Replication

Feb 10, 2016
CPSC 416
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
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 (e.g., Coda file system that allows clients to work “offline”)
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 as a non-replicated version (except more reliable, and perhaps different performance)

(A lot like your assignment 4 (where $f = r-1$) except without durable storage)
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
  • 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.
Primary-Backup

- Clients talk to a primary
- The primary handles requests, atomically and idempotently
- Executes them
- Sends the request to the backups
- Backups reply, “OK”
- ACKs to the client
• 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
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?

Client → Primary → Backup

Commit!

Log

OK!

Commit!

Log

OK!

Failure here:
Commit logged only at primary
Primary dies? Client must re-send to backup
p-b: Happened twice

Client → Primary → Backup

Commit!

Commit!

Commit logged at backup

Failure here:

Primary dies? Client must check with backup

(Seems like at-most-once / at-least-once... :)
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

• 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 is still alive