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 deviations/abstractions done that may differ from the implementation are:

- The election logic. The leader is chosen randomly, and, for now, only one leader is chosen per epoch.
- The quorum of monitors. For now, the specification considers the quorum to be the set of all monitors and that the quorum does not change over time.
- The communication layer. The variable messages holds the messages waiting to be handled. For now, messages cannot be randomly duplicated nor lost, and some messages can be received out of order.
- The transactions. In this specification, transactions represent only a change of value in the variable monitor_store.
- Failure model. For now, if a monitor crashes it will instantly restart, resetting some variables and continuing to participate in the quorum.

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

EXTENDS *Integers*, *FiniteSets*, *Sequences*, *TLC*, *SequencesExt*, *FiniteSetsExt*

Constants

Set of monitors.

CONSTANTS *Monitors*

Sequence of monitors and the rank predicate, used to compute proposal numbers.

$$ranks \triangleq SetToSeq(Monitors)$$
$$rank(mon) \triangleq \textsc{choose}\ i \in 1 \,..\, Len(ranks) : ranks[i] = mon$$

Set of possible values.

CONSTANTS *Value_set*

Reserved value.

CONSTANTS *Nil*

Paxos states:

CONSTANTS *STATE_RECOVERING*, *STATE_ACTIVE*,
STATE_UPDATING, *STATE_UPDATING_PREVIOUS*,
STATE_WRITING, *STATE_WRITING_PREVIOUS*,
STATE_REFRESH, *STATE_SHUTDOWN*

$$\begin{aligned} state\_names \triangleq \{&STATE\_RECOVERING, STATE\_ACTIVE,\\ &STATE\_UPDATING, STATE\_UPDATING\_PREVIOUS,\\ &STATE\_WRITING, STATE\_WRITING\_PREVIOUS,\\ &STATE\_REFRESH, STATE\_SHUTDOWN\}\end{aligned}$$

Paxos auxiliary phase states:

They are used to force some sequence of steps.

```
CONSTANTS PHASE_ELECTION,
          PHASE_PRE_COLLECT, PHASE_COLLECT,
          PHASE_LEASE, PHASE_LEASE_DONE,
          PHASE_BEGIN, PHASE_BEGIN_DONE,
          PHASE_COMMIT, PHASE_COMMIT_DONE

phase_names ≜ {PHASE_ELECTION,
        PHASE_PRE_COLLECT, PHASE_COLLECT,
        PHASE_LEASE, PHASE_LEASE_DONE,
        PHASE_BEGIN, PHASE_BEGIN_DONE,
        PHASE_COMMIT, PHASE_COMMIT_DONE}
```

Paxos message types:

```
CONSTANTS OP_COLLECT, OP_LAST,
          OP_BEGIN, OP_ACCEPT, OP_COMMIT,
          OP_LEASE, OP_LEASE_ACK

messages_types ≜ {OP_COLLECT, OP_LAST,
        OP_BEGIN, OP_ACCEPT, OP_COMMIT,
        OP_LEASE, OP_LEASE_ACK}
```

Global variables

Integer representing the current epoch. If is odd trigger an election.

Type: Integer

```
VARIABLE epoch
```

A function that stores messages.

Type: ⟨*message*⟩

```
VARIABLE messages
```

Stores history of message events. Can be useful to find specific states.

Type: {*messages*}

```
VARIABLE message_history
```

State variables

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

Type: [*Monitors* ↦ *Bool*]

```
VARIABLE isLeader
```

A function that stores the state of each monitor.

Type: [*Monitors* ↦ *state_names*]

```
VARIABLE state
```

A function that stores the phase of each monitor.

Type: [*Monitors* ↦ *phase_names*]

VARIABLE *phase*

Restart variables

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.
Type: [*Monitors* ↦ value version]

VARIABLE *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.
Type: [*Monitors* ↦ *Value_set*]

VARIABLE *uncommitted_value*

Data variables

A function that stores, for each monitor, the current store where the transactions are applied.
In this model, a transaction represents changing the value in the store.
Type: [*Monitors* ↦ *Value_set*]

VARIABLE *monitor_store*

A function that stores the transaction log of each monitor.
Type: [*Monitors* ↦ [value *version* ↦ *Value_set*]]

VARIABLE *values*

A function that stores the last proposal number accepted by each monitor.
Type: [*Monitors* ↦ proposal number]

VARIABLE *accepted_pn*

A function that stores the first value version committed for each monitor.
Type: [*Monitors* ↦ value version]

VARIABLE *first_committed*

A function that stores the last value version committed for each monitor.
Type: [*Monitors* ↦ value version]

VARIABLE *last_committed*

Collect phase variables

A function that stores the number of peers that accepted a collect request.
Type: [*Monitors* ↦ number of peers that accepted]

VARIABLE *num_last*

Used by leader when receiving responses in collect phase.
Type: [*Monitors* ↦ [*Monitors* ↦ value version]]

VARIABLE *peer_first_committed*

Used by leader when receiving responses in collect phase.

Type: $[Monitors \mapsto [Monitors \mapsto \text{ value version}]]$

VARIABLE $peer\_last\_committed$

Lease phase variables

A function that stores, for each monitor, which of the peers have acked the lease request.
Type: $[Monitors \mapsto [Monitors \mapsto Bool]]$

VARIABLE $acked\_lease$

Commit phase variables

A function that stores, for each monitor, the value proposed by a client.
Type: $[Monitors \mapsto Value\_set \cup \{Nil\}]$

VARIABLE $pending\_proposal$

A function that stores, for each monitor, the value to be committed in the begin phase.
Type: $[Monitors \mapsto Value\_set \cup \{Nil\}]$

VARIABLE new_value

A function that stores, for each monitor, which of the peers have acked the begin request.
Type: $[Monitord \mapsto [Monitors \mapsto Bool]]$

VARIABLE $accepted$

Debug variables

Variables to help debug a behavior.
step is the diameter of a behavior/path.
$step\_x$ the current predicate being called.

VARIABLE $step$, $step\_x$

Variables to limit the number of monitors crashes that can occur over a behavior.
This variable is used to limit the search space.

VARIABLE $number\_refreshes$

Variables initialization

$$
\begin{aligned}
global\_vars &\triangleq \langle epoch, messages, message\_history \rangle \\
state\_vars &\triangleq \langle isLeader, state, phase \rangle \\
restart\_vars &\triangleq \langle uncommitted\_v, uncommitted\_value \rangle \\
data\_vars &\triangleq \langle monitor\_store, values, accepted\_pn, first\_committed, last\_committed \rangle \\
collect\_vars &\triangleq \langle num\_last, peer\_first\_committed, peer\_last\_committed \rangle \\
lease\_vars &\triangleq \langle acked\_lease \rangle \\
commit\_vars &\triangleq \langle pending\_proposal, new\_value, accepted \rangle
\end{aligned}
$$

$$
\begin{aligned}
vars \triangleq \langle & global\_vars, state\_vars, restart\_vars, data\_vars, collect\_vars, \\
& lease\_vars, commit\_vars \rangle
\end{aligned}
$$

$Init\_global\_vars \triangleq$

```
    $\land\ epoch = 1$
    $\land\ messages = \langle\rangle$
    $\land\ message\_history = \{\}$

$Init\_state\_vars\ \triangleq$
    $\land\ isLeader = [mon \in Monitors \mapsto \text{FALSE}]$
    $\land\ state\ \ = [mon \in Monitors \mapsto Nil]$
    $\land\ phase = [mon \in Monitors \mapsto Nil]$

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

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

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

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

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

$Init\ \triangleq$
    $\land\ Init\_global\_vars$
    $\land\ Init\_state\_vars$
    $\land\ Init\_restart\_vars$
    $\land\ Init\_data\_vars$
    $\land\ Init\_collect\_vars$
    $\land\ Init\_lease\_vars$
    $\land\ Init\_commit\_vars$
    $\land\ step = 0 \land step\_x = \text{``init''} \land number\_refreshes = 0$
```

Message manipulation

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

Add message m to the network $msgs$.

$WithMessage(m,\ msgs) \triangleq$

$\quad Append(msgs,\ m)$

Remove message m from the network $msgs$.

$WithoutMessage(m,\ msgs) \triangleq$

$\quad Remove(msgs,\ m)$

Adds the message m to the network.
Variables changed: messages, $message\_history$.

$Send(m) \triangleq$

$\quad \land\ messages' = WithMessage(m,\ messages)$

$\quad \land\ message\_history' = message\_history \cup \{m\}$

$\quad \land\ \text{UNCHANGED}\ \ message\_history$

Adds a set of messages to the network.
Variables changed: messages, $message\_history$.

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

$\quad \land\ messages' = messages \circ SetToSeq(m\_set)$

$\quad \land\ message\_history' = message\_history \cup \{m\_set\}$

$\quad \land\ \text{UNCHANGED}\ \ message\_history$

Removes the request from network and adds a set of messages.
Variables changed: messages, $message\_history$.

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

$\quad \land\ messages' = WithoutMessage(request,\ messages) \circ SetToSeq(response\_set)$

$\quad \land\ message\_history' = message\_history \cup \{response\_set\}$

$\quad \land\ \text{UNCHANGED}\ \ message\_history$

Removes message m from the network.
Variables changed: messages, $message\_history$.

$Discard(m) \triangleq$

$\quad \land\ \ messages' = WithoutMessage(m,\ messages)$

$\quad \land\ \ \text{UNCHANGED}\ \ message\_history$

Removes the request from network and adds the response.
Variables changed: messages, $message\_history$.

$Reply(response,\ request) \triangleq$

$\quad \land\ messages' = WithoutMessage(request,\ WithMessage(response,\ messages))$

$\quad \land\ message\_history' = message\_history \cup \{response\}$

$\quad \land\ \text{UNCHANGED}\ \ message\_history$

Helper predicates

Compute a new unique proposal number for a given monitor.
Example: $oldpn = 305$, $rank(mon) = 5$, $newpn = 405$.

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

 Clear the variable $peer\_first\_committed$.
 Variables changed: $peer\_first\_committed$.
$clear\_peer\_first\_committed(mon) \triangleq$
    $peer\_first\_committed' = [peer\_first\_committed$ EXCEPT $![mon] =$
                                     $[m \in Monitors \mapsto -1]]$

 Clear the variable $peer\_last\_committed$.
 Variables changed: $peer\_last\_committed$.
$clear\_peer\_last\_committed(mon) \triangleq$
    $peer\_last\_committed' = [peer\_last\_committed$ EXCEPT $![mon] =$
                                     $[m \in Monitors \mapsto -1]]$

 Store peer values and update $first\_committed$, $last\_committed$ and $monitor\_store$ accordingly.
 Variables changed: values, $first\_committed$, $last\_committed$, $monitor\_store$.
$store\_state(mon, msg) \triangleq$
      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 $i \notin$ DOMAIN $values[mon]$
                    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: $uncommitted\_v$, $uncommitted\_value$.
$check\_and\_correct\_uncommitted(mon) \triangleq$
    IF $uncommitted\_v[mon] \leq last\_committed'[mon]$
     THEN $\land uncommitted\_v' = [uncommitted\_v$ EXCEPT $![mon] = 0]$
          $\land uncommitted\_value' = [uncommitted\_value$ EXCEPT $![mon] = Nil]$
     ELSE UNCHANGED $\langle uncommitted\_v, uncommitted\_value \rangle$

 Trigger new election by incrementing epoch.
```

Variables changed: epoch.

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

Lease phase predicates

Changes *mon* state to *STATE_ACTIVE*.

Variables changed: state.

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

Resets the variable acked lease and adds events to send lease messages to peers.

Variables changed: *acked_lease*, messages, *message_history*, phase.

```
$extend\_lease(mon) \triangleq$
    $\land\ isLeader[mon] = \text{TRUE}$
    $\land\ acked\_lease' = [acked\_lease\ \text{EXCEPT}\ ![mon] =$
        $[m \in Monitors \mapsto \text{IF}\ m = mon\ \text{THEN TRUE ELSE FALSE}]]$

    $\land\ Send\_set($
        ${[type            \mapsto OP\_LEASE,$
           $from            \mapsto mon,$
           $dest            \mapsto dest,$
           $last\_committed \mapsto last\_committed[mon]] : dest \in (Monitors \setminus \{mon\})$
          $})$

    $\land\ phase' = [phase\ \text{EXCEPT}\ ![mon] = PHASE\_LEASE]$
```

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.

```
$handle\_lease(mon, msg) \triangleq$
    $\land$  discard if not peon or peon is behind
      $\text{IF}\ \lor\ isLeader[mon] = \text{TRUE}$
           $\lor\ last\_committed[mon] \neq msg.last\_committed$
       $\text{THEN}\ \land\ Discard(msg)$
              $\land\ \text{UNCHANGED}\ state$
       $\text{ELSE}\ \land\ state' = [state\ \text{EXCEPT}\ ![mon] = STATE\_ACTIVE]$
        $\land\ Reply([type             \mapsto OP\_LEASE\_ACK,$
              $from             \mapsto mon,$
              $dest             \mapsto msg.from,$
              $first\_committed \mapsto first\_committed[mon],$
              $last\_committed  \mapsto last\_committed[mon]], msg)$
             $\land\ Discard(msg)$
    $\land\ \text{UNCHANGED}\ \langle epoch, isLeader, phase \rangle$
    ∧ UNCHANGED ⟨restart_vars, data_vars, collect_vars, lease_vars, commit_vars⟩
```

Handle a lease ack message. The leader updates the *acked_lease* variable.
Once 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*.

```
handle_lease_ack(mon, msg) ≜
    ∧ phase[mon] = PHASE_LEASE
    ∧ acked_lease′ = [acked_lease EXCEPT ![mon] =
        [acked_lease[mon] EXCEPT ![msg.from] = TRUE]]
    ∧ Discard(msg)
    ∧ UNCHANGED ⟨epoch⟩
    ∧ UNCHANGED ⟨state_vars, restart_vars, data_vars, collect_vars, commit_vars⟩
```

Predicate that is called when all peers ack the lease. The phase is changed to prevent loops.
Once 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.

```
post_lease_ack(mon) ≜
    ∧ phase[mon] = PHASE_LEASE
    ∧ phase′ = [phase EXCEPT ![mon] = PHASE_LEASE_DONE]
    ∧ ∀ m ∈ Monitors : acked_lease[mon][m] = TRUE
    ∧ UNCHANGED ⟨isLeader, state⟩
    ∧ UNCHANGED ⟨global_vars, restart_vars, data_vars, collect_vars,
                 lease_vars, commit_vars⟩
```

Commit phase predicates

Start a commit phase by the leader. The variable *new_value* is assigned and the events to send
begin messages to the peers are added to *send_queue*.
The value of *uncommitted_v* and *uncommitted_value* are assigned in order for the leader to be
able to recover from a crash/restart.
Variables changed: accepted, *new_value*, phase, messages, *message_history*, values, *uncommitted_v*, *uncommitted_value*.

```
begin(mon, v) ≜
    ∧ isLeader[mon] = TRUE
    ∧ ∨ state′[mon] = STATE_UPDATING
      ∨ state′[mon] = STATE_UPDATING_PREVIOUS
    ∧ Len(ranks) = 1 ∨ num_last[mon] > Len(ranks) ÷ 2
    ∧ new_value[mon] = Nil
    ∧ accepted′ = [accepted EXCEPT ![mon] =
        [m ∈ Monitors ↦ IF m = mon THEN TRUE ELSE FALSE]]
    ∧ new_value′ = [new_value EXCEPT ![mon] = v]
    ∧ phase′ = [phase EXCEPT ![mon] = PHASE_BEGIN]
    ∧ values′ = [values EXCEPT ![mon] =
        (values[mon] @@ ((last_committed[mon] + 1) :> new_value′[mon]))]

    ∧ Send_set(
        {[type             ↦ OP_BEGIN,
          from             ↦ mon,
          dest             ↦ dest,
          last_committed   ↦ last_committed[mon],
          values           ↦ values′[mon],
          pn               ↦ accepted_pn[mon]] : dest ∈ (Monitors \ {mon})
         })

    ∧ uncommitted_v′ = [uncommitted_v EXCEPT ![mon] = last_committed[mon] + 1]
    ∧ uncommitted_value′ = [uncommitted_value 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/restart.

Variables changed: messages, *message_history*, state, values, *uncommitted_v*, *uncommitted_value*.

```
handle_begin(mon, msg) ≜
    ∧ isLeader[mon] = FALSE
    ∧ IF msg.pn < accepted_pn[mon]
        THEN
         ∧ Discard(msg)
         ∧ UNCHANGED ⟨state, restart_vars⟩
        ELSE
         ∧ msg.pn = accepted_pn[mon]
         ∧ msg.last_committed = last_committed[mon]

          assign values[mon][last_committed[mon] + 1]
         ∧ values′ = [values EXCEPT ![mon] =
             (values[mon] @@ ((last_committed[mon] + 1) :> msg.values[last_committed[mon] + 1]))]

         ∧ state′ = [state EXCEPT ![mon] = STATE_UPDATING]
         ∧ uncommitted_v′ = [uncommitted_v EXCEPT ![mon] = last_committed[mon] + 1]
         ∧ uncommitted_value′ = [uncommitted_value EXCEPT ![mon] =
             values′[mon][last_committed[mon] + 1]]

         ∧ Reply([type             ↦ OP_ACCEPT,
                  from             ↦ mon,
                  dest             ↦ msg.from,
                  last_committed   ↦ last_committed[mon],
                  pn               ↦ accepted_pn[mon]], msg)

    ∧ UNCHANGED ⟨epoch, isLeader, phase, monitor_store, accepted_pn, first_committed,
                 last_committed⟩
    ∧ UNCHANGED ⟨collect_vars, lease_vars, commit_vars⟩
```

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

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

Predicate that is enabled and called when all peers accept begin request from leader.
The leader commits the transaction in *new_value* and adds events in *send_queue* to send commit messages to his peers.

Variables changed: *first_committed*, *last_committed*, *monitor_store*, *new_value*, messages, *message_history*, state, phase

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

    $\land\ last\_committed' = [last\_committed$ EXCEPT $![mon] = last\_committed[mon] + 1]$
    $\land$ IF $first\_committed[mon] = 0$
       THEN $first\_committed' = [first\_committed$ EXCEPT $![mon] = first\_committed[mon] + 1]$
       ELSE UNCHANGED $first\_committed$

    $\land\ monitor\_store' = [monitor\_store$ EXCEPT $![mon] = values[mon][last\_committed[mon] + 1]]$
    $\land\ new\_value' = [new\_value$ EXCEPT $![mon] = Nil]$

    $\land\ Send\_set($
       $\{[type             \mapsto OP\_COMMIT,$
         $from              \mapsto mon,$
         $dest              \mapsto dest,$
         $last\_committed   \mapsto last\_committed'[mon],$
```

```
        $pn \qquad\qquad \mapsto accepted\_pn[mon],$
        $values \qquad\quad \mapsto values[mon]] : dest \in (Monitors \setminus \{mon\})$
       $\})$

    $\land\ state' \ = [state \text{ EXCEPT } ![mon] \ = STATE\_REFRESH]$
    $\land\ phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_COMMIT]$
    $\land \text{ UNCHANGED } \langle isLeader, values, accepted\_pn, pending\_proposal, accepted \rangle$
    $\land \text{ UNCHANGED } \langle epoch, restart\_vars, collect\_vars, lease\_vars \rangle$
```

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*.

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

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_value*.

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

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.
This predicate has a big cost on performance, so there were some requirements added (monitor phase and state) to mitigate that.
Variables changed: *pending_proposal*.

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

Start a commit phase with the value on pending proposal.
Variables changed: state, *pending_proposal*, accepted, *new_value*, phase, messages, *message_history*, values, *uncommitted_v*, *uncommitted_value*.

```
$propose\_pending(mon)\ \triangleq$
    $\wedge\ phase[mon] = PHASE\_LEASE \vee phase[mon] = PHASE\_ELECTION$
    $\wedge\ state[mon]\ \ = STATE\_ACTIVE$
    $\wedge\ pending\_proposal[mon] \neq Nil$
    $\wedge\ pending\_proposal' = [pending\_proposal\ \textsc{except}\ ![mon] = Nil]$
    $\wedge\ state' = [state\ \textsc{except}\ ![mon] = STATE\_UPDATING]$
    $\wedge\ begin(mon,\ pending\_proposal[mon])$
    $\wedge\ \textsc{unchanged}\ \langle isLeader,\ monitor\_store,\ accepted\_pn,\ first\_committed,\ last\_committed \rangle$
    $\wedge\ \textsc{unchanged}\ \langle epoch,\ collect\_vars,\ lease\_vars \rangle$
```

Collect phase predicates

Start collect phase. This first part of the collect phase is divided in two parts (collect and *pre_send_collect*) in order to simplify variable changes (when collect is triggered from *handle_last*).
Variables changed: *accepted_pn*, phase.

```
$collect(mon,\ oldpn)\ \triangleq$
    $\wedge\ state[mon] = STATE\_RECOVERING$
    $\wedge\ isLeader[mon] = \textsc{true}$
    $\wedge\ \textsc{let}\ new\_pn\ \triangleq\ get\_new\_proposal\_number(mon,\ Max(\{oldpn,\ accepted\_pn[mon]\}))$
      $\textsc{in}\quad \wedge\ accepted\_pn' = [accepted\_pn\ \textsc{except}\ ![mon] = new\_pn]$
    $\wedge\ phase' = [phase\ \textsc{except}\ ![mon] = PHASE\_PRE\_COLLECT]$
```

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.
Add events to *send_queue* to send collect messages to the peers.
Variables changed: *peer_first_committed*, *peer_last_committed*, *uncommitted_v*, *uncommitted_value*, *num_last*, messages, *message_history*, phase.

```
$pre\_send\_collect(mon)\ \triangleq$
    $\wedge\ state[mon] = STATE\_RECOVERING$
    $\wedge\ isLeader[mon] = \textsc{true}$
    $\wedge\ phase[mon] = PHASE\_PRE\_COLLECT$
    $\wedge\ clear\_peer\_first\_committed(mon)$
    $\wedge\ clear\_peer\_last\_committed(mon)$

    $\wedge\ \textsc{if}\ last\_committed[mon] + 1 \in \textsc{domain}\ values[mon]$
      $\textsc{then}\ \wedge\ uncommitted\_v' =$
                  $[uncommitted\_v\ \textsc{except}\ ![mon] = last\_committed[mon] + 1]$
               $\wedge\ uncommitted\_value' =$
                  $[uncommitted\_value\ \textsc{except}\ ![mon] = values[mon][last\_committed[mon] + 1]]$
      $\textsc{else}\ \ \textsc{unchanged}\ \langle restart\_vars \rangle$
    ∧ num_last′ = [num_last EXCEPT ![mon] = 1]

    ∧ Send_set(
        {[type            ↦ OP_COLLECT,
          from            ↦ mon,
          dest            ↦ dest,
          first_committed ↦ first_committed[mon],
          last_committed  ↦ last_committed[mon],
          pn              ↦ accepted_pn[mon]] : dest ∈ (Monitors \ {mon})
         })

    ∧ phase′ = [phase EXCEPT ![mon] = PHASE_COLLECT]
    ∧ UNCHANGED ⟨isLeader, state⟩
    ∧ UNCHANGED ⟨epoch, data_vars, lease_vars, commit_vars⟩
```

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*

```
handle_collect(mon, msg) ≜
    ∧ isLeader[mon] = FALSE
    ∧ state′ = [state EXCEPT ![mon] = STATE_RECOVERING]
    ∧ ∨ ∧ msg.first_committed > last_committed[mon] + 1
        ∧ bootstrap
        ∧ Discard(msg)
        ∧ UNCHANGED ⟨accepted_pn⟩
      ∨ ∧ msg.first_committed ≤ last_committed[mon] + 1
        ∧ IF msg.pn > accepted_pn[mon]
            THEN accepted_pn′ = [accepted_pn EXCEPT ![mon] = msg.pn]
            ELSE UNCHANGED accepted_pn
        ∧ Reply([type            ↦ OP_LAST,
                 from            ↦ mon,
                 dest            ↦ msg.from,
                 first_committed ↦ first_committed[mon],
                 last_committed  ↦ last_committed[mon],
                 values          ↦ values[mon],
                 pn              ↦ accepted_pn′[mon]], msg)
        ∧ UNCHANGED epoch
    ∧ UNCHANGED ⟨isLeader, phase, values, first_committed, last_committed, monitor_store⟩
    ∧ UNCHANGED ⟨restart_vars, collect_vars, lease_vars, commit_vars⟩
```

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 with that.
Variables changed: messages, *message_history*, epoch, phase, *uncommitted_v*, *uncommitted_value*, *monitor_store*, values, *accepted_pn*, *first_committed*, *last_committed*, *num_last*, *peer_first_committed*, *peer_last_committed*.

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

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

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

             $\land\ \lor\ \land\ msg.pn > accepted\_pn[mon]$
                    $\land\ collect(mon,\ msg.pn)$
                    $\land\ check\_and\_correct\_uncommitted(mon)$
                    $\land\ \text{UNCHANGED}\ num\_last$

                $\lor\ \land\ msg.pn = accepted\_pn[mon]$
                $\land num\_last' = [num\_last \text{ EXCEPT } ![mon] = num\_last[mon] + 1]$

                $\land \text{IF } \land msg.last\_committed + 1 \in \text{DOMAIN } msg.values$
                        $\land msg.last\_committed \geq last\_committed'[mon]$
                        $\land msg.last\_committed + 1 \geq uncommitted\_v[mon]$
                    $\text{THEN } \land uncommitted\_v' =$
                                $[uncommitted\_v \text{ EXCEPT } ![mon] = msg.last\_committed + 1]$
                            $\land uncommitted\_value' =$
                                $[uncommitted\_value \text{ EXCEPT } ![mon] = msg.values[msg.last\_committed + 1]]$
                    $\text{ELSE } check\_and\_correct\_uncommitted(mon)$

                $\land \text{UNCHANGED } \langle phase, accepted\_pn \rangle$

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

  $\land \text{UNCHANGED } \langle isLeader, state \rangle$
  $\land \text{UNCHANGED } \langle lease\_vars, commit\_vars \rangle$
```

Predicate that is enabled and called when all peers 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_v*, *uncommitted_value*, *acked_lease*.

```
$post\_last(mon) \triangleq$
  $\land isLeader[mon] = \text{TRUE}$
  $\land num\_last[mon] = Len(ranks)$
  $\land phase[mon] = PHASE\_COLLECT$

  $\land clear\_peer\_first\_committed(mon)$
  $\land clear\_peer\_last\_committed(mon)$

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

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

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.

```
$leader\_election \triangleq$
    $\land \exists\, mon \in Monitors :$
        $\land\ isLeader' = [m \in Monitors \mapsto \textsc{if}\ m = mon\ \textsc{then}\ \textsc{true}\ \textsc{else}\ \textsc{false}]$
        $\land\ state' = [m \in Monitors \mapsto$
            $\textsc{if}\ Len(ranks) = 1\ \textsc{then}\ STATE\_ACTIVE\ \textsc{else}\ STATE\_RECOVERING]$
    $\land\ phase' = [m \in Monitors \mapsto PHASE\_ELECTION]$
    $\land\ new\_value' = [m \in Monitors \mapsto Nil]$
    $\land\ pending\_proposal' = [m \in Monitors \mapsto Nil]$
    $\land\ epoch' = epoch + 1$
    $\land\ \textsc{unchanged}\ \langle accepted, messages, message\_history \rangle$
    $\land\ \textsc{unchanged}\ \langle data\_vars, restart\_vars, collect\_vars, lease\_vars \rangle$
```

Start recovery phase if number of monitors is greater than 1.

Variables changed: *accepted_pn*, phase.

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

Timeouts and restart

Restart a monitor and wipe variables that are not persistent.

Variables changed: messages, *isLeader*, phase, state, *pending_proposal*, *new_value*, *number_refreshes*.

```
$restart\_mon(mon) \triangleq$
    $\land\ messages' = SelectSeq(messages, \textsc{lambda}\ t : t.from \neq mon)$
    $\land\ isLeader' = [isLeader\ \textsc{except}\ ![mon] = \textsc{false}]$
    $\land\ phase' = [phase\ \textsc{except}\ ![mon] = PHASE\_ELECTION]$
    $\land\ state' = [state\ \textsc{except}\ ![mon] = \textsc{if}\ Len(ranks) = 1$
                                              $\textsc{then}\ STATE\_ACTIVE$
                                              $\textsc{else}\ STATE\_RECOVERING]$
    $\land\ pending\_proposal' = [pending\_proposal\ \textsc{except}\ ![mon] = Nil]$
    $\land\ new\_value' = [new\_value\ \textsc{except}\ ![mon] = Nil]$
    $\land\ number\_refreshes' = number\_refreshes + 1$
    $\land\ \textsc{unchanged}\ \langle epoch, message\_history, accepted \rangle$
    $\land\ \textsc{unchanged}\ \langle restart\_vars, data\_vars, collect\_vars, lease\_vars \rangle$
```

Monitor timeout (simulate message not received). Triggers new elections.

Messages in network and events in *send_queue* are cleared.

Variables changed: epoch, *send_queue*, messages.

```
$Timeout(mon) \triangleq$
    $\land\ \ phase[mon] = PHASE\_COLLECT \lor phase[mon] = PHASE\_BEGIN$
    $\land$ $bootstrap$
    $\land$ $messages' = \langle\rangle$
    $\land$ UNCHANGED $\langle message\_history, state\_vars, restart\_vars, data\_vars, collect\_vars,$
                      $lease\_vars, commit\_vars\rangle$
```

Dispatchers and next statement

```
Handle a message.
$Receive(msg) \triangleq$
    $\land$ $\lor phase[msg.dest] = PHASE\_COLLECT$
           $\lor phase[msg.dest] = PHASE\_BEGIN$
           $\lor phase[msg.dest] = PHASE\_ELECTION$
    $\land$
        $\lor \land msg.type = OP\_COLLECT$
           $\land handle\_collect(msg.dest, msg)$
           $\land step\_x' =$ "receive collect"

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

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

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

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

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

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

Limit some variables to reduce search space.
$reduce\_search\_space \triangleq$
    $\land epoch \neq 6 \land number\_refreshes \neq 2$
    $\land \exists mon \in Monitors : last\_committed[mon] = 2$
        $\Rightarrow \forall mon2 \in Monitors : new\_value[mon2] = Nil$
    $\land \forall mon \in Monitors : accepted\_pn[mon] < 300$
```

```
State transitions.
$Next \triangleq$
    $\land\ reduce\_search\_space$
    $\land$ IF $epoch \% 2 = 1$ THEN
        $\land\ leader\_election$
        $\land\ step\_x' = \text{“election”} \land step' = step + 1$
        $\land$ UNCHANGED $number\_refreshes$
      ELSE
        $\lor$
            $\land\ \exists\, mon \in Monitors : election\_recover(mon)$
            $\land\ step\_x' = \text{“election\_recover”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : pre\_send\_collect(mon)$
            $\land\ step\_x' = \text{“pre\_send\_collect”}\ \ \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : post\_last(mon)$
            $\land\ step\_x' = \text{“post\_last”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : post\_lease\_ack(mon)$
            $\land\ step\_x' = \text{“post\_lease\_ack”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : post\_accept(mon)$
            $\land\ step\_x' = \text{“post\_accept”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : finish\_commit(mon)$
            $\land\ step\_x' = \text{“finish\_commit”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : \exists\, v \in Value\_set : client\_request(mon,\ v)$
            $\land\ step\_x' = \text{“client\_request”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : propose\_pending(mon)$
            $\land\ step\_x' = \text{“propose\_pending”} \land step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, i \in 1 \mathrel{..} Len(messages) : Receive(messages[i])$
            $\land\ step' = step + 1$
            $\land$ UNCHANGED $number\_refreshes$

        $\lor$  $\land\ \exists\, mon \in Monitors : restart\_mon(mon)$
            $\land\ step\_x' = \text{“restart mon”} \land step' = step + 1$
```

```
$\lor \; \land \exists\, mon \in Monitors : Timeout(mon)$
    $\land\, step\_x' = \text{“timeout and restart”} \land step' = step + 1$
    $\land\, \text{UNCHANGED}\ \ number\_refreshes$
```

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'.

$Inv\_diam(size) \;\triangleq\; step \neq size - 1$

Invariants to test in model check

```
$Inv \;\triangleq\; \land\, \text{TRUE}$
          $\land\, Inv\_diam(20)$
```

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.

```
$Inv\_find\_state($
  $\land\, step\_x = \text{“restart mon”}$

  \* 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.
  $\land \forall\, i \in 1 \,..\, Len(messages) :$
    $messages[i].type \neq OP\_COLLECT$

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

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

```
$Inv\_find\_state($
  $\land\, step\_x = \text{“restart mon”}$
  $\land \exists\, i \in 1 \,..\, Len(messages) :$
    $messages[i].type = OP\_ACCEPT$
```

```
   $\land \forall mon \in Monitors :$
      $isLeader[mon] = \textsc{false}$
   $\land epoch = 2$
)
```

Note: After finding a state, that complete state can be used as an initial state to analyze behaviors from there.

\ ∗ Modification History
\ ∗ Last modified *Tue Mar* 09 13:31:06 WET 2021 by *afonsonf*
\ ∗ Created *Mon Jan* 11 16:15:26 WET 2021 by *afonsonf*