Distributed Systems

15-640 (Section B)
Fall 2017

11 – Distributed Replication
Organizational Updates

• P1
  • part A checkpoint individual submissions
  • part B will be released next week

• Lecture on DNS moved to before midterm
  • Plan: no lectures in midterm week

• Final exam will be on Sunday, Dec 17
Fault Tolerance Techniques So Far?

- Redundancy: information / time / physical redundancy
  - E.g., used in airplanes

- Recovery: checkpointing and logging (ARIES)
  - E.g., used in commercial databases

- Previous (concurrency) protocols rely on recovery techniques
  - E.g., Two Phase Commit is not fault tolerant by itself

• Why not always use these techniques?
  → Long wait in case of failure
Our Goal Today: Stay Up During Failures

• Provide a service
• Replicate the machines that serve clients
• Survive the failure of up to $f$ replicas
• Provide identical service to a non-replicated version
  • (except more reliable, and perhaps different performance)
Outline for Today

Consistency when content is replicated

Primary-backup replication model

Consensus replication model
Simple Examples of Replication

• 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

• When is replication easy? When hard?
  • Workload assumptions
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 **Read-write** Data...

- Requires write replication, and some degree of consistency
  - **Strict** Consistency
    - Read always returns value from latest write
  - **Sequential** Consistency
    - All nodes see operations in some sequential order
    - Operations of each process appear in-order in this sequence
Sequential Consistency (1)

P1: \( W(x)a \)

P2: \( R(x)\text{NIL} \) \( R(x)a \)

• 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 ‘\( \text{NIL} \)’ from “\( x \)” first and then ‘\( a \)’

Adapted from: Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall, Inc. All rights reserved. 0-13-239227-5
Sequential Consistency (2)

(a) A sequentially consistent data store.

(b) A data store that is not sequentially consistent.
But **Read-write Data**...

- Requires write replication, and some degree of consistency
  - **Strict** Consistency
    - Read always returns value from latest write
  - **Sequential** Consistency
    - All nodes see operations in some sequential order
    - Operations of each process appear in-order in this sequence
  - **Causal Consistency**
    - All nodes see causally related writes in same order
    - But concurrent writes may be seen in different order on different machines
Causal Consistency (1)

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

This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.
Causal Consistency (2)

A violation of a causally-consistent store.

(W(x)a causally related to R(x)a, W(x)b.)
But Read-write Data...

• Requires write replication, and some degree of consistency
  • **Strict** Consistency
    • Read always returns value from latest write
  • **Sequential** Consistency
    • All nodes see operations in some sequential order
    • Operations of each process appear in-order in this sequence
  • **Causal Consistency**
    • All nodes see causally related writes in same order
    • But concurrent writes may be seen in different order on different machines
  • **Eventual Consistency**
    • All nodes will learn eventually about all writes, in the absence of updates
Example of Consistency Guarantees

• In practice we often have a choice

• 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.
Replication Strategies

What to replicate: State versus Operations

• 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: Push vs Pull

• **Pull Based**
  • Replicas/Clients poll for updates (caches)

• **Push Based**
  • Server pushes updates (stateful)
Outline for Today

Consistency when content is replicated

Primary-backup replication model

Consensus replication model
Assumptions Today

- **Group membership** manager
  - Allow replica nodes to join/leave

- **Fail-stop** (not Byzantine) failure model
  - Servers might crash, might come up again
  - Delayed/lost messages

- **Failure detector**
  - E.g., process-pair monitoring, etc.
Primary-Backup: Remote Write Protocol

• Writes always go to primary, read from any backup

• Implementation
  • Stream the log

• Common in practice
  • Simple

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
Local-Write P-B Protocol

Primary migrates to the process wanting to process update
For performance, use non-blocking op.
Primary-Backup Properties

• This looks cool. How many failures can we deal with? What are some problems?
  • What do we do if a replica has failed?
  • We wait... how long? Until it’s marked dead.

• Advantage: With N servers, can tolerate loss of N-1 copies
• Not a great solution if you want very tight response time even when something has failed: Must wait for failure detector

• 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”).
Outline for Today

Consistency when content is replicated

Primary-backup replication model

Consensus replication model
Quorum Based Consensus

• Designed to have fast response time even under failures
• Operate as long as majority of machines is still alive
• No master, per se
• To handle f failures, must have 2f + 1 replicas
• Also, for replicated-write => write to all replica’s not just one
• Usually boils down to Paxos [Lamport]
The Paxos Approach

Decompose the problem:

• Basic Paxos (“single decree”):
  • One or more servers propose values
  • System must agree on a single value as chosen
  • Only one value is ever chosen

• Multi-Paxos:
  • Combine several instances of Basic Paxos to agree on a series of values forming the log

Requirements for Basic Paxos

• **Correctness** (safety):
  • Only a single value may be chosen
  • A machine never learns that a value has been chosen unless it really has been
  • The agreed value X has been proposed by some node

• **Liveness** (termination):
  • Some proposed value is eventually chosen
  • If a value is chosen, servers eventually learn about it

• Fault-tolerance:
  • If less than N/2 nodes fail, the rest should reach agreement eventually w.h.p
  • **Liveness is not guaranteed**
[FLP’85] Impossibility Result

- **Synchronous** DS: bounded amount of time node can take to process and respond to a request
- **Asynchronous** DS: timeout is not perfect

**Fischer-Lynch-Paterson 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.
Paxos Components

• Proposers:
  • Active: put forth particular values to be chosen
  • Handle client requests

• Acceptors:
  • Passive: respond to messages from proposers
  • Responses represent votes that form consensus
  • Store chosen value, state of the decision process

• For this presentation:
  • Each Paxos server contains both components
  • Ignore third role, aka Learner

• “Round”: (proposal, messages/voting, decision)
  • We may need several rounds
Strawman: 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?
  - What if new coordinator chooses a different value?
  - What if some of the nodes or the coordinator fails during the communication?
  - What if there is a network partition?
Let’s Discuss Some Problems & Solutions

• Problem: can’t trust a single node
  • Solution: everyone can potentially propose

• Problem: several concurrent proposers
  • Solution: Quorum (require majority of acceptors)

• Problem: split votes, no proposer reaches majority
  • Solution: acceptors need to allow updating of their value

• Problem: conflicting choices (due to updating)
  • Solution a): prioritize proposal with highest unique time stamp (Lamport clocks)
  • Solution b): once majority has agreed on value, future proposals forced to propose/choose same value
Single Decree Paxos: Informal Description

• **Phase 1: Prepare message**
  - Find out about any chosen values
  - Block older proposals that have not yet completed

• **Phase 2: Accept message**
  - Ask acceptors to accept a specific value

• *(Phase 3): Proposer decides*
  - If majority again: chosen value, commit.
  - If no majority: delay and restart Paxos

**Proposers**
- Prepare
- Accept
- Decision

**Acceptors**
- Check, Return
- Wait for majority
- Check Again, Return
**Single Decree Paxos: Protocol**

**Proposers**
1) Choose new proposal number \( n \), value \( v \)
2) Broadcast \( \text{Prepare}(n) \) to all servers
4) When responses received from majority:
   • If any \( \text{acceptedValues} \) returned
     \( v = \text{acceptedValue} \) of highest \( \text{acceptedProposal} \)
5) Broadcast \( \text{Accept}(n, \text{value}) \) to all servers

**Acceptors**
3) Respond to \( \text{Prepare}(n) \):
   • If \( n > \text{minProposal} \) then \( \text{minProposal} = n \)
     \( \text{Prepare-OK}(\text{acceptedProposal}, \text{acceptedValue}) \)
   else
     \( \text{Prepare-REJECT()} \)
6) Respond to \( \text{Accept}(n, \text{value}) \):
   • If \( n \geq \text{minProposal} \)
     \( \text{acceptedProposal} = \text{minProposal} = n \)
     \( \text{acceptedValue} = \text{value} \)
     \( \text{Accept-OK()} \)
   else
     \( \text{Accept-REJECT()} \)

6) When Accept-OK from majority
   - **Value is chosen (Commit)**
   - Else
     - Restart: goto 1, with larger number \( n \)

**Acceptors must record** \( \text{minProposal} \), \( \text{acceptedProposal} \), and \( \text{acceptedValue} \) on stable storage (disk)
Paxos Examples

a) Successful Round with a Single Proposer
b) Dueling Proposers
Some Remarks

• Only proposer knows chosen value (majority accepted)
• Only a single value is chosen $\rightarrow$ MultiPaxos
• No guarantee that proposer’s original value $v$ is chosen by itself

• Number $n$ is basically a Lamport clock $\rightarrow$ always unique $n$
• Key invariant:
  • If a proposal with value `v` is chosen, all higher proposals must have value `v`

• Dueling proposer
  • Resolved using number $n$ in prepare

• There are challenging corner cases
Single Decree Paxos: Protocol

**Proposers**

1) Choose new proposal number \( n \), value \( v \)

2) Broadcast \( \text{Prepare}(n) \) to all servers

4) When responses received from majority:
   - If any acceptedValues returned
     \( v = \text{acceptedValue} \) of highest acceptedProposal

5) Broadcast \( \text{Accept}(n, \text{value}) \) to all servers

6) When Accept-OK from majority
   - Value is chosen (Commit)
   - Else
     Restart (goto 1, with larger number \( n \))

**Acceptors**

3) Respond to \( \text{Prepare}(n) \):
   - If \( n > \text{minProposal} \) then \( \text{minProposal} = n \)
     \( \text{Prepare-OK}(\text{acceptedProposal}, \text{acceptedValue}) \)
   - Else
     \( \text{Prepare-REJECT()} \)

6) Respond to \( \text{Accept}(n, \text{value}) \):
   - If \( n \geq \text{minProposal} \)
     \( \text{acceptedProposal} = \text{minProposal} = n \)
     \( \text{acceptedValue} = \text{value} \)
     \( \text{Accept-OK()} \)
   - Else
     \( \text{Accept-REJECT()} \)

Acceptors must record \( \text{minProposal}, \text{acceptedProposal}, \) and \( \text{acceptedValue} \) on stable storage (disk)
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 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
Multi-Paxos

- Separate instance of Basic Paxos for each new decision
  - Add \texttt{index} argument to Prepare and Accept (selects entry in log)
- E.g., distributed log

![Diagram of multi-Paxos]

- Performance optimizations:
  - Use leader to reduce proposer conflicts
  - Eliminate most Prepare requests
More Remarks

• Multi-Paxos not clearly defined in literature. Slight variations in practice.

• 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. Usually, used for critical, smaller bits of data and to coordinate cheaper replication techniques such as primary-backup for big bulk data.
Summary

• Primary-backup
  • Writes handled by primary, stream log to backup(s)
  • Replicas are “passive”, follow primary
  • Good: Simple protocol. N machines, can handle N-1 failures
  • Bad: Slow response times in case of failures.

• 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 (corner cases). To handle $f$ failures, must have $2f + 1$ replicas.