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 1: Get to use multiple servers to handle the load (scalability!)

• 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
What consistency model?

- Just like in distributed filesystems, must consider consistency model you supply
- R/L example: Google mail (mix of consistency models)
  - *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.
  - *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.
Goal

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

• Also known as the "replicated state machine" (RSM) abstraction
  • As with other abstractions (e.g., RPC), there are many ways to achieve/implement a RSM
We’ll cover

• Primary-backup
  • Operations handled by primary, it streams copies to backup(s)
  • Replicas are “passive”
  • Good: Simple protocol. Bad: Clients must participate in recovery.

• Quorum consensus using Paxos or Raft (later in the course)
  • Designed to have fast response time even under failures
  • Replicas are “active” - participate in protocol; there is no master, per se.
  • Good: Clients don’t even see the failures. Bad: More complex.
• Clients talk to a primary
• The primary handles requests, atomically and idempotently
• Executes them
• Sends the request to the backups
• Backups reply, “OK”
• Primary ACKs to the client
We describe the principle of this technique, and ignore failures for a while.

The principle is the following:

- $q_1$ (primary): the client sends its invocation to the primary $q_1$ which receives the invocation and performs the operation. At the end of the operation, the change of state of $q_1$ is forwarded to $q_2$ and $q_3$ ("update" message). The "ack" is sent by the backups after they have updated their states. The primary sends the response to the client after having received "ack" from all backups. If the primary does not crash, then order and atomicity are ensured.

- $q_2$ (backup): the order is defined by the primary.

- atomicity is ensured because the update is forwarded to all the backups and "ack" is waited by the primary before sending the response.

The crash of a backup is easy to handle. The crash of the primary is more difficult to handle. There are three cases to distinguish:

- replicas of object $q$
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?

• This is OK for some services, not OK for others

• Advantage: With N servers, can tolerate loss of N-1 copies
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
failures in p-b

- Use timeout-based failure detector for detection
- Backup failures: timeout and remove from set (later add new backups)
- Primary failures: complex because unclear when the primary failed (before/after replicating)
- Handling primary failures requires client participation
implementing primary-backup

- Remember logging (if you’ve taken databases)
- 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 (i.e., on disk).
- You have to replay the log before you can be online again, but it’s pretty cheap.
p-b: Did it happen?

Failure here:
Commit logged only at primary

Primary dies? **Client** must re-send to backup
(idempotency important)
p-b: Happened twice

Client

Primary

Backup

Operation

Commit!

Log

OK!

Log

OK!

Log

OK!

Failure here:
Commit logged at backup

Primary dies? Client must check with backup
(Seems like at-most-once / at-least-once... :)

We describe the principle of this technique, and ignore failures for a while.
Problems with p-b

• Not a great solution if you want very tight response time even when something has failed: Must wait for failure detector

• For that, *quorum* based schemes are used

• As name implies, different result:

  • To handle $f$ failures, must have $2f + 1$ replicas. *Why?*
Problems with p-b

- Client must be involved in primary recovery
- Requires client state (at least operation + id)
- Client must be aware of backups (violates the RSM abstraction)
- Bringing up a new primary is complicated
  - All clients must sign off on their outstanding ops
  - Vote a new backup to become primary?
  - Download all state to new primary?
Problems with p-b

• Not a great solution if you want very tight response time even when something has failed: Must wait for failure detector

• For that, *quorum* based schemes are used

• As name implies, different result:
  
  • To handle $f$ failures, must have $2f + 1$ replicas. *Why?* so that a majority $(f+1)$ is still alive after $(f)$ failures