The ABCDs of Paxos

Consensus: a set of processes *decide* on an input value

Main application: Replicated state machines

Paxos asynchronous consensus algorithm

AP	Abstract Paxos:	generic, non-local version
CP	Classic Paxos:	stopping failures, compare-and-swap
	1989: Lamport, Liskov and Oki	
DP	Disk Paxos:	stopping failures, read-write
	1999: Gafni and Lamport	
BP	Byzantine Paxos:	arbitrary failures
	1999: Castro and Liskov	

The paper and slides are 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 applications:

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

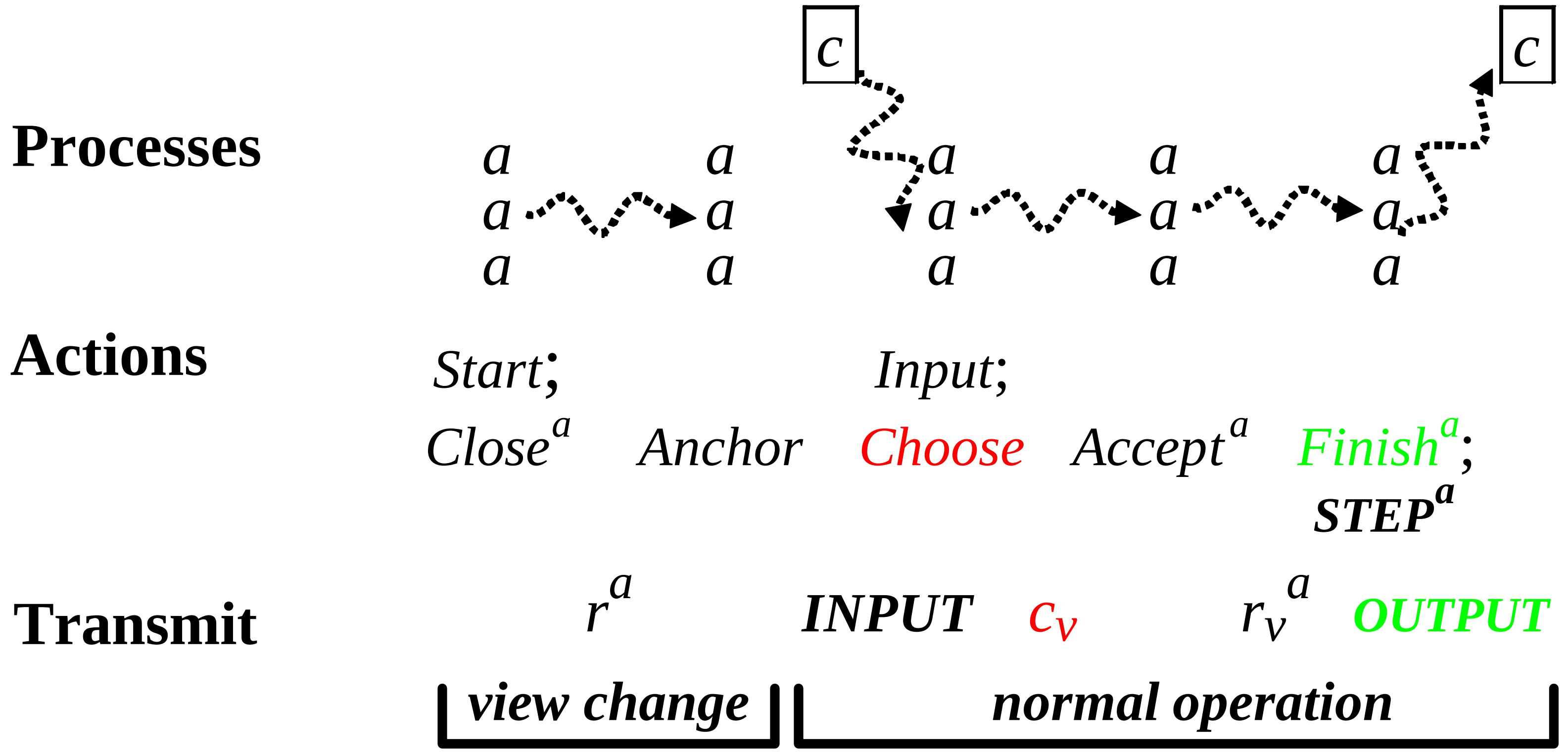
The Idea of Paxos

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

Each view v *chooses* an *anchored* value c_v, equal to 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 if there was a decision



Design Methodology

- Communicate only *stable* predicates: once true always true
- Structure the 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:

Name	**Guard**	**State change**
Close$_v$	$c_v = nil \wedge x \in anchor_v$	$\rightarrow c_v := x$

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. State freezes after failure.

Limits on failure:

Z_F = set of sets of processes that can all be faulty
Z_S = set of sets of processes that can all be stopped
Z_{FS} = set of sets of processes that can all be failed

Examples:

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

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

Intel-Microsoft: $n_I + n_M$ processes, $Z_F=$any subset of one side

Quorums and Predicates

Quorum set Q: set of sets of processes; q in $\Rightarrow$ any superset in.

State predicate g. Predicate on processes G, so G^m is a predicate.

A *stable* predicate once true remains true.

$Q\#G$: A predicate G appears to hold in quorum Q, $\{m \mid G^m \vee F^m\} \in Q$

Shorthand: $Q[r_v{}^*=x]$ for $Q\#(\lambda\, m \mid r_v{}^m = x)$.

A *good* quorum is not all faulty: $Q_{\sim F} = \{q \mid q \not\subseteq Z_F\}$

Q_1 and Q_2 *exclusive*: Q_1 quorum for $G \Rightarrow$ no Q_2 quorum for its negation.

Means $q_1 \cap q_2 \in Q_{\sim F}$ for any q_1 and q_2. Example: size > $(n + f)/2$

Lift local $r_v{}^a=x \Rightarrow \sim(r_v{}^a=out)$ to global $Q_1[r_v{}^*=x] \Rightarrow \sim Q_2[r_v{}^*=out]$

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

A *live* quorum has $Q^+ \neq \{\}$

Specification for Consensus

type	X	$= \ldots$	values to decide on
var	d	$: (X \cup \{nil\}) \;:= nil$	Decision
	$input$	$:$ **set** $X := \{\}$	

Name	**Guard**	**State change**
***Input*(*x*)**		$input := input \cup \{x\}$
Decision*: *X	$d \neq nil$	$\rightarrow$**ret** d
Decide	$d = nil \wedge x \in input$	$\rightarrow d := x$

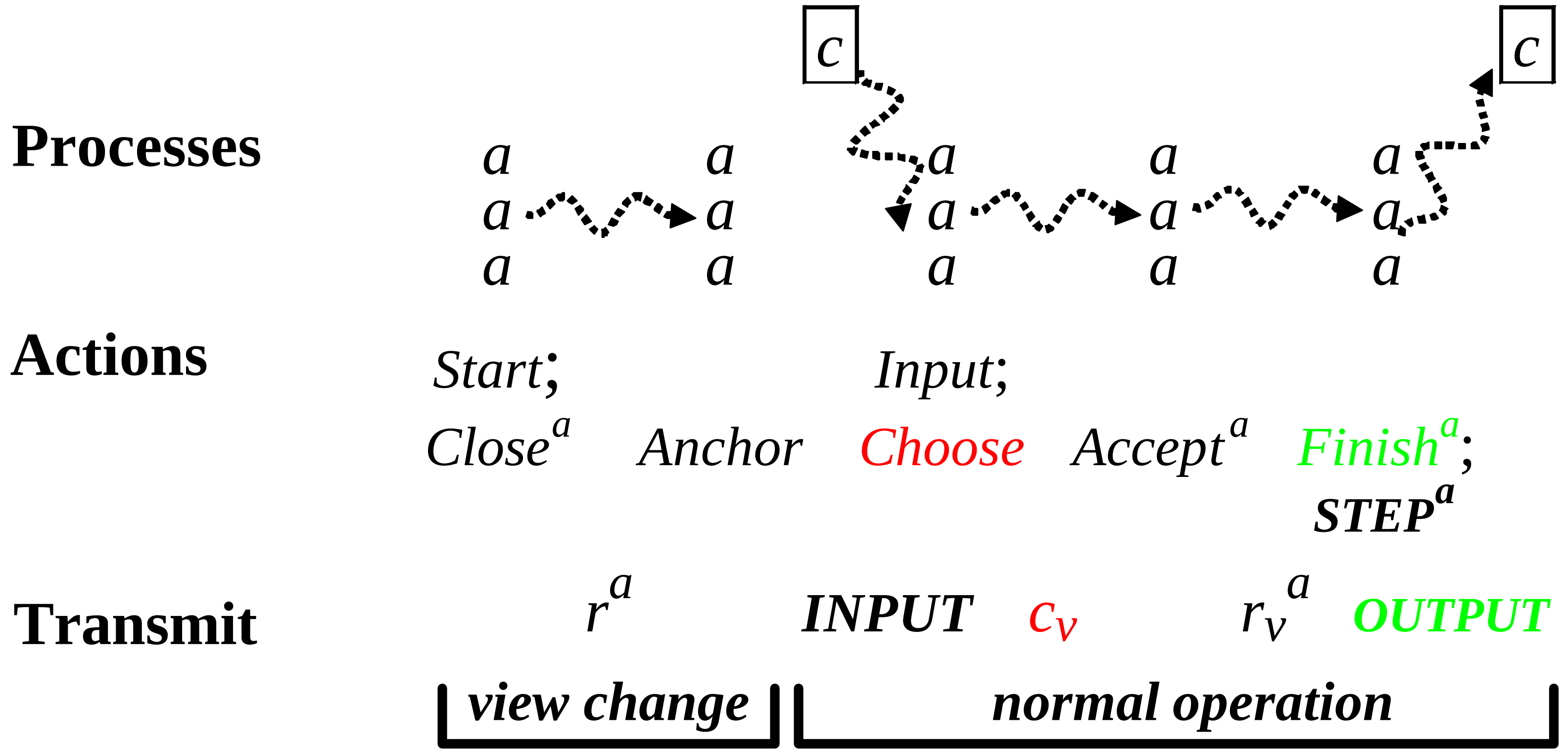
The Idea of Paxos

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.

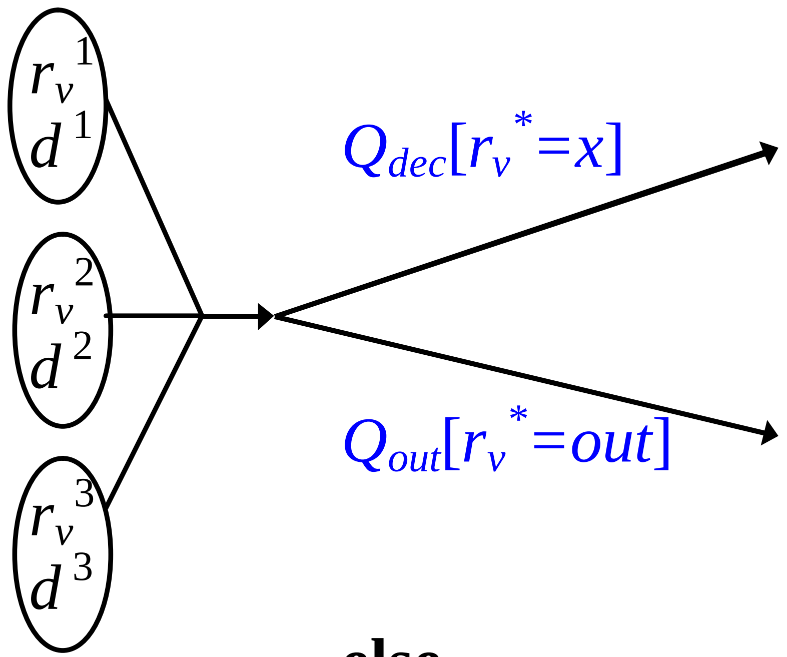
If a quorum *accepts* the choice, decision!

Decision is irrevocable, may be invisible, but is any later view's choice.
Choice is changeable, must be visible to *Anchor* if there was a decision.



Abstract Paxos—AP: State

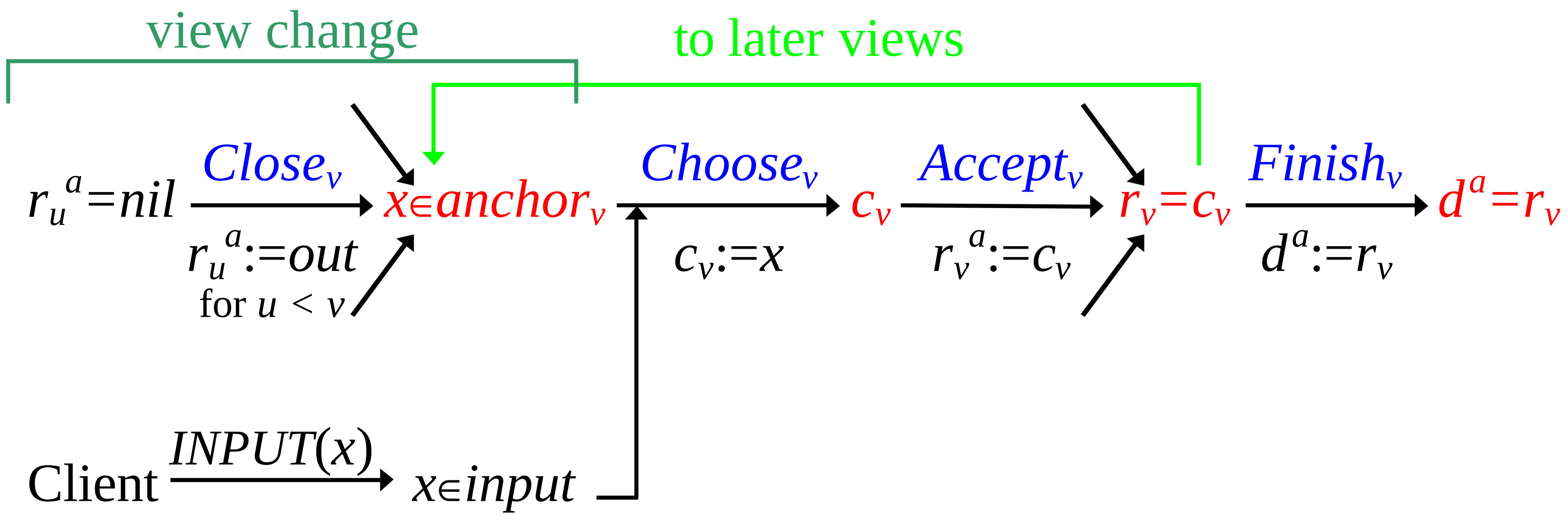
Non-local	Agents' State		State functions		View is
			r_v	d	
c_v	1: r_v^1 d^1	$Q_{dec}[r_v^*=x]$	x	x	decided
$input$	2: r_v^2 d^2				
		$Q_{out}[r_v^*=out]$	out	nil	out
$active_v$	3: r_v^3 d^3				
		else	nil	nil	open

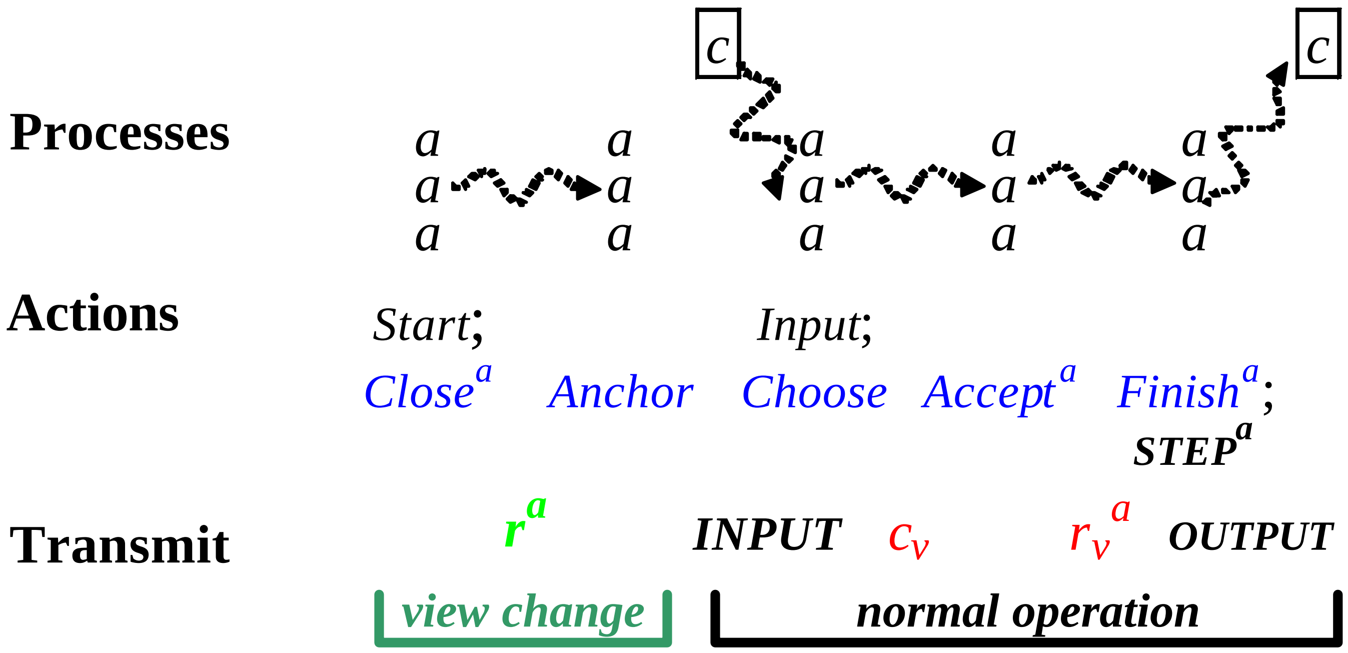


Q_{dec} and Q_{out} exclusive

$var = const$ is stable for all these except *input*, and $x \in input$ is stable.

AP: Data Flow


view change
to later views
$r_u{}^a$=nil
$Close_v$
$r_u{}^a$:=out
for $u < v$
$x \in anchor_v$
$Choose_v$
c_v:=x
c_v
$Accept_v$
$r_v{}^a$:=c_v
r_v=c_v
$Finish_v$
d^a:=r_v
d^a=r_v
Client
$INPUT(x)$
$x \in input$


Processes
c
c
a a a
a a a
a a a
a a a
a a a
Actions
Start;
Input;
$Close^a$
Anchor
Choose
$Accept^a$
$Finish^a$;
$STEP^a$
Transmit
r^a
INPUT
c_v
$r_v{}^a$
OUTPUT
view change
normal operation

Example

	c_v	r_v^a	r_v^b	r_v^c	c_v	r_v^a	r_v^b	r_v^c
View 1	7	7	*out*	*out*	8	8	*out*	*out*
View 2	8	*out*	8	*out*	9	9	*out*	9
View 3	9	*out*	*out*	9	9	*out*	*out*	9
input $\cap$ *anchor*$_4$	= {7, 8, 9} seeing *a*, *b*, *c* $\supseteq$ {8} seeing *a*, *b* $\supseteq$ {9} seeing *a*, *c* or *b*, *c*				{9} no matter what quorum we see			

Two runs of AP with

agents *a*, *b*, *c*,

two agents in a quorum,

input = {7, 8, 9}

Anchoring

invariant $r_v = x \wedge r_u = x' \Rightarrow x = x'$ — all results agree

$= \forall\, x', u \mid r_v = x \wedge \underbrace{r_u = x'} \Rightarrow x = x'$ — assume $u<v$

$= r_v = x \Rightarrow (\forall\, u < v, x' \neq x \mid \sim \boxed{Q_{dec}[r_u^{*}=x']})$

$\Leftarrow r_v = x \Rightarrow (\forall\, u < v \mid Q_{out}[r_u^{*} \in \{x, out\}])$

$r_u^{a} \in \{x, out\}$
$\Rightarrow \sim(r_u^{a} = x')$

sfunc $anchor_v$

$= \{x \mid (\forall\, u < v \mid Q_{out}[r_u^{*} \in \{x, out\}])\}$

$= \{x \mid (\forall\, w \mid v_0 = w < u \Rightarrow Q_{out}[r_w^{*} \in \{x, out\}])\}$ $= anchor_u$

$\cap \{x \mid Q_{out}[r_u^{*} \in \{x, out\}]\}$

$\cap \{x \mid (\forall\, w \mid u < w < v \Rightarrow Q_{out}[r_w^{*} \in \{x, out\}])\}$ $= X$ **if** $out_{u,v}$

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

$\supseteq$ **if** $out_{v0,v}$ **then** X **elseif** $out_{u,v} \wedge r_u^{a} = x$ **then** $\{x\}$ **else** $\{\}$

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

AP: Algorithm

Start$_v$	$u<v$ *too slow*	$\rightarrow active_v := true$	
Close$_v^a$	$active_v$	$\rightarrow$ **for all** $u < v$ **do** **if** $r_u^a = nil$ **then** $r_u^a := out$	**post** $u<v \Rightarrow r_u^a \neq nil$

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

Anchor$_v$	$anchor_v \neq \{\}$	$\rightarrow$no state change
Choose$_v$	$c_v^a = nil$ $\wedge\ x \in input \cap anchor_v$	$\rightarrow c_v := x$
Accept$_v^a$	$r_v^a = nil$ $\wedge\ c_v \neq nil$	$\rightarrow r_v^a := c_v$; $Close_v^a$
Finish$_v^a$	$r_v \in X$	$\rightarrow d^a := r_v$

$$\begin{tikzcd}
r_u^a=nil \arrow[r, "Close_v", "r_u^a:=out \text{ for } u<v"'] & x\in anchor_v \arrow[r, "Choose_v", "c_v:=x"'] & c_v \arrow[r, "Accept_v", "r_v^a:=c_v"'] & r_v=c_v \arrow[r, "Finish_v", "d^a:=r_v"'] \arrow[ll, bend right, "\text{to later views}"'] & d^a=r_v
\end{tikzcd}$$

AP: Liveness

$Choose_v$ must see an element of $input \cap anchor_v$.

Recall $anchor_v$

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

$\supseteq \quad$ **if** $out_{v0,v}$ **then** X **elseif** $out_{u,v} \wedge r_u^a = x$ **then** $\{x\}$ **else** $\{\}$

After $Close_v^a$, an OK agent a has $r_u^a \neq nil$ 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 $r_u^a = c_u \in input \cap anchor_u$

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

With faults (in BP), we might not know.
But if $anchor_u$ is visible, that is enough.

Still not live if new views start too fast.

Optimizations

Fixed-size agent state:

$$\begin{array}{llcccc} r_w^a = & \longleftarrow & \text{don't know} \longrightarrow & x_{last}^a & \longleftarrow \textit{out} \longrightarrow & \longleftarrow \textit{nil} \longrightarrow \\ & | & & | & & | \\ \text{view} & v_0 & & vX_{last}^a & & v_{last}^a \end{array}$$

Successive steps:

Because *anchor*$_v$ doesn't depend on *input*, can compute it for lots of steps at once.

This is called a *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.

$\boldsymbol{Accept_v^a} \quad r_v^a = nil \wedge c_v \neq nil \quad \rightarrow r_v^a := c_v;\ Close_v^a$

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

$\boldsymbol{Accept_v^a} \quad c_v \neq nil \quad \rightarrow rx_v^a := c_v;\ Close_v^a$

$\boldsymbol{Close_v^a} \quad active_v \quad \rightarrow \textbf{for all } u < v \textbf{ do } ro_u^a := out$

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

$rx_v^a =$	$\wedge$	$ro_v^a =$	$\Rightarrow$	r_v^a
nil		nil		$= nil$
nil		out		$= out$
x		nil		$= x$
x		out		$\neq nil$

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*)** $\quad g@k \vee F^k \rightarrow T^m := T^m \wedge (g@k \vee F^k)$ **post** $(g@k \vee F^k)@m$

A faulty k can transmit anything:

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

***Broadcast*^{m}(*g*)** $\quad Q_{\sim F}{}^{+}[g@*] \wedge OK^m \rightarrow T^m := T^m \wedge g$ $\quad$ **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.

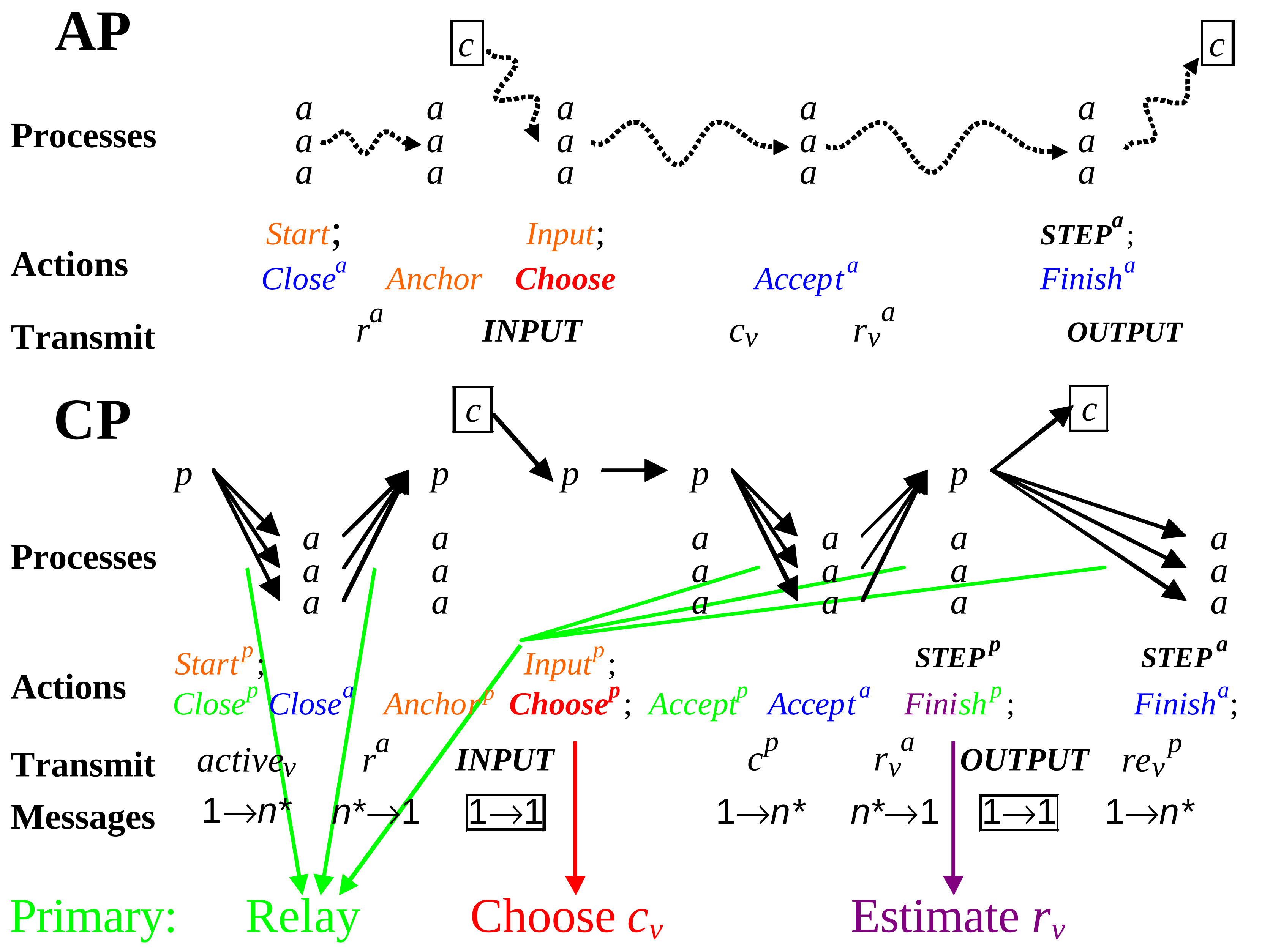
So the primary sends $active_v$ to agents to enable $Close_v$, collects r^a, computes *anchor*, gets inputs, does *Choose*, sends c^p to agents, collects r^a again to compute re_v, and sends d.

$$\textbf{\textit{Choose}}^p \qquad active^p \wedge c^p = nil \qquad \rightarrow c^p := x$$
$$\wedge\, x \in input^p \cap anchor^p$$

Must have only one c^p per view. Get this with

- At most one primary per view, and
- Primary chooses at most once per view

AP and CP


AP
Processes
Actions
Transmit
c
a a a
Start; Close^a
Anchor
Input; Choose
Accept^a
STEP^a; Finish^a
r^a
INPUT
c_v
r_v^a
OUTPUT
CP
p
Start^p; Close^p Close^a
Anchor^p
Input^p; Choose^p;
Accept^p Accept^a
STEP^p Finish^p;
STEP^a Finish^a;
Messages
active_v
1→n*
n*→1
1→1
c^p
1→n*
r_v^a
n*→1
1→1
re_v^p
1→n*
Primary:
Relay
Choose c_v
Estimate r_v

Byzantine Paxos—BP

The goal—Tolerate faulty processes

The idea—To ensure one c_v, a self-exclusive quorum Q_{ch} chooses it

Still have a primary to propose c_v; an OK agent only chooses this

Primary's proposal should be anchored and input at agents
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 $\underline{c}_v$ = **if** $Q_{ch}[c_v^* = x]$ **then** x **else** *nil*

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

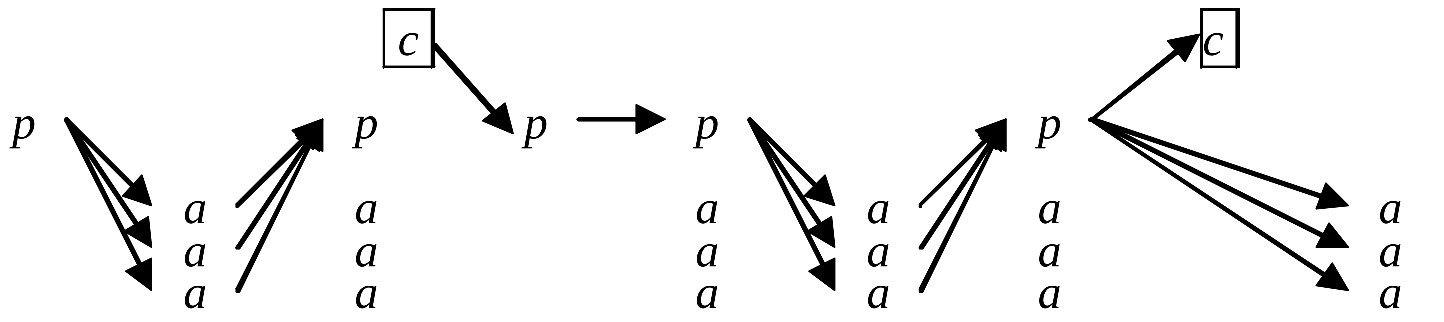
$anchor_v^a = anchor_u \cap \{x \mid Q_{out}[r_u^* \in \{x, out\}]@a\}$ **if** $out_{u,v}^a$

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

CP and BP

CP

Processes



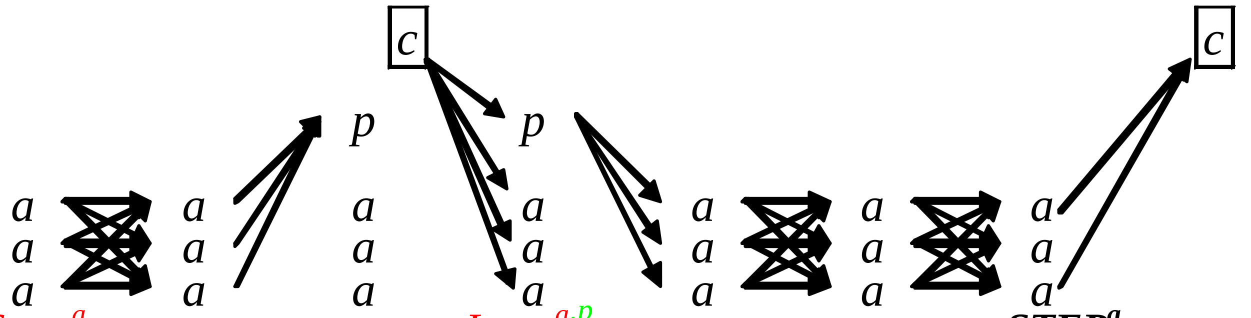
Actions

$Start^p$; $Close^p$ $Close^a$ | $Anchor^p$ | $Input^p$; $Choose^p$; | $Accept^p$ | $Accept^a$ | $STEP^p$ $Finish^p$; | $STEP^a$ $Finish^a$;

Transmit	$active_v$	r^a	***INPUT***	c^p	$r_v{}^a$	***OUTPUT***	$re_v{}^p$
Messages	$1\to n^*$	$n^*\to 1$	$\boxed{1\to 1}$	$1\to n^*$	$n^*\to 1$	$\boxed{1\to 1}$	$1\to n^*$

BP

Processes



Actions

$Start^a$; $Close^a$ | $Anchor^a$ | $Anchor^p$ | $Input^{a,p}$; $Choose^p$ | $Choose^a$ | $Accept^a$ | $STEP^a$ $Finish^a$;

Transmit	r^a, c^a	$anchor_v{}^a$	***INPUT***	$c_v{}^p$	$c_v{}^a$	$r_v{}^a$	***OUTPUT***
Messages	$n\to n$	$n^*\to 1$	$\boxed{1\to n}$	$1\to n^*$	$n^*\to n$	$n\to n$	$\boxed{n_g\to 1}$

Liveness of BP

Choose must see an element of $input \cap anchor_v$.

Recall $anchor_v \supseteq anchor_u \cap \{x \mid Q_{out}[r_u^{*} \in \{x, out\}]\}$

After $Close_v^{a}$, an OK agent a has $r_u^{a} \neq nil$ 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 $r_u^{a} = c_u \in input \cap anchor_u$

Unfortunately, we don't know whether a is OK.

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

So if Q_{ch} is live, $x \in anchor_u$ is broadcast, which is enough.

So either we eventually see all previous views out,
or we see $x \in anchor_u$ 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.