COPS

Context

Paper coins the term "ALPS":

- availability: all operations issued to the data store complete successfully; no operation will block indefinitely or return an error signifying that data is unavailable. Hence, no operation can block waiting for a replica to recover. Rules out paxos / 2pc / other coordination mechanisms on the critical path of reads and writes.

- low latency: client operations complete "quickly," on the order of a few milliseconds. Rules out cross-data-center RTTs.

- partition tolerance: data store continues to operate under network partitions. Rules out strong consistency and sequential consistency (see below)

- high scalability: linear, incremental scalability

Linearizability and sequential consistency provide a global ordering of updates, even if those updates are unrelated. One of the implications of these strong-ish consistency models is the need to propagate updates synchronously to all replicas:

- an update cannot commit until its order is defined relative to other updates
- if have asynchronous update propagation, other updates may show up at replicas in a different order than ours, leading to inconsistent read orders at different replicas

So, ALPS systems require a weaker-than-sequential consistency model. Claim in this paper (proven in [35]) is that causal+ is the strongest consistency model achievable under these constraints.

Causal consistency:

- uses the usual lamport-style causal definition
  - in this system, a "context" argument to the data store defines what it means to be a single thread of execution
  - if the OS process were the thread of execution, could introduce many false dependencies
- if a get can see an update X, that get reflects all the updates that X depends on causally
  - however, causally concurrent updates can propagate in any order
    - implies concurrent updates to the same key can conflict and result in divergent replicas
  - causal+ -- conflicts will converge at all replicas (eventually)

System design

- all data is fully replicated at each COPS site
  - a COPS site is a cluster or a data center
  - inside a site, assume network RTTs and failures are bounded so that you can provide strong consistency with low latency, using paxos or chain replication
  - inside a site, data is partitioned across nodes using chain replication

- each key has one primary node in a cluster
    - set of primary nodes across cluster is called the set of "equivalent nodes" for the key
    - after a write completes locally, the primary node places it in a replication queue, from which it is sent asynchronously to the equivalent nodes
    - the equivalent nodes wait until the value's causal dependencies are satisfied before locally committing
        - so, a write will commit at different times in different clusters, partly because of asynchrony, and partly because of delays introduced by satisfying the causal ordering property

Writes in detail

First, a write goes to the local cluster

- a client calls put(key, val, ctx) into its local library.
    - library calls put_after(key, version=0, nearest), where nearest is the nearest dependency in the dependency chain
    - primary node in the local cluster assigns the key a version number using a Lamport timestamp
        - allows COPS to derive a single global order over all writes for each key
        - last-writer-wins convergent conflict handling
    - after write commits locally, return success to client
    - after write commits locally, primary node asynchronously replicates to equivalent nodes using a stream of put_after operations
        - a node that receives put_after from another cluster must wait until dependencies have been satisfied locally
        - does this by issuing a "dep_check(key, version=nearest)" call, that blocks until that key/version has been written
        - then, it is safe to commit to the cluster

Reads

- clients call get(key, ctx) in the library
    - library calls get_by_version(key, version=LATEST) in local cluster
    - library returns value to client

Failure handling

- within a cluster, rely on a cluster-specific linearizable, fault tolerant store. e.g., bigtable, FAWN, or something else.
- that leaves only data center failures
    - if a put_after originates in a failed data center but hasn't been copied out, it is lost
    - not clear what happens if a put_after gets copied out to a subset of remote stores
        - creates a dependency that will never be satisfied…block forever in dep_check?
    - replication queues in active data centers will grow until failed data center either recovers or is marked as permanently down
        - huge issue; rate at which replication queue grows is proportional to the write rate of the system!!

Evaluation

- a couple of "ugh"s in the experimental setup
    - "All reads and writes in FAWN-KV go to disk, but most operations in our experiments hit the kernel buffer cache." Not clear if writes are synchronous out to disk: probably not, given the latencies.
    - Single cluster testbed split into emulated datacenters, but it doesn't seem as though they configured their "WAN" to have WAN latencies.

Microbenchmarks

- ping and get_by_version have very similar latencies and throughputs
    - very little computation needs to be done, and reads satisfied out of buffer cache rather than actually hitting disk
- ping and put_after shows that put_after has higher latency and lower throughput
    - more computationally expensive; have to update metadata and write values in the local stores

High-level observations

- we didn't talk about the COPS-GT implementation, but it is more expensive, as it has to pass around dependency graphs
    - throughput depends very much on how long the dependency chains that get passed around grow
        - garbage collection can truncate these, but only after long enough
        - get weird inflections in the graphs as a result