MODULE *CASPaxos*

****

This is an interesting extension of the Single-decree *Paxos* algorithm to a compare-and-swap type register. The algorithm is very similar to *Paxos*, but before starting the ACCEPT phase, proposers are free to mutate the value. The result is that the *Paxos* instance turns from a write-once register into a reusable register with atomic semantics, for example a compare-and-swap register.

****

EXTENDS $Integers, FiniteSets$

****

The data one has to define for the model. In this case, a set of possible *Values*, a set of acceptors, and a mutator which maps a ballot number and a value to a new value.

The *Mutator* is a good approximation of how compare-and-swap'ping proposers would choose the new values, abstracting away nondeterminism by using the ballot number to decide on the new value.

For model checking, infinite sets must be avoided. For convenience, we more or less explicitly assume that values can be compared.

****

CONSTANT $Values, Acceptors, Mutator(\_, \_)$

****

The set of quorums. We automatically construct this by taking all the subsets of *Acceptors* which are majorities, *i.e.* larger in number than the unchosen ones. This is for convenience; any definition for which *QuorumAssumption* below holds is valid.

****

$$Quorums \triangleq \{S \in \text{SUBSET } (Acceptors) : Cardinality(S) > Cardinality(Acceptors \setminus S)\}$$

****

The register in this algorithm can repeatedly change its value. For simplicity, we don't let it start from "no value" but explicitly specify a first value here. Choosing the smallest value is reasonable, but you can change this however you like.

****

$$InitialValue \triangleq \text{CHOOSE } v \in Values : \forall vv \in Values : v \leq vv$$

****

Sanity check that our defined quorums all have nontrivial pair-wise intersections. This is pretty clear with the above definition of *Quorums*, but note that you could specify any set of quorums (even minorities) and the algorithm should work the same way, as long as *QuorumAssumption* holds.

****

$$\text{ASSUME } QuorumAssumption \triangleq \begin{array}{l} \land \forall Q \in Quorums : Q \subseteq Acceptors \\ \land \forall Q1, Q2 \in Quorums : Q1 \cap Q2 \neq \{\} \end{array}$$

****

Ballot numbers are natural numbers, but it's good to have an alias so that you know when you're talking about ballots.

****

$$Ballot \triangleq Nat$$

****

Now that we know what a ballot is, check that the *Mutator* maps Ballots and *Values* to *Values*.

****

ASSUME $\land \forall b \in Ballot, v \in Values : Mutator(b, v) \in Values$

****

Define the set of all possible messages. In this specification, proposers are implicit. Messages originating from them are created "out of thin air" and not addressed to a specific acceptor. In practice they would be, though note that each acceptor would receive the same "message body", and omitting the explicit recipient reduces the state space. Note also that messages are not explicitly rejected but simply not reacted to. In particular, the implicit proposer has no notion of which ballot to try next. The spec lets them try arbitrary ballots instead.

A message is either a prepare request for a ballot, a prepare response, an accept request for a ballot with a new value, or an accept response.

****

$Message \triangleq$ $[type : \{\text{"prepare-req"}\}, bal : Ballot]$
$\quad \cup$ $[$
$\quad\quad type : \{\text{"prepare-rsp"}\}, acc : Acceptors,$
$\quad\quad promised : Ballot,$ ballot for which promise is given
$\quad\quad accepted : Ballot,$ ballot at which *val* was accepted
$\quad\quad val : Values$
$\quad]$
$\quad \cup$ $[type : \{\text{"accept-req"}\}, bal : Ballot, newVal : Values]$
$\quad \cup$ $[type : \{\text{"accept-rsp"}\}, acc : Acceptors, accepted : Ballot]$

****

$\langle prepared[a], accepted[a], value[a] \rangle$ is the state of acceptor a: The ballot for which a promise has been made (*i.e.* no smaller ballot's value will be accepted); the ballot at which the last value has been accepted; and the last accepted value.

****

VARIABLE $prepared,$
$\quad accepted,$
$\quad value$

****

The set of all messages sent. In each state transition of the model, a message which could solicit a transaction may be reacted to. Note that this implicitly models that a message sent may be received multiple times, and that everything can arbitrarily reorder.

****

VARIABLE $msgs$

****

An invariant which checks that the variables have values which make sense.

****

$TypeOK \triangleq \land prepared \in [Acceptors \rightarrow Ballot]$
$\quad\quad \land accepted \in [Acceptors \rightarrow Ballot]$

```
            ∧ value ∈   [Acceptors → Values]
            ∧ msgs ⊆ Message
```

****

The initial state of the model. Note that the state here has an initial committed value, ie the register doesn't start “empty”. This is an inconsequential simplification.

****

```
Init ≜  ∧ prepared = [a ∈ Acceptors ↦ 0]
        ∧ accepted = [a ∈ Acceptors ↦ 0]
        ∧ value    = [a ∈ Acceptors ↦ InitialValue]
        ∧ msgs = {}
```

****

Sending a message just means adding it to the set of all messages.

****

```
Send(m) ≜ msgs′ = msgs ∪ {m}
```

****

A ballot is started by sending a prepare request (with the hope that responses will be received from a quorum). We could allow multiple prepare requests for a single ballot, but since they are all identical and we already model multiple-receipt for all messages, this adds only state space complexity. So a ballot will only be prepared once in this model.

****

```
BallotActive(b) ≜ ∃ m ∈ msgs :
                        ∧ m.type = "prepare-req"
                        ∧ m.bal = b
PrepareReq(b) ≜
    ∧ ¬BallotActive(b)
    ∧ Send([
            type ↦ "prepare-req",
            bal  ↦ b
          ])
    ∧ UNCHANGED (⟨prepared, accepted, value⟩)
```

****

A prepare response can be sent if by an acceptor if a) a response was demanded via a prepare request and b) the acceptor has not already prepared that or any larger ballot. On success, the acceptor remembers that it has prepared the new ballot, and sends to response.

****

```
PrepareRsp(a) ≜
    ∧ ∃ m ∈ msgs :
        ∧ m.type = "prepare-req"
        ∧ m.bal > prepared[a]
        ∧ prepared′ = [prepared EXCEPT ![a] = m.bal]
        ∧ Send([
                acc     ↦ a,
                type    ↦ "prepare-rsp",
```

```
                    promised ↦ m.bal,
                    accepted ↦ accepted[a],
                    val      ↦ value[a]
                 ])
   ∧ UNCHANGED ⟨accepted, value⟩
```

****

An accept request can only be sent ($i.e.$ be fabricated from thin air) if a) once; b) if prepare responses for the ballot have been received from a quorum; c) with a new value based on the most recently accepted value from the prepare responses.

****

```
AcceptReq(b, v) ≜
   ∧ ¬∃ m ∈ msgs : m.type = "accept-req" ∧ m.bal = b
   ∧ ∃ Q ∈ Quorums :
       LET M ≜ {m ∈ msgs : ∧ m.type = "prepare-rsp"
                           ∧ m.promised = b
                           ∧ m.acc ∈ Q}
       IN  ∧ ∀ a ∈ Q : ∃ m ∈ M : m.acc = a
           ∧ ∃ m ∈ M :
               ∧ m.val = v
               ∧ ∀ mm ∈ M : mm.accepted ≤ m.accepted
   ∧ LET newVal ≜ Mutator(b, v)   crucial difference from Paxos
     IN  Send([
                type   ↦ "accept-req",
                bal    ↦ b,
                newVal ↦ newVal
              ])
   ∧ UNCHANGED (⟨accepted, value, prepared⟩)
```

****

An acceptor can reply to an accept request only if it hasn't yet prepared a higher ballot. Before replying, it makes sure it marks the ballot as prepared (as the particular acceptor may not have received the associated prepare request earlier), and updates its accepted ballot and the new value.

****

```
AcceptRsp(a) ≜
   ∧ ∃ m ∈ msgs :
       ∧ m.type = "accept-req"
       ∧ m.bal ≥ prepared[a]
       ∧ prepared′ = [prepared EXCEPT ![a] = m.bal]
       ∧ accepted′ = [accepted EXCEPT ![a] = m.bal]
       ∧ value′    = [value    EXCEPT ![a] = m.newVal]
       ∧ Send([
                 acc      ↦ a,
                 type     ↦ "accept-rsp",
                 accepted ↦ m.bal
               ])
```

Next is true if and only if the new state (*i.e.* that with primed variables) is valid. This is used to simulate the model by constructing new states until we run out of options. Concretely, the below means that either we prepare a ballot, or can react successfully to prepare request, or there is an acceptor which can find a message it can react to.

$$
\begin{aligned}
&\mathit{Next} \triangleq \ \lor \exists\, b \in \mathit{Ballot} : \lor \mathit{PrepareReq}(b) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \lor \exists\, v \in \mathit{Values} : \mathit{AcceptReq}(b,\, v) \\
&\qquad\quad\ \ \lor \exists\, a \in \mathit{Acceptors} : \mathit{PrepareRsp}(a) \lor \mathit{AcceptRsp}(a)
\end{aligned}
$$

Spec is the (default) entry point for the TLA+ model runner. The below formula is a temporal formula and means that the valid behaviors of the specification are those which initially satisfy *Init*, and from each step to the following the formula *Next* is satisfied, unless none of the variables changes (which is called a "stuttering step"). The model runner uses this to expand all possible behaviors.

$$
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{prepared},\, \mathit{accepted},\, \mathit{value},\, \mathit{msgs} \rangle}
$$

Equipped with a model, what invariants do we want to hold? Or, in other words, what exactly is it that we think the algorithm guarantees? Naively, each newly committed value should be in some relation to a previous value, so when you're not thinking to hard about it, you could hope that when you take a committed value A and the previously committed value B, then B was created by mutating A. It's not quite as simple, but going down that wrong track highlights how to use the model checker to find interesting histories.

If you have a minute, figure out why the above assumption is wrong. You don't need more than three ballots and two concurrent proposals.

For a given ballot, find the acceptors which (at some point in time) accepted that ballot.

$$
\begin{aligned}
&\mathit{AcceptedBy}(b) \triangleq \{a \in \mathit{Acceptors} : \\
&\quad \exists\, m \in \mathit{msgs} : \land\, m.\mathit{type} = \text{"accept-rsp"} \\
&\qquad\qquad\qquad\quad\ \land\, m.\mathit{acc} = a \\
&\qquad\qquad\qquad\quad\ \land\, m.\mathit{accepted} = b\}
\end{aligned}
$$

For a given ballot, find out whether the ballot was ever accepted by a quorum.

$$
\mathit{AcceptedByQuorum}(b) \triangleq \exists\, Q \in \mathit{Quorums} : \mathit{AcceptedBy}(b) \cap Q = Q
$$

The set of committed ballot numbers. Note that 0 is trivially committed thanks to the initialization of the state.

$$
\mathit{CommittedBallots} \triangleq \{b \in \mathit{Ballot} : \mathit{AcceptedByQuorum}(b)\} \cup \{0\}
$$

****

For a given ballot $b > 0$, find the next lowest ballot number which was committed (note that this doesn't have to be b-1).

****

```
BallotCommittedBefore(b) ≜ CHOOSE c ∈ CommittedBallots :
                              ∧ c < b
                              ∧ ∀ cc ∈ CommittedBallots :
                                   cc ≥ b ∨ cc ≤ c
```

****

For a given ballot, collect all the values which an acceptor requested. This set will always either be empty or a singleton, but that's not something you can tell from this definition, though we assert it below.

****

```
ValuesAt(b) ≜ IF b = 0 THEN {InitialValue}
                 ELSE {v ∈ Values :
                         ∃ m ∈ msgs :
                           ∧ m.type     = "accept-rsp"
                           ∧ m.accepted = b
                           ∧ ∃ mm ∈ msgs :
                               ∧ mm.type   = "accept-req"
                               ∧ mm.bal    = b
                               ∧ mm.newVal = v
                      }
OnlyOneValuePerBallot ≜ ∀ b ∈ Ballot : Cardinality(ValuesAt(b)) ≤ 1
```

****

MutationsLineUp is the main (ill-fated) assertion that each new value of the register is based on a previously committed value.

****

```
UnwrapSingleton(s) ≜ CHOOSE v ∈ s : TRUE   {x} ↦ x
MutationsLineUp ≜
    ∀ b ∈ CommittedBallots \ {0} :
       LET newVal ≜ UnwrapSingleton(ValuesAt(b))
           prevCommitBallot ≜ BallotCommittedBefore(b)
           oldVal ≜ UnwrapSingleton(ValuesAt(prevCommitBallot))
       IN  newVal = Mutator(b, oldVal)
```

****

DesiredProperties is a formula that we will tell the model checker to verify for each state in all valid behaviors. When it is something that actually holds, folks usually call it *Inv* (meaning an inductive invariant), and may try to prove it mechanically using the TLA proof checker, but it is a lot of work and often nontrivial.

In our case, there will be behaviors that violate *MutationsLineUp*.

****

```
DesiredProperties ≜ ∧ TypeOK
```

$\land$ *OnlyOneValuePerBallot*
$\land$ *MutationsLineUp*

---

* Modification History
* Last modified *Fri Apr* 07 02:26:16 *EDT* 2017 by *tschottdorf*
* Created *Thu Apr* 06 02:12:06 *EDT* 2017 by *tschottdorf*