Replication

How'd we get here?

- Failures & single systems; fault tolerance techniques added redundancy (ECC memory, RAID, etc.)
- Conceptually, ECC & RAID both put a "master" in front of the redundancy to mask it from clients -- ECC handled by memory controller, RAID looks like a very reliable hard drive behind a (special) controller

Simpler examples...

- Replicated web sites
- e.g., Yahoo! or Amazon:
  - DNS-based load balancing (DNS returns multiple IP addresses for each name)
  - Hardware load balancers put multiple machines behind each IP address
  - (Diagram. :)

Read-only content

- Easy to replicate - just make multiple copies of it.
  - Performance boost: Get to use multiple servers to handle the load;
  - Perf boost 2: Locality. We'll see this later when we discuss CDNs, can often direct client to a replica *near* it
  - Availability boost: Can fail-over (done at both DNS level -- slower, because clients cache DNS answers -- and at front-end hardware level)

But for read-write data...

- Must implement write replication, typically with some degree of consistency

Important ?: What consistency model?

- Just like in filesystems, want to look at the consistency model you supply
- R/L example: Google mail.
  - *Sending mail* is replicated to ~2 physically separated datacenters (users hate it when they think they sent mail and it got lost); mail will pause while doing this replication.
    - Q: How long would this take with 2-phase commit? in the wide area?
  - *Marking mail read* is only replicated in the background - you can mark it read, the replication can fail, and you'll have no clue (re-reading a read email once in a while is no big deal)
- Weaker consistency is cheaper if you can get away with it.

- Strict transactional consistency (you saw before)
- *sequentially consistent*: if client a executes operations {a1, a2, a3, ...} , b executes {b1, b2, b3, ...}, then you could create some serialized version (as if the ops had been performed through a single server) a1, b1, b2, a2, ... (or whatever) executed by the clients using a central server
  - Note this is *not* transactional consistency - we didn't enforce preserving happens-before. It's just per-program

Failure model

- We'll assume for today that failures and disconnections are relatively rare events - they may happen pretty often, but, say, any server is up more than 90% of the time.
- We'll come back later and look at "disconnected operation" models. In particular, a CMU system called Coda, that allowed AFS filesystem clients to work "offline" and then reconnect later. But not today. :)

Tools we'll assume

- Group membership manager
  - Allow replica nodes to join/leave
- Failure detector
  - e.g., process-pair monitoring, etc.

Goal

- Provide a service
- Survive the failure of up to *f* replicas
- Provide identical service to a non-replicated version (except more reliable, and perhaps different performance)

We'll cover today...

- Primary-backup
  - Operations handled by primary, it streams copies to backup(s)
- quorum consensus
  - Designed to have fast response time even under failures

Primary-Backup

- Clients talk to a primary
- The primary handles requests, atomically and idempotently, just like your lock server would
- Executes them
- Sends the request to the backups
- Backups reply, "OK"
- ACKs to the client

primary-backup

- Note: If you don't care about strong consistency (e.g., the "mail read" flag), you can reply to client *before* reaching agreement with backups (sometimes called "asynchronous replication").
- This looks cool. What's the problem?
  - What do we do if a replica has failed?
  - We wait... how long? Until it's marked dead.
  - Primary-backup has a strong dependency on the failure detector
- This is OK for some services, not OK for others
- Advantage: With N servers, can tolerate loss of N-1 copies

implementing primary-backup

- Remember logging? :-)
- Common technique for replication in databases and filesystem-like things: Stream the log to the backup. They don't have to actually apply the changes before replying, just make the log durable.
- You have to replay the log before you can be online again, but it's pretty cheap.

Problems with p-b

- Not a great solution if you want very tight response time even when something has failed
- For that, *quorum* based schemes are used
- As name implies, different result:
- To handle f failures, must have $2f + 1$ replicas (so that a majority is still alive)

Paxos [Lamport]

- quorum consensus usually boils down to the Paxos algorithm.
- *Very* useful functionality in big systems/clusters.
- Some notes in advance:
  - Paxos is painful to get right, particularly the corner cases. Steal an implementation if you can. See Yahoo's "Zookeeper" as a starting point.
  - There are lots of optimizations to make the common / no or few failures cases go faster; if you find yourself implementing, research these.
  - Paxos is *expensive*, as we'll see. Usually, used for critical, smaller bits of data and to coordinate cheaper replication techniques such as primary-backup for big bulk data.

Paxos requirement

- Correctness (safety):
  - –All nodes agree on the same value
  - –The agreed value X has been proposed by some node
- Fault-tolerance:
  - –If less than N/2 nodes fail, the rest should reach agreement *eventually w.h.p*
  - –Liveness is not *guaranteed*

Paxos: general approach

- Elect a replica to be the Leader
- Leader proposes a value and solicits acceptance from others
- If a majority ACK, the leader then broadcasts a commit message.

- This process may be repeated many times, as we'll see.

Paxos slides adapted from Jinyang Li, NYU; some terminology from "Paxos Made Live" (Google)

Why is agreement hard?

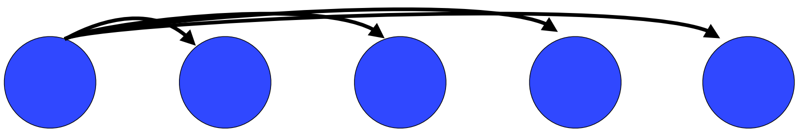
- What if >1 nodes think they're leaders simultaneously?
- What if there is a network partition?
- What if a leader crashes in the middle of solicitation?
- What if a leader crashes after deciding but before broadcasting commit?
- What if the new leader proposes different values than already committed value?

Basic two-phase

- Coordinator tells replicas: "Value V"
- Replicas ACK
- Coordinator broadcasts "Commit!"

- This isn't enough
  - –What if there's more than 1 coordinator at the same time? (let's solve this first)
  - – What if some of the nodes or the coordinator fails during the communication?

Combined leader election and two-phase



Prepare(N) -- dude, I'm the master

if N >= hN, Promise(N) -- ok, you're the boss. (I haven't seen anyone with a higher N)

if majority promised: Accept(V, N) -- please agree on the value V

if N >= nH, ACK(V, N) -- Ok!
if majority ACK: Commit(V)

Multiple coordinators

- The value N is basically a lamport clock.
- Nodes that want to be the leader generate an N higher than any they've seen before
- If you get NACK'd on the propose, back off for a while - someone else is trying to be leader
- Have to check N at later steps, too, e.g.:
- L1: N = 5 --> propose --> promise
- L2: N = 6 --> propose --> promise
- L1: N = 5 --> accept(V1, ...)
- Replicas: NACK! Someone beat you to it.
- L2: N = 6 --> accept(V2, ...)
- Replicas: Ok!

But...

- What happens if there's a failure? Let's say the coordinator crashes before sending the commit message
- Or only one or two of the replicas received it
-

Paxos solution

- Proposals are ordered by proposal #
- Each acceptor may accept multiple proposals
  - If a proposal with value v is chosen, all higher proposals must have value v

Paxos operation: node state

- Each node maintains:
  - n_a, v_a: highest proposal # and its corresponding accepted value
  - n_h: highest proposal # seen
  - my_n: my proposal # in current Paxos

Paxos operation: 3-phase protocol

- Phase 1 (Prepare)
  - A node decides to be leader (and propose)
  - Leader choose $my_n > n_h$
  - Leader sends <prepare, my_n> to all nodes
  - Upon receiving <prepare, n>

    If $n < n_h$

      reply <prepare-reject>

    Else

      $n_h = n$

      reply <prepare-ok, n_a,v_a>

See the relation to lamport clocks?

This node will not accept any proposal lower than n

Paxos operation

- Phase 2 (Accept):
  - If leader gets prepare-ok from a majority

    V = non-empty value corresponding to the highest n_a received

    If V= null, then leader can pick any V

    Send <accept, my_n, V> to all nodes
  - If leader fails to get majority prepare-ok
    - Delay and restart Paxos
  - Upon receiving <accept, n, V>

    If $n < n_h$

      reply with <accept-reject>

    else

      $n_a = n$; $v_a = V$; $n_h = n$

      reply with <accept-ok>

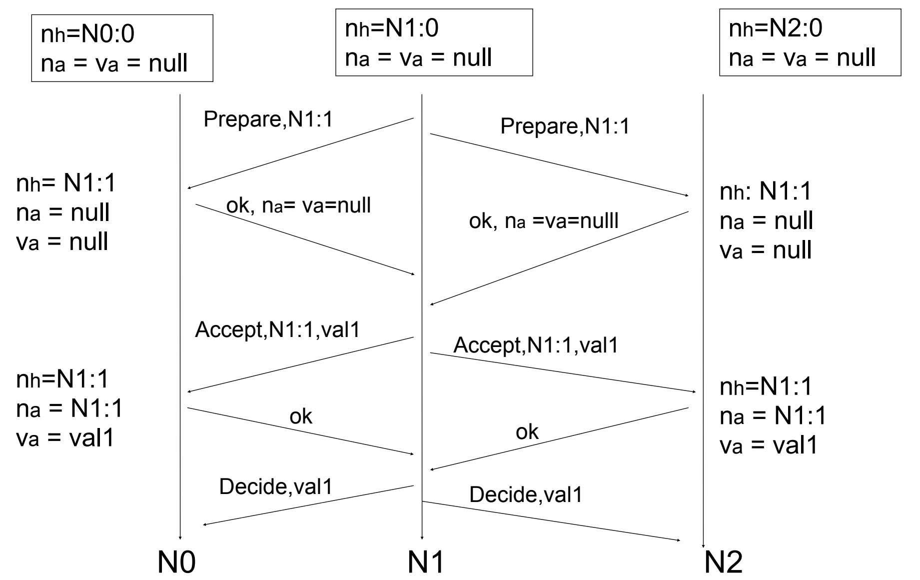
Paxos operation

- Phase 3 (Commit)
  - If leader gets accept-ok from a majority
    - Send <commit, v_a> to all nodes
  - If leader fails to get accept-ok from a majority
    - Delay and restart Paxos

Paxos Examples

- Failure after getting 1 node to accept the value
  - –One example where the master hears the value from one of the nodes
  - –One example where a new value wins
- Failure after getting > 1/2 nodes to accept the value
- Simultaneous failure of master and the 1 node that accepted in a 5 node system

Paxos operation: an example



Replication Wrap-Up

- Primary/Backup quite common, works well, introduces some time lag to recovery when you switch over to a backup. Doesn't handle as large a set of failures. f+1 nodes can handle f failures.
- Paxos is a general, quorum-based mechanism that can handle lots of failures, still respond quickly. 2f+1 nodes.