Dynamo

Context

Dynamo observes that the storage layer sets a bound on the reliability and scalability of a scalable system. Therefore, they try hard to provide the usual properties:

- scalability: incremental; add a node at a time
- symmetry: no “special roles” – all features exist in each node. Simplifies management? An extension of this is decentralization: no centralized control. Avoids outages caused by failure of the centralized control nodes.
- availability: reads and writes must succeed even if nodes have failed. Leads dynamo to pick a relaxed eventual consistency model, implying conflict resolution is needed.
- SLAs: focus in 99.9\textsuperscript{th} percentile of latency (1 in a 1000). They don’t describe any particular technique to lower this tail latency, however; they just measure it.

Design

Data partitioning
- consistent hashing
  - circular ID space
    - map both the key (hash) and the node (random ID) to it
    - key is stored on first node that is a successor
  - virtual nodes
    - each physical node is stored as many virtual nodes
    - lets you adapt to node heterogeneity (# virtual nodes ∝ capacity)
    - if many virtual nodes, then on failure, load spread out evenly across ring
- advantages
  - automatically adapts data partitioning as node membership changes, with minimal “reshuffling” of data during repartitioning
  - random node and key assignment gives an approximation to load balance
- disadvantage
  - uneven distribution of key storage is a natural consequence of random node names; leads to uneven query load
  - key management can be expensive when nodes transiently fail
    - as must transfer state on failure, then transfer back on recovery
- need a routing algorithm
  - given a key, how do you know which node is responsible?
  - Dynamo: O(1) routing by having all nodes know about all nodes; flat, complete routing table

Replication
- each data item is replicated at N hosts (usually N=3)
- “preference list”: the set of nodes that is responsible for storing a particular key
  - the node the key is assigned to, followed by its N-1 physically distinct successors
• when new replica is created (e.g., in response to permanent failure), or when doing pairwise anti-entropy, data transfer is coordinated using “merkle tree”
  o hash tree
  o lets you quickly “zoom in” on parts of the data that differ, and minimize the data that has to be transferred to check for inconsistencies

Data versioning
• dynamo provides “eventual consistency” – updates propagate asynchronously, i.e., a put() call will return to its caller before the update is applied at all the replicas
  o implies get() operations may return an object that does not have the latest updates
  o also implies that concurrent put()’s to the same key can result in replica divergence – why?
  o also implies that failures can result in replica divergence – why? (nodes partitioned off won’t get update; update propagates only within the partition)
• idea: each modification creates a new, immutable version of the data
  o multiple versions can be present at the same time
  o most of the time, the system will be able to determine which version is authoritative (“syntactic reconciliation”)
  o sometimes, the client needs to step in to reconcile multiple branches (“semantic reconciliation”)
• idea: name versions using vector clocks to capture causality
  o clock stores list of (coordination server, version at that server) pairs
  o can examine clocks to understand causal history; helpful for clients during semantic reconciliation
  o issue: need to store vector clocks, and those clocks may grow in size if many servers act as coordination server for a key. truncate this list over time.

How get() and put() work
• Dynamo uses a (sloppy) quorum based consistency protocol
  o R: minimum number of nodes that must participate in a read
  o W: minimum number of nodes that must participate in a write
  o if R+W > N, you get a quorum, and per-key sequential consistency
    ▪ dynamo typically operates with N=3, R=2, W=2
    ▪ but, applications can override to choose their own settings
• reads and writes go to the first N healthy nodes in the preference list, skipping those that are down or inaccessible
  o for a put(), the coordinator (the first node in the list) generates a vector timestamp, writes the new version locally, then sends the new version to the N highest-ranked reachable nodes. If at least W-1 respond, the write is successful.
  o for a get(), coordinator requests all versions of the key from the N highest-ranked reachable nodes, waits for R responses, and returns gathered results to the client.
    ▪ implies client may get multiple causally distinct versions of the data, in which case its up to the client to reconcile.
    ▪ nice side-effect: since R < N, waiting for first R helps deals with stragglers!
Evaluation

Latency as a function of time:

- writes 2x slower than reads – why? (disk access)
- 99.9th percentile 10x of average – why? is this good? how do they achieve it?
  - “99th percentile affected by factors such as variability in request load, object sizes, and locality patterns” -- bursts in load, large objects, cold objects
- note that latency $\propto$ load. why?

Write latency can be reduced by using “buffered write” -- write into an in-memory buffer, slowly drain that buffer to disk.

- sacrifices durability under some failure modes for performance in the common case
- what do you think of relying on “store in multiple nodes’ memory” as a durability guarantee?

Divergent version frequency

- 99.94% of requests saw one version
- 0.00057% saw two versions
- 0.00047% saw three versions
- 0.00009% saw four versions

- “experience shows that the increase in number of divergent versions is contributed not by failures but due to the increase in the number of concurrent writers.”
  - concurrent write sharing is rare, so divergence is rare
  - “triggered by busy robots – sensitive nature of the story” 😊