Dynamo

I. Background

Highly available key-value store (hash table)

- S3 is a highly available file system
- EC2 is a highly available VM hosting service

Goals:

- HA despite high load, many faults
- Internal service => non-malicious clients (other Amazon services only)
  - each client service runs its own dynamo instances
- enable tradeoffs among consistency, availability, cost and performance
- simple primary key operations only
- no multi-key atomic operations; no isolation except single key updates
- incremental scalability & heterogeneity
- minimal management required
- eventual consistency
- replace DBMS for many needs
  - expensive to buy; expensive to operate/manage
  - tends to pick C over A
- 99.9 percentile metrics

Techniques:

- consistent hashing to avoid heavy repartition with varying participants
- versioning for consistency
- quorom-like consistency protocol
- decentralized replica synchronization
- gossip-based failure detection

Design issues

- choose A, P, and then work on conflict resolution
- when to resolve conflicts? on reads or on writes? (dynamo picks reads; writes always work)

Dynamo

II. Architecture

- get(key) -> {list of conflicting versions}, context
  - key is array of bytes
  - Internal keys are 182-bit MD5 hash of the key
- put(key, context, object)
  - object is array of bytes

Context is a complex version number, covered below.

Partition key among nodes via consistent hashing:

- oversample by some factor to get more even distribution; ie. each node supports 16 virtual nodes, so variation in load reduced by sqrt(16) = 4
- virtual nodes can be spread unevenly to support heterogeneity of nodes

N-way replication for each key-value pair

- n-1 successors also store a replica
- ... but would like other nodes to be able to know all n nodes, not just the first one (in case that one is down); more on this later
- but want independent failure, so use first n-1 *distinct* nodes (since one real node may support multiple virtual nodes among the the n)

Versioning

- choose A, so updates work all the time
- given a partition, the two sides may have different versions
- eventually both visible (and both returned by get()); **this is exposed in the API. This is the right call as there is no generic way to hide inconsistent versions well.**
- add/delete are both updates and both increment version number
- Vector clocks:
  - list of (node, counter) pairs
  - given two VCs, v1 and v2, v1 is newer if it has a superset (or the same set) of nodes and for every node in common, the counter value for v1 is >= that of v2
  - usually only one node entry, but expands in the presence of faults/partitions
  - scalability limited by # of distinct nodes

Operation:

- use load balancer or smart client approach to find server node
- typically start at first of N replicas, called coordinator; if guess was wrong (ie. not a top N node), then forward to correct node
-
- put (write): need W replicas to participate; R+W > N
  - generate new vector clock (given context and node)
  - write new version locally

Dynamo

- send object and VC to N highest ranked reachable nodes
- wait for W-1responses
- return to client

o get (read): need R replicas to participate

- request versions from N best successors
- wait for R reponses
- find current versions (remove duplicates, dominated VCs)
- return current versions (client may do conflict resolution)

Hinted handoff:

- o may need to send to non-optimal node due to failures/partitions
- o once partition fixed we can forward local versions to their correct home

N vs R vs. W

Replica synchronization:

- o Merkle trees: hierachical hash function; leaves are the keys and a parent node is the hash of its children
- o To compare replicas, just compare their trees top down. If root is the same, then whole tree is the same. Walk down the path that doesn't match to find the mismatched keys
- o one tree for each virtual node
- o churn => some changes in virtual nodes and therefore a need to recalculate trees sometimes

Explicit join/leave of nodes

- o other "leaves" are temporary, likely due to faults
- o having explicit join/leave differentiates between temporary and permanent. The latter implies repartitioning for example.