6.824 2015 Lecture 18: Dynamo

Dynamo: Amazon's Highly Available Key-value Store
DeCandia et al, SOSP 2007

Why are we reading this paper?
  Database, eventually consistent, write any replica
    Like Ficus -- but a database! A surprising design.
  A real system: used for e.g. shopping cart at Amazon
  More available than PNUTS, Spanner, &c
  Less consistent than PNUTS, Spanner, &c
  Influential design; inspired e.g. Cassandra
  2007: before PNUTS, before Spanner

Their Obsessions
  SLA, e.g. 99.9th percentile of delay < 300 ms
  constant failures
  "data centers being destroyed by tornadoes"
  "always writeable"

Big picture
  [lots of data centers, Dynamo nodes]
  each item replicated at a few random nodes, by key hash

Why replicas at just a few sites? Why not replica at every site?
  with two data centers, site failure takes down 1/2 of nodes
    so need to be careful that *everything* replicated at *both* sites
  with 10 data centers, site failure affects small fraction of nodes
    so just need copies at a few sites

Consequences of mostly remote access (since no guaranteed local copy)
  most puts/gets may involve WAN traffic -- high delays
    maybe distinct Dynamo instances with limited geographical scope?
    paper quotes low average delays in graphs but does not explain
  more vulnerable to network failure than PNUTS
    again since no local copy

Consequences of "always writeable"
  always writeable => no master! must be able to write locally.
  always writeable + failures = conflicting versions

Idea #1: eventual consistency
  accept writes at any replica
  allow divergent replicas
  allow reads to see stale or conflicting data
  resolve multiple versions when failures go away
    latest version if no conflicting updates
    if conflicts, reader must merge and then write
  like Bayou and Ficus -- but in a DB

Unhappy consequences of eventual consistency
  May be no unique "latest version"
  Read can yield multiple conflicting versions
  Application must merge and resolve conflicts
  No atomic operations (e.g. no PNUTS test-and-set-write)

Idea #2: sloppy quorum
  try to get consistency benefits of single master if no failures
    but allows progress even if coordinator fails, which PNUTS does not

```
    when no failures, send reads/writes through single node
      the coordinator
      causes reads to see writes in the usual case
    but don't insist! allow reads/writes to any replica if failures

  Where to place data -- consistent hashing
    [ring, and physical view of servers]
    node ID = random
    key ID = hash(key)
    coordinator: successor of key
      clients send puts/gets to coordinator
    replicas at successors -- "preference list"
    coordinator forwards puts (and gets...) to nodes on preference list

  Why consistent hashing?
    Pro
      naturally somewhat balanced
      decentralized -- both lookup and join/leave
    Con (section 6.2)
      not really balanced (why not?), need virtual nodes
      hard to control placement (balancing popular keys, spread over sites)
      join/leave changes partition, requires data to shift

  Failures
    Tension: temporary or permanent failure?
      node unreachable -- what to do?
      if temporary, store new puts elsewhere until node is available
      if permanent, need to make new replica of all content
    Dynamo itself treats all failures as temporary

  Temporary failure handling: quorum
    goal: do not block waiting for unreachable nodes
    goal: put should always succeed
    goal: get should have high prob of seeing most recent put(s)
    quorum: R + W > N
      never wait for all N
      but R and W will overlap
      cuts tail off delay distribution and tolerates some failures
    N is first N *reachable* nodes in preference list
      each node pings successors to keep rough estimate of up/down
      "sloppy" quorum, since nodes may disagree on reachable
    sloppy quorum means R/W overlap *not guaranteed*

  coordinator handling of put/get:
    sends put/get to first N reachable nodes, in parallel
    put: waits for W replies
    get: waits for R replies
    if failures aren't too crazy, get will see all recent put versions

  When might this quorum scheme *not* provide R/W intersection?

  What if a put() leaves data far down the ring?
    after failures repaired, new data is beyond N?
    that server remembers a "hint" about where data really belongs
    forwards once real home is reachable
    also -- periodic "merkle tree" sync of key range

  How can multiple versions arise?
    Maybe a node missed the latest write due to network problem
    So it has old data, should be superseded by newer put()s
    get() consults R, will likely see newer version as well as old
```

```
How can *conflicting* versions arise?
  N=3 R=2 W=2
  shopping cart, starts out empty ""
  preference list n1, n1, n3, n4
  client 1 wants to add item X
    get() from n1, n2, yields ""
    n1 and n2 fail
    put("X") goes to n3, n4
  client 2 wants to delete X
    get() from n3, n4, yields "X"
    put("") to n3, n4
  n1, n2 revive
  client 3 wants to add Y
    get() from n1, n2 yields ""
    put("Y") to n1, n2
  client 3 wants to display cart
    get() from n1, n3 yields two values!
      "X" and "Y"
      neither supersedes the other -- the put()s conflicted

How should clients resolve conflicts on read?
  Depends on the application
  Shopping basket: merge by taking union?
    Would un-delete item X
    Weaker than Bayou (which gets deletion right), but simpler
  Some apps probably can use latest wall-clock time
    E.g. if I'm updating my password
    Simpler for apps than merging
  Write the merged result back to Dynamo

How to detect whether two versions conflict?
  As opposed to a newer version superseding an older one
  If they are not bit-wise identical, must client always merge+write?
  We have seen this problem before...

Version vectors
  Example tree of versions:
    [a:1]
           [a:1,b:2]
    VVs indicate v1 supersedes v2
    Dynamo nodes automatically drop [a:1] in favor of [a:1,b:2]
  Example:
    [a:1]
           [a:1,b:2]
    [a:2]
    Client must merge

get(k) may return multiple versions, along with "context"
  and put(k, v, context)
  put context tells coordinator which versions this put supersedes/merges

Won't the VVs get big?
  Yes, but slowly, since key mostly served from same N nodes
  Dynamo deletes least-recently-updated entry if VV has > 10 elements

Impact of deleting a VV entry?
  won't realize one version subsumes another, will merge when not needed:
    put@b: [b:4]
    put@a: [a:3, b:4]
    forget b:4: [a:3]
```

```
    now, if you sync w/ [b:4], looks like a merge is required
  forgetting the oldest is clever
    since that's the element most likely to be present in other branches
    so if it's missing, forces a merge
    forgetting *newest* would erase evidence of recent difference

Is client merge of conflicting versions always possible?
  Suppose we're keeping a counter, x
  x starts out 0
  incremented twice
  but failures prevent clients from seeing each others' writes
  After heal, client sees two versions, both x=1
  What's the correct merge result?
  Can the client figure it out?

What if two clients concurrently write w/o failure?
  e.g. two clients add diff items to same cart at same time
  Each does get-modify-put
  They both see the same initial version
  And they both send put() to same coordinator
  Will coordinator create two versions with conflicting VVs?
    We want that outcome, otherwise one was thrown away
    Paper doesn't say, but coordinator could detect problem via put() context

Permanent server failures / additions?
  Admin manually modifies the list of servers
  System shuffles data around -- this takes a long time!

The Question:
  It takes a while for notice of added/deleted server to become known
    to all other servers. Does this cause trouble?
  Deleted server might get put()s meant for its replacement.
  Deleted server might receive get()s after missing some put()s.
  Added server might miss some put()s b/c not known to coordinator.
  Added server might serve get()s before fully initialized.
  Dynamo probably will do the right thing:
    Quorum likely causes get() to see fresh data as well as stale.
    Replica sync (4.7) will fix missed get()s.

Is the design inherently low delay?
  No: client may be forced to contact distant coordinator
  No: some of the R/W nodes may be distant, coordinator must wait

What parts of design are likely to help limit 99.9th pctile delay?
  This is a question about variance, not mean
  Bad news: waiting for multiple servers takes *max* of delays, not e.g. avg
  Good news: Dynamo only waits for W or R out of N
    cuts off tail of delay distribution
    e.g. if nodes have 1% chance of being busy with something else
    or if a few nodes are broken, network overloaded, &c

No real Eval section, only Experience

How does Amazon use Dynamo?
  shopping cart (merge)
  session info (maybe Recently Visited &c?) (most recent TS)
  product list (mostly r/o, replication for high read throughput)

They claim main advantage of Dynamo is flexible N, R, W
  What do you get by varying them?
  N-R-W
```

```
    3-2-2 : default, reasonable fast R/W, reasonable durability
    3-3-1 : fast W, slow R, not very durable, not useful?
    3-1-3 : fast R, slow W, durable
    3-3-3 : ??? reduce chance of R missing W?
    3-1-1 : not useful?

  They had to fiddle with the partitioning / placement / load balance (6.2)
    Old scheme:
      Random choice of node ID meant new node had to split old nodes' ranges
      Which required expensive scans of on-disk DBs
    New scheme:
      Pre-determined set of Q evenly divided ranges
      Each node is coordinator for a few of them
      New node takes over a few entire ranges
      Store each range in a file, can xfer whole file

  How useful is ability to have multiple versions? (6.3)
    I.e. how useful is eventual consistency
    This is a Big Question for them
    6.3 claims 0.001% of reads see divergent versions
      I believe they mean conflicting versions (not benign multiple versions)
      Is that a lot, or a little?
    So perhaps 0.001% of writes benefitted from always-writeable?
      I.e. would have blocked in primary/backup scheme?
    Very hard to guess:
      They hint that the problem was concurrent writers, for which
        better solution is single master
      But also maybe their measurement doesn't count situations where
        availability would have been worse if single master

  Performance / throughput (Figure 4, 6.1)
    Figure 4 says average 10ms read, 20 ms writes
      the 20 ms must include a disk write
      10 ms probably includes waiting for R/W of N
    Figure 4 says 99.9th pctil is about 100 or 200 ms
      Why?
      "request load, object sizes, locality patterns"
      does this mean sometimes they had to wait for coast-coast msg?

  Puzzle: why are the average delays in Figure 4 and Table 2 so low?
    Implies they rarely wait for WAN delays
    But Section 6 says "multiple datacenters"
      you'd expect *most* coordinators and most nodes to be remote!
      Maybe all datacenters are near Seattle?

  Wrap-up
    Big ideas:
      eventual consistency
      always writeable despite failures
      allow conflicting writes, client merges
    Awkward model for some applications (stale reads, merges)
      this is hard for us to tell from paper
    Maybe a good way to get high availability + no blocking on WAN
      but PNUTS master scheme implies Yahoo thinks not a problem
    No agreement on whether eventual consistency is good for storage systems
```