

# Distributed Systems

## 25. Case Study: Amazon Dynamo

Paul Krzyzanowski

[pxk@cs.rutgers.edu](mailto:pxk@cs.rutgers.edu)

# Amazon Dynamo

---

- Not exposed as a web service
  - Used to power parts of Amazon Web Services (such as S3)
  - Highly available, key-value storage system
- In an infrastructure with millions of components, something is always failing!
  - Failure is the normal case
- A lot of services within Amazon only need primary-key access to data
  - Best seller lists, shopping carts, preferences, session management, sales rank, product catalog
  - No need for complex querying or management offered by an RDBMS
  - Full relational database is overkill: limits scale and availability
    - Still not easy to scale or load balance RDBMS on a large scale

# Core Assumptions & Design Decisions

---

- Two operations: get(key) and put(key, data)
  - Binary objects (data) identified by a unique key. Objects tend to be small (< 1MB)
- ACID gives poor availability
  - Use weaker consistency (C) for higher availability.
- Apps should be able to configure Dynamo for desired latency & throughput, balancing performance, cost, availability, durability guarantees.
- At least 99.9% of read/write operations must be performed within a few hundred milliseconds:
  - Avoid routing requests through multiple nodes
- Dynamo can be seen as a zero-hop DHT

# Core Assumptions & Design Decisions

---

- **Incremental scalability**
  - System should be able to grow by adding a storage host (node) at a time
- **Symmetry**
  - Every node has the same set of responsibilities
- **Decentralization**
  - Favor decentralized techniques over central coordinators
- **Heterogeneity**
  - Workload partitioning should be proportional to capabilities of servers

# Compared to Google Bigtable

---

- Dynamo targets apps that only need key/value access with a primary focus on high availability
  - key-value store versus column-store (column families and columns within them)
  - Bigtable: distributed DB built on GFS
  - Dynamo: distributed hash table
  - Updates are not rejected even during network partitions or server failures

# Consistency & Availability

---

- Strong consistency & high availability cannot be achieved simultaneously
- Optimistic replication techniques – eventually consistent model
  - propagate changes to replicas in the background
  - can lead to conflicting changes that have to be detected & resolved
- When do you resolve conflicts?
  - During writes: traditional approach - reject write if cannot reach all (or majority) of replicas
  - During reads: Dynamo approach
    - Design for an "always writable" data store - highly available
    - read/write operations can continue even during network partitions
    - Rejecting customer updates won't be a good experience
      - A customer should always be able to add or remove items in a shopping cart

# Consistency & Availability

---

- Who resolves conflicts?
  - Choices: the data store system or the application
  - Data store
    - application-unaware, so choices limited
    - simple policy, such as "last write wins"
  - Application
    - app is aware of the meaning of the data
    - can do application-aware conflict resolution
    - e.g., merge shopping cart versions to get a unified shopping cart.
  - fall back on "last write wins" if app doesn't want to bother

# Reads & Writes

---

- Two operations:
- **get(key)** returns
  1. object or list of objects with conflicting versions
  2. context (resultant version)
- **put(key, context, value)**
  - stores replicas
  - the nodes that hold replicas are based on the key.
  - *context*: ignored by the application but includes version of object
  - key is hashed with MD5 to create a 128-bit identifier that is used to determine the storage nodes that serve the key

# Partitioning

---

- Break up database into chunks distributed over all nodes
  - Key to scalability
  - Example: Bigtable's tablets, Map-Reduce partitioning
- Relies on **consistent hashing**
  - Regular hashing: change in # slots requires all keys to be remapped
  - Consistent hashing:
    - $K/n$  keys need to be remapped,  $K = \# \text{ keys}$ ,  $n = \# \text{ slots}$
- **Logical ring of nodes**
  - Each node assigned random value in the hash space: position in ring
  - Responsible for all hash values between its value and predecessor's value
  - Hash(key); then walk ring clockwise to find first node with position > hash
  - Adding/removing nodes affects only immediate neighbors

# Partitioning: virtual nodes

---

- A node is assigned to multiple points in the ring
- Each point is a “virtual node”
- Advantages: **balanced load distribution**
  - if a node becomes unavailable, load is evenly dispersed among available nodes
  - if a node is added, it accepts an equivalent amount of load from other available nodes
  - # of virtual nodes per system can be based on the capacity of that node

# Replication

---

- Data replicated on N hosts (N is configurable)
  - Key is assigned a **coordinator** node (via the hashing)
  - Coordinator is in charge of replication
- Coordinator replicates keys at the N-1 clockwise successor nodes in the ring

# Versioning

---

- Not all updates may arrive at all replicas
- **Application-based reconciliation**
  - Each modification of data is treated as a new version
- **Vector clocks** are used for versioning
  - Capture causality between different versions of the same object
  - Vector clock is a set of (node, counter) pairs
  - Returned as a context from a get() operation

# Availability

---

- **configurable values**
  - $R$ : minimum # of nodes that must participate in a successful read operation
  - $W$ : minimum # of nodes that must participate in a successful write operation
- **Metadata hints**
  - If a node was unreachable, replica is sent to another node in the ring
  - Metadata sent with the data contains a hint stating the original desired destination
  - Periodically, a node checks if the originally targeted node is alive
    - if so, it will transfer the object and may delete it locally to keep # of replicas in the system consistent
- **Data center failure**
  - System must handle the failure of a data center
  - Each object is replicated across multiple data centers

# Storage Nodes

---

Each node has three components

## 1. request coordination

- coordinator executes read/write requests on behalf of requesting clients
- state machine contains all logic for identifying nodes responsible for a key, sending requests, waiting for responses, retries, processing retries, packaging response
- each state machine instance handles one request

## 2. membership and failure detection

## 3. local persistent storage

- different storage engines may be used depending on application needs
  - Berkeley Database (BDB) Transactional Data Store (most popular)
  - BDB Java Edition
  - MySQL (for large objects)
  - in-memory buffer with persistent backing store

# Dynamo Service: Amazon S3

# Amazon S3 service (powered by Dynamo)

---

- Storage via web services interfaces (REST, SOAP, BitTorrent)
  - stores more than 449 billion objects
  - 99.9% uptime guarantee (43 minutes downtime per month)
  - proprietary design
  - stores arbitrary objects up to 5 TB in size
- Objects organized into buckets and within a bucket identified by a unique user-assigned key
- Buckets & objects can be created, listed, and retrieved via REST or SOAP
  - <http://s3.amazonaws.com/bucket/key>
- objects can be downloaded via HTTP GET or BitTorrent protocol
  - S3 acts as a seed host and any BitTorrent client can retrieve the file
  - reduces bandwidth costs
- S3 can also host static websites

The End