MODULE *BlueRep*

Adapted from *Ryan*'s spec.

Replication protocol with a constant number of replicas. In the stable state, there is one primary replica, to which all requests should be sent. Reads are handled locally by the primary and do not involve the other replicas. To process a write, the primary writes the new value to its disk and then sends *RPC* write requests to the other replicas. The number of replicas that must report success in order for the primary to report success to the client is configurable.

EXTENDS *MCPrimSecRep*

VARIABLES

CONSTANTS *IPrim*, initial primary replica (an element of *Rep*)
MinOk, min number of successful replica writes for write to succeed
HdlReconciliation(_, _), * (*o*, *v*) – handles reconcilliation at *MC* level
BObject blue object type

$NoVal \triangleq$ CHOOSE $v : v \notin Val$

$Blue\_ErrorTypes \triangleq \{$"error", error at secondary while writing
"badver"$\}$ primary has the bad version

what do I do *w*/ staled???
$Blue\_DataType \triangleq Val \cup \{NoVal\}$ only one address!!!
$Blue\_WriteType \triangleq Val \cup \{NoVal\}$ only one address supported
$Blue\_NoWrite \triangleq NoVal$

$Blue\_InitDataVal \triangleq NoVal$
$Blue\_InitPrim(o) \triangleq IPrim$
$Blue\_InitSec(o) \triangleq Rep \setminus \{IPrim\}$ all the other reps are secondaries

$Blue\_PrepareWr(r, o, w) \triangleq w$

$Blue\_TypeInvariant \triangleq$
$\quad \wedge TypeInvariant$
$Blue\_Init \triangleq$
$\quad \wedge MCInit$

MASTER functioning

master decrees that read or modify the global state. They are always sent from a replica in the system, which must be alive.

$gsm \triangleq master \setminus *$ renamed

A primary replica reports that one of its secondaries is dead.

$PrimReportsDeath(o, reporter, reportee) \triangleq$ primary reports that a secondary is dead
$\quad \wedge reporter = cache[reporter][o].prim$ reporter thinks it's the primary
$\quad \wedge reporter = master.objects[o].prim$ master also thinks reporter is the primary
$\quad \wedge reportee \in cache[reporter][o].sec$ primary thinks the *reportee* is a secondary

$\land$ $reportee \in master.objects[o].sec$ the master knows *reportee* is among the living secondaries

$\land$ $master' = [master$ EXCEPT $!.health[reportee] =$ "dead", *reportee* is dead
$!.objects[o].sec = @ \setminus \{reportee\}$, no longer a *sec*
$!.objects[o].version = @ + 1]$ get on to the next *version*!

$\land$ $cache' = [cache$ EXCEPT $![reporter][o] = master'.objects[o]]$ refresh cache at end of protocol
$\land$ UNCHANGED $\langle data, stat, channel, resps \rangle$
$\land$ $NoReply$

Before a new primary can become a prim and serve reads/writes, it must first reconcile *w/* the other replicas. This is modeled simply as another write request, which is started by the new primary, *w/* the value of its local disk. This action will block the new prim. This write will suffer the same possible failures. If this write fails, then the replicas who've failed are removed. If this new prim is removed by another replica, then that replica will do the same thing.

$Reconcile(o, newp) \triangleq$ A new primary reconciles *w/* the other replicas
$\land$ $PrimaryContinuesWrite(newp, o, data[newp][o], \{cache'[newp][o].prim\} \cup cache'[newp][o].sec, cache'[$

A secondary replica reports that its primary replica is dead. This action will only be enabled if *reportee* is indeed the primary replica. Because I don't model read leases, I ensure that a primary can only be declared dead when it realy is dead. Because I already ensure that no replica can be stale at end of write, the master agrees *w/* making any secondary the new primary.

$PrimReportedDead(o, reporter, reportee) \triangleq$
$\land$ $stat[reportee].phase =$ "dead" * ASSUMPTION : ensure primary is not falsely declared dead

$\land$ $master.health[reporter] =$ "alive" master must think *sec* is alive
$\land$ $reporter \in master.objects[o].sec$ reporter should be a live secondary
$\land$ $reportee = cache[reporter][o].prim$ the *sec* must believe reporter is primary
$\land$ $reportee = master.objects[o].prim$ master thinks reported replica is primary

$\land$ $master' = [master$ EXCEPT $!.health[reportee] =$ "dead",
$!.objects[o] = [prim \mapsto reporter,$ make reporter the primary
$sec \mapsto @.sec \setminus \{reporter\},$ reporter is no longer a *sec*
$version \mapsto @.version + 1]]$ get on to the next version

$\land$ $cache' = [cache$ EXCEPT $![reporter][o] = master'.objects[o]]$
refresh cache at end of protocol
$\land$ $Reconcile(o, reporter)$ reconcile new primary with all the rest of the replicas
$\land$ UNCHANGED $\langle stat, channel, resps \rangle$

A replica detects that another replica is dead, which can happen for any reason (or no reason at all). This simulates a replica detecting another replica dead based on heartbeats. Aside from that, allowing death detection to occur at any time is a conservative way to model and check all possible paths of execution. This decree is split into two cases, which are handled above.

$\_DeathDecree(o, reporter, reportee) \triangleq$ reporter reports that *reportee* is dead
$\land$ $reporter \neq reportee$
$\land$ $master.health[reportee] =$ "alive" the *reportee* must have been believed alive
$\land$ $master.objects[o].version = cache[reporter][o].version$ reporter must be up-to-date

```
    ∧ ∨ PrimReportsDeath(o, reporter, reportee)
      ∨ PrimReportedDead(o, reporter, reportee)
DeathDecree ≜
    ∃ o ∈ BObject :
      ∃ reporter ∈ Rep, reportee ∈ Rep :
          ∨ _DeathDecree(o, reporter, reportee)

Blue_MasterActions ≜ DeathDecree
```

REPLICAS

Store an update. If the advertised version is older then the local vers reply *w/* badversion and do not update store. Otherwise, update store & reply *w/ ok*.

```
Blue_StoreUpdate(r, o, w, version) ≜
    data' = [data EXCEPT ![r][o] = w]

SelectSubset(set, func(_)) ≜
  LET F[s ∈ SUBSET (set)] ≜
        IF s = {} THEN {}
         ELSE IF ∃ r ∈ s : func(r) THEN
                  LET r ≜ CHOOSE r ∈ s : func(r) IN
                    F[s \ {r}] ∪ {r}
              ELSE {}
  IN  F[set]
```

The primary kills all stale replicas who've failed to perform write. This action is only enabled if the primary hasn't received any *badver*. The primary goes to the master and requests that the failed resp be killed. The prim can only remove secondaries, not itself. The action is not enabled when ONLY the primary has failed writing. When this happens, the primary will eventually get removed by some replica. Until this happens, though, the primary cannot respond to new writes/reads because it's "busy" and it will never finish the current write.

```
_PrimaryKillsFailedRep(reporter, o) ≜
  LET WriteFinished ≜
          ∀ r ∈ Rep :
              ∧ resps[reporter][o][r] ∉ {"waiting", "badver"}
      ExistReplicasToKill ≜
          ∧ ∃ r ∈ Rep :  exists a replica that should be killed
              ∧ r ≠ reporter  the reporter cannot kill itself
              ∧ resps[reporter][o][r] ∉ {"ok", "n/a"}  r has failed (timeout/error)

      SelectOk(r) ≜ resps[reporter][o][r] = "ok"
  IN
    ∧ reporter = cache[reporter][o].prim  reporter thinks it's the primary
    ∧ stat[reporter].lock[o] = "busy"  reporter is mid-write
    ∧ WriteFinished  have responses from all live replicas, and all agree on version number
    ∧ ExistReplicasToKill  there exist some replicas to kill

     Go to master and kill the replicas that have failed
```

```
    ∧ master.objects[o].version = cache[reporter][o].version  master must agree on version number
    ∧ master.objects[o].prim = reporter  master believes reporter is prim (redundant)

    ∧ master' = [master EXCEPT
                    !.health = [r ∈ Rep ↦ IF resps[reporter][o][r] ≠ "ok"
                                           THEN "dead" ELSE @[r]],
                    !.objects[o].version = @ + 1,
                    !.objects[o].sec = SelectSubset(@, SelectOk)]
    ∧ resps' = [resps EXCEPT ![reporter][o] = [r ∈ Rep ↦ IF @[r] ≠ "ok"
                                                          THEN "n/a" ELSE @[r]]]  mark as killed
    ∧ cache' = [cache EXCEPT ![reporter][o] = master'.objects[o]]
    ∧ UNCHANGED ⟨stat, data, channel⟩
PrimaryKillsFailedRep ≜
  ∃ r ∈ Rep, o ∈ BObject : _PrimaryKillsFailedRep(r, o)
```

A helper function used to count the number of replicas that have successfully completed their part of a write request. Given a function with a subset of replicas as its domain, it counts the number of replicas that the function takes to "*ok*".

```
NumOk(func) ≜
  LET F[d ∈ SUBSET Rep] ≜
        IF ∃ r ∈ d : func[r] = "ok"
         THEN 1 + F[d \ {CHOOSE r ∈ d : func[r] = "ok"}]
         ELSE 0
  IN
    F[DOMAIN func]
```

R Replies to the client after the write is finished. A write can finish only when:
- all replicas have responded (no "waiting" *resps*) (checked in *PrimSecRep*)
- no replica has responded *w/ BadVer*
- no "error" in *resps* remains (all of those that were "error" were "killed")

```
Blue_ReplyFinishedWrite(r, o, w) ≜
  LET NoBadVersion ≜   no replica replied badver
        ∀ r1 ∈ Rep : resps[r][o][r1] ≠ "badver"
      NoStaleReplica ≜   no stale replica has remained
                         stale means that a replica has failed to commit the write
        ∀ r1 ∈ Rep : resps[r][o][r1] ∈ {"ok", "n/a"}

      Success ≜   the write succeeds if it has been committed to at least MinOk reps
        NumOk(resps[r][o]) ≥ MinOk
  IN
    ∧ NoBadVersion   r cannot finish a write if it received a bad version
    ∧ NoStaleReplica   no slate replicas have remaining

    ∧ IF Success THEN   success
        Reply(r, "wrFinished", [ack ↦ "ok", object ↦ o, w ↦ w])
      ELSE   failure
        Reply(r, "wrFinished", [ack ↦ "error", object ↦ o, w ↦ w])
```

Perform the read on the local store of replica r.

$Blue\_StoreRead(r,\ o) \triangleq$
$[ack \mapsto \text{“ok”},\ object \mapsto o,\ val \mapsto data[r][o]]$

$Blue\_ReplicaActions \triangleq$
$ReplicaActions \lor PrimaryKillsFailedRep$

CLIENT actions.

Read-from-primary policy of Blue.

$Blue\_CliRead(o) \triangleq$
$\exists\, r \in Rep :$
$\land\ master.health[r] = \text{“alive”}$ ASSUMPTION : r is not a stale primary. This is ensured in the real protocol via the *ReadLease*. But I don't model read leases here.
$\land\ stat[r].phase = \text{“alive”}$
$\land\ r = cache[r][o].prim$ Read-from-primary policy
$\land\ stat[r].lock[o] = \text{“rdy”}$ no writes in-progress at r
$\land\ \_CliRead(r,\ o)$

$Blue\_ClientActions \triangleq$
$\exists\, o \in Object :$
$\lor \exists\, w \in WriteType : CliWrite(o,\ w)$
$\lor Blue\_CliRead(o)$

$Blue\_Next \triangleq$
$\lor Blue\_MasterActions$
$\lor TimeActions$
$\lor Blue\_ReplicaActions$
$\lor Blue\_ClientActions$

$Blue\_Impl \triangleq Blue\_Init \land \Box[Blue\_Next]_{mcvars}$

THEOREM $Blue\_Impl \Rightarrow \Box Blue\_TypeInvariant$