MODULE *Paxos*

This is a high-level specification of the *Paxos* consensus algorithm. It refines the spec in module *Voting*, which you should read before reading this module. In the *Paxos* consensus algorithm, acceptors communicate by sending messages. There are additional processes called leaders. The specification here essentially considers there to be a separate leader for each ballot number. We can consider "leader" to be a role, where in an implementation there will be a finite number of leader processes each of which plays infinitely many of these leader roles.

Note: The algorithm described here is the *Paxos* consensus algorithm. It is the crucial component of the *Paxos* algorithm, which implements a fault-tolerant state machine using a sequence of instances of the consensus algorithm. The *Paxos* algorithm is sometimes called *MultiPaxos*, with the *Paxos* consensus algorithm being incorrectly called the *Paxos* algorithm. I'm afraid I have contributed to this confusion by being lazy and calling the module "*Paxos*" instead of "*PaxosConsensus*". But incarnations of this module have already appeared, so I'm reluctant to change its name now.

EXTENDS *Integers*

The constants and the assumptions about them are the same as for the Voting algorithm. However, the second conjunct of the assumption, which asserts that any two quorums have a non-empty intersection, is not needed for the *Paxos* consensus algorithm to implement the *Voting* algorithm. The *Voting* algorithm, and it, do not satisfy consensus without that assumption.

CONSTANTS *Value*, *Acceptor*, *Quorum*

$$\text{ASSUME} \quad \begin{aligned} &\land \forall Q \in Quorum : Q \subseteq Acceptor \\ &\land \forall Q1, Q2 \in Quorum : Q1 \cap Q2 \neq \{\} \end{aligned}$$

$$Ballot \triangleq Nat$$

$$None \triangleq \text{CHOOSE } v : v \notin Ballot$$

This defines *None* to be an unspecified value that is not a ballot number.

We now define *Message* toe be the set of all possible messages that can be sent in the algorithm. In TLA+, the expression

(1) $[type \mapsto \text{"1a"}, bal \mapsto b]$

is a record r with two components, a *type* component, $r.type$, that equals "1a" and whose *bal* component, $r.bal$, that equals b. The expression

(2) $[\text{type} : \{\text{"1a"}\}, bal : Ballot]$

is the set of all records r with a components *type* and *bal* such that $r.type$ is an element of $\{\text{"1a"}\}$ and $r.bal$ is an element of *Ballot*. Since "1a" is the only element of $\{\text{"1a"}\}$, formula (2) is the set of all elements (1) such that $b \in Ballot$.

The function of each type of message in the set *Message* is explained below with the action that can send it.

$$\begin{aligned} Message \triangleq \\ &\quad [type : \{\text{"1a"}\}, bal : Ballot] \\ &\cup [type : \{\text{"1b"}\}, acc : Acceptor, bal : Ballot, \\ &\qquad mbal : Ballot \cup \{-1\}, mval : Value \cup \{None\}] \\ &\cup [type : \{\text{"2a"}\}, bal : Ballot, val : Value] \\ &\cup [type : \{\text{"2b"}\}, acc : Acceptor, bal : Ballot, val : Value] \end{aligned}$$

We now declare the following variables:

maxBal – Is the same as the variable of that name in the *Voting* algorithm.

maxVBal
maxVal – As in the *Voting* algorithm, a vote is a $\langle ballot, value \rangle$ pair. The pair $\langle maxVBal[a], maxVal\{a] is\ the\ vote\ with$
$the\ largest\ ballot\ number\ cast\ by\ acceptor\ a.Tt\ equals$
$\langle -1, None \rangle if\ a\ has\ not\ cast\ any\ vote.$

$msgs - The\ set\ of\ all\ messages\ that\ have\ been\ sent.$

$Messages\ are\ added\ to\ msgs\ when\ they\ are\ sent\ and\ are\ never\ removed.$
$An\ operation\ that\ is\ performed\ upon\ receipt\ of\ a\ message\ is\ represented$
$by\ an\ action\ that\ is\ enabled\ when\ the\ message\ is\ in\ msgs. This$
$simplifies\ the\ spec\ in\ the\ following\ ways:$

$-\ A\ message\ can\ be\ broadcast\ to\ multiple\ recipients\ by\ just\ adding$
$(a\ single\ copy\ of)it\ to\ msgs.$

$-\ Never\ removing\ the\ message\ automatically\ allows\ the\ possibility\ of$
$the\ same\ message\ being\ received\ twice.$

$Since\ we\ are\ considering\ only\ safety,\ there\ is\ no\ need\ to\ explicitly$
$model\ message\ loss. The\ safety\ part\ of\ the\ spec\ says\ only\ what$
$messages\ may\ be\ received\ and\ does\ not\ assert\ that\ any\ message\ actually$
$is\ received. Thus,\ there\ is\ no\ difference\ between\ a\ lost\ message\ and$
$one\ that\ is\ never\ received.$

VARIABLES *maxBal*, *maxVBal*, *maxVal*, *msgs*

$vars \triangleq \langle maxBal, maxVBal, maxVal, msgs \rangle$

It's convenient to name the tuple of all variables in a spec.

The invariant that describes the "types" of the variables.

$$
\begin{aligned}
TypeOK \triangleq\ & \land maxBal \in [Acceptor \rightarrow Ballot \cup \{-1\}] \\
& \land maxVBal \in [Acceptor \rightarrow Ballot \cup \{-1\}] \\
& \land maxVal \in [Acceptor \rightarrow Value \cup \{None\}] \\
& \land msgs \subseteq Message
\end{aligned}
$$

The initial predicate should be obvious from the descriptions of the variables given above.

$$
\begin{aligned}
Init \triangleq\ & \land maxBal = [a \in Acceptor \mapsto -1] \\
& \land maxVBal = [a \in Acceptor \mapsto -1] \\
& \land maxVal = [a \in Acceptor \mapsto None] \\
& \land msgs = \{\}
\end{aligned}
$$

We now define the subactions of the next-state actions. We begin by defining an action that will be used in those subactions. The action *Send*(*m*) asserts that message *m* is added to the set *msgs*.

$Send(m) \triangleq msgs' = msgs \cup \{m\}$

The ballot b leader can perform actions $Phase1a(b)$ and $Phase2a(b)$. In the $Phase1a(b)$ action, it sends to all acceptors a phase $1a$ message (a message m with $m.type =$ "1a") that begins ballot b. Remember that the same process can perform the role of leader for many different ballot numbers b. In practice, it will stop playing the role of leader of ballot b when it begins a higher-numbered ballot. (Remember the definition of $[type \mapsto \text{"1a"}, bal \mapsto b]$ from the comment preceding the definition of *Message*.)

$$
\begin{aligned}
Phase1a(b) \triangleq\ & \land Send([type \mapsto \text{"1a"}, bal \mapsto b]) \\
& \land \text{UNCHANGED } \langle maxBal, maxVBal, maxVal \rangle
\end{aligned}
$$

Note that there is no enabling condition to prevent sending the phase $1a$ message a second time. Since messages are never removed from msg, performing the action a second time leaves msg and all the other spec variables unchanged, so it's a stuttering step. Since stuttering steps are always allowed, there's no reason to try to prevent them.

Upon receipt of a ballot b phase $1a$ message, acceptor a can perform a $Phase1b(a)$ action only if $b > maxBal[a]$. The action sets $maxBal[a]$ to b and sends a phase $1b$ message to the leader containing the values of $maxVBal[a]$ and $maxVal[a]$. This action implements the $IncreaseMaxBal(a, b)$ action of the *Voting* algorithm for $b = m.bal$.

$$
\begin{aligned}
& Phase1b(a) \triangleq \\
& \quad \land \exists m \in msgs : \\
& \qquad \land m.type = \text{"1a"} \\
& \qquad \land m.bal > maxBal[a] \\
& \qquad \land maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal] \\
& \qquad \land Send([type \mapsto \text{"1b"}, acc \mapsto a, bal \mapsto m.bal, \\
& \qquad\qquad\quad mbal \mapsto maxVBal[a], mval \mapsto maxVal[a]]) \\
& \quad \land \text{UNCHANGED } \langle maxVBal, maxVal \rangle
\end{aligned}
$$

In the $Phase2a(b, v)$ action, the ballot b leader sends a type "2a" message asking the acceptors to vote for v in ballot number b. The enabling conditions of the action–its first two conjuncts–ensure that three of the four enabling conditions of action $VoteFor(a, b, v)$ in module *Voting* will be true when acceptor a receives that message. Those three enabling conditions are the second through fourth conjuncts of that action.

The first conjunct of $Phase2a(b, v)$ asserts that at most one phase $2a$ message is ever sent for ballot b. Since an acceptor will vote for a value in ballot b only when it receives the appropriate phase $2a$ message, the phase $2a$ message sent by this action this ensures that these two enabling conjuncts of $VoteFor(a, b, v)$ will be true forever:

$$
\begin{aligned}
& \land \forall vt \in votes[a] : vt[1] \neq b \\
& \land \forall c \in Acceptor \setminus \{a\} : \forall vt \in votes[c] : (vt[1] = b) \Rightarrow (vt[2] = v)
\end{aligned}
$$

The second conjunct of the $Phase2a(b, v)$ action is the heart of the *Paxos* consensus algorithm. It's a bit complicated, but I've tried a number of times to write it in *English*, and it's much easier to understand when written in mathematics. The LET /IN construct locally defines $Q1$ to be the set of phase $1b$ messages sent in ballot number b by acceptors in quorum Q; and it defines $Q1bv$ to be the subset of those messages indicating that the sender had voted in some ballot (which must have been numbered less than b). You should study the IN clause to convince yourself that it equals $ShowsSafeAt(Q, b, v)$, defined in module *Voting*, using the values of $maxBal[a]$, $maxVBal[a]$, and $maxVal[a]a$ sent in its phase $1b$ message to describe what votes it had cast when it sent that message. Moreover, since a will no longer cast any votes in ballots numbered less than b, the IN clause implies that $ShowsSafeAt(Q, b, v)$ is still true and will remain true forever. Hence, this conjunct of $Phase2a(b, v)$ checks the enabling condition

$$\land \exists Q \in Quorum : ShowsSafeAt(Q, b, v)$$

of module *Voting*'s $VoteFor(a, b, v)$ action.

The type "2a" message sent by this action therefore tells every acceptor a that, when it receives the message, all the enabling conditions of $VoteFor(a, b, v)$ but the first, $maxBal[a] \leq b$, are satisfied.

$$
\begin{array}{l}
Phase2a(b, v) \triangleq \\
\quad \land \neg \exists m \in msgs : m.type = \text{“2a”} \land m.bal = b \\
\quad \land \exists Q \in Quorum : \\
\qquad \text{LET } Q1b \triangleq \{m \in msgs : \land m.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land m.acc \in Q \\
\qquad\qquad\qquad\qquad\qquad\qquad \land m.bal = b\} \\
\qquad\qquad Q1bv \triangleq \{m \in Q1b : m.mbal \geq 0\} \\
\qquad \text{IN } \quad \land \forall a \in Q : \exists m \in Q1b : m.acc = a \\
\qquad\qquad \land \lor Q1bv = \{\} \\
\qquad\qquad\quad \lor \exists m \in Q1bv : \\
\qquad\qquad\qquad \land m.mval = v \\
\qquad\qquad\qquad \land \forall mm \in Q1bv : m.mbal \geq mm.mbal \\
\quad \land Send([type \mapsto \text{“2a”}, bal \mapsto b, val \mapsto v]) \\
\quad \land \text{UNCHANGED } \langle maxBal, maxVBal, maxVal \rangle
\end{array}
$$

The $Phase2b(a)$ action describes what acceptor a does when it receives a phase $2a$ message m, which is sent by the leader of ballot $m.bal$ asking acceptors to vote for $m.val$ in that ballot. Acceptor a acts on that request, voting for $m.val$ in ballot number $m.bal$, iff $m.bal \geq maxBal[a]$, which means that a has not participated in any ballot numbered greater than $m.bal$. Thus, this enabling condition of the $Phase2b(a)$ action together with the receipt of the phase $2a$ message m implies that the $VoteFor(a, m.bal, m.val)$ action of module *Voting* is enabled and can be executed. The $Phase2b(a)$ message updates $maxBal[a]$, $maxVBal[a]$, and $maxVal[a]$ so their values mean what they were claimed to mean in the comments preceding the variable declarations.

$$
\begin{array}{l}
Phase2b(a) \triangleq \\
\quad \exists m \in msgs : \\
\qquad \land m.type = \text{“2a”} \\
\qquad \land m.bal \geq maxBal[a] \\
\qquad \land maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal] \\
\qquad \land maxVBal' = [maxVBal \text{ EXCEPT } ![a] = m.bal] \\
\qquad \land maxVal' = [maxVal \text{ EXCEPT } ![a] = m.val] \\
\qquad \land Send([type \mapsto \text{“2b”}, acc \mapsto a, \\
\qquad\qquad bal \mapsto m.bal, val \mapsto m.val])
\end{array}
$$

The definitions of *Next* and *Spec* are what we expect them to be.

$$
\begin{array}{l}
Next \triangleq \lor \exists b \in Ballot : \lor Phase1a(b) \\
\qquad\qquad\qquad\qquad\qquad \lor \exists v \in Value : Phase2a(b, v) \\
\qquad\quad\; \lor \exists a \in Acceptor : Phase1b(a) \lor Phase2b(a)
\end{array}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

We define *votes* to be the function such that $votes[a]$ is the set of pairs $\langle b, v \rangle$ such that acceptor a has voted for v in ballot number b by sending executing the $Phase2b(a)$ action to send the appropriate type "2b" message. The *Paxos* consensus algorithm implements the *Voting* algorithm by implementing the variable *votes* of module *Voting* with the expression *votes* of the current module, and implementing the variable *maxBal* of module *Voting* with the variable of the same name of the current module.

$$
\begin{array}{l}
votes \triangleq \\
\quad [a \in Acceptor \mapsto \\
\qquad \{\langle m.bal, m.val \rangle : m \in \{mm \in msgs : \begin{array}{l} \land\, mm.type = \text{"2b"} \\ \land\, mm.acc = a \}\}] \end{array}
\end{array}
$$

The following INSTANCE statement omits the redundant clause

WITH $votes \leftarrow votes,\ maxBal \leftarrow maxBal,$
$Value \leftarrow Value,\ Acceptor \leftarrow Acceptor,\ Quorum \leftarrow Quorum$

$$V \triangleq \text{INSTANCE } Voting$$

The inductive invariant *Inv* explains why the *Paxos* consensus algorithm implements the *Voting* algorithm. It is defined after the INSTANCE statement because it uses the operator $V\,!ShowsSafeAt$ imported by that statement.

$$
\begin{array}{l}
Inv \triangleq \\
\quad \land\, TypeOK \\
\quad \land\, \forall a \in Acceptor : maxBal[a] \geq maxVBal[a] \\
\quad \land\, \forall a \in Acceptor : \text{IF } maxVBal[a] = -1 \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } maxVal[a] = None \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } \langle maxVBal[a], maxVal[a] \rangle \in votes[a] \\
\quad \land\, \forall m \in msgs : \\
\qquad \land\, (m.type = \text{"1b"}) \Rightarrow \begin{array}[t]{l} \land\, maxBal[m.acc] \geq m.bal \\ \land\, (m.mbal \geq 0) \Rightarrow \\ \quad \langle m.mbal, m.mval \rangle \in votes[m.acc] \end{array} \\
\qquad \land\, (m.type = \text{"2a"}) \Rightarrow \begin{array}[t]{l} \land\, \exists Q \in Quorum : \\ \qquad V\,!ShowsSafeAt(Q, m.bal, m.val) \\ \land\, \forall mm \in msgs : \begin{array}[t]{l} \land\, mm.type = \text{"2a"} \\ \land\, mm.bal = m.bal \\ \Rightarrow mm.val = m.val \end{array} \end{array} \\
\qquad \land\, (m.type = \text{"2b"}) \Rightarrow \begin{array}[t]{l} \land\, maxVBal[m.acc] \geq m.bal \\ \land\, \exists mm \in msgs : \begin{array}[t]{l} \land\, mm.type = \text{"2a"} \\ \land\, mm.bal = m.bal \\ \land\, mm.val = m.val \end{array} \end{array}
\end{array}
$$

The following two theorems assert that *Inv* is an invariant of the *Paxos* cconsensus algorithm and that this algorithm implements the *Voting* algorithm with the declared variables and constants of that algorithm implemented by the correspondingly-named expressions in the current module.

$$\text{THEOREM } Invariance \triangleq Spec \Rightarrow \Box Inv$$

$$\text{THEOREM } Implementation \triangleq Spec \Rightarrow V\,!Spec$$

The ASSUME statement of this module trivial implies trivially implies the instantiated version of the ASSUME statement of module *Voting*. (Because the INSTANCE statement substitutes the constants of the current module for the constants of the same name in module *Voting*, the imported assumption is the same as the assumption of the current module.) Hence, this theorem imported from module *Voting* is true in the current module

$$\text{THEOREM}\quad V!Implementation \triangleq V!Spec \Rightarrow V!C!Spec$$

Theorems *Implementation* and $V!Implementation$ imply

$$\text{THEOREM}\quad Spec \Rightarrow V!C!Spec$$

This theorem asserts that the *Paxos* consensus algorithm implements the Consensus specification by substituting for the variable *chosen* of the *Consensus* specification the value $V!chosen$ of the current module. The expression $V!chosen$ is obtained by substituting the expression *votes* of the current module for the variable *votes* of module *Voting* in the expression *chosen* of module *Voting*.

In other words, as we should expect, "implements" is a transitive relation–under a suitable understanding of what transitivity means in this situation.

This current module is distributed with two models, *TinyModel* and *SmallModel*. *SmallModel* is the same as the model by that name for the Voting specification. *TinyModel* is the same except it defines *Ballot* to contain only two elements. Run *TLC* on these models. You should find that it takes a couple of seconds to run *TinyModel* and two or three minutes to run *SmallModel*.

Next, try the same thing you did with the *Voting* algorithm: Modify the models so the assumption that any pair of quorums has an element in common is no longer true. (Again, it's best to modify clones of the models.) This time, running *TLC* will not find an error. The correctness of theorems *Invariance* and *Implementation* does not depend on that assumption. The *Paxos* consensus algorithm still correctly implements the *Voting* algorithm; but the *Voting* algorithm is incorrect if the assumption does not hold.

Now go back to the original *SmallModel*, in which the quorum assumption holds. The sets *Acceptor* and *Value* are symmetry sets for the spec. (See the "Model Values and Symmetry" help page to find out what that means.) Try editing the values substituted for *Acceptor* and/or *Value* by selecting the "Symmetry set" option and comparing the number of reachable states *TLC* found and the time it took. (Remember to use cloned models.)

When you have other things to do while *TLC* is running, try increasing the size of the model a very little bit at a time and see how the running time increases. You'll find that it increases exponentially with the numbers of acceptors, values, and ballots.

Fortunately, exhaustively checking a small model is very effective at finding errors. Since the *Paxos* consensus algorithm has been proved correct, and that proof has been read by many people, I'm sure that the basic algorithm is correct. Checking this spec on *SmallModel* makes me quite confident that there are no "coding errors" in this TLA+ specification of the algorithm.

For checking safety properties, *TLC* can obtain close to linear speedup using dozens of processors. After designing a new distributed algorithm, you will have plenty of time to run *TLC* while the algorithm is being implemented and the implementation tested. Use that time to run it for as long as you can on the largest *machine*(s) that you can. Testing the implementation is unlikely to find subtle errors in the algorithm.