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 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: \texttt{LEARN(o, t)}
- When the leader notices that value \( o \) has not yet been learned, it retransmits \texttt{ACCEPT(o, t)} with the original timestamp.

Never commit an value before it is clear that is 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)

Majorities 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 $o$ until it has received at least one (other) LEARN$(o)$ message, so that it knows that *a majority of servers will commit o*.

Assumptions before taking the next steps:

- Initially, $S_1$ is the leader.
- A server can *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 *w/ correct recovery*

Leader crashes after executing $o_1$

$S_3$ is completely ignorant of any activity by $S_1$

$S_2$ received ACCEPT$(o^1, 1)$, detects crash, and becomes leader. $S_3$ never received ACCEPT$(o^1, 1)$

If $S_2$ sends ACCEPT$(o^2, 2)$, $S_3$ sees unexpected timestamp and tells $S_2$ that it missed timestamp 1. $S_2$ retransmits ACCEPT$(o^1, 1)$, allowing $S_3$ to catch up.

$S_2$ missed ACCEPT$(o^1, 1)$

$S_2$ detects crash and becomes new leader

If $S_2$ sends ACCEPT$(o^1, 1)$ ⇒ $S_3$ retransmits LEARN$(o^1)$.

If $S_2$ sends ACCEPT$(o^2, 1)$ ⇒ $S_3$ tells $S_2$ that it apparently missed ACCEPT$(o^1, 1)$ from $S_1$, so that $S_2$ can catch up.
Example Failures

Leader crashes after sending ACCEPT(o\(^1\), 1):

\( S_3 \) is completely ignorant of any activity by \( S_1 \).

As soon as \( S_2 \) announces that \( o^2 \) is to be accepted, \( S_3 \) will notice that it missed an value and can ask \( S_2 \) to help recover.

\( S_2 \) had missed ACCEPT(o\(^1\), 1)

As soon as \( S_2 \) proposes an value, it will be using a stale timestamp, allowing \( S_3 \) to tell \( S_2 \) that it missed value \( o^1 \).

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

\( S_3 \) receives ACCEPT(o\(^1\), 1), but much later than ACCEPT(o\(^2\), 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 \textit{value} same as Steen’s \textit{operation}
- value \textit{commit} same as Steen’s \textit{execute}

- \textit{accept / learn} are second phase not first as we have seen
Paxos \textit{original “single decree” Paxos}

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

\begin{center}
\[\begin{array}{ccc}
C_1 & S_2 & C_2 \\
S_1 & S_3 & \\
\end{array}\]
\end{center}

Paxos \textit{phases}

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

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

*single proposer*

<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></td>
<td>Request(Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prepare(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promise(1, null, null)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accept!(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepted(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Response</td>
<td></td>
</tr>
</tbody>
</table>

Client sends proposal to random proposer

Proposer send prepare w/ 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>
<tr>
<td>X-------</td>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request(Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prepare(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promise(1, null, null)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accept!(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepted(1, Va)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Response</td>
<td></td>
</tr>
</tbody>
</table>

- $P_1$ fails, client request fails
- Acceptors all append $V_a,1$ to promises for proposal 2
- After gathering a majority, $P_2$ sends accept with:
  - new proposal ID of 2
  - Value $V_a$ from highest proposal promised by any acceptor
- $P_2$ is accepted, but value committed is actually from the earlier proposal ($V_a$ 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>-&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| | | | | | | | | ... and so on ...

- progress not guaranteed...

derived from wikipedia

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>-&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X---</td>
<td>------</td>
<td>-&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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-------</td>
<td>Request(V0)</td>
</tr>
<tr>
<td>X-------</td>
<td>Accept!(N, 0, V0)</td>
</tr>
<tr>
<td>X-------</td>
<td>Request(V1)</td>
</tr>
<tr>
<td>X-------</td>
<td>Accepted(N, 1)</td>
</tr>
<tr>
<td>X-------</td>
<td>Request(V2)</td>
</tr>
<tr>
<td>X-------</td>
<td>Accepted(N, 2)</td>
</tr>
</tbody>
</table>

- Stable leader allows one round per committed value.
- Competing leader starts everything all over.

---

#### Multi-Paxos Interrupted!

<table>
<thead>
<tr>
<th>Client</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-------</td>
<td>Request(V0)</td>
<td>Prepare(N, V0)</td>
<td></td>
</tr>
<tr>
<td>X-------</td>
<td>Accept!(N, 0, V0)</td>
<td>Accepted(N, 0, V0)</td>
<td></td>
</tr>
<tr>
<td>X-------</td>
<td>Request(V1)</td>
<td>Accept!(N, 1, V1)</td>
<td></td>
</tr>
<tr>
<td>X-------</td>
<td>Accepted(N, 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-------</td>
<td>Request(V2)</td>
<td>Accept!(N, 2, V2)</td>
<td></td>
</tr>
<tr>
<td>X-------</td>
<td>Accepted(N, 2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- S2 takes over.
- S1 accept fails.
- S1 succeeds.
- S1 back on track.

---

*Derived from Wikipedia*
Q1 Leader/backup
2 Points

Q1.1
1 Point
What issues make this example of a leader/backup system incorrect?

- $S_1$ and $S_2$ commit different values first
- $S_1$’s accept msg does not make it to $S_2$.
- $S_2$’s accept msg does not make it to $S_1$.
- all of the above
- none of the above

Q1.2
1 Point
How can it be fixed?

Explanation
wait for an acknowledgement of the accept message before committing the value.

Q2 Leader/backup
3 Points

Q2.1
1 Point
Why did $S_1/S_2$ and $S_3$ diverge?

Explanation
It did not communicate with enough other servers to find out about the prior committed values.

Q2.2
1 Point
How to fix this?

Explanation
Require any leader to talk to a majority of all servers, including itself, before committing a value. Since all majority quorums overlap, at least one of the quorum members will have seen any prior value commits.

Q2.3
1 Point
Will this example nonetheless give the correct answers?

Explanation
No. In more detail, committing value in different orders is expected to result in different final results, unless all the values commute.
Q3 Leader/backup
1 Point

What's wrong w/ this example (still not paxos yet):

Explanation
S2 replied to client before S3 knew that any other servers knew about the choice.

Q4 Paxos!
4 Points

Q4.1
1 Point

Can an acceptor promise more than one proposal?

- yes
- no

Q4.3
1 Point

Now additionally assume that:

- all acceptors see the same sequence of prepare messages, and that
- all prepare and propose messages are received and processed before any accept messages are ever sent.

Which proposals are eventually accepted (w/ an accepted message)?

- 1/o1
- 5/o5
- 5/o5
- 4/o4
- 8/o8

Explanation
All of the acceptors promise proposals 1, 5, and then 8. Because of the last one, they cannot accept any lower ordered proposals.

Q4.4
1 Point

Which value is finally accepted and committed?

- o1
- o2
- o4
- o5
- o6

Explanation
A proposer only uses its own value if no promise tells it of any prior promised values. If it has seen such values, it chooses the value of the max previously promised proposal.
Final Exam

- 32 pts comprehensive
  - fill in the blank
- 36 pts GeekOS
  - should be straightforward if you worked through the projects
  - short answer
- 32 pts Paxos/MultiPaxos
  - 14 pts true/false
  - 6 pts fill-in-the-blanks
  - 12 pts short answer