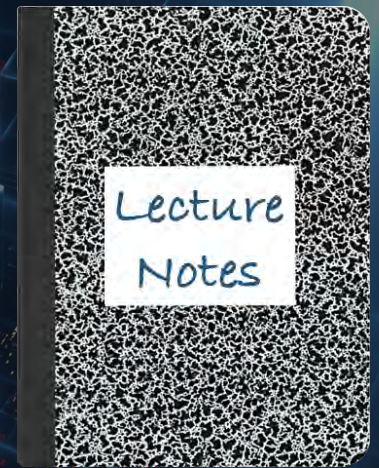


CS 417 – DISTRIBUTED SYSTEMS

Week 7: Distributed Lookup: Part 2: Amazon Dynamo

Paul Krzyzanowski



© 2021 Paul Krzyzanowski. No part of this content, may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

Amazon Dynamo

- Not exposed as a web service
 - Used to power parts of Amazon Web Services and internal services
 - 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 efficient to scale or load balance RDBMS on a large scale

Core Assumptions & Design Decisions

- Two operations: **get** and **put**
 - Binary objects (data) identified by a unique key
 - Objects tend to be small (< 1MB)
- Strongly consistent distributed databases provide poor availability
 - Use weaker consistency (C) for higher availability.
- Apps should be able to configure Dynamo for desired latency & throughput
 - Balance 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 thought of 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 (mix of slow and fast systems)**
 - Workload partitioning should be proportional to capabilities of servers

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 – *but don't deal with conflicts*
 - **Resolve conflicts 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 per object)
- **put(key, context, value)**
 - stores replicas
 - *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:

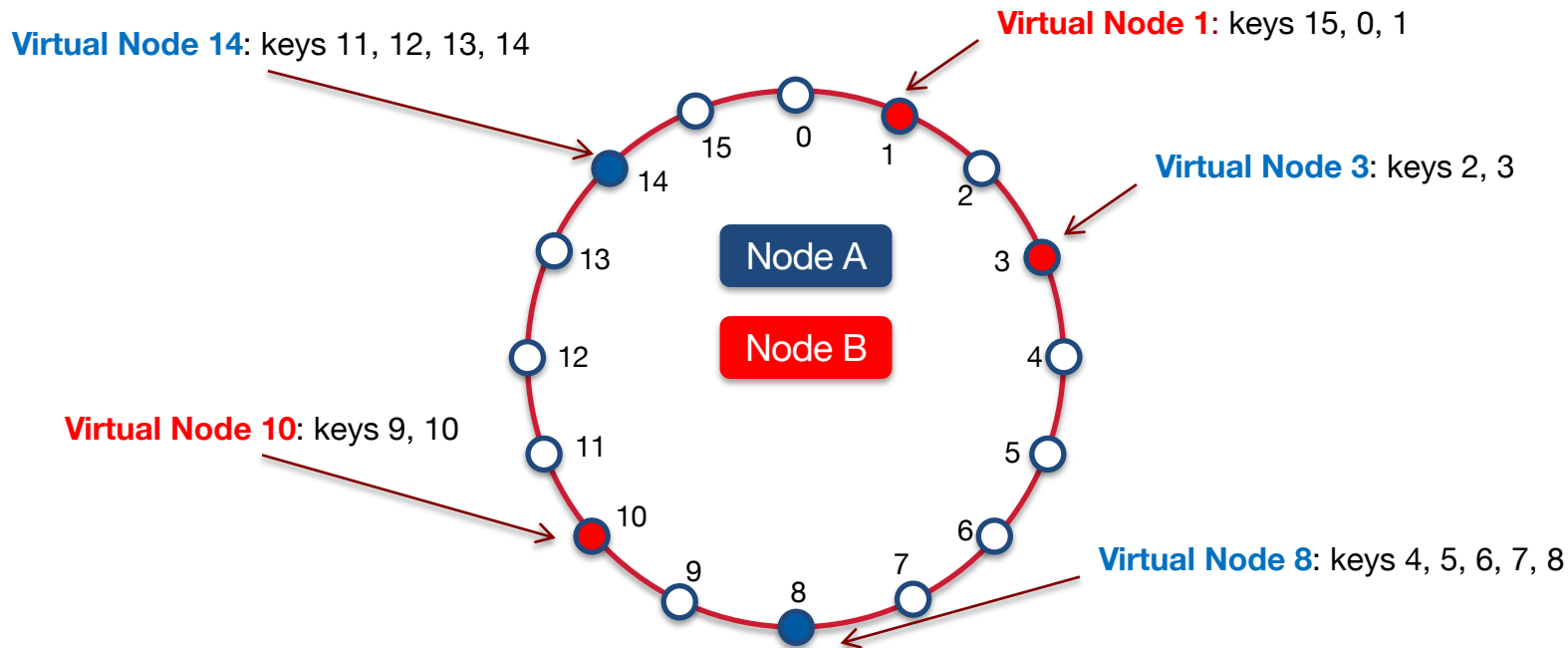
hash(key) identifies node

Partitioning the data

- Break up database into chunks distributed over all nodes
 - Key to scalability
- Relies on **consistent hashing**
 - K/n keys need to be remapped, $K = \# \text{ keys}$, $n = \# \text{ slots}$
- **Logical ring of nodes: just like Chord**
 - Each node assigned a 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: Dynamo virtual nodes

- A physical node holds contents of multiple virtual nodes at multiple points in the ring
- In this example: 2 physical nodes running 5 virtual nodes



Partitioning: virtual nodes

Advantage: **balanced load distribution**

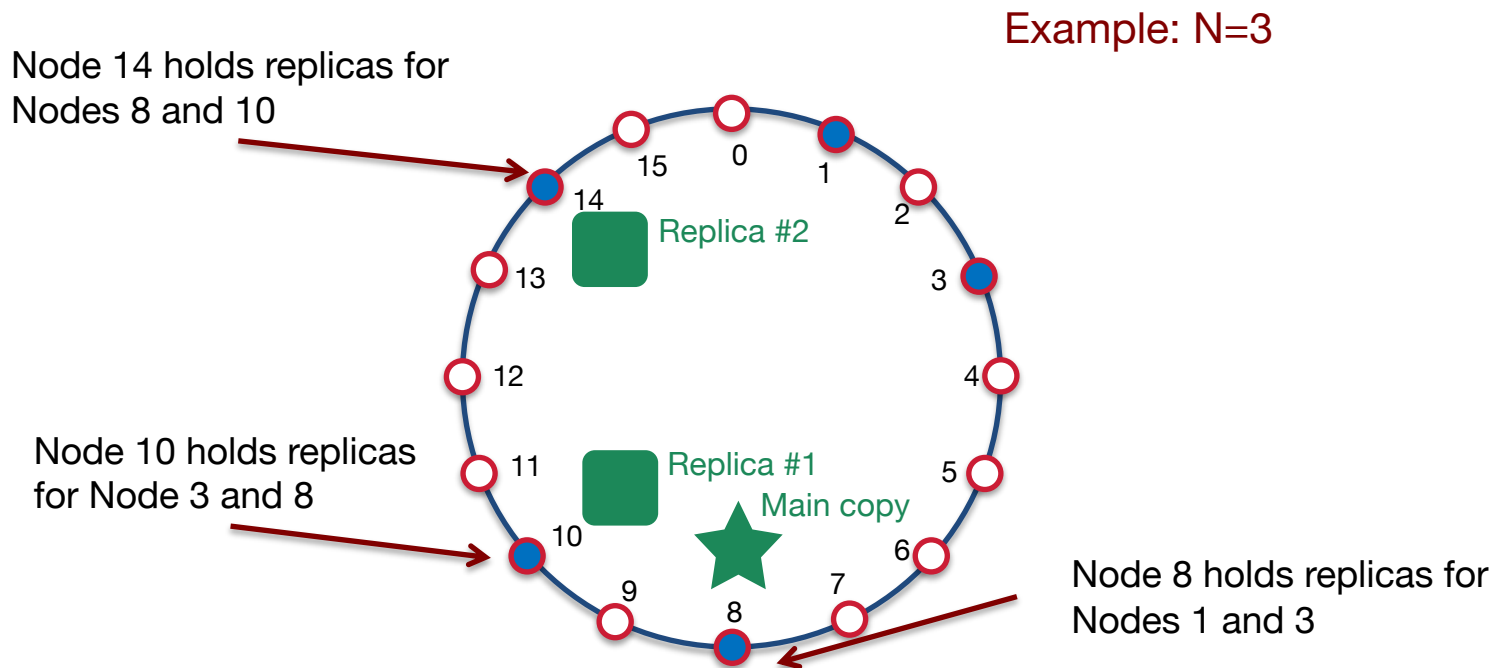
- If a node becomes unavailable, load is evenly dispersed among available nodes
- If a node is added, it accepts an equivalent amount of load from other available nodes
- # of virtual nodes per system can be based on the capacity of that node
 - Makes it easy to support changing technology and addition of new, faster systems

Replication

- Storing/reading key-value data
 - Key is assigned a **coordinator** node (via hashing) \Rightarrow main node
- Replication
 - Data replicated on N hosts (N is configurable)
 - Coordinator oversees replication
 - Coordinator replicates keys at the $N-1$ clockwise successor nodes in the ring

Dynamo Replication

Coordinator replicates keys at the $N-1$ clockwise successor nodes in the ring



Availability & Consistency

- **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 to remember original destination**
 - If a node was unreachable, the data is sent to another node in the ring
 - Metadata sent with the data states 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

Versioning

- Not all updates may arrive at all replicas
 - Clients may modify or read stale data
- 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 and sent via *put()*

Dynamo Storage Nodes

Each node has three components

1. Request coordination

- Node coordinator determined by hash(key)
- Coordinator executes *get/put* 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

Amazon S3 (Simple Storage Service)

Commercial service that implements many of Dynamo's features

- 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