MODULE *paxos*

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

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

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

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

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

Constants

Set of monitors.

CONSTANTS *Monitors*

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

$ranks \triangleq SetToSeq(Monitors)$

$rank(mon) \triangleq \text{CHOOSE } i \in 1 \,..\, Len(ranks) : ranks[i] = mon$

Set of possible values.

CONSTANTS *Value_set*

Reserved value.

CONSTANTS *Nil*

Paxos states:

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

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

Paxos auxiliary phase states:

They are used to force some sequence of steps.

CONSTANTS $PHASE\_ELECTION$,
$PHASE\_PRE\_COLLECT$, $PHASE\_COLLECT$,
$PHASE\_LEASE$, $PHASE\_LEASE\_DONE$,
$PHASE\_BEGIN$, $PHASE\_BEGIN\_DONE$,
$PHASE\_COMMIT$, $PHASE\_COMMIT\_DONE$

$phase\_names \triangleq \{PHASE\_ELECTION$,
$PHASE\_PRE\_COLLECT$, $PHASE\_COLLECT$,
$PHASE\_LEASE$, $PHASE\_LEASE\_DONE$,
$PHASE\_BEGIN$, $PHASE\_BEGIN\_DONE$,
$PHASE\_COMMIT$, $PHASE\_COMMIT\_DONE\}$

Paxos message types:
CONSTANTS $OP\_COLLECT$, OP_LAST,
OP_BEGIN, OP_ACCEPT, OP_COMMIT,
OP_LEASE, OP_LEASE_ACK

$messages\_types \triangleq \{OP\_COLLECT$, 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: [message 'm' $\mapsto$ 1 if 'm' is in the network else 0]
VARIABLE $messages$

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 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* $\mapsto$ [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 for each monitor.
Type: [*Monitors* $\mapsto$ value version]
VARIABLE *first_committed*

A function that stores the last value version committed for each monitor.
Type: [*Monitors* $\mapsto$ value version]
VARIABLE *last_committed*

Collect phase variables

A function that stores the number of peers that accepted a collect request.
Type: [*Monitors* $\mapsto$ number of peers that accepted]
VARIABLE *num_last*

Used by leader when receiving responses in collect phase.
Type: [*Monitors* $\mapsto$ [*Monitors* $\mapsto$ value version]]
VARIABLE *peer_first_committed*

Used by leader when receiving responses in collect phase.
Type: [*Monitors* $\mapsto$ [*Monitors* $\mapsto$ value version]]
VARIABLE *peer_last_committed*

Lease phase variables

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

Commit phase variables

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

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

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

Auxiliary variables

A function that stores, for each monitor, a queue of messages types to send.
Type: $[Monitors \mapsto \langle Monitors \times messages\_types \rangle]$
VARIABLE *send_queue*

Debug variables

Variables to help debug a behavior.
step is the diameter of a behavior/path.
step_x the current predicate being called.
VARIABLE *step*, *step_x*

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

Variables initialization

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

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

$Init\_global\_vars \triangleq$
$\quad \land\ epoch = 1$
$\quad \land\ messages = [m \in \{\} \mapsto 0]$

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

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

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

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

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

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

$Init\_auxiliary\_vars \triangleq$
$\quad \land\ send\_queue = [mon \in Monitors \mapsto \langle\rangle]$

$Init \triangleq$
$\quad \land\ Init\_global\_vars$
$\quad \land\ Init\_state\_vars$
$\quad \land\ Init\_restart\_vars$
$\quad \land\ Init\_data\_vars$
$\quad \land\ Init\_collect\_vars$
$\quad \land\ Init\_lease\_vars$
$\quad \land\ Init\_auxiliary\_vars$
$\quad \land\ Init\_commit\_vars$
$\quad \land\ step = 0 \land step\_x = \text{“init”} \land number\_refreshes = 0$

Message manipulation

Add message m to the network $msgs$.
$WithMessage(m,\ msgs) \triangleq$
$\quad (m :> 1) \mathbin{@@} msgs$

Remove message m from the network $msgs$.
$WithoutMessage(m,\ msgs) \triangleq$
$\quad (m :> 0) \mathbin{@@} msgs$

Set of messages in network.
$ValidMessage(msgs) \triangleq$
$\quad \{m \in \text{DOMAIN}\ messages : msgs[m] = 1\}$

Adds the message m to the network.
Variables changed: messages.
$Send(m) \triangleq$
$\quad messages' = WithMessage(m,\ messages)$

Removes message m from the network.
Variables changed: messages.
$Discard(m) \triangleq$
$\quad messages' = WithoutMessage(m,\ messages)$

Removes the request from network and adds the response.
Variables changed: messages.
$Reply(response,\ request) \triangleq$
$\quad messages' = WithoutMessage(request,\ WithMessage(response,\ messages))$

Helper predicates

Compute a new unique proposal number for a given monitor.
Example: $oldpn = 305$, $rank(mon) = 5$, $newpn = 405$.
$get\_new\_proposal\_number(mon,\ oldpn) \triangleq$
$\quad ((oldpn \div 100) + 1) * 100 + rank(mon)$

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

Clear the variable $peer\_last\_committed$.
Variables changed: $peer\_last\_committed$.
$clear\_peer\_last\_committed(mon) \triangleq$

$peer\_last\_committed' = [peer\_last\_committed \text{ EXCEPT } ![mon] = [m \in Monitors \mapsto -1]]$

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

$store\_state(mon, msg) \triangleq$

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

Update last committed and first committed.

$\land\ last\_committed' = [last\_committed \text{ EXCEPT } ![mon] = Max(logs \cup \{last\_committed[mon]\})]$
$\land$ IF $logs \neq \{\} \land first\_committed[mon] = 0$
THEN $first\_committed' = [first\_committed \text{ EXCEPT } ![mon] = Min(logs)]$
ELSE $first\_committed' = [first\_committed \text{ EXCEPT } ![mon] = Min(logs \cup \{first\_committed[mon]\})]$

Update monitor store.

$\land$ IF $last\_committed'[mon] = 0$
THEN UNCHANGED $monitor\_store$
ELSE $monitor\_store' = [monitor\_store \text{ EXCEPT } ![mon] = values'[mon][last\_committed'[mon]]]$

Check if uncommitted value version is still valid, else reset it.
Variables changed: *uncommitted_v*, *uncommitted_value*.

$check\_and\_correct\_uncommitted(mon) \triangleq$

IF $uncommitted\_v[mon] \leq last\_committed'[mon]$
THEN $\land\ uncommitted\_v' = [uncommitted\_v \text{ EXCEPT } ![mon] = 0]$
$\land\ uncommitted\_value' = [uncommitted\_value \text{ EXCEPT } ![mon] = Nil]$
ELSE UNCHANGED $\langle uncommitted\_v, uncommitted\_value \rangle$

Trigger new election by incrementing epoch.
Variables changed: epoch.

$bootstrap \triangleq$

$\land\ epoch' = epoch + 1$

Lease phase predicates

Changes *mon* state to *STATE_ACTIVE*.
Variables changed: state.

$finish\_round(mon) \triangleq$

$\land\ isLeader[mon] = \text{TRUE}$
$\land\ state' = [state \text{ EXCEPT } ![mon] = STATE\_ACTIVE]$

Resets the variable acked lease and adds events to send lease messages to peers.
Variables changed: *acked_lease*, *send_queue*, phase.

$$
\begin{array}{l}
extend\_lease(mon) \triangleq \\
\quad \land\ isLeader[mon] = \text{TRUE} \\
\quad \land\ acked\_lease' = [acked\_lease \text{ EXCEPT } ![mon] = \\
\qquad [m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]] \\
\quad \land\ send\_queue' = [send\_queue \text{ EXCEPT } ![mon] = \\
\qquad send\_queue[mon] \circ SetToSeq((Monitors \setminus \{mon\}) \times \{OP\_LEASE\})] \\
\quad \land\ phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_LEASE]
\end{array}
$$

Send a lease message from the leader to a peer.
Variables changed: messages.

$$
\begin{array}{l}
send\_extend\_lease(mon,\ dest) \triangleq \\
\quad \land\ isLeader[mon] = \text{TRUE} \\
\quad \land\ phase[mon] = PHASE\_LEASE \\
\quad \land\ Send([type \mapsto OP\_LEASE, \\
\qquad\qquad from \mapsto mon, \\
\qquad\qquad dest \mapsto dest, \\
\qquad\qquad last\_committed \mapsto last\_committed[mon]]) \\
\quad \land\ \text{UNCHANGED } \langle epoch \rangle \\
\quad \land\ \text{UNCHANGED } \langle restart\_vars,\ data\_vars,\ state\_vars,\ collect\_vars,\ lease\_vars,\ commit\_vars \rangle
\end{array}
$$

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, state.

$$
\begin{array}{l}
handle\_lease(mon,\ msg) \triangleq \\
\quad \land\ \text{discard if not peon or peon is behind} \\
\qquad \text{IF } \lor\ isLeader[mon] = \text{TRUE} \\
\qquad\quad \lor\ last\_committed[mon] \neq msg.last\_committed \\
\qquad \text{THEN } \land\ Discard(msg) \\
\qquad\qquad\ \land\ \text{UNCHANGED } state \\
\qquad \text{ELSE } \land\ state' = [state \text{ EXCEPT } ![mon] = STATE\_ACTIVE] \\
\qquad\quad \land\ Reply([type \mapsto OP\_LEASE\_ACK, \\
\qquad\qquad from \mapsto mon, \\
\qquad\qquad dest \mapsto msg.from, \\
\qquad\qquad first\_committed \mapsto first\_committed[mon], \\
\qquad\qquad last\_committed \mapsto last\_committed[mon]],\ msg) \\
\qquad\quad \land\ Discard(msg) \\
\quad \land\ \text{UNCHANGED } \langle epoch,\ isLeader,\ phase \rangle \\
\quad \land\ \text{UNCHANGED } \langle restart\_vars,\ data\_vars,\ collect\_vars,\ lease\_vars,\ commit\_vars,\ auxiliary\_vars \rangle
\end{array}
$$

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.

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

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.

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

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, *send_queue*, values, *uncommitted_v*, *uncommitted_value*.

$begin(mon,\ v) \triangleq$
$\land\ isLeader[mon] = \text{TRUE}$
$\land\ \lor\ state'[mon] = STATE\_UPDATING$
$\lor\ state'[mon] = STATE\_UPDATING\_PREVIOUS$
$\land\ Len(ranks) = 1 \lor num\_last[mon] > Len(ranks) \div 2$
$\land\ new\_value[mon] = Nil$
$\land\ accepted' = [accepted \text{ EXCEPT } ![mon] =$
$[m \in Monitors \mapsto \text{IF } m = mon \text{ THEN TRUE ELSE FALSE}]]$
$\land\ new\_value' = [new\_value \text{ EXCEPT } ![mon] = v]$
$\land\ phase' = [phase \text{ EXCEPT } ![mon] = PHASE\_BEGIN]$
$\land\ send\_queue' = [send\_queue \text{ EXCEPT } ![mon] =$
$send\_queue[mon] \circ SetToSeq((Monitors \setminus \{mon\}) \times \{OP\_BEGIN\})]$
$\land\ values' = [values \text{ EXCEPT } ![mon] =$
$(values[mon] \mathbin{@@} ((last\_committed[mon] + 1) :> new\_value'[mon]))]$
$\land\ uncommitted\_v' = [uncommitted\_v \text{ EXCEPT } ![mon] = last\_committed[mon] + 1]$
$\land\ uncommitted\_value' = [uncommitted\_value \text{ EXCEPT } ![mon] = v]$

Sends a begin message from the leader to the peer.
Variables changed: messages.

$$
\begin{array}{l}
send\_begin(mon,\ dest) \triangleq \\
\quad \land\ isLeader[mon] = \text{TRUE} \\
\quad \land\ phase[mon] = PHASE\_BEGIN \\
\quad \land\ Send([type \mapsto OP\_BEGIN, \\
\qquad\qquad from \mapsto mon, \\
\qquad\qquad dest \mapsto dest, \\
\qquad\qquad last\_committed \mapsto last\_committed[mon], \\
\qquad\qquad values \mapsto values[mon], \\
\qquad\qquad pn \mapsto accepted\_pn[mon]]) \\
\quad \land\ \text{UNCHANGED}\ \langle epoch \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle state\_vars,\ restart\_vars,\ data\_vars,\ collect\_vars,\ lease\_vars,\ commit\_vars \rangle
\end{array}
$$

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, state, values, *uncommitted_v*, *uncommitted_value*.

$$
\begin{array}{l}
handle\_begin(mon,\ msg) \triangleq \\
\quad \land\ isLeader[mon] = \text{FALSE} \\
\quad \land\ \text{IF}\ msg.pn < accepted\_pn[mon] \\
\qquad \text{THEN} \\
\qquad\quad \land\ Discard(msg) \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle state,\ restart\_vars \rangle \\
\qquad \text{ELSE} \\
\qquad\quad \land\ msg.pn = accepted\_pn[mon] \\
\qquad\quad \land\ msg.last\_committed = last\_committed[mon] \\
\qquad\quad \text{assign}\ values[mon][last\_committed[mon] + 1] \\
\qquad\quad \land\ values' = [values\ \text{EXCEPT}\ ![mon] = \\
\qquad\qquad\quad (values[mon]\ @@\ ((last\_committed[mon] + 1) :> msg.values[last\_committed[mon] + 1]))] \\
\qquad\quad \land\ state' = [state\ \text{EXCEPT}\ ![mon] = STATE\_UPDATING] \\
\qquad\quad \land\ uncommitted\_v' = [uncommitted\_v\ \text{EXCEPT}\ ![mon] = last\_committed[mon] + 1] \\
\qquad\quad \land\ uncommitted\_value' = [uncommitted\_value\ \text{EXCEPT}\ ![mon] = \\
\qquad\qquad\quad values'[mon][last\_committed[mon] + 1]] \\
\qquad\quad \land\ Reply([type \mapsto OP\_ACCEPT, \\
\qquad\qquad\qquad\qquad from \mapsto mon, \\
\qquad\qquad\qquad\qquad dest \mapsto msg.from, \\
\qquad\qquad\qquad\qquad last\_committed \mapsto last\_committed[mon], \\
\qquad\qquad\qquad\qquad pn \mapsto accepted\_pn[mon]],\ msg) \\
\quad \land\ \text{UNCHANGED}\ \langle epoch,\ isLeader,\ phase,\ monitor\_store,\ accepted\_pn,\ first\_committed, \\
\qquad\qquad\qquad last\_committed \rangle
\end{array}
$$

$\land$ UNCHANGED $\langle collect\_vars,\ lease\_vars,\ commit\_vars,\ auxiliary\_vars\rangle$

Handle an accept message. If the leader receives a positive response from the peer, it will add it to the variable accepted.
Variables changed: messages, accepted

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

Predicate that is enabled and called when all peers accept begin request from leader.
The leader commits the transaction in *new_value* and adds events in *send_queue* to send commit messages to his peers.
Variables changed: *first_committed*, *last_committed*, *monitor_store*, *new_value*, *send_queue*, state, phase

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

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

$\land\ monitor\_store' = [monitor\_store$ EXCEPT $![mon] = values[mon][last\_committed[mon] + 1]]$
$\land\ new\_value' = [new\_value$ EXCEPT $![mon] = Nil]$
$\land\ send\_queue' = [send\_queue$ EXCEPT $![mon] =$
$send\_queue[mon] \circ SetToSeq((Monitors \setminus \{mon\}) \times \{OP\_COMMIT\})]$
$\land\ state' = [state$ EXCEPT $![mon] = STATE\_REFRESH]$
$\land\ phase' = [phase$ EXCEPT $![mon] = PHASE\_COMMIT]$

$\land$ UNCHANGED $\langle isLeader, values, accepted\_pn, pending\_proposal, accepted\rangle$
$\land$ UNCHANGED $\langle global\_vars, restart\_vars, collect\_vars, lease\_vars\rangle$

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*, *send_queue*.

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

Send a commit message from the leader to a peer.
Variables changed: messages.

$send\_commit(mon, dest) \triangleq$
$\land\ isLeader[mon] = \text{TRUE}$
$\land\ Send([type \mapsto OP\_COMMIT,$
$\quad from \mapsto mon,$
$\quad dest \mapsto dest,$
$\quad last\_committed \mapsto last\_committed[mon],$
$\quad pn \mapsto accepted\_pn[mon],$
$\quad values \mapsto values[mon]])$
$\land$ UNCHANGED $\langle epoch\rangle$
$\land$ UNCHANGED $\langle state\_vars, restart\_vars, data\_vars, collect\_vars, lease\_vars, commit\_vars\rangle$

Handle a commit message. The monitor stores the values sent by the leader commit message.
Variables changed: messages, values, *first_committed*, *last_committed*, *monitor_store*, *uncommitted_v*, *uncommitted_value*.

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

Client Request

Request a transaction v to the monitor. The transaction is saved on pending proposal to be committed in the next available commit phase.
This predicate has a big cost on performance, so there were some requirements added (monitor phase and state) to mitigate that.
Variables changed: *pending_proposal*.

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

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

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

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.

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

Continue the start of the collect phase. Initialize the number of peers that accepted the proposal (*num_last*) and the variables with peers version numbers. Check if there is an uncommitted value.
Add events to *send_queue* to send collect messages to the peers.
Variables changed: *peer_first_committed*, *peer_last_committed*, *uncommitted_v*, *uncommitted_value*, *num_last*, *send_queue*, phase.

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

```
    ∧ IF last_committed[mon] + 1 ∈ DOMAIN values[mon]
       THEN ∧ uncommitted_v' =
                  [uncommitted_v EXCEPT ![mon] = last_committed[mon] + 1]
            ∧ uncommitted_value' =
                  [uncommitted_value EXCEPT ![mon] = values[mon][last_committed[mon] + 1]]
       ELSE UNCHANGED ⟨restart_vars⟩

    ∧ num_last' = [num_last EXCEPT ![mon] = 1]
    ∧ send_queue' = [send_queue EXCEPT ![mon] =
       send_queue[mon] ∘ SetToSeq((Monitors \ {mon}) × {OP_COLLECT})]
    ∧ phase' = [phase EXCEPT ![mon] = PHASE_COLLECT]
    ∧ UNCHANGED ⟨isLeader, state⟩
    ∧ UNCHANGED ⟨global_vars, data_vars, lease_vars, commit_vars⟩
```

Send a collect message from the leader to a peer.
Variables changed: messages.

```
send_collect(mon, dest) ≜
    ∧ state[mon] = STATE_RECOVERING
    ∧ isLeader[mon] = TRUE
    ∧ phase[mon] = PHASE_COLLECT
    ∧ Send([type           ↦ OP_COLLECT,
            from           ↦ mon,
            dest           ↦ dest,
            first_committed ↦ first_committed[mon],
            last_committed ↦ last_committed[mon],
            pn             ↦ accepted_pn[mon]])
    ∧ UNCHANGED epoch
    ∧ UNCHANGED ⟨state_vars, restart_vars, data_vars, collect_vars, lease_vars, commit_vars⟩
```

Handle a collect message. The peer will accept the proposal number from the leader if it is bigger than the last proposal number he accepted.
Variables changed: messages, epoch, state, *accepted_pn*

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

$first\_committed \mapsto first\_committed[mon]$,
$last\_committed \mapsto last\_committed[mon]$,
$values \mapsto values[mon]$,
$pn \mapsto accepted\_pn'[mon]]$, msg)
$\land$ UNCHANGED $epoch$
$\land$ UNCHANGED $\langle isLeader, phase, values, first\_committed, last\_committed, monitor\_store\rangle$
$\land$ UNCHANGED $\langle restart\_vars, collect\_vars, lease\_vars, commit\_vars, auxiliary\_vars\rangle$

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, epoch, phase, *uncommitted_v*, *uncommitted_value*, *monitor_store*, values, *accepted_pn*, *first_committed*, *last_committed*, *num_last*, *peer_first_committed*, *peer_last_committed*, *send_queue*.

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

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

$\land$ IF $msg.first\_committed > last\_committed[mon] + 1$
THEN
$\land$ $bootstrap$
$\land$ UNCHANGED $\langle num\_last, accepted\_pn, values, phase, monitor\_store\rangle$
$\land$ UNCHANGED $\langle first\_committed, last\_committed, restart\_vars, auxiliary\_vars\rangle$
ELSE
$\land$ $store\_state(mon, msg)$
$\land$ IF $\exists\, peer \in Monitors$:
$\land$ $peer \neq mon$
$\land$ $peer\_last\_committed'[mon][peer] \neq -1$
$\land$ $peer\_last\_committed'[mon][peer] + 1 < first\_committed[mon]$
$\land$ $first\_committed[mon] > 1$
THEN
$\land$ $bootstrap$
$\land$ $check\_and\_correct\_uncommitted(mon)$
$\land$ UNCHANGED $\langle phase, accepted\_pn, num\_last, auxiliary\_vars\rangle$
ELSE
$\land$ LET $monitors\_behind \triangleq \{peer \in Monitors$:
$\land$ $peer \neq mon$
$\land$ $peer\_last\_committed'[mon][peer] \neq -1$
$\land$ $peer\_last\_committed'[mon][peer] < last\_committed[mon]\}$
IN $send\_queue' = [send\_queue$ EXCEPT $![mon] =$
$send\_queue[mon] \circ SetToSeq((monitors\_behind) \times \{OP\_COMMIT\})]$

```
                    ∧ ∨ ∧ msg.pn > accepted_pn[mon]
                        ∧ collect(mon, msg.pn)
                        ∧ check_and_correct_uncommitted(mon)
                        ∧ UNCHANGED num_last

                      ∨ ∧ msg.pn = accepted_pn[mon]
                        ∧ num_last' = [num_last EXCEPT ![mon] = num_last[mon] + 1]

                        ∧ IF ∧ msg.last_committed + 1 ∈ DOMAIN msg.values
                             ∧ msg.last_committed ≥ last_committed'[mon]
                             ∧ msg.last_committed + 1 ≥ uncommitted_v[mon]
                          THEN ∧ uncommitted_v' =
                                    [uncommitted_v EXCEPT ![mon] = msg.last_committed + 1]
                               ∧ uncommitted_value' =
                                    [uncommitted_value EXCEPT ![mon] = msg.values[msg.last_committed + 1]]
                          ELSE check_and_correct_uncommitted(mon)

                        ∧ UNCHANGED ⟨phase, accepted_pn⟩

                      ∨ ∧ msg.pn < accepted_pn[mon]
                        ∧ check_and_correct_uncommitted(mon)
                        ∧ UNCHANGED ⟨phase, accepted_pn, num_last⟩
                    ∧ UNCHANGED epoch
                ∧ UNCHANGED ⟨epoch⟩

            ∧ Discard(msg)
            ∧ UNCHANGED ⟨isLeader, state⟩
            ∧ UNCHANGED ⟨lease_vars, commit_vars⟩
```

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, *send_queue*, values, *uncommitted_v*, *uncommitted_value*, *acked_lease*.

```
post_last(mon) ≜
    ∧ isLeader[mon] = TRUE
    ∧ num_last[mon] = Len(ranks)
    ∧ phase[mon] = PHASE_COLLECT

    ∧ clear_peer_first_committed(mon)
    ∧ clear_peer_last_committed(mon)

    ∧ IF ∧ uncommitted_v[mon] = last_committed[mon] + 1
         ∧ uncommitted_value[mon] ≠ Nil
      THEN ∧ state' = [state EXCEPT ![mon] = STATE_UPDATING_PREVIOUS]
           ∧ begin(mon, uncommitted_value)
           ∧ UNCHANGED ⟨acked_lease⟩
      ELSE ∧ finish_round(mon)
```

$\land$ *extend_lease*(*mon*)
$\land$ UNCHANGED $\langle$*accepted*, *new_value*, *values*, *restart_vars*$\rangle$

$\land$ UNCHANGED $\langle$*isLeader*, *monitor_store*, *accepted_pn*, *first_committed*, *last_committed*$\rangle$
$\land$ UNCHANGED $\langle$*num_last*, *pending_proposal*$\rangle$
$\land$ UNCHANGED $\langle$*global_vars*$\rangle$

Leader election

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

leader_election $\triangleq$
$\land \exists\, mon \in Monitors$:
$\land$ *isLeader*$'$ = [$m \in Monitors \mapsto$ IF $m = mon$ THEN TRUE ELSE FALSE]
$\land$ *state*$'$ = [$m \in Monitors \mapsto$
IF *Len*(*ranks*) = 1 THEN *STATE_ACTIVE* ELSE *STATE_RECOVERING*]
$\land$ *phase*$'$ = [$m \in Monitors \mapsto PHASE\_ELECTION$]
$\land$ *new_value*$'$ = [$m \in Monitors \mapsto Nil$]
$\land$ *pending_proposal*$'$ = [$m \in Monitors \mapsto Nil$]
$\land$ *epoch*$'$ = *epoch* + 1
$\land$ UNCHANGED $\langle$*accepted*, *messages*, *send_queue*$\rangle$
$\land$ UNCHANGED $\langle$*data_vars*, *restart_vars*, *collect_vars*, *lease_vars*$\rangle$

Start recovery phase if number of monitors is greater than 1.
Variables changed: *accepted_pn*, phase.

election_recover(*mon*) $\triangleq$
$\land$ *Len*(*ranks*) > 1
$\land$ *phase*[*mon*] = *PHASE_ELECTION*
$\land$ *collect*(*mon*, 0)
$\land$ UNCHANGED $\langle$*isLeader*, *state*, *values*, *first_committed*, *last_committed*, *monitor_store*$\rangle$
$\land$ UNCHANGED $\langle$*global_vars*, *restart_vars*, *collect_vars*, *lease_vars*, *commit_vars*, *auxiliary_vars*$\rangle$

Timeouts and restart

Restart a monitor and wipe variables that are not persistent.
Variables changed: messages, *isLeader*, phase, state, *pending_proposal*, *new_value*, *number_refreshes*.

restart_mon(*mon*) $\triangleq$
$\land$ *messages*$'$ = [$m \in$ DOMAIN *messages* $\mapsto$ IF $m.from = mon$ THEN 0 ELSE *messages*[m]]
$\land$ *isLeader*$'$ = [*isLeader* EXCEPT ![*mon*] = FALSE]
$\land$ *phase*$'$ = [*phase* EXCEPT ![*mon*] = *PHASE_ELECTION*]
$\land$ *state*$'$ = [*state* EXCEPT ![*mon*] = IF *Len*(*ranks*) = 1
THEN *STATE_ACTIVE*
ELSE *STATE_RECOVERING*]
$\land$ *pending_proposal*$'$ = [*pending_proposal* EXCEPT ![*mon*] = *Nil*]
$\land$ *new_value*$'$ = [*new_value* EXCEPT ![*mon*] = *Nil*]

$\land$ $number\_refreshes' = number\_refreshes + 1$
$\land$ UNCHANGED $\langle epoch, accepted \rangle$
$\land$ UNCHANGED $\langle restart\_vars, data\_vars, collect\_vars, lease\_vars, auxiliary\_vars \rangle$

Monitor timeout (simulate message not received). Triggers new elections.
Messages in network and events in *send_queue* are cleared.
Variables changed: epoch, *send_queue*, messages.

$Timeout(mon) \triangleq$
$\land$ $phase[mon] = PHASE\_COLLECT \lor phase[mon] = PHASE\_BEGIN$
$\land$ $bootstrap$
$\land$ $send\_queue' = [m \in Monitors \mapsto \langle\rangle]$
$\land$ $messages' = [m \in \text{DOMAIN } messages \mapsto 0]$
$\land$ UNCHANGED $\langle state\_vars, restart\_vars, data\_vars, collect\_vars, lease\_vars, commit\_vars \rangle$

Dispatchers and next statement

Handle a message.

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

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

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

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

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

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

$\lor \land msg.type = OP\_COMMIT$

```
            ∧ handle_commit(msg.dest, msg)
            ∧ step_x' = "receive commit"

    Send a message from the event queue.
Send_from_queue(mon) ≜
    ∧ Len(send_queue[mon]) > 0
    ∧
       ∨ ∧ Head(send_queue[mon])[2] = OP_COLLECT
         ∧ send_collect(mon, Head(send_queue[mon])[1])
         ∧ step_x' = "send_collect"

       ∨ ∧ Head(send_queue[mon])[2] = OP_LEASE
         ∧ send_extend_lease(mon, Head(send_queue[mon])[1])
         ∧ step_x' = "send_extend_lease"

       ∨ ∧ Head(send_queue[mon])[2] = OP_BEGIN
         ∧ send_begin(mon, Head(send_queue[mon])[1])
         ∧ step_x' = "send_begin"

       ∨ ∧ Head(send_queue[mon])[2] = OP_COMMIT
         ∧ send_commit(mon, Head(send_queue[mon])[1])
         ∧ step_x' = "send_commit"
    ∧ send_queue' = [send_queue EXCEPT ![mon] = Tail(send_queue[mon])]

    Limit some variables to reduce search space.
reduce_search_space ≜
    ∧ epoch < 5
    ∧ ∀ mon ∈ Monitors : accepted_pn[mon] < 300
    ∧ ∀ mon ∈ Monitors : last_committed[mon] < 2
    ∧ number_refreshes < 3
     ∧ step < 100

    State transitions.
Next ≜
    ∧ reduce_search_space
    ∧ IF epoch%2 = 1 THEN
        ∧ leader_election
        ∧ step_x' = "election" ∧ step' = step + 1
        ∧ UNCHANGED number_refreshes
       ELSE
       IF ∃ mon ∈ Monitors : Len(send_queue[mon]) > 0
        THEN ∧ ∃ mon ∈ Monitors : Send_from_queue(mon)
             ∧ step' = step + 1
             ∧ UNCHANGED number_refreshes
        ELSE
        ∨
           ∧ ∃ mon ∈ Monitors : election_recover(mon)
```

$\land\ step\_x' = \text{“election\_recover”} \land step' = step + 1$
$\land$ UNCHANGED $number\_refreshes$

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

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

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

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

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

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

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

$\lor\ \land \exists\, m \in ValidMessage(messages) : Receive(m)$
$\land\ step' = step + 1$
$\land$ UNCHANGED $number\_refreshes$

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

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

Test/Debug invariants

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

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

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

$Inv\_diam(size) \triangleq step < size - 1$

Invariants to test in model check

$Inv \triangleq \land \text{TRUE}$
$\quad \land Inv\_diam(20)$

Examples:

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

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

```
Inv_find_state(
  ∃ msg1, msg2 ∈ DOMAIN messages :
     ∧ msg1.type = OP_COLLECT ∧ msg2.type = OP_COLLECT
     ∧ msg1.from ≠ msg2.from
)
```

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

```
Inv_find_state(
   ∧ step_x = "restart mon"

   \ * The system is in collect phase and no OP_LAST message has been received.
   \ * isLeader[mon] = TRUE assures that the leader was not the one that crashed.
   ∧ ∃ mon ∈ Monitors :
      ∧ isLeader[mon] = TRUE
      ∧ phase[mon] = PHASE_COLLECT
      ∧ num_last[mon] = 1

   \ * All the collect requests have been handled by the peers.
   ∧ ∀ m ∈ ValidMessage(messages) :
     m.type ≠ OP_COLLECT

   ∧ number_refreshes = 2
   ∧ epoch = 2
)
```

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

```
Inv_find_state(
   ∧ step_x = "restart mon"
   ∧ ∃ msg ∈ ValidMessage(messages) :
     msg.type = OP_ACCEPT
   ∧ ∀ mon ∈ Monitors :
     isLeader[mon] = FALSE
   ∧ number_refreshes = 2
   ∧ 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 *Mon Mar* 01 11:27:57 WET 2021 by *afonsonf*

\ ∗ Created *Mon Jan* 11 16:15:26 WET 2021 by *afonsonf*