MODULE *PaxosMadeSimple*

A specification of the algorithm described in "Paxos Made Simple", with some parts copied from the specification found at: http://research.microsoft.com/en-us/um/people/lamport/tla/PConProof.tla

The specification can be found at: https://github.com/nano-o/PaxosMadeSimple

EXTENDS *Integers*, *FiniteSets*

CONSTANT *Value*, *Acceptor*, *Quorum*

The quorum assumptions

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

Ballot numbers are the natural numbers.

$Ballot \triangleq Nat$

We are going to have a leader *PlusCal* process for each ballot and an acceptor *PlusCal* process for each acceptor (see the "process" definitions below). A proposer that proposes a value in ballot n is modeled by the leader *PlusCal* process identified by n. Thus a single proposer is modeled by multiple leader *PlusCal* processes, one for each ballot in which the proposer proposes a value. We use the ballot numbers and the acceptors themselves as the identifiers for these processes and we assume that the set of ballots and the set of acceptors are disjoint. We also assume that -1 is not an acceptor, although that is probably not necessary.

ASSUME $BallotAssump \triangleq (Ballot \cup \{-1\}) \cap Acceptor = \{\}$

We define *None* to be an unspecified value that is not in the set *Value*.

$None \triangleq$ CHOOSE $v : v \notin Value$

This is a message-passing algorithm, so we begin by defining the set Message of all possible messages. The messages are explained below with the actions that send them. A message m with $m.type =$ "1a" is called a $1a$ message, and similarly for the other message types.

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

The algorithm is easiest to understand in terms of the set *msgs* of all messages that have ever been sent. A more accurate model would use one or more variables to represent the messages actually in transit, and it would include actions representing message loss and duplication as well as message receipt.

In the current spec, there is no need to model message loss explicitly. 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. The liveness property of the spec will make it clear what messages must be received (and hence either not lost or successfully retransmitted if lost) to guarantee progress.

Another advantage of maintaining the set of all messages that have ever been sent is that it allows us to define the state function *votes* that implements the variable of the same name in the voting algorithm without having to introduce a history variable.

```
83  --algorithm PCon{
84    variables maxBal  = [a ∈ Acceptor ↦ −1],
85              maxVBal = [a ∈ Acceptor ↦ −1],
86              maxVVal = [a ∈ Acceptor ↦ None],
87              msgs = {}
88    define {
89      sentMsgs(t, b) ≜ {m ∈ msgs : (m.type = t) ∧ (m.bal = b)}

91      Max(xs, LessEq(_, _)) ≜
92            CHOOSE x ∈ xs : ∀ y ∈ xs : LessEq(y, x)

94      HighestAcceptedValue(Q1bMessages) ≜
95          Max(Q1bMessages, LAMBDA m1, m2 : m1.mbal ≤ m2.mbal).mval
```

We define *ShowsSafeAt* so that *ShowsSafeAt*(Q, b, v) is true for a quorum Q iff the *msgs* contained in ballot-b 1b messages from the acceptors in Q show that v is safe at b.

```
102     ShowsSafeAt(Q, b, v) ≜
103       LET Q1b ≜ {m ∈ sentMsgs("1b", b) : m.acc ∈ Q}
104       IN  ∧ ∀ a ∈ Q : ∃ m ∈ Q1b : m.acc = a
105           ∧ ∨ ∀ m ∈ Q1b : m.mbal = −1
106             ∨ v = HighestAcceptedValue(Q1b)
107     }
```

We describe each action as a macro.

In *PlusCal*, *self* is used by convention to designate the id of the *PlusCal* process being defined.

The leader for ballot *self* can execute a *Phase*1a() action, which sends the ballot *self* 1a message (remember that for leader *PlusCal* processes, the identifier of the process is the ballot number).

```
119   macro Phase1a(){msgs := msgs ∪ {[type ↦ "1a", bal ↦ self]} ; }
```

Acceptor *self* can perform a *Phase*1b(b) action, which is enabled iff $b >$ *maxBal*[*self*]. The action sets *maxBal*[*self*] to b (therefore promising never to accept proposals with ballot lower than b: see *Phase*2b below) and sends a phase 1b message to the leader of ballot b containing the values of *maxVBal*[*self*] and *maxVVal*[*self*].

```
128   macro Phase1b(b){
129     when (b > maxBal[self]) ∧ (sentMsgs("1a", b) ≠ {}) ;
130     maxBal[self] := b ;
131     msgs := msgs ∪ {[type ↦ "1b", acc ↦ self, bal ↦ b,
132                      mbal ↦ maxVBal[self], mval ↦ maxVVal[self]]} ;
```

```
133   }
```

The ballot *self* leader can perform a $Phase2a(v)$ action, sending a $2a$ message for value v, if it has not already sent a $2a$ message (for this ballot) and it can determine that v is safe at ballot *self* .

```
140   macro Phase2a(v){
141     when  ∧ sentMsgs("2a", self) = {}
142           ∧ ∃ Q ∈ Quorum : ShowsSafeAt(Q, self, v) ;
143     msgs := msgs ∪ {[type ↦ "2a", bal ↦ self, val ↦ v]} ;
144   }
```

The $Phase2b(b)$ action is executed by acceptor *self* in response to a ballot-b $2a$ message. Note this action can be executed multiple times by the acceptor, but after the first one, all subsequent executions are stuttering steps that do not change the value of any variable. Note that the acceptor *self* does not accept any proposal with a ballot lower than b, as per its promise to the leader of ballot b in $Phase1b$ above.

Note that there is not need to update *maxBal*.

```
157   macro Phase2b(b){
158     when b ≥ maxBal[self] ;
159     with (m ∈ sentMsgs("2a", b)){
160       if (b ≥ maxVBal[self]){
161         maxVBal[self] := b ;
162         maxVVal[self] := m.val
163       } ;
164       msgs := msgs ∪ {[type ↦ "2b", acc ↦ self,
165                                bal ↦ b, val ↦ m.val]}
166     }
167   }
```

An acceptor performs the body of its *while* loop as a single atomic action by nondeterministically choosing a ballot in which its $Phase1b$ or $Phase2b$ action is enabled and executing that enabled action. If no such action is enabled, the acceptor does nothing.

```
175   process (acceptor ∈ Acceptor){
176     acc: while (TRUE){
177             with (b ∈ Ballot){either Phase1b(b)or Phase2b(b) }}
178   }
```

The leader of a ballot nondeterministically chooses one of its actions that is enabled (and the argument for which it is enabled) and performs it atomically. It does nothing if none of its actions is enabled.

```
186   process (leader ∈ Ballot){
187     ldr: while (TRUE){
188           either Phase1a()
189           or     with (v ∈ Value){Phase2a(v)}
190         }
191   }
```

```
193  }
197
```

\ * Modification History
\ * Last modified *Mon Feb* 26 09:19:01 *PST* 2018 by *nano*
\ * Created *Thu Sep* 03 22:58:03 *EDT* 2015 by *nano*