```
──────────────────────────── MODULE PrimSecRep ────────────────────────────
EXTENDS Util, TLC
───────────────────────────────────────────────────────────────────────────

CONSTANTS  Values:
           Rep,             replicas
           Val,

            **HOLES:
            Types:
           Object,          object IDs or addresses
           DataType,        data type
           WriteType,       write structure (the one that gets written to the data)
           ErrorTypes,      different errors returned by the secondaries

            Abstract methods ("holes") :

           StoreUpdate(_, _, _),    (r,o,w) perform write w ( ∈ WriteType)
                                  on local data of object o of replica r
           StoreRead(_, _),   read(r, o) peform local read at replica r
                                  of object r. Returns the value
           PrepareWr(_, _, _),    prepareWr(r, o, w) prepares write w to send out
                                    to secondaries. This preparation is done by the primary.
           ReplyFinishedWrite(_, _, _),   (r, o, w) replica r replies to a finished write on object o
           GetWriteResponse(_, _, _, _, _),   (r,o,version,sender) returns the response that should be sent out
                                          In case of some error (e.g., wrong vers), should return [type ↦ "noResp"],
                                          in which case nothing will be committed to disk or sent out


           Reply(_, _, _),              (r, type, reply) reply to client
           NoReply,      no reply is sent yet, so just keep the aux variables unchanged
           InitDataVal,     initial data value for the disks
           InitPrim(_),     InitPrim(o) =  value of the initial primary of object o
           InitSec(_),      initial secondaries for object o

           NoRep,
           Client,
           NoWrite

VARIABLES master,           reliable master state

          data,             data at each replica
          cache,            cache of the master state at each replica
          stat,             state of each replica + object locks of each replica
          channel,          in-channel of each replica
          resps             write acknowledgements for each replica from the others

vars  ≜  ⟨master, cache, data, channel, resps, stat⟩
```

$NoRep \triangleq \textsc{choose}\ v : v \notin Rep$
$Client \triangleq \textsc{choose}\ v : v \notin Rep \land v \neq NoRep$
$NoWrite \triangleq \textsc{choose}\ v : v \notin WriteTypes$

$WrReqMsg \triangleq$ Write requests
$\quad [type : \{\text{"wrReq"}\},\ sender : Rep \cup \{Client\},\ object : Object,$
$\quad\ w : WriteType,\ version : Nat]$
$WrRespMsg \triangleq$ Write responses
$\quad [type : \{\text{"wrResp"}\},\ sender : Rep,\ object : Object,\ ack : (\{\text{"ok"}\} \cup ErrorTypes),$
$\quad\ w : WriteType,\ version : Nat]$
$Messages \triangleq$ Message types
$\quad WrReqMsg \cup WrRespMsg$

$TypeInvariant \triangleq$
$\quad \land\ Print(\text{"1"}, \textsc{true})$
$\quad \land\ master \in$
$\qquad\quad [objects : [Object \rightarrow [version : Nat,\ prim : Rep \cup \{NoRep\},\ sec : \textsc{subset}\ (Rep)]],$
$\qquad\quad\ health : [Rep \rightarrow \{\text{"dead"}, \text{"alive"}\}]]$

$\quad \land\ Print(\text{"2"}, \textsc{true})$
$\quad \land\ data \in [Rep \rightarrow [Object \rightarrow DataType]]$
$\quad \land\ Print(\text{"3"}, \textsc{true})$
$\quad \land\ cache \in [Rep \rightarrow [Object \rightarrow [version : Nat,\ prim : Rep \cup \{NoRep\},\ sec : \textsc{subset}\ (Rep)]]]$
$\quad \land\ Print(\text{"5"}, \textsc{true})$
$\quad \land\ channel \in [Rep \rightarrow Seq(Messages)]$

$\quad \land\ Print(\text{"7"}, \textsc{true})$
$\quad \land\ stat \in [Rep \rightarrow [phase : \{\text{"alive"}, \text{"dead"}\},\ lock : [Object \rightarrow \{\text{"rdy"}, \text{"busy"}\}],$
$\qquad\qquad\qquad in\_progress : [Object \rightarrow [val : WriteType,\ version : Nat]]]]$

$\quad \land\ Print(\text{"8"}, \textsc{true})$
$\quad \land\ resps \in [Rep \rightarrow [Object \rightarrow [Rep \rightarrow \{\text{"ok"}, \text{"waiting"}, \text{"timeout"}, \text{"n/a"}\} \cup ErrorTypes]]]$
$\qquad\qquad$ separate set of responses per object
$\quad \land\ Print(\text{"9"}, \textsc{true})$
$Init \triangleq$
$\quad \land\ master =$
$\qquad [objects \mapsto [o \in Object \mapsto$
$\qquad\qquad [version \mapsto 0,\ prim \mapsto InitPrim(o),\ sec \mapsto InitSec(o)]],$
$\qquad\ health \mapsto [r \in Rep \mapsto \text{"alive"}]]$ but all replicas are alive

$\quad \land\ cache = [r \in Rep \mapsto [o \in Object \mapsto$
$\qquad\qquad [version \mapsto 0,\ prim \mapsto InitPrim(o),\ sec \mapsto InitSec(o)]]]$

$\quad \land\ data = [r \in Rep \mapsto [o \in Object \mapsto InitDataVal]]$
$\quad \land\ channel = [r \in Rep \mapsto \langle\rangle]$ no message yet
$\quad \land\ resps = [r \in Rep \mapsto [o \in Object \mapsto [r1 \in Rep \mapsto \text{"waiting"}]]]$

```
    $\land\ stat = [r \in Rep \mapsto [phase \mapsto \text{“alive”},\ lock \mapsto [o \in Object \mapsto \text{“rdy”}],$
                                 $in\_progress \mapsto [o \in Object \mapsto [val \mapsto NoWrite,\ version \mapsto 0]]]]$
                                 all reps alive and ready to accept updates
```

MASTER functioning

In the most general case, the master does not participate actively. It's only for mitigation and for a reliable source of accurate information. In master-monitored replicated systems (like *GFS*), the master is active and monitors replicas, releases primary leases, removes replicas, adds replicas, etc. In self-monitoring replica sys, (like Blue), master is not active in the spec.

$MasterActions \triangleq \text{FALSE}$

TIME AND CHANNELS: (sources of timeouts and losses)

The channel to replica r loses one of the messages that should have arrived there.

```
$\_TransLoss(r) \triangleq$
    $\land\ channel[r] \neq \langle\rangle$  the channel is not empty

    $\land\ channel' = [channel\ \text{EXCEPT}\ ![r] = Tail(@)]$
    $\land\ \text{UNCHANGED}\ \langle master,\ cache,\ data,\ stat,\ resps\rangle$
$TransLoss \triangleq \exists\, r \in Rep : \_TransLoss(r)$
```

(Ex-)Primary r times out waiting for response from replica s.

```
$\_Timeout(r,\ o,\ s) \triangleq$
  LET $updateMsgIsLost(chan,\ update) \triangleq$
          $\forall\, i \in 1 \mathrel{..} Len(chan) : chan[i].w \neq update.val$
  IN
    $\land\ stat[r].phase = \text{“alive”}$  $r$ is alive
    $\land\ stat[r].lock[o] = \text{“busy”}$  $r$ is indeed waiting for responses to a write request
    $\land\ resps[r][o][s] = \text{“waiting”}$  $r$ is still waiting for response from $s$

     $\land\ updateMsgIsLost(channel[s],\ stat[r].in\_progress[o])$ \* REDUCE SPACE: the update message sent to $s$ was lost – TODO

    $\land\ resps' = [resps\ \text{EXCEPT}\ ![r][o][s] = \text{“timeout”}]$  response of $s$ timed out
    $\land\ \text{UNCHANGED}\ \langle master,\ cache,\ data,\ stat,\ channel\rangle$
$Timeout \triangleq$
   $\exists\, r \in Rep,\ o \in Object,\ s \in Rep :$
       $\land\ r \neq s$
       $\land\ \_Timeout(r,\ o,\ s)$

$TimeActions \triangleq$
   $\lor\ TransLoss$         $\lor\ Timeout$
```

REPLICA actions.

Replica r suddenly dies.

```
$\_ReplicaDeath(r) \triangleq$
    $\land\ stat[r].phase = \text{"alive"}$   replica used to be alive

    $\land\ stat' = [stat\ \text{EXCEPT}\ ![r].phase = \text{"dead"}]$   declares itself dead
    $\land\ \text{UNCHANGED}\ \langle master, cache, data, channel, resps \rangle$
$ReplicaDeath \triangleq \exists r \in Rep : \_ReplicaDeath(r)$
```

Replica r updates its cache $w/$ the accurate version from master. One could imagine multiple times when this could be triggered in the actual implem of, *e.g.*, *GFS* (*e.g.*, during *HeartBeat* protocol, after a client comes $w/$ a higher $version \neq$, etc.). Since I don't know what happens in the real protocol exactly, I will assume it can happen anytime.

```
$\_ReadVersion(r, o) \triangleq$
    $\land\ stat[r].phase \neq \text{"dead"}$   replica is not dead yet
    $\land\ cache[r][o].version \neq master.objects[o].version$   $r$'s cache is out-of-date

    $\land\ cache' = [cache\ \text{EXCEPT}\ ![r][o] = master.objects[o]]$   cache new version
    $\land\ \text{UNCHANGED}\ \langle master, data, channel, resps, stat \rangle$
$ReadVersion \triangleq$
    $\land\ \exists r \in Rep, o \in Object : \_ReadVersion(r, o)$
```

Replica r drops a write req w. The reasons might be various. (*E.g.*, in *GFS*, one reason can be that prim cannot find the data associated $w/$ this request in its *LRU*).

```
$DropWrite(r, o, w) \triangleq$   changes $\langle data, channel, stat, resps \rangle$
    $\land\ \text{UNCHANGED}\ \langle data, channel, stat, resps \rangle$
```

Primary continues $w/$ a client request. It pushes the data to its local store (always considered to succeed from this at this point, b/c anyway I provide for general request dropping). The primary then sends update messages to all secondaries. The write needs preparation before it's given out to secondaries. In *GFS* (*e.g.*), the preparation means deciding on an address to write at and setting this adr in the msg.

```
$PrimaryContinuesWrite(r, o, w, allreplicas, version) \triangleq$   changes $\langle data, channel, stat, resps \rangle$, sends $Reply$
    $\land\ \text{LET}$
        $prepared\_wr \triangleq PrepareWr(r, o, w)$
        $wr\_req \triangleq [type \mapsto \text{"wrReq"}, sender \mapsto r, object \mapsto o,$
                    $version \mapsto version, w \mapsto prepared\_wr]$
                        the write needs to be prepared for the secondaries
      $\text{IN}$
        $\land\ \text{IF}\ prepared\_wr \neq NoWrite\ \text{THEN}$   No error while preparing

            $\land\ StoreUpdate(r, o, prepared\_wr)$   perform the update locally
            $\land\ channel' = [r1 \in Rep \mapsto$   send the update req to secondaries
                $\text{IF}\ r1 \in allreplicas \setminus \{r\}\ \text{THEN}$
                    $channel[r1] \circ \langle wr\_req \rangle$
                 $\text{ELSE}\ channel[r1]]$   not a secondary

            $\land\ stat' = [stat\ \text{EXCEPT}\ ![r].lock[o] = \text{"busy"},$   the replica becomes busy
                                                    and will not accept further updates on object $o$
                                                    until this write finishes.
```

```
                                  ![r].in_progress[o] = [val ↦ prepared_wr, version ↦ version]]
                                     maintain the write, for future reference
          ∧ resps′ = [resps EXCEPT ![r][o] = [r1 ∈ Rep ↦
                              IF r = r1 THEN "ok"  this replica has already answered
                               ELSE IF r1 ∈ allreplicas
                               THEN "waiting"  wait for ack from secondaries
                               ELSE "n/a"]]  don't wait for ack from non-secondaries
          ∧ NoReply  don't return yet the response

          ∧ UNCHANGED data

         ELSE   Preparing the message resulted in error, indicating
                that this write shouldn't go ahead.
                Skip it and announce the client of error
            ∧ UNCHANGED ⟨data, channel, stat, resps⟩
            ∧ Reply(r, "NoWrite", prepared_wr)
```

Primary r handles a write request. Note that the writes are blocking and processed by one primary one at a time. Only after a write finishes, does the primary start another one.

```
PrimaryWrite(r, o, w, ver) ≜   changes ⟨data, channel, stat, resps⟩, sends Reply
   ∧ cache[r][o].prim = r        r believes itself to be the primary
   ∧ stat[r].lock[o] = "rdy"     the primary doesn't have other writes in progress on object o

   ∧  ∨ ∧ PrimaryContinuesWrite(r, o, w, {cache[r][o].prim} ∪ cache[r][o].sec, cache[r][o].version)
                                     EITHER: perform the write

      ∨ ∧ DropWrite(r, o, w) \* OR: drop the request alltogether
        ∧ Reply(r, "wrFinished", [ack ↦ "error", object ↦ o, w ↦ w])

VersionBased_GetWriteResponse(r, o, w, version, sender) ≜
   IF (cache[r][o].version ≤ version) THEN   good version
     [type ↦ "wrResp", sender ↦ r, object ↦ o,
      w ↦ w, ack ↦ "ok", version ↦ cache[r][o].version]
    ELSE   bad version
     [type ↦ "wrResp", sender ↦ r, object ↦ o,
      w ↦ w, ack ↦ "badver", version ↦ cache[r][o].version]
```

Secondary continues the write request from prim. It pushes the data to its local store (always considered to succeed from this at this point, b/c anyway I provide for general request dropping & version has been or doesn't need to be checked.

```
SecondaryContinuesWrite(r, o, w, ver, sender) ≜   changes ⟨data, channel⟩
      LET
           wr_resp ≜ GetWriteResponse(r, o, w, ver, sender)
     IN
        IF wr_resp.type = "wrResp" THEN
              ∧ wr_resp.ack = "ok" ⇒ StoreUpdate(r, o, w)  store the update persistenly locally
              ∧ wr_resp.ack ≠ "ok" ⇒ UNCHANGED data  don't store, there's an error
```

```
                ∧ channel′ = [channel EXCEPT
                                ![sender] = @ ∘ ⟨wr_resp⟩,        send response
                                ![r] = Tail(@)]  remove from channel
          ELSE  ∧ channel′ = [channel EXCEPT ![r] = Tail(@)]  no response should be sent back
                ∧ UNCHANGED data
```

A secondary replica processes a write request from primary. The secondary fully executes it and sends a reply back to sender. We don't simulate dropping of the request by *sec.* because there's no need to, given the general *TransLoss*.

```
SecondaryWrite(r, o, w, ver, sender) ≜   changes ⟨data, channel, stat, resps⟩
  ∧ ∨ SecondaryContinuesWrite(r, o, w, ver, sender)

      No need to drop the write at the secondaries. This drop will happen due to
      TransLoss().
      ∨ DropWrite(r, o w) \* OR: drop the request alltogether
  ∧ UNCHANGED ⟨stat, resps⟩
```

A(*n ex* −)primary processes a response to a write request from replica sender. The response might already be too late (it has already timed out), or it might be or might be in-time.

```
ProcessWriteResp(r, o, w, sender, ack) ≜   changes ⟨data, channel, stat, resps⟩
  ∧ resps′ = [resps EXCEPT ![r][o][sender] =
                  IF @ = "timeout" ∨ @ = "n/a" THEN @  resp came too late, it's already expired
                   ELSE  ack]  either ok or some system-specific error

  ∧ channel′ = [channel EXCEPT ![r] = Tail(@)]
  ∧ UNCHANGED ⟨data, stat⟩
```

A replica processes a message in its incoming channel. Depending on the type of message (update req, update resp), replica acts according to the three functions above.

```
_ProcessMessage(r) ≜
   ∧ channel[r] ≠ ⟨⟩  I have a message to process

   ∧ LET m ≜ Head(channel[r])IN
          ∧ IF m.type = "wrReq" THEN
               IF m.sender = Client THEN   upate request from client to a prim
                    Enters here only when Client writes are not a 0-stage action
                    PrimaryWrite (r, m.object, m.w, m.version)
                     TODO: NB: This function sends a Reply, while the others don't – BUG

                ELSE    It's a wrReq from a primary to a secondary
                      SecondaryWrite(r, m.object, m.w, m.version, m.sender)

             ELSE  IF m.type = "wrResp" THEN
                      ProcessWriteResp(r, m.object, m.w, m.sender, m.ack)
             ELSE
                    ∧ Print("BUGGGG!!!! Wrong message type!", m) ≠ ⟨⟩

   ∧ UNCHANGED ⟨master, cache⟩
```

```
$ProcessMessage \triangleq$
  $\exists\, r \in Rep :$
      $\land\ stat[r].phase = \text{"alive"}$   $r$ must be alive to process a message
      $\land\ \_ProcessMessage(r)$
```

A replica (either a current primary or not) finishes a write that it started some time ago (when it considered itself to be a primary).

```
$\_FinishWrite(r,\ o) \triangleq$
 LET
     $NotFinishedWrReqs(m) \triangleq m.object \neq o \lor m.sender \neq r$
                this message does not refer to a write on object $o$
                initiated by replica $r$. So, I don't declare it finished
     $NotFinishedWrResps(m) \triangleq m.object \neq o$
 IN
   $\land\ stat[r].phase = \text{"alive"}$   allow writes to finish even for those started by dead replicas
                                 $TODO$: $Shouldn$'t do this for Blue??????
   $\land\ stat[r].lock[o] = \text{"busy"}$   $r$ has indeed a write started

   $\land\ \forall\, r1 \in Rep : resps[r][o][r1] \neq \text{"waiting"}$   all the replicas have submitted
                                                  their responses, or have timed-out
                                                  or weren't supposed to answer

   $\land\ ReplyFinishedWrite(r,\ o,\ stat[r].in\_progress[o])$
           should send the appropriate $Reply$, based on state of the $resps$
           but not change any of master, data, cache, etc.!

   Re-initialize all write-related state, to prepare the replica for the next write
   $\land\ stat' = [stat \text{ EXCEPT } ![r].lock[o] = \text{"rdy"},$   $r$ is ready to accept new updates
                                               (if it's still prim, of course)
                   $![r].in\_progress[o] = [val \mapsto NoWrite,\ version \mapsto 0]]$
   re-init the responses, to prepare $r$ for the next write
   $\land\ resps' = [resps \text{ EXCEPT } ![r][o] = [r1 \in Rep \mapsto \text{"waiting"}]]$

   Remove ALL messages related to this object. Leave the rest intact
   $\land\ channel' = [r1 \in Rep \mapsto \text{IF } r1 \neq r \text{ THEN } SelectSeq(channel[r1],\ NotFinishedWrReqs)$
                                  $\text{ELSE } SelectSeq(channel[r],\ NotFinishedWrResps)]$
           I think that the above is not actually cheating, b/c you can imagine
           an implementation where replicas would simply identify older
           updates (via a sequence number, which is anyway necessary for
           keeping the order of updates.)
   $\land\ \text{UNCHANGED } \langle master,\ data,\ cache \rangle$

$FinishWrite \triangleq$
  $\exists\, r \in Rep,\ o \in Object : \_FinishWrite(r,\ o)$

$ReplicaActions \triangleq$
   $\lor\ ReplicaDeath$
```

```
    $\lor$ $ReadVersion$
    $\lor$ $ProcessMessage$
    $\lor$ $FinishWrite$        a replica finishes a write it has started
```

CLIENT actions.

Client wants to perform write w on object o. The request is performed in a 0-stage fashion. The method models caching of replica locations in the client – BUT IT'S NOT CORRECT !!!*TODO TODO*.

```
$\_CliWrite(prim, o, w) \triangleq$
    $\land$ $PrimaryWrite(prim, o, w, master.objects[o].version)$
    $\land$ UNCHANGED $\langle master, cache \rangle$
$CliWrite(o, w) \triangleq$
   $\exists r \in Rep$ :
       $\land$ $stat[r].phase = $ "alive"
       $\land$ $\_CliWrite(r, o, w)$
```

In the most general case, a read is typically performed on the local copy of some replica that stores that object (is either a *sec* or a primary). How you choose that replica depends on the protocol. In *GFS*, it's any replica. In Blue, it's gotta be prm The semantic of the read is that I read a whole object.

```
$\_CliRead(r, o) \triangleq$
    $\land$ LET $val \triangleq StoreRead(r, o)$
      IN    $Reply(r,$ "rd"$, val)$
    $\land$ UNCHANGED $\langle master, cache, data, stat, resps, channel \rangle$

$CliRead(o) \triangleq$
   $\exists r \in Rep$ :
       $\land$ $stat[r].phase = $ "alive"

       $\land$ $master.health[r] = $ "alive"  ASSUMPTION : *Uncomment* this if you want to test
                                       read-last-successful-X ($X =$ write/append).
                                       This ensures I don't read from a live but stale replica.
       $\land$ $r \in cache[r][o].sec \cup \{cache[r][o].prim\}$
               $r$ considers itself either a *sec* or primary of $o$
       $\land$ $\_CliRead(r, o)$

$ClientActions \triangleq$
   $\exists o \in Object$ :
        $\lor$ $\exists w \in WriteType : CliWrite(o, w)$
        $\lor$ $CliRead(o)$
```

```
$Next \triangleq$  $\lor$ $MasterActions$
            $\lor$ $TimeActions$
            $\lor$ $ReplicaActions$
            $\lor$ $ClientActions$
```

$Spec \;\triangleq\; \land\, Init \land \Box[Next]_{vars}$

Invariants

$AllInvariants \;\triangleq$

$\quad \land\, TypeInvariant$

THEOREM $Spec \Rightarrow \Box AllInvariants$