Distributed Replication

Lecture 11, Oct 5th 2017
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
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
Sequential Consistency (1)

- Behavior of two processes operating on the same data item. The horizontal axis is time.

- P1: Writes “W” value a to variable “x”

- P2: Reads `NIL’ from “x” first and then `a’
Sequential Consistency (2)

• A data store is sequentially consistent when:

• The result of any execution is the same as if the (read and write) operations by all processes on the data store …

• Were executed in some sequential order and …

• the operations of each individual process appear …
  ▪ in this sequence
  ▪ in the order specified by its program.
Sequential Consistency (3)

(a) A sequentially consistent data store.

(b) A data store that is not sequentially consistent.
Causal Consistency (I)

• For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

• Writes that are potentially causally related …
  • must be seen by all processes
  • in the same order.

• Concurrent writes …
  • may be seen in a different order
  • on different machines.
Causal Consistency (2)

<table>
<thead>
<tr>
<th>P1</th>
<th>W(x)a</th>
<th>W(x)c</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>R(x)a</td>
<td>W(x)b</td>
</tr>
<tr>
<td>P3</td>
<td>R(x)a</td>
<td>R(x)c</td>
</tr>
<tr>
<td>P4</td>
<td>R(x)a</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>

Figure 7-8. This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.
Causal Consistency (3)

<table>
<thead>
<tr>
<th></th>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P2:</td>
</tr>
<tr>
<td></td>
<td>R(x)a     W(x)b</td>
</tr>
<tr>
<td>P3:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(x)b     R(x)a</td>
</tr>
<tr>
<td>P4:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(x)a     R(x)b</td>
</tr>
</tbody>
</table>

- Figure 7-9. (a) A violation of a causally-consistent store.
Causal Consistency (4)

- Figure 7-9. (b) A correct sequence of events in a causally-consistent store.
Important ?: What is the consistency model?

- Just like in filesystems, want to look at the consistency model you supply
- Real Life 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.
Replicate: State versus Operations

Possibilities for what is to be propagated:

• Propagate only a notification of an update.
  - Sort of an “invalidation” protocol

• Transfer data from one copy to another.
  - Read-to-Write ratio high, can propagate logs (save bandwidth)

• Propagate the update operation to other copies
  - Don’t transfer data modifications, only operations – “Active replication”
When to Replicate: Pull versus Push Protocols

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State at server</td>
<td>List of client replicas and caches</td>
<td>None</td>
</tr>
<tr>
<td>Messages sent</td>
<td>Update (and possibly fetch update later)</td>
<td>Poll and update</td>
</tr>
<tr>
<td>Response time at client</td>
<td>Immediate (or fetch-update time)</td>
<td>Fetch-update time</td>
</tr>
</tbody>
</table>

Comparison between push- and pull-based protocols in the case of multiple-client, single-server systems.

- Pull Based: Replicas/Clients poll for updates (caches)
- Push Based: Server pushes updates (stateful)
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 looked at “disconnected operation” models. For example, the CMU CODA system, that allowed AFS filesystem clients to work “offline” and then reconnect later.
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)
  • Replicas are “passive”, i.e. follow the primary
  • 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, just like your lock server would
• Executes them
• Sends the request to the backups
• Backups reply, “OK”
• ACKs to the client
Remote-Write PB Protocol

W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Updates are blocking, although non-blocking possible
Local-Write P-B Protocol

Primary migrates to the process wanting to process update
For performance, use non-blocking op.
What does this scheme remind you of?
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 P-B

- 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.
p-b: Did it happen?

Client → Primary → Backup

Commit!

Log → 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! → Log

OK! → OK! → Log

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

(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 (so that a majority is still alive)

• Also, for replicated-write => write to all replica’s not just one.
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. Start from a good 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: fault tolerant consensus

• Paxos lets all nodes agree on the same value despite node failures, network failures and delays

• Some good use Cases:
  e.g. Nodes agree that X is the primary
e.g. Nodes agree that W should be the most recent operation executed
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

- Termination (not guaranteed)
Fischer-Lynch-Paterson [FLP’85] impossibility result

• It is impossible for a set of processors in an asynchronous system to agree on a binary value, even if only a single processor is subject to an unannounced failure.

• **Synchrony** --> bounded amount of time node can take to process and respond to a request

**Asynchrony** --> timeout is not perfect
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?
Paxos setup

- Each node runs as a proposer, acceptor and learner

- Proposer (leader) proposes a value and solicit acceptance from acceptors

- Leader announces the chosen value to learners
Combined leader election and two-phase

Prepare(N) -- dude, I’m the master

if $N \geq n_H$, 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 $N \geq n_H$, 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
  – $m_y n$: my proposal # in current Paxos
Paxos operation: 3-phase protocol

• Phase 1 (Prepare)
  – A node decides to be leader (and propose)
  – Leader chooses \( my_n > n_h \)
  – Leader sends \(<\text{prepare}, \ my_n>\) to all nodes
  – Upon receiving \(<\text{prepare}, n>\)
    
    If \( n < n_h \)
    
    reply \(<\text{prepare-reject}>\)
    
    Else
    
    \( n_h = n \)
    
    reply \(<\text{prepare-ok, n}_a,v_a}>\)
Paxos operation

• Phase 2 (Accept):
  – If leader gets prepare-ok from a majority
    V = non-empty value corresponding to the highest na received
    If V = null, then leader can pick any V
    Send <accept, myn, V> to all nodes
  – If leader fails to get majority prepare-ok
    • Delay and restart Paxos
  – Upon receiving <accept, n, V>
    If n < nh
      reply with <accept-reject>
    else
      na = n; va = V; nh = n
      reply with <accept-ok>
Paxos operation

• Phase 3 (Commit)
  – If leader gets accept-ok from a majority
    • Send <commit, va> to all nodes
  – If leader fails to get accept-ok from a majority
    • Delay and restart Paxos
Paxos operation: an example

\[ \text{nh= N0:0} \quad \text{na = va = null} \]

\[ \text{nh= N1:0} \quad \text{na = va = null} \]

\[ \text{nh= N2:0} \quad \text{na = va = null} \]

\[ \text{nh= N1:1} \quad \text{na = null} \quad \text{va = null} \]

\[ \text{nh= N1:1} \quad \text{na = N1:1} \quad \text{va = val1} \]

\[ \text{nh= N1:1} \quad \text{na = val1} \quad \text{va = val1} \]

\[ \text{nh= N1:1} \quad \text{na = N1:1} \quad \text{va = val1} \]

\[ \text{nh= N1:1} \quad \text{na = null} \quad \text{va = null} \]
Paxos: dueling proposers

- Duelling proposers (leader)
  - Violates Liveness
  - Likelihood that proposers observe each other and let one go first.
  - Or have a leader election

Source: http://the-paper-trail.org/blog/consensus-protocols-paxos/
Paxos properties

• When is the value $V$ chosen?

1. When leader receives a majority prepare-ok and proposes $V$

2. When a majority nodes accept $V$

3. When the leader receives a majority accept-ok for value $V$
Paxos is widespread!

• Industry and academia
  • Google: Chubby (distributed lock service)
  • Yahoo: Zookeeper (distributed lock service)
  • MSR: Frangipani (distributed lock service)

• OpenSource implementations
  ■ Libpaxos (paxos based atomic broadcast)
  ■ Zookeeper is open source, integrated w/Hadoop

Paxos slides adapted from Jinyang Li, NYU;
Paxos History

• It took 25 years to come up with safe protocol
  – 2PC appeared in 1979 (Gray)
  – In 1981, a basic, unsafe 3PC was proposed (Stonebraker)
  – In 1998, the safe, mostly live Paxos appeared (Lamport), 2001 ”Paxos made simple”.
  – In ~2007 RAFT appears
Understanding Paxos
(for you to think about)

- What if more than one leader is active?
- Suppose two leaders use different proposal number, N0:10, N1:11
- Can both leaders see a majority of prepare-ok?
Understanding Paxos
(for you to think about)

• What if leader fails while sending accept?

• What if a node fails after receiving accept?
  • If it doesn’t restart …
  • If it reboots …

• What if a node fails after sending prepare-ok?
  • If it reboots …
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.
Beyond PAXOS

• Many follow ups and variants
• RAFT consensus algorithm
  • https://raft.github.io/
• Great visualization of how it work
  • http://thesecretlivesofdata.com/raft/