The ABCD's of Paxos

Butler W. Lampson
Microsoft
180 Lake View Ave
Cambridge, MA 02138
1-617-547-9580
blampson@microsoft.com

Abstract

We explain how consensus is used to implement replicated state machines, the general mechanism for fault-tolerance. We describe an abstract version of Lamport's Paxos algorithm for asynchronous consensus. Then we derive the Byzantine, classic, and disk versions of Paxos from the abstract one, show how they are related to each other, and discuss the safety, liveness, and performance of each one.

Categories and Subject Descriptors

D.2.4 [**Software]** Correctness Proofs—abstraction function, invariant, simulation; Fault Tolerance—Byzantine, Paxos, replicated state machine, view change.
[**Theory**]—consensus, liveness, safety.

General Terms

Algorithms, Reliability, Security, Theory

Keywords

Paxos, asynchronous consensus, fault-tolerant, replication, Lamport, Byzantine, state machine

1 Introduction

We give an abstract version AP of Lamport's Paxos algorithm for asynchronous consensus that captures its idea, but is not directly implementable because some of the actions touch non-local state. Then we give three implementations of AP that solve this problem in different ways, together with the abstractions and invariants of their simulation proofs:

> Classic Paxos, CP, from Lamport's original paper [9] and from Liskov and Oki [13], tolerates $n/2$ stopped processes and requires conditional write (compare and swap) operations on persistent state variables.
>
> Disk Paxos, DP, from Gafni and Lamport's recent paper [5], is a generalization of AP and CP that requires only read and write operations on persistent state variables.
>
> Byzantine Paxos, BP, described by Castro and Liskov in [1] and [2], tolerates $n/3$ processes with arbitrary faults. Their papers also describe a replicated state machine implementation, based on BP, that has good performance and the same fault tolerance.

AP, CP, and BP are summarized in the appendix.

I've tried to answer all the questions I had when I read these papers, about how simple the algorithms can be made, the minimum conditions for them to work, and how they are related. The role that General Λαμπσων played in the original Paxos paper makes it especially appropriate for me to write about a Byzantine version.

I don't know whether a practical algorithm could be developed in this top-down fashion. Certainly the three that we give were not invented in this way, but our exposition does clarify the relationships among them and perhaps will suggest other variations.[1]

The top-down development often works by introducing new variables that are related to the abstract variables by an invariant, and modifying the actions so that they depend only on the new variables and not on the abstract ones. The abstract variables thus become history variables in the proof.

1.1 Replicated state machines

The main application for fault-tolerant consensus is replicated state machines. This is the fundamental technique for general fault-tolerance, first described by Lamport [7]. It goes like this:

> Cast your problem as a deterministic state machine that takes input requests for state transitions, called *steps*, from the client, performs the steps, and returns the output to the client. It's hard to find a problem that can't be done this way.
>
> Implement the state machine and make n copies of it, often called 'replicas'.
>
> Using consensus, feed all the replicas the same input sequence. Then they all generate the same output sequence.

If a replica fails, it can recover by starting in the initial state and replaying all the inputs. As with transaction systems [6], it can speed up this complete replay by starting with a previous state instead of at the beginning.

The steps of the state machine can be arbitrarily complicated as long as they are deterministic, atomic, and strictly local to one replica. To make a big step atomic, use transactions [6]. Of course a replica can involve more than one physical machine; in fact, like any good idea in computer science, the entire method can be applied recursively.

Even reading the state must be done with a step, unless the client is willing to accept output based on an arbitrarily old state. If a read also returns the sequence number of the last step that affected it, the client can pay for better read performance with complexity by doing an occasional step to learn the current step, and then

[1] There is a similar treatment of reliable messages in [10] and [11].

accepting read outputs that are not too far out of date. With a sloppy notion of real time the state machine can give the client a bound on number of seconds a read might be out of date.

Since fault-tolerant consensus makes all the inputs persistent, exactly-once semantics needs no extra persistent writes. The state machine does have to check that an input hasn't been accepted already, which it can do by caching the sequence number or timestamp of the most recent input from each client.

The most common application is to data storage systems such as a file system [13]. The method is much more general, however. For instance, state machine actions can be used to change the sets of processes that form the various quorums on which consensus depends, so that no special algorithms are needed to deal with processes that arrive and depart in an orderly way.

Many applications combine a replicated state machine with *leases*, which are locks on portions of the state. A lease differs from a lock because it times out, so the system doesn't block indefinitely if the leaseholder fails. To keep the lock the holder must renew the lease. There is an obvious tradeoff between the cost of frequent renewals and the cost of waiting for the lease to expire. A client (or a subordinate state machine) with a lease can do arbitrary reads and writes of the leased state without taking any steps of the main state machine, except for a single step that combines all the writes. The most important use of leases is to allow holders to cache part of the state. Like locks, leases can be hierarchical and can have different modes such as shared and exclusive.

Consensus is also useful for group membership and transaction commit.

1.2 The idea of Paxos

A consensus algorithm decides on one from a set of *input* values (such as the state machine inputs). It uses a set of processes, called *agents* in this paper. The simplest form of consensus decides when a majority of agents choose the same value. This is not very fault-tolerant for two reasons: there may never be a majority, and even when there is, it may remain permanently invisible if some of its agents stop. Since we can't distinguish a stopped agent from a slow one, we can't tell whether the invisible majority will reappear, so we can't ignore it.

To avoid these problems, Paxos uses a sequence of *views*.[2] A majority in any view decides (or more generally, a decision quorum; see section 4.2), but if a view doesn't work out, a later view can supersede it. This makes the algorithm fault-tolerant, but introduces a new problem: decisions in all views must agree.

The key idea of Paxos is that a later view v need not know that an earlier view decided in order to agree with it. Instead, it's enough to classify each earlier view u into one of two buckets: either it can *never* decide, in which case we say that it's *out*, or it has made a *choice* and it must decide for that choice if it decides at all. In the latter case v just needs to know u's choice.

Thus a view chooses and then decides. The choice can be superseded, but the decision cannot. On the other hand, the choice must be visible unless the view is visibly out, but the decision need not be visible because we can run another view to get a visible decision. This separation between decision and visibility is the heart of the algorithm.

A decision will be unique as long as every later choice agrees with it. We ensure this by *anchoring* the choice: if all previous views are out, v can choose any input value; if not, it can take the choice of the latest previous view that isn't known to be out. By induction, this ensures that v will agree with any previous decision. To keep from blocking the algorithm, we must be able to make each previous view visibly out unless it has a visible choice. See section 4.3 for a picture of the anchor-choose-decide sequence.

1.3 Design methodology

Our description of the algorithms is based on a methodology for designing fault-tolerant systems. There are five principles:

> Use only *stable* predicates to communicate state among processes. A predicate is stable if once true, it never becomes false. Hence information you have about non-local state can never become false. This makes it much easier to reason about the effects of failures and other concurrent actions. We say that a variable is stable if its non-nil value doesn't change: y is stable if ($y = constant \wedge y \neq nil$) is stable.
>
> Structure the program as a set of separate atomic actions. This simplifies reasoning about failures. If sequencing is necessary, code it into the state; the actions of the primary in CP below are an example of this. This avoids having a PC and invariants that connect it to the state. State should be either persistent or local to a sequence of actions that can be abandoned.
>
> Make the actions as non-deterministic as possible, with the weakest possible guards. This allows more implementations, and also makes it clearer why the algorithm works.
>
> Separate safety, liveness, and performance. Start with an algorithm that satisfies a safety property expressed as a state machine specification. Then strengthen the guards on some of the actions to ensure liveness or to schedule the actions; this reduces the number of possible state transitions and therefore cannot affect safety.
>
> Use an abstraction function and a simulation proof to show that an algorithm satisfies its safety specification.[3] Put all the relationships between actions into invariants; it should never be necessary to do an explicit induction on the number of actions. Liveness proofs are more ad hoc.

1.4 Related work

Classic Paxos was invented independently by Lamport [9] and by Liskov and Oki [13]. This version of Paxos tolerates only stopping faults.

Lamport's work was neglected because of the complicated fiction he used to describe it. He calls an agent a 'priest' and a view a 'ballot', and describes the application to replicated state machines in detail. A recent extension called Disk Paxos allows read-write memory such as a disk to be used as an agent [5]. My previous exposition of Classic Paxos and state machines calls a view a 'round' and a primary a 'leader' [12].

Liskov and Oki's work is embedded in an algorithm for data replication, so the fact that they describe a consensus algorithm was overlooked. Not surprisingly, they call an agent a 'replica'; they also use the terms 'primary' and 'backup'.

Castro and Liskov introduced Byzantine Paxos, which tolerates arbitrary faults [1][2]. They present it in the same way as Liskov and Oki.

[2] Views are 'ballots' in Lamport's original paper, and 'rounds' in other papers. 'View' suggests a view of the state or a view of the membership of a group, although these are only applications of consensus.

[3] See [8] and [12] for informal explanations of simulation proofs, and [14] for a thorough account.

There is an extensive literature on consensus problems, thoroughly surveyed by Lynch [14]. Malkhi and Reiter treat Byzantine quorums [15].

1.5 Organization

Section 2 gives the background: notation, failure model, and quorums. Section 3 is the specification for consensus, followed by AP in section 4 and its DP generalization in section 5. Section 6 explains how we abstract communication, and sections 7 and 8 use this abstraction for CP and BP. Section 9 concludes. An appendix summarizes the notation and main actions of AP, CP, and BP.

2 Background

2.1 Notation

To avoid a clutter of parentheses, we usually write subscripts and superscripts for function arguments, so $g(v, a)$ becomes g_v^a. We use subscripts for views and superscripts for processes. Other subscripts are part of the name, as in v_0 or Q_{out}.

We use lower-case letters for variables and upper-case letters for sets and predicates (except that we use q and z for sets of processes, so that Q and Z can denote sets of sets). We treat a type as a set, but also use it to overload functions and operators. We use names starting with t for variables of type T.

We define no-argument functions on the state called 'state functions' and use them like variables, except that we don't assign to them. Rather than recompute such an r each time it's used, a real program might have a variable r' and maintain the invariant $r = r'$.

We use g for a predicate on the state, and G for a process predicate, a function from a process to a predicate. F and S stand for specific process predicates described below. We lift logical operators to process predicates, writing $G_1 \wedge G_2$ for $(\lambda\, m \mid G_1^m \wedge G_2^m)$.

We write $\{x \in X \mid G(x)\}$ in the usual way to describe a set: the subset of X whose elements satisfy G. This extends to $\{x, y \mid G(x, y) \mid f(x, y)\}$ for $\{z \mid (\exists x, y \mid G(x, y) \wedge z = f(x, y)\}$.

We describe actions with the following schema:

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

The name of the action is in bold. The guard is a predicate that must be true for the action to happen. The last column describes how the state changes; read "guard ⟶ state change" as "if guard then state change". A free variable in an action can take on any value of its type. An action takes place atomically.

A variable declaration

var y : $Y := nil$

gives the variable's name y, type Y, and initial value nil.

When we derive an action or formula from a previous version, boxes highlight the parts that change, except for process superscripts. Shading highlights non-local information. Underlines mark the abstract variables in a simulation proof of refinement.

The appendix has a summary of the notation as well as the variables and main actions of the various algorithms.

2.2 Failure model

We have a set M (for Machine) of processes, and write m or k for a process, and later a and p for agent and primary processes.

We admit faulty processes that can send any messages, and stopped processes that do nothing. A failed process is faulty or stopped; a process that isn't failed is OK. Our model is asynchronous, which means that you can't tell a stopped process from a slow one (after all, both begin with 's'). A process that crashes and restarts without losing its state is not stopped, but only slow. A process can also have a crash or reset action that does lose some state; this is also not a failure.

We define predicates on processes: F^m is true when m is faulty, S^m when m is stopped. These are stable, since a process that fails stays failed. $OK = \sim(F \vee S)$. When a process fails its state stops changing, since failed processes don't do actions. Thus every action at m has $\wedge\, OK^m$ in its guard, except a send from a faulty process. To reduce clutter we don't write this explicitly.

A faulty process can send arbitrary messages. For reasoning from the contents of messages to be sound, any g inferred from a message from m must therefore be weaker than F^m, that is, equal to $g \vee F^m$. You might think that the state of a faulty process should change arbitrarily, but this is unnecessary. It does all its damage by sending arbitrary messages. Those are its external actions, and they are the same for arbitrary state and for frozen state.

The reason for distinguishing faulty from stopped processes is that if we get the faulty processes wrong, we lose safety: the system does the wrong thing. If we get the stopped processes wrong, we just lose liveness: the system does nothing. In many applications safety is much more important than liveness. This is like the distinction between integrity and availability (or preventing denial of service) in security.

We limit the extent of failures with sets Z_F, the set of all sets of processes that can be faulty simultaneously, Z_S, the same for stopped, and Z_{FS} the same for failed. Clearly $Z_F \subseteq Z_{FS}$ and $Z_S \subseteq Z_{FS}$. The simplest example is bounds f and s on the number of faulty and stopped processes. We define $Z_{\leq i} = \{z \mid |z| \leq i\}$. Then $Z_F = Z_{\leq f}$, any set of size $\leq f$, and $Z_S = Z_{\leq s}$, any set of size $\leq s$. If $f = 0$ there are no faulty processes and only $\{\}$ is in Z_F.

A different example for faults is mutual mistrust. Each process belongs either to Intel or to Microsoft, and both an Intel and a Microsoft process cannot be faulty:

$Z_F = \{z \mid z \subseteq z_{Intel} \vee z \subseteq z_{Microsoft}\}$.

Similarly, for stops we might use geographical separation. All the processes in Boston or in Seattle can stop (perhaps because of an earthquake), but at most one in the other place:

$Z_S = \{z_b \subseteq z_{Boston}, z_s \subseteq z_{Seattle} \mid |z_b| \leq 1 \vee |z_s| \leq 1 \mid z_b \cup z_s\}$

It seems natural to assume that $F \Rightarrow S$, since a faulty process might appear stopped by sending no messages. This implies $Z_F \subseteq Z_S = Z_{FS}$. For the bounded case, it implies $f \leq s$. It's not essential, however, that faulty imply stopped. The important thing about a faulty process is that it can send a false message, which can affect safety, while a stopped process can only affect liveness.

For example, $F \Rightarrow S$ implies that Intel-Microsoft has no live quorums (see below), since all the Intel processes can be faulty, but if they can all be stopped then none are left to form the Intel part of a quorum. We could, however, configure such a system on the assumption that no more than two processes will stop; then any three processes from each side is a live quorum. This makes sense if each side insists that no decision can depend entirely on the other side, but is willing to wait for a decision if the other side is completely stopped.

2.3 Quorums

A quorum set Q is a set of sets of processes. Define $Q\#G = \{m \mid G^m \vee F^m\} \in Q$, that is, $G \vee F$ holds at every process in some quorum in Q. F is there to make the predicate a sound conclusion from a message. We require Q to be monotonic $(q \in Q \wedge q \subseteq q' \Rightarrow q' \in Q)$, so that making G true at more processes

doesn't falsify $Q\#G$. If $G_1 \Rightarrow G_2$ then $Q\#G_1 \Rightarrow Q\#G_2$. It's natural to define $Q\sim_F = \{q \mid q \notin Z_F\}$, and similarly for $Q\sim_S$.

Quorum sets Q and Q' are (mutually) *exclusive* if we can't have both a Q quorum for G and a Q' quorum for its negation: $(\forall G \mid Q\#G \Rightarrow \sim Q'\#\sim G)$. This holds if every Q quorum intersects every Q' quorum in a set of processes that can't all be faulty:

$\forall q \in Q, q' \in Q' \mid q \cap q' \in Q\sim_F$

This is how we lift local exclusion $G_1 \Rightarrow \sim G_2$ to global exclusion $Q\#G_1 \Rightarrow \sim Q'\#G_2$. Exclusion is what we need for safety.

For liveness we need to relate various quorums to the sets of possibly faulty or stopped processes.

> To ensure G holds at some non-faulty process, we need to hear it from a *good* quorum, one that can't all be faulty, that is, one in $Q\sim_F$. If $g = G^m$ is independent of m, then $Q_{\sim F}\#G \Rightarrow g$; this is how we establish g by hearing from some processes.
>
> To ensure that henceforth there's a visible Q quorum satisfying a predicate G, we need a quorum Q^+ satisfying G that still leaves a Q quorum after losing *any* set that can fail:
>
> $Q^+ = \{q' \mid (\forall z \in Z_{FS} \mid q' - z \in Q\}$.
>
> If $Q^+ \neq \{\}$ then Q is *live*: there's always some quorum in Q that isn't failed.

The most popular quorum sets are based only on the size of the quorums: $Q_{\geq i} = \{q \mid |q| \geq i\}$. If there are n processes, then for $Q_{\geq i}$ and $Q_{\geq j}$ to be exclusive, we need $i + j > n + f$. If $Z_F = Z_{\leq f}$ then $Q\sim_F = Q_{\geq f+1}$. If $Z_{FS} = Z_{\leq s}$ then $Q_{\geq i}{}^+ = Q_{\geq s+i}$ and $Q_{\geq i}$ live requires $i \leq n - s$, since $Q_{>n} = \{\}$. So we get $n + f < i + j \leq 2(n - s)$, or $n > f + 2s$. Also $i > n + f - j \geq n + f - (n - s)$, or $i > f + s$. With the minimum $n = f + 2s + 1$, $f + s < i \leq f + s + 1$, so we must have $i = f + s + 1$. If $s = f$, we get $n = 3f + 1$ and $i = 2f + 1$.

With $f = 0$ there are exclusive 'grid' quorum sets: arrange the processes in a rectangular grid and take Q to be the rows and Q' the columns. If Q must exclude itself, take a quorum to be a row and a column, minus the intersection if both have more than two processes. The advantage: a quorum is only $\sqrt{n}$ or $2(\sqrt{n} - 1)$ processes, not $n/2$. This generalizes to $f > 0$ because quorums of i rows and j columns intersect in $2ij$ processes [15].

For the Intel-Microsoft example, an exclusive quorum must be the union of an exclusive quorum on each of the two sides.

3 The specification for consensus

The external actions are *Input*, which provides an input value from the client, and *Decision*, which returns the decision, waiting until there is one.[4] Consensus collects the inputs in the *input* set, and the internal *Decide* action picks one from the set.

type	X	$=\ldots$	values to agree on
var	d	$: (X \cup \{nil\}) := nil$	Decision; *x*/*nil*, not *out*
	$input$	$:$ **set** $X := \{\}$	

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

For replicated state machines, the inputs are requests from the clients. Typically there is more than one at a time; those that don't win are carried over to *input* for the next step.

[4] A different spec would allow it to return *nil* if there's no decision, but then it must be able to return *nil* even if there has already been a decision, since a client may do the *Decision* action at a process that hasn't yet heard about the decision. For this paper it makes no difference.

It's interesting to observe that there is a simpler spec with identical behavior.[5] It has the same d and *Decision*, but drops *input* and *Decide*, doing the work in *Input*.

var	d	$: (X \cup \{nil\}) := nil$	Decision; *x*/*nil*, not *out*

***Input*(*x*)**		**if** $d = nil$ **then** optionally $d := x$
Decision: X	$d \neq nil$	$\rightarrow$**ret** d

A simulation proof that the first spec implements the second, however, requires a prophecy variable or backward simulation.

This spec says nothing about liveness, because there is no live algorithm for asynchronous consensus [4].

4 Abstract Paxos

As we said in section 1.2, the idea of Paxos is to have a sequence of views until one of them forms a quorum that is noticed. So each view has three stages:

> *Choose* an input value that is anchored: guaranteed to be the same as any previous decision.
>
> Try to get a decision quorum of agents to *accept* the value.
>
> If successful, *finish* the algorithm by recording the decision at the agents.

This section describes AP, an abstract version of Paxos. AP can't run on your computers because some of the actions refer to non-local state (marked like this so you can easily see where the implementation must differ). In particular, *Choose* and c_v are completely non-local in AP. In later sections we will see different ways to implement AP with actions that are entirely local. The key problem is implementing *Choose*.

AP has external actions with the same names as the spec, of course. They are almost identical to the actions of the spec.

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

4.1 State variables

type	V	$=$ ...		View; totally ordered
	Y	$= X \cup \{out, nil\}$		
	A	$\subseteq M =$ …		Agent
	Q	$=$ **set** A		Quorum
const	Q_{dec}	:	**set** Q := ...	decision Quorum set
	Q_{out}	:	**set** Q := ...	out Quorum set
	v_0	:	V := ...	smallest V

The views must be totally ordered, with a first view v_0. Q_{dec} and Q_{out} must be exclusive.

var	$r_v{}^a$	:	$Y := nil$, but $r_{v0}{}^a := out$	Result
	d^a	:	$Y := nil$	Decision; *x*/*nil*, not *out*
	c_v	:	$Y := nil$	Choice; *x*/*nil*, not *out*
	$input$	:	**set** $X := \{\}$	
	$active_v$	:	$Bool := false$	

Each agent has a decision d^a, and a result $r_v{}^a$ for each view; we take $r_{v0}{}^a = out$ for every a. AP doesn't say where the other variables live.

4.2 State functions and invariants

We define a state function r_v that is a summary of the $r_v{}^a$: the view's choice if there's a decision quorum for that among the agents, or *out* if there's an out quorum for that, or *nil* otherwise.

[5] I am indebted to Michael Jackson for a remark that led to this idea.

We write $R_v^{y,a}$ for $r_v^a = y$, $RO_v^{x,a}$ for $r_v^a = x \vee r_v^a = out$, and $D^{x,a}$ for $d^a = x$. These predicates are for # expressions like those in (A1).

stateF	r_v: $Y =$ **if**	$Q_{dec}\#R_v^x$	**then** x	view v decided x	(A1)
	elseif	$Q_{out}\#R_v^{out}$	**then** *out*	view v is out	
	else		*nil*	a view can stay *nil*!	

According to the main idea of Paxos, there should be a decision if there's a decision quorum in some view. Thus

abstract	$\underline{d}$	$=$ **if** $r_v \in X$ **then** r_v **else** *nil*
	input	$=$ *input*

All the variables with short names are stable: r_v^a, d^a, r_v, c_v. In addition, $active_v$, $x \in input$, and $x \in anchor_v$ are stable, although the sets are not because they can grow:

input grows in *Input*;

$anchor_v$ is empty until every earlier view is out or has a choice, and then becomes X or that choice; see (A8) below.

AP maintains the following plausible invariants:

invariant	$d^a \neq nil \Rightarrow (\exists v \mid r_v = d^a)$	decision is a result	(A2)
	$r_v = x \wedge r_u = x' \Rightarrow x = x'$	all results agree	(A3)
	$r_v^a = x \Rightarrow r_v^a = c_v$	agent's result is view's c_v	(A4)
	$c_v = x \Rightarrow c_v \in input \cap anchor_v$	c_v is input and anchored	(A5)
	$r_v^a \neq nil \wedge u < v \Rightarrow r_u^a \neq nil$	*Close*/*Accept*$_v$ do all $u<v$	(A6)

Invariant (A3) ensures a unique decision. To see how to maintain it, we rewrite it with some of the universal quantifiers made explicit so that we can push them around:

$$\forall\, x', u \mid r_v = x \wedge r_u = x' \Rightarrow x = x'$$

By symmetry, we can assume $u < v$. Symbol-pushing and substituting the definition of r_u yields

$$r_v = x \Rightarrow (\forall\, x' \neq x,\ u < v \mid \sim Q_{dec}\#R_u^{x'}) \qquad \text{(A7)}$$

How can we exclude $Q_{dec}\#R_u^{x'}$? In the scope of $x' \neq x$,

$RO_u^{x,a} = (r_u^a = x \vee r_u^a = out) \Rightarrow \sim(r_u^a = x') = \sim R_u^{x',a}$.

Lifting this exclusion to the exclusive decision and out quorums (see section 2.3), we get $Q_{out}\#RO_u^x \Rightarrow \sim Q_{dec}\#R_u^{x'}$. In addition, $c_u = x \Rightarrow \sim Q_{dec}\#R_u^{x'}$ by (A4), since a decision quorum can't be all faulty. Substituting the stronger predicates, we see (A7) is implied by

$$r_v = x \Rightarrow\ (\forall\, u < v \mid c_u = x \vee Q_{out}\#RO_u^x)$$

where we drop the quantifier over x' since x' no longer appears. You might think that by (A4) $Q_{out}\#R_u^{out}$ would be just as good as $Q_{out}\#RO_u^x$, but in fact it's too strong if there are faults, since we can get x from a faulty agent in the quorum even though $c_u \neq x$.

If we limit r_v^a to values of X that satisfy the right hand side, this will be an invariant. With this in mind, we define

stateF	$anchor_v$: **set** $X =$	$\{x \mid (\forall\, u < v \mid c_u = x \vee Q_{out}\#RO_u^x)\}$	(A8)

This says that x is in $anchor_v$ if each view less than v chose x or has an out quorum for (*out* or x). If all the earlier views are out, $anchor_v$ is all of X. If we make c_v anchored and set r_v^a only to c_v, then (A3) will hold.

Note that this definition does not require every previous view to be decided or out (that would be ... $Q_{dec}\#R_u^x \vee Q_{out}\#R_u^{out}$, which is $r_u \neq nil$). It's strong enough, however, to ensure that if there is a previous decision it is the only element of *anchor*, because a decision excludes an out quorum for anything else.

To compute *anchor* directly from the definition (A8), we need to know a choice or out for each previous view. We can, however, compute it recursively by splitting the quantifier's domain at u:

$$\begin{aligned} & anchor_v \\ = \ & \{x \mid (\forall\, w \mid v_0 \le w < v \Rightarrow c_w = x \vee Q_{out}\#RO_w^x)\} \\ = \ & \{x \mid (\forall\, w \mid v_0 \le w < u \Rightarrow c_w = x \vee Q_{out}\#RO_w^x)\} \\ & \cap \{x \mid c_w = x \vee Q_{out}\#RO_u^x\} \\ & \cap \{x \mid (\forall\, w \mid u < w < v \Rightarrow c_w = x \vee Q_{out}\#RO_w^x)\} \end{aligned}$$

We define $out_{u,v} = (\forall\, w \mid u < w < v \Rightarrow r_w = out)$. If this is true, then the third term is just X, so since $c_u \in anchor_u$ by (A5):

$$anchor_v = \{x \mid c_u = x\} \cup (anchor_u \cap \{x \mid Q_{out}\#RO_u^x\}) \quad \text{if } out_{u,v} \qquad \text{(A9)}$$

If $r_u^a = x$ is the latest visible x, then $c_u = x$ by (A4), and the $Close_v$ action below makes all views later than u out and ensures that x is in $anchor_v$; note that this x is not necessarily unique. If all the views earlier than v are out, $anchor_v = X$.

$$anchor_v \supseteq \textbf{if } out_{u,v} \wedge r_u^a = x \textbf{ then } \{x\} \quad \textbf{elseif } out_{v_0,v} \textbf{ then } X \textbf{ else } \{\} \qquad \text{(A10)}$$

In BP, however, r_u^a may not be visible, so we need the more inclusive (A9) to ensure that *Choose* can happen; see section 8.3.

4.3 The algorithm

With this machinery the algorithm is straightforward. We *Choose* an anchored input and then *Accept* (which can't happen until after *Choose*, since it needs $c_v \neq nil$). That leads to a decision, which *Finish* records for posterity. This is the whole story for safety.

Name	**Guard**	**State change**
***Choose*$_v$**	$c_v = nil \wedge x \in input \cap anchor_v$	$\rightarrow c_v := x$
***Accept*$_v^a$**	$c_v \neq nil \wedge r_v^a = nil$	$\rightarrow r_v^a := c_v$; $Close_v^a$
***Finish*$_v^a$**	$r_v \in X$	$\rightarrow d^a := r_v$

For the safety proof, *Input* and *Decision*^{a} simulate *Input* and *Decision* in the spec. All the other actions do not change the abstract state and therefore simulate **skip** in the spec, except for the *Accept* that forms a decision quorum of agents for c_v. This *Accept* simulates *Decide*. However, the agent whose *Accept* simulates *Decide* has no way of knowing this. In fact, if some agent in the quorum fails before anyone else finds out that it accepted c_v, there's no way for anyone to know that there's been a decision. There will be another view, and by the magic of anchoring it will choose the value already decided. This can happen repeatedly, until finally there's a view in which the agents in a decision quorum stay up long enough that others can find out about it; see section 4.4 for an example.

For liveness, however, this is not enough, because *Choose* needs a non-empty *anchor*, which we get by doing *Close* on enough agents to ensure that every previous view either is out or has made a choice. An out quorum is definitely enough. *Anchor* happens when an out quorum has done *Close*; it marks the end of a view change (see section 4.9) even though there's no state change.

***Start*$_v$**	$u<v$ too slow	$\rightarrow active_v := true$	
***Close*$_v^a$**	$active_v$	$\rightarrow$**for all** $u < v$ **do**	**post** $u<v \Rightarrow r_u^a \neq nil$
		if $r_u^a = nil$ **then** $r_u^a := out$	
***Anchor*$_v$**	$anchor_v \neq \{\}$	$\rightarrow$none	

Note that we do *not* need, and do not necessarily get, $r_u \neq nil$, since some agents may never close, and even closing all the agents may yield a view that's neither decided or out.

Agents are just memories; they don't do anything complicated. They cannot be simple read-write memories, however, since they must do the conditional-write operations of *Close* and *Accept*. Disk Paxos (section 5) implements AP without conditional writes.

With these actions AP finishes provided there are quorums of OK agents and a final view that is the last one in which *Close* actions occur; see section 4.5 for details.

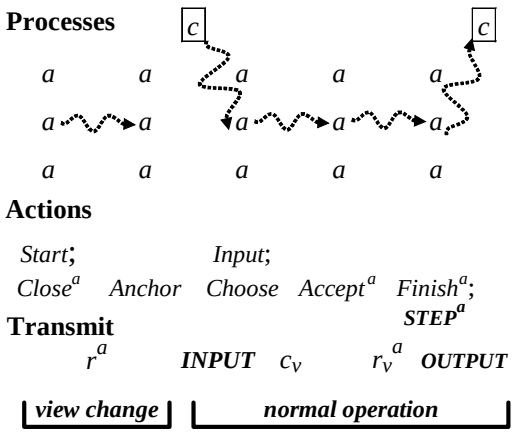

Figure 1: Abstract Paxos

Figure 1 shows the actions of AP for one complete view. It shows communication with vague wavy arrows, since AP abstracts away from that, but the "transmit" part says what information needs to flow to enable the next action. Capitalized items refer to the state machine application of Paxos: the client (boxed in the figure) provides an input request, the machine takes a step, and it sends the client some output. If there are no faults, any agent could send the output.

4.4 Example

An example may help your intuition about why this works when there are no faults. The table shows views 1-3 in two runs of AP with agents *a*, *b*, *c*, two agents in a quorum, and *input* = {7, 8, 9}.

	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	8	*out*	*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* ⊇{8} seeing *a, b* ⊇{9} seeing *a, c* or *b, c*				{9} no matter what quorum we see			

In the left run all three views are out, so if we compute $anchor_4$ by seeing all three agents, we can choose any input value. If we see only *a* and *b*, view 3 appears out but view 2 does not, and hence we must choose 8. If we see only *a* and *c* or *b* and *c*, view 3 doesn't appear out and hence we must choose 9.

In the right run, view 2 is decided, but we don't see that if either *a* or *c* is stopped. Nonetheless, we must choose 9, since we see that value in a non-out view no matter what out quorum we see. Thus a successful view such as 2 acts as a barrier which keeps any later view from choosing another value.

The reason there were three views in both runs is that each view was interrupted by *Close* in a later one before it had a chance to finish. Otherwise view 1 would have succeeded. If new views keep starting, the algorithm can continue indefinitely.

4.5 Liveness

We want AP to finish in a final view *v* if there's no *Close* action in any later view. It will do so if the actions can see certain things:

- $Finish_v$ must see a decision *d* (that is, must see $Q_{dec}\#R_v^d$). This means that Q_{dec} must be live. Since there are no later views to mess with r_v^a, if Q_{dec} is live *Accept* will eventually run at enough agents to make *d* visible. However, *d* need not be visible in the view that made it. In fact, it's fundamental to Paxos that until *Finish* has run at a live quorum, you may have to run another view to make *d* visible. This can only happen in *u*, however, if a later view does *Close* and sets some r_u^a to *out.*
- $Accept_v$ must see the choice c_v, though again perhaps not in every view if processes fail at inopportune times. This depends on how *c* is represented, which differs in different implementations, as we shall see. It is trivial with no faults: one process, called a *primary*, chooses c_v and announces it, which works if there's only one primary for *v* and it doesn't stop. With faults, BP uses a quorum to get a unique and visible *c*, which works if all OK processes choose the same *c* and the quorum is live.
- $Choose_v$ must see a non-empty *anchor*. Since this doesn't get easier when you run another view, we insist that it be true in every view. This means that *every* previous view *w* must become visibly out ($Q_{out}\#R_w^{out}$ is visible) back to a view *u* that has a visible choice (A10) or at least is visibly anchored (A9). Hence Q_{out} must be live. Since *anchor* involves the choice, this also depends on the implementation.

If Q_{out} is live, $Close_v$ always leads eventually to a visible out quorum of OK agents in every $u < v$. In this quorum either every agent is out, in which case *u* is out, or some $r_u^a = c_u$ by (A4). So if no faults are allowed, we get a non-empty $anchor_v$ immediately from this out quorum by (A10). If there are faults, there may be other out quorums as well, in which we see $r_u^a = x \neq c_u$ if *a* is faulty. Since we can't tell which out quorum is OK, (A10) isn't enough to anchor *v*. We need (A9) and some delicate reasoning; see section 8.3.

A view can finish by seeing only an out quorum (for $x \in anchor_v$, which *Choose* needs) and a decision quorum (for $r_v = x$, which *Finish* needs). Thus the requirement is Q_{out} and Q_{dec} both live. With no faults and size-based quorums, both quorums are the same: more than *n*/2 agents.

How do we know $anchor_v \cap input \neq \{\}$?

4.6 Scheduling

Doing *Close* in views later than *v* may keep $Accept_v$ from happening, by setting too many r_v^a to *out* before $Accept_v$ has a chance to set them to c_v; of course this can't happen in the final view because there are no later views. To get a final view, $active_v$ controls the scheduling of *Close*. Since asynchronous consensus can't be guaranteed to terminate, there is no foolproof way to do this scheduling.

Schedulers either randomize or estimate the longest time *RT* for a round-trip from one process to another and back; note that *RT* includes the time for the processes to run as well as the time for the messages to travel. The idea is that if a view doesn't complete within 2*RT*, you multicast a new view *v*. If *v* is smaller than any other view you hear about within another *RT*, *v* becomes active. Obviously this can fail in an asynchronous system, where there is no guarantee that the *RT* estimate is correct.

Castro and Liskov [1] use the Ethernet's exponential backoff technique to estimate *RT*; a process backs off whenever it fails to hear from a quorum within its current *RT* estimate. This works as long as *RT* does not increase without bound. The estimate can be as much as *b* times the actual *RT*, where *b* is the backoff multiplier, commonly 2. More serious is that if processes stop and then recover, the estimate may be much too large.

Summing this up, with proper scheduling AP finishes as soon as there are Q_{dec} and Q_{out} quorums of processes that haven't failed. We can't implement proper scheduling in general, but it's not hard in most practical situations.

4.7 Cleanup

Once a $Q_{\sim F}{}^{+}$ quorum knows a decision all the other state can be discarded, since no matter what failures occur there will be a good quorum to report *d*.

Isn't it enough to know $d^a \neq nil$? If so, fix D and all *Cleanup*s.

Cleanup^a $\quad Q_{\sim F}{}^{+}\#D^x \quad \rightarrow$**for all** v **do** $r_v{}^a := nil$; $input := \{\}$

The decision itself can be discarded once the client knows about it. In the state machine application the decision must be kept until the state change it causes has been recorded in a sufficiently persistent way; this is the same as the rule for truncating the log in a transaction system.

4.8 Optimizing agent state

$Close_v{}^a$ leaves $r_u{}^a$ out or c_u for all $u < v$, and by (A10) *anchor* only depends on the latest view with $r_u{}^a = x$. Hence an agent *a* only needs to keep track of the latest view *u* for which $r_u{}^a = x$ and the range (maybe empty) of later views *w* for which $r_w{}^a = out$. The following variables do this:

$vX_{last}{}^a$	the latest *u* for which $r_u{}^a = x$ (v_0 if there is no such *u*)
$x_{last}{}^a$	*x* (arbitrary if there's no such *u*), and
$v_{last}{}^a$	the earliest $v \geq u$ for which $r_u{}^a \neq out$.

For views *w* between $vX_{last}{}^a$ and $v_{last}{}^a$, $r_w{}^a = out$; for views past $v_{last}{}^a$, $r_w{}^a = nil$. Thus $vX_{last}{}^a = u \neq v_0$ and $x_{last}{}^a = x$ encode the predicate $r_u{}^a = x$, and $vX_{last}{}^a = u$ and $v_{last}{}^a = v$ encode

$\forall\, w \mid (u<w<v \Rightarrow r_w{}^a = out) \wedge (v<w \Rightarrow r_w{}^a = nil) \wedge (u \neq v \Rightarrow r_v{}^a = nil)$.

This encoding uses space logarithmic rather than linear in the number of views, which makes it cheaper both to store and to transmit the agent state. In practice, of course, we use a fixed amount of space for a view. *Close* and *Accept* become

Close$_v^a$	$active_v \wedge v_{last}{}^a < v$	$\rightarrow v_{last}{}^a := v$
Accept$_v^a$	$c_v \neq nil \wedge v_{last}{}^a = v$	$\rightarrow vX_{last}{}^a := v$; $x_{last}{}^a := c_v$; $v_{last}{}^a := v$

4.9 Multi-step optimizations

When we use Paxos (or any other consensus algorithm) to implement a replicated state machine, we need to reach consensus on a sequence of values: the first step of the state machine, the second step, etc. By observing that $anchor_v$ does not depend on c_v, we can compute it in parallel for any number of steps. For most of these, of course, there will have been no previous activity, so the agent states for all the steps can be represented in the same way. We only need to keep track of the last step for which this is not true, and keep separate *last* triples just for this and any preceding steps that are not decided. To bound this storage, we don't start a step if too many previous steps are not known to be decided.

With this optimization we do *Close* and compute *anchor* only when the view changes, and we can use one view for a whole sequence of steps. Each step then requires *Choose* and *Accept* to reach a decision, and *Finish* to tell everyone. Finish can be piggybacked on the next accept, so this halves the number of messages.

It's possible to run several steps in parallel. However, in the state machine application the ordering of steps is important: to maintain external consistency a step should not decide an input *x* that arrives after a later step decided *y* and sent its output. Otherwise the clients will see that the inputs execute in the order *x*; *y* even though they also see that *x* was not submitted until after *y* completed; this is generally considered to be bad. To avoid this problem, fill any gaps in the sequence of steps with a special **skip** step. Of course there shouldn't be nothing but skips.

If there are lots of state machine steps they can be batched, so one run of AP decides on many steps. This is like group commit for transactions [6], with the same tradeoffs: more bandwidth but greater latency, since the client gets no output until a batch runs.

4.10 Other optimizations

An agent can send its $r_v{}^a$ directly to the client as well as to the other agents, reducing the client's latency by one message delay. Of course the client must see the same result from a decision quorum of agents; otherwise it retransmits the request. A state machine agent can tentatively do a step and send the output to the client, which again must wait for a decision quorum. In this case the agent must be able to undo the step in case *v* doesn't reach a decision and a later view decides on a different step. Castro and Liskov call this 'tentative execution' [1].

If a step is read-only (doesn't change the state of the state machine), an agent can do it immediately and send the client the output. The client still needs a decision quorum, which it may not get if different agents order the read-only step differently with respect to concurrent write steps that affect the read-only result. In this case, the client must try the step again without the read-only optimization.

If the only reason for running AP is to issue a lease, the agents don't need persistent state. An agent that fails can recover with empty state after waiting long enough that any previous lease has expired. This means that you can't reliably tell the owner of such a lease, but you don't care because it has expired anyway. Schemes for electing a leader usually depend on this observation to avoid disk writes.

It's convenient to describe an algorithm in terms of the persistent variables. In practice we don't keep each one in its own disk block, but instead log all the writes to them in a persistent log. In some applications this log can be combined with the log used for local transactions.

5 Disk Paxos

We would like to implement the agent with memory that has only read and write operations, rather than the conditional writes that AP does in *Close* and *Accept*. The main motivation for this is to use commodity disks as agents; hence the name Disk Paxos (DP) [5]. These disks implement block read and write operations, but not the conditional-write operations that AP agents use.

To this end we add separate state variables $rx_v{}^a$ and $ro_v{}^a$ in the agent for *x* and *out*, and change *Close* and *Accept* to unconditionally write *out* into *ro* and c_v into *rx*. We want the code to look only at the values of *rx* and *ro*, so that $r_v{}^a$ becomes a history variable, that is, the behavior of the algorithm is unchanged when we remove it.

What makes this work is an invariant that allows us to infer a lot about $r_v{}^a$ from $rx_v{}^a$ and $ro_v{}^a$:

invariant relates state to history (D1)

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

In particular, if *anchor* is non-empty we can still always compute at least one of its elements, because the only information lost is some cases in which the view is out, and in those cases we get $r_v{}^a$ instead, which is enough by (A10). We may miss *anchor* = *X*, but we only need a non-empty *anchor* (and this can happen in AP as well if we don't hear from some agents that are out). We may

also sometimes miss a decision because we only know $r_v^a \neq nil$ when actually $r_v^a = x$, but this only costs another view (and this too can happen in AP if we don't hear from some agents that accept). In the final view $ro_v^a = nil$ and we don't lose any information, so liveness is unaffected.

var	$\boxed{rx_v^a}$	: $Y := nil$		Result X
	$\boxed{ro_v^a}$	: $Y := nil$		Result *out*
	r_v^a	: $Y := nil$		*history*
***Close*$_v^a$**	$active_v$	→**for all** $u < v$ **do** $\boxed{ro_u^a := out;}$ **if** $r_u^a = nil$ **then** $r_u^a := out$	**post** $u < v \Rightarrow r_u^a \neq nil$	
***Choose*$_v$**	$c_v = nil \wedge x \in input \cap anchor_v$	→ $c_v := x$		
***Accept*$_v^a$**	$c_v \neq nil$	→ $\boxed{rx_v^a := c_v;}$ *Close*$_v^a$; **if** $r_v^a = nil$ **then** $r_v^a := c_v$		

invariant $rx_v^a = x \Rightarrow rx_v^a = c_v$ (D2)

With the abstraction $r_v^a = r_v^a$, DP simulates AP.

A more general version encompasses both AP and DP, by allowing either a conditional or an unconditional write in *Close* and *Accept*. It replaces the boxed sections with the following:

***Close*$_v^a$** … **if** $rx_u^a = nil$, or optionally anyway, $ro_u^a := out$

***Accept*$_v^a$** … **if** $ro_u^a = nil$, or optionally anyway, $rx_v^a := c_v$

Liveness and scheduling are the same as for AP. The *last*-triple optimization needs special handling; it is discussed in section 0.

6 Communication

For the algorithm to progress, the processes must communicate. We abstract away from messages by adding to m's state a stable predicate T^m called its 'truth' that includes everything m knows to be true from others; T also stands for 'transmitted'. If g is a stable predicate, we write $g@m$ for $T^m \Rightarrow g$, and read it "m knows g" or "m sees g" or "g is visible at m". The safety invariant is

invariant $g@m \Rightarrow g$ (T1)

In other words, everything a non-faulty process knows is actually true. This invariant allows us to replace a non-local guard g in an action at m with the stronger local $g@m$. The resulting code makes fewer transitions and therefore satisfies all the safety properties of the original, non-local code. Liveness may be a challenge.

We lift $@$ to process predicates: $G@m = (\lambda\, k \mid G^k@m)$. Then $(Q\#G)@m = Q\#(G@m)$: seeing G from a quorum is the same as seeing a quorum for G. We also write $Q\#g$ for $Q\#(\lambda\, m \mid g@m)$, read "a Q quorum knows g"

Note that m can't communicate $g@m$ if m might be faulty. We write $g@m^*$ for $g@m \vee F^m$, which m can communicate. This is not an issue when we use $g@m$ in a guard at m, but we can only get $(g@m^*)@k$ rather than $g@k$.

The implementation, of course, is that $g@m$ becomes true when m receives a message from k asserting g; recall that g is stable and therefore cannot become false if k fails. We model the message channel as a set ch of terms $g_{k\to m}$ (read "g from k to m"). Here are all the actions that affect ch or T:

Name	Guard	State change	
***Local*^{k}(*g*)**	g	$\to T^k := T^k \wedge (g \vee F^k)$	**post** $g@k^*$
***Send*^{k,m}(*g*)**	$g@k$	$\to ch := ch \cup \{g_{k\to m}\}$	**post** $g_{k\to m} \in ch$
***SendF*^{k,m}(*g*)**	F^k	$\to ch := ch \cup \{g_{k\to m}\}$	**post** $g_{k\to m} \in ch$
***Receive*^{m}(*g*)**	$g_{k\to m} \in ch$	$\to T^m := T^m \wedge g@k^*$	**post** $(g@k^*)@m$
***Drop*(*g*)**	$g_{k\to m} \in ch$	$\to ch := ch - \{g_{k\to m}\}$	

So k can use *Local* to add to T^k any true predicate g; presumably g will only mention k's local state, since otherwise it would be in T^k already.[6] Then k can send $g_{k\to m}$ to any process m if either k knows g is true or k is faulty. We separate the two send actions because *SendF* is not fair: there's no guarantee that a faulty process will send any messages.

invariant $g_{k\to m} \in ch \Rightarrow g@k^*$ (T2)

$g@k^*@m \Rightarrow g@k^*$ (T3)

From the two send actions we have (T2) since g is stable and therefore $g@k^*$ is stable. *Receive*^{m}(g) adds $g@k^*$ to m's truth. Since this is the only way to establish $g@k^*@m$, (T3) follows from (T2). (T1) follows from this and *Local*, since they are the only ways to establish $g@m$.

These actions express our assumption that the only way m can receive g from a non-faulty k is for g to be true. In other words, there's no way to fake the source of a message. Usually we get this security either by trusting the source address of a message or by cryptographic message authentication; see [1] for details of how this works for BP.

A history variable can appear in a predicate g in T^k, even though it can't appear directly in a guard or in an expression assigned to an ordinary variable, since it's not supposed to affect the actions that occur. Such a g can get into T^k initially if an invariant (such as (C1)) says it's implied by a g' that doesn't contain a history variable. Once it's in T^k, g can be transmitted in the usual way. This is just a way of encoding "g' was true at some time in the past". So if g' has no history variables, and g and $g' \Rightarrow g$ are stable:

***Local*^{k}(*g*)** $g' \wedge (g' \Rightarrow g) \to T^k := T^k \wedge (g \vee F^k)$ **post** $g@k^*$

We now abstract away from the channel to actions that establish $g@m$ directly:

> k can transmit $g@k$ to all the other OK processes, even if k fails. This allows for messages that remain in ch after k fails.
>
> A faulty k can transmit anything.

***Transmit*^{k,m}(*g*)** $g@k \wedge OK^m \to T^m := T^m \wedge g@k^*$ **post** $g@k^*@m$

***TransmitF*^{k,m}(*g*)** $F^k \wedge OK^m \to T^m := T^m \wedge g@k^*$ **post** $g@k^*@m$

We say that m hears $g@k^*$ from k. When there's a quorum $Q\#G@m$, we say that m hears G from a Q quorum. In the simulation proof *Receive*^{m}(g) of $g_{k\to m}$ simulates *Transmit*^{k,m}(g) by (T2) because $g@k$ is stable, and the *Send* actions simulate **skip**.

As before, both *Transmit*^{k,m} and *Broadcast*^{k,m} (see below) are fair if k is OK, and so is *Broadcast*^{m}, but *TransmitF* is not. This means that if $g@k$, OK^k, and OK^m continue to hold, then eventually $g@k^*@m$ or $g@m$ will hold.

The *Local*, *Transmit*, and *Broadcast* actions are the only ones we need for the rest of the paper.

6.1 Broadcast

If a $Q\sim_F{}^+$ quorum ever knows g, then henceforth there's always a $Q\sim_F$ quorum of OK processes that knows g. Hence repeated *Transmit*s will establish $(Q\sim_F\#g)@m$ at every OK process m. But $Q\sim_F\#g \Rightarrow g$, so this establishes $g@m$. We package this in an action:

***Broadcast*^{m}(*g*)** $Q\sim_F{}^+\#g \wedge OK^m \to T^m := T^m \wedge g$ **post** $g@m$

If we have broadcast messages (signed by public keys) there's a more direct way to broadcast a predicate. We can drop the m from $g_{k\to m}$, since any process can read the messages.[7] This means that if

[6] The "$\vee F$" is there to simplify the definition of *Broadcast*^{k,m} in section 6.1

[7] A more careful treatment would reflect the fact that a receiver must remember the message and its signature in order to forward them, since $G_{k\to m}$ may disappear from ch.

Receive^k establishes *g@k*, then *g@m* follows too, not simply *g@k*@m*. In other words, *k* can transmit *g@k* without weakening it, by simply forwarding the messages that *k* received. If *g@k* follows from *Local*^{k}, *g@k* = *g@k**. Thus, provided *k* remembers the signed evidence for *g*, it can do

***Broadcast*^{k,m}(*g*)** $g@k \wedge OK^m \rightarrow T^m := T^m \wedge g$ **post** $g@m$

6.2 Implementation and scheduling

We transmit predicates, but since they take only a few forms, an implementation encodes a predicate as a message with a *kind* field that says what kind of predicate it is, plus one field for each part of the predicate that varies. For example, after doing *Close*$_v^a$ agent *a* sends (*closed-state, a, last-triple*^{a}).

The *Send* actions that implement *Transmit* need to be scheduled to provide congestion and flow control, any necessary retransmission, and prudent use of network resources. How this is done depends on the properties of the message channel. For example, TCP is a standard way to do it for unicast on an IP network. For a multicast such as *Broadcast*, scheduling may be more complex.

Since processes can fail, you may have to retransmit a message even after a quorum has acknowledged its delivery.

7 Classic Paxos

To turn AP into an implementation, we can take AP's agent almost as is, since the agent's *Close*, *Accept*, and *Finish* actions only touch its local state r_v^a. We need to implement *input*, *active*$_v$, and c_v, which are the non-local variables of AP, and the *Input*, *Start*, and *Choose* actions that set them. We also need to tell the agents that they should invoke their actions, and give them *active*$_v$ and c_v. Our first implementation, CP, tolerates stopped processes but no faults.

Since CP is a real implementation, the actions refer only to local state. We still use shading, but now it marks state in *T* transmitted from other processes. We discuss the scheduling of these *Transmit* actions in section 7.1. Look at Table 4 to see how non-local information in AP becomes either local state or transmitted information in CP and BP.

CP implements AP by doing *Input*, *Start*, and *Choose* in a *primary* process. For fault tolerance there can be several primaries. However, for each view there is exactly one process that can be its primary; in other words, there is a function *p*(*v*) that maps each view to its primary. If a primary never reuses a view for which it has already chosen a value, there is at most one c_v for each *v*. A simple implementation of p_v is to represent a view by a pair, with the name of its primary as the least significant part.

An agent's state must be persistent, but we allow a primary to reset, lose its state, and restart in a fixed state. Then it starts working on a new view, one for which it never chose a result before. We discuss later how to find such a view.

The primary's job is to coax the agents to a decision, by telling them when to close, choosing c_v, and relaying information among them. Once it has a new view, the primary's *Choose* action chooses an anchored value c_v for the view. To do this it must collect enough information from the agents to compute a non-empty subset of *anchor*$_v$. (A10) tells us how much information suffices: either that all previous views are out, or that all views since *u* are out and c_u. So it's enough to trigger *Close*$_v^a$ at an out quorum (with *Close*^{p}) and then collect the state from that quorum.

Once the primary has c_v, it can try (with *Accept*^{p}) to get the agents to accept it. They respond with their state, and if the primary sees a decision quorum for c_v, then there is a decision which the primary can tell all the agents about (with *Finish*^{p}).

We fearlessly overload variable names, so we have c_v and c^p, for example, and *v* and v^p.

The agent variables of AP become agent variables of CP.

var	r_v^a	:	$Y := nil$, but $r_{v_0}{}^a := out$	Result
	d^a	:	$Y := nil$	Decision; not *out*

All the other variables of AP become primary variables of CP, except that *active*^{p} is coded by v^p:

type	P		$\subseteq M = \ldots$	Primary
var	v^p	:	$V := v_0$	Primary's View
	c^p	:	$Y := nil$	Primary's Choice
	input^p	:	**set** $X := \{\}$	
stateF	*ctive*^{p} =		$(v^p \neq v_0)$	

These are not stable across resets, so we add history variables that are, with the obvious invariants relating them to c^p and *active*^{p}.

var	c_v	:	$Y := nil$	*history*
	input	:	**set** $X := \{\}$	*history*
	active$_v$	:	*Bool* := *false*	*history*

invariant $active^p \wedge c^p \neq nil \Rightarrow c^p = c_{vp}$ (C1)

$input^p \subseteq input$

$active^p \neq nil \Rightarrow active^p = active_{vp}$ (C2)

Thus all the variables of AP are also variables of CP with the identity abstraction function to AP. The invariants (A2-A6) of AP are also invariants of CP.

Any primary can accept an input. For a state machine, this means that any primary can receive requests from clients. The client might have to do *Input*^{p} at several primaries if some fail.

***Input*^{p}(x)** $input^p := input^p \cup \{x\}$; $input := input \cup \{x\}$

We define the primary's estimates of r_v and *anchor*$_v$ in the obvious way. We define re_v^p rather than just re^p because *p* needs views earlier than v^p to compute *anchor*. From (A1) for r_v:

stateF	re_v^p =	**if**	$(Q_{dec}\#R_v^x)@p$	**then** *x*	view *v* decided *x*	(C3)
		elseif	$(Q_{out}\#R_v^{out})@p$	**then** *out*	view *v* is out	
		else		*nil*	a view can stay *nil*!	

From (A10) for *anchor*:

	$out_{u..v}^p$ =	$(\forall w \mid u < w < v \Rightarrow re_w^p = out)$			
stateF	$anchor^p \supseteq$	**if**	$out_{u..v}^p \wedge (r_u^a = x)@p$	**then**	$\{x\}$ (C4)
		elseif	$out_{v_0..v}^p$	**then**	X
		else			$\{\}$

We avoid a program counter variable by using the variables v^p and c^p to keep track of what the primary is doing:

v^p	c^p	***p*'s view of agent state**	**Action**
$= v_0$	-	-	*Start*^{p}
$\neq v_0$	$= nil$	$anchor^p = \{\}$	*Close*^{p}
$\neq v_0$	$= nil$	$anchor^p \neq \{\}$	*Choose*^{p}
$\neq v_0$	$\neq nil$	$re_{vp}^p \notin X$	*Accept*^{p}
$\neq v_0$	$\neq nil$	$re_{vp}^p \in X$	*Finish*^{p}

To keep c_{vp} stable we need to know $c_{vp} = nil$ before setting it. The following invariant lets us establish this from local state:

invariant $active^p \wedge c^p = nil \Rightarrow c_{vp} = nil$ (C5)

To maintain this invariant we put a suitable guard on the *Start*^{p} action that makes *p* active. This is an abstract action since it involves c_v; section 7.3 discusses how to implement it.

***Start*$_v^p$** $u < v$ too slow $\wedge\ p_v = p \wedge c_v = nil$ $\rightarrow v^p := v;\ c^p := nil;\ active_{vp} := true$

With this machinery, we can define *Choose*^{p} as a copy of AP's *Choose*$_v$, with *active*^{p} added to the guard and the primary's versions

of *c*, *input*, and *anchor* replacing the truth. (C3) and (C5) ensure that $Choose_v$'s guard is not weakened.

$$\textbf{Choose}^p \quad \boxed{active^p} \wedge c^p = nil \quad \rightarrow \boxed{c^p := x;}\ c_{vp} := x \\ \wedge\ x \in input^p \cap anchor^p$$

The agent's actions are the same as in AP with $c_v@a$ and $re_v{}^p@a$ for c_v and r_v. With these actions it's easy to show that CP simulates AP, using (A2-A6) and two more unsurprising invariants:

$$\textbf{invariant}\ \ re_v{}^p \neq nil \Rightarrow re_v{}^p = \underline{r}_v \quad (C6)$$
$$anchor^p \subseteq \underline{anchor}_{vp} \quad (C7)$$

We can use the *last* optimization in CP just as in AP, and of course the view change optimization works the same way.

7.1 Communicating with agents

As we saw above, the definitions of $re_v{}^p$ and $anchor^p$ imply that the agents tell the primary their state after $Close^a$ and $Accept^a$. In addition, the primary tells the agents when to close, and what values to use for accept and finish. It implements these actions by sending trigger messages to the agents, using the invariants shown; since we are abstracting away from messages, we describe them informally. The agents respond by returning their state.

$\textbf{Close}^p$	$active^p \wedge c^p = nil$	→trigger $Close_v{}^a$ at all agents, sending v^p, $active^p$ as *v*, $active_v$ (C2)
$\textbf{Anchor}^p$	$anchor^p \neq \{\}$	→none
$\textbf{Accept}^p$	$c^p \neq nil \wedge re_{vp}{}^p = nil$	→trigger $Accept_v{}^a$ at all agents, sending v^p, c^p as *v*, c_v (C1)
$\textbf{Finish}^p$	$re_{vp}{}^p \in X$	→trigger $Finish_v{}^a$ at all agents, sending v^p, $re_{vp}{}^p$ as *v*, r_v (C6)

Figure 2 shows the actions of CP for one complete view, with $n = 3$ agents; compare figure 1. The arrows show the flow of messages, and the "transmit" part shows their contents and whether they are unicast or multicast. An n^* means that if the primary is also an agent, only $n - 1$ messages need to flow. To finish, of course, only a quorum of agents is needed, and only the corresponding messages. In normal operation, however, when no processes are stopped, it's desirable to keep all *n* of them up to date, so they should all get at least the *Finish* message.

Liveness, scheduling, and cleanup are the same as AP's. A primary can discard all its state at any time with *Reset* (section 7.3).

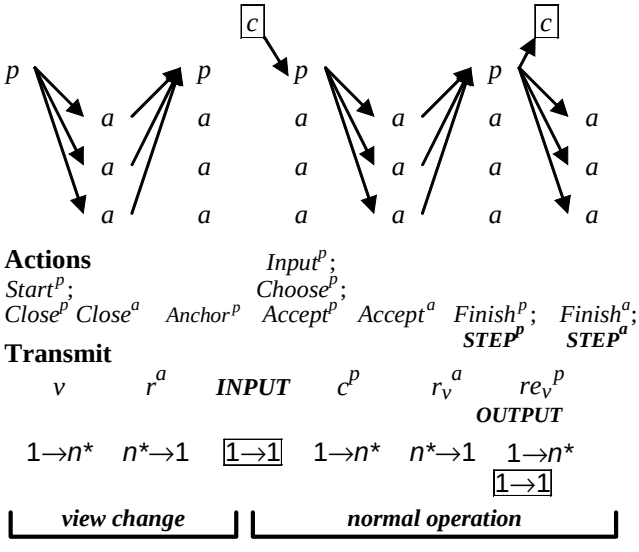

Figure 2: Classic Paxos

In practice the primary is usually one of the agents, and only two other agents are needed to tolerate one stopped process. It's also possible to compute only at the primary and use the agents just to store the state of the state machine; in this case the *Finish* message contains the state changes instead of *d*.

7.2 Implementing DP

Implementing DP with CP is completely straightforward except for the log-space representation of the agent state. We can't just use the triple of *last* values, because if a primary overwrites one of those unconditionally with an earlier view, it will change some $r_v{}^a$ back to *nil*. Instead, we keep a triple for each primary, so the state of an agent is the *last* triple as in AP, but each component is a function from *p* to a value (implemented, of course, as an array indexed by *p*). Then the primary rather than the agents can enforce the guards on writing the agent state, since each variable has only one writer. We abstract vX_{last} and v_{last} as the maximum over the primaries, and x_{last} as the value that goes with vX_{last}. Reading an agent's state thus requires reading the triples for all the primaries.

This read operation is not atomic, however, so these abstractions are not enough to show that DP-*last* implements AP-*last*. Fortunately, they don't need to be, since what we care about is implementing DP. For this we don't need the *last* values but only enough information about $rx_u{}^a$ and $rx_u{}^a$ to do the actions. As we saw in section 4.8, each $last_p{}^a$ triple encodes two predicates on r^a, and all of them together encode the conjunction of the predicates. Thus setting $vX_{last,p}{}^a := u$ and $x_{last,p}{}^a := x$ is equivalent to setting $rx_u{}^a := x$, and setting $x_{last,p}{}^a := u$ and $v_{last,p}{}^a := v$ is equivalent to setting $ro_w{}^a := out$ for all *w* between *u* and *v*. (In addition, some information about earlier values of rx^a and ro^a may be lost, but nothing is changed.) There's never a contradiction in these predicates, because c_v is the only value we write into $rx_v{}^a$. By reading all the triples, we get a predicate that implies the facts about rx^a and ro^a that would follow from:

$$\underline{vX_{last}}^a = \text{max over } p \text{ of } vX_{last,p}{}^a$$
$$\underline{x_{last}}^a = x_{last,p}{}^a \text{ for the } p \text{ for which } vX_{last,p}{}^a = \underline{vX_{last}}^a$$
$$\underline{v_{last}}^a = \text{max over } p \text{ of } v_{last,p}{}^a$$

It follows that DP-*last* implements DP.

A primary *p* can write all three values at once provided it finds suitable values $vX_{last,p}$ and $x_{last,p}$ to write into $vX_{last,p}{}^a$ and $x_{last,p}{}^a$ in *Close*. This is useful because it allows *a* to keep the whole triple in a single disk block. The values already there are suitable; so are those that accompany the largest $v_{last,p}{}^a$ in an out quorum.

Precisely, we have:

$$\textbf{Close}_{v,p}{}^a \quad v_{last,p}{}^a := v; \\ x_{last,p}{}^a := x_{last,p};\ vX_{last,p}{}^a := vX_{last,p}$$
$$\textbf{Accept}_{v,p}{}^a\ c_v \neq nil \quad \rightarrow v_{last,p}{}^a := v;\ vX_{last,p}{}^a := v;\ vX_{last,p}{}^a := c_v$$

7.3 Finding a new view

If the primary has a little persistent state, for example a clock, it can use that to implement $Start^p$, by choosing (*clock*, *p*) as a *v* that it has never used before, which ensures $c_v = nil$.

To get by without any persistent state at the primary, $Start^p$ queries the agents and chooses a view later than some view in which a decision quorum of agents is not closed.

$$\textbf{Reset}^p \quad v^p := v_0;\ input^p := \{\};\ c^p := nil$$
$$\textbf{Start}_v{}^p \quad u < v \text{ too slow} \quad \rightarrow v^p := v;\ active_{vp} := true \\ \wedge\ \sim active^p \wedge p_v = p \\ \wedge\ (\exists\ u < v \mid Q_{dec}\#R_u{}^{nil})@p$$

This works because before choosing a result, a primary closes an out quorum at all previous views, and the two quorums must intersect. The invariants we need are (A6) and

invariant $Q_{dec}\#R_u^{\,nil} \wedge v > u \Rightarrow c_v = nil$ (C8)

This argument is trickier than it looks, since $Q_{dec}\#R_v^{\,nil}$ is not stable. The true, stable condition is "at some time after the primary reset, a decision quorum of agents was still open". Then p can conclude $c_v = nil$ if $p_v = p$, since only p can change c_v. To establish this condition, the query must not get a reply that was generated before the reset. We can ensure this if there's a known upper bound on how long the reply can take to arrive (which is true for SCSI disks, for example), or with standard techniques for at-most-once messages on channels with unbounded delays. Unfortunately, the latter require some persistent state in the primary, which is what we are trying to avoid. We won't formalize this argument.

If the primary sees any agent out in v^p or sees any non-*nil* agent variable for a bigger view u, it restarts, since this means that a later view has superseded the current one. To restart, p chooses one of its views that is bigger than any it has seen to be out. This is another implementation of the abstract $Start^p$, more efficient when the primary's state hasn't been lost.

***Restart*$_v^p$** $active^p \wedge v^p < u < v \wedge p_v = p \;\rightarrow v^p := v;\; c^p := nil$
$\wedge\, (\exists a \mid (re_{vp}{}^a = out)@p$
$\vee\, (r_u{}^a \neq nil\,)@p)$

7.4 Performance

As figure 2 shows, a normal run of CP that doesn't need a view change multicasts two messages from the primary to the agents, and each agent sends one reply. The output to the client can go in parallel with the second multicast, so that the client's latency is one client-primary round-trip plus one primary-agents round trip. Usually the finish message piggybacks on the accept message for the next step, so its cost is negligible. Cleanup takes another (piggybacked) agents-primary-agents round trip. See table 1.

A view change adds another primary-agents round trip, and if the primary has to run *Start*, there is a third one. The last only happens when the primary crashes, however, in which case this cost is probably small compared to others.

For a more detailed analysis see [3].

8 Byzantine Paxos

BP is a different implementation of AP, due to Castro and Liskov [1], that tolerates arbitrary faults in Z_F of the agents. Their description interweaves the consensus algorithm and the state machine, and distinguishes the primary and the other agents (called 'backups') much more than we do here. They use different names than ours; see table 2 in the appendix for a translation.

With faults it is unattractive to have separate primary processes for *Choose* or for relaying information among the agents, so we do *Choose* in the agents and use multicast for communication among them. Thus BP starts with AP, keeps all the agent variables $r_v{}^a$ and d^a, and adds agent versions of the other variables.

const	Q_{ch}	:	**set** Q := …	choice Quorum set
var	$r_v{}^a$	:	$Y := nil$, except $r_{v0}{}^a := out$	Result
	d^a	:	$Y := nil$	Decision; not *out*
	$c_v{}^a$	:	$Y := nil$	Choice; not *out*
	$input^a$	:	**set** X := {}	
	$active_v{}^a$	:	$Bool := false$	

Thus all the variables of AP are also variables of BP with the identity abstraction function, except for c_v and *input*. The abstraction to $\underline{c}_v$ is a choice quorum of the agents' choices, and to $\underline{input}$ is the union of the agents' $input^a$ as in CP. We write $C_v^{\,y,a}$ for the predicate $c_v{}^a = y$, and $Inp^{x,a}$ for $x \in input^a$.

abstract	$\underline{c}_v$	= **if** $Q_{ch}\#C_v^{\,x}$ **then** x **else** nil
	$\underline{input}$	$= \cup_{a \in A}\, input^a$

Agent a adds a value to $input^a$ when a client transmits it; we don't formalize this transmission. Since clients can also fail, other agents may not see this value: $Input^a$ isn't fair.

***Input*^{a}(*x*)** $input^a := input^a \cup \{x\}$

There's still only one choice $\underline{c}_v$ for a view, however, because Q_{ch} excludes itself, and the quorum must agree that the input came from the client. Thus any decision is still for a client input and still unique no matter how many faulty clients there are. For the effect of faulty clients on liveness, see the end of section 8.3.

We define $ce_v{}^a$ as a's estimate of $\underline{c}_v$, like $\underline{c}_v$ except for the "$@a$":

stateF $ce_v{}^a$ = **if** $(Q_{ch}\#C_v^{\,x})@a$ **then** x **else** nil a's estimate of c_v (B1)

Similarly, a quorum of $r_v{}^a$s makes a result (as in AP), and $re_v{}^a$ is a's estimate of that result, the same as CP's $re_v^{\,p}$:

stateF	$re_v{}^a$ =	**if**	$(Q_{dec}\#R_v^{\,x})@a$	**then** x	view v decided x (B2)
		elseif	$(Q_{out}\#R_v{}^{out})@a$	**then** *out*	view v is out
		else		nil	view can stay *nil*!

The state function $\underline{r}_v$ is defined in AP; it's (B2) without the "$@a$".

With these definitions, the agents' non-*nil* estimates of r and c agree with the abstract ones, because they all come from stable quorums and (A4) means we see at most one $r_v{}^a$ from the OK agents. These invariants are parallel to (C1) and (C6):

invariant	$ce_v{}^a \neq nil \Rightarrow ce_v{}^a = \underline{c}_v$	c estimates agree	(B3)
	$re_v{}^a \neq nil \Rightarrow re_v{}^a = \underline{r}_v$	r estimates agree	(B4)

We take AP's $anchor_v$ as a state function of BP also. Following (A9) with $re_v{}^a$ for r_v and without the c_u term, we define:

$out_{u,v}{}^a = (\forall w \mid u < w < v \Rightarrow re_w{}^a = out)$

stateF $anchor_v{}^a = anchor_u \cap \{x \mid Q_{out}\#RO_u^{\,x}@a\}$ if $out_{u,v}{}^a$ (B5)

The missing a on $anchor_u$ is not a misprint. We have the obvious

invariant $anchor_v{}^a \subseteq anchor_v$ (B6)

There is still a role for a primary, however: to propose a choice to the agents. This is essential for liveness, since if the agents can't get a quorum for some choice, the view can't proceed. BP is thus roughly a merger of AP's agents and CP's primary. As in CP, the primary is usually an agent too, but we describe it separately.

Safety cannot depend on the primary, since it may be faulty and propose different choices to different agents. If there's no quorum for any choice, the view never does *Accept* and BP advances to the next view as discussed in section 8.4.

The primary has a persistent stable $c_v{}^p$; a volatile c^p like CP's won't do, because p mustn't propose different choices for the same view. The primary needs $input^p$ to choose, but it doesn't need v^p, since it just works on the last anchored view.

var	$c_v{}^p$	:	$Y := nil$	Primary's Choice
	$input^p$	:	**set** X := {}	

An annoying complication is that when the primary chooses $c_v{}^p$, it needs to be able to broadcast $c_v{}^p \in anchor_v$ so that all the agents will go along with it. To broadcast, p needs $Q_{\sim F}{}^+\#(c_v{}^p \in anchor_v)$ (see the discussion of broadcast at the end of section 6), so a value that's anchored at the primary had better be anchored at enough agents, because $anchor_v{}^a$ is their only approximation of $anchor_v$. Like $Inp^{x,a}$, we define $Anc_v^{\,x,a} = x \in anchor_v{}^a$. Then

stateF $anchor_v^p = \boxed{\{x \mid Q_{\sim F}{}^+ \# Anc_v^x @p\}}$ (B7)

Thus to compute $anchor_v^p$, p needs to hear from $Q_{\sim F}{}^+$ agents.

8.1 The algorithm

The agents' actions are essentially the same as in AP; (B3-B4) imply that the guards are stronger and the state change is the same.

$\boldsymbol{Close_v^a}$	$active_v^a$	$\rightarrow$**for all** $u < v$ **do** **if** $r_u^a = nil$ **then** $r_u^a := out$
$\boldsymbol{Anchor_v^a}$	$anchor_v^a \neq \{\}$	none
$\boldsymbol{Accept_v^a}$	$\boxed{ce_v^a} \neq nil \wedge\ r_v^a = nil$	$\rightarrow r_v^a := \boxed{ce_v^a}$; $Close_v^a$
$\boldsymbol{Finish_v^a}$	$re_v^a \in X$	$\rightarrow d^a := re_v^a$

The primary does *Input* and *Anchor* as in CP, though the definition of $anchor_v^p$ is quite different.

$\boldsymbol{Input_v^p(x)}$		$input^p := input^p \cup \{x\}$
$\boldsymbol{Anchor_v^p}$	$anchor_v^p \neq \{\}$	$\rightarrow$none

Choose is like AP's *Choose*, but at both agents and primary:

> The primary chooses for a view that belongs to it and is anchored, but where it hasn't chosen already.
>
> An agent only chooses the primary's apparent choice.

(B5)-(B6) mean that the agents' guards are stronger than in AP; this is what matters, since c_v^a is what's in the abstraction to $\underline{c_v}$.

$\boldsymbol{Choose_v^p}$	$\boxed{p_v = p \wedge}\ c_v^p = nil$	$\rightarrow c_v^p := x$
	$\wedge\, x \in input^p \cap anchor_v^p$	
$\boldsymbol{Choose_v^a}$	$c_v^a = nil$	$\rightarrow c_v^a := x$
	$\wedge\, x \in input^a \cap anchor_v^a$	
	$\boxed{\wedge\, x = c_v^{p_v} @ p_v * @ a}$	

There's no guarantee that c_v^p is in $\underline{input}$, but this wouldn't be strong enough anyway, since for liveness it must be in $input^a$ for a choice quorum.

invariant $c_v^p \neq nil \Rightarrow c_v^p \in input^p \cap anchor_v^p$ (B8)

$c_v^a \neq nil \Rightarrow c_v^a \in \underline{input} \cap anchor_v^a$ (B9)

A client must hear d^a from a good quorum of agents.

For safety, in addition to AP's assumption that Q_{dec} and Q_{out} are exclusive, Q_{ch} must exclude itself. Then the invariants (A2-A6) of AP hold in BP, and the $Close^a$, $Accept^a$, and $Finish^a$ actions of BP simulate the same actions in AP. All the other actions simulate **skip** except the $Choose^a$ action that forms a quorum, which simulates AP's *Choose*.

Figure 3 shows the flow of messages in BP. This is the logical flow. In practice the client-agents network is often slower than the inter-agent network, so the client sends an input just to the primary, including message authenticators for all the agents, and the primary forwards the input to the agents. This does not change any costs or affect cryptographic authentication of messages. It does mean that the client may have to resend to another primary if this one turns out to be faulty.

8.2 Communicating with agents

In BP the primary's only job is to propose c_v^p to the agents, who are responsible for everything else including scheduling, since they can't count on the possibly faulty primary. So *Transmit* and *Broadcast* are all there is to say about communication.

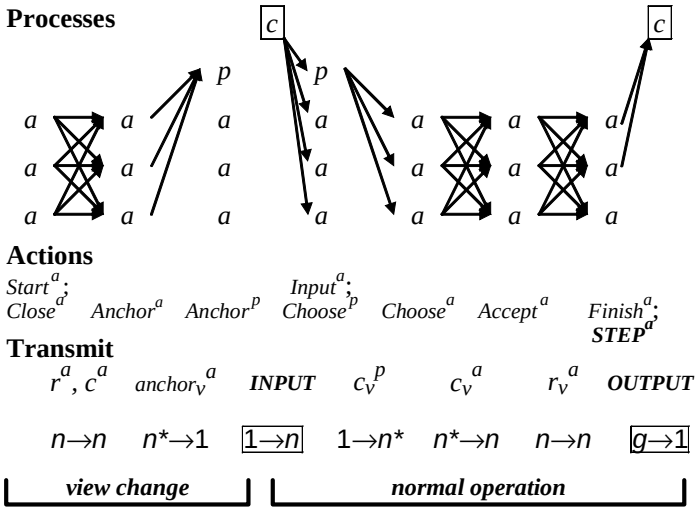

Figure 3: Byzantine Paxos

8.3 Liveness

We want to show that a view with an OK primary will produce a decision unless a later view starts. We assume that Q_{dec}, Q_{out}, and Q_{ch} are live, and $Q_{ch} \subseteq Q_{\sim F}{}^+$; if these don't hold BP is still safe, but it may not decide. Suppose initially that $anchor_v^a \supseteq anchor_v^p \neq \{\}$ at a choice quorum of OK agents; this is the case after a view change. Then we have normal operation, which is the easy part of the liveness argument:

> Since $anchor_v^p \neq \{\}$, an OK primary can do $Choose_v^p$, which leads to $c_v^p @ a$ at all the agents, so that they can all do $Choose_v^a$ by (B7) provided $c_v^p \in input^a$, that is, a knows the client really sent c_v^p. An OK client will send its input to all the agents, but a faulty client may fail to do so.
>
> This leads to an OK choice quorum for c_v^p because Q_{ch} is live, which leads to knowing that quorum at all the agents so that they can all do $Accept^a$.
>
> This leads to an OK decision quorum for $r_v^a = c_v^p$ because Q_{dec} is live, which leads to knowing that quorum at all the agents, so that they can all do $Finish^a$.
>
> This leads to knowing a good quorum for $d^a = c_v^p$ (or alternatively for the output) at the client, because $Q_{\sim F}$ is live.

Now we consider what happens in a view change. For an OK agent a to get $anchor_v^a \neq \{\}$, it needs $x \in anchor_u^a$ and $Q_{out} \# RO_u^x @ a$ for some $u < v$, and $re_w^a = out$ for each $u < w < v$, from (B5). This is the tricky part, since these require quorums, and in general having a quorum is no guarantee that it's visible.

As we saw in section 4.5, in computing re_w^a an agent a will eventually hear non-*nil* $r_w^{a'}$ from a quorum $q \in Q_{out}$ of OK agents, since *Close* makes $r_w^{a'} \neq nil$ and Q_{out} is live. This is not as wonderful as it looks, however, since a can't tell when an agent is OK and therefore can't tell when it has seen such a quorum. If q includes more than one value from X, however, then some of the agents providing those values must be faulty, since an OK agent a' has $r_w^{a'} = c_w$. That means that there are OK agents still to be heard from, and a waits for them.

Eventually we will have $(Q_{out} \# RO_w^x) @ a$ from a quorum q of OK agents. If the agents in q all say *out*, then $re_w^a = out$ whether or not they are all OK. If one of them says x, then since it is OK we must have $x = c_w$. If we could conclude this, we would be done. Unfor-

tunately, *a* can't tell which quorum is OK, or even how long it has to wait to see such a quorum, so we need a less direct path.

If $x = c_w$ then $Q_{ch}\#C_w^x$, and hence $Q_{ch}\#(\lambda x \mid x \in anchor_w)$ by (A5). Since $Q_{ch} \subseteq Q\sim_F^+$, $x \in anchor_w$ can be broadcast, so eventually we have $(x \in anchor_w)@a$ as a result of $(Q_{\sim F}\#C_w^x)@a$, as required. Note that *a* doesn't know $x = c_w$, and in fact c_w might be *nil*; this is a more delicate argument that we used for CP.

To sum this up, since *a* eventually hears from a Q_{out} of OK agents, it hears, for some view *u* and each *w* between *u* and *v*,

an out quorum for *w* and

$Q_{out}\#RO_u^x@a$ and $(x \in anchor_u)@a$,

and this is all we need to anchor *v*, by (B5).

All that remains is the liveness of $Anchor_v^p$: the primary must see a non-empty intersection of $anchor_v^a$ sets from a $Q\sim_F^+$ quorum, but such a quorum is live, and eventually every agent in it will hear from the *same* $q \in Q_{out}$ of OK agents, and either *w* will be out or $x \in anchor_w$ will be broadcast.

Thus BP is live, except for faulty clients, although it's hanging on by its fingernails.

If the client is faulty, it can fail to deliver inputs to some agents. A view change that broadcasts $x \in anchor_w$ can broadcast $x \in input$ as well and override the client's failings. This is essential, since there might be a decision for *x*. During normal operation, however, a faulty client can cause a view change if the primary chooses an input that the client did not send to Q_{ch} OK agents.

Agents can keep track of such clients and refuse to accept more input from them. If there are lots of them mounting a denial of service attack, however, performance can still be significantly affected. I don't know any way to prevent this except for the primary to insist that each input be broadcast by getting an ack from $Q\sim_F^+$ agents, or by public key as in section 8.6. This is expensive, and it happens in normal operation, not just in a view change.

8.4 Scheduling

A faulty primary cannot keep BP from satisfying its safety spec, but it can certainly prevent progress. We therefore need a way to ensure that there are times when there's only a non-faulty primary. To do this, we let the agents become primary in round-robin order. That is, we use integers as views and take a view's primary to be the view modulo *n*: $p_v = v$ **mod** *n*.

An agent *a* keeps an estimate *PT* of the time to process a client input. If *a* gets input from a client at time *t* and doesn't see some decision by *t* + *PT*, *a* assumes that the primary has failed. It advances to the next view *v*, does $Close_v^a$, and multicasts its state in a $Close_v$ message. Other agents' timers expire, they do the same thing, and when *a* sees enough $Close_v$ messages it does $Anchor_v^a$ and sends its $anchor_v^a$ set to the new primary p_v; see section 8.3.

If *a* gets $Close_u$ messages for various $u > v$ from a good quorum, it changes its *v* to the smallest *u* and does $Close_u$. Thus the OK agents increase *v* at most *n* times before they agree on the next view, and faulty agents can't disrupt this agreement. Like *Anc*, we define $Act_v^a = active_v^a$.

***Start*$_v^a$** *v*–1 too slow $\vee$ $Q_{\sim F}\#Act_v@a$ $\rightarrow active_v^a := true$

BP uses the same exponential backoff as AP to adjust *PT*.

8.5 Cleanup

This is similar to AP, but there is a lot more agent state. As in CP, the primary can discard its state at any time, and the extra transmits for *Cleanup* can be piggy-backed on the next step.

***Cleanup*^{a}** $Q_{\sim F}^+\#D^x$ $\rightarrow r_v^a := nil;\ in^a := \{\};$ $c_v^a := nil;\ active_v^a := false$

8.6 Public key BP

As we saw in section 6, if messages can be broadcast securely, that is, signed by public keys, then a process can forward information to other processes so that they don't have to get it from the source. This does not add any new power, but it avoids the $Q_{\sim F}^+$ acknowledgements otherwise needed for a broadcast.

There are two points where BP needs a broadcast, of $input^a$ in normal operation and of $anchor^a$ in a view change:

1. The primary can broadcast an input to all the agents, so a faulty client cannot force a view change.
2. During a view change the $Anchor_v^p$ action is not needed. That is, an agent does not need to acknowledge $x \in anchor_v^a$ to the primary, since the information on which $anchor_v^a$ is based is broadcast.

This does not reduce the amount of message traffic in the normal case, since we are cheating there by not broadcasting *input* and taking some risk from faulty clients. Thus there is no performance gain to balance the large loss from doing public key operations, except when there are lots of faulty clients.

8.7 Performance using multicast

Figure 3 shows that in the normal case there is one client-agents round trip (*g* is a good quorum), by comparison with a client-primary round-trip in CP. In addition, there is one $1 \rightarrow n$ message from the primary as in CP, and two $n \rightarrow n$ messages among the agents, compared with one $n \rightarrow 1$ message to the primary in CP, and one $1 \rightarrow n$ message from the primary that can go in parallel with output. Thus BP has one extra message latency before the client gets output. What about throughput?

In a network that supports multicast efficiently (for example, any broadcast LAN or a switched LAN whose switches support it), the extra cost for *n* receivers is small. Table 1 shows the cost comparison on this assumption. BP is about twice as expensive as CP, or almost three times as expensive for the same number of failures (*f* or *s*). It's not surprising that faults are much more costly.

If there's no efficient multicast, agents can relay their messages to other agents through the primary, complete with authenticators, so that there are $2n$ messages after $Choose^a$ or $Accept^a$ rather than n^2.

Table 1: Cost of a normal run of BP and CP

Enables	Message flow	BP	cost	CP	cost
$Input^p$	client⟶agents/primary	$1 \rightarrow n$	1	$1 \rightarrow 1$	1
output	agents/primary⟶client	$g \rightarrow 1$	$f + 1$	$1 \rightarrow 1$	1
Total external			$f + 2$		2
$Choose^a$	primary⟶agents	$1 \rightarrow n\text{-}1$	1		
Accept	agents/primary⟶agents	$n\text{-}1 \rightarrow n$	n-1	$1 \rightarrow n\text{-}1$	1
Finish	agents⟶agents/primary	$n \rightarrow n$	n	$n\text{-}1 \rightarrow 1$	n-1
$Finish^a$	primary⟶agents (piggy-backed)			$1 \rightarrow n\text{-}1$	0
Total internal			$2n$ $\geq 6f + 2$		n $\geq 2s + 1$
Smallest non-trivial *n*			$f = 1$ $n = 4$		$s = 1$ $n = 3$
Total for this *n*			8		3

We separate the client-Paxos costs from the internal costs. They are not really comparable, for two reasons:

They often involve a network with very different properties.

Internal traffic can often have much bigger batches since it can combine the traffic from all the clients.

8.8 Optimizations

The optimizations of AP work in BP: compressing state with the *last*-triple, using one view change for many steps, and batching.

BP does not have to transmit the client's entire input in each message. It's sometimes enough to just send an 'authenticator', a signature of the message implemented by hashing it with a key shared between sender and receiver.

An undesirable property of BP's view change is that the agents must remember all their $c_w^{\,a}$ values, since they don't know which one might be needed. This means that the *last*-triple optimization is not enough to avoid storage linear in the number of views. To avoid this, notice that if $x \in anchor_v^{\,p}$ then $x \in anchor_v$ is broadcast by (B7), so if agent a is the primary for v then a can discard $c_w^{\,a}$ for all $w < v$, since these are only needed for finding an element of $anchor_w$, and $anchor_w \supseteq anchor_v$. For this to work, each agent a' must remember its contribution to $x \in anchor_v$. If $anchor_v^{\,a'} = \{c_v^{\,a'}\}$, remembering $c_v^{\,a'}$ is enough. If $anchor_v^{\,a'} = X$, then a' must remember that; this is a new requirement. An agent must remember at most n values of $c_w^{\,a}$ or $anchor_w^{\,a} = X$ before its turn as primary comes along. If agents don't act as primaries, then they need to collect the $Anc_v^{\,x,a}$ facts themselves at regular intervals.

It's unfortunate that the primary has a persistent $c_v^{\,p}$. If it's also an agent, then this can be the agent's $c_v^{\,a}$, so the only cost is that it must be persisted before it's sent to any other agent. To get a primary with no persistent state, follow the model of CP: introduce a volatile c^p, make $c_v^{\,p}$ a history variable, and maintain invariants corresponding to (C1) and (C8) as in section 7.3:

invariant $c^p \neq nil \Rightarrow c^p = c_v{}^p$ (B10)

$Q_{dec} \# R_u^{\,nil} \wedge v > u \Rightarrow c_v^{\,p} = nil$ (B11)

To do $Choose_v^{\,p}$ the primary must establish $c_v^{\,p} = nil$ using (B11). This may require a new view; to preserve the round-robin scheduling of primaries, make a V a pair (i, j), where i determines the primary ($p_{(i,j)} = i$ **mod** n) and p can use j to start another view.

9 Conclusion

We started with an abstract Paxos algorithm AP that uses n agents and has only the agent actions *Close*, *Accept*, and *Finish* and an abstract *Choose* (plus the external actions *Input* and *Decision*). AP works by running a sequence of views until there's one that runs for long enough to make a visible decision quorum for some input. Provided no later view starts, this will always happen as long as $Choose_v$ happens. AP's operation is divided into view change and normal operation; the latter requires one round-trip of agent-agent communication. AP can do any number of successive decisions with a single view change plus one normal operation per decision. AP's agents are memories that can do conditional writes, but DP is a generalization that works with read-write memories.

AP can't be implemented directly because it has actions that touch state at more than one process, in particular the $Choose_v$ action. We showed two implementations in which the processes communicate stable predicates about their state that are strong enough to convey all the information that AP's actions need. Both CP and BP have essentially the same agent actions as AP. Both implement AP's *Anchor* and *Choose* actions in a primary process that is logically separate, though it practice it is combined with an agent unless the agents are disks.

CP also uses the primary to relay information among the agents. It doesn't tolerate any faults. It needs Q_{out} and Q_{dec} exclusive for safety, and live for liveness. For size-based quorums we have $f = 0$, $s < n/2$ and $Q_{out} = Q_{dec} = Q_{\geq s+1}$. In normal operation there are n internal messages if a multicast counts as 1, and the client latency is one client-primary round-trip plus one primary-agent round trip.

BP does tolerate faults, so it needs *Anchor* and *Choose* actions at both agents and primary, and uses multicast to share information among agents. In addition to CP's requirements on quorums, it also needs Q_{ch} exclusive with itself for safety, and Q_{ch} live and $Q_{ch} \subseteq Q\sim_F{}^+$ for liveness. For size-based quorums and $F \Rightarrow S$ we have $Q\sim_F = Q_{\geq f+1}$ and $Q\sim_F{}^+ = Q_{out} = Q_{dec} = Q_{ch} = Q_{\geq 2f+1}$. In normal operation there are $2n$ internal messages, and CP's primary-agent round-trip is replaced by a primary-agent multicast plus an agent-agent round trip.

The main application for Paxos is replicated state machines.

References

[1] Castro, M. and Liskov, B. Practical Byzantine fault tolerance. *Proc. 3rd OSDI*, New Orleans, Feb. 1999.

[2] Castro, M. and Liskov, B. Proactive recovery in a Byzantine-fault-tolerant system. *Proc. 4th OSDI*, San Diego, Oct. 2000.

[3] De Prisco, R., Lampson, B., and Lynch, N. Revisiting the Paxos algorithm. *Proc. WDAG'97*, LNCS **1320**, Springer, 1997, 111-125.

[4] Fischer, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one faulty process. *J. ACM* **32**, 2, April 1985.

[5] Gafni, E. and Lamport, L. Disk Paxos. *Proc. DISC 2000*, LNCS **1914**, Springer, 2000, 330-344.

[6] Gray, J. and Reuter, A. *Transaction Processing*: *Concepts and Techniques*. Morgan Kaufmann, 1993.

[7] Lamport, L. Time, clocks and the ordering of events in a distributed system, *Comm. ACM* **21**, 7, July 1978, 558-565.

[8] Lamport, L**.** A simple approach to specifying concurrent systems. *Comm. ACM* **32**, 1, Jan. 1989, 32-45.

[9] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems* **16**, 2, May 1998, 133-169. Originally appeared as Research Report 49, Digital Systems Research Center, Palo Alto CA, Sep. 1989.

[10] Lampson, B., Lynch, N., and Søgaard-Andersen, J. Correctness of at-most-once message delivery protocols. *Proc. 6th Conf. on Formal Description Techniques*, Boston, 1993, 387-402.

[11] Lampson, B. Reliable messages and connection establishment. In *Distributed Systems*, ed. S. Mullender, 2nd ed., Addison-Wesley, 1993, 251-281.

[12] Lampson, B. How to build a highly available system using consensus. In *Distributed Algorithms*, ed. Babaoglu and Marzullo, LNCS **1151**, Springer, 1996, 1-17.

[13] Liskov, B. and Oki, B. Viewstamped replication, *Proc. 7th PODC*, Aug. 1988.

[14] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[15] Malkhi, D. and Reiter, M. Byzantine quorum systems. In *Proc. 29th ACM STOC*, El Paso, Texas, May 1997, 569–578.

Appendix

Table 2 gives some correspondences between the terminology of this paper and that of Castro and Liskov.

Table 3 lists all the names for variables and constants in alphabetical order, followed by the @, #, *, and Q^+ notation and the names of the actions for communication.

Table 4 collects the variables, abstractions, state functions, actions, and invariants of AP, CP, and BP to help you see how they are related. To save space, we shorten the names of actions to two characters, and shorten *input*, *active*, and *anchor* to *in*, *act*, and *anc*.

The external actions are first, then the internal ones in the order of a complete run.

Changes from the item to the left are marked by boxes except for *p* and *a* superscripts. A ditto mark " means that the entry is a copy of the corresponding entry to the left.

The legend in the lower left corner summarizes the way we mark non-local, changed, and abstract variables. We mark as non-local anything in an action that came from other processes, even though in CP and BP it is of course local when the action occurs.

Table 2: Our terminology for BP vs. Castro and Liskov's

Action⟶ C-L state		**Our state**	**C-L msg**	**Our msg**
$Close^p$			view-change	r^a, c^a
$Anchor_v^a$	in view *v*	$anchor_v^a \neq \{\}$	view-ack	$anchor_v^a$
$Anchor_v^p$	in view *v*	$anchor_v^p \neq \{\}$	new-view	
$Choose^p$	pre-prepared	$c_v^p \neq nil$	pre-prepare	c_v^p
$Choose^a$	pre-prepared	$c_v^a \neq nil$	prepare	c_v^a
Accept	prepared	$r_v^a \neq nil$	commit	r_v^a
Finish	committed	$d^a \neq nil$		

$Q_{\sim F}$	weak certificate
$Q_{\sim F}^+$	quorum certificate

Table 3: Variables, constants, notation, and communication

		Spec, failure, quorum	**AP**	**DP** Δ *from AP*	**CP** Δ *from AP*	**BP** Δ *from CP*
in section		§ 2, 3	§ 4	§ 5	§ 7	§ 8
Agent	*a*		*a*			
Choice	*c*		c_v		c^p	c_v^a, ce_v^a, c_v^p
Decision	*d*	*d*	d^a			
Faulty predicate	*f*, *G*	f, F^m				
Integer process	*i*, *j* *k*, *m*					
|*A*|		*n*				
Primary	*p*				p, p_v	
Quorum	*Q*, *q*	$Q_{\sim F}$, Q^+	Q_{dec}, Q_{out}			Q_{ch}
Result	*r*		r_v^a, r_v	rx_v^a, ro_v^a	re_v^p	re_v^a, ~~re_v^p~~
Stopped		*s*, S^m				
Truth	*T*	*T* (§6)				
View	*u*, *v*, *w*		*v*		v^p	~~v^p~~
value	*x*, *y*					
failures	*Z*, *z*	Z_F, Z_S, Z_{FS}				

g@m	$T^m \Rightarrow g$	**Communication**
*g@m**	$g@m \vee F^m$	
G@m	$(\lambda\, k \mid G^k@m)$	$\boldsymbol{Local^k(g)}$
Q#G	$\{m \mid G^m \vee F^m\} \in Q$	$\boldsymbol{Transmit^{k,m}(g)}$
Q#g	$Q\#(\lambda\, m \mid g@m)$	$\boldsymbol{TransmitF^{k,m}(g)}$
Q^+	$\{q' \mid (\forall z \in Z_{FS} \mid q' - z \in Q\}$	$\boldsymbol{Broadcast^m(g)}$
$Q_{\sim F}$	$\{q \mid q \notin Z_F\}$	

Figure 4 collects from figures 1-3 the pictures for the flow of actions and messages in AP, CP, and BP. Notice the fact that they start slightly differently, the extra *Choose* action in BP, and the extra $Finish^a$ action in CP.

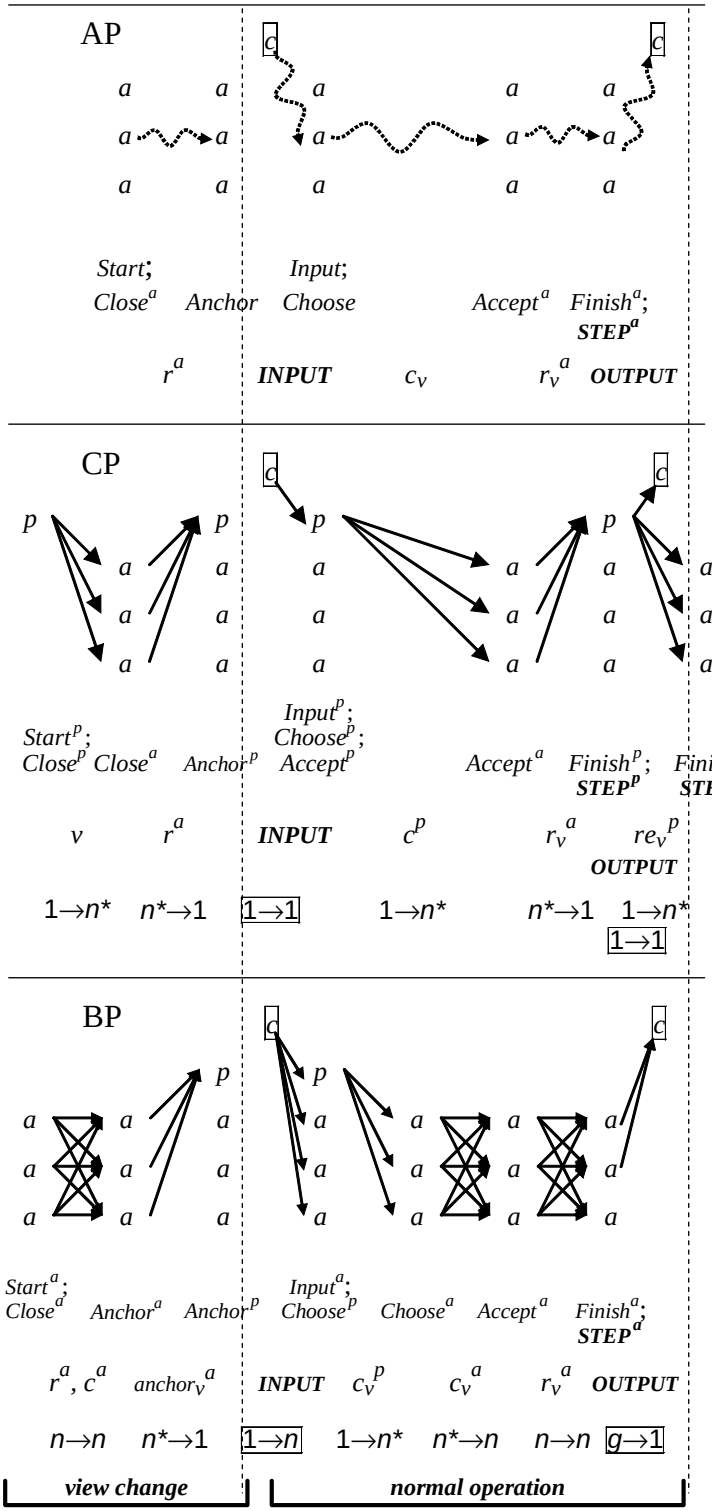

Figure 4: Summary of actions

Table 4: Summary of declarations, actions, and invariants

	AP	implements spec	***CP***	implements AP	***BP***	implements AP
var	r_v^a, d^a	result, decision	r_v^a, d^a	$= \underline{r_v}^a, \underline{d}^a$	r_v^a, d^a	$= \underline{r_v}^a, \underline{d}^a$
	c_v	choice	c_v	*history*, $= \underline{c_v}$	c_v^a	
			v^p, c^p	view, choice	c_v^p	
			in^p			
input	in		in	*history*, $= \underline{in}$	in^a	
active	act_v		act_v	*history*, $= \underline{act_v}$	act_v^a	
abstract	$\underline{d}$ =	**if** $r_v \in X$ **then** r_v **else** *nil*				
	$\underline{in}$ =	*in*				
					$\underline{in}$ =	$\bigcup_{a \in A} input^a$
					$\underline{c_v}$ =	**if** $Q_{ch}\#C_v^x$ **then** x **else** *nil*
					$\underline{act_v}$ =	$(\exists a \mid act_v^a)$
stateF			act^p =	$(v^p \neq v_0)$	ce_v^a =	**if** $(Q_{ch}\#C_v^x)$@a **then** x **else** *nil* (B1)
r_v =	**if** $Q_{dec}\#R_v^x$ **then** x (A1)		re_v^p =	**if** $(Q_{dec}\#R_v^x)$@p **then** x (C3)	re_v^a =	**if** $(Q_{dec}\#R_v^x)$@a **then** x (B2)
	elseif $Q_{out}\#R_v^{out}$ **then** *out*			**elseif** $(Q_{out}\#R_v^{out})$@p **then** *out*		**elseif** $(Q_{out}\#R_v^{out})$@a **then** *out*
	else *nil*			**else** *nil*		**else** *nil*
$anchor_v$ =	$\{x \mid (\forall u<v \mid c_u = x \vee Q_{out}\#RO_u^x)\}$ (A8)				" =	"
$anchor_v$ =	$anc_u \cap \{x \mid Q_{out}\#RO_u^x\}$ if $out_{u,v}$ (A9)				anc_v^a =	$anc_u \cap \{x \mid Q_{out}\#RO_u^x$@$a\}$ if $out_{u,v}^a$ (B5)
$anchor_v \supseteq$	**if** $out_{u,v} \wedge r_u^a = x$ **then** $\{x\}$ (A10)		$anc^p \supseteq$	**if** $out_{u,v}^p \wedge (r_u^a = x)$@$p$ **then** $\{x\}$ (C4)		
	elseif $out_{v_0,v}$ **then** X **else** $\{\}$			**elseif** $out_{v_0,v}^p$ **then** X **else** $\{\}$		
					anc_v^p =	$\{x \mid Q_{\sim F}^+\#Anc_v^x$@$p\}$ (B7)
$out_{u,v}$ =	$(\forall w \mid u<w<v \Rightarrow r_w = out)$		$out_{u,v}^p$ =	$(\forall w \mid u<w<v \Rightarrow re_w^p = out)$	$out_{u,v}^a$ =	$(\forall w \mid u<w<v \Rightarrow re_w^a = out)$
$R_v^{y,a}$ =	$r_v^a = y$	$D^{x,a} = d^a = x$	" =	"	" =	" $Inp^{x,a} = x \in in^a$
$RO_v^{x,a}$	$r_v^a = x \vee r_v^a = out$				" =	" $Anc_v^{x,a} = x \in anc_v^a$
					$C_v^{y,a}$	$= c_v^a = y$ $Act_v^a = act_v^a$
Actions						
Name	**Guard**	**State change**	**Name**	**Guard**	**State change**	**Name**	**Guard**	**State change**
***Input*(x)**		$in := in \cup \{x\}$	In^p		$in^p := in^p \cup \{x\}$; $in := in \cup \{x\}$	In^a		$in^a := in^a \cup \{x\}$
						In^p		$in^p := in^p \cup \{x\}$
***Decision*^{a}**	$d^a \neq nil$	$\rightarrow$**ret** d^a	"	"		"	"	
***Start*$_v$**	$u<v$ too slow	$\rightarrow act_v := true$	St_v^p	$u < v$ too slow $\wedge p_v = p \wedge c_v = nil$	$\rightarrow v^p := v$; $c^p := nil$; $act_{vp} := true$	St_v^a	v–1 too slow $\vee Q_{\sim F}\#Act_v$@a	$\rightarrow act_v^a := true$
***Close*$_v^a$**	act_v	$\rightarrow$**for all** $u<v$ **do** **if** $r_u^a = nil$ **then** $r_u^a := out$	"	"		"	act_v^a	"
***Anchor*$_v$**	$anc_v \neq \{\}$	$\rightarrow$none	An^p	$anc^p \neq \{\}$	$\rightarrow$none	An_v^a	$anc_v^a \neq \{\}$	$\rightarrow$none
						An_v^p	$anc_v^p \neq \{\}$	$\rightarrow$none
***Choose*$_v$**	$c_v = nil \wedge x \in in \cap anc_v$	$\rightarrow c_v := x$	Ch^p	$act^p \wedge c^p = nil \wedge x \in in^p \cap anc^p$	$\rightarrow c^p := x$; $c_{vp} := x$	Ch_v^p	$p_v = p \wedge c_v^p = nil \wedge x \in in^p \cap anc_v^p$	$\rightarrow c_v^p := x$
						Ch_v^a	$c_v^a = nil \wedge x \in in^a \cap anc_v^a \wedge x = c_v^{p_v}$@$p_v$*@$a$	$\rightarrow c_v^a := x$
***Accept*$_v^a$**	$c_v \neq nil \wedge r_v^a = nil$	$\rightarrow r_v^a := c_v$; $Close_v^a$	"	c_v@$a \neq nil \wedge r_v^a = nil$	$\rightarrow r_v^a := c_v$@a; $Close_v^a$	"	$ce_v^a \neq nil \wedge r_v^a = nil$	$\rightarrow r_v^a := ce_v^a$; $Close_v^a$
***Finish*$_v^a$**	$r_v \in X$	$\rightarrow d^a := r_v$	"	re_v^p@$a \in X$	$\rightarrow d^a := re_v^p$@$a$	"	$re_v^a \in X$	$\rightarrow d^a := re_v^a$
***Cleanup*^{a}**	$Q_{\sim F}^+\#D^x$	$\rightarrow r_v^a := nil$; $in := \{\}$	"	"		"	"	$\rightarrow r_v^a := nil$; …
invariant	$d^a \neq nil \Rightarrow (\exists v \mid r_v = d^a)$ (A2)			"			"	
	$r_v = x \wedge r_u = x' \Rightarrow x = x'$ (A3)			"			"	
	$r_v^a = x \Rightarrow r_v^a = c_v$ (A4)			"			"	
	$c_v = x \Rightarrow c_v \in in \cap anc_v$ (A5)			"			"	
	$r_v^a \neq nil \wedge u < v \Rightarrow r_u^a \neq nil$ (A6)			"			"	
				$act^p \wedge c^p \neq nil \Rightarrow c^p = c_{vp}$ (C1)			$ce_v^a \neq nil \Rightarrow ce_v^a = c_v$ (B3)	
				$re_v^p \neq nil \Rightarrow re_v^p = r_v$ (C6)			$re_v^a \neq nil \Rightarrow re_v^a = r_v$ (B4)	
				$anc^p \subseteq anc_{vp}$ (C7)			$anc_v^a \subseteq anc_v$ (B6)	
				$Q_{dec}\#R_u^{nil} \wedge v>u \Rightarrow c_v = nil$ (C8)			$Q_{dec}\#R_u^{nil} \wedge v>u \Rightarrow c_v^p = nil$ (B11)	
				$act^p \Rightarrow act_{vp}$; $in^p \subseteq in$ (C2)			$c_v^p \neq nil \Rightarrow c_v^p \in in^p \cap anc_v^p$ (B8)	
				$act^p \wedge c^p = nil \Rightarrow c_{vp} = nil$ (C5)			$c_v^a \neq nil \Rightarrow c_v^a \in in^a \cap anc_v^a$ (B9)	

Legend

anc_v	non-local
$\underline{in}$	abstract variable
$act^p \wedge$	changed from item on left
"	copy of item on left