MODULE *Voting*

This is a high-level algorithm in which a set of processes cooperatively choose a value. It is a high-level abstraction of the Paxos consensus algorithm. Although I don't remember exactly what went through my mind when I invented/discovered that algorithm, I'm pretty sure that this spec formalizes the way I first thought of the algorithm. It would have been very hard to find this algorithm had my mind been distracted by the irrelevant details introduced by having the processes communicate by messages.

EXTENDS *Integers*

For historical reasons, the processes that choose a value are called acceptors. We now declare the set *Value* of values, the set Acceptors of acceptors, and another set *Quorum* that is a set of sets of acceptors called quorums.

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

The following assumption asserts that *Quorum* is a set of subsets of the set *Acceptor*, and that any two elements of *Quorum* have at least one element (an acceptor) in common. Think of a quorum as a set consisting of a majority (more than half) of the acceptors.

ASSUME $\land \forall Q \in Quorum : Q \subseteq Acceptor$
$\qquad\land \forall Q1, Q2 \in Quorum : Q1 \cap Q2 \neq \{\}$

Ballot is a set of "ballot numbers". For simplicity, we let it be the set of natural numbers. However, we write *Ballot* for that set to distinguish ballots from natural numbers used for other purposes.

$Ballot \triangleq Nat$

The algorithm works by having acceptors cast votes in numbered ballots. Each acceptor can cast one or more votes, where each vote cast by an acceptor has the form $\langle b, v \rangle$ indicating that the acceptor has voted for value v in ballot number b. A value is chosen if a quorum of acceptors have voted for it in the same ballot.

We now declare the algorithm's variables 'votes' and 'maxBal'. For each acceptor a, he value of $votes[a]$ is the set of votes cast by a ; and $maxBal[a]$ is an integer such that a will never cast any further vote in a ballot numbered less than $maxBal[a]$.

VARIABLES *votes*, *maxBal*

TypeOK asserts the "types" of the two variables. They are both functions with domain *Acceptor* (arrays indexed by acceptors). For any acceptor a, the value of $votes[a]$ a set of $\langle ballot, value \rangle$ pairs; and the value of $maxBal[a]$ is either a ballot number or -1.

$TypeOK \triangleq$
$\quad\land votes \in [Acceptor \rightarrow \text{SUBSET } (Ballot \times Value)]$
$\quad\land maxBal \in [Acceptor \rightarrow Ballot \cup \{-1\}]$

Next comes a sequence of definitions of concepts used to explain the algorithm.

$VotedFor(a, b, v) \triangleq \langle b, v \rangle \in votes[a]$

True iff (if and only if) acceptor a has votted for value v in ballot number b.

$ChosenAt(b, v) \triangleq$
$\quad\exists Q \in Quorum : \forall a \in Q : VotedFor(a, b, v)$

True iff a quorum of acceptors have all voted for value v in ballot number b.

$chosen \triangleq \{v \in Value : \exists\, b \in Ballot : ChosenAt(b, v)\}$

Defines *chosen* to be the set of all values v for which $ChosenAt(b, v)$ is true for some ballot number b. This is the definition of what it means for a value to be chosen under which the *Voting* algorithm implements the *Consensus* specification.

$DidNotVoteAt(a, b) \triangleq \forall\, v \in Value : \neg VotedFor(a, b, v)$

True iff acceptor a has not voted in ballot number .

$$CannotVoteAt(a, b) \triangleq \begin{array}{l} \land\ maxBal[a] > b \\ \land\ DidNotVoteAt(a, b) \end{array}$$

The algorithm will not allow acceptor a to vote in ballot number b if $maxBal[a] > b$. Hence, $CannotVoteAt(a, b)$ implies that a has not and never will vote in ballot number b.

$$\begin{array}{l} NoneOtherChoosableAt(b, v) \triangleq \\ \quad \exists\, Q \in Quorum : \\ \quad\quad \forall\, a \in Q : VotedFor(a, b, v) \lor CannotVoteAt(a, b) \end{array}$$

This is true iff there is some quorum Q such that each acceptor a in Q either has voted for v in ballot b or has not and never will vote in ballot b. It implies that no value other than v has been or ever can be chosen at ballot b. This is because for a value w to be chosen at ballot b, all the acceptors in some quorum R must have voted for w in ballot b. But any two ballots have an acceptor in common, so some acceptor a in R that voted for w is in Q, and an acceptor in Q can only have voted for v, so w must equal v.

$SafeAt(b, v) \triangleq \forall\, c \in 0 \,..\, (b - 1) : NoneOtherChoosableAt(c, v)$

True iff no value other than v has been or ever will be chosen in any ballot numbered less than b. We read $SafeAt(b, v)$ as "v is safe at b".

This theorem asserts that every value is safe at ballot 0.

$\textsc{theorem}\ AllSafeAtZero \triangleq \forall\, v \in Value : SafeAt(0, v)$

The following theorem asserts that *NoneOtherChoosableAt* means what it's name implies. The comments after its definition essentially contain a proof of this theorem.

$$\begin{array}{l} \textsc{theorem}\ ChoosableThm \triangleq \\ \quad\quad \forall\, b \in Ballot, v \in Value : \\ \quad\quad\quad ChosenAt(b, v) \Rightarrow NoneOtherChoosableAt(b, v) \end{array}$$

Now comes the definition of the inductive invariant *Inv* that essentially explains why the algorithm is correct.

$$\begin{array}{l} OneValuePerBallot \triangleq \\ \quad \forall\, a1, a2 \in Acceptor, b \in Ballot, v1, v2 \in Value : \\ \quad\quad VotedFor(a1, b, v1) \land VotedFor(a2, b, v2) \Rightarrow (v1 = v2) \end{array}$$

This formula asserts that if any acceptors $a1$ and $a2$ have voted in a ballot b, then they voted for the same value in ballot b. For $a1 = a2$, this implies that an acceptor can vote for at most one value in any ballot.

$$\begin{array}{l} VotesSafe \triangleq \forall\, a \in Acceptor, b \in Ballot, v \in Value : \\ \quad\quad\quad\quad VotedFor(a, b, v) \Rightarrow SafeAt(b, v) \end{array}$$

This formula asserts that an acceptors can have voted in a ballot b only if that value is safe at b.

The algorithm is essentially derived by ensuring that this formula *Inv* is always true.

$$Inv \triangleq TypeOK \land VotesSafe \land OneValuePerBallot$$

This definition is used in the defining the algorithm. You should study it and make sure you understand what it says.

$$
\begin{aligned}
&ShowsSafeAt(Q, b, v) \triangleq \\
&\quad \land \forall a \in Q : maxBal[a] \geq b \\
&\quad \land \exists c \in -1 \,..\, (b-1) : \\
&\qquad \land (c \neq -1) \Rightarrow \exists a \in Q : VotedFor(a, c, v) \\
&\qquad \land \forall d \in (c+1) \,..\, (b-1),\ a \in Q : DidNotVoteAt(a, d)
\end{aligned}
$$

This is the theorem that's at the heart of the algorithm. It shows that if the algorithm has maintained the invariance of *Inv*, then the truth of $ShowsSafeAt(Q, b, v)$ for some quorum Q ensures that v is safe at b, so the algorithm can let an acceptor vote for v in ballot b knowing *VotesSafe* will be preserved.

$$
\begin{aligned}
&\textsc{theorem}\ ShowsSafety \triangleq \\
&\qquad Inv \Rightarrow \forall Q \in Quorum,\ b \in Ballot,\ v \in Value : \\
&\qquad\qquad ShowsSafeAt(Q, b, v) \Rightarrow SafeAt(b, v)
\end{aligned}
$$

Finally, we get to the definition of the algorithm. The initial predicate is obvious.

$$
\begin{aligned}
Init \triangleq\ & \land votes \quad\ = [a \in Acceptor \mapsto \{\}] \\
& \land maxBal = [a \in Acceptor \mapsto -1]
\end{aligned}
$$

An acceptor a can increase $maxBal[a]$ at any time.

$$
\begin{aligned}
&IncreaseMaxBal(a, b) \triangleq \\
&\quad \land b > maxBal[a] \\
&\quad \land maxBal' = [maxBal\ \textsc{except}\ ![a] = b] \\
&\quad \land \textsc{unchanged}\ votes
\end{aligned}
$$

The heart of the algorithm is the action in which an acceptor a votes for a value v in ballot number b. The enabling condition contains the following conjuncts, which ensure that the invariance of *Inv* is maintained.

– a cannot vote in a ballot numbered less than b

– a cannot already have voted in ballot number b

- No other acceptor can have voted for a value other than v in ballot b.

- Uses Theorem *ShowsSafety* to ensure that v is safe at b.

In TLA+, a tuple t is a function (array) whose first element is $t[1]$, whose second element if $t[2]$, and so on. Thus, a vote vt is the pair
$\langle vt[1], vt[2] \rangle$

$$
\begin{aligned}
&VoteFor(a, b, v) \triangleq \\
&\quad \land \quad maxBal[a] \leq b \\
&\quad \land \quad \forall vt \in votes[a] : vt[1] \neq b
\end{aligned}
$$

$\land \quad \forall\, c \in \mathit{Acceptor} \setminus \{a\} :$
$\qquad \forall\, vt \in \mathit{votes}[c] : (vt[1] = b) \Rightarrow (vt[2] = v)$
$\land \quad \exists\, Q \in \mathit{Quorum} : \mathit{ShowsSafeAt}(Q,\ b,\ v)$
$\land \quad \mathit{votes}' \quad = [\mathit{votes}\ \text{EXCEPT}\ ![a] = \mathit{votes}[a] \cup \{\langle b,\ v\rangle\}]$
$\land \quad \mathit{maxBal}' = [\mathit{maxBal}\ \text{EXCEPT}\ ![a] = b]$

The rest of the spec is straightforward.

$$\mathit{Next} \triangleq \exists\, a \in \mathit{Acceptor},\ b \in \mathit{Ballot} : \begin{array}{l} \lor\ \mathit{IncreaseMaxBal}(a,\ b) \\ \lor\ \exists\, v \in \mathit{Value} : \mathit{VoteFor}(a,\ b,\ v) \end{array}$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{votes},\ \mathit{maxBal}\rangle}$$

This theorem asserts that *Inv* is an invariant of the algorithm. The high-level steps in its proof are given.

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

$\langle 1\rangle 1.\ \mathit{Init} \Rightarrow \mathit{Inv}$

$\langle 1\rangle 2.\ \mathit{Inv} \land [\mathit{Next}]_{\langle \mathit{votes},\ \mathit{maxBal}\rangle} \Rightarrow \mathit{Inv}'$

$\langle 1\rangle 3.$ QED
BY $\langle 1\rangle 1$, $\langle 1\rangle 2$ DEF *Spec*

This INSTANCE statement imports definitions from module *Consensus* into the current module. All definition in module *Consensus* can be expanded to definitions containing only TLA+ primitives and the declared names Value and chosen. To import a definition from *Consensus* into the current module, we have to say what expressions from the current module are substituted for *Value* and chosen. The INSTANCE statement says that expressions of the same name are substituted for them. (Because of this, the WITH statement is redundant.) Note that in both modules, Value is just a declared constant. However, in the current module, *chosen* is an expression defined in terms of the variables votes and *maxBal* while it is a variable in module consensus. The "$C! \triangleq$ " in the statement means that defined identifiers are imported with "$C!$" prepended to their names. Thus *Spec* of module *Consensus* is imported, with these substitutions, as $C!\mathit{Spec}$.

$C \triangleq$ INSTANCE *Consensus*
WITH $\mathit{Value} \leftarrow \mathit{Value},\quad \mathit{chosen} \leftarrow \mathit{chosen}$

The following theorem asserts that the *Voting* algorithm implements the Consensus specification, where the expression *chosen* of the current module implements the variable *chosen* of *Consensus*. The high-level steps of the the proof are also given.

THEOREM $\mathit{Implementation} \triangleq \mathit{Spec} \Rightarrow C!\mathit{Spec}$

$\langle 1\rangle 1.\ \mathit{Init} \Rightarrow C!\mathit{Init}$

$\langle 1\rangle 2.\ \mathit{Inv} \land \mathit{Inv}' \land [\mathit{Next}]_{\langle \mathit{votes},\ \mathit{maxBal}\rangle} \Rightarrow [C!\mathit{Next}]_{\mathit{chosen}}$

$\langle 1\rangle 3.$ QED
BY $\langle 1\rangle 1$, $\langle 1\rangle 2$, *Invariance* DEF *Spec*, $C!\mathit{Spec}$

This *Voting* specification comes with a *TLC* model named *SmallModel*. That model tells *TLC* that *Spec* is the specification, that *Acceptor* and Values should equal sets of model values with 3 acceptors and 2 values and *Quorums* should equal the indicated set of values, and that it should check theorems *Invariance* and *Implementation*. Observe that you can't tell *TLC* simply to check those theorems; you have to tell *TLC* to check the properties the theorems assert that *Spec* satisfies. (Instead of telling *TLC* that *Inv* should be an invariant, you can tell it that the spec should satisfy the temporal property $\Box Inv$.)

Even though the constants are finite sets, the spec has infinitely many reachable sets because a ballot number can be any element of the set *Nat* of natural numbers. The model modifies the spec so it has a finite set of reachable states by using a definition override (on the *Spec* Options page) to redefine *Ballot* to equal $0 \,..\, 2$. (Alternatively, we could override the definition of *Nat* to equal $0 \,..\, 2$.)

Run *TLC* on the model. It should just take a couple of seconds.

After doing that, show why the assumption that any pair of quorums has an element in common is necessary by modifying the model so that assumption doesn't hold. (It's best to clone *SmallModel* and modify the clone.) Since *TLC* reports an error if an assumption isn't true, you will have to comment out the second conjunct of the ASSUME statement, which asserts that assumption. Comments can be written as follows:

* This is an end-of-line comment.

To help you see what the problem is, use the Trace Explorer on the Error page to show the value of *chosen* in each state of the trace.