

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/afonsof/ceph-consensus-spec>

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

Constants

If true run in debug mode.

$DEBUG \triangleq \text{FALSE}$

Set of *Monitors*.

CONSTANTS *Monitors*

$MonitorsSeq \triangleq TLCVal(SetToSeq(Monitors))$

$MonitorsLen \triangleq TLCVal(Len(MonitorsSeq))$

Rank predicate, used to compute proposal numbers.

$rank(mon) \triangleq \text{CHOOSE } i \in 1 \dots MonitorsLen : MonitorsSeq[i] = mon$

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

$state_names \triangleq \{STATE_RECOVERING, STATE_ACTIVE, STATE_UPDATING, STATE_UPDATING_PREVIOUS, STATE_WRITING, STATE_WRITING_PREVIOUS, STATE_REFRESH, STATE_SHUTDOWN\}$

Paxos auxiliary phase states:

They are used to force some sequence of steps.

CONSTANTS *PHASE_ELECTION*,
PHASE_SEND_COLLECT, *PHASE_COLLECT*,
PHASE_LEASE, *PHASE_LEASE_DONE*,
PHASE_BEGIN,
PHASE_COMMIT

$phase_names \triangleq \{PHASE_ELECTION, PHASE_SEND_COLLECT, PHASE_COLLECT, PHASE_LEASE, PHASE_LEASE_DONE, PHASE_BEGIN, PHASE_COMMIT\}$

Paxos message types:

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

$messages_types \triangleq \{OP_COLLECT, OP_LAST, OP_BEGIN, OP_ACCEPT, OP_COMMIT, OP_LEASE, OP_LEASE_ACK\}$

Global variables

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

Type: Integer

VARIABLE *epoch*

Store messages waiting to be handled.

Type: $[Monitors \mapsto [Monitors \mapsto \langle message \rangle]]$

VARIABLE *messages*

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

Type: $\{messages\}$

VARIABLE *message_history*

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

Type: $[Monitors \mapsto Bool]$

VARIABLE *quorum*

Size of the current quorum.

Type: *Int*

VARIABLE *quorum_sz*

State variables

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

Type: [*Monitors* \mapsto *Bool*]

VARIABLE *isLeader*

A function that stores the state of each monitor.

Type: [*Monitors* \mapsto *state_names*]

VARIABLE *state*

A function that stores the phase of each monitor.

Type: [*Monitors* \mapsto *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* \mapsto 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* \mapsto *Value_set*]

VARIABLE *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.

Type: [*Monitors* \mapsto *Value_set*]

VARIABLE *monitor_store*

A function that stores the transaction log of each monitor.

Type: [*Monitors* \mapsto [value version \mapsto *Value_set*]]

VARIABLE *values*

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

Type: [*Monitors* \mapsto proposal number]

VARIABLE *accepted_pn*

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

Type: [*Monitors* \mapsto value version]

VARIABLE *first_committed*

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

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

VARIABLE *last_committed*

Collect phase variables

A function that stores the number of peers that accepted a collect request.

Type: $[Monitors \mapsto \text{number of peers that accepted}]$

VARIABLE *num_last*

Used by leader when receiving responses in collect phase.

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

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: $[Monitors \mapsto [Monitors \mapsto \text{Bool}]]$

VARIABLE *acked_lease*

Commit phase variables

A function that stores, for each monitor, the value proposed by a client.

Type: $[Monitors \mapsto \text{Value_set} \cup \{\text{Nil}\}]$

VARIABLE *pending_proposal*

A function that stores, for each monitor, the value to be committed in the begin phase.

Type: $[Monitors \mapsto \text{Value_set} \cup \{\text{Nil}\}]$

VARIABLE *new_value*

A function that stores, for each monitor, which of the peers have acked the begin request.

Type: $[Monitors \mapsto [Monitors \mapsto \text{Bool}]]$

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_crashes*

Variables initialization

$$\begin{aligned}
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_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 \\
vars &\triangleq \langle global_vars, state_vars, restart_vars, data_vars, collect_vars, \\
&\quad lease_vars, commit_vars \rangle \\
Init_global_vars &\triangleq \\
&\quad \wedge epoch = 1 \\
&\quad \wedge messages = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \langle \rangle]] \\
&\quad \wedge message_history = \{\} \\
&\quad \wedge quorum = [mon \in Monitors \mapsto \text{TRUE}] \\
&\quad \wedge quorum_sz = MonitorsLen \\
Init_state_vars &\triangleq \\
&\quad \wedge isLeader = [mon \in Monitors \mapsto \text{FALSE}] \\
&\quad \wedge state = [mon \in Monitors \mapsto Nil] \\
&\quad \wedge phase = [mon \in Monitors \mapsto Nil] \\
Init_restart_vars &\triangleq \\
&\quad \wedge uncommitted_v = [mon \in Monitors \mapsto 0] \\
&\quad \wedge uncommitted_value = [mon \in Monitors \mapsto Nil] \\
Init_data_vars &\triangleq \\
&\quad \wedge monitor_store = [mon \in Monitors \mapsto Nil] \\
&\quad \wedge values = [mon \in Monitors \mapsto [version \in \{\} \mapsto Nil]] \\
&\quad \wedge accepted_pn = [mon \in Monitors \mapsto 0] \\
&\quad \wedge first_committed = [mon \in Monitors \mapsto 0] \\
&\quad \wedge last_committed = [mon \in Monitors \mapsto 0] \\
Init_collect_vars &\triangleq \\
&\quad \wedge num_last = [mon \in Monitors \mapsto 0] \\
&\quad \wedge peer_first_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]] \\
&\quad \wedge peer_last_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]] \\
Init_lease_vars &\triangleq \\
&\quad \wedge acked_lease = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \text{FALSE}]] \\
Init_commit_vars &\triangleq \\
&\quad \wedge pending_proposal = [mon \in Monitors \mapsto Nil] \\
&\quad \wedge new_value = [mon \in Monitors \mapsto Nil] \\
&\quad \wedge accepted = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \text{FALSE}]]
\end{aligned}$$

$$\begin{aligned}
Init &\triangleq \\
&\wedge Init_global_vars \\
&\wedge Init_state_vars \\
&\wedge Init_restart_vars \\
&\wedge Init_data_vars \\
&\wedge Init_collect_vars \\
&\wedge Init_lease_vars \\
&\wedge Init_commit_vars \\
&\wedge step = 0 \wedge step_x = \text{"init"} \wedge number_crashes = 0
\end{aligned}$$

Message manipulation

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

Add message *m* to the network *msgs*.

$$\begin{aligned}
WithMessage(m, msgs) &\triangleq \\
&[msgs \text{ EXCEPT } ![m.from] = \\
&\quad [msgs[m.from] \text{ EXCEPT } ![m.dest] = Append(msgs[m.from][m.dest], m)]]
\end{aligned}$$

Remove message *m* from the network *msgs*.

$$\begin{aligned}
WithoutMessage(m, msgs) &\triangleq \\
&[msgs \text{ EXCEPT } ![m.from] = \\
&\quad [msgs[m.from] \text{ EXCEPT } ![m.dest] = Remove(msgs[m.from][m.dest], m)]]
\end{aligned}$$

Adds the message *m* to the network.

Variables changed: messages, *message_history*.

$$\begin{aligned}
Send(m) &\triangleq \\
&\wedge messages' = WithMessage(m, messages) \\
&\quad \wedge message_history' = message_history \cup \{m\} \\
&\wedge \text{UNCHANGED } message_history
\end{aligned}$$

Adds a set of messages to the network.

Variables changed: messages, *message_history*.

$$\begin{aligned}
Send_set(from, m_set) &\triangleq \\
&\wedge messages' = [messages \text{ EXCEPT } ![from] = \\
&\quad [mon \in Monitors \mapsto \\
&\quad \quad messages[from][mon] \circ SetToSeq(\{m \in m_set : m.dest = mon\})]] \\
&\quad \wedge message_history' = message_history \cup m_set \\
&\wedge \text{UNCHANGED } message_history
\end{aligned}$$

Removes the request from network and adds the response.

Variables changed: messages, *message_history*.

$$\begin{aligned}
Reply(response, request) &\triangleq \\
&\wedge messages' = WithoutMessage(request, WithMessage(response, messages)) \\
&\quad \wedge message_history' = message_history \cup \{response\} \\
&\wedge \text{UNCHANGED } message_history
\end{aligned}$$

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

Variables changed: *messages*, *message_history*.

$$\begin{aligned} \text{Reply_set}(\text{from}, \text{response_set}, \text{request}) &\triangleq \\ &\wedge \text{LET } \text{msgs} \triangleq \text{WithoutMessage}(\text{request}, \text{messages}) \\ &\text{IN } \text{messages}' = [\text{msgs} \text{ EXCEPT } ![from] = \\ &\quad [mon \in \text{Monitors} \mapsto \\ &\quad \quad \text{msgs}[from][mon] \circ \text{SetToSeq}(\{m \in \text{response_set} : m.\text{dest} = mon\})]] \\ &\wedge \text{message_history}' = \text{message_history} \cup \text{response_set} \\ &\wedge \text{UNCHANGED } \text{message_history} \end{aligned}$$

Removes message *m* from the network.

Variables changed: *messages*, *message_history*.

$$\begin{aligned} \text{Discard}(m) &\triangleq \\ &\wedge \text{messages}' = \text{WithoutMessage}(m, \text{messages}) \\ &\wedge \text{UNCHANGED } \text{message_history} \end{aligned}$$

Helper predicates

Computes a new unique proposal number for a given monitor.

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

$$\begin{aligned} \text{get_new_proposal_number}(\text{mon}, \text{oldpn}) &\triangleq \\ ((\text{oldpn} \div 100) + 1) * 100 + \text{rank}(\text{mon}) \end{aligned}$$

Clear the variable *peer_first_committed*.

Variables changed: *peer_first_committed*.

$$\begin{aligned} \text{clear_peer_first_committed}(\text{mon}) &\triangleq \\ \text{peer_first_committed}' &= [\text{peer_first_committed} \text{ EXCEPT } ![mon] = \\ &\quad [m \in \text{Monitors} \mapsto -1]] \end{aligned}$$

Clear the variable *peer_last_committed*.

Variables changed: *peer_last_committed*.

$$\begin{aligned} \text{clear_peer_last_committed}(\text{mon}) &\triangleq \\ \text{peer_last_committed}' &= [\text{peer_last_committed} \text{ EXCEPT } ![mon] = \\ &\quad [m \in \text{Monitors} \mapsto -1]] \end{aligned}$$

Store peer values and update *first_committed*, *last_committed* and *monitor_store* accordingly.

Variables changed: *values*, *first_committed*, *last_committed*, *monitor_store*.

$$\begin{aligned} \text{store_state}(\text{mon}, \text{msg}) &\triangleq \\ &\text{Choose peer values from } \text{mon} \text{ last committed } + 1 \text{ to peer last committed.} \\ &\wedge \text{LET } \text{logs} \triangleq (\text{DOMAIN } \text{msg.values}) \cap (\text{last_committed}[\text{mon}] + 1 \dots \text{msg.last_committed}) \\ &\text{IN } \wedge \text{values}' = [\text{values} \text{ EXCEPT } ![mon] = \\ &\quad [i \in \text{DOMAIN } \text{values}[\text{mon}] \cup \text{logs} \mapsto \\ &\quad \quad \text{IF } i \in \text{logs} \\ &\quad \quad \text{THEN } \text{msg.values}[i] \\ &\quad \quad \text{ELSE } \text{values}[\text{mon}][i]]] \\ &\text{Update last committed and first committed.} \end{aligned}$$

$\wedge last_committed' = [last_committed \text{ EXCEPT } ![mon] = Max(logs \cup \{last_committed[mon]\})]$
 $\wedge \text{IF } logs \neq \{\} \wedge first_committed[mon] = 0$
 $\text{ THEN } first_committed' =$
 $\quad [first_committed \text{ EXCEPT } ![mon] = Min(logs)]$
 $\text{ ELSE } first_committed' =$
 $\quad [first_committed \text{ EXCEPT } ![mon] = Min(logs \cup \{first_committed[mon]\})]$
 Update monitor store.
 $\wedge \text{IF } last_committed'[mon] = 0$
 $\text{ THEN UNCHANGED } monitor_store$
 $\text{ ELSE } monitor_store' = [monitor_store \text{ EXCEPT } ![mon] = values'[mon][last_committed'[mon]]]$
 Check if uncommitted value version is still valid, else reset it.
 Variables changed: *uncommitted_v*, *uncommitted_value*.
 $check_and_correct_uncommitted(mon) \triangleq$
 $\text{IF } uncommitted_v[mon] \leq last_committed'[mon]$
 $\text{ THEN } \wedge uncommitted_v' = [uncommitted_v \text{ EXCEPT } ![mon] = 0]$
 $\quad \wedge uncommitted_value' = [uncommitted_value \text{ EXCEPT } ![mon] = Nil]$
 $\text{ ELSE UNCHANGED } \langle uncommitted_v, uncommitted_value \rangle$
 Trigger new election by incrementing epoch.
 Variables changed: *epoch*.
 $bootstrap \triangleq$
 $\wedge epoch' = epoch + 1$

Lease phase predicates

Changes *mon* state to *STATE_ACTIVE*.
 Variables changed: *state*.
 $finish_round(mon) \triangleq$
 $\wedge isLeader[mon] = \text{TRUE}$
 $\wedge state' = [state \text{ EXCEPT } ![mon] = STATE_ACTIVE]$
 Resets the variable *acked_lease* and send lease messages to peers.
 Variables changed: *acked_lease*, *messages*, *message_history*, *phase*.
 $extend_lease(mon) \triangleq$
 $\wedge isLeader[mon] = \text{TRUE}$
 $\wedge acked_lease' = [acked_lease \text{ EXCEPT } ![mon] =$
 $\quad [m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]]$
 $\wedge Send_set(mon,$
 $\quad \{[type \mapsto OP_LEASE,$
 $\quad \quad from \mapsto mon,$
 $\quad \quad dest \mapsto dest,$
 $\quad \quad last_committed \mapsto last_committed[mon]] : dest \in \{m \in Monitors \setminus \{mon\} : quorum[m]\}$
 $\quad \})$
 $\wedge phase' = [phase \text{ EXCEPT } ![mon] = PHASE_LEASE]$

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*.

$$\begin{aligned}
& \text{handle_lease}(mon, msg) \triangleq \\
& \quad \wedge \text{discard if not peon or peon is behind} \\
& \quad \text{IF } \vee isLeader[mon] = \text{TRUE} \\
& \quad \quad \vee last_committed[mon] \neq msg.last_committed \\
& \quad \text{THEN } \wedge Discard(msg) \\
& \quad \quad \wedge \text{UNCHANGED } state \\
& \quad \text{ELSE } \wedge state' = [state \text{ EXCEPT } ![mon] = STATE_ACTIVE] \\
& \quad \quad \wedge \text{Reply}([type \mapsto OP_LEASE_ACK, \\
& \quad \quad \quad from \mapsto mon, \\
& \quad \quad \quad dest \mapsto msg.from, \\
& \quad \quad \quad first_committed \mapsto first_committed[mon], \\
& \quad \quad \quad last_committed \mapsto last_committed[mon]], msg) \\
& \quad \quad \wedge Discard(msg) \\
& \quad \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, isLeader, phase \rangle \\
& \quad \wedge \text{UNCHANGED } \langle restart_vars, data_vars, collect_vars, lease_vars, commit_vars \rangle
\end{aligned}$$

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*.

$$\begin{aligned}
& \text{handle_lease_ack}(mon, msg) \triangleq \\
& \quad \wedge phase[mon] = PHASE_LEASE \\
& \quad \wedge acked_lease' = [acked_lease \text{ EXCEPT } ![mon] = \\
& \quad \quad [acked_lease[mon] \text{ EXCEPT } ![msg.from] = \text{TRUE}]] \\
& \quad \wedge Discard(msg) \\
& \quad \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz \rangle \\
& \quad \wedge \text{UNCHANGED } \langle state_vars, restart_vars, data_vars, collect_vars, commit_vars \rangle
\end{aligned}$$

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*.

$$\begin{aligned}
& \text{post_lease_ack}(mon) \triangleq \\
& \quad \wedge phase[mon] = PHASE_LEASE \\
& \quad \wedge phase' = [phase \text{ EXCEPT } ![mon] = PHASE_LEASE_DONE] \\
& \quad \wedge \forall m \in Monitors : quorum[m] \Rightarrow acked_lease[mon][m] = \text{TRUE} \\
& \quad \wedge \text{UNCHANGED } \langle isLeader, state \rangle \\
& \quad \wedge \text{UNCHANGED } \langle global_vars, restart_vars, data_vars, collect_vars, \\
& \quad \quad lease_vars, commit_vars \rangle
\end{aligned}$$

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_v*, *uncommitted_value*.

```

begin(mon, v)  $\triangleq$ 
   $\wedge$  isLeader[mon] = TRUE
   $\wedge$   $\vee$  state'[mon] = STATE_UPDATING
     $\vee$  state'[mon] = STATE_UPDATING_PREVIOUS
   $\wedge$  quorum_sz = 1  $\vee$  num_last[mon] > MonitorsLen  $\div$  2
   $\wedge$  new_value[mon] = Nil
   $\wedge$  accepted' = [accepted EXCEPT ![mon] =
    [m  $\in$  Monitors  $\mapsto$  IF m = mon THEN TRUE ELSE FALSE]]
   $\wedge$  new_value' = [new_value EXCEPT ![mon] = v]
   $\wedge$  phase' = [phase EXCEPT ![mon] = PHASE_BEGIN]
   $\wedge$  values' = [values EXCEPT ![mon] =
    ((last_committed[mon] + 1) :> new_value'[mon]) @@ values[mon]]
   $\wedge$  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 \ {mon} : quorum[m]}
    })
   $\wedge$  uncommitted_v' = [uncommitted_v EXCEPT ![mon] = last_committed[mon] + 1]
   $\wedge$  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.

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

```

handle_begin(mon, msg)  $\triangleq$ 
   $\wedge$  isLeader[mon] = FALSE
   $\wedge$  IF msg.pn < accepted_pn[mon]
    THEN
       $\wedge$  Discard(msg)
       $\wedge$  UNCHANGED <state, values, restart_vars>
    ELSE
       $\wedge$  msg.pn = accepted_pn[mon]
       $\wedge$  msg.last_committed = last_committed[mon]

      assign values[mon][last_committed[mon] + 1]
       $\wedge$  values' = [values EXCEPT ![mon] =
        ((last_committed[mon] + 1) :> msg.values[last_committed[mon] + 1]) @@ values[mon]]

```

$$\begin{aligned}
& \wedge state' = [state \text{ EXCEPT } ![mon] = STATE_UPDATING] \\
& \wedge uncommitted_v' = [uncommitted_v \text{ EXCEPT } ![mon] = last_committed[mon] + 1] \\
& \wedge uncommitted_value' = [uncommitted_value \text{ EXCEPT } ![mon] = \\
& \quad values'[mon][last_committed[mon] + 1]] \\
& \wedge Reply([type \quad \mapsto OP_ACCEPT, \\
& \quad \quad from \quad \mapsto mon, \\
& \quad \quad dest \quad \mapsto msg.from, \\
& \quad \quad last_committed \mapsto last_committed[mon], \\
& \quad \quad pn \quad \mapsto accepted_pn[mon]], msg) \\
& \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, isLeader, phase, monitor_store, \\
& \quad \quad \quad accepted_pn, first_committed, last_committed \rangle \\
& \wedge \text{UNCHANGED } \langle collect_vars, lease_vars, commit_vars \rangle
\end{aligned}$$

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

$$\begin{aligned}
& handle_accept(mon, msg) \triangleq \\
& \quad \wedge isLeader[mon] = \text{TRUE} \\
& \quad \wedge \vee state[mon] = STATE_UPDATING_PREVIOUS \\
& \quad \quad \vee state[mon] = STATE_UPDATING \\
& \quad \wedge phase[mon] = PHASE_BEGIN \\
& \quad \wedge new_value[mon] \neq Nil \\
& \quad \wedge \text{IF } \vee msg.pn \neq accepted_pn[mon] \\
& \quad \quad \vee \wedge last_committed[mon] > 0 \\
& \quad \quad \quad \wedge msg.last_committed < last_committed[mon] - 1 \\
& \quad \quad \text{THEN UNCHANGED } accepted \\
& \quad \quad \text{ELSE } accepted' = [accepted \text{ EXCEPT } ![mon] = \\
& \quad \quad \quad [accepted[mon] \text{ EXCEPT } ![msg.from] = \text{TRUE}]] \\
& \quad \wedge Discard(msg) \\
& \quad \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, pending_proposal, new_value \rangle \\
& \quad \wedge \text{UNCHANGED } \langle restart_vars, state_vars, data_vars, collect_vars, lease_vars \rangle
\end{aligned}$$

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

$$\begin{aligned}
& post_accept(mon) \triangleq \\
& \quad \wedge phase[mon] = PHASE_BEGIN \\
& \quad \wedge \forall m \in Monitors : quorum[m] \Rightarrow accepted[mon][m] = \text{TRUE} \\
& \quad \wedge new_value[mon] \neq Nil \\
& \quad \wedge \vee state[mon] = STATE_UPDATING_PREVIOUS \\
& \quad \quad \vee state[mon] = STATE_UPDATING \\
& \quad \wedge last_committed' = [last_committed \text{ EXCEPT } ![mon] = last_committed[mon] + 1] \\
& \quad \wedge \text{IF } first_committed[mon] = 0 \\
& \quad \quad \text{THEN } first_committed' = [first_committed \text{ EXCEPT } ![mon] = first_committed[mon] + 1] \\
& \quad \quad \text{ELSE UNCHANGED } first_committed
\end{aligned}$$

$\wedge \text{monitor_store}' = [\text{monitor_store} \text{ EXCEPT } ![mon] = \text{values}[mon][\text{last_committed}[mon] + 1]]$
 $\wedge \text{new_value}' = [\text{new_value} \text{ EXCEPT } ![mon] = \text{Nil}]$
 $\wedge \text{Send_set}(mon,$
 $\quad \{[type \quad \mapsto OP_COMMIT,$
 $\quad \quad from \quad \mapsto mon,$
 $\quad \quad dest \quad \mapsto dest,$
 $\quad \quad last_committed \mapsto last_committed'[mon],$
 $\quad \quad pn \quad \mapsto accepted_pn[mon],$
 $\quad \quad values \quad \mapsto \text{values}[mon]] : dest \in \{m \in \text{Monitors} \setminus \{mon\} : \text{quorum}[m]\}$
 $\quad \})$
 $\wedge \text{state}' = [\text{state} \text{ EXCEPT } ![mon] = \text{STATE_REFRESH}]$
 $\wedge \text{phase}' = [\text{phase} \text{ EXCEPT } ![mon] = \text{PHASE_COMMIT}]$
 $\wedge \text{UNCHANGED } \langle isLeader, values, accepted_pn, pending_proposal, accepted \rangle$
 $\wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, restart_vars, collect_vars, lease_vars \rangle$

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*.

$\text{finish_commit}(mon) \triangleq$
 $\quad \wedge \text{state}[mon] = \text{STATE_REFRESH}$
 $\quad \wedge \text{phase}[mon] = \text{PHASE_COMMIT}$
 $\quad \wedge \text{finish_round}(mon)$
 $\quad \wedge \text{extend_lease}(mon)$
 $\quad \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, isLeader \rangle$
 $\quad \wedge \text{UNCHANGED } \langle restart_vars, data_vars, collect_vars, commit_vars \rangle$

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*.

$\text{handle_commit}(mon, msg) \triangleq$
 $\quad \wedge isLeader[mon] = \text{FALSE}$
 $\quad \wedge \text{store_state}(mon, msg)$
 $\quad \wedge \text{check_and_correct_uncommitted}(mon)$
 $\quad \wedge \text{Discard}(msg)$
 $\quad \wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, accepted_pn \rangle$
 $\quad \wedge \text{UNCHANGED } \langle state_vars, collect_vars, lease_vars, commit_vars \rangle$

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*.

$\text{client_request}(mon, v) \triangleq$
 $\quad \wedge isLeader[mon] = \text{TRUE}$
 $\quad \wedge \text{state}[mon] = \text{STATE_ACTIVE}$

$\wedge \text{pending_proposal}[mon] = Nil$
 $\wedge \text{pending_proposal}' = [\text{pending_proposal} \text{ EXCEPT } ![mon] = v]$
 $\wedge \text{UNCHANGED } \langle \text{new_value}, \text{accepted} \rangle$
 $\wedge \text{UNCHANGED } \langle \text{global_vars}, \text{state_vars}, \text{restart_vars}, \text{data_vars}, \text{collect_vars}, \text{lease_vars} \rangle$

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*.

$\text{propose_pending}(mon) \triangleq$
 $\wedge \text{phase}[mon] = \text{PHASE_LEASE} \vee \text{phase}[mon] = \text{PHASE_ELECTION}$
 $\wedge \text{state}[mon] = \text{STATE_ACTIVE}$
 $\wedge \text{pending_proposal}[mon] \neq Nil$
 $\wedge \text{pending_proposal}' = [\text{pending_proposal} \text{ EXCEPT } ![mon] = Nil]$
 $\wedge \text{state}' = [\text{state} \text{ EXCEPT } ![mon] = \text{STATE_UPDATING}]$
 $\wedge \text{begin}(mon, \text{pending_proposal}[mon])$
 $\wedge \text{UNCHANGED } \langle \text{isLeader}, \text{monitor_store}, \text{accepted_pn}, \text{first_committed}, \text{last_committed} \rangle$
 $\wedge \text{UNCHANGED } \langle \text{epoch}, \text{quorum}, \text{quorum_sz}, \text{collect_vars}, \text{lease_vars} \rangle$

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*.

$\text{collect}(mon, \text{oldpn}) \triangleq$
 $\wedge \text{state}[mon] = \text{STATE_RECOVERING}$
 $\wedge \text{isLeader}[mon] = \text{TRUE}$
 $\wedge \text{LET } \text{new_pn} \triangleq \text{get_new_proposal_number}(mon, \text{Max}(\{\text{oldpn}, \text{accepted_pn}[mon]\}))$
 $\quad \text{IN } \wedge \text{accepted_pn}' = [\text{accepted_pn} \text{ EXCEPT } ![mon] = \text{new_pn}]$
 $\wedge \text{phase}' = [\text{phase} \text{ EXCEPT } ![mon] = \text{PHASE_SEND_COLLECT}]$

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_v*, *uncommitted_value*, *num_last*, *messages*, *message_history*, *phase*.

$\text{send_collect}(mon) \triangleq$
 $\wedge \text{state}[mon] = \text{STATE_RECOVERING}$
 $\wedge \text{isLeader}[mon] = \text{TRUE}$
 $\wedge \text{phase}[mon] = \text{PHASE_SEND_COLLECT}$
 $\wedge \text{clear_peer_first_committed}(mon)$
 $\wedge \text{clear_peer_last_committed}(mon)$
 $\wedge \text{IF } \text{last_committed}[mon] + 1 \in \text{DOMAIN } \text{values}[mon]$
 $\quad \text{THEN } \wedge \text{uncommitted_v}' =$
 $\quad \quad [\text{uncommitted_v} \text{ EXCEPT } ![mon] = \text{last_committed}[mon] + 1]$

$\wedge \text{uncommitted_value}' =$
 $\quad [\text{uncommitted_value} \text{ EXCEPT } ![mon] = \text{values}[mon][\text{last_committed}[mon] + 1]]$
 $\text{ELSE UNCHANGED } \langle \text{restart_vars} \rangle$
 $\wedge \text{num_last}' = [\text{num_last} \text{ EXCEPT } ![mon] = 1]$
 $\wedge \text{Send_set}(mon,$
 $\quad \{[type \mapsto OP_COLLECT,$
 $\quad \quad from \mapsto mon,$
 $\quad \quad dest \mapsto dest,$
 $\quad \quad first_committed \mapsto first_committed[mon],$
 $\quad \quad last_committed \mapsto last_committed[mon],$
 $\quad \quad pn \mapsto \text{accepted_pn}[mon]] : dest \in \{m \in \text{Monitors} \setminus \{mon\} : \text{quorum}[m]\}$
 $\quad \})$
 $\wedge \text{phase}' = [\text{phase} \text{ EXCEPT } ![mon] = PHASE_COLLECT]$
 $\wedge \text{UNCHANGED } \langle isLeader, state \rangle$
 $\wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, data_vars, lease_vars, commit_vars \rangle$

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

$\text{handle_collect}(mon, msg) \triangleq$
 $\wedge isLeader[mon] = \text{FALSE}$
 $\wedge \text{state}' = [\text{state} \text{ EXCEPT } ![mon] = STATE_RECOVERING]$
 $\wedge \vee \wedge msg.first_committed > last_committed[mon] + 1$
 $\quad \wedge bootstrap$
 $\quad \wedge Discard(msg)$
 $\quad \wedge \text{UNCHANGED } \langle \text{accepted_pn} \rangle$
 $\vee \wedge msg.first_committed \leq last_committed[mon] + 1$
 $\quad \wedge \text{IF } msg.pn > \text{accepted_pn}[mon]$
 $\quad \quad \text{THEN } \text{accepted_pn}' = [\text{accepted_pn} \text{ EXCEPT } ![mon] = msg.pn]$
 $\quad \quad \text{ELSE UNCHANGED } \text{accepted_pn}$
 $\quad \wedge Reply([type \mapsto OP_LAST,$
 $\quad \quad from \mapsto mon,$
 $\quad \quad dest \mapsto msg.from,$
 $\quad \quad first_committed \mapsto first_committed[mon],$
 $\quad \quad last_committed \mapsto last_committed[mon],$
 $\quad \quad values \mapsto values[mon],$
 $\quad \quad pn \mapsto \text{accepted_pn}'[mon]], msg)$
 $\quad \wedge \text{UNCHANGED } epoch$
 $\wedge \text{UNCHANGED } \langle isLeader, phase, values, first_committed, last_committed, monitor_store \rangle$
 $\wedge \text{UNCHANGED } \langle quorum, quorum_sz, restart_vars, collect_vars, lease_vars, commit_vars \rangle$

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_v*, *uncommitted_value*, *monitor_store*, *values*, *accepted_pn*, *first_committed*, *last_committed*, *num_last*, *peer_first_committed*, *peer_last_committed*.

$$\begin{aligned}
& \text{handle_last}(\text{mon}, \text{msg}) \triangleq \\
& \quad \wedge \text{isLeader}[\text{mon}] = \text{TRUE} \\
& \quad \wedge \text{peer_first_committed}' = [\text{peer_first_committed} \text{ EXCEPT } ![\text{mon}] = \\
& \quad \quad [\text{peer_first_committed}[\text{mon}] \text{ EXCEPT } ![\text{msg.from}] = \text{msg.first_committed}]] \\
& \quad \wedge \text{peer_last_committed}' = [\text{peer_last_committed} \text{ EXCEPT } ![\text{mon}] = \\
& \quad \quad [\text{peer_last_committed}[\text{mon}] \text{ EXCEPT } ![\text{msg.from}] = \text{msg.last_committed}]] \\
& \quad \wedge \text{IF } \text{msg.first_committed} > \text{last_committed}[\text{mon}] + 1 \\
& \quad \quad \text{THEN} \\
& \quad \quad \quad \wedge \text{bootstrap} \\
& \quad \quad \quad \wedge \text{Discard}(\text{msg}) \\
& \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{num_last}, \text{accepted_pn}, \text{values}, \text{phase}, \text{monitor_store} \rangle \\
& \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{first_committed}, \text{last_committed}, \text{restart_vars} \rangle \\
& \quad \quad \text{ELSE} \\
& \quad \quad \quad \wedge \text{store_state}(\text{mon}, \text{msg}) \\
& \quad \quad \quad \wedge \text{IF } \exists \text{peer} \in \text{Monitors} : \\
& \quad \quad \quad \quad \wedge \text{peer} \neq \text{mon} \\
& \quad \quad \quad \quad \wedge \text{peer_last_committed}'[\text{mon}][\text{peer}] \neq -1 \\
& \quad \quad \quad \quad \wedge \text{peer_last_committed}'[\text{mon}][\text{peer}] + 1 < \text{first_committed}[\text{mon}] \\
& \quad \quad \quad \quad \wedge \text{first_committed}[\text{mon}] > 1 \\
& \quad \quad \quad \text{THEN} \\
& \quad \quad \quad \quad \wedge \text{bootstrap} \\
& \quad \quad \quad \quad \wedge \text{check_and_correct_uncommitted}(\text{mon}) \\
& \quad \quad \quad \quad \wedge \text{Discard}(\text{msg}) \\
& \quad \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{phase}, \text{accepted_pn}, \text{num_last} \rangle \\
& \quad \quad \quad \text{ELSE} \\
& \quad \quad \quad \text{LET } \text{monitors_behind} \triangleq \{ \text{peer} \in \text{Monitors} : \\
& \quad \quad \quad \quad \wedge \text{peer} \neq \text{mon} \\
& \quad \quad \quad \quad \wedge \text{peer_last_committed}'[\text{mon}][\text{peer}] \neq -1 \\
& \quad \quad \quad \quad \wedge \text{peer_last_committed}'[\text{mon}][\text{peer}] < \text{last_committed}[\text{mon}] \\
& \quad \quad \quad \quad \wedge \text{quorum}[\text{peer}] \} \\
& \quad \quad \text{IN } \text{Reply_set}(\text{mon}, \\
& \quad \quad \quad \{ [\text{type} \mapsto \text{OP_COMMIT}, \\
& \quad \quad \quad \text{from} \mapsto \text{mon}, \\
& \quad \quad \quad \text{dest} \mapsto \text{dest}, \\
& \quad \quad \quad \text{last_committed} \mapsto \text{last_committed}'[\text{mon}], \\
& \quad \quad \quad \text{pn} \mapsto \text{accepted_pn}[\text{mon}], \\
& \quad \quad \quad \text{values} \mapsto \text{values}[\text{mon}]] : \text{dest} \in \text{monitors_behind} \\
& \quad \quad \quad \}, \text{msg}) \\
& \quad \wedge \vee \wedge \text{msg.pn} > \text{accepted_pn}[\text{mon}] \\
& \quad \quad \wedge \text{collect}(\text{mon}, \text{msg.pn}) \\
& \quad \quad \wedge \text{check_and_correct_uncommitted}(\text{mon})
\end{aligned}$$

$\wedge \text{UNCHANGED } num_last$
 $\vee \wedge msg.pn = accepted_pn[mon]$
 $\wedge num_last' = [num_last \text{ EXCEPT } ![mon] = num_last[mon] + 1]$
 $\wedge \text{IF } \wedge msg.last_committed + 1 \in \text{DOMAIN } msg.values$
 $\wedge msg.last_committed \geq last_committed'[mon]$
 $\wedge msg.last_committed + 1 \geq uncommitted_v[mon]$
 $\text{THEN } \wedge uncommitted_v' =$
 $\quad [uncommitted_v \text{ EXCEPT } ![mon] = msg.last_committed + 1]$
 $\wedge uncommitted_value' =$
 $\quad [uncommitted_value \text{ EXCEPT } ![mon] = msg.values[msg.last_committed + 1]]$
 $\text{ELSE } check_and_correct_uncommitted(mon)$
 $\wedge \text{UNCHANGED } \langle phase, accepted_pn \rangle$
 $\vee \wedge msg.pn < accepted_pn[mon]$
 $\wedge check_and_correct_uncommitted(mon)$
 $\wedge \text{UNCHANGED } \langle phase, accepted_pn, num_last \rangle$
 $\wedge \text{UNCHANGED } epoch$
 $\wedge \text{UNCHANGED } \langle epoch \rangle$
 $\wedge \text{UNCHANGED } \langle quorum, quorum_sz, isLeader, state \rangle$
 $\wedge \text{UNCHANGED } \langle lease_vars, commit_vars \rangle$

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_v*, *uncommitted_value*, *acked_lease*.

$post_last(mon) \triangleq$
 $\wedge isLeader[mon] = \text{TRUE}$
 $\wedge num_last[mon] = quorum_sz$
 $\wedge phase[mon] = PHASE_COLLECT$
 $\wedge clear_peer_first_committed(mon)$
 $\wedge clear_peer_last_committed(mon)$
 $\wedge \text{IF } \wedge uncommitted_v[mon] = last_committed[mon] + 1$
 $\quad \wedge uncommitted_value[mon] \neq Nil$
 $\text{THEN } \wedge state' = [state \text{ EXCEPT } ![mon] = STATE_UPDATING_PREVIOUS]$
 $\quad \wedge begin(mon, uncommitted_value[mon])$
 $\quad \wedge \text{UNCHANGED } \langle acked_lease \rangle$
 $\text{ELSE } \wedge finish_round(mon)$
 $\quad \wedge extend_lease(mon)$
 $\quad \wedge \text{UNCHANGED } \langle accepted, new_value, values, restart_vars \rangle$
 $\wedge \text{UNCHANGED } \langle isLeader, monitor_store, accepted_pn, first_committed, last_committed \rangle$
 $\wedge \text{UNCHANGED } \langle epoch, quorum, quorum_sz, num_last, pending_proposal \rangle$

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*.

$leader_election \triangleq$

- $\wedge \exists mon \in Monitors :$
 - $\wedge quorum[mon]$
 - $\wedge isLeader' = [m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]$
 - $\wedge state' = [m \in Monitors \mapsto$
 - $\text{IF } quorum_sz = 1 \text{ THEN } STATE_ACTIVE \text{ ELSE } STATE_RECOVERING}]$
 - $\wedge phase' = [m \in Monitors \mapsto PHASE_ELECTION]$
 - $\wedge new_value' = [m \in Monitors \mapsto Nil]$
 - $\wedge pending_proposal' = [m \in Monitors \mapsto Nil]$
 - $\wedge epoch' = epoch + 1$
 - $\wedge messages' = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \langle \rangle]]$
 - $\wedge \text{UNCHANGED } \langle quorum, quorum_sz, accepted, message_history \rangle$
 - $\wedge \text{UNCHANGED } \langle data_vars, restart_vars, collect_vars, lease_vars \rangle$

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

Variables changed: *accepted_pn*, *phase*.

$election_recover(mon) \triangleq$

- $\wedge quorum_sz > 1$
- $\wedge phase[mon] = PHASE_ELECTION$
- $\wedge collect(mon, 0)$
- $\wedge \text{UNCHANGED } \langle isLeader, state, values, first_committed, last_committed, monitor_store \rangle$
- $\wedge \text{UNCHANGED } \langle global_vars, restart_vars, collect_vars, lease_vars, commit_vars \rangle$

Timeouts and restart

$crash_mon(mon) \triangleq$

- $\wedge quorum_sz > (MonitorsLen \div 2) + 1$
- $\wedge quorum[mon] = \text{TRUE}$
- $\wedge quorum' = [quorum \text{ EXCEPT } ![mon] = \text{FALSE}]$
- $\wedge quorum_sz' = quorum_sz - 1$
- $\wedge bootstrap$
 - $\wedge number_crashes' = number_crashes + 1$
- $\wedge \text{UNCHANGED } \langle messages, message_history \rangle$
- $\wedge \text{UNCHANGED } \langle state_vars, restart_vars, data_vars, collect_vars, lease_vars, commit_vars \rangle$

$restore_mon(mon) \triangleq$

- $\wedge quorum[mon] = \text{FALSE}$
- $\wedge quorum' = [quorum \text{ EXCEPT } ![mon] = \text{TRUE}]$
- $\wedge quorum_sz' = quorum_sz + 1$
- $\wedge bootstrap$

$\wedge \text{UNCHANGED } \langle \text{messages}, \text{message_history} \rangle$
 $\wedge \text{UNCHANGED } \langle \text{state_vars}, \text{restart_vars}, \text{data_vars}, \text{collect_vars}, \text{lease_vars}, \text{commit_vars} \rangle$

Monitor timeout (simulate the various timeouts that can occur). Triggers new elections.

Variables changed: epoch.

$\text{Timeout}(\text{mon}) \triangleq$
 $\wedge \text{bootstrap}$
 $\wedge \text{UNCHANGED } \langle \text{messages}, \text{quorum}, \text{quorum_sz}, \text{message_history}, \text{state_vars}, \text{restart_vars},$
 $\text{data_vars}, \text{collect_vars}, \text{lease_vars}, \text{commit_vars} \rangle$

Dispatchers and next statement

Handle a message.

$\text{Receive}(\text{msg}) \triangleq$
 $\wedge \vee \wedge \text{msg.type} = \text{OP_COLLECT}$
 $\wedge \text{handle_collect}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive collect"}$

 $\vee \wedge \text{msg.type} = \text{OP_LAST}$
 $\wedge \text{handle_last}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive last"}$

 $\vee \wedge \text{msg.type} = \text{OP_LEASE}$
 $\wedge \text{handle_lease}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive lease"}$

 $\vee \wedge \text{msg.type} = \text{OP_LEASE_ACK}$
 $\wedge \text{handle_lease_ack}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive lease_ack"}$

 $\vee \wedge \text{msg.type} = \text{OP_BEGIN}$
 $\wedge \text{handle_begin}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive begin"}$

 $\vee \wedge \text{msg.type} = \text{OP_ACCEPT}$
 $\wedge \text{handle_accept}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive accept"}$

 $\vee \wedge \text{msg.type} = \text{OP_COMMIT}$
 $\wedge \text{handle_commit}(\text{msg.dest}, \text{msg})$
 $\wedge \text{step_x}' = \text{"receive commit"}$

Limit some variables to reduce search space.

$\text{reduce_search_space} \triangleq$
 $\wedge \text{epoch} \neq 8$
 $\wedge \forall \text{mon} \in \text{Monitors} : \text{last_committed}[\text{mon}] < 2$
 $\vee \forall \text{mon2} \in \text{Monitors} : \text{new_value}[\text{mon2}] = \text{Nil}$

$\wedge \forall mon \in Monitors : accepted_pn[mon] < 300$
 $\wedge number_crashes \neq 4$

State transitions.

$Next \triangleq$

$\wedge reduce_search_space$
 $\wedge \text{IF } DEBUG \text{ THEN } step' = step + 1$
 $\quad \text{ELSE } step' = step$
 $\wedge \text{IF } epoch \% 2 = 1 \text{ THEN}$
 $\quad \wedge leader_election$
 $\quad \wedge step_x' = \text{"election"}$
 $\quad \wedge \text{UNCHANGED } number_crashes$
 ELSE
 $\quad \vee \wedge \exists mon \in Monitors : election_recover(mon)$
 $\quad \quad \wedge step_x' = \text{"election_recover"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : send_collect(mon)$
 $\quad \quad \wedge step_x' = \text{"send_collect"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : post_last(mon)$
 $\quad \quad \wedge step_x' = \text{"post_last"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : post_lease_ack(mon)$
 $\quad \quad \wedge step_x' = \text{"post_lease_ack"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : post_accept(mon)$
 $\quad \quad \wedge step_x' = \text{"post_accept"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : finish_commit(mon)$
 $\quad \quad \wedge step_x' = \text{"finish_commit"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : \exists v \in Value_set : client_request(mon, v)$
 $\quad \quad \wedge step_x' = \text{"client_request"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon \in Monitors : propose_pending(mon)$
 $\quad \quad \wedge step_x' = \text{"propose_pending"}$
 $\quad \quad \wedge \text{UNCHANGED } number_crashes$
 $\quad \vee \wedge \exists mon1, mon2 \in Monitors :$
 $\quad \quad \wedge mon1 \neq mon2$
 $\quad \quad \wedge Len(messages[mon1][mon2]) > 0$

$$\begin{aligned}
& \wedge \text{Receive}(\text{messages}[\text{mon1}][\text{mon2}][1]) \\
& \wedge \text{UNCHANGED } \text{number_crashes} \\
\\
& \vee \wedge \exists \text{mon} \in \text{Monitors} : \text{crash_mon}(\text{mon}) \\
& \quad \wedge \text{step_x}' = \text{"crash_mon"} \\
& \quad \wedge \text{UNCHANGED } \text{number_crashes} \\
\\
& \vee \wedge \exists \text{mon} \in \text{Monitors} : \text{restore_mon}(\text{mon}) \\
& \quad \wedge \text{step_x}' = \text{"restore_mon"} \\
& \quad \wedge \text{UNCHANGED } \text{number_crashes} \\
\\
& \vee \wedge \exists \text{mon} \in \text{Monitors} : \text{Timeout}(\text{mon}) \\
& \quad \wedge \text{step_x}' = \text{"timeout_and_restart"} \\
& \quad \wedge \text{UNCHANGED } \text{number_crashes}
\end{aligned}$$

Safety invariants

If two monitors are in state active then their *monitor_store* must have the same value.

$$\begin{aligned}
\text{same_monitor_store} & \triangleq \forall \text{mon1}, \text{mon2} \in \text{Monitors} : \\
& \text{state}[\text{mon1}] = \text{STATE_ACTIVE} \wedge \text{state}[\text{mon2}] = \text{STATE_ACTIVE} \\
& \Rightarrow \text{monitor_store}[\text{mon1}] = \text{monitor_store}[\text{mon2}]
\end{aligned}$$

$$\text{Inv} \triangleq \wedge \text{same_monitor_store}$$

Test/Debug invariants

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

$$\text{Inv_find_state}(x) \triangleq \neg x$$

Invariant used to search for a behavior of diameter equal to 'size'.

$$\text{Inv_diam}(\text{size}) \triangleq \text{step} \neq \text{size} - 1$$

Invariants to test in model check

$$\begin{aligned}
\text{DEBUG_Inv} & \triangleq \wedge \text{TRUE} \\
& \wedge \text{Inv_diam}(20)
\end{aligned}$$

Examples:

Find a behavior with a diameter of size 60.

$$\text{Inv_diam}(60)$$

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

$$\begin{aligned}
& \text{Inv_find_state} (\\
& \quad \exists \text{msg1}, \text{msg2} \in \text{message_history} : \\
& \quad \quad \wedge \text{msg1.type} = \text{OP_COLLECT} \wedge \text{msg2.type} = \text{OP_COLLECT} \\
& \quad \quad \wedge \text{msg1.from} \neq \text{msg2.from} \\
&)
\end{aligned}$$

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

```

Inv_find_state(
  ∧ step_x = "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.
  ∧ ∃ mon ∈ Monitors :
    ∧ isLeader[mon] = TRUE
    ∧ phase[mon] = PHASE_COLLECT
    ∧ num_last[mon] = 1

  \ * All the collect requests have been handled by the peers.
  ∧ ∀ mon1, mon2 ∈ Monitors :
    ∀ i ∈ 1 .. Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type ≠ OP_COLLECT

  ∧ epoch = 2
)

```

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

```

Inv_find_state(
  ∧ step_x = "crash mon"
  ∧ ∃ mon1, mon2 ∈ Monitors :
    ∃ i ∈ 1 .. Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type = OP_ACCEPT
  ∧ ∀ mon ∈ Monitors :
    isLeader[mon] = FALSE
  ∧ 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 Sat Apr 03 15:41:22 WEST 2021 by afonsonf
\ * Created Mon Jan 11 16:15:26 WET 2021 by afonsonf

```