The ABCDs of Paxos

Replicated state machines

Consensus: a set of processes \emph{decide} on an input value

Paxos asynchronous consensus algorithm

\begin{itemize}
\item[AP] Abstract Paxos: \quad generic, non-local version
\item[CP] Classic Paxos: \quad stopping failures, compare-and-swap
\quad 1989: Lamport, Liskov and Oki
\item[DP] Disk Paxos: \quad stopping failures, read-write
\quad 1999: Gafni and Lamport
\item[BP] Byzantine Paxos: \quad arbitrary failures
\quad 1999: Castro and Liskov
\end{itemize}

The paper is at research.microsoft.com/lampson
Replicated State Machines

Lamport 1978: *Time, clocks and the ordering of events* …

Cast your problem as a deterministic state machine
   Takes client input requests for state transitions, called *steps*
   Performs the steps
   Returns the output to the client.

Make \( n \) copies or ‘replicas’ of the state machine.

Use consensus to feed all the replicas the same inputs.

Steps must be deterministic, local to replica, atomic (use transactions)
Recover by replaying the steps (like transactions)
Even a read needs a step, unless the result is “as of step \( n \)”. 
Applications of RSM

Reliable, available data storage system
Airplane flight control
Reflexive: Changing quorums of the consensus algorithm

Issuing a lease:
A lock on part of the state that times out, hence is fault tolerant
Leaseholder can work on its state without consensus
Like any lock, a lease can have modes or be hierarchical
The Idea of Paxos

A sequence of *views*; get a decision quorum in one of them.

Each view *v* *chooses* an *anchored value* $c_v$: equals any earlier decision.

If a quorum *accepts* the choice, decision!

Decision is irrevocable, may be invisible, but is any later view’s choice.

Choice is changeable, must be visible

---

Processes

$\begin{align*}
\text{a} & \rightarrow a \\
\text{a} & \rightarrow a \\
\text{a} & \rightarrow a
\end{align*}$

$\begin{align*}
\text{c} & \rightarrow a \\
\text{a} & \rightarrow a \\
\text{a} & \rightarrow a
\end{align*}$

Actions

$\begin{align*}
\text{Start;} \\
\text{Input;} \\
\text{Close}^a \\
\text{Anchor} \\
\text{Choose} \\
\text{Accept}^a \\
\text{Finish}^a; \\
\text{STEP}^a
\end{align*}$

Transmit

$\begin{align*}
\text{r}^a \\
\text{INPUT} \\
\text{c}_v \\
\text{r}^a
\end{align*}$

\[\text{view change} \hspace{5em} \text{normal operation} \hspace{5em} \text{OUTPUT}\]
Design Methodology

- Communicate only *stable* predicates: once true always true
- Structure program as a set of atomic actions
- Make actions as non-deterministic as possible: weakest guards
  
  Allows more freedom for the implementation
  Makes it clear what is essential

- Separate safety, liveness, and performance
  
  Safety first, then strengthen guards for liveness and scheduling

- Abstraction functions and simulation proofs
Notation

Subscripts and superscripts for function arguments: $r_v^a$ for $r(v, a)$

State functions used like variables

Actions described like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>State change</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Close_v$</td>
<td>$c_v = nil \land x \in anchor_v$</td>
<td>$? c_v := x$</td>
</tr>
</tbody>
</table>
Failure Model

A set $M$ of processes (machines)

A *faulty* process can send arbitrary messages: $F^m$

A *stopped* process does nothing: $S^m$

A *failed* process is faulty or stopped. Failure doesn’t lose state.

Limits on failure:

$Z_F = \text{set of sets of processes that can all be faulty}$

$Z_S = \text{set of sets of processes that can all be stopped}$

$Z_{FS} = \text{set of sets of processes that can all be failed}$

Examples:

Fail-stop: $n$ processes, $Z_F=\{\}$, $Z_S=Z_{FS}=\text{any set of size } < (n+1)/2$

Byzantine: $n$ processes, $Z_F = Z_S = Z_{FS} = \text{any set of size } < (n+1)/3$

Intel-Microsoft: $n_I + n_M$ processes, $Z_F = \text{any subset of one side}$
Quorums and Predicates

Quorum: monotonic set of sets of processes: \( q \) in \( \Rightarrow \) any superset in.
Predicates \( g \). Predicates on processes \( G \), so \( G^m \) is a predicate.
A **stable** predicate once true remains true.

A predicate \( G \) holds in a quorum \( Q \): \( Q\#G = \{ m \mid G^m \lor F^m \} \in Q \)

Shorthand: \( Q[r^*_v = x] \) for \( Q\#(\ ? m \mid r^m_v = x) \).

A **good** quorum is not all faulty: \( Q\sim_F = \{ q \mid q \notin Z_F \} \)

\( Q \) and \( Q' \) **exclusive**: \( Q \) quorum for \( G \) \( \Rightarrow \) no \( Q' \) quorum for its negation.

Means \( q \cap q' \in Q\sim_F \) for any two quorums. Ex: size > \((n + f)/2\)

Lifts local exclusion \( G_1 \Rightarrow \sim G_2 \) to global: \( Q\#G_1 \Rightarrow \sim Q'\#G_2 \)

\( Q^+ \): ensures \( Q \) even after failures: \( q^+ - z_{FS} \in Q \) for any \( q^+ \), \( z_{FS} \)

A **live** quorum has \( Q^+ \) ? \{\}
Specification

type $X$ = ...

values to decide on

var $d$ : $(X \cup \{nil\}) := nil$ Decision

$\text{input} : \text{set } X := \{\}$

Name Guard State change

$\text{Input}(x)$ $\text{input} := \text{input} \cup \{x\}$

$\text{Decision: } X \ d?\ nil$ ? ret $d$

$\text{Decide}$ $d = \text{nil} \land x \in \text{input}$ ? $d := x$
The Idea of Paxos

A sequence of views; get a decision quorum in one of them.

Each view $v$ chooses an anchored value $c_v$: equals any earlier decision. If a quorum accepts the choice, decision!

Decision is irrevocable, may be invisible, but is any later view’s choice. Choice is changeable, must be visible.

Processes

Actions

Transmit

normal operation
### Abstract Paxos—AP: State

<table>
<thead>
<tr>
<th>Non-local Agents</th>
<th>State functions</th>
<th>View is</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_v$</td>
<td>$Q_{dec}[r_v^* = x]$</td>
<td>decided</td>
</tr>
<tr>
<td>$input$</td>
<td>$Q_{out}[r_v^* = out]$</td>
<td>out nil open</td>
</tr>
<tr>
<td>$active_v$</td>
<td>else</td>
<td>nil nil open</td>
</tr>
</tbody>
</table>

**Example:**

1. $r_v^1$  
   $d_v^1$  

2. $r_v^2$  
   $d_v^2$  

3. $r_v^3$  
   $d_v^3$  

---

**Note:** The diagram represents the state transitions and decisions in Paxos protocol, where $r_v$ and $d$ denote the view and decision, respectively.
**AP: Data Flow**

to later views

\[
\begin{align*}
    r_u^a &= \text{nil} & \text{Close}_v \\
    x &\in \text{anchor}_v \\
    r_u^a &= \text{out} & \text{for } u < v \\
    \text{Choose}_v &\quad c_v \\
    \text{Accept}_v &\quad r_v^a = c_v \\
    \text{Finish}_v &\quad d^a = r_v
\end{align*}
\]

Each value is \textit{nil} or = the previous one

Client

\[
\text{INPUT } x \quad x \in \text{input}
\]

Processes

\[
\begin{align*}
    a &\quad a \\
    a &\quad a \\
    a &\quad a \\
    a &\quad a \\
    a &\quad a \\
    a &\quad a \\
\end{align*}
\]

Actions

\[
\begin{align*}
    \text{Start;} \\
    \text{Close}^a \\
    \text{Anchor} \\
    \text{Input;} \\
    \text{Choose} \\
    \text{Accept}^a \\
    \text{Finish}^a ; \\
    \text{STEP}^a
\end{align*}
\]

Transmit

\[
\begin{align*}
    r^a &\quad \text{INPUT } c_v &\quad r_v^a &\quad \text{OUTPUT }
\end{align*}
\]

\[\text{view change} \quad \text{normal operation} \]
## Example

<table>
<thead>
<tr>
<th></th>
<th>$c_v$</th>
<th>$r_v^a$</th>
<th>$r_v^b$</th>
<th>$r_v^c$</th>
<th></th>
<th>$c_v$</th>
<th>$r_v^a$</th>
<th>$r_v^b$</th>
<th>$r_v^c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>View 1</td>
<td>7</td>
<td>7</td>
<td>out</td>
<td>out</td>
<td>View 2</td>
<td>8</td>
<td>8</td>
<td>out</td>
<td>out</td>
</tr>
<tr>
<td>View 2</td>
<td>8</td>
<td>8</td>
<td>out</td>
<td>out</td>
<td>View 3</td>
<td>9</td>
<td>9</td>
<td>out</td>
<td>9</td>
</tr>
<tr>
<td>View 3</td>
<td>9</td>
<td>out</td>
<td>out</td>
<td>9</td>
<td></td>
<td>9</td>
<td>out</td>
<td>out</td>
<td>9</td>
</tr>
</tbody>
</table>

`input \cap anchor_4 \supseteq \{7, 8, 9\}` seeing $a, b, c$  
`input \cap anchor_4 \supseteq \{8\}` seeing $a, b$  
`input \cap anchor_4 \supseteq \{9\}` seeing $a, c$ or $b, c$

Two runs of AP with  
agents $a, b, c$,  
two agents in a quorum,  
`input = \{7, 8, 9\}`
Anchoring

invariant \( r_v = x \land r_u = x' \Rightarrow x = x' \)

\[
\begin{align*}
V x', u \mid r_v = x & \land \overline{r_u = x'} \Rightarrow x = x' \\
\Rightarrow r_v = x & \Rightarrow (\forall u < v \mid x' \land x) \land \sim Q_{dec}[r_u = x'] \\
\Leftarrow r_v = x & \Rightarrow (\forall u < v \mid c_u = x \lor Q_{out}[r_u = x'])
\end{align*}
\]

\[
\text{sfunc } anchor_v = \{ x \mid (\forall u < v \mid c_u = x \lor Q_{out}[r_u = x']) \}
\]

\[
\begin{align*}
= \{ x \mid (\forall w \mid v_0 = w < u \Rightarrow c_w = x \lor Q_{out}[r_w = x']) \} & = anchor_u \\
\cap \{ x \mid c_u = x \lor Q_{out}[r_u = x'] \} & = X \text{ if } out_{u,v} \\
\cap \{ x \mid (\forall w \mid u < w < v \Rightarrow c_w = x \lor Q_{out}[r_w = x']) \} & \text{ since } c_u \in anchor_u \\
= \{ x \mid c_u = x \} \cup (anchor_u \cap \{ x \mid Q_{out}[r_u = x'] \}) & \text{ if } out_{u,v}
\end{align*}
\]

\\[
\begin{align*}
\begin{align*}
\begin{align*}
\begin{align*}
\text{if } out_{u,v} \land r_u^a = x & \text{ then } \{ x \} \text{ elseif } out_{v_0,v} \text{ then } X \text{ else } \{ \}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

where \( out_{u,v} = (\forall w \mid u < w < v \Rightarrow r_w = out) \)
AP: Algorithm

Start_v  u<v too slow  ?  active_v := true
Close_v^a  active_v
?  for all u < v do  post u<v
  if  r_u^a = nil  then  r_u^a := out

anchor_v = \{ x | c_u = x \} \cup (anchor_u \cap \{ x | Q_{out}[r_u^* \in \{x, out\}]\}) if out_{u,v}

Anchor_v  anchor_v ? {}  ?  no state change
Choose_v  c_v = nil  \land x \in input \cap anchor_v  ?  c_v := x
Accept_v^a  r_v^a = nil  \land c_v ? nil  ?  r_v^a := c_v; Close_v^a
Finish_v^a  r_v \in X  ?  d_v^a := r_v

to later views

r_u^a=nil  \Rightarrow  Close_v  x \in anchor_v  Choose_v  c_v  Accept_v  r_v=c_v  Finish_v  d_v^a=r_v

r_u^a:=out  for u < v
AP: Liveness

Choose must see an element of \( \text{input} \cap \text{anchor}_v \).

Recall \( \text{anchor}_v \)

\[
\text{anchor}_v = \{x | c_u = x\} \cup (\text{anchor}_u \cap \{x | Q_{out}[r_u^* \in \{x, \text{out}\}]\})
\]

\[
\supseteq \text{if } \text{out}_{u,v} \land r_u^a = x \text{ then } \{x\} \text{ elseif } \text{out}_{v_0,v} \text{ then } X \text{ else } \{\}
\]

After \( \text{Close}_v^a \), an OK agent \( a \) has \( r_u^a \neq \text{nil} \) for all \( u < v \).

So if \( Q_{out} \) is live, we see either \( u < v \) is out, or \( r_u^a = x \) for some OK \( a \).

But \( r_u^a = c_u \in \text{input} \cap \text{anchor}_u \)

If we know \( a \) is OK, then \( r_u^a \) is what we want

With faults (in BP), we might not know. But if \( \text{anchor}_u \) is visible, that is enough.
Optimizations

Fixed-size agent state:

\[
\begin{align*}
  r_w^a &= \quad \text{don’t know} \quad x_{last}^a \quad \text{out} \quad \text{nil} \\
  \text{view} &= v_0 \\
  \text{view} &= v_{X_{last}}^a \\
  \text{view} &= v_{last}^a
\end{align*}
\]

Successive steps:

Because \textit{anchor}_v doesn’t depend on \textit{input}, can compute it for lots of steps at once.

This is called a \textit{view change}

One view change is enough for any number of steps

Can batch steps with one Paxos/batch.

Can run steps in parallel, subject to external consistency.
Disk Paxos—DP

The goal—Replace the conditional writes in $Close$ and $Accept$ with simple writes.

\[
Accept_v^a \quad r_v^a = nil \land c_v \ ? nil \quad ? \quad r_v^a := c_v; \ Close_v^a
\]

The idea—Replace $r_v^a$ with $rx_v^a$ and $ro_v^a$.

\[
Accept_v^a \quad c_v \ ? nil \quad ? \quad rx_v^a := c_v; \ Close_v^a
\]

\[
Close_v^a \quad active_v \quad ? \quad \text{for all } u < v \text{ do } ro_u^a := \text{out}
\]

Proof: Keep $r_v^a$ as a history variable. Abstract it to AP’s $\overline{r}_v^a$.
This invariant makes it work (sometimes with an extra view).

\[
rx_v^a = \land \quad ro_v^a = \quad \Rightarrow \quad r_v^a
\]

\[
nil = \land \quad nil = \quad = nil
\]

\[
nil = \land \quad out = \quad = out
\]

\[
x = \land \quad nil = \quad = x
\]

\[
x = \land \quad out = \quad ? \quad nil
\]
Communication

A process has knowledge $T$ of stable non-local facts

$$g@m = (T^m \Rightarrow g)$$

We transmit these facts (note that transmitter $k$ may be failed):

$Transmit^{k,m}(g) \quad g@k \land OK^m \land T^m := T^m \land (g@k \lor F^k) \land post (g@k \lor F^k)@m$

A faulty $k$ can transmit anything:

$Transmit^{F^k,m}(g) \quad F^k \land OK^m \land T^m := T^m \land (g@k \lor F^k) \land post (g@k \lor F^k)@m$

A fact known to a $Q_{-F}^+$ quorum is henceforth known to a $Q_{-F}$ quorum of OK agents, and therefore eventually known to everyone.

$Broadcast^{m}(g) \quad Q_{-F}^+ #g \land OK^m \land T^m := T^m \land g \land post g@m$

Implement $Transmit^{k,m}$ by sending messages. It’s fair if $k$ is OK. This works because the facts are stable.
Classic Paxos—CP

The goal—Tolerate stopped processes

The idea—Agents are the same as in AP. Use a primary process to:

- Implement *Choose*
- Compute an estimate $re_v$ of $r_v$
- Relay facts among the agents
- Do all the scheduling.

So the primary sends $active_v$ to agents to enable $Close_v$, collects $r^a$, computes $anchor$, gets inputs, does *Choose*, sends $c^p$ to agents, collects $r^a$ again to compute $re_v$, and broadcasts $d$.

$$Choose^p \quad active^p \land c^p = nil \quad ? c^p := x$$
$$\land x \in input^p \cap anchor^p$$

Must have only one $c^p$ per view. Get this with
- At most one primary per view
- Primary chooses at most once per view
AP and CP

AP

Processes

Actions

Transmit

CP

Processes

Actions

Transmit Messages

Primary:

Butler Lampson

ABCDs of Paxos: PODC 2001

21
Byzantine Paxos—BP

The goal—Tolerate faulty processes

The idea—To get one $c_v$, a self-exclusive quorum $Q_{ch}$ must choose it
Still have a primary to propose $c_v$; an OK agent only chooses this
A faulty primary can stop its view from deciding

Every agent needs an estimate $ce_v^a$ of $c_v$ and an estimate $re_v^a$ of $r_v$

Invariant: The estimates either are nil or equal the true values.

Every agent also needs its own input$^a$

abstract $c_v = \text{if } Q_{ch}[c_v^* = x] \text{ then } x \text{ else nil}$
sfunc $ce_v^a = \text{if } (Q_{ch}[c_v^* = x])@a \text{ then } x \text{ else nil}$

$anchor_v^a = anchor_u \cap \{x \mid Q_{out}[r_u^* \in \{x, out\}]@a\}$ \quad if $out_{u,v}^a$

$anchor_v^p = \{x \mid Q_{\sim F}^+[x \in anchor_v^*]@p\}$
CP and BP

CP

Processes

Actions

Transmit

Messages

BP

Processes

Actions

Transmit

Messages
Liveness of BP

Choose must see an element of \( \text{input} \cap \text{anchor}_v \).

Recall \( \text{anchor}_v \supseteq \text{anchor}_u \cap \{ x \mid Q_{\text{out}}[r_u^* \in \{ x, \text{out} \}] \} \)

After \( \text{Close}_v^a \), an OK agent \( a \) has \( r_u^a \neq \text{nil} \) for all \( u < v \).

So if \( Q_{\text{out}} \) is live, we see either \( u < v \) is out, or \( r_u^a = x \) for some OK \( a \).

But \( r_u^a = c_u \in \text{input} \cap \text{anchor}_u \)

Unfortunately, we don’t know whether \( a \) is OK.

But we do have \( Q_{\text{ch}}[c_u^* = x] \), hence \( Q_{\text{ch}}[(x \in \text{anchor}_u)@a] \)

So if \( Q_{\text{ch}} \) is live, \( x \in \text{anchor}_u \) is broadcast, which is enough.

So either we eventually see all previous views out, or we see \( x \in \text{anchor}_u \) and all views between \( u \) and \( v \) out.

A faulty client can wreck a view by not sending input to all agents.
Conclusion

Paxos is a practical protocol for fault-tolerant asynchronous consensus. Paxos is efficient in replicated state machines, which are the best mechanism for most fault-tolerant systems.

Paxos works in a sequence of views,

    Each view chooses a value and then seeks a decision quorum. A later view chooses any possible earlier decision

Abstract Paxos chooses a consensus value non-locally, and then decides by local actions of the agents.

    The agents are read-modify-write memories. Disk Paxos generalizes this to read-write memories.

Classic Paxos uses a primary process to choose.

Byzantine Paxos uses a primary to propose, a quorum to choose.