MODULE *Correctness*

We are interested in implementing State-Machine Replication (*SMR*) accross multiple replicas communicating in an asynchronous network. Each replica has a copy of the same state machine, which accepts commands that modify the state of the state machine and return an output. Roughly speaking, the goal of the $M2Paxos$ algorithm is to meet the following requirements despite the possible failure of strictly less than half of the replicas:

1) ensure that if two replicas produce an output at position i in their sequence of outputs, then the output is the same;
2) if new commands to be accepted keep coming, then every replica that does not fail keeps producing new outputs.

Here we are concerned with requirement 1.

One way to meet requirement 1 is to have each replica execute requests in a unique total order, *e.g.* like in the Multi-Paxos algorithm. However, sometime the order in which two or more commands are executed does not influence their output and the output of any future commands; we say that such commands commute. By allowing different replicas to execute commuting commands in a different order, an *SMR* algorithm might improve its performance compared to Multi-Paxos. This is our goal with $M2Paxos$.

To take advantage of commutativity, we assume that each command accesses a determined set of objects, modifying those objects' state and computing command output based on those object's state. Observe that in this model, two commands that access disjoint sets of objects commute, and therefore need not be executed in the same order on all replicas.

Instead of enforcing that all replicas execute commands in a unique total order, a replica running $M2Paxos$ maintains one sequence of commands per object (where sequences contain no duplicate commands) called an object-commands map. $M2Paxos$ enforces that, for each object, the sequences of the replicas are all prefixes of the same total order.

Replicas execute commands based on their object-commands map according to the following execution rule. Once a command c appears in all the sequences of the objects it accesses, and all commands comming before c in those sequences have been executed, then c can be executed.

However, ensuring a total order per object is not sufficient to meet requirement 1 because commands accessing overlapping sets of objects must have a consistent order across their common objects.

For example, consider two objects $o1$ and $o2$ and two commands $c1$ and $c2$ that access both objects. Also consider two replicas and a global system state (example 1) in which replica 1 computed the following sequences for object $o1$ and $o2$:

$replica1[o1] = \langle c1, c2 \rangle$
$replica1[o2] = \langle c2 \rangle$

while replica 2 computed the following sequences for object $o1$ and $o2$:

$replica2[o1] = \langle c1 \rangle$
$replica2[o2] = \langle c2, c1 \rangle$

In this situation, for each object, the sequence of the replicas are both prefix of the same total order ($\langle c1, c2 \rangle$ for $o1$ and $\langle c2, c1 \rangle$ for $o2$). However, according to the rule for execution, replica 1 may execute $c1$ before $c2$, while replica 2 may execute $c2$ before $c1$, potentially violating requirement 1.

Also, according to the execution rule, a replica in the following configuration (example 2) can execute neither $c1$ nor $c2$ because $c1$ and $c2$ for a dependency cycle.

$replica2[o1] = \langle c1, c2 \rangle$
$replica2[o2] = \langle c2, c1 \rangle$

$M2Paxos$ therefore enforces an additional property of the per-object sequences that replicas build. Given two commands $c1$ and $c2$ and an object-commands map, we say that $c1$ depends on $c2$ if $c2$ appears before $c1$ in the sequence of any object in the map. We can prevent the situation in example 2 by requiring that the dependency relation built from any replica's object-commands map be acyclic. However, this does not rule at the situation in example 1. To rule it out, define the global object-commands map of the system as mapping an object o to the longest sequence than any replica has in its local map for o. $M2Paxos$ ensures that the dependency relation given by the global object-commands map is acyclic. This guarantees that if replicas follow the execution rule, then every two commands that do not commute will be executed in the same order by all replicas. Observe that in the example above, the global object-commands map has a cycle.

In this TLA module, we formally define the guarantee of $M2Paxos$ that were informally described above.

EXTENDS $Objects$, $Sequences$, $Naturals$, $Maps$, $SequenceUtils$, TLC

INSTANCE $DiGraph$

An object-commands map is well-formed when there are no duplicate commands in any sequence, and if c is in object o's sequence, then o is in the set of objects accessed by c.

$$
\begin{aligned}
&WellFormed(map) \triangleq \\
&\quad \land\ map \in [Objects \rightarrow Seq(Commands \cup \{NotACommand\})] \\
&\quad \land\ \forall\, o \in Objects : NoDup(map[o]) \\
&\quad \land\ \forall\, o \in Objects : \forall\, c \in Image(map[o]) : \\
&\qquad o \in AccessedBy(c)
\end{aligned}
$$

A command $c1$ depends on a command $c2$ if there is an object accessed by both $c1$ and $c2$ for which $c1$ appears before $c2$ in the object's sequence, or $c1$ appears in the object sequence but not $c2$ (which will therefore have to come after $c1$). The dependency relation can be seen as a graph.

$$
\begin{aligned}
&DependencyGraph(map) \triangleq \\
&\quad \text{LET } Vs \triangleq \text{UNION } \{Image(map[o]) : o \in Objects\} \quad \text{Vertices} \\
&\quad\phantom{\text{LET }} Es \triangleq \{e \in Vs \times Vs : \exists\, o \in Objects \quad : \quad \text{Edges} \\
&\qquad\qquad \text{LET } s \triangleq map[o] \text{ IN } \quad \exists\, i \in \text{DOMAIN } s : \\
&\qquad\qquad\quad \lor\ \land\ i \neq Len(s) \\
&\qquad\qquad\quad\ \land\ s[i] = e[1] \\
&\qquad\qquad\quad\ \land\ s[i+1] = e[2] \\
&\qquad\qquad\quad \lor\ \land\ o \in AccessedBy(e[2]) \\
&\qquad\qquad\quad\ \land\ s[i] = e[1] \\
&\qquad\qquad\quad\ \land\ e[2] \notin Image(s)\} \\
&\quad \text{IN } \langle Vs, Es \rangle
\end{aligned}
$$

An object-commands map is correct when its dependency graph has no cycles.

$$MapCorrectness(gm) \triangleq \neg HasCycle(DependencyGraph(gm))$$

The global system state associates to each replica node a local object-commands map.

$$
\begin{aligned}
&IsGlobalState(gs) \triangleq \\
&\quad \forall\, s \in \text{DOMAIN } gs : gs[s] \in [Objects \rightarrow Seq(Commands)]
\end{aligned}
$$

The global object-commands map.

$GlobalMap(gs) \triangleq$
$\quad$ LET $MaxSeq(ss) \triangleq$ CHOOSE $s \in ss : \forall\, t \in ss : Len(t) \leq Len(s)$
$\quad\quad\quad ObjSeqs(o) \triangleq \{s[o] : s \in \{gs[x] : x \in$ DOMAIN $gs\}\}$
$\quad$ IN $[o \in Objects \mapsto MaxSeq(ObjSeqs(o))]$

Correctness of the global state:
1) Every replica has a well-formed local object-commands map;
2) For each object, all replicas agree on a total order of commands;
3) The global object-commands map is acyclic.

$Correctness(gs) \triangleq$
$\quad$ LET $replicas \triangleq$ DOMAIN gs
$\quad$ IN $\land\, \forall\, r \in replicas : WellFormed(gs[r])$
$\quad\quad\quad \land\, \forall\, o \in Objects : \forall\, r1,\, r2 \in replicas :$
$\quad\quad\quad\quad$ LET $s1 \triangleq gs[r1][o]$
$\quad\quad\quad\quad\quad\quad s2 \triangleq gs[r2][o]$
$\quad\quad\quad\quad$ IN $Prefix(s1,\, s2) \lor Prefix(s2,\, s1)$
$\quad\quad\quad \land\, \neg HasCycle(DependencyGraph(GlobalMap(gs)))$

A temporal specification.

CONSTANT $Replica$

VARIABLES $replicaState$

$TypeInvariant \triangleq$
$\quad replicaState \in [Replica \rightarrow [Objects \rightarrow Seq(Commands)]]$

$Init \triangleq replicaState = [r \in Replica \mapsto [o \in Objects \mapsto \langle\rangle]]$

$Next \triangleq \exists\, c \in Commands,\, o \in Objects,\, r \in Replica :$
$\quad \land\, o \in AccessedBy(c)$
$\quad \land\, replicaState' = [replicaState$ EXCEPT $![r] =$
$\quad\quad\quad [@$ EXCEPT $![o] = Append(@,\, c)]]$
$\quad \land\, Correctness(replicaState)'$

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

* Modification History
* Last modified *Mon Jun* 27 11:12:44 *EDT* 2016 by *nano*
* Created *Mon Jun* 06 14:59:29 *EDT* 2016 by *nano*

MODULE *AbstractM2Paxos*

EXTENDS *Objects*, *Maps*, *SequenceUtils*, *Integers*, *FiniteSets*, *TLC*

In this module we describe how *M2Paxos* uses leases on objects to maintain the correctness property of the global object-commands map (defined in the *Correctness* module) while repeatedly increasing the set of commands that can be executed by the replicas. This specification describes at an abstract level how leases and the global object-commands map evolve, without any distributed-system model.

We do not use the temporal part of the *Correctness* module, therefore the substitutions below are just there to make the TLA+ toolbox happy.

$C \triangleq$ INSTANCE *Correctness* WITH $Replica \leftarrow \{\}$, $replicaState \leftarrow \{\}$

The algorithm maintains for each object a sequence of values which are commands or the special value *NotACommand* (the object-values map). Each position in such a sequence is called an instance. The global object-commands map is obtained from the object-values map by truncating every sequence of instances at the first *NotACommand* value encountered, and then removing duplicate commands.

CONSTANT *Instances*

ASSUME $Instances = Nat \setminus \{0\} \lor \exists\, i \in Nat : Instances = 1 \,..\, i$

Truncate a sequence of instances right before the first *NotACommand* value.

RECURSIVE $Truncate(\_)$

$Truncate(vs) \triangleq$

  IF $vs = \langle\rangle \lor Head(vs) = NotACommand$

   THEN $\langle\rangle$

   ELSE $\langle Head(vs)\rangle \circ Truncate(Tail(vs))$

$ObjectCommandsMap(is) \triangleq$

  $[o \in Objects \mapsto RemDup(Truncate(is[o]))]$

$Correctness(is) \triangleq C!MapCorrectness(ObjectCommandsMap(is))$

CONSTANT *LeaseId*

ASSUME $LeaseId \subseteq Nat$

At any moment, an object is part of a unique lease, $lease[o]$. The variable named instances is a map from object to sequence of instances.

VARIABLE *instances*, *lease*

A invariant describing the type of the variables.

$TypeInvariant \triangleq$

  $\land\; instances \in [Objects \rightarrow [Instances \rightarrow Commands \cup \{NotACommand\}]]$

  $\land\; \exists\, D \in \text{SUBSET } Objects : lease \in [D \rightarrow LeaseId]$

$ActiveLeases \triangleq \{l \in LeaseId : \exists\, o \in Objects :$

  $o \in \text{DOMAIN } lease \land lease[o] = l\}$

$LeaseObjects(l) \triangleq \{o \in Objects : o \in \text{DOMAIN } lease \land lease[o] = l\}$

A command c can be assigned to a set of instances $\{i[o] : o \in Objects\}$, one per object it accesses, when:

1) all the objects that c accesses are part of the same lease;
2) $instances[i[o]]$ holds value $NotACommand$ for all object accessed by the command;
3) after the assignement, the object-commands map obtained by restricting the global object-commands map to the objects accessed by c satisfies the correctness condition for object-commands.

This process models a lease owner executing commands on the objects that are part of its lease.

The condition 3 is specified in the definition $LocalCorrectness(\_)$ below, while the full action is specified in $Order(\_)$.

$$
\begin{aligned}
& LocalCorrectness(l) \triangleq \\
& \quad \text{LET } view \triangleq [o \in Objects \mapsto \\
& \qquad\quad \text{IF } o \in LeaseObjects(l) \\
& \qquad\quad \text{THEN } instances[o] \\
& \qquad\quad \text{ELSE } \langle\rangle] \\
& \quad \text{IN } \quad Correctness(view)
\end{aligned}
$$

A lease is safe to break when: for every object o in the lease, instance i, and instance $j < i$, if $instances[o][i]$ holds a command, then $instances[o][j]$ holds a command.

$$
\begin{aligned}
& Safe(l) \triangleq \forall\, o \in LeaseObjects(l) : \forall\, i, j \in Instances : \\
& \qquad i < j \land instances[o][j] \in Commands \Rightarrow instances[o][i] \in Commands
\end{aligned}
$$

The initial state.

$$
\begin{aligned}
& Init \triangleq \\
& \qquad \land\, instances = [o \in Objects \mapsto [i \in Instances \mapsto NotACommand]] \\
& \qquad \land\, lease = \langle\rangle
\end{aligned}
$$

A new lease on the set of objects $objs$ can be acquired only when the existing leases on those objects are safe. Comment-out the first conjunct and model-check to see what happens if we remove the restriction that only safe leases may be broken.

This means that breaking a big lease requires making sure that there are no "holes" in the sequences of values of the objects in the lease.

$$
\begin{aligned}
& Acquire(objs) \triangleq \\
& \qquad \land\ \forall\, l \in ActiveLeases : \\
& \qquad\qquad LeaseObjects(l) \cap objs \neq LeaseObjects(l) \Rightarrow Safe(l) \\
& \qquad \land\ \exists\, l \in LeaseId \setminus ActiveLeases : \\
& \qquad\qquad \text{LET } broken \triangleq \{lease[o] : o \in objs \cap \text{DOMAIN } lease\} \\
& \qquad\qquad\qquad leased \triangleq (\text{DOMAIN } lease \setminus \text{UNION } \{LeaseObjects(l2) : l2 \in broken\}) \\
& \qquad\qquad\qquad\qquad \cup\, objs \\
& \qquad\qquad \text{IN} \\
& \qquad\qquad\quad lease' = [o \in leased \mapsto \text{IF } o \in objs \text{ THEN } l \text{ ELSE } lease[o]] \\
& \qquad \land\ \text{UNCHANGED } instances
\end{aligned}
$$

A command c can be executed if there is a lease on a superset of its accessed objects.

$Exec(c) \triangleq \exists\, l \in ActiveLeases :$

$\quad \land\; AccessedBy(c) \subseteq LeaseObjects(l)$

Choose one free instance per accessed object and update it.

$\quad \land\; \exists\, is \in [AccessedBy(c) \rightarrow Instances] :$

$\quad\quad \land\; \forall\, o \in AccessedBy(c) : instances[o][is[o]] = NotACommand$

$\quad\quad \land\; instances' = [o \in Objects \mapsto$

$\quad\quad\quad\quad \text{IF } o \notin AccessedBy(c) \text{ THEN } instances[o]$

$\quad\quad\quad\quad \text{ELSE } [instances[o] \text{ EXCEPT } ![is[o]] = c]]$

$\quad \land\; \text{UNCHANGED } lease$

Ensure that a lease owner does not create cycles on its own:

$\quad \land\; LocalCorrectness(l)'$

$Next \triangleq$

$\quad \lor\; \exists\, objs \in \text{SUBSET } Objects : Acquire(objs)$

$\quad \lor\; \exists\, c \in Commands : Exec(c)$

$Spec \triangleq Init \land \Box[Next]_{\langle lease,\, instances\rangle}$

$Safety \triangleq Correctness(instances)$

THEOREM $Spec \Rightarrow \Box Safety$

* Modification History

* Last modified *Wed Jun* 29 10:35:01 *EDT* 2016 by *nano*

* Created *Tue Jun* 07 09:31:03 *EDT* 2016 by *nano*

MODULE $M2Paxos2$

An abstract specification of $M2Paxos$. It consists in coordinating several $MultiPaxos$ instances (one per object) using exclusive leases on objects.

$\text{EXTENDS } Sequences, Objects, FiniteSets, Integers, Maps, TLC$

$\text{CONSTANT } Acceptors, Quorums, MaxBallot, MaxInstance, LeaseId$

$\text{ASSUME } LeaseId \subseteq Nat$

$\text{ASSUME } \forall Q \in Quorums : Q \subseteq Acceptors$

$\text{ASSUME } \forall Q1, Q2 \in Quorums : Q1 \cap Q2 \neq \{\}$

Majority quorums.

$$MajQuorums \triangleq \{Q \in \text{SUBSET } Acceptors : Cardinality(Q) > Cardinality(Acceptors) \div 2\}$$

$$Instances \triangleq 1 \,..\, MaxInstance$$

$$Ballots \triangleq 0 \,..\, MaxBallot$$

A proposal is tied to a lease and assigns one instance to each object accessed by the command.

$$Lease(p) \triangleq p[3]$$
$$Command(p) \triangleq p[1]$$
$$Slots(p) \triangleq p[2]$$
$$\begin{aligned} IsProposal(p) \triangleq \\ &\land Command(p) \in Commands \\ &\land Slots(p) \in [AccessedBy(p[1]) \rightarrow Instances] \\ &\land Lease(p) \in LeaseId \end{aligned}$$

VARIABLES
$\quad ballots, votes, leases, proposals$

A ghost variable for the refinement:

$\text{VARIABLE } lease$

$$\begin{aligned} TypeInvariant \triangleq \\ &\land\ ballots \in [Acceptors \rightarrow [Objects \rightarrow Ballots \cup \{-1\}]] \\ &\land\ votes \in [Acceptors \rightarrow [Objects \rightarrow \\ &\qquad [Instances \rightarrow [Ballots \rightarrow Commands \cup \{NotACommand\}]]]] \\ &\land\ \exists L \in (\text{SUBSET } LeaseId) \setminus \{\} : \\ &\qquad \forall l \in L : \exists D \in (\text{SUBSET } Objects) \setminus \{\} : \\ &\qquad\quad \forall o \in D : leases[l][o] \in Ballots \\ &\land\ \forall p \in proposals : IsProposal(p) \\ &\land\ \exists D \in \text{SUBSET } Objects : lease \in [D \rightarrow LeaseId] \end{aligned}$$

Another invariant

$Inv1 \triangleq \forall p \in proposals :$
$\quad \land\ Lease(p) \in \text{DOMAIN } leases$
$\quad \land\ AccessedBy(Command(p)) \subseteq \text{DOMAIN } leases[Lease(p)]$

The initial state.

$Init \triangleq$
$\quad \land\ ballots = [a \in Acceptors \mapsto [o \in Objects \mapsto -1]]$
$\quad \land\ votes = [a \in Acceptors \mapsto [o \in Objects \mapsto$
$\qquad [i \in Instances \mapsto [b \in Ballots \mapsto NotACommand]]]]$
$\quad \land\ leases = \langle\rangle$
$\quad \land\ proposals = \{\}$
$\quad \land\ lease = \langle\rangle$

A command c (or the value Aborted) is chosen in instance i at ballot b for object o if there is a quorum of acceptors that voted for c in instance i and at ballot b of object o.

$ChosenAt(o, i, b, c) \triangleq$
$\quad \exists Q \in Quorums : \forall a \in Q : votes[a][o][i][b] = c$

A command c is chosen in instance i for object o if there is a ballot b such that c is chosen at i and b for object o.

$Chosen(o, i, c) \triangleq$
$\quad \exists b \in Ballots : ChosenAt(o, i, b, c)$

$Executed(c) \triangleq \forall o \in AccessedBy(c) : \exists i \in Instances :$
$\quad Chosen(o, i, c)$

$ExecutedWithLease(c, l) \triangleq \exists Q \in Quorums : \forall a \in Q :$
$\quad \forall o \in AccessedBy(c) : \exists i \in Instances :$
$\qquad votes[a][o][i][leases[l][o]] = c$

A lease is valid if a quorum of acceptors have it locally.

$IsLocalActiveLease(l, a) \triangleq$
$\quad \land\ l \in \text{DOMAIN } leases$
$\quad \land\ \forall o \in \text{DOMAIN } leases[l] : ballots[a][o] = leases[l][o]$

$Active(l) \triangleq \exists Q \in Quorums : \forall a \in Q : IsLocalActiveLease(l, a)$

Create a lease on an arbitrary non-empty set of objects with arbitrary ballots.

$NewLease(l) \triangleq$
$\quad \land\ l \notin \text{DOMAIN } leases$
$\quad \land\ \exists os \in (\text{SUBSET } Objects) \setminus \{\} : \exists bs \in [os \rightarrow Ballots] :$
$\qquad \land\ os \neq \{\}$
A lease own a ballot exclusively:
$\qquad \land\ \forall l2 \in \text{DOMAIN } leases : \forall o \in os :$
$\qquad\quad o \in \text{DOMAIN } leases[l2] \Rightarrow leases[l2][o] \neq bs[o]$
$\qquad \land\ leases' = leases \mathbin{++} \langle l, bs\rangle$

$\wedge$ UNCHANGED $\langle ballots, votes, proposals, lease\rangle$

Accept a new lease l on a set of objects os.

$AcceptLease(a, l) \triangleq$
$\quad \wedge\ l \in$ DOMAIN $leases$
$\quad \wedge\ \forall\, o \in$ DOMAIN $leases[l] : ballots[a][o] < leases[l][o]$
$\quad \wedge\ ballots' = [ballots$ EXCEPT $![a] = [o \in Objects \mapsto$
$\qquad$ IF $o \in$ DOMAIN $leases[l]$
$\qquad$ THEN $leases[l][o]$
$\qquad$ ELSE $ballots[a][o]]]$
$\quad \wedge$ UNCHANGED $\langle votes, proposals, leases\rangle$
$\quad$ A ghost transition:
$\quad \wedge$ IF $Active(l)' \wedge \neg Active(l)$
$\qquad$ THEN
$\qquad\quad$ LET $broken \triangleq \{lease[o] : o \in$ DOMAIN $lease \cap$ DOMAIN $leases[l]\}$
$\qquad\qquad\quad leased \triangleq (($DOMAIN $lease) \setminus ($UNION $\{$DOMAIN $leases[l2] : l2 \in broken\}))$
$\qquad\qquad\qquad \cup$ DOMAIN $leases[l]$
$\qquad\quad$ IN $\quad lease' = [o \in leased \mapsto$
$\qquad\qquad\qquad$ IF $o \in$ DOMAIN $leases[l]$ THEN l
$\qquad\qquad\qquad$ ELSE $lease[o]]$
$\qquad$ ELSE UNCHANGED $lease$

The *Paxos* rule for determining which values are safe to propose.

$SafeValues(Q, o, i) \triangleq$
$\quad$ LET $bals \triangleq \{b \in Ballots : \exists\, a \in Q : votes[a][o][i][b] \in Commands\}$
$\quad$ IN $\quad$ IF $bals \neq \{\}$
$\qquad$ THEN
$\qquad\quad$ LET
$\qquad\qquad maxBal \triangleq Max(bals,$ LAMBDA $x, y : x \leq y)$
$\qquad\qquad maxAcc \triangleq$ CHOOSE $a \in Q : votes[a][o][i][maxBal] \in Commands$
$\qquad\quad$ IN $\{votes[maxAcc][o][i][maxBal]\}$
$\qquad$ ELSE $Commands$

Making a proposal based on lease l.

$Propose(c, l) \triangleq$
$\quad \wedge\ l \in$ DOMAIN $leases$
$\quad \wedge\ \forall\, o \in AccessedBy(c) : o \in$ DOMAIN $leases[l]$
$\quad \wedge\ \forall\, p \in proposals : Lease(p) = l \Rightarrow Command(p) \neq c$
$\quad$ Wait for all other proposals in the same lease to be executed.
$\quad \wedge\ \forall\, p \in proposals : Lease(p) = l \Rightarrow ExecutedWithLease(Command(p), l)$
$\quad$ Choose an instance for every object accessed by c, leaving no holes:
$\quad \wedge\ \exists\, is \in [AccessedBy(c) \rightarrow Instances] :$
$\qquad \wedge\ \forall\, o \in AccessedBy(c) : \forall\, i \in Instances :$
$\qquad\quad \wedge\ i < is[o] \Rightarrow \exists\, c2 \in Commands : Chosen(o, i, c2)$ Leave no gaps.
$\qquad\quad \wedge\ i = is[o] \Rightarrow \exists\, Q \in Quorums : c \in SafeValues(Q, o, i)$ the *Paxos* rule for proposing commands.

$\wedge\ proposals' = proposals \cup \{\langle c, is, l \rangle\}$
$\wedge$ UNCHANGED $\langle ballots, votes, leases, lease \rangle$

The acceptor a can vote for a proposal when its lease is active locally.

$$
\begin{aligned}
&Vote(a) \triangleq \\
&\quad \wedge \exists\, p \in proposals : \\
&\qquad \wedge IsLocalActiveLease(Lease(p), a) \\
&\qquad \wedge votes' = [votes \text{ EXCEPT } ![a] = [o \in Objects \mapsto \\
&\qquad\quad \text{IF } o \in AccessedBy(Command(p)) \\
&\qquad\quad\ \text{THEN } [votes[a][o] \text{ EXCEPT } ![Slots(p)[o]] = \\
&\qquad\qquad [@ \text{ EXCEPT } ![ballots[a][o]] = Command(p)]] \\
&\qquad\quad\ \text{ELSE } votes[a][o]]] \\
&\quad \wedge \text{UNCHANGED } \langle ballots, leases, proposals, lease \rangle
\end{aligned}
$$

$$
\begin{aligned}
&Next \triangleq \\
&\quad \vee\ \exists\, l \in LeaseId : NewLease(l) \\
&\quad \vee\ \exists\, a \in Acceptors, l \in LeaseId : AcceptLease(a, l) \\
&\quad \vee\ \exists\, c \in Commands : \exists\, l \in LeaseId : Propose(c, l) \\
&\quad \vee\ \exists\, a \in Acceptors : Vote(a)
\end{aligned}
$$

$$Spec \triangleq Init \wedge \Box[Next]_{\langle ballots, votes, proposals, leases, lease \rangle}$$

A tenative refinement to $AbstractM2Paxos$. See below for why it cannot work.

$$
\begin{aligned}
&AInstances \triangleq [o \in Objects \mapsto [i \in Instances \mapsto \\
&\qquad IfExistsElse(Commands, \text{LAMBDA } c : Chosen(o, i, c), NotACommand)]]
\end{aligned}
$$

$$
\begin{aligned}
&A \triangleq \text{INSTANCE } AbstractM2Paxos \text{ WITH} \\
&\quad instances \leftarrow AInstances, \\
&\quad lease \leftarrow lease
\end{aligned}
$$

This property does not hold because a quorum of votes can form after some members of the quorum departed from the corresponding lease. To fix that, we would need to track leases by instance, and not only by object. We could also introduce a prophecy variable . . .

THEOREM $Spec \Rightarrow A!Spec$ Wrong!

This property should hold.

THEOREM $Spec \Rightarrow A!Safety$

* Modification History
* Last modified *Wed Jun 29 10:39:27 EDT* 2016 by *nano*
* Created *Mon Jun 06 13:48:20 EDT* 2016 by *nano*

MODULE *Objects*

This module defines the constants *Commands*, the set of commands, Object, the set of objects that commands may access, and $AccessedBy(\_)$, where $AccessedBy(c)$ is the set of objects accessed by the command c.

EXTENDS *Misc*

CONSTANTS $Commands,\ AccessedBy(\_),\ Objects$

ASSUME $\forall\, c \in Commands : AccessedBy(c) \in \text{SUBSET } Objects$

$NotACommand \triangleq \text{CHOOSE } x : x \notin Commands$

* Modification History
* Last modified *Fri Jun* 24 13:20:03 *EDT* 2016 by *nano*
* Created *Wed Nov* 18 23:02:07 *EST* 2015 by *nano*

MODULE *DiGraph*

A few notions related to directed graphs.

EXTENDS $FiniteSets$, $Sequences$, $Naturals$, $Misc$, $SequenceUtils$

A digraph is a set of vertices and a set of edges, where an edge is a pair of vertices.

$Vertices(G) \triangleq G[1]$

$Edges(G) \triangleq G[2]$

$IsDigraph(G) \triangleq Edges(G) \subseteq (Vertices(G) \times Vertices(G))$

True when there exists a path from $v1$ to $v2$ in the graph G

RECURSIVE $DominatesRec(\_, \_, \_, \_)$
$Dominates(v1, v2, G) \triangleq$
$\quad DominatesRec(v1, v2, G, \{\})$

Recursive implementation of $Dominates(v1, v2, G)$.

$DominatesRec(v1, v2, G, acc) \triangleq$
$\quad \lor \langle v1, v2 \rangle \in Edges(G)$
$\quad \lor \exists\, v \in Vertices(G):$
$\quad\quad \land \neg v \in acc$
$\quad\quad \land \langle v1, v \rangle \in Edges(G)$
$\quad\quad \land DominatesRec(v, v2, G, acc \cup \{v1\})$

$HasCycle(G) \triangleq$
$\quad \exists\, v1, v2 \in Vertices(G):$
$\quad\quad \land\ v1 \neq v2$
$\quad\quad \land\ Dominates(v1, v2, G)$
$\quad\quad \land\ Dominates(v2, v1, G)$

* Modification History
* Last modified *Wed Jun* 22 16:56:37 *EDT* 2016 by *nano*
* Created *Tue Jul* 28 03:10:02 *CEST* 2015 by *nano*

MODULE *Maps*

Adding a key-value mapping ($kv[1]$ is the key, $kv[2]$ the value) to a map

$f ++ kv \triangleq [x \in \text{DOMAIN } f \cup \{kv[1]\} \mapsto \text{IF } x = kv[1] \text{ THEN } kv[2] \text{ ELSE } f[x]]$

The image of a map

$Image(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$IsBijection(f, X, Y) \triangleq$

$\quad \land \text{DOMAIN } f = X$

$\quad \land Image(f) = Y$

$\quad \land \forall x, y \in X : x \neq y \Rightarrow f[x] \neq f[y]$

$\quad \land \forall y \in Y : \exists x \in X : f[x] = y$

$IsInjective(f) \triangleq \forall x, y \in \text{DOMAIN } f : x \neq y \Rightarrow f[x] \neq f[y]$

* Modification History
* Last modified *Wed Jun* 08 11:17:10 *EDT* 2016 by *nano*
* Created *Mon* May 02 21:01:30 *EDT* 2016 by *nano*

MODULE *SequenceUtils*

```
EXTENDS Sequences, Maps, Naturals, Misc

IsIncreasing(f) ≜
    ∀ x, y ∈ DOMAIN f : x ≤ y ⇒ f[x] ≤ f[y]

IsSubSequence(s1, s2) ≜
    ∃ f ∈ [DOMAIN s1 → DOMAIN s2] :
        ∧ IsInjective(f)
        ∧ IsIncreasing(f)
        ∧ ∀ i ∈ DOMAIN s1 : s1[i] = s2[f[i]]

Last(s) ≜ s[Len(s)]

 Sequences with no duplicates:
RECURSIVE NoDupRec(_, _)
NoDupRec(es, seen) ≜
    IF es = ⟨⟩
     THEN TRUE
     ELSE
        IF es[1] ∈ seen
         THEN FALSE
         ELSE NoDupRec(Tail(es), seen ∪ {es[1]})

NoDup(es) ≜
  NoDupRec(es, {})

NoDupSeq(E) ≜
  {es ∈ Seq(E) : NoDup(es)}

 Removing duplicates from a sequence:
RECURSIVE RemDupRec(_, _)
RemDupRec(es, seen) ≜
  IF es = ⟨⟩
   THEN ⟨⟩
   ELSE
    IF es[1] ∈ seen
     THEN RemDupRec(Tail(es), seen)
     ELSE ⟨es[1]⟩ ∘ RemDupRec(Tail(es), seen ∪ {es[1]})
RemDup(es) ≜ RemDupRec(es, {})

 Sequence prefix:
Prefix(s1, s2) ≜
    ∧ Len(s1) ≤ Len(s2)
    ∧ ∀ i ∈ DOMAIN s1 : s1[i] = s2[i]

 The longest common prefix of two sequences:
```

RECURSIVE $LongestCommonPrefixLenRec(\_, \_, \_)$

$$
\begin{array}{l}
LongestCommonPrefixLenRec(S, n, e1) \triangleq \\
\quad \text{IF } S = \{\} \\
\quad\;\; \text{THEN } 0 \\
\quad\;\; \text{ELSE} \\
\quad\quad\quad \text{IF } \land \forall e \in S : Len(e) \geq n + 1 \\
\quad\quad\quad\quad\;\; \land \forall e \in S : e[n + 1] = e1[n + 1] \\
\quad\quad\quad\;\; \text{THEN } LongestCommonPrefixLenRec(S, n + 1, e1) \\
\quad\quad\quad\;\; \text{ELSE } n
\end{array}
$$

$$LongestCommonPrefixLenSet(S) \triangleq LongestCommonPrefixLenRec(S, 0, Some(S))$$

$$
\begin{array}{l}
LongestCommonPrefix(S) \triangleq \\
\quad \text{LET } n \triangleq LongestCommonPrefixLenSet(S) \\
\quad \text{IN } \quad \text{IF } n = 0 \\
\quad\quad\quad\;\; \text{THEN } \langle\rangle \\
\quad\quad\quad\;\; \text{ELSE } [i \in 1 \,..\, LongestCommonPrefixLenSet(S) \mapsto Some(S)[i]]
\end{array}
$$

* Modification History

* *Last* modified *Wed Jun* 08 11:28:55 *EDT* 2016 by *nano*

* Created *Wed Jun* 08 11:11:31 *EDT* 2016 by *nano*

MODULE *Misc*

EXTENDS *Naturals*

$Some(S) \triangleq$ CHOOSE $e \in S$: TRUE

All sequences of elements of X which have a length smaller or equal to b.

$BSeq(X, b) \triangleq \{\langle\rangle\} \cup$ UNION $\{[1 \,..\, n \rightarrow X] : n \in 1 \,..\, b\}$

$Min(i, j) \triangleq$ IF $i < j$ THEN i ELSE j

$Max(S, LessEq(\_, \_)) \triangleq$ CHOOSE $e \in S : \forall\, e1 \in S : LessEq(e1, e)$

$IfExistsElse(S, P(\_), d) \triangleq$ IF $\exists\, x \in S : P(x)$ THEN CHOOSE $x \in S : P(x)$ ELSE d

* Modification History
* Last modified *Fri Jun* 24 14:27:44 *EDT* 2016 by *nano*
* Created *Thu Feb* 04 16:55:11 *EST* 2016 by *nano*