Motivating Paxos by looking at consensus

Assumptions (rather weak ones, and realistic)
- System is partially synchronous (may even be asynchronous).
- Communication between processes may be unreliable:
  - messages may be lost, duplicated, or reordered.
- Corrupted messages can be detected
  - and thus subsequently ignored
- All values are deterministic:
  - once an execution is started, it is known exactly what it will do.
- Processes may exhibit crash failures, but not arbitrary failures.
- Processes do not collude.

Understanding Paxos
- We will build up to Paxos by looking at problems that occur.

Two Servers leader + backup
- The leader sends an accept message $\text{ACCEPT}(o, t)$ to backups when assigning a timestamp $t$ to command $o$. 
Two Servers and a crash!

Problem
Servers have diverged because primary crashes after executing an value, but the backup never received the accept message.

Solution
- A backup responds by sending a learn message: \text{LEARN}(o, t)
- When the leader notices that value $o$ has not yet been learned, it retransmits \text{ACCEPT}(o, t) with the original timestamp.

\textit{Never commit an value before it is clear that it has been learned.}
Three servers and two crashes: still a problem?

Scenario:
- Assume reliable fault detection.
- $S_1$ is waiting for a majority before committing (and gets it when it hears from $S_2$)
- But if $S_1, S_2$ crash there is no guarantee $S_3$ knows anything…… and $S_3$ commits $o^2...$ bad!

One possible solution:
- No server should commit until it gets learns from all non-failed servers.
- However, this is a high bar, and reliable fault detection is impossible, so need something else.

Fundamental Rule

Another approach: a server $S$ cannot commit an value $o$ until it has received a LEARN($o$) from a majority of learners.

Practice

Reliable failure detection is practically impossible. A solution is to set timeouts, but accept that a detected failure may be false.

S1, S2 opposite sides of a partition
- Each think the other has crashed. Who’s the real leader? (neither)
- Majorties to commit values necessary:
  - Any two majorities are guaranteed to intersect - intersection property guarantees knowledge of past commits is never lost.
So Consensus Needs at Least Three Servers

Adapted fundamental rule
- With three servers, a server S cannot commit an value v until it has received at least one (other) \text{LEARN}(v) message, so that it knows that \textit{a majority of servers will commit v}.

Assumptions before taking the next steps:
- Initially, S_1 is the leader.
- A server can \textit{reliably detect it has missed a message}, and recover from that miss (timestamps, message IDs, ask for resends, etc.).
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as S_1 \rightarrow S_2 \rightarrow S_3.
- A client cannot be asked to help the servers to resolve a situation.

Observation:
If either one of the backups (S_2 or S_3) crashes, consensus still correct:
- values at nonfaulty servers are committed in the same order.

Example Failures \textit{w/ correct recovery}

Leader crashes after executing \(o_1\)

\(S_3\) is completely ignorant of any activity by \(S_1\)
S_2 received \text{ACCEPT}(o_1, 1), detects crash, and becomes leader. S_3 never received \text{ACCEPT}(o_1, 1)
If S_2 sends \text{ACCEPT}(o_2, 2), S_3 sees unexpected timestamp and tells S_2 that it missed timestamp 1. S_2 retransmits \text{ACCEPT}(o_1, 1), allowing S_3 to catch up.

\(S_2\) missed \text{ACCEPT}(o_1, 1)
S_2 detects crash and becomes new leader
If S_2 sends \text{ACCEPT}(o_1, 1) \Rightarrow S_3 retransmits \text{LEARN}(o_1).
If S_2 sends \text{ACCEPT}(o_2, 1) \Rightarrow S_3 tells S_2 that it apparently missed \text{ACCEPT}(o_1, 1) from S_1, so that S_2 can catch up.
Example Failures
Leader crashes after sending ACCEPT(o₁, 1):

- S₃ is completely ignorant of any activity by S₁
- As soon as S₂ announces that o² is to be accepted, S₃ will notice that it missed an value and can ask S₂ to help recover.

- S₂ had missed ACCEPT(o₁, 1)
- As soon as S₂ proposes an value, it will be using a stale timestamp, allowing S₃ to tell S₂ that it missed value o₁.

Observation
Consensus (with three servers) behaves correctly when a single server crashes, regardless of when that crash took place.

False Crash Detections

Problem and solution
S₃ receives ACCEPT(o₁, 1), but much later than ACCEPT(o², 1). If it knew who the current leader was, it could safely reject the delayed accept message
⇒ leaders should include their ID in messages.
But What About Progress?

Problem:
When S3 crashes no other server knows what it did.

Essence of solution
When S2 takes over, it needs to make sure that any outstanding values initiated by S1 have been properly flushed, i.e., committed by enough servers. This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages.

Terminology
- proposed value same as Steen’s operation
- value commit same as Steen’s execute
- accept / learn are second phase not first as we have seen
Paxos  

original “single decree” Paxos

- Server roles:
  - proposer: attempts proposes client’s command
  - acceptor: accepts a proposed command
  - learner: learns of acceptances
  - Once a server learns a majority have accepted a proposal, it can be accepted and result sent to the client.
  - All roles often played by each server

Paxos  

phases

- A proposal has:
  - timestamp, or “proposal number”, or “ID”
  - value, “value”
- We want correctness and liveness, so:
  - there can be concurrent proposals (by different servers)
  - phase 1: arbitrate between competing proposals
    - proposer sends a prepare msg w/ proposal number, \( n \), and value \( o_n \), to each acceptor
    - If the prepare’s \( n \) is higher than any previously seen proposal, an acceptor promises to ignore later proposals with lower or same numbers
  - phase 2: decide on accepted value
    - proposer sends accept w/ its timestamp, and value from previously promise (or it’s own value if none)
    - acceptors respond accepted, and tell all learners

proposer can time out and restart w/ higher proposal number
**Paxos**

*single proposer*

- Client sends proposal to random *proposer*
- Proposer sends *prepare* with bigger proposal number (ID) than previously seen
- Acceptors:
  - do not respond if already promised w/ ID >= 1, or
  - respond with:
    - *promise* not to accept proposal w/ ID <= 1
    - value of highest proposal it has *promised* so far
    - this promise returns *null,null* because it is the first seen *prepare* msg
- If majority promises, proposer sends *accept* with:
  - ID and value from proposal w/ highest ID promised by an acceptor
- If learner gets *accepted* from majority of acceptors, proposal is *committed*

---

**Paxos**

*proposer (server leader) failure*

<table>
<thead>
<tr>
<th>Clients</th>
<th>Proposers</th>
<th>Acceptor</th>
<th>Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 C2</td>
<td>P1 P2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- *P₁* fails, client request fails
- Acceptors all append *Vₐ,1* to promises for proposal 2
- After gathering a majority, *P₂* sends *accept* with:
  - new proposal ID of 2
  - value *Vₐ* from highest proposal promised by any acceptor
- *P₂* is accepted, but value committed is actually from the earlier proposal (*Vₐ* from 1)

- Single-decree Paxos can accept multiple proposals, but:
  *all accepted values must be the same*
### Dueling Proposers

<table>
<thead>
<tr>
<th>Client</th>
<th>Proposer</th>
<th>Acceptor</th>
<th>Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------&gt;</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Request(Va)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Prepare(1.Va)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Prepare(1.null.null)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accept!(1.Va)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accepted(1.Va)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X-X-X-X Response</td>
</tr>
</tbody>
</table>

- progress not guaranteed…

### Multiple Single Decrees

<table>
<thead>
<tr>
<th>Client</th>
<th>Proposer</th>
<th>Acceptor</th>
<th>Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------&gt;</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Request(Vb)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Prepare(1.Vb)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Prepare(1.null.null)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X--------&gt;]</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accept!(1.Vb)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accepted(1.Vb)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X-X-X-X Response</td>
</tr>
</tbody>
</table>

- each round takes two round trips (not counting client)
  - first identifies a leader
  - second gets value accepted
  - maybe we can dispense w/ the first…
Multi-Paxos *chasing performance*

Multi-Paxos Collapsed Roles

<table>
<thead>
<tr>
<th>Client</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-------&gt;</td>
<td></td>
</tr>
<tr>
<td>X-&gt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;-X--X Promise(N)</td>
</tr>
<tr>
<td>X-&gt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td>X&lt;&gt;X&lt;&gt;X Accepted(N,0,V0)</td>
<td></td>
</tr>
<tr>
<td>&lt;--------X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-------&gt;</td>
<td></td>
</tr>
<tr>
<td>X-&gt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td>X&lt;&gt;X&lt;&gt;X Accepted(N,1)</td>
<td></td>
</tr>
<tr>
<td>&lt;--------X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-------&gt;</td>
<td></td>
</tr>
<tr>
<td>X-&gt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td>X&lt;&gt;X&lt;&gt;X Accepted(N,2)</td>
<td></td>
</tr>
<tr>
<td>&lt;--------X</td>
<td></td>
</tr>
</tbody>
</table>

- stable leader allows *one round per committed value*
- competing leader starts everything all over

*multi-paxos implementations usually do not change accepted values (promise only returns the proposal number)*

derived from wikipedia