N TLA+ Specification

MODULE *Nezha*

EXTENDS *Naturals*, *TLC*, *FiniteSets*, *Sequences*

Bounds for Model Check [Configurable]

Time Range [Configurable]

$MaxTime \triangleq 3$

Each client is only allowed to submit *MaxReqNum* requests [Configurable]
In the specification, we will only consider two roles, client and replicas
(*i.e.* it can be considered as co-locating one proxy with one client)
For the proxy-based design, we just need to replace client with proxy,
and then the specification describes the interaction between proxy and replicas

$MaxReqNum \triangleq 1$

The leader is only allowed to crash when the view $< MaxViews$ [Configurable]

$MaxViews \triangleq 3$

These variables are used to implment at-most-once primitives
i.e. The variables record the messages processed by *Replicas/Clients*, so
that the *Replicas/Clients* will not process twice

VARIABLE $vReplicaProcessed$, Messages that have been processed by replicas
$\quad vClientProcessed$ Messages that have been processed by clients

VARIABLE $DebugAction$

Constants

The set of replicas and an ordering of them

CONSTANTS $Replicas$, $ReplicaOrder$, $Clients$, $LatencyBounds$
ASSUME $IsFiniteSet(Replicas)$
ASSUME $ReplicaOrder \in Seq(Replicas)$

$F \triangleq (Cardinality(Replicas) - 1) \div 2$
$ceilHalfF \triangleq \text{IF } (F \div 2) * 2 = F \text{ THEN } F \div 2 \text{ ELSE } (F + 1) \div 2$
$floorHalfF \triangleq F \div 2$
$QuorumSize \triangleq F + 1$
$FastQuorumSize \triangleq F + ceilHalfF + 1$
$RecoveryQuorumSize \triangleq ceilHalfF + 1$
$FastQuorums \triangleq \{R \in \text{SUBSET } (Replicas) : Cardinality(R) \geq FastQuorumSize\}$
$Quorums \triangleq \{R \in \text{SUBSET } (Replicas) : Cardinality(R) * 2 > Cardinality(Replicas)\}$

Replica Statuses

$StNormal \triangleq 1$

$StViewChange \triangleq 2$

$StRecovering \triangleq 3$

Message Types

$MClientRequest \triangleq 1$ Sent by client to replicas

$MFastReply \triangleq 2$ Fast Reply Message

$MSlowReply \triangleq 3$ Slow Reply Message

$MLogIndex \triangleq 4$ *LogIndex*

$MLogEntry \triangleq 5$ Log entry, different from index, it includes command field, which can be large in practice

$MIndexSync \triangleq 6$ Sync message during the index sync process

$MMissEntryRequest \triangleq 7$ Sent by followers once they fail to find the entry on itself

$MMissEntryReply \triangleq 8$ Response to *MMissEntryRequest*, providing the missing entries

$MViewChangeReq \triangleq 9$ Sent when leader/sequencer failure detected

$MViewChange \triangleq 10$ Sent to *ACK* view change

$MStartView \triangleq 11$ Sent by new leader to start view

The following messages are mainly used for periodic sync

Just as described in *NOPaxos*, it is an optional optimization to enable fast recovery after failure

$MSyncPrepare \triangleq 12$ Sent by the leader to ensure *log* durability

$MSyncRep \triangleq 13$ Sent by followers as *ACK*

$MSyncCommit \triangleq 14$ Sent by leaders to indicate stable *log*

The following messages are mainly used for replica recovery

$MCrashVectorReq \triangleq 15$

$MCrashVectorRep \triangleq 16$

$MRecoveryReq \triangleq 17$

$MRecoveryRep \triangleq 18$

$MStateTransferReq \triangleq 19$

$MStateTransferRep \triangleq 20$

Message Schemas

$ViewIDs \triangleq [\; leaderNum \mapsto n \in \; (1\,..\,)\;]$

\ ∗ < *clientID*, *requestID* > uniquely identifies one request on one replica

\ ∗ But across replicas, the same < *clientID*, *requestID* > may have different deadlines

\ ∗ (the leader may modify the deadline to make the request eligible to enter the early-buffer)

\ ∗ so < *deadline*, *clientID*, *reqID* > uniquely identifes one request across replicas

```
ClientRequest
  [ mtype      ↦ MClientRequest,
    sender     ↦ c ∈ Clients,
    dest       ↦ r ∈ Replicas,
    requestID  ↦ i ∈  (1 . . ),
    command    ↦ “”,
    s          ↦ t ∈  (1 . . MaxTime),
    l          ↦ l ∈  (1 . . MaxBound)
  ]
```

```
\ * logSlotNum is not necessary and it is not described in the paper
\ *  Here we include logSlotNum in FastReply and SlowReply messages
\ *  to facilitate the check of Linearizability invariant
FastReply
   [ mtype       ↦ MFastReply,
     sender      ↦ r ∈ Replicas,
     dest        ↦ c ∈ Clients,
     viewID      ↦ v ∈ ViewIDs,
     requestID ↦ i ∈  (1 . . vClientReqNum)
     hash        ↦  [ log ↦ vLogs[1 . . n],
                      cv ↦ crashVector
                    ]
     deadline ↦ i ∈  (1 . . MaxTime + MaxBound),
     logSlotNum ↦ n ∈  (1 . . )
   ]

SlowReply
   [ mtype       ↦ MSlowReply,
     sender      ↦ r ∈ Replicas,
     dest        ↦ c ∈ Clients,
     viewID      ↦ v ∈ ViewIDs,
     requestID ↦ i ∈  (1 . . vClientReqNum)
     logSlotNum ↦ n ∈  (1 . . )
   ]

LogIndex
   [ mtype       ↦ MLogIndex,
     clientID ↦ c ∈ Clients,
     requestID ↦ i ∈  (1 . . vClientReqNum),
     deadline ↦ i ∈  (1 . . MaxTime + MaxBound),
   ]

LogEntry
   [ mtype       ↦ MLogEntry,
     clientID ↦ c ∈ Clients,
     requestID ↦ i ∈  (1 . . vClientReqNum),
     deadline ↦ i ∈  (1 . . MaxTime + MaxBound),
     command ↦ “”
   ]

IndexSync
   [ mtype       ↦ MIndexSync,
     sender      ↦ r ∈ Replicas,
     dest        ↦ c ∈ Clients,
     viewID      ↦ v ∈ ViewIDs,
     logindcies ↦ index ∈ vLogs[leaderIdx]
   ]

 MMissEntryRequest
   [ mtype       ↦ MMissEntryRequest,
     sender      ↦ r ∈ Replicas,
     dest        ↦ d ∈ Replicas,
    viewID     ↦ v ∈ ViewIDs,
    miss       ↦ {log indices}
  ]

MMissEntryRequest
  [ mtype      ↦ MMissEntryReply,
    sender     ↦ r ∈ Replicas,
    dest       ↦ d ∈ Replicas,
    viewID     ↦ v ∈ ViewIDs,
    entries ↦ {log entries}
  ]

ViewChangeReq
  [ mtype ↦ MViewChangeReq,
    sender ↦ r ∈ Replicas,
    dest ↦ r ∈ Replicas,
    viewID ↦ v ∈ ViewIDs,
    cv     ↦  crash vector
  ]

ViewChange
  [ mtype      ↦ MViewChange,
    sender     ↦ r ∈ Replicas,
    dest       ↦ r ∈ Replicas,
    viewID     ↦ v ∈ ViewIDs,
    lastNormal ↦ v ∈ ViewIDs,
    log        ↦ l ∈ vLogs[1 .. n],
    cv         ↦  crash vector
  ]

StartView
  [ mtype      ↦ MStartView,
    dest       ↦ r ∈ Replicas,
    viewID     ↦ v ∈ ViewIDs,
    log        ↦ l ∈ vLogs[1 .. n],
    cv         ↦  crash vector
  ]

SyncPrepare
  [ mtype      ↦ MSyncPrepare,
    dest       ↦ r ∈ Replicas,
    sender     ↦ r ∈ Replicas,
    viewID     ↦ v ∈ ViewIDs,
    log        ↦ l ∈ vLogs[1 .. n] ]

SyncRep
  [ mtype         ↦ MSyncRep,
    dest          ↦ r ∈ Replicas,
    sender        ↦ r ∈ Replicas,
    viewID        ↦ v ∈ ViewIDs,
    logSlotNumber ↦ n ∈  (1 .. ) ]

SyncCommit
```

```
  [ mtype        ↦ MSyncCommit,
    dest         ↦ r ∈ Replicas,
    sender       ↦ r ∈ Replicas,
    viewID       ↦ v ∈ ViewIDs,
    log          ↦ l ∈ vLogs[1 . . n] ]

CrashVectorReq
  [ mtype        ↦ MCrashVectorReq,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    nonce        ↦ nonce
  ]
CrashVectorRep
  [ mtype        ↦ MCrashVectorRep,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    nonce        ↦ nonce,
    cv           ↦  vector of counters
  ]

RecoveryReq
  [ mtype        ↦ MRecoveryReq,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    cv           ↦  vector of counters
  ]

RecoveryRep
  [ mtype        ↦ MRecoveryRep,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    viewID       ↦ v ∈ ViewIDs,
    cv           ↦  vector of counters
  ]

StateTransferReq
  [ mtype        ↦ MStateTransferReq,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    cv           ↦  vector of counters
  ]
StateTransferRep
  [ mtype        ↦ MStateTransferRep,
    sender       ↦ r ∈ Replicas,
    dest         ↦ r ∈ Replicas,
    viewID       ↦ v ∈ ViewIDs,
    log          ↦ l ∈ vLogs[1 . . n] ],
    cv           ↦  vector of counters
  ]
```

Variables

Network State

VARIABLE $messages$ Set of all messages sent

$networkVars \triangleq \langle messages \rangle$
$InitNetworkState \triangleq messages = \{\}$

Used as a dummy value

$$NULLLog \triangleq [\; deadline \mapsto 0,\; clientID \mapsto 0,\; requestID \mapsto 0 \;]$$

Replica State

VARIABLES	$vLog$,	Log of values
	$vEarlyBuffer$,	The early buffer to hold request,
		and release it after clock passes its deadline ($s + l$)
	$vReplicaStatus$,	One of *StNormal*, *StViewChange*, *StRecovering*
	$vViewID$,	Current *viewID* replicas recognize
	$vReplicaClock$,	Current Time of the replica
	$vLastNormView$,	Last views in which replicas had status *StNormal*
	$vViewChanges$,	Used for logging view change votes
	$vSyncPoint$,	Latest synchronization point,
		to which the replica state (*vLog*) is consistent with the leader.
	$vLateBuffer$,	The late buffer Used to store the requests
		which are not eligible to enter *vEarlyBuffer*
	$vTentativeSync$,	Used by leader to mark current *syncPrepare* point (during periodic sync process)
		(Actually, *vSyncPoint* and *vTentativeSync* can be merged into one *Var*
		However, we decouple them to make the spec easy to understand)
	$vSyncReps$,	Used for logging sync reps at leader
	$vCommitPoint$,	Different from *vSyncPoint*,
		vCommitPoint indicates that the logs before this point has been replicated to majority
		So followers can safely execute requests (*log* entries) up to *vCommitPoint*
		Refer to “Acceleration of Recovery“ para in *Sec* 6
	$vUUIDCounter$,	Locally unique string (for *CrashVectorReq*)
	$vCrashVector$,	*CrashVector*, initialized as all-zero vector
	$vCrashVectorReps$,	*CrashVectorRep* Set
	$vRecoveryReps$	*RecoveryRep* Set

$$\begin{aligned} replicaVars \triangleq \langle & vLog,\ vEarlyBuffer, \\ & vViewID,\ vReplicaClock, \\ & vLastNormView,\ vViewChanges,\ vReplicaStatus, \\ & vSyncPoint,\ vLateBuffer, \\ & vTentativeSync,\ vSyncReps,\ vCommitPoint, \\ & vUUIDCounter,\ vCrashVector, \end{aligned}$$

$$vCrashVectorReps,\ vRecoveryReps\rangle$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{InitReplicaState\ \triangleq} \\
\quad \land\ vLog & = [r \in Replicas \mapsto \langle\rangle] \\
\quad \land\ vEarlyBuffer & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vViewID & = [r \in Replicas \mapsto 1] \quad \text{0 should also be okay} \\
\quad \land\ vReplicaClock & = [r \in Replicas \mapsto 1] \\
\quad \land\ vLastNormView & = [r \in Replicas \mapsto 1] \\
\quad \land\ vViewChanges & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vReplicaStatus & = [r \in Replicas \mapsto StNormal] \\
\quad \land\ vSyncPoint & = [r \in Replicas \mapsto 0] \\
\quad \land\ vLateBuffer & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vTentativeSync & = [r \in Replicas \mapsto 0] \\
\quad \land\ vSyncReps & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vCommitPoint & = [r \in Replicas \mapsto 0] \\
\quad \land\ vCrashVector & = [r \in Replicas \mapsto [rr \in Replicas \mapsto 0]] \\
\quad \land\ vCrashVectorReps & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vRecoveryReps & = [r \in Replicas \mapsto \{\}] \\
\quad \land\ vUUIDCounter & = [c \in Replicas \ \ \mapsto 0]
\end{array}
$$

Client State

$$
\begin{array}{lll}
\text{VARIABLES} & vClientClock, & \text{Current Clock Time of the client} \\
 & vClientReqNum & \text{The number of requests that have been sent by this client}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{InitClientState\ \triangleq} \\
\quad \land\ vClientClock & = [c \in Clients \ \ \mapsto 1] \\
\quad \land\ vClientReqNum & = [c \in Clients \ \ \mapsto 0]
\end{array}
$$

$$clientVars \triangleq \langle vClientClock,\ vClientReqNum\rangle$$

Set of all vars

$$vars \triangleq \langle networkVars,\ replicaVars,\ clientVars\rangle$$

\ ∗ Initial state

$$
\begin{array}{ll}
Init \triangleq & \land\ InitNetworkState \\
 & \land\ InitReplicaState \\
 & \land\ InitClientState \\
 & \land\ vReplicaProcessed = [r \in Replicas \mapsto \{\}] \\
 & \land\ vClientProcessed = [c \in Clients \mapsto \{\}] \\
 & \land\ DebugAction = \langle \text{“Init”},\ \text{“”}\rangle
\end{array}
$$

Helpers

$$NumofReplicas(status) \triangleq Cardinality(\{r \in Replicas \quad : vReplicaStatus[r] = status\})$$

$$DuplicateRep(ReplySet,\ m) \triangleq m.sender \in \{mm.sender : mm \in ReplySet\}$$

```
$Pick(S) \triangleq$ CHOOSE $s \in S$ : TRUE
```

Convert a Set to Sequence

```
RECURSIVE $Set2Seq(\_)$
$Set2Seq(S) \triangleq$ IF $Cardinality(S) = 0$ THEN $\langle\rangle$
                 ELSE
                LET
                  $x \triangleq$ CHOOSE $x \in S$ : TRUE
                IN
                  $\langle x \rangle \circ Set2Seq(S \setminus \{x\})$
```

Convert a Sequence to Set

```
$Seq2Set(seq) \triangleq \{seq[i] : i \in$ DOMAIN $seq\}$

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

$Min(S) \triangleq$ CHOOSE $x \in S : \forall y \in S : x \leq y$
```

View ID Helpers

```
$LeaderID(viewID) \triangleq (viewID \% Len(ReplicaOrder)) + ($IF $viewID \geq Len(ReplicaOrder)$ THEN $1$ ELSE $0)$

$Leader(viewID) \triangleq ReplicaOrder[LeaderID(viewID)]$
```

remember $\langle\rangle$ are 1-indexed

Log Manipulation Helpers

The order of 2 *log* entries are decided by the tuple $< deadline, clientID, requestID >$
Usually, deadline makes the two entries comparable
When 2 different entries have the same deadline, the tie is broken with *clientID*
Further, the tie is broken is *requestID*
(unnecessary if we only allow client to submit one request at one tick)

```
$EntryLeq(l1, l2)$          $\triangleq$ $\land\ l1.deadline \leq l2.deadline$
                              $\land\ l1.clientID \leq l2.clientID$
                              $\land\ l1.requestID \leq l2.requestID$

$EntryEq(l1, l2)$           $\triangleq$ $\land\ l1.deadline = l2.deadline$
                              $\land\ l1.clientID = l2.clientID$
                              $\land\ l1.requestID = l2.requestID$

$EntryLessThan(l1, l2)$     $\triangleq$ $\land\ EntryLeq(l1, l2)$
                              $\land\ \neg(EntryEq(l1, l2))$
```

Find entry in one replica's *log* ($< clientID, reqID >$ can uniquely identify the *log* entry)
We do not check deadline, because the leader may have modified the request's deadline
Return 0 when we fail to find it (remember Sequence is 1-indexed in TLA+, so 0 can serve as a dummy value)

```
$FindEntry(clientID, reqID, log) \triangleq$
                    LET
                      $entryIndexSet \triangleq \{i \in 1 \,..\, Len(log) :$ $\land\ log[i].clientID = clientID$
                                                      $\land\ log[i].reqID = reqID\}$
```

```
                                      IN
                                        IF  Cardinality(entryIndexSet) = 0 THEN
                                            0
                                         ELSE
                                             Pick(entryIndexSet)

SortLogSeq(seq) ≜ SortSeq(seq, LAMBDA x, y : EntryLessThan(x, y))
```

Given a set of logs, return the sorted *log* list

```
GetSortLogSeq(S) ≜ LET
                          seq ≜ Set2Seq(S)
                      IN
                          SortLogSeq(seq)
```

Merge logs, first put all *log* items together, deduplicated (*i.e.* UNION them into a set). Then, do filtering and only keep those that have appeared in at least $\lceil f/2 \rceil + 1\mathit{replicas}$.

```
CountVotes(logll, x) ≜  Cardinality({logSet ∈ logll : x ∈ logSet})

MergeUnSyncLogs(unSyncedLogs, lastSyncedLog) ≜
    LET
        unSyncedLogSet ≜ UNION unSyncedLogs
        votedLogSet ≜ {x ∈ unSyncedLogSet :
                            ∧ EntryLessThan(lastSyncedLog, x)
                            ∧ CountVotes(unSyncedLogs, x) ≥ RecoveryQuorumSize}
    IN
        GetSortLogSeq(votedLogSet)
```

Network Helpers

Add a message to the network

```
Send(ms) ≜ messages′ = messages ∪ ms
```

Convert the request format to a *log* format (by summing up *s* and *l* to get deadline)

```
Req2Log(req) ≜ [  mtype        ↦ MLogEntry,
                  deadline     ↦ req.s + req.l,
                  clientID     ↦ req.sender,
                  requestID    ↦ req.requestID,
                  command      ↦ req.command
               ]
```

Index does not need to include command field, which is the body of the request/*log*, and can be very large

```
GetLogIndex(entry) ≜ [
                        mtype       ↦ MLogIndex,
                        deadline    ↦ entry.deadline,
                        clientID    ↦ entry.clientID,
                        requestID   ↦ entry.requestID
                     ]
```

$$\mathit{GetLogIndexFromReply}(\mathit{reply}) \triangleq [\begin{array}{ll} \mathit{mtype} & \mapsto \mathit{MLogIndex}, \\ \mathit{deadline} & \mapsto \mathit{reply.deadline}, \\ \mathit{clientID} & \mapsto \mathit{reply.dest}, \\ \mathit{requestID} & \mapsto \mathit{reply.requestID} \end{array}]$$

$$\mathit{IndexEq}(\mathit{index}, \mathit{msg}) \triangleq \begin{array}{l} \land\ \mathit{index.deadline} = \mathit{msg.deadline} \\ \land\ \mathit{index.clientID} = \mathit{msg.clientID} \\ \land\ \mathit{index.requestID} = \mathit{msg.requestID} \end{array}$$

Add local time to the message (for easy debug)

$$\mathit{Msg2RLog}(\mathit{msg}, r) \triangleq \mathit{msg}\ @@\ [\mathit{tl} \mapsto \mathit{vReplicaClock}[r]]$$

$$\mathit{LastLog}(\mathit{logList}) \triangleq \text{IF } \mathit{Len}(\mathit{logList}) = 0 \text{ THEN } \mathit{NULLLog} \text{ ELSE } \mathit{logList}[\mathit{Len}(\mathit{logList})]$$

$$\mathit{MergeCrashVector}(\mathit{cv}1, \mathit{cv}2) \triangleq [r \in \mathit{Replicas} \mapsto \mathit{Max}(\{\mathit{cv}1[r], \mathit{cv}2[r]\})]$$

```
$CheckCrashVector(m, r) \triangleq$
    IF $m.cv[m.sender] < vCrashVector[r][m.sender]$ THEN
        FALSE    Potential stray message
    ELSE
        $vCrashVector' = [vCrashVector$ EXCEPT $![r] = MergeCrashVector(m.cv, vCrashVector[r])]$
```

$$\mathit{FilterStrayMessage}(\mathit{MSet}, \mathit{cv}) \triangleq \{m \in \mathit{MSet} : m.\mathit{cv}[m.\mathit{sender}] \geq \mathit{cv}[m.\mathit{sender}]\}$$

Message Handlers and Actions

Client action

Client c sends a request

We assume client can only send one request in one tick of time

If time has reached the bound, this client cannot send request any more

```
$ClientSendRequest(c) \triangleq$
    $\land vClientClock[c] < MaxTime$
    $\land vClientReqNum[c] < MaxReqNum$
    $\land Send(\{[mtype \mapsto MClientRequest,$
            $sender     \mapsto c,$                         clientID
            $requestID  \mapsto vClientReqNum[c] + 1,$      requestID
            $command    \mapsto$ “”,
            $s          \mapsto vClientClock[c],$           submission time
            $l          \mapsto LatencyBounds[c],$          latency bound
            $dest       \mapsto r$
            $] : r \in Replicas\})$
    $\land vClientClock' = [vClientClock$ EXCEPT $![c] = vClientClock[c] + 1]$
```

```
                                 ∧ vClientReqNum′ = [vClientReqNum EXCEPT ![c] = vClientReqNum[c] + 1]
                                 ∧ UNCHANGED ⟨replicaVars⟩


Duplicate(entry, logSet) ≜
  LET
      findSet ≜ {x ∈ logSet : ∧ x.clientID = entry.clientID
                              ∧ x.requestID = entry.requestID}
  IN
      Cardinality(findSet) > 0

 Replica r receives MClientRequest, m
HandleClientRequest(r, m) ≜
  LET
    mlog ≜ Req2Log(m)
  IN
   If the request is duplicate, it will no longer be appended to the log
   Replicas simply reply the previous execution result of this request
   (we do not model execution in this spec)
  ∧ ¬Duplicate(mlog, Seq2Set(vLog[r]) ∪ vEarlyBuffer[r])
  ∧ vReplicaStatus[r] = StNormal
     The request can enter the early buffer
  ∧ ∨ ∧ EntryLessThan(LastLog(vLog[r]), mlog)
      ∧ vEarlyBuffer′ = [
            vEarlyBuffer EXCEPT ![r] = vEarlyBuffer[r] ∪ {mlog}
        ]
      ∧ UNCHANGED ⟨networkVars, clientVars,
                   vLog, vViewID, vReplicaClock,
                   vLastNormView, vViewChanges, vReplicaStatus,
                   vSyncPoint, vLateBuffer,
                   vTentativeSync, vSyncReps, vCommitPoint,
                   vUUIDCounter, vCrashVector,
                   vCrashVectorReps, vRecoveryReps⟩
     (1) Followers' early buffers do not accept the request
       if its deadline is smaller than previously appended (last released) entry,
       so followers directly put the request into the late buffer
     (2) Leader modifies its deadline to be larger than the last released entry
       so as to make it eligible for entering the early buffer
    ∨ ∧ EntryLessThan(mlog, LastLog(vLog[r]))
      ∧ IF   r = Leader(vViewID[r])  THEN  this replica is the leader in the current view
              ∧ vEarlyBuffer′ = [
                   vEarlyBuffer EXCEPT ![r] = vEarlyBuffer[r] ∪ {[
                      mtype      ↦ MLogEntry,
                      clientID   ↦ mlog.clientID,
                      requestID  ↦ mlog.requestID,
                deadline      ↦ LastLog(vLog[r]).deadline + 1,
                command       ↦ mlog.command
             ]}
          ]
        ∧ UNCHANGED  ⟨networkVars, clientVars,
                      vLog, vViewID, vReplicaClock,
                      vLastNormView, vViewChanges, vReplicaStatus,
                      vSyncPoint, vLateBuffer,
                      vTentativeSync, vSyncReps, vCommitPoint,
                      vUUIDCounter, vCrashVector,
                      vCrashVectorReps, vRecoveryReps⟩
    ELSE   this replica is a follower in the current view
         ∧ vLateBuffer′ = [
               vLateBuffer EXCEPT ![r] = vLateBuffer[r] ∪ {mlog}
           ]

         ∧ UNCHANGED  ⟨networkVars, clientVars,
                       vLog, vEarlyBuffer, vViewID, vReplicaClock,
                       vLastNormView, vViewChanges, vReplicaStatus,
                       vSyncPoint, vTentativeSync,
                       vSyncReps, vCommitPoint,
                       vUUIDCounter, vCrashVector,
                       vCrashVectorReps, vRecoveryReps⟩
```

Release relevant requests from *vEarlyBuffer* and append to *vLog*, and then send a fast reply

```
FlushEarlyBuffer(r) ≜
  LET
    validLogSet ≜ {x ∈ vEarlyBuffer[r] :
                      ∧ x.deadline < vReplicaClock[r]   < rather than ≤
                      ∧ EntryLessThan(LastLog(vLog[r]), x)}
    validLogs ≜ GetSortLogSeq(validLogSet)
    newLogStart ≜ Len(vLog[r]) + 1
  IN
   ∧ vLog′ = [vLog EXCEPT ![r] = vLog[r] ∘ validLogs]
   ∧ vEarlyBuffer′ = [vEarlyBuffer EXCEPT ![r]
                   = {x ∈ vEarlyBuffer[r] : x.deadline ≥ vReplicaClock[r]}]   ≥ rather than >
   ∧ Send({[mtype        ↦ MFastReply,
            sender       ↦ r,
            dest         ↦ vLog′[r][i].clientID,
            viewID       ↦ vViewID[r],
            requestID    ↦ vLog′[r][i].requestID,
            hash         ↦ [
                              log ↦ SubSeq(vLog′[r], 1, i),
                              cv  ↦ vCrashVector
```

```
                         ],
            deadline     ↦ vLog'[r][i].deadline,
            logSlotNum   ↦ i
            ] : i ∈ newLogStart .. Len(vLog'[r])})
    ∧ IF r = Leader(vViewID[r]) THEN
          ∧ vSyncPoint' = [vSyncPoint EXCEPT ![r] = Len(vLog'[r])]
          ∧ UNCHANGED ⟨ clientVars, vViewID, vLastNormView, vViewChanges,
                        vReplicaStatus, vReplicaClock, vLateBuffer,
                        vTentativeSync, vSyncReps, vCommitPoint,
                        vUUIDCounter, vCrashVector,
                        vCrashVectorReps, vRecoveryReps⟩
      ELSE
          UNCHANGED ⟨clientVars, vViewID, vLastNormView, vViewChanges,
                     vReplicaStatus, vReplicaClock,
                     vSyncPoint, vLateBuffer,
                     vTentativeSync, vSyncReps, vCommitPoint,
                     vUUIDCounter, vCrashVector,
                     vCrashVectorReps, vRecoveryReps ⟩
```

Clock can be random value (*RandomElement*(1 .. *MaxTime*)),
because clock sync algorithm can give negative offset, or even fails
But *Nezha* depend on clock for performance but not for correctness
If the replica clock goes beyond *MaxTime*, it will stop processing
Since Clock is moved, then replicas can release relevant requests and append to logs

```
ReplicaClockMove(r) ≜ ∧ IF vReplicaClock[r] < MaxTime THEN
                            vReplicaClock' = [
                                vReplicaClock EXCEPT ![r] = RandomElement(1 .. MaxTime)
                            ]
                          ELSE
                            UNCHANGED vReplicaClock
                      ∧ UNCHANGED ⟨networkVars, clientVars,
                                   vLog, vEarlyBuffer, vViewID,
                                   vLastNormView, vViewChanges, vReplicaStatus,
                                   vSyncPoint, vLateBuffer, vTentativeSync,
                                   vSyncReps, vCommitPoint,
                                   vUUIDCounter, vCrashVector,
                                   vCrashVectorReps, vRecoveryReps⟩
```

Client clock move does not change any other things

```
ClientClockMove(c) ≜ ∧ IF vClientClock[c] < MaxTime THEN
                           vClientClock' = [
                               vClientClock EXCEPT ![c] = RandomElement(1 .. MaxTime)
                           ]
                         ELSE
                           UNCHANGED vClientClock
                     ∧ UNCHANGED ⟨networkVars, replicaVars, vClientReqNum⟩
```

Index Synchronization to Fix Set Inequality

```
Leader replica r starts index synchronization
StartIndexSync(r) ≜
  LET
    indices ≜ {GetLogIndex(vLog[r][i]) : i ∈ 1 .. Len(vLog[r])}
  IN
   ∧ r = Leader(vViewID[r])
   ∧ vReplicaStatus[r]  = StNormal
   ∧ Cardinality(indices) > 0   leader has log entries to sync
   ∧ Send({[mtype       ↦ MIndexSync,
            sender      ↦ r,
            dest        ↦ d,
            viewID      ↦ vViewID[r],
            logindcies  ↦ indices] : d ∈ Replicas})
   ∧ UNCHANGED ⟨clientVars, replicaVars⟩


GetSyncLogs(logSeq, indices) ≜
    LET
        logSet ≜ {l ∈ Seq2Set(logSeq) : ∃ index ∈ indices : EntryEq(index, l)}
    IN
        GetSortLogSeq(logSet)

GetUnSyncLogs(logSeq, lastSyncedLog) ≜
    LET
        logSet ≜ {l ∈ Seq2Set(logSeq) : EntryLessThan(lastSyncedLog, l)}
    IN
        GetSortLogSeq(logSet)

Replica r receives IndexSync message, m
HandleIndexSync(r, m) ≜
   ∧ r ≠ Leader(vViewID[r])
   ∧ vReplicaStatus[r] = StNormal
   ∧ m.viewID = vViewID[r]
   ∧ m.sender  = Leader(vViewID[r])
   ∧ vSyncPoint[r] < Len(m.logindcies)
   ∧ LET
        entries ≜ {vLog[r][i] : i ∈ 1 .. Len(vLog[r])}
        indices ≜ {GetLogIndex(vLog[r][i]) : i ∈ 1 .. Len(vLog[r])}
        missedEntries ≜ m.indices \ indices
     IN
         Missing some log entries → Send MMissEntryRequest
        IF Cardinality(missedEntries) > 0 THEN
            ∧ Send({[ mtype        ↦ MMissEntryRequest,
                       sender       ↦ r,
                       dest         ↦ d,
                       viewID       ↦ vViewID[r],
                       miss         ↦ missedEntries] : d ∈ (Replicas \ {r})})
                ∧ UNCHANGED ⟨vLog, vSyncPoint⟩
            No missing entries, update vLog and vSyncPoint, and send relevant slow replies
            ELSE
                LET
                    syncLogs ≜  GetSyncLogs(vLog[r], indices)
                    unsyncLogs ≜  GetUnSyncLogs(vLog[r], LastLog(syncLogs))
                IN
                ∧ vLog′ = [vLog EXCEPT ![r] = syncLogs ∘ unsyncLogs]
                ∧ vSyncPoint′ = [vSyncPoint EXCEPT ![r] = Len(syncLogs)]
                ∧ Send({[  mtype       ↦ MSlowReply,
                           sender      ↦ r,
                           dest        ↦ vLog′[r][i].clientID,
                           viewID      ↦ vViewID[r],
                           requestID   ↦ vLog′[r][i].requestID,
                           logSlotNum  ↦ i] : i ∈ (1 . . Len(syncLogs))})

   ∧ UNCHANGED ⟨clientVars, vEarlyBuffer, vViewID, vReplicaClock,
                vLastNormView, vViewChanges,  vReplicaStatus,
                vLateBuffer, vTentativeSync, vSyncReps, vCommitPoint,
                vUUIDCounter, vCrashVector,
                vCrashVectorReps, vRecoveryReps⟩


FindEntries(log, indices) ≜
    {l ∈ Seq2Set(log)  : ∃ x ∈ indices : IndexEq(l, x)}

 Replica r receives a request from other replicas, asking for a missing log entry
HandleMissEntryRequest(r, m) ≜
   ∧ m.viewID = vViewID[r]
   ∧ LET
        findentries ≜ FindEntries(vLog[r], m.miss)
     IN
     ∧ Cardinality(findentries) > 0
     ∧ Send({[  mtype       ↦ MMissEntryReply,
                sender      ↦ r,
                dest        ↦ m.sender,
                viewID      ↦ vViewID[r],
                entries     ↦ findentries]})
     ∧ UNCHANGED ⟨clientVars, replicaVars⟩


 Replica r receives a reply from other replicas, providing the missing entries
```

```
$HandleMissEntryReply(r, m) \triangleq$
    $\land m.viewID = vViewID[r]$
    $\land$ LET
        $mergedSet \triangleq Seq2Set(vLog[r]) \cup m.entries$
      IN
        $vLog' = [vLog$ EXCEPT $![r] = GetSortLogSeq(mergedSet)]$
    $\land$ UNCHANGED $\langle networkVars, clientVars,$
                        $vEarlyBuffer, vViewID, vReplicaClock,$
                        $vLastNormView, vViewChanges, vReplicaStatus,$
                        $vSyncPoint, vLateBuffer,$
                        $vTentativeSync, vSyncReps, vCommitPoint,$
                        $vUUIDCounter, vCrashVector,$
                        $vCrashVectorReps, vRecoveryReps\rangle$
```

Replica Rejoin

Failed replica loses all states

```
$StartReplicaFail(r) \triangleq$
    $\land NumofReplicas(StRecovering) < F$   We assume at most $F$ replicas can fail at the same time
    $\land vReplicaStatus' = [vReplicaStatus$ EXCEPT $![r] = StRecovering]$
    $\land vLog' = [vLog$ EXCEPT $![r] = \langle\rangle]$
    $\land vEarlyBuffer' = [vEarlyBuffer$ EXCEPT $![r] = \{\}]$
    $\land vViewID' = [vViewID$ EXCEPT $![r] = 1]$
    $\land vLastNormView' = [vLastNormView$ EXCEPT $![r] = 1]$
    $\land vViewChanges' = [vViewChanges$ EXCEPT $![r] = \{\}]$
    $\land vSyncPoint' = [vSyncPoint$ EXCEPT $![r] = 0]$
    $\land vLateBuffer' = [vLateBuffer$ EXCEPT $![r] = \{\}]$
    $\land vTentativeSync' = [vTentativeSync$ EXCEPT $![r] = 0]$
    $\land vSyncReps' = [vSyncReps$ EXCEPT $![r] = \{\}]$
    $\land vCommitPoint' = [vCommitPoint$ EXCEPT $![r] = 0]$
    $\land vCrashVector' = [vCrashVector$ EXCEPT $![r] = [rr \in Replicas \mapsto 0]]$
    $\land vCrashVectorReps' = [vCrashVectorReps$ EXCEPT $![r] = \{\}]$
    $\land vRecoveryReps' = [vRecoveryReps$ EXCEPT $![r] = \{\}]$
    $\land$ UNCHANGED $\langle vReplicaClock, vUUIDCounter, clientVars, networkVars\rangle$
```

Recovering replica starts recovery (by first sending $CrashVectorReq$)

```
$StartReplicaRecovery(r) \triangleq$
    $\land vReplicaStatus[r] = StRecovering$
    $\land vUUIDCounter' = [vUUIDCounter$ EXCEPT $![r] = vUUIDCounter[r] + 1]$
    $\land Send(\{[mtype \mapsto MCrashVectorReq,$
               $sender \mapsto r,$
               $dest \mapsto d,$
```

```
              $nonce \mapsto vUUIDCounter'[r]] : d \in Replicas\})$
    $\land$ UNCHANGED $\langle vLog, vEarlyBuffer, vViewID, vReplicaClock,$
                      $vLastNormView, vViewChanges, vReplicaStatus,$
                      $vSyncPoint, vLateBuffer,$
                      $vTentativeSync, vSyncReps, vCommitPoint,$
                      $vCrashVector, vCrashVectorReps, vRecoveryReps,$
                      $clientVars \rangle$

$HandleCrashVectorReq(r, m) \triangleq$
    $\land vReplicaStatus[r] = StNormal$
    $\land Send(\{[mtype \mapsto MCrashVectorRep,$
                $sender \mapsto r,$
                $dest \mapsto m.sender,$
                $nonce \mapsto m.nonce,$
                $cv \mapsto vCrashVector[r]]\})$
    $\land$ UNCHANGED $\langle replicaVars, clientVars \rangle$

$HandleCrashVectorRep(r, m) \triangleq$
    $\land vReplicaStatus[r] = StRecovering$
    $\land vUUIDCounter[r] = m.nonce$
    $\land Cardinality(vCrashVectorReps[r]) \leq F$
    $\land \neg DuplicateRep(vCrashVectorReps[r], m)$
    $\land vCrashVectorReps' = [vCrashVectorReps$ EXCEPT $![r] = vCrashVectorReps[r] \cup \{m\}]$
    $\land vCrashVector' = [vCrashVector$ EXCEPT $![r] = MergeCrashVector(vCrashVector[r], m.cv)]$
    $\land$ IF $Cardinality(vCrashVectorReps') = F + 1$ THEN    got enough replies and can settle down $cv$
        $Send(\{[mtype \mapsto MRecoveryReq,$
                 $sender \mapsto r,$
                 $dest \mapsto d,$
                 $nonce \mapsto m.nonce,$
                 $cv \mapsto vCrashVector'[r]] : d \in Replicas\})$
        ELSE
        UNCHANGED $\langle networkVars \rangle$

    $\land$ UNCHANGED $\langle vLog, vEarlyBuffer, vViewID, vReplicaClock,$
                       $vLastNormView, vViewChanges, vReplicaStatus,$
                       $vSyncPoint, vLateBuffer,$
                       $vTentativeSync, vSyncReps, vCommitPoint,$
                       $vUUIDCounter, vRecoveryReps,$
                       $clientVars \rangle$

$HandleRecoveryReq(r, m) \triangleq$
    $\land vReplicaStatus[r] = StNormal$
    $\land vCrashVector' = [vCrashVector$ EXCEPT $![r] = MergeCrashVector(vCrashVector[r], m.cv)]$
    $\land Send(\{[\ mtype \mapsto MRecoveryRep,$
                 $sender \mapsto r,$
                 $dest \mapsto m.sender,$
                 $viewID \mapsto vViewID[r],$
                 $cv \mapsto vCrashVector'[r]] : d \in Replicas\})$

    $\land$ UNCHANGED $\langle vLog, vEarlyBuffer, vViewID, vReplicaClock,$
                      $vLastNormView, vViewChanges, vReplicaStatus,$
                      $vSyncPoint, vLateBuffer,$
                      $vTentativeSync, vSyncReps, vCommitPoint,$
                      $vUUIDCounter, vCrashVectorReps, vRecoveryReps,$
                      $clientVars\ \rangle$


$HandleRecoveryRep(r, m) \triangleq$
    $\land vReplicaStatus[r] = StRecovering$
    $\land Cardinality(vRecoveryReps[r]) \leq F$
    $\land \neg DuplicateRep(vRecoveryReps[r], m.sender)$
    $\land CheckCrashVector(m, r)$
Note: After crash vector is updated, those previously accepted messages may also become stray message.
Those messages should also be filtered out.
    $\land vRecoveryReps' = [vRecoveryReps$ EXCEPT
                            $![r] = FilterStrayMessage(vRecoveryReps[r] \cup \{m\}, vCrashVector'[r])\ ]$

    $\land$ IF $Cardinality(vRecoveryReps') = F + 1$ THEN     got enough replies
        LET
            $newView \triangleq Max(\{mm.viewID : mm \in vRecoveryReps'[r]\})$
            $leaderId \triangleq newView \% Cardinality(Replicas)$
        IN
            $Send(\{[mtype \mapsto MStateTransferReq,$
                    $sender \mapsto r,$
                    $dest \mapsto leaderId,$
                    $cv \mapsto vCrashVector'[r]] : d \in Replicas\})$
        ELSE
        UNCHANGED $\langle networkVars \rangle$

    $\land$ UNCHANGED $\langle vLog, vEarlyBuffer, vViewID, vReplicaClock,$
                     $vLastNormView, vViewChanges, vReplicaStatus,$
                     $vSyncPoint, vLateBuffer,$
                     $vTentativeSync, vSyncReps, vCommitPoint,$
                     $vUUIDCounter, vCrashVectorReps,$
                     $clientVars \rangle$


$HandleStateTransferReq(r, m) \triangleq$
    $\land vReplicaStatus[r] = StNormal$
        ∧ CheckCrashVector(m, r)
        ∧ Send({[ mtype  ↦ MStateTransferRep,
                  sender ↦ r,
                  dest   ↦ m.sender,
                  log    ↦ vLog[r],
                  sp     ↦ vSyncPoint[r],
                  cp     ↦ vCommitPoint[r],
                  cv     ↦ vCrashVector′[r]]})
        ∧ UNCHANGED  ⟨vLog, vEarlyBuffer, vViewID, vReplicaClock,
                      vLastNormView, vViewChanges, vReplicaStatus,
                      vSyncPoint, vLateBuffer,
                      vTentativeSync, vSyncReps, vCommitPoint,
                      vUUIDCounter, vCrashVectorReps, vRecoveryReps,
                      clientVars⟩

HandleStateTransferRep(r, m) ≜
        ∧ vReplicaStatus[r] = StRecovering
        ∧ CheckCrashVector(m, r)
        ∧ vLog′ = [vLog EXCEPT ![r] = m.log]
        ∧ vSyncPoint′ = [vSyncPoint EXCEPT ![r] = m.sp]
        ∧ vCommitPoint′ = [vCommitPoint EXCEPT ![r] = m.cp]
        ∧ vViewID′ = [vViewID EXCEPT  ![r] = m.viewID]
        ∧ vEarlyBuffer′ = [vEarlyBuffer EXCEPT ![r] = {}]
        ∧ vLastNormView′ = [vLastNormView EXCEPT ![r] = m.viewID]
        ∧ vViewChanges′ = [vViewChanges EXCEPT ![r] = {}]
        ∧ vReplicaStatus′ = [vReplicaStatus EXCEPT ![r] = StNormal]
        ∧ vLateBuffer′ = [vLateBuffer EXCEPT ![r] = {}]
        ∧ vTentativeSync′ = [vTentativeSync EXCEPT ![r] = m.sp]
        ∧ vSyncReps′ = [vSyncReps EXCEPT ![r] = {}]
        ∧ vCrashVectorReps′ = [vCrashVectorReps EXCEPT ![r] = {}]
        ∧ vRecoveryReps′ = [vRecoveryReps EXCEPT ![r] = {}]
        ∧ UNCHANGED  ⟨vReplicaClock, vUUIDCounter, clientVars⟩
```

Leader Change

Replica *r* starts a *Leader* change

```
StartLeaderChange(r) ≜
  ∧ Send({[mtype  ↦ MViewChangeReq,
           sender ↦ r,
           dest   ↦ d,
           viewID ↦ vViewID[r] + 1,
           cv     ↦ vCrashVector[r]] : d ∈ Replicas})
  ∧ UNCHANGED ⟨replicaVars, clientVars⟩
```

View Change Handlers

```
Replica r gets MViewChangeReq, m
HandleViewChangeReq(r, m) ≜
  LET
     currentViewID  ≜ vViewID[r]
     newViewID      ≜ Max({currentViewID, m.viewID})
     newLeaderNum   ≜ LeaderID(newViewID)
  IN
     Recovering replica does not participate in view change
   ∧ vReplicaStatus[r] ≠ StRecovering
   ∧ currentViewID    ≠ newViewID
   ∧ CheckCrashVector(m, r)
   ∧ vReplicaStatus' = [vReplicaStatus EXCEPT ![r] = StViewChange]
   ∧ vViewID'        = [vViewID EXCEPT ![r] = newViewID]
   ∧ vViewChanges' = [vViewChanges EXCEPT ![r] = {}]
   ∧ Send({[mtype        ↦ MViewChange,
             dest         ↦ Leader(newViewID),
             sender       ↦ r,
             viewID       ↦ newViewID,
             lastNormal   ↦ vLastNormView[r],
             syncedLog    ↦ SubSeq(vLog[r], 1, vSyncPoint[r]),
             unsyncedLog  ↦ SubSeq(vLog[r], vSyncPoint[r] + 1, Len(vLog[r])),
             cv           ↦ vCrashVector[r]]} ∪
             Send the MViewChangeReqs in case this is an entirely new view
            {[mtype  ↦ MViewChangeReq,
              sender ↦ r,
              dest   ↦ d,
              viewID ↦ newViewID,
              cv     ↦ vCrashVector[r]] : d ∈ Replicas})
   ∧ UNCHANGED ⟨clientVars, vLog, vEarlyBuffer, vReplicaClock,
                  vLastNormView, vSyncPoint, vLateBuffer,
                  vTentativeSync, vSyncReps, vCommitPoint,
                  vUUIDCounter, vCrashVectorReps, vRecoveryReps⟩

Replica r receives MViewChange, m
HandleViewChange(r, m) ≜
     Recovering replica does not participate in view change
   ∧ vReplicaStatus[r] ≠ StRecovering
     Add the message to the log
   ∧ vViewID[r]          = m.viewID
   ∧ vReplicaStatus[r]   = StViewChange
     This replica is the leader
   ∧ Leader(vViewID[r]) = r
   ∧ CheckCrashVector(m, r)
```

```
Note: Similar to $vRecoveryReps$, (potential) stray messages should be filtered out.
$\land\ vViewChanges' = [vViewChanges$ EXCEPT
                      $![r] = FilterStrayMessage(vViewChanges[r] \cup \{m\},\ vCrashVector'[r])]$
If there's enough replies, start the new view
$\land$ LET
    $isViewPromise(M) \triangleq \land \{n.sender : n \in M\} \in Quorums$
                         $\land \exists\, n \in M\ : n.sender = r$
    $vCMs \triangleq \{n \in vViewChanges'[r] :$
                 $\land\ n.mtype\ = MViewChange$
                 $\land\ n.viewID = vViewID[r]\}$
     Create the state for the new view
    $normalViews \triangleq \{n.lastNormal : n \in vCMs\}$
     Choose the largest normal view (i.e. the newest)
    $lastNormal \triangleq$ (CHOOSE $v \in normalViews : \forall\, v2 \in normalViews : v2 \leq v)$
     For logs before $vSyncPoint$ (i.e. $syncedLog$), we directly copy from the $bestCandiates$
     For $unsyncedLog$, we do quorum check to decide which ones should be added to recovery $Log$
    $goodCandidates \triangleq \{o \in vCMs : o.lastNormal = lastNormal\}$
     $bestCandidate$ can only be picked from $goodCandidates$,
     because previous views may include invalid logs
    $bestCandidate \triangleq$ CHOOSE $n \in goodCandidates :$
                        $\forall\, y \in goodCandidates : Len(n.syncedLog) \geq Len(y.syncedLog)$
    $unSyncedLogs \triangleq \{Seq2Set(n.unsyncedLog) : n \in goodCandidates\}$

  IN
    IF $isViewPromise(vCMs)$ THEN
      $Send(\{[mtype \mapsto MStartView,$
              $dest \mapsto d,$
              $viewID \mapsto vViewID[r],$
              $log \mapsto bestCandidate.syncedLog$
                       $\circ\ MergeUnSyncLogs(unSyncedLogs,\ LastLog(bestCandidate.syncedLog))$
             $] : d \in Replicas\})$
     ELSE
      UNCHANGED $networkVars$
$\land$ UNCHANGED $\langle clientVars,\ vLog,\ vEarlyBuffer,\ vViewID,\ vReplicaClock,$
                 $vLastNormView,\ vReplicaStatus,\ vSyncPoint,\ vLateBuffer,$
                 $vTentativeSync,\ vSyncReps,\ vCommitPoint,$
                 $vUUIDCounter,\ vCrashVectorReps,\ vRecoveryReps\rangle$


Replica $r$ receives a $MStartView$, $m$
$HandleStartView(r,\ m) \triangleq$
  $\land\ vReplicaStatus[r] \neq StRecovering$
  $\land\ \lor\ vViewID[r]\ < m.viewID$
     $\lor\ vViewID[r]\ = m.viewID \land vReplicaStatus[r] = StViewChange$
  $\land\ CheckCrashVector(m,\ r)$
  ∧ vLog′              = [vLog EXCEPT ![r] = m.log]
  ∧ vReplicaStatus′    = [vReplicaStatus EXCEPT ![r] = StNormal]
  ∧ vViewID′            = [vViewID EXCEPT ![r] = m.viewID]
  ∧ vLastNormView′ = [vLastNormView EXCEPT ![r] = m.viewID]
  ∧ vEarlyBuffer′ = [vEarlyBuffer EXCEPT ![r] = {}]  clear Early Buffer for the new view
  ∧ vLateBuffer′ = [vLateBuffer EXCEPT ![r] = {}]  clear Late Buffer for the new view
  ∧ vSyncPoint′ = [vSyncPoint EXCEPT ![r] = Len(m.log)]
  ∧ vTentativeSync′ = [vTentativeSync EXCEPT ![r] = Len(m.log)]
   Send replies (in the new view) for all log items
  ∧ IF r = Leader(m.viewID) THEN      Leader only sends fast reply
        Send({[ mtype      ↦ MFastReply,
                sender     ↦ r,
                dest       ↦ m.log[i].clientID,
                viewID     ↦ m.viewID,
                requestID  ↦ m.log[i].requestID,
                hash       ↦ [
                                 log ↦ SubSeq(m.log, 1, i),
                                 cv  ↦ vCrashVector
                              ],
                deadline   ↦ m.log[i].deadline,
                logSlotNum ↦ i] : i ∈ (1 .. Len(m.log))})
     ELSE   While staring view, followers knows the log is synced with the leader, so send slow-reply
        Send({[ mtype      ↦ MSlowReply,
                sender     ↦ r,
                dest       ↦ m.log[i].clientID,
                viewID     ↦ m.viewID,
                requestID  ↦ m.log[i].requestID,
                logSlotNum ↦ i] : i ∈ (1 .. Len(m.log))})
  ∧ UNCHANGED ⟨clientVars, vReplicaClock, vViewChanges,
                vSyncReps, vCommitPoint, vCrashVector,
                vUUIDCounter, vCrashVectorReps, vRecoveryReps⟩
```

Periodic Synchronization

Leader replica r conduct synchronization periodically

This periodic sync process is different from index sync process

It ensures that all replicas' logs are stable up to their *CommitPoint* (for fast recovery)

Our *CommitPoint* is essentially the *sync-point* defined in *NOPaxos* paper

Just as mentioned in *NOPaxos* paper, it is an optional optimization for fast recovery

Nezha still works even without this part

```
StartSync(r) ≜
  ∧ Leader(vViewID[r]) = r
  ∧ vReplicaStatus[r]     = StNormal
  ∧ vTentativeSync[r] < Len(vLog[r])   If ≥ then no need to sync
  ∧ vSyncReps′            = [vSyncReps EXCEPT ![r] = {}]
```

```
 ∧ vTentativeSync′    = [vTentativeSync EXCEPT ![r] = Len(vLog[r])]
 ∧ Send({[mtype       ↦ MSyncPrepare,
          sender      ↦ r,
          dest        ↦ d,
          viewID      ↦ vViewID[r],
          log         ↦ vLog[r]] : d ∈ Replicas})
 ∧ UNCHANGED ⟨clientVars, vLog, vEarlyBuffer, vViewID, vReplicaClock,
              vLastNormView, vViewChanges, vReplicaStatus,
              vSyncPoint, vLateBuffer, vCommitPoint,
              vUUIDCounter, vCrashVector,
              vCrashVectorReps, vRecoveryReps⟩


 Replica r receives MSyncPrepare, m
HandleSyncPrepare(r, m) ≜
 LET
   newLog ≜ m.log ∘ GetUnSyncLogs(vLog[r], LastLog(m.log))
 IN
 ∧ vReplicaStatus[r] = StNormal
 ∧ m.viewID            = vViewID[r]
 ∧ m.sender           = Leader(vViewID[r])
 ∧ IF     vSyncPoint[r]  < Len(m.log) THEN
          ∧ vSyncPoint′ = [vSyncPoint EXCEPT ![r] = Len(m.log)]
          ∧ vLog′         = [vLog EXCEPT ![r] = newLog]
          ∧ Send({[  mtype       ↦ MSlowReply,
                     sender      ↦ r,
                     dest        ↦ m.log[i].clientID,
                     viewID      ↦ m.viewID,
                     requestID   ↦ m.log[i].requestID,
                     logSlotNum ↦ i] : i ∈ (1 .. Len(m.log))})
   ELSE
          UNCHANGED ⟨vLog, vSyncPoint⟩
 ∧ Send({[mtype            ↦ MSyncRep,
          sender           ↦ r,
          dest             ↦ m.sender,
          viewID           ↦ vViewID[r],
          logSlotNumber ↦ Len(m.log)]}
        )
 ∧ UNCHANGED ⟨clientVars, vEarlyBuffer, vViewID,  vReplicaClock,
              vLastNormView, vViewChanges, vReplicaStatus,
              vLateBuffer, vTentativeSync, vSyncReps, vCommitPoint,
              vUUIDCounter, vCrashVector,
              vCrashVectorReps, vRecoveryReps⟩
```

```
Replica r receives MSyncRep, m
HandleSyncRep(r, m) ≜
  ∧ m.viewID          = vViewID[r]
  ∧ vReplicaStatus[r] = StNormal
  ∧ vSyncReps′        = [vSyncReps EXCEPT ![r] = vSyncReps[r] ∪ {m}]
  ∧ LET isViewPromise(M) ≜ ∧ {n.sender : n ∈ M} ∈ Quorums
                            ∧ ∃ n ∈ M  : n.sender = r
        sRMs             ≜ {n ∈ vSyncReps′[r] :
                              ∧ n.mtype         = MSyncRep
                              ∧ n.viewID        = vViewID[r]
                              ∧ n.logSlotNumber = vTentativeSync[r]}
        committedLog     ≜ IF vTentativeSync[r] ≥ 1 THEN
                             SubSeq(vLog[r], 1, vTentativeSync[r])
                            ELSE
                             ⟨⟩
     IN
      IF isViewPromise(sRMs) THEN
        ∧ Send({[mtype     ↦ MSyncCommit,
                 sender    ↦ r,
                 dest      ↦ d,
                 viewID    ↦ vViewID[r],
                 log       ↦ committedLog] :
                 d ∈ Replicas})
        ∧ vCommitPoint′ = [vCommitPoint EXCEPT ![r] = vTentativeSync[r]]
       ELSE
        UNCHANGED ⟨networkVars, vCommitPoint⟩
  ∧ UNCHANGED ⟨clientVars, vLog, vEarlyBuffer, vViewID,
                vReplicaClock, vLastNormView, vViewChanges,
                vReplicaStatus, vSyncPoint, vLateBuffer,
                vTentativeSync, vUUIDCounter, vCrashVector,
                vCrashVectorReps, vRecoveryReps⟩


Replica r receives MSyncCommit, m
HandleSyncCommit(r, m) ≜
  LET
    newLog ≜ m.log ∘ GetUnSyncLogs(vLog[r], LastLog(m.log))
  IN
  ∧ vReplicaStatus[r] = StNormal
  ∧ m.viewID          = vViewID[r]
  ∧ m.sender          = Leader(vViewID[r])
  ∧ IF  Len(m.log) ≤  vCommitPoint[r] THEN
        UNCHANGED ⟨vCommitPoint, vLog⟩
     ELSE
       ∧ vLog′        = [vLog EXCEPT ![r] = newLog]
```

```
        ∧ vCommitPoint' = [vCommitPoint EXCEPT ![r] = Len(m.log)]
        ∧ Send({[mtype       ↦ MSlowReply,
                 sender      ↦ r,
                 dest        ↦ m.log[i].clientID,
                 viewID      ↦ m.viewID,
                 requestID   ↦ m.log[i].requestID,
                 logSlotNum  ↦ i] : i ∈ (1 .. Len(m.log))})
    ∧ UNCHANGED ⟨networkVars, clientVars, vEarlyBuffer,
                 vViewID, vReplicaClock, vLastNormView, vViewChanges,
                 vReplicaStatus, vSyncPoint, vLateBuffer,
                 vTentativeSync, vSyncReps,
                 vUUIDCounter, vCrashVector,
                 vCrashVectorReps, vRecoveryReps⟩
```

Invariants and Helper Functions

A request/*log* is committed in two possible cases:
(1) A fast quorum has sent either slow-reply messages, or fast-reply messages with consistent hashes [Fast Path]
(2) A simple quorum has sent slow-reply messages [Slow Path] Both quorums should include the leader

Check whether *log* < *clientID*, *requestID* > is committed at position *logSlotNum*

```
Committed(clientID, requestID, logSlotNum) ≜
     Fast path
     ∨ ∃ M ∈ SUBSET ({m ∈ messages : ∧ ∨ m.mtype = MFastReply
                                        ∨ m.mtype = MSlowReply
                                      ∧ m.logSlotNum = logSlotNum
                                      ∧ m.dest = clientID
                                      ∧ m.requestID = requestID}) :
          Sent from a fast quorum
        ∧ {m.sender : m ∈ M} ∈ FastQuorums
          Matching view-id
        ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewID = m2.viewID
          One from the leader
        ∧ ∃ m ∈ M : m.sender = Leader(m.viewID)
          Hash values are consistent
        ∧  LET
                leaderReply ≜ CHOOSE m ∈ M : m.sender = Leader(m.viewID)
           IN
           ∀ m1 ∈ M : IF m1.mtype = MFastReply THEN
                          m1.hash = leaderReply.hash
                        ELSE
                          TRUE   SlowReply has consistent hash for sure
     Slow path
     ∨ ∃ M ∈ SUBSET ({m ∈ messages : ∧ ∨ m.mtype = MSlowReply
                                                ∨ ∧ m.mtype = MFastReply   Leader only sends fast-reply
                                                  ∧ m.sender = Leader(m.viewID)
                                              ∧ m.logSlotNum = logSlotNum
                                              ∧ m.dest = clientID
                                              ∧ m.requestID = requestID}) :
        ∧ {m.sender : m ∈ M} ∈ Quorums
         Matching view-id
        ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewID = m2.viewID
         One from the leader
        ∧ ∃ m ∈ M : m.sender = Leader(m.viewID)


 Check whether log < clientID, requestID > is committed in view viewID
CommittedInView(clientID, requestID, viewID) ≜
   Fast path
  ∨ ∃ M ∈ SUBSET ({m ∈ messages : ∧ ∨ m.mtype = MFastReply
                                    ∨ m.mtype = MSlowReply
                                  ∧ m.dest = clientID
                                  ∧ m.requestID = requestID
                                  ∧ m.viewID = viewID}) :
       Sent from a fast quorum
      ∧ {m.sender : m ∈ M} ∈ FastQuorums
       One from the leader
      ∧ ∃ m ∈ M : m.sender = Leader(m.viewID)
       Hash values are the same
      ∧  LET
              leaderReply ≜ CHOOSE m ∈ M : m.sender = Leader(m.viewID)
         IN
         ∀ m1 ∈ M : IF m1.mtype = MFastReply THEN
                       m1.hash = leaderReply.hash
                     ELSE
                       TRUE   SlowReply has consistent hash for sure
   Slow path
  ∨ ∃ M ∈ SUBSET ({m ∈ messages : ∧ ∨ m.mtype = MSlowReply
                                    ∨ ∧ m.mtype = MFastReply   Leader only sends fast-reply
                                      ∧ m.sender = Leader(m.viewID)
                                  ∧ m.dest = clientID
                                  ∧ m.requestID = requestID
                                  ∧ m.viewID = viewID}) :
      ∧ {m.sender : m ∈ M} ∈ Quorums
       Hash values are the same
      ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.hash = m2.hash
       One from the leader
      ∧ ∃ m ∈ M : m.sender = Leader(m.viewID)
```

```
$SystemRecovered(viewID) \triangleq$   $\land \exists RM \in$ SUBSET $(Replicas)$ :
                                           $\land Cardinality(RM) \geq QuorumSize$
                                           $\land \forall r \in RM : vLastNormView[r] \geq viewID$
                                           $\land \forall r \in RM : vReplicaStatus[r] = StNormal$  These replicas must be normal
                                        The leader of this view has also recovered or even goes beyond this view
                                       $\land$  $vLastNormView[Leader(viewID)] \geq viewID$
```

Invariants

Durability: *Committed* Requests always survive failure
i.e. If a request is committed in one view, then it will remain committed in the higher views
One thing to note, the check of ”committed“ only happens when the system is still ”normal“
While the system is under recovery (*i.e.* less than $f + 1$ replicas are normal),
the check of committed does not make sense

```
$Durability \triangleq \forall v1, v2 \in 1 \, .. \, MaxViews$ :
                  If a request is committed in lower view ($v1$,),
                  it is impossible to make this request uncommited in higher view ($v2$)
                    $\neg( \land v1 < v2$
                        To check Durability of request in higher views,
                        the system should have entered the higher views
                       $\land SystemRecovered(v2)$
                       $\land \exists c \in Clients$ :
                          $\exists r \in 1 \, .. \, MaxReqNum$ :
                              $\land CommittedInView(c, r, v1)$
                              $\land \neg CommittedInView(c, r, v2))$
```

Consistency: *Committed* requests have the same history even after view changes
i.e. If a request is committed in a lower view ($v1$), then (based on *Durability* Property)
it remains committed in higher view ($v2$)
Consistency requires the history of the request (*i.e.* all the request before this request) remain the same

```
$Consistency \triangleq$
    $\forall v1, v2 \in 1 \, .. \, MaxViews$ :
          $\neg( \land v1 < v2$
                To check Consistency of request in higher views,
                the system should have entered the higher views
               $\land SystemRecovered(v2)$
               $\land \exists c \in Clients$ :
                  $\exists r \in 1 \, .. \, MaxReqNum$ :
                  $\exists t \in 1 \, .. \, MaxTime$ :
                     Durability has been checked in another invariant
                    $\land CommittedInView(c, r, v1)$
                    $\land CommittedInView(c, r, v2)$
                    $\land$ LET
                          $v1LeaderReply \triangleq$ CHOOSE $m \in messages$ :
                                                  $\land m.mtype = MFastReply$
                                                  $\land m.deadline = t$
```

```
                                       ∧ m.dest = c
                                       ∧ m.requestID = r
                                       ∧ m.viewID = v1
                                       ∧ m.sender = Leader(v1)
                    v2LeaderReply ≜ CHOOSE m ∈ messages :
                                       ∧ m.mtype = MFastReply
                                       ∧ m.deadline = t
                                       ∧ m.dest = c
                                       ∧ m.requestID = r
                                       ∧ m.viewID = v2
                                       ∧ m.sender = Leader(v2)
                 IN
                    v1LeaderReply.hash ≠ v2LeaderReply.hash)
```

Linearizability: Only one request can be committed for a given position

i.e. If one request has committed at position *i*, then no contrary observation can be made

i.e. there cannot be a second request committed at the same position

```
Linearizability ≜
  LET
    maxLogPosition ≜ Max({1} ∪
      {m.logSlotNum : m ∈ {m ∈ messages :
                             ∨ m.mtype = MFastReply
                             ∨ m.mtype = MSlowReply}})
  IN  ¬(∃ c1, c2 ∈ Clients :
          ∃ r1, r2 ∈ 1 .. MaxReqNum :
            ∧ ⟨c1, r1⟩ ≠ ⟨c2, r2⟩
            ∧ ∃ i ∈ (1 .. maxLogPosition) :
               ∧ Committed(c1, r1, i)
               ∧ Committed(c2, r2, i)
        )
```

Main Transition Function

```
Next ≜   Handle Messages
    ∨ ∃ m ∈ messages :
                 ∧ m.mtype = MClientRequest
                 ∧ m ∉ vReplicaProcessed[m.dest]
                 ∧ HandleClientRequest(m.dest, m)
                 ∧ vReplicaProcessed′ =
                     [vReplicaProcessed EXCEPT ![m.dest] =
                     vReplicaProcessed[m.dest] ∪ {Msg2RLog(m, m.dest)}]
                 ∧ UNCHANGED vClientProcessed
                 ∧ DebugAction′ = ⟨"HandleClientRequest", m⟩

    ∨ ∃ m ∈ messages :
            $\land\ m.mtype = MViewChangeReq$
            $\land\ m \notin vReplicaProcessed[m.dest]$
            $\land\ HandleViewChangeReq(m.dest,\ m)$
            $\land\ vReplicaProcessed' =$
              $[vReplicaProcessed\ \textsc{except}\ ![m.dest] =$
              $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m,\ m.dest)\}]$
            $\land\ \textsc{unchanged}\ vClientProcessed$
            $\land\ DebugAction' = \langle \text{“HandleViewChangeReq”},\ m\rangle$

$\lor\ \exists\, m \in messages :$
            $\land\ m.mtype = MViewChange$
            $\land\ m \notin vReplicaProcessed[m.dest]$
            $\land\ HandleViewChange(m.dest,\ m)$
            $\land\ vReplicaProcessed' =$
              $[vReplicaProcessed\ \textsc{except}\ ![m.dest] =$
              $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m,\ m.dest)\}]$
            $\land\ \textsc{unchanged}\ vClientProcessed$
            $\land\ DebugAction' = \langle \text{“HandleViewChange”},\ m\rangle$

$\lor\ \exists\, m \in messages :$
            $\land\ m.mtype = MStartView$
            $\land\ m \notin vReplicaProcessed[m.dest]$
            $\land\ HandleStartView(m.dest,\ m)$
            $\land\ vReplicaProcessed' =$
              $[vReplicaProcessed\ \textsc{except}\ ![m.dest] =$
              $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m,\ m.dest)\}]$
            $\land\ \textsc{unchanged}\ vClientProcessed$
            $\land\ DebugAction' = \langle \text{“HandleStartView”},\ m\rangle$

$\lor\ \exists\, m \in messages :$
            $\land\ m.mtype = MSyncPrepare$
            $\land\ m \notin vReplicaProcessed[m.dest]$
            $\land\ HandleSyncPrepare(m.dest,\ m)$
            $\land\ vReplicaProcessed' =$
              $[vReplicaProcessed\ \textsc{except}\ ![m.dest] =$
                 $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m,\ m.dest)\}]$
            $\land\ \textsc{unchanged}\ vClientProcessed$
            $\land\ DebugAction' = \langle \text{“HandleSyncPrepare”},\ m\rangle$

$\lor\ \exists\, m \in messages :$
            $\land\ m.mtype = MSyncRep$
            $\land\ m \notin vReplicaProcessed[m.dest]$
            $\land\ HandleSyncRep(m.dest,\ m)$
            $\land\ vReplicaProcessed' =$
              $[vReplicaProcessed\ \textsc{except}\ ![m.dest] =$
              $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m,\ m.dest)\}]$
```

```
                    $\land$ UNCHANGED $vClientProcessed$
                    $\land$ $DebugAction' = \langle$ "HandleSyncRep", $m \rangle$
$\lor \exists\, m \in messages :$
                    $\land$ $m.mtype = MSyncCommit$
                    $\land$ $m \notin vReplicaProcessed[m.dest]$
                    $\land$ $HandleSyncCommit(m.dest, m)$
                    $\land$ $vReplicaProcessed' =$
                        $[vReplicaProcessed$ EXCEPT $![m.dest] =$
                        $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m, m.dest)\}]$
                    $\land$ UNCHANGED $vClientProcessed$
                    $\land$ $DebugAction' = \langle$ "HandleSyncCommit", $m \rangle$

$\lor \exists\, m \in messages :$
                      $\land$ $m.mtype = MMissEntryRequest$
                      $\land$ $m \notin vReplicaProcessed[m.dest]$
                      $\land$ $HandleMissEntryRequest(m.dest, m)$
                      $\land$ $vReplicaProcessed' =$
                          $[vReplicaProcessed$ EXCEPT $![m.dest] =$
                           $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m, m.dest)\}]$
                      $\land$ UNCHANGED $vClientProcessed$
                      $\land$ $DebugAction' = \langle$ "HandleMissEntryRequest", $m \rangle$

$\lor \exists\, m \in messages :$
                      $\land$ $m.mtype = MMissEntryReply$
                      $\land$ $m \notin vReplicaProcessed[m.dest]$
                      $\land$ $HandleMissEntryReply(m.dest, m)$
                      $\land$ $vReplicaProcessed' =$
                          $[vReplicaProcessed$ EXCEPT $![m.dest] =$
                           $vReplicaProcessed[m.dest] \cup \{Msg2RLog(m, m.dest)\}]$
                      $\land$ UNCHANGED $vClientProcessed$
                      $\land$ $DebugAction' = \langle$ "HandleMissEntryReply", $m \rangle$

Client Actions
$\lor \exists\, c \in Clients :$
                  $\land$ $vClientReqNum[c] < MaxReqNum$
                  $\land$ $ClientSendRequest(c)$
                  $\land$ UNCHANGED $\langle vReplicaProcessed, vClientProcessed \rangle$
                  $\land$ $DebugAction' = \langle$ "ClientSendRequest", "" $\rangle$

Start Synchronization
$\lor \exists\, r \in Replicas :$
                  $\land$ $StartSync(r)$
                  $\land$ UNCHANGED $\langle vReplicaProcessed, vClientProcessed \rangle$
                  $\land$ $DebugAction' = \langle$ "StartSync", "" $\rangle$

Replica Fail
```

```
$\lor \exists r \in Replicas :$
            $\land vReplicaStatus[r] = StNormal$
            $\land StartReplicaFail(r)$
            $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
            $\land DebugAction' = \langle \text{“StartReplicaFail”}, \text{“”} \rangle$

Leader Change
$\lor \exists r \in Replicas :$
            $\land vViewID[r] < MaxViews$
            $\land StartLeaderChange(r)$
            $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
            $\land DebugAction' = \langle \text{“StartLeaderChange”}, \text{“”} \rangle$

Replica Rejoin
$\lor \exists r \in Replicas :$
            $\land vReplicaStatus[r] = StRecovering$
            $\land StartReplicaRecovery(r)$
            $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
            $\land DebugAction' = \langle \text{“StartReplicaRecovery”}, \text{“”} \rangle$

Replica Actions:
$\lor \exists r \in Replicas :$
             $\land StartIndexSync(r)$
             $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
             $\land DebugAction' = \langle \text{“StartIndexSync”}, \text{“”} \rangle$

$\lor \exists r \in Replicas :$
        $\land FlushEarlyBuffer(r)$
        $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
        $\land DebugAction' = \langle \text{“FlushReplicaBuffer”}, \text{“”} \rangle$
Clock Move
$\lor \exists r \in Replicas :$
            $\land ReplicaClockMove(r)$
            $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
            $\land DebugAction' = \langle \text{“ReplicaClockMove”}, \text{“”} \rangle$

$\lor \exists c \in Clients  :$
            $\land ClientClockMove(c)$
            $\land \text{UNCHANGED } \langle vReplicaProcessed, vClientProcessed \rangle$
            $\land DebugAction' = \langle \text{“ClientClockMove”}, \text{“”} \rangle$
```