reconciled (Semantic Reconciliation) and the reconciled version superseding the current versions is written back.

# Let's Break This Down A Little

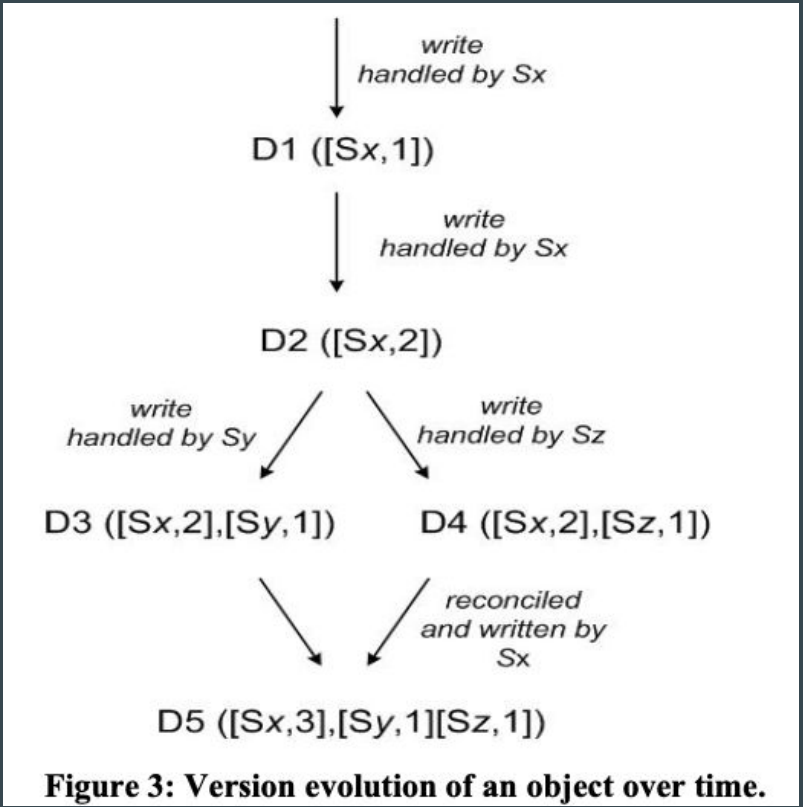
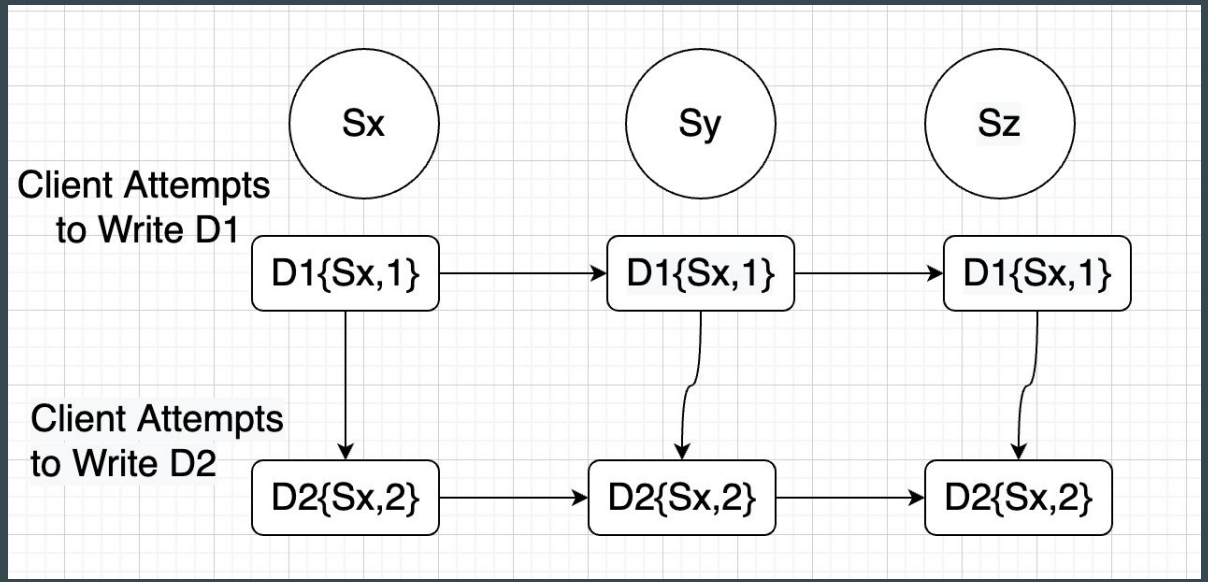


**Figure 3: Version evolution of an object over time.**

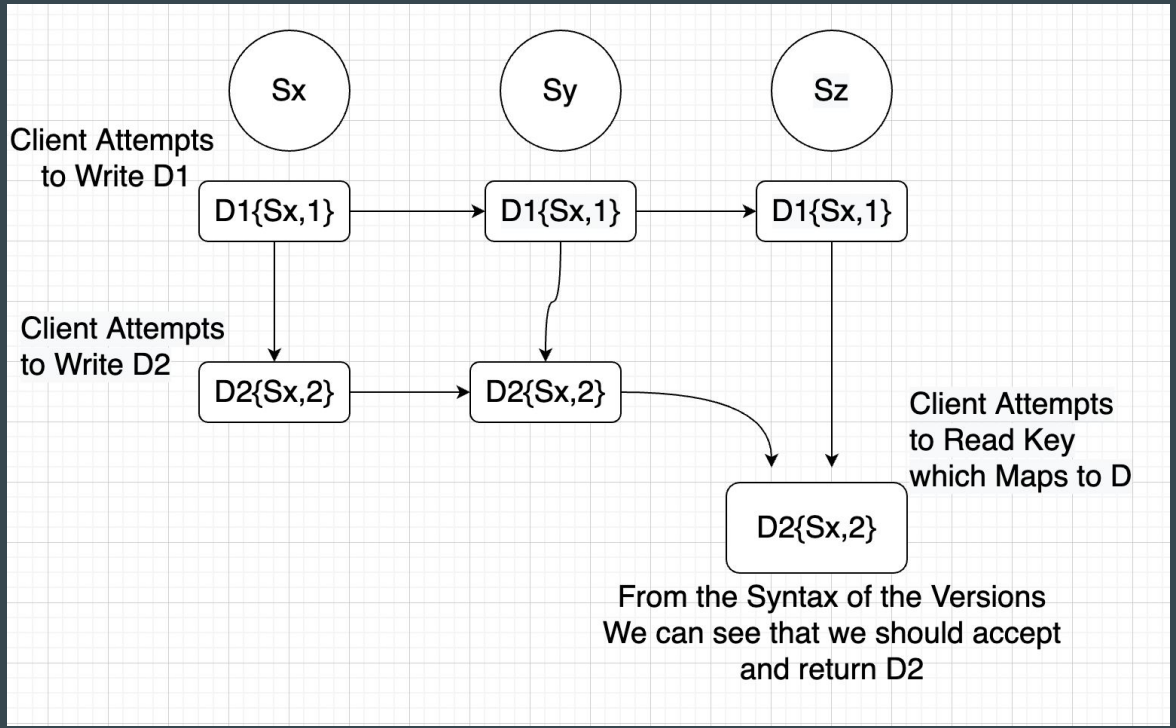
This can be seen as the evolution of a Shopping Cart Object.

Or when a customer is actually placing their order.

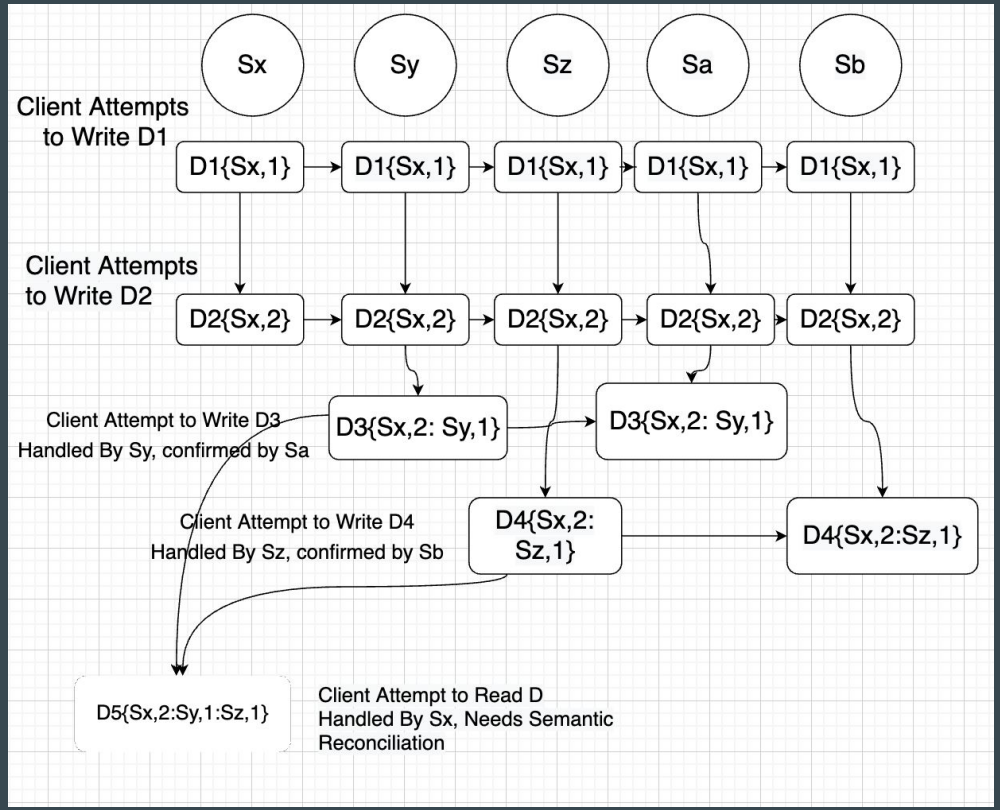
# No Partitions/Divergence - N = 3 , W = 2



# Divergence with Syntactic Reconciliation N = 3 , W = 2 , R = 2



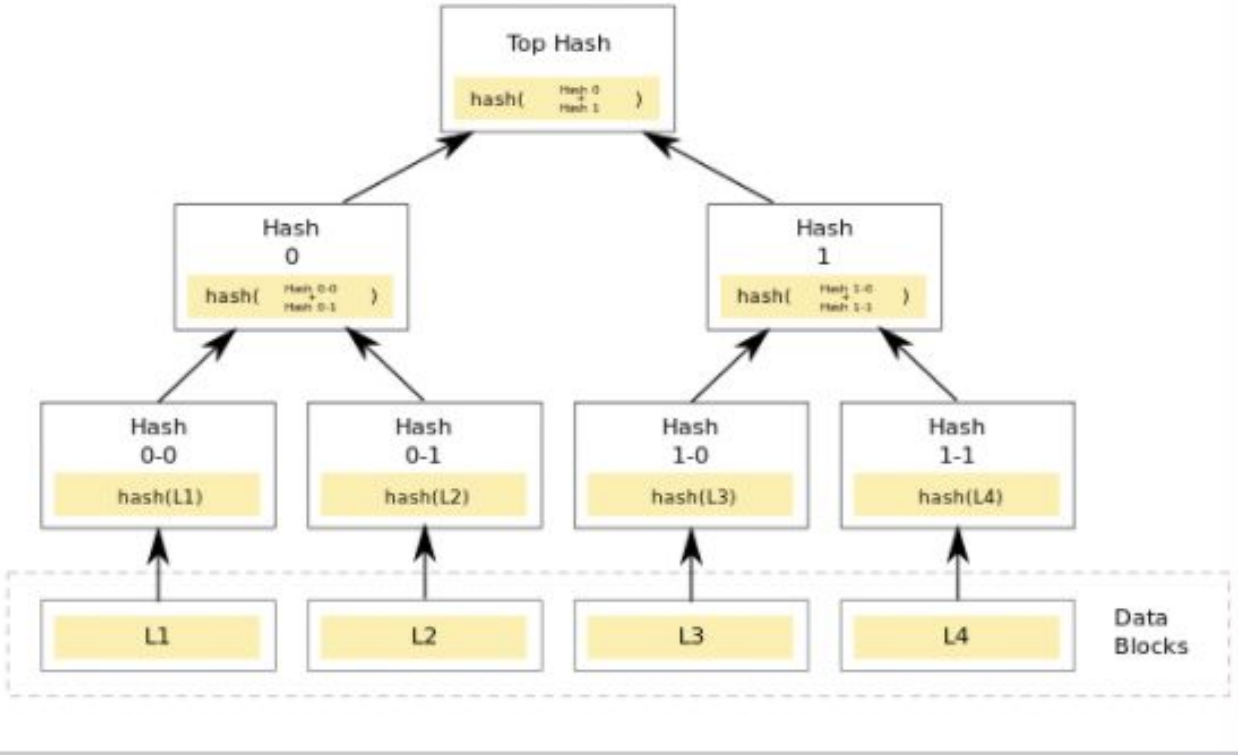
# Divergence With Semantic Reconciliation N = 5 , W = 2 , R = 2



# Merkle Trees to Detect and Resolve Inconsistencies

- A Merkle tree is a hash tree where leaves are hashes of the values of individual keys
- If the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization
- If not, it implies that the values of some replicas are different.
- In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are “out of sync”

# Example Merkle Tree




An example of a binary hash tree. Hashes 0-0 and 0-1 are the hash values of data blocks L1 and L2, respectively, and hash 0 is the hash of the concatenation of hashes 0-0 and 0–1.

# How Dynamo Uses Merkle Trees

Each node maintains a separate Merkle tree for each key range that it hosts.

Two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action.

# Did it Work?

Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly- available applications.

# Summary and Some Concluding Thoughts

- Discussed some of the challenges Amazon was facing and why they needed a new tool like Dynamo to solve them.
- Why Dynamo was able to provide availability and (eventual) consistency and its general structure and data store architecture.
- Examined in detail multiple use cases with different types of consistency issues that can occur between nodes, and what steps Dynamo would take to reconcile them.
- Concluded by discussing whether this did in fact work not only in theory but also, more importantly, in production.

# Thank You