MODULE *paxos*

This is a specification of the paxos algorithm implemented in Ceph. The specification is based on the following source file: https://github.com/ceph/ceph/blob/master/src/mon/Paxos.cc

The main mechanism abstracted that may differ from the version implemented in Ceph are:

- The election logic. The leader is chosen randomly, and, for now, only one leader is chosen per epoch. When a new epoch begins, the messages from the previous epoch are discarded.
- Monitor quorum. The quorum is defined in the election phase, using all monitors that are up. Different epochs can have different quorums.
- The communication layer. The variable messages represents connections between monitors (e.g. messages[mon1][mon2] holds the messages sent from mon1 to mon2). Within a connection the messages are sent and received in order.
- The transactions. Transactions are simplified to represent only a change of a value in the variable monitor_store.
- Failure model. A monitor can crash if the remaining number of monitors is sufficient to form a quorum. When a monitor crashes, new elections are triggered and the monitor is marked to not be part of a quorum until he recovers.
- Timeouts. A timeout can occur at any point in the algorithm and it will trigger new elections.

For a more detailed overview of the specification: https://github.com/afonsonf/ceph-consensus-spec

EXTENDS $Integers$, $FiniteSets$, $Sequences$, TLC

Utils

Max element from a set.
@type: *Set*(*Int*) ⇒ *Int*;

$Max(S) \triangleq \text{CHOOSE } x \in S : \forall y \in S : x \geq y$

Min element from a set.
@type: *Set*(*Int*) ⇒ *Int*;

$Min(S) \triangleq \text{CHOOSE } x \in S : \forall y \in S : x \leq y$

Set of monitors to a sequence.

```
RECURSIVE $SetToSeq(\_)$
  @type: Set(MONITOR) ⇒ Seq(MONITOR);
$SetToSeq(S) \triangleq$
    IF $S = \{\}$ THEN $\langle\rangle$
                  ELSE LET $x \triangleq$ CHOOSE $x \in S$ : TRUE
                       IN  $\langle x \rangle \circ SetToSeq(S \setminus \{x\})$
```

Constants

Set of *Monitors*.

CONSTANTS @type: *Set*(*MONITOR*); $Monitors$

Sequence of monitors.

@type: $Seq(MONITOR)$;
$MonitorsSeq \triangleq TLCEval(SetToSeq(Monitors))$

Number of monitors.
@type: Int;
$MonitorsLen \triangleq TLCEval(Len(MonitorsSeq))$

Rank predicate, used to compute proposal numbers.
@type: $MONITOR \Rightarrow Int$;
$rank(mon) \triangleq \textsc{choose}\ i \in 1 \,..\, MonitorsLen : MonitorsSeq[i] = mon$

Set of possible values.
CONSTANTS @type: $Set(VALUE)$; $Value\_set$

Predicate used in the cfg file to define the symmetry set.
Workaround for typechecker.
@$typeAlias$: $MONITOR = T$;
@$typeAlias$: $VALUE = T$;
$SYMM \triangleq Permutations(Monitors) \cup Permutations(Value\_set)$

Reserved value.
CONSTANTS @type: VALUE; Nil

Paxos states.
CONSTANTS @type: $STATE\_NAME$; $STATE\_RECOVERING$, @type: $STATE\_NAME$; $STATE\_ACTIVE$,
@type: $STATE\_NAME$; $STATE\_UPDATING$, @type: $STATE\_NAME$; $STATE\_UPDATING\_PREVI$
@type: $STATE\_NAME$; $STATE\_WRITING$, @type: $STATE\_NAME$; $STATE\_WRITING\_PREVIOU$
@type: $STATE\_NAME$; $STATE\_REFRESH$, @type: $STATE\_NAME$; $STATE\_SHUTDOWN$

Paxos auxiliary phase states.
They are used to force some sequence of steps.
CONSTANTS @type: $PHASE\_NAME$; $PHASE\_ELECTION$,
@type: $PHASE\_NAME$; $PHASE\_SEND\_COLLECT$, @type: $PHASE\_NAME$; $PHASE\_COLLECT$,
@type: $PHASE\_NAME$; $PHASE\_LEASE$, @type: $PHASE\_NAME$; $PHASE\_LEASE\_DONE$,
@type: $PHASE\_NAME$; $PHASE\_BEGIN$, @type: $PHASE\_NAME$; $PHASE\_COMMIT$

Paxos message types.
CONSTANTS @type: $MESSAGE\_OP$; $OP\_COLLECT$, @type: $MESSAGE\_OP$; OP_LAST,
@type: $MESSAGE\_OP$; OP_BEGIN, @type: $MESSAGE\_OP$; OP_ACCEPT,
@type: $MESSAGE\_OP$; OP_COMMIT,
@type: $MESSAGE\_OP$; OP_LEASE, @type: $MESSAGE\_OP$; OP_LEASE_ACK

Global variables

Integer representing the current epoch. If is odd trigger an election.
VARIABLE @type: Int; $epoch$

Store messages waiting to be handled.

VARIABLE @type: $MONITOR \rightarrow (MONITOR \rightarrow Seq(MESSAGE))$;*messages*

Stores history of messages. Can be useful to find specific states.

VARIABLE @type: $Set(MESSAGE)$;*message_history*

Stores if a monitor is up or down. All available monitors, in a given epoch, are part of the quorum.

VARIABLE @type: $MONITOR \rightarrow Bool$;*quorum*

Size of the current quorum.

VARIABLE @type: Int;*quorum_sz*

State variables

A function that stores the current leader. *isLeader*[*mon*] is True iff *mon* is a leader, else False.

VARIABLE @type: $MONITOR \rightarrow Bool$;*isLeader*

A function that stores the state of each monitor.

VARIABLE @type: $MONITOR \rightarrow STATE\_NAME$;*state*

A function that stores the phase of each monitor.

VARIABLE @type: $MONITOR \rightarrow PHASE\_NAME$;*phase*

Restart variables

A function that stores, for each monitor, a proposal number when the commit phase starts.
This proposal number can be retrieved after a monitor crashes and restarts.

VARIABLE @type: $MONITOR \rightarrow PN$;*uncommitted_pn*

A function that stores, for each monitor, a value version when the commit phase starts.
This value version can be retrieved after a monitor crashes and restarts.

VARIABLE @type: $MONITOR \rightarrow VALUE\_VERSION$;*uncommitted_v*

A function that stores, for each monitor, a value when the commit phase starts.
This value can be retrieved after a monitor crashes and restarts.

VARIABLE @type: $MONITOR \rightarrow VALUE$;*uncommitted_value*

Data variables

A function that stores, for each monitor, the store where the transactions are applied.
In this model, a transaction represents changing the value in the store.

VARIABLE @type: $MONITOR \rightarrow VALUE$;*monitor_store*

A function that stores the transaction log of each monitor.

VARIABLE @type: $MONITOR \rightarrow (VALUE\_VERSION \rightarrow VALUE)$;*values*

A function that stores the last proposal number accepted by each monitor.

VARIABLE @type: $MONITOR \rightarrow PN$;*accepted_pn*

A function that stores the first value version committed by each monitor.

VARIABLE @type: $MONITOR \rightarrow VALUE\_VERSION$; $first\_committed$

A function that stores the last value version committed by each monitor.
VARIABLE @type: $MONITOR \rightarrow VALUE\_VERSION$; $last\_committed$

Collect phase variables

A function that stores the number of peers that accepted a collect request.
VARIABLE @type: $MONITOR \rightarrow Int$; num_last

Used by leader when receiving responses in collect phase.
VARIABLE @type: $MONITOR \rightarrow (MONITOR \rightarrow VALUE\_VERSION)$; $peer\_first\_committed$

Used by leader when receiving responses in collect phase.
VARIABLE @type: $MONITOR \rightarrow (MONITOR \rightarrow VALUE\_VERSION)$; $peer\_last\_committed$

Lease phase variables

A function that stores, for each monitor, which of the peers have acked the lease request.
VARIABLE @type: $MONITOR \rightarrow (MONITOR \rightarrow Bool)$; $acked\_lease$

Commit phase variables

A function that stores, for each monitor, the value proposed by a client.
VARIABLE @type: $MONITOR \rightarrow VALUE$; $pending\_proposal$

A function that stores, for each monitor, the value to be committed in the begin phase.
VARIABLE @type: $MONITOR \rightarrow VALUE$; new_value

A function that stores, for each monitor, which of the peers have acked the begin request.
VARIABLE @type: $MONITOR \rightarrow (MONITOR \rightarrow Bool)$; $accepted$

Debug variables

Variables to help debug a behavior.
step is the diameter of a behavior/path.
step_name the current predicate being called.
VARIABLE @type: Str; $step\_name$

Variables to limit the number of monitors crashes that can occur over a behavior.
This variable is used to limit the search space.
VARIABLE @type: Int; $number\_crashes$

Variables initialization

@$typeAlias$: $VALUE\_VERSION = Int$;
@$typeAlias$: $PN = Int$;

$$global\_vars \triangleq \langle epoch, messages, message\_history, quorum, quorum\_sz \rangle$$

$state\_vars \triangleq \langle isLeader, state, phase \rangle$
$restart\_vars \triangleq \langle uncommitted\_pn, uncommitted\_v, uncommitted\_value \rangle$
$data\_vars \triangleq \langle monitor\_store, values, accepted\_pn, first\_committed, last\_committed \rangle$
$collect\_vars \triangleq \langle num\_last, peer\_first\_committed, peer\_last\_committed \rangle$
$lease\_vars \triangleq acked\_lease$
$commit\_vars \triangleq \langle pending\_proposal, new\_value, accepted \rangle$

$vars \triangleq \langle global\_vars, state\_vars, restart\_vars, data\_vars, collect\_vars,$
$\quad lease\_vars, commit\_vars \rangle$

$Init\_global\_vars \triangleq$
$\quad \land\ epoch = 1$
$\quad \land\ messages = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \langle\rangle]]$
$\quad \land\ message\_history = \{\}$
$\quad \land\ quorum = [mon \in Monitors \mapsto \text{TRUE}]$
$\quad \land\ quorum\_sz = MonitorsLen$

$Init\_state\_vars \triangleq$
$\quad \land\ isLeader = [mon \in Monitors \mapsto \text{FALSE}]$
$\quad \land\ state = [mon \in Monitors \mapsto STATE\_RECOVERING]$
$\quad \land\ phase = [mon \in Monitors \mapsto PHASE\_ELECTION]$

$Init\_restart\_vars \triangleq$
$\quad \land\ uncommitted\_pn = [mon \in Monitors \mapsto 0]$
$\quad \land\ uncommitted\_v = [mon \in Monitors \mapsto 0]$
$\quad \land\ uncommitted\_value = [mon \in Monitors \mapsto Nil]$

$Init\_data\_vars \triangleq$
$\quad \land\ monitor\_store = [mon \in Monitors \mapsto Nil]$
$\quad \land\ values = [mon \in Monitors \mapsto [version \in \{\} \mapsto Nil]]$
$\quad \land\ accepted\_pn = [mon \in Monitors \mapsto 0]$
$\quad \land\ first\_committed = [mon \in Monitors \mapsto 0]$
$\quad \land\ last\_committed = [mon \in Monitors \mapsto 0]$

$Init\_collect\_vars \triangleq$
$\quad \land\ num\_last = [mon \in Monitors \mapsto 0]$
$\quad \land\ peer\_first\_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]]$
$\quad \land\ peer\_last\_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]]$

$Init\_lease\_vars \triangleq$
$\quad \land\ acked\_lease = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \text{FALSE}]]$

$Init\_commit\_vars \triangleq$
$\quad \land\ pending\_proposal = [mon \in Monitors \mapsto Nil]$
$\quad \land\ new\_value = [mon \in Monitors \mapsto Nil]$
$\quad \land\ accepted = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \text{FALSE}]]$

$Init \triangleq$

```
    ∧ $Init\_global\_vars$
    ∧ $Init\_state\_vars$
    ∧ $Init\_restart\_vars$
    ∧ $Init\_data\_vars$
    ∧ $Init\_collect\_vars$
    ∧ $Init\_lease\_vars$
    ∧ $Init\_commit\_vars$
    ∧ $step\_name = $ "init" ∧ $number\_crashes = 0$
```

Message manipulation

@*typeAlias*: $MESSAGE =$ [type: $MESSAGE\_OP$, from: $MONITOR$, *dest*: $MONITOR$, *first_committed*: $VALUE\_VERSION$, *last_committed*: $VALUE\_VERSION$, values: $(VALUE\_VERSION \rightarrow VALUE)$, *uncommitted_pn*: PN, *pn*: PN];
@*typeAlias*: $MESSAGE\_QUEUE = MONITOR \rightarrow (MONITOR \rightarrow Seq(MESSAGE))$;

Note: Variable *message_history* has impact in performace, update only when debugging.

Converts a set with at most one element to a sequence.
@type: $Set(MESSAGE) \Rightarrow Seq(MESSAGE)$;

```
$SingleMessageSetToSeq(S) \triangleq$
    IF $\exists\, elem \in S$ : TRUE THEN LET $elem \triangleq$ CHOOSE $x \in S$ : TRUE
                                      IN $\langle elem \rangle$
                                 ELSE $\langle\rangle$
```

Add message m to the network *msgs*.
@type: (MESSAGE, $MESSAGE\_QUEUE$) $\Rightarrow MESSAGE\_QUEUE$;

```
$WithMessage(m, msgs) \triangleq$
    $[msgs$ EXCEPT $![m.from] =$
        $[msgs[m.from]$ EXCEPT $![m.dest] = Append(msgs[m.from][m.dest], m)]]$
```

Remove message m from the network *msgs*.
@type: (MESSAGE, $MESSAGE\_QUEUE$) $\Rightarrow MESSAGE\_QUEUE$;

```
$WithoutMessage(m, msgs) \triangleq$
    $[msgs$ EXCEPT $![m.from] =$
        $[msgs[m.from]$ EXCEPT $![m.dest] = Tail(msgs[m.from][m.dest])]]$
```

Adds the message m to the network.
Variables changed: messages, *message_history*.
@type: $MESSAGE \Rightarrow Bool$;

```
$Send(m) \triangleq$
      ∧ $messages' = WithMessage(m, messages)$
       ∧ $message\_history' = message\_history \cup \{m\}$
      ∧ UNCHANGED $message\_history$
```

Adds a set of messages to the network.

Variables changed: messages, *message_history*.

@type: (MONITOR, $Set(MESSAGE)$) $\Rightarrow Bool$;

$Send\_set(from,\ m\_set) \triangleq$

```
  ∧ $messages' = [messages$ EXCEPT $![from] =$
      $[mon \in Monitors \mapsto$
         $messages[from][mon] \circ SingleMessageSetToSeq(\{m \in m\_set : m.dest = mon\})]]$
  $\land\ message\_history' = message\_history \cup m\_set$
  ∧ UNCHANGED $message\_history$
```

Removes the request from network and adds the response.

Variables changed: messages, *message_history*.

@type: (MESSAGE, MESSAGE) $\Rightarrow Bool$;

$Reply(response,\ request) \triangleq$

```
  ∧ $messages' = WithoutMessage(request,\ WithMessage(response,\ messages))$
  $\land\ message\_history' = message\_history \cup \{response\}$
  ∧ UNCHANGED $message\_history$
```

Removes the request from network and adds a set of messages.

Variables changed: messages, *message_history*.

@type: (MONITOR, $Set(MESSAGE)$, $MESSAGE$) $\Rightarrow Bool$;

$Reply\_set(from,\ response\_set,\ request) \triangleq$

```
  ∧ LET $msgs \triangleq WithoutMessage(request,\ messages)$
    IN   $messages' = [msgs$ EXCEPT $![from] =$
           $[mon \in Monitors \mapsto$
              $msgs[from][mon] \circ SingleMessageSetToSeq(\{m \in response\_set : m.dest = mon\})]]$
  $\land\ message\_history' = message\_history \cup response\_set$
  ∧ UNCHANGED $message\_history$
```

Removes message m from the network.

Variables changed: messages, *message_history*.

@type: $MESSAGE \Rightarrow Bool$;

$Discard(m) \triangleq$

```
  ∧   $messages' = WithoutMessage(m,\ messages)$
  ∧   UNCHANGED $message\_history$
```

Helper predicates

Computes a new unique proposal number for a given monitor.

Version A - Equal to the one in the source.

This version breaks the symmetry of the monitor set.

Example: $oldpn = 305$, $rank(mon) = 5$, $newpn = 405$.

@type: (MONITOR, Int) $\Rightarrow Int$;

$get\_new\_proposal\_number(mon,\ oldpn) \triangleq ((oldpn \div 100) + 1) * 100 + rank(mon)$

Version $B -$ Adapted to not break symmetry.

Example: $oldpn = 300$, $rank(mon) = 5$, $newpn = 400$.
@type: (MONITOR, Int) $\Rightarrow Int$;

```
$get\_new\_proposal\_number(mon, oldpn) \triangleq ((oldpn \div 100) + 1) * 100$
```

Clear the variable $peer\_first\_committed$.
Variables changed: $peer\_first\_committed$.
@type: $MONITOR \Rightarrow Bool$;

```
$clear\_peer\_first\_committed(mon) \triangleq$
    $peer\_first\_committed' = [peer\_first\_committed$ EXCEPT $![mon] =$
                                       $[m \in Monitors \mapsto -1]]$
```

Clear the variable $peer\_last\_committed$.
Variables changed: $peer\_last\_committed$.
@type: $MONITOR \Rightarrow Bool$;

```
$clear\_peer\_last\_committed(mon) \triangleq$
    $peer\_last\_committed' = [peer\_last\_committed$ EXCEPT $![mon] =$
                                       $[m \in Monitors \mapsto -1]]$
```

Store peer values and update $first\_committed$, $last\_committed$ and $monitor\_store$ accordingly.
Variables changed: values, $first\_committed$, $last\_committed$, $monitor\_store$.
@type: (MONITOR, MESSAGE) $\Rightarrow Bool$;

```
$store\_state(mon, msg) \triangleq$
    Choose peer values from $mon$ last committed $+1$ to peer last committed.
    $\land$ LET $logs \triangleq$ (DOMAIN $msg.values) \cap (last\_committed[mon] + 1 \,..\, msg.last\_committed)$
      IN   $\land$ $values' = [values$ EXCEPT $![mon] =$
                 $[i \in$ DOMAIN $values[mon] \cup logs \mapsto$
                    IF $i \in logs$
                     THEN $msg.values[i]$
                     ELSE $values[mon][i]]]$
           Update last committed and first committed.
           $\land$ $last\_committed' = [last\_committed$ EXCEPT $![mon] = Max(logs \cup \{last\_committed[mon]\})]$
           $\land$ IF $logs \neq \{\} \land first\_committed[mon] = 0$
               THEN $first\_committed' =$
                          $[first\_committed$ EXCEPT $![mon] = Min(logs)]$
               ELSE $first\_committed' =$
                          $[first\_committed$ EXCEPT $![mon] = Min(logs \cup \{first\_committed[mon]\})]$
    Update monitor store.
    $\land$ IF $last\_committed'[mon] = 0$
        THEN UNCHANGED $monitor\_store$
        ELSE $monitor\_store' = [monitor\_store$ EXCEPT $![mon] = values'[mon][last\_committed'[mon]]]$
```

Check if uncommitted value version is still valid, else reset it.
Variables changed: $uncommitted\_pn$, $uncommitted\_v$, $uncommitted\_value$.
@type: $MONITOR \Rightarrow Bool$;

```
$check\_and\_correct\_uncommitted(mon) \triangleq$
    IF $uncommitted\_v[mon] \leq last\_committed'[mon]$
    THEN  $\land$ $uncommitted\_v' = [uncommitted\_v$ EXCEPT $![mon] = 0]$
          $\land$ $uncommitted\_pn' = [uncommitted\_pn$ EXCEPT $![mon] = 0]$
          $\land$ $uncommitted\_value' = [uncommitted\_value$ EXCEPT $![mon] = Nil]$
    ELSE  UNCHANGED $\langle uncommitted\_pn, uncommitted\_v, uncommitted\_value \rangle$
```

Trigger new election by incrementing epoch.
Variables changed: epoch.
@type: *Bool*;

```
$bootstrap \triangleq$
    $\land$ $epoch' = epoch + 1$
```

Lease phase predicates

Changes *mon* state to *STATE_ACTIVE*.
Variables changed: state.
@type: *MONITOR* ⇒ *Bool*;

```
$finish\_round(mon) \triangleq$
    $\land$ $isLeader[mon] =$ TRUE
    $\land$ $state' = [state$ EXCEPT $![mon] = STATE\_ACTIVE]$
```

Resets the variable acked lease and send lease messages to peers.
Variables changed: *acked_lease*, messages, *message_history*, phase.
@type: *MONITOR* ⇒ *Bool*;

```
$extend\_lease(mon) \triangleq$
    $\land$ $isLeader[mon] =$ TRUE
    $\land$ $acked\_lease' = [acked\_lease$ EXCEPT $![mon] =$
        $[m \in Monitors \mapsto$ IF $m = mon$ THEN TRUE ELSE FALSE$]]$
    $\land$ $Send\_set(mon,$
        $\{[type$            $\mapsto OP\_LEASE,$
          $from$            $\mapsto mon,$
          $dest$            $\mapsto dest,$
          $last\_committed$ $\mapsto last\_committed[mon]] : dest \in \{m \in Monitors \setminus \{mon\} : quorum[m]\}$
         $\})$
    $\land$ $phase' = [phase$ EXCEPT $![mon] = PHASE\_LEASE]$
```

Handle a lease message. The peon changes his state and replies with a lease ack message.
The reply is commented because the lease ack is only used to check if all peers are up.
In the model this is done by “randomly” triggering the predicate *Timeout*. In this way, the search space is reduced.
Variables changed: messages, *message_history*, state.
@type: (MONITOR, MESSAGE) ⇒ *Bool*;

```
$handle\_lease(mon, msg) \triangleq$
    $\land$   discard if not peon or peon is behind
      IF  $\lor$ $isLeader[mon] =$ TRUE
          $\lor$ $last\_committed[mon] \neq msg.last\_committed$
       THEN  $\land$ $Discard(msg)$
```

```
                $\land$ UNCHANGED $state$
          ELSE  $\land$ $state' = [state$ EXCEPT $![mon] = STATE\_ACTIVE]$
           $\land$ $Reply([type \mapsto OP\_LEASE\_ACK,$
                $from \mapsto mon,$
                $dest \mapsto msg.from,$
                $first\_committed \mapsto first\_committed[mon],$
                $last\_committed \mapsto last\_committed[mon]], msg)$
           $\land$ $Discard(msg)$
    $\land$ UNCHANGED $\langle epoch, quorum, quorum\_sz, isLeader, phase \rangle$
    $\land$ UNCHANGED $\langle restart\_vars, data\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Handle a lease ack message. The leader updates the *acked_lease* variable.
Because the *lease_ack* messages are not sent, this predicate is never called.
The reasoning for this is given in *handle_lease* comment.
Variables changed: *acked_lease*, messages, *message_history*.
@type: (MONITOR, MESSAGE) $\Rightarrow$ *Bool*;

```
$handle\_lease\_ack(mon, msg) \triangleq$
    $\land$ $phase[mon] = PHASE\_LEASE$
    $\land$ $acked\_lease' = [acked\_lease$ EXCEPT $![mon] =$
        $[acked\_lease[mon]$ EXCEPT $![msg.from] =$ TRUE$]]$
    $\land$ $Discard(msg)$
    $\land$ UNCHANGED $\langle epoch, quorum, quorum\_sz \rangle$
    $\land$ UNCHANGED $\langle state\_vars, restart\_vars, data\_vars, collect\_vars, commit\_vars \rangle$
```

Predicate that is called when all peers ack the lease. The phase is changed to prevent loops.
Because the *lease_ack* messages are not sent, this predicate is never called.
The reasoning for this is given in *handle_lease* comment.
Variables changed: phase.
@type: *MONITOR* $\Rightarrow$ *Bool*;

```
$post\_lease\_ack(mon) \triangleq$
    $\land$ $phase[mon] = PHASE\_LEASE$
    $\land$ $phase' = [phase$ EXCEPT $![mon] = PHASE\_LEASE\_DONE]$
    $\land$ $\forall m \in Monitors : quorum[m] \Rightarrow acked\_lease[mon][m] =$ TRUE
    $\land$ UNCHANGED $\langle isLeader, state \rangle$
    $\land$ UNCHANGED $\langle global\_vars, restart\_vars, data\_vars, collect\_vars,$
                     $lease\_vars, commit\_vars \rangle$
```

Commit phase predicates

Start a commit phase by the leader. The variable *new_value* is assigned. Send begin messages to the peers.
The value of *uncommitted_v* and *uncommitted_value* are assigned in order for the leader to be
able to recover from a crash.
Variables changed: accepted, *new_value*, phase, messages, *message_history*, values, *uncommitted_pn*, *uncommitted_v*, *uncomm*
@type: (MONITOR, VALUE) $\Rightarrow$ *Bool*;

```
$begin(mon, v) \triangleq$
```

```
$\land$ $isLeader[mon] = \text{TRUE}$
$\land$ $\lor$ $state'[mon] = STATE\_UPDATING$
   $\lor$ $state'[mon] = STATE\_UPDATING\_PREVIOUS$
$\land$ $quorum\_sz = 1 \lor num\_last[mon] > MonitorsLen \div 2$
$\land$ $new\_value[mon] = Nil$
$\land$ $accepted' = [accepted \text{ EXCEPT } ![mon] =$
   $[m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]]$
$\land$ $new\_value' = [new\_value \text{ EXCEPT } ![mon] = v]$
$\land$ $phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_BEGIN]$
$\land$ $values' = [values \text{ EXCEPT } ![mon] =$
   $((last\_committed[mon] + 1) :> new\_value'[mon]) @@ values[mon]]$
$\land$ $Send\_set(mon,$
   $\{[type \mapsto OP\_BEGIN,$
     $from \mapsto mon,$
     $dest \mapsto dest,$
     $last\_committed \mapsto last\_committed[mon],$
     $values \mapsto values'[mon],$
     $pn \mapsto accepted\_pn[mon]] : dest \in \{m \in Monitors \setminus \{mon\} : quorum[m]\}$
    $\})$
$\land$ $uncommitted\_pn' = [uncommitted\_pn \text{ EXCEPT } ![mon] = accepted\_pn[mon]]$
$\land$ $uncommitted\_v' = [uncommitted\_v \text{ EXCEPT } ![mon] = last\_committed[mon] + 1]$
$\land$ $uncommitted\_value' = [uncommitted\_value \text{ EXCEPT } ![mon] = v]$
```

Handle a begin message. The monitor will accept if the proposal number in the message is greater or equal than the one he accepted.
Similar to what happens in begin, *uncommitted_v* and *uncommitted_value* are assigned in order for the monitor to recover in case of a crash.
Variables changed: messages, *message_history*, state, values, *uncommitted_pn*, *uncommitted_v*, *uncommitted_value*.
@type: (MONITOR, MESSAGE) $\Rightarrow$ *Bool*;

```
$handle\_begin(mon, msg) \triangleq$
  $\land$ $isLeader[mon] = \text{FALSE}$
  $\land$ $\text{IF } msg.pn < accepted\_pn[mon]$
     THEN
      $\land$ $Discard(msg)$
      $\land$ $\text{UNCHANGED } \langle state, values, restart\_vars \rangle$
     ELSE
      $\land$ $msg.pn = accepted\_pn[mon]$
      $\land$ $msg.last\_committed = last\_committed[mon]$

      assign $values[mon][last\_committed[mon] + 1]$
      $\land$ $values' = [values \text{ EXCEPT } ![mon] =$
         $((last\_committed[mon] + 1) :> msg.values[last\_committed[mon] + 1]) @@ values[mon]]$

      $\land$ $state' = [state \text{ EXCEPT } ![mon] = STATE\_UPDATING]$
      $\land$ $uncommitted\_pn' = [uncommitted\_pn \text{ EXCEPT } ![mon] = accepted\_pn[mon]]$
      $\land$ $uncommitted\_v' = [uncommitted\_v \text{ EXCEPT } ![mon] = last\_committed[mon] + 1]$
```

```
        $\land\ uncommitted\_value' = [uncommitted\_value\ \text{EXCEPT}\ ![mon] =$
            $values'[mon][last\_committed[mon] + 1]]$
        $\land\ Reply([type \mapsto OP\_ACCEPT,$
                     $from \mapsto mon,$
                     $dest \mapsto msg.from,$
                     $last\_committed \mapsto last\_committed[mon],$
                     $pn \mapsto accepted\_pn[mon]], msg)$
    $\land\ \text{UNCHANGED}\ \langle epoch, quorum, quorum\_sz, isLeader, phase, monitor\_store,$
                    $accepted\_pn, first\_committed, last\_committed\rangle$
    $\land\ \text{UNCHANGED}\ \langle collect\_vars, lease\_vars, commit\_vars\rangle$
```

Handle an accept message. If the leader receives a positive response from the peer, it will add it to the variable accepted.
Variables changed: messages, *message_history*, accepted
@type: (MONITOR, MESSAGE) $\Rightarrow$ *Bool*;

```
$handle\_accept(mon, msg) \triangleq$
    $\land\ isLeader[mon] = \text{TRUE}$
    $\land\ \lor\ state[mon] = STATE\_UPDATING\_PREVIOUS$
      $\lor\ state[mon] = STATE\_UPDATING$
    $\land\ phase[mon] = PHASE\_BEGIN$
    $\land\ new\_value[mon] \neq Nil$
    $\land\ \text{IF}\ \ \lor\ msg.pn \neq accepted\_pn[mon]$
            $\lor\ \land\ last\_committed[mon] > 0$
              $\land\ msg.last\_committed < last\_committed[mon] - 1$
        $\text{THEN}\ \text{UNCHANGED}\ accepted$
        $\text{ELSE}\ \ accepted' = [accepted\ \text{EXCEPT}\ ![mon] =$
                    $[accepted[mon]\ \text{EXCEPT}\ ![msg.from] = \text{TRUE}]]$
    $\land\ Discard(msg)$
    $\land\ \text{UNCHANGED}\ \langle epoch, quorum, quorum\_sz, pending\_proposal, new\_value\rangle$
    $\land\ \text{UNCHANGED}\ \langle restart\_vars, state\_vars, data\_vars, collect\_vars, lease\_vars\rangle$
```

Predicate that is enabled and called when all peers in the quorum accept begin request from leader.
The leader commits the transaction in *new_value* and sends commit messages to his peers.
Variables changed: *first_committed*, *last_committed*, *monitor_store*, *new_value*, messages, *message_history*, state, phase
@type: *MONITOR* $\Rightarrow$ *Bool*;

```
$post\_accept(mon) \triangleq$
    $\land\ phase[mon] = PHASE\_BEGIN$
    $\land\ \forall\, m \in Monitors : quorum[m] \Rightarrow accepted[mon][m] = \text{TRUE}$
    $\land\ new\_value[mon] \neq Nil$
    $\land\ \lor\ state[mon] = STATE\_UPDATING\_PREVIOUS$
      $\lor\ state[mon] = STATE\_UPDATING$
    $\land\ last\_committed' = [last\_committed\ \text{EXCEPT}\ ![mon] = last\_committed[mon] + 1]$

    $\land\ \text{IF}\ first\_committed[mon] = 0$
        $\text{THEN}\ first\_committed' = [first\_committed\ \text{EXCEPT}\ ![mon] = first\_committed[mon] + 1]$
        $\text{ELSE}\ \ \text{UNCHANGED}\ first\_committed$
    $\land$ $monitor\_store' = [monitor\_store \text{ EXCEPT } ![mon] = values[mon][last\_committed[mon] + 1]]$
    $\land$ $new\_value' = [new\_value \text{ EXCEPT } ![mon] = Nil]$
    $\land$ $Send\_set(mon,$
        ${[type \mapsto OP\_COMMIT,$
          $from \mapsto mon,$
          $dest \mapsto dest,$
          $last\_committed \mapsto last\_committed'[mon],$
          $pn \mapsto accepted\_pn[mon],$
          $values \mapsto values[mon]] : dest \in \{m \in Monitors \setminus \{mon\} : quorum[m]\}$
        $\})$
    $\land$ $state' = [state \text{ EXCEPT } ![mon] = STATE\_REFRESH]$
    $\land$ $phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_COMMIT]$
    $\land$ $\text{UNCHANGED } \langle isLeader, values, accepted\_pn, pending\_proposal, accepted\rangle$
    $\land$ $\text{UNCHANGED } \langle epoch, quorum, quorum\_sz, restart\_vars, collect\_vars, lease\_vars\rangle$
```

Predicate that is called after *post_accept*. The leader finishes the commit phase by updating his state to *STATE_ACTIVE* and by extending the lease to his peers.
Variables changed: state, phase, *acked_lease*, messages, *message_history*.
@type: *MONITOR* ⇒ *Bool*;

```
$finish\_commit(mon) \triangleq$
    $\land$ $state[mon] = STATE\_REFRESH$
    $\land$ $phase[mon] = PHASE\_COMMIT$
    $\land$ $finish\_round(mon)$
    $\land$ $extend\_lease(mon)$
    $\land$ $\text{UNCHANGED } \langle epoch, quorum, quorum\_sz, isLeader\rangle$
    $\land$ $\text{UNCHANGED } \langle restart\_vars, data\_vars, collect\_vars, commit\_vars\rangle$
```

Handle a commit message. The monitor stores the values sent by the leader commit message.
Variables changed: messages, *message_history*, values, *first_committed*, *last_committed*, *monitor_store*, *uncommitted_v*, *uncommitted_pn*, *uncommitted_value*.
@type: (MONITOR, MESSAGE) ⇒ *Bool*;

```
$handle\_commit(mon, msg) \triangleq$
    $\land$ $isLeader[mon] = \text{FALSE}$
    $\land$ $store\_state(mon, msg)$
    $\land$ $check\_and\_correct\_uncommitted(mon)$
    $\land$ $Discard(msg)$
    $\land$ $\text{UNCHANGED } \langle epoch, quorum, quorum\_sz, accepted\_pn\rangle$
    $\land$ $\text{UNCHANGED } \langle state\_vars, collect\_vars, lease\_vars, commit\_vars\rangle$
```

Client Request

Request a transaction v to the monitor. The transaction is saved on pending proposal to be committed in the next available commit phase.
Variables changed: *pending_proposal*.
@type: (MONITOR, VALUE) ⇒ *Bool*;

```
$client\_request(mon, v) \triangleq$
    $\land isLeader[mon] = \text{TRUE}$
    $\land state[mon] = STATE\_ACTIVE$
    $\land pending\_proposal[mon] = Nil$
    $\land pending\_proposal' = [pending\_proposal \text{ EXCEPT } ![mon] = v]$
    $\land \text{UNCHANGED } \langle new\_value, accepted \rangle$
    $\land \text{UNCHANGED } \langle global\_vars, state\_vars, restart\_vars, data\_vars, collect\_vars, lease\_vars \rangle$
```

Start a commit phase with the value on pending proposal.
Variables changed: state, *pending_proposal*, accepted, *new_value*, phase, messages, *message_history*, values, *uncommitted_pn*, *uncommitted_v*, *uncommitted_value*.
@type: *MONITOR* ⇒ *Bool*;

```
$propose\_pending(mon) \triangleq$
    $\land phase[mon] = PHASE\_LEASE \lor phase[mon] = PHASE\_ELECTION$
    $\land state[mon] = STATE\_ACTIVE$
    $\land pending\_proposal[mon] \neq Nil$
    $\land pending\_proposal' = [pending\_proposal \text{ EXCEPT } ![mon] = Nil]$
    $\land state' = [state \text{ EXCEPT } ![mon] = STATE\_UPDATING]$
    $\land begin(mon, pending\_proposal[mon])$
    $\land \text{UNCHANGED } \langle isLeader, monitor\_store, accepted\_pn, first\_committed, last\_committed \rangle$
    $\land \text{UNCHANGED } \langle epoch, quorum, quorum\_sz, collect\_vars, lease\_vars \rangle$
```

Collect phase predicates

Start collect phase. This first part of the collect phase is divided in two parts (collect and *send_collect*) in order to simplify variable changes (when collect is triggered from *handle_last*).
Variables changed: *accepted_pn*, phase.
@type: (MONITOR, *Int*) ⇒ *Bool*;

```
$collect(mon, oldpn) \triangleq$
    $\land state[mon] = STATE\_RECOVERING$
    $\land isLeader[mon] = \text{TRUE}$
    $\land \text{LET } new\_pn \triangleq get\_new\_proposal\_number(mon, Max(\{oldpn, accepted\_pn[mon]\}))$
      $\text{IN } \quad \land accepted\_pn' = [accepted\_pn \text{ EXCEPT } ![mon] = new\_pn]$
    $\land phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_SEND\_COLLECT]$
```

Continue the start of the collect phase. Initialize the number of peers that accepted the proposal (*num_last*) and the variables with peers version numbers. Check if there is an uncommitted value.
Send collect messages to the peers.
Variables changed: *peer_first_committed*, *peer_last_committed*, *uncommitted_pn*, *uncommitted_v*, *uncommitted_value*, *num_las*
messages, *message_history*, phase.
@type: *MONITOR* ⇒ *Bool*;

```
$send\_collect(mon) \triangleq$
    $\land state[mon] = STATE\_RECOVERING$
    $\land isLeader[mon] = \text{TRUE}$
    $\land phase[mon] = PHASE\_SEND\_COLLECT$
    $\land$ $clear\_peer\_first\_committed(mon)$
    $\land$ $clear\_peer\_last\_committed(mon)$

    $\land$ IF $last\_committed[mon] + 1 \in$ DOMAIN $values[mon]$
       THEN $\land$ $uncommitted\_v' =$
                 $[uncommitted\_v$ EXCEPT $![mon] = last\_committed[mon] + 1]$
            $\land$ $uncommitted\_value' =$
                 $[uncommitted\_value$ EXCEPT $![mon] = values[mon][last\_committed[mon] + 1]]$
            $\land$ $uncommitted\_pn' = uncommitted\_pn$
       ELSE UNCHANGED $\langle restart\_vars \rangle$

    $\land$ $num\_last' = [num\_last$ EXCEPT $![mon] = 1]$
    $\land$ $Send\_set(mon,$
       $\{[type \mapsto OP\_COLLECT,$
         $from \mapsto mon,$
         $dest \mapsto dest,$
         $first\_committed \mapsto first\_committed[mon],$
         $last\_committed \mapsto last\_committed[mon],$
         $pn \mapsto accepted\_pn[mon]] : dest \in \{m \in Monitors \setminus \{mon\} : quorum[m]\}$
       $\})$
    $\land$ $phase' = [phase$ EXCEPT $![mon] = PHASE\_COLLECT]$
    $\land$ UNCHANGED $\langle isLeader, state \rangle$
    $\land$ UNCHANGED $\langle epoch, quorum, quorum\_sz, data\_vars, lease\_vars, commit\_vars \rangle$
```

Handle a collect message. The peer will accept the proposal number from the leader if it is bigger than the last proposal number he accepted.

Variables changed: messages, *message_history*, epoch, state, *accepted_pn*.

@type: (MONITOR, MESSAGE) $\Rightarrow$ *Bool*;

```
$handle\_collect(mon, msg) \triangleq$
    $\land$ $isLeader[mon] =$ FALSE
    $\land$ $state' = [state$ EXCEPT $![mon] = STATE\_RECOVERING]$
    $\land$ $\lor$ $\land$ $msg.first\_committed > last\_committed[mon] + 1$
            $\land$ $bootstrap$
            $\land$ $Discard(msg)$
            $\land$ UNCHANGED $\langle accepted\_pn \rangle$
       $\lor$ $\land$ $msg.first\_committed \leq last\_committed[mon] + 1$
            $\land$ IF $msg.pn > accepted\_pn[mon]$
                 THEN $accepted\_pn' = [accepted\_pn$ EXCEPT $![mon] = msg.pn]$
                 ELSE UNCHANGED $accepted\_pn$
            $\land$ $Reply([type \mapsto OP\_LAST,$
                    $from \mapsto mon,$
                    $dest \mapsto msg.from,$
                    $first\_committed \mapsto first\_committed[mon],$
                    $last\_committed \mapsto last\_committed[mon],$
                    $values \mapsto values[mon],$
                    $uncommitted\_pn \mapsto uncommitted\_pn[mon],$
                    $pn \qquad\qquad \mapsto accepted\_pn'[mon]], msg)$
            $\land$ UNCHANGED $epoch$
        $\land$ UNCHANGED $\langle isLeader, phase, values, first\_committed, last\_committed, monitor\_store \rangle$
        $\land$ UNCHANGED $\langle quorum, quorum\_sz, restart\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Handle a last message (response from a peer to the leader collect message).
The peers first and last committed version are stored. If the leader is behind, bootstraps. Stores any value that the peer may have committed (*store_state*). If peer is behind send commit message with leader values.
If peer accepted proposal number increase num last, if he sent a bigger proposal number start a new collect phase.
Variables changed: messages, *message_history*, epoch, phase, *uncommitted_pn*, *uncommitted_v*, *uncommitted_value*, *monitor_s*
accepted_pn, *first_committed*, *last_committed*, *num_last*, *peer_first_committed*, *peer_last_committed*.
@type: (MONITOR, MESSAGE) $\Rightarrow$ *Bool*;

```
$handle\_last(mon, msg) \triangleq$
    $\land isLeader[mon] = $ TRUE

    $\land peer\_first\_committed' = [peer\_first\_committed$ EXCEPT $![mon] =$
        $[peer\_first\_committed[mon]$ EXCEPT $![msg.from] = msg.first\_committed]]$
    $\land peer\_last\_committed' = [peer\_last\_committed$ EXCEPT $![mon] =$
        $[peer\_last\_committed[mon]$ EXCEPT $![msg.from] = msg.last\_committed]]$

    $\land$ IF $msg.first\_committed > last\_committed[mon] + 1$
        THEN
         $\land bootstrap$
         $\land Discard(msg)$
         $\land$ UNCHANGED $\langle num\_last, accepted\_pn, values, phase, monitor\_store \rangle$
         $\land$ UNCHANGED $\langle first\_committed, last\_committed, restart\_vars \rangle$
        ELSE
         $\land store\_state(mon, msg)$
         $\land$ IF $\exists peer \in Monitors :$
                $\land peer \neq mon$
                $\land peer\_last\_committed'[mon][peer] \neq -1$
                $\land peer\_last\_committed'[mon][peer] + 1 < first\_committed[mon]$
                $\land first\_committed[mon] > 1$
            THEN
             $\land bootstrap$
             $\land check\_and\_correct\_uncommitted(mon)$
             $\land Discard(msg)$
             $\land$ UNCHANGED $\langle phase, accepted\_pn, num\_last \rangle$
            ELSE
             $\land$ LET $monitors\_behind \triangleq \{peer \in Monitors :$
                    $\land peer \neq mon$
                    $\land peer\_last\_committed'[mon][peer] \neq -1$
                    $\land peer\_last\_committed'[mon][peer] < last\_committed[mon]$
                    $\land quorum[peer]\}$
               IN  $Reply\_set(mon,$
                    $\{[type \qquad\qquad \mapsto OP\_COMMIT,$
                    $from \mapsto mon,$
                    $dest \mapsto dest,$
                    $last\_committed \mapsto last\_committed'[mon],$
                    $pn \mapsto accepted\_pn[mon],$
                    $values \mapsto values[mon]] : dest \in monitors\_behind$
                  $\}, msg)$
          $\land \lor \land msg.pn > accepted\_pn[mon]$
                $\land collect(mon, msg.pn)$
                $\land check\_and\_correct\_uncommitted(mon)$
                $\land$ UNCHANGED $num\_last$

             $\lor \land msg.pn = accepted\_pn[mon]$
                $\land num\_last' = [num\_last$ EXCEPT $![mon] = num\_last[mon] + 1]$
                $\land$ IF $\land msg.last\_committed + 1 \in$ DOMAIN $msg.values$
                       $\land msg.last\_committed \geq last\_committed'[mon]$
                       $\land msg.last\_committed + 1 \geq uncommitted\_v[mon]$
                       $\land msg.uncommitted\_pn \geq uncommitted\_pn[mon]$
                  THEN $\land uncommitted\_v' =$
                            $[uncommitted\_v$ EXCEPT $![mon] = msg.last\_committed + 1]$
                       $\land uncommitted\_pn' =$
                            $[uncommitted\_pn$ EXCEPT $![mon] = msg.uncommitted\_pn]$
                       $\land uncommitted\_value' =$
                            $[uncommitted\_value$ EXCEPT $![mon] = msg.values[msg.last\_committed + 1]]$
                  ELSE $check\_and\_correct\_uncommitted(mon)$
                $\land$ UNCHANGED $\langle phase, accepted\_pn \rangle$

             $\lor \land msg.pn < accepted\_pn[mon]$
                $\land check\_and\_correct\_uncommitted(mon)$
                $\land$ UNCHANGED $\langle phase, accepted\_pn, num\_last \rangle$
          $\land$ UNCHANGED $epoch$
       $\land$ UNCHANGED $\langle epoch \rangle$

    $\land$ UNCHANGED $\langle quorum, quorum\_sz, isLeader, state \rangle$
    $\land$ UNCHANGED $\langle lease\_vars, commit\_vars \rangle$
```

Predicate that is enabled and called when all peers in quorum accept collect request from leader. If there is an uncommitted value, a commit phase is started with that value, else the leader changes to $ACTIVE\_STATE$ and extends the lease to his peers.
Variables changed: $peer\_first\_committed$, $peer\_last\_committed$, state, accepted, new_value, phase, messages, $message\_history$, values, $uncommitted\_pn$, $uncommitted\_v$, $uncommitted\_value$, $acked\_lease$.
@type: $MONITOR \Rightarrow Bool$;

```
$post\_last(mon) \triangleq$
    $\land isLeader[mon] = $ TRUE
    $\land num\_last[mon] = quorum\_sz$
    $\land phase[mon] = PHASE\_COLLECT$
    $\land$ $clear\_peer\_first\_committed(mon)$
    $\land$ $clear\_peer\_last\_committed(mon)$

    $\land$ IF $\land$ $uncommitted\_v[mon] = last\_committed[mon] + 1$
           $\land$ $uncommitted\_value[mon] \neq Nil$
       THEN $\land$ $state' = [state$ EXCEPT $![mon] = STATE\_UPDATING\_PREVIOUS]$
            $\land$ $begin(mon, uncommitted\_value[mon])$
            $\land$ UNCHANGED $\langle acked\_lease \rangle$
       ELSE $\land$ $finish\_round(mon)$
            $\land$ $extend\_lease(mon)$
            $\land$ UNCHANGED $\langle accepted, new\_value, values, restart\_vars \rangle$

    $\land$ UNCHANGED $\langle isLeader, monitor\_store, accepted\_pn, first\_committed, last\_committed \rangle$
    $\land$ UNCHANGED $\langle epoch, quorum, quorum\_sz, num\_last, pending\_proposal \rangle$
```

Leader election

Elect one monitor as a leader and initialize the remaining ones as peons.
Variables changed: *isLeader*, state, phase, *new_value*, *pending_proposal*, epoch.
@type: *Bool*;

```
$leader\_election \triangleq$
    $\land$ $\exists\, mon \in Monitors$ :
        $\land$ $quorum[mon]$
        $\land$ $isLeader' = [m \in Monitors \mapsto$ IF $m = mon$ THEN TRUE ELSE FALSE$]$
        $\land$ $state' = [m \in Monitors \mapsto$
            IF $quorum\_sz = 1$ THEN $STATE\_ACTIVE$ ELSE $STATE\_RECOVERING]$
    $\land$ $phase' = [m \in Monitors \mapsto PHASE\_ELECTION]$
    $\land$ $new\_value' = [m \in Monitors \mapsto Nil]$
    $\land$ $pending\_proposal' = [m \in Monitors \mapsto Nil]$
    $\land$ $epoch' = epoch + 1$
    $\land$ $messages' = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \langle\rangle]]$
    $\land$ UNCHANGED $\langle quorum, quorum\_sz, accepted, message\_history \rangle$
    $\land$ UNCHANGED $\langle data\_vars, restart\_vars, collect\_vars, lease\_vars \rangle$
```

Start recovery phase if number of monitors in quorum is greater than 1.
Variables changed: *accepted_pn*, phase.
@type: *MONITOR* $\Rightarrow$ *Bool*;

```
$election\_recover(mon) \triangleq$
    $\land$ $quorum\_sz > 1$
    $\land$ $phase[mon] = PHASE\_ELECTION$
    $\land$ $collect(mon, 0)$
    $\land$ UNCHANGED $\langle isLeader, state, values, first\_committed, last\_committed, monitor\_store \rangle$
    $\land$ UNCHANGED $\langle global\_vars, restart\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Timeouts and restart

Remove monitor from quorum, if there are enough monitors in the quorum.
@type: *MONITOR* ⇒ *Bool*;

```
$crash\_mon(mon) \triangleq$
    $\land\ quorum\_sz > (MonitorsLen \div 2) + 1$
    $\land\ quorum[mon] = \text{TRUE}$
    $\land\ quorum' = [quorum\ \text{EXCEPT}\ ![mon] = \text{FALSE}]$
    $\land\ quorum\_sz' = quorum\_sz - 1$
    $\land\ bootstrap$
    $\land\ number\_crashes' = number\_crashes + 1$
    $\land\ \text{UNCHANGED}\ \langle messages, message\_history \rangle$
    $\land\ \text{UNCHANGED}\ \langle state\_vars, restart\_vars, data\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Add monitor to the quorum.
@type: *MONITOR* ⇒ *Bool*;

```
$restore\_mon(mon) \triangleq$
    $\land\ quorum[mon] = \text{FALSE}$
    $\land\ quorum' = [quorum\ \text{EXCEPT}\ ![mon] = \text{TRUE}]$
    $\land\ quorum\_sz' = quorum\_sz + 1$
    $\land\ bootstrap$
    $\land\ \text{UNCHANGED}\ \langle messages, message\_history \rangle$
    $\land\ \text{UNCHANGED}\ \langle state\_vars, restart\_vars, data\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Monitor timeout (simulate the various timeouts that can occur). Triggers new elections.
Variables changed: epoch.
@type: *MONITOR* ⇒ *Bool*;

```
$Timeout(mon) \triangleq$
    $\land\ bootstrap$
    $\land\ \text{UNCHANGED}\ \langle messages, quorum, quorum\_sz, message\_history, state\_vars, restart\_vars,$
                          $data\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$
```

Dispatchers and next statement

Handle a message.
@type: *MESSAGE* ⇒ *Bool*;

```
$Receive(msg) \triangleq$
    $\land$  $\lor\ \land\ msg.type = OP\_COLLECT$
            $\land\ handle\_collect(msg.dest, msg)$
            $\land\ step\_name' =$ "receive collect"

         $\lor\ \land\ msg.type = OP\_LAST$
            $\land\ handle\_last(msg.dest, msg)$
            $\land\ step\_name' =$ "receive last"

         $\lor\ \land\ msg.type = OP\_LEASE$
            $\land\ handle\_lease(msg.dest, msg)$
            $\land\ step\_name' =$ "receive lease"
    $\lor$ $\land$ $msg.type = OP\_LEASE\_ACK$
       $\land$ $handle\_lease\_ack(msg.dest, msg)$
       $\land$ $step\_name' =$ "receive lease_ack"

    $\lor$ $\land$ $msg.type = OP\_BEGIN$
       $\land$ $handle\_begin(msg.dest, msg)$
       $\land$ $step\_name' =$ "receive begin"

    $\lor$ $\land$ $msg.type = OP\_ACCEPT$
       $\land$ $handle\_accept(msg.dest, msg)$
       $\land$ $step\_name' =$ "receive accept"

    $\lor$ $\land$ $msg.type = OP\_COMMIT$
       $\land$ $handle\_commit(msg.dest, msg)$
       $\land$ $step\_name' =$ "receive commit"

Limit some variables to reduce search space.
@type: $Bool$;
$reduce\_search\_space \triangleq$
    $\land$ $epoch \neq 8$
    $\land$ $\lor$ $\forall mon \in Monitors : last\_committed[mon] < 2$
       $\lor$ $\forall mon2 \in Monitors: new\_value[mon2] = Nil$
    $\land$ $\forall mon \in Monitors : accepted\_pn[mon] < 300$
    $\land$ $number\_crashes \neq 4$

State transitions.
@type: $Bool$;
$Next \triangleq$
    $\land$ $reduce\_search\_space$
    $\land$ IF $epoch \% 2 = 1$ THEN
        $\land$ $leader\_election$
        $\land$ $step\_name' =$ "election"
        $\land$ UNCHANGED $number\_crashes$
       ELSE
        $\lor$ $\land$ $\exists mon \in Monitors : election\_recover(mon)$
           $\land$ $step\_name' =$ "election_recover"
           $\land$ UNCHANGED $number\_crashes$

        $\lor$ $\land$ $\exists mon \in Monitors : send\_collect(mon)$
           $\land$ $step\_name' =$ "send_collect"
           $\land$ UNCHANGED $number\_crashes$

        $\lor$ $\land$ $\exists mon \in Monitors : post\_last(mon)$
           $\land$ $step\_name' =$ "post_last"
           $\land$ UNCHANGED $number\_crashes$

        $\lor$ $\land$ $\exists mon \in Monitors : post\_lease\_ack(mon)$
        ∧ $step\_name' =$ “post_lease_ack”
        ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : post\_accept(mon)$
      ∧ $step\_name' =$ “post_accept”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : finish\_commit(mon)$
      ∧ $step\_name' =$ “finish_commit”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : \exists\, v \in Value\_set : client\_request(mon, v)$
      ∧ $step\_name' =$ “client_request”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : propose\_pending(mon)$
      ∧ $step\_name' =$ “propose_pending”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon1, mon2 \in Monitors :$
            ∧ $mon1 \neq mon2$
            ∧ $Len(messages[mon1][mon2]) > 0$
            ∧ $Receive(messages[mon1][mon2][1])$
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : crash\_mon(mon)$
      ∧ $step\_name' =$ “crash_mon”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : restore\_mon(mon)$
      ∧ $step\_name' =$ “restore_mon”
      ∧ UNCHANGED $number\_crashes$

    ∨ ∧ $\exists\, mon \in Monitors : Timeout(mon)$
      ∧ $step\_name' =$ “timeout_and_restart”
      ∧ UNCHANGED $number\_crashes$
```

Safety invariants

If two monitors are in state active then their *monitor_store* must have the same value.
@type: *Bool*;

$$same\_monitor\_store \triangleq \forall\, mon1, mon2 \in Monitors : \\ state[mon1] = STATE\_ACTIVE \land state[mon2] = STATE\_ACTIVE \\ \Rightarrow monitor\_store[mon1] = monitor\_store[mon2]$$

Invariant.
@type: *Bool*;

$Inv \triangleq \land\ same\_monitor\_store$

Test/Debug invariants

Invariant used to search for a state where 'x' happens.

$Inv\_find\_state(x) \triangleq \neg x$

Invariant used to search for a behavior of diameter equal to 'size'.
$TLCGet($"level"$)$ not supported by snowcat typechecker.
$Inv\_diam(size) \triangleq TLCGet(\text{"level"}) \neq size - 1$

Invariants to test in model check

```
$DEBUG\_Inv \triangleq \land$ TRUE
                 $\land Inv\_diam(20)$
```

Examples:

Find a behavior with a diameter of size 60.
$Inv\_diam(60)$

Find a behavior where two different monitors assume the role of a leader.

```
$Inv\_find\_state($
  $\exists\, msg1, msg2 \in message\_history :$
      $\land\ msg1.type = OP\_COLLECT \land msg2.type = OP\_COLLECT$
      $\land\ msg1.from \neq msg2.from$
$)$
```

Find a state where a monitor crashed during the collect phase and fails to send a OP_LAST message.

```
$Inv\_find\_state($
  $\land\ step\_name = \text{"crash mon"}$

  \* The system is in collect phase and no OP_LAST message has been received.
  \* isLeader[mon] = TRUE assures that the leader was not the one that crashed.
  $\land\ \exists\, mon \in Monitors :$
      $\land\ isLeader[mon] = \text{TRUE}$
      $\land\ phase[mon] = PHASE\_COLLECT$
      $\land\ num\_last[mon] = 1$

  \* All the collect requests have been handled by the peers.
  $\land\ \forall\, mon1, mon2 \in Monitors :$
      $\forall\, i \in 1 \,..\, Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type \neq OP\_COLLECT$

  $\land\ epoch = 2$
$)$
```

Find a state where the leader crashes during the commit phase, failing to complete the commit.

```
$Inv\_find\_state($
  $\land\ step\_name = \text{"crash mon"}$
  $\land\ \exists\, mon1, mon2 \in Monitors :$
      $\exists\, i \in 1 \,..\, Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type = OP\_ACCEPT$
  $\land\ \forall\, mon \in Monitors :$
      $isLeader[mon] = \text{FALSE}$
```

    $\land\ epoch = 2$
)
Note: After finding a state, that complete state can be used as an initial state to analyze behaviors from there.

\ $*$ Modification History
\ $*$ Last modified *Wed Apr* 14 16:32:51 WEST 2021 by *afonsonf*
\ $*$ Created *Mon Jan* 11 16:15:26 WET 2021 by *afonsonf*