

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. When a new epoch begins the messages from the previous epoch are discarded.
- The quorum of monitors. 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` represents the 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 value in the variable `monitor_store`.
- Failure model. When 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/afonsof/ceph-consensus-spec>

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

Constants

Monitors.

CONSTANTS *Monitors*

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

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

Sequence of monitors and the 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*

A function that stores messages.

Type: $\langle message \rangle$

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: $[Monitors \mapsto Bool]$

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

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

VARIABLE *values*

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

Type: $[Monitors \mapsto \text{proposal number}]$

VARIABLE *accepted_pn*

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

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

VARIABLE *first_committed*

A function that stores the last value version committed for 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_refreshes*

Variables initialization

$global_vars \triangleq \langle epoch, messages, message_history \rangle$
 $state_vars \triangleq \langle isLeader, state, phase \rangle$
 $restart_vars \triangleq \langle uncommitted_v, uncommitted_value \rangle$
 $data_vars \triangleq \langle monitor_store, values, accepted_pn, first_committed, last_committed \rangle$
 $collect_vars \triangleq \langle num_last, peer_first_committed, peer_last_committed \rangle$
 $lease_vars \triangleq \langle acked_lease \rangle$
 $commit_vars \triangleq \langle pending_proposal, new_value, accepted \rangle$
 $vars \triangleq \langle global_vars, state_vars, restart_vars, data_vars, collect_vars, lease_vars, commit_vars \rangle$

$$\begin{aligned}
Init_global_vars &\triangleq \\
&\wedge epoch = 1 \\
&\wedge messages = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto \langle \rangle]] \\
&\wedge message_history = \{\} \\
Init_state_vars &\triangleq \\
&\wedge isLeader = [mon \in Monitors \mapsto FALSE] \\
&\wedge state = [mon \in Monitors \mapsto Nil] \\
&\wedge phase = [mon \in Monitors \mapsto Nil] \\
Init_restart_vars &\triangleq \\
&\wedge uncommitted_v = [mon \in Monitors \mapsto 0] \\
&\wedge uncommitted_value = [mon \in Monitors \mapsto Nil] \\
Init_data_vars &\triangleq \\
&\wedge monitor_store = [mon \in Monitors \mapsto Nil] \\
&\wedge values = [mon \in Monitors \mapsto [version \in \{\} \mapsto Nil]] \\
&\wedge accepted_pn = [mon \in Monitors \mapsto 0] \\
&\wedge first_committed = [mon \in Monitors \mapsto 0] \\
&\wedge last_committed = [mon \in Monitors \mapsto 0] \\
Init_collect_vars &\triangleq \\
&\wedge num_last = [mon \in Monitors \mapsto 0] \\
&\wedge peer_first_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]] \\
&\wedge peer_last_committed = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto -1]] \\
Init_lease_vars &\triangleq \\
&\wedge acked_lease = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto FALSE]] \\
Init_commit_vars &\triangleq \\
&\wedge pending_proposal = [mon \in Monitors \mapsto Nil] \\
&\wedge new_value = [mon \in Monitors \mapsto Nil] \\
&\wedge accepted = [mon1 \in Monitors \mapsto [mon2 \in Monitors \mapsto FALSE]] \\
Init &\triangleq \\
&\wedge Init_global_vars \\
&\wedge Init_state_vars \\
&\wedge Init_restart_vars \\
&\wedge Init_data_vars \\
&\wedge Init_collect_vars \\
&\wedge Init_lease_vars \\
&\wedge Init_commit_vars \\
&\wedge step = 0 \wedge step_x = \text{"init"} \wedge number_refreshes = 0
\end{aligned}$$

Message manipulation

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

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

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

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

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

Adds the message *m* to the network.

Variables changed: *messages*, *message_history*.

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

Adds a set of messages to the network.

Variables changed: *messages*, *message_history*.

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

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

Variables changed: *messages*, *message_history*.

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

Removes message *m* from the network.

Variables changed: *messages*, *message_history*.

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

Removes the request from network and adds the response.

Variables changed: *messages*, *message_history*.

$$\text{Reply}(\text{response}, \text{request}) \triangleq$$

$\wedge \text{messages}' = \text{WithoutMessage}(\text{request}, \text{WithMessage}(\text{response}, \text{messages}))$
 $\wedge \text{message_history}' = \text{message_history} \cup \{\text{response}\}$
 $\wedge \text{UNCHANGED } \text{message_history}$

Helper predicates

Compute a new unique proposal number for a given monitor.

Example: $\text{oldpn} = 305$, $\text{rank}(\text{mon}) = 5$, $\text{newpn} = 405$.

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

Clear the variable $\text{peer_first_committed}$.

Variables changed: $\text{peer_first_committed}$.

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

Clear the variable $\text{peer_last_committed}$.

Variables changed: $\text{peer_last_committed}$.

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

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

Variables changed: values , first_committed , last_committed , monitor_store .

$\text{store_state}(\text{mon}, \text{msg}) \triangleq$

Choose peer values from mon last committed + 1 to peer last committed.

$\wedge \text{LET } \text{logs} \triangleq (\text{DOMAIN } \text{msg.values}) \cap (\text{last_committed}[\text{mon}] + 1 .. \text{msg.last_committed})$
 $\text{IN } \wedge \text{values}' = [\text{values} \text{ EXCEPT } ![\text{mon}] =$
 $[i \in \text{DOMAIN } \text{values}[\text{mon}] \cup \text{logs} \mapsto$
 $\text{IF } i \notin \text{DOMAIN } \text{values}[\text{mon}]$
 $\text{THEN } \text{msg.values}[i]$
 $\text{ELSE } \text{values}[\text{mon}][i]]$

Update last committed and first committed.

$\wedge \text{last_committed}' = [\text{last_committed} \text{ EXCEPT } ![\text{mon}] = \text{Max}(\text{logs} \cup \{\text{last_committed}[\text{mon}]\})]$
 $\wedge \text{IF } \text{logs} \neq \{\}$ $\wedge \text{first_committed}[\text{mon}] = 0$
 $\text{THEN } \text{first_committed}' =$
 $[\text{first_committed} \text{ EXCEPT } ![\text{mon}] = \text{Min}(\text{logs})]$
 $\text{ELSE } \text{first_committed}' =$
 $[\text{first_committed} \text{ EXCEPT } ![\text{mon}] = \text{Min}(\text{logs} \cup \{\text{first_committed}[\text{mon}]\})]$

Update monitor store.

$\wedge \text{IF } \text{last_committed}'[\text{mon}] = 0$
 $\text{THEN UNCHANGED } \text{monitor_store}$
 $\text{ELSE } \text{monitor_store}' = [\text{monitor_store} \text{ EXCEPT } ![\text{mon}] = \text{values}'[\text{mon}][\text{last_committed}'[\text{mon}]]]$

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 $\wedge uncommitted_v' = [uncommitted_v \text{ EXCEPT } ![mon] = 0]$
 $\wedge uncommitted_value' = [uncommitted_value \text{ EXCEPT } ![mon] = Nil]$
 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 adds events to send lease messages to peers.

Variables changed: *acked_lease*, *messages*, *message_history*, *phase*.
 $extend_lease(mon) \triangleq$
 $\wedge isLeader[mon] = \text{TRUE}$
 $\wedge acked_lease' = [acked_lease \text{ EXCEPT } ![mon] =$
 $[m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]]$
 $\wedge Send_set(mon,$
 $\{[type \mapsto OP_LEASE,$
 $from \mapsto mon,$
 $dest \mapsto dest,$
 $last_committed \mapsto last_committed[mon]] : dest \in (Monitors \setminus \{mon\})$
 $\})$
 $\wedge phase' = [phase \text{ EXCEPT } ![mon] = PHASE_LEASE]$

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*.

$handle_lease(mon, msg) \triangleq$
 \wedge discard if not peon or peon is behind
 IF $\vee isLeader[mon] = \text{TRUE}$
 $\vee last_committed[mon] \neq msg.last_committed$
 THEN $\wedge Discard(msg)$
 \wedge UNCHANGED *state*

ELSE $\wedge state' = [state \text{ EXCEPT } ![mon] = STATE_ACTIVE]$

$\wedge Reply([type \mapsto OP_LEASE_ACK,$
 $from \mapsto mon,$
 $dest \mapsto msg.from,$
 $first_committed \mapsto first_committed[mon],$
 $last_committed \mapsto last_committed[mon]], msg)$

$\wedge Discard(msg)$

$\wedge UNCHANGED \langle epoch, isLeader, phase \rangle$

$\wedge UNCHANGED \langle restart_vars, data_vars, collect_vars, lease_vars, commit_vars \rangle$

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

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*.

$handle_lease_ack(mon, msg) \triangleq$

$\wedge phase[mon] = PHASE_LEASE$

$\wedge acked_lease' = [acked_lease \text{ EXCEPT } ![mon] =$
 $[acked_lease[mon] \text{ EXCEPT } ![msg.from] = TRUE]]$

$\wedge Discard(msg)$

$\wedge UNCHANGED \langle epoch \rangle$

$\wedge UNCHANGED \langle state_vars, restart_vars, data_vars, collect_vars, commit_vars \rangle$

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

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*.

$post_lease_ack(mon) \triangleq$

$\wedge phase[mon] = PHASE_LEASE$

$\wedge phase' = [phase \text{ EXCEPT } ![mon] = PHASE_LEASE_DONE]$

$\wedge \forall m \in Monitors : acked_lease[mon][m] = TRUE$

$\wedge UNCHANGED \langle isLeader, state \rangle$

$\wedge UNCHANGED \langle global_vars, restart_vars, data_vars, collect_vars,$
 $lease_vars, commit_vars \rangle$

Commit phase predicates

Start a commit phase by the leader. The variable *new_value* is assigned 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*.

$begin(mon, v) \triangleq$

$\wedge isLeader[mon] = TRUE$

$\wedge \vee state'[mon] = STATE_UPDATING$

$\vee state'[mon] = STATE_UPDATING_PREVIOUS$

$$\begin{aligned}
& \wedge \text{MonitorsLen} = 1 \vee \text{num_last}[\text{mon}] > \text{MonitorsLen} \div 2 \\
& \wedge \text{new_value}[\text{mon}] = \text{Nil} \\
& \wedge \text{accepted}' = [\text{accepted} \text{ EXCEPT } ![\text{mon}] = \\
& \quad [m \in \text{Monitors} \mapsto \text{IF } m = \text{mon} \text{ THEN TRUE ELSE FALSE}]] \\
& \wedge \text{new_value}' = [\text{new_value} \text{ EXCEPT } ![\text{mon}] = v] \\
& \wedge \text{phase}' = [\text{phase} \text{ EXCEPT } ![\text{mon}] = \text{PHASE_BEGIN}] \\
& \wedge \text{values}' = [\text{values} \text{ EXCEPT } ![\text{mon}] = \\
& \quad (\text{values}[\text{mon}] @@ ((\text{last_committed}[\text{mon}] + 1) :> \text{new_value}'[\text{mon}]))] \\
& \wedge \text{Send_set}(\text{mon}, \\
& \quad \{[\text{type} \quad \mapsto \text{OP_BEGIN}, \\
& \quad \text{from} \quad \mapsto \text{mon}, \\
& \quad \text{dest} \quad \mapsto \text{dest}, \\
& \quad \text{last_committed} \mapsto \text{last_committed}[\text{mon}], \\
& \quad \text{values} \quad \mapsto \text{values}'[\text{mon}], \\
& \quad \text{pn} \quad \mapsto \text{accepted_pn}[\text{mon}]] : \text{dest} \in (\text{Monitors} \setminus \{\text{mon}\}) \\
& \quad \}) \\
& \wedge \text{uncommitted_v}' = [\text{uncommitted_v} \text{ EXCEPT } ![\text{mon}] = \text{last_committed}[\text{mon}] + 1] \\
& \wedge \text{uncommitted_value}' = [\text{uncommitted_value} \text{ EXCEPT } ![\text{mon}] = v]
\end{aligned}$$

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*.

$$\begin{aligned}
& \text{handle_begin}(\text{mon}, \text{msg}) \triangleq \\
& \quad \wedge \text{isLeader}[\text{mon}] = \text{FALSE} \\
& \quad \wedge \text{IF } \text{msg.pn} < \text{accepted_pn}[\text{mon}] \\
& \quad \quad \text{THEN} \\
& \quad \quad \quad \wedge \text{Discard}(\text{msg}) \\
& \quad \quad \quad \wedge \text{UNCHANGED } \langle \text{state}, \text{restart_vars} \rangle \\
& \quad \quad \text{ELSE} \\
& \quad \quad \quad \wedge \text{msg.pn} = \text{accepted_pn}[\text{mon}] \\
& \quad \quad \quad \wedge \text{msg.last_committed} = \text{last_committed}[\text{mon}] \\
& \quad \quad \quad \text{assign } \text{values}[\text{mon}][\text{last_committed}[\text{mon}] + 1] \\
& \quad \quad \quad \wedge \text{values}' = [\text{values} \text{ EXCEPT } ![\text{mon}] = \\
& \quad \quad \quad \quad (\text{values}[\text{mon}] @@ ((\text{last_committed}[\text{mon}] + 1) :> \text{msg.values}[\text{last_committed}[\text{mon}] + 1]))] \\
& \quad \quad \quad \wedge \text{state}' = [\text{state} \text{ EXCEPT } ![\text{mon}] = \text{STATE_UPDATING}] \\
& \quad \quad \quad \wedge \text{uncommitted_v}' = [\text{uncommitted_v} \text{ EXCEPT } ![\text{mon}] = \text{last_committed}[\text{mon}] + 1] \\
& \quad \quad \quad \wedge \text{uncommitted_value}' = [\text{uncommitted_value} \text{ EXCEPT } ![\text{mon}] = \\
& \quad \quad \quad \quad \text{values}'[\text{mon}][\text{last_committed}[\text{mon}] + 1]] \\
& \quad \quad \quad \wedge \text{Reply}([\text{type} \quad \mapsto \text{OP_ACCEPT}, \\
& \quad \quad \quad \text{from} \quad \mapsto \text{mon},
\end{aligned}$$

$$\begin{aligned}
dest &\mapsto msg.from, \\
last_committed &\mapsto last_committed[mon], \\
pn &\mapsto accepted_pn[mon]], msg)
\end{aligned}$$

$$\begin{aligned}
&\wedge \text{UNCHANGED } \langle epoch, isLeader, phase, monitor_store, accepted_pn, first_committed, \\
&\quad last_committed \rangle \\
&\wedge \text{UNCHANGED } \langle collect_vars, lease_vars, commit_vars \rangle
\end{aligned}$$

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`

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

Predicate that is enabled and called when all peers 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`

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

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

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*.

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

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*.

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

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*.

$client_request(mon, v) \triangleq$
 $\wedge phase[mon] = PHASE_LEASE \vee phase[mon] = PHASE_ELECTION$
 $\wedge isLeader[mon] = TRUE$
 $\wedge state[mon] = STATE_ACTIVE$
 $\wedge pending_proposal[mon] = Nil$
 $\wedge pending_proposal' = [pending_proposal \text{ EXCEPT } ![mon] = v]$
 $\wedge UNCHANGED \langle new_value, accepted \rangle$
 $\wedge UNCHANGED \langle global_vars, state_vars, restart_vars, data_vars, collect_vars, lease_vars \rangle$

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*.

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

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*.

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

Continue the start of the collect phase. Initialize the number of peers that accepted the proposal (*num_last*) and the variables with peers version numbers. Check if there is an uncommitted value.

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*.

$pre_send_collect(mon) \triangleq$
 $\wedge state[mon] = STATE_RECOVERING$
 $\wedge isLeader[mon] = TRUE$
 $\wedge phase[mon] = PHASE_SEND_COLLECT$
 $\wedge clear_peer_first_committed(mon)$
 $\wedge clear_peer_last_committed(mon)$

```

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

 $\wedge num\_last' = [num\_last \text{ EXCEPT } ![mon] = 1]$ 

 $\wedge Send\_set(mon,$ 
   $\{[type \mapsto OP\_COLLECT,$ 
     $from \mapsto mon,$ 
     $dest \mapsto dest,$ 
     $first\_committed \mapsto first\_committed[mon],$ 
     $last\_committed \mapsto last\_committed[mon],$ 
     $pn \mapsto accepted\_pn[mon]] : dest \in (Monitors \setminus \{mon\})$ 
   $\})$ 

 $\wedge phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_COLLECT]$ 
 $\wedge$  UNCHANGED  $\langle isLeader, state \rangle$ 
 $\wedge$  UNCHANGED  $\langle epoch, data\_vars, lease\_vars, commit\_vars \rangle$ 

```

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*

```

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

```

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 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)  $\triangleq$ 
   $\wedge$  isLeader[mon] = TRUE

   $\wedge$  peer_first_committed' = [peer_first_committed EXCEPT ![mon] =
    [peer_first_committed[mon] EXCEPT ![msg.from] = msg.first_committed]]
   $\wedge$  peer_last_committed' = [peer_last_committed EXCEPT ![mon] =
    [peer_last_committed[mon] EXCEPT ![msg.from] = msg.last_committed]]

   $\wedge$  IF msg.first_committed > last_committed[mon] + 1
    THEN
       $\wedge$  bootstrap
       $\wedge$  Discard(msg)
       $\wedge$  UNCHANGED  $\langle$ num_last, accepted_pn, values, phase, monitor_store $\rangle$ 
       $\wedge$  UNCHANGED  $\langle$ first_committed, last_committed, restart_vars $\rangle$ 
    ELSE
       $\wedge$  store_state(mon, msg)
       $\wedge$  IF  $\exists$  peer  $\in$  Monitors :
         $\wedge$  peer  $\neq$  mon
         $\wedge$  peer_last_committed'[mon][peer]  $\neq$  -1
         $\wedge$  peer_last_committed'[mon][peer] + 1 < first_committed[mon]
         $\wedge$  first_committed[mon] > 1
      THEN
         $\wedge$  bootstrap
         $\wedge$  check_and_correct_uncommitted(mon)
         $\wedge$  Discard(msg)
         $\wedge$  UNCHANGED  $\langle$ phase, accepted_pn, num_last $\rangle$ 
      ELSE
         $\wedge$  LET monitors_behind  $\triangleq$  {peer  $\in$  Monitors :
           $\wedge$  peer  $\neq$  mon
           $\wedge$  peer_last_committed'[mon][peer]  $\neq$  -1
           $\wedge$  peer_last_committed'[mon][peer] < last_committed[mon]}
        IN Reply_set(mon,
          {[type  $\mapsto$  OP_COMMIT,
            from  $\mapsto$  mon,
            dest  $\mapsto$  dest,
            last_committed  $\mapsto$  last_committed'[mon],
            pn  $\mapsto$  accepted_pn[mon],
            values  $\mapsto$  values[mon]] : dest  $\in$  monitors_behind
          }, msg)

```

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

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*.

$$\begin{aligned}
post_last(mon) & \triangleq \\
& \wedge isLeader[mon] = \text{TRUE} \\
& \wedge num_last[mon] = MonitorsLen \\
& \wedge phase[mon] = PHASE_COLLECT \\
& \wedge clear_peer_first_committed(mon) \\
& \wedge clear_peer_last_committed(mon) \\
& \wedge \text{IF } \wedge uncommitted_v[mon] = last_committed[mon] + 1 \\
& \quad \wedge uncommitted_value[mon] \neq Nil \\
& \quad \text{THEN } \wedge state' = [state \text{ EXCEPT } ![mon] = STATE_UPDATING_PREVIOUS] \\
& \quad \quad \wedge begin(mon, uncommitted_value) \\
& \quad \quad \wedge UNCHANGED \langle acked_lease \rangle \\
& \quad \text{ELSE } \wedge finish_round(mon) \\
& \quad \quad \wedge extend_lease(mon)
\end{aligned}$$

$\wedge \text{UNCHANGED } \langle \text{accepted}, \text{new_value}, \text{values}, \text{restart_vars} \rangle$
 $\wedge \text{UNCHANGED } \langle \text{isLeader}, \text{monitor_store}, \text{accepted_pn}, \text{first_committed}, \text{last_committed} \rangle$
 $\wedge \text{UNCHANGED } \langle \text{epoch}, \text{num_last}, \text{pending_proposal} \rangle$

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*.

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

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

Variables changed: *accepted_pn*, *phase*.

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

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*.

$\text{restart_mon}(\text{mon}) \triangleq$
 $\wedge \text{messages}' = [\text{messages} \text{ EXCEPT } ![\text{mon}] = [\text{mon1} \in \text{Monitors} \mapsto \langle \rangle]]$
 $\wedge \text{isLeader}' = [\text{isLeader} \text{ EXCEPT } ![\text{mon}] = \text{FALSE}]$
 $\wedge \text{phase}' = [\text{phase} \text{ EXCEPT } ![\text{mon}] = \text{PHASE_ELECTION}]$
 $\wedge \text{state}' = [\text{state} \text{ EXCEPT } ![\text{mon}] = \text{IF } \text{MonitorsLen} = 1$
 $\quad \text{THEN } \text{STATE_ACTIVE}$
 $\quad \text{ELSE } \text{STATE_RECOVERING}]$
 $\wedge \text{pending_proposal}' = [\text{pending_proposal} \text{ EXCEPT } ![\text{mon}] = \text{Nil}]$
 $\wedge \text{new_value}' = [\text{new_value} \text{ EXCEPT } ![\text{mon}] = \text{Nil}]$
 $\wedge \text{number_refreshes}' = \text{number_refreshes} + 1$
 $\wedge \text{UNCHANGED } \langle \text{epoch}, \text{message_history}, \text{accepted} \rangle$

$\wedge \text{UNCHANGED } \langle \text{restart_vars}, \text{data_vars}, \text{collect_vars}, \text{lease_vars} \rangle$

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.

$\text{Timeout}(\text{mon}) \triangleq$

- $\wedge \text{phase}[\text{mon}] = \text{PHASE_COLLECT} \vee \text{phase}[\text{mon}] = \text{PHASE_BEGIN}$
- $\wedge \text{bootstrap}$
- $\wedge \text{messages}' = [\text{mon1} \in \text{Monitors} \mapsto [\text{mon2} \in \text{Monitors} \mapsto \langle \rangle]]$
- $\wedge \text{UNCHANGED } \langle \text{message_history}, \text{state_vars}, \text{restart_vars}, \text{data_vars}, \text{collect_vars}, \text{lease_vars}, \text{commit_vars} \rangle$

Dispatchers and next statement

Handle a message.

$\text{Receive}(\text{msg}) \triangleq$

- $\wedge \vee \text{phase}[\text{msg.dest}] = \text{PHASE_COLLECT}$
- $\vee \text{phase}[\text{msg.dest}] = \text{PHASE_BEGIN}$
- $\vee \text{phase}[\text{msg.dest}] = \text{PHASE_ELECTION}$
- \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.
 $reduce_search_space \triangleq$
 $\wedge epoch \neq 6 \wedge number_refreshes \neq 3$
 $\wedge \forall mon \in Monitors : last_committed[mon] < 2$
 $\Rightarrow \forall mon2 \in Monitors : new_value[mon2] = Nil$
 $\wedge \forall mon \in Monitors : accepted_pn[mon] < 300$

State transitions.
 $Next \triangleq$
 $\wedge reduce_search_space$
 $\wedge \text{IF } epoch \% 2 = 1 \text{ THEN}$
 $\quad \wedge leader_election$
 $\quad \wedge step_x' = \text{"election"} \wedge step' = step + 1$
 $\quad \wedge \text{UNCHANGED } number_refreshes$
 ELSE
 $\quad \vee$
 $\quad \quad \wedge \exists mon \in Monitors : election_recover(mon)$
 $\quad \quad \wedge step_x' = \text{"election_recover"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : pre_send_collect(mon)$
 $\quad \quad \wedge step_x' = \text{"pre_send_collect"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : post_last(mon)$
 $\quad \quad \wedge step_x' = \text{"post_last"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : post_lease_ack(mon)$
 $\quad \quad \wedge step_x' = \text{"post_lease_ack"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : post_accept(mon)$
 $\quad \quad \wedge step_x' = \text{"post_accept"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : finish_commit(mon)$
 $\quad \quad \wedge step_x' = \text{"finish_commit"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : \exists v \in Value_set : client_request(mon, v)$
 $\quad \quad \wedge step_x' = \text{"client_request"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$
 $\quad \vee \wedge \exists mon \in Monitors : propose_pending(mon)$
 $\quad \quad \wedge step_x' = \text{"propose_pending"} \wedge step' = step + 1$
 $\quad \quad \wedge \text{UNCHANGED } number_refreshes$

$$\begin{aligned}
& \vee \wedge \exists \text{mon1}, \text{mon2} \in \text{Monitors} : \\
& \quad \wedge \text{mon1} \neq \text{mon2} \\
& \quad \wedge \text{Len}(\text{messages}[\text{mon1}][\text{mon2}]) > 0 \\
& \quad \wedge \text{Receive}(\text{messages}[\text{mon1}][\text{mon2}][1]) \\
& \quad \wedge \text{step}' = \text{step} + 1 \\
& \quad \wedge \text{UNCHANGED } \text{number_refreshes} \\
& \vee \wedge \exists \text{mon} \in \text{Monitors} : \text{restart_mon}(\text{mon}) \\
& \quad \wedge \text{step_x}' = \text{"restart mon"} \wedge \text{step}' = \text{step} + 1 \\
& \vee \wedge \exists \text{mon} \in \text{Monitors} : \text{Timeout}(\text{mon}) \\
& \quad \wedge \text{step_x}' = \text{"timeout and restart"} \wedge \text{step}' = \text{step} + 1 \\
& \quad \wedge \text{UNCHANGED } \text{number_refreshes}
\end{aligned}$$

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

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

$\wedge \text{Inv_diam}(20)$

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.

$\text{Inv_find_state}(\$
 $\quad \exists \text{msg1}, \text{msg2} \in \text{message_history} :$
 $\quad \wedge \text{msg1.type} = \text{OP_COLLECT} \wedge \text{msg2.type} = \text{OP_COLLECT}$
 $\quad \wedge \text{msg1.from} \neq \text{msg2.from}$
 $\quad \left. \right)$

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

$\text{Inv_find_state}(\$
 $\quad \wedge \text{step_x} = \text{"restart mon"}$
 $\quad \backslash * \text{The system is in collect phase and no OP_LAST message has been received.}$
 $\quad \backslash * \text{isLeader}[\text{mon}] = \text{TRUE assures that the leader was not the one that crashed.}$
 $\quad \wedge \exists \text{mon} \in \text{Monitors} :$
 $\quad \quad \wedge \text{isLeader}[\text{mon}] = \text{TRUE}$
 $\quad \quad \wedge \text{phase}[\text{mon}] = \text{PHASE_COLLECT}$
 $\quad \quad \wedge \text{num_last}[\text{mon}] = 1$
 $\quad \left. \right)$

```

\ * All the collect requests have been handled by the peers.
 $\wedge \forall mon1, mon2 \in Monitors :$ 
   $\forall i \in 1 \dots Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type \neq OP\_COLLECT$ 
 $\wedge epoch = 2$ 
)

Find a state where the leader crashes during the commit phase, failing to complete the commit.
Inv_find_state(
   $\wedge step\_x = \text{"restart mon"}$ 
   $\wedge \exists mon1, mon2 \in Monitors :$ 
     $\exists i \in 1 \dots Len(messages[mon1][mon2]) : messages[mon1][mon2][i].type = OP\_ACCEPT$ 
   $\wedge \forall mon \in Monitors :$ 
     $isLeader[mon] = FALSE$ 
   $\wedge epoch = 2$ 
)

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

```

```

\ * Modification History
\ * Last modified Sun Mar 14 16:10:33 WET 2021 by afonsonf
\ * Created Mon Jan 11 16:15:26 WET 2021 by afonsonf

```