MODULE *ZabWithFLEAndSYNC*

This is the formal specification for the *Zab* consensus algorithm, which means *Zookeeper* Atomic *Broadcast*. The differences from *ZabWithFLE* is that we implement phase RECOVERY-SYNC.

Reference: *FLE*: *FastLeaderElection.java*, *Vote.java*, *QuorumPeer.java*, *e.g.* in https://*github.com*/*apache*/zookeeper.
ZAB: *QuorumPeer.java*, *Learner.java*, *Follower.java*, *LearnerHandler.java*, *Leader.java*, *e.g.* in https://*github.com*/*apache*/zookeeper. https://*cwiki.apache.org*/confluence/display/*ZOOKEEPER*/*Zab*1.0.

EXTENDS *FastLeaderElection*

The set of requests that can go into history
CONSTANT *Value*

Zab states
CONSTANTS *ELECTION*, *DISCOVERY*, *SYNCHRONIZATION*, *BROADCAST*

Sync modes & message types
CONSTANTS *DIFF*, *TRUNC*

Message types
CONSTANTS *FOLLOWERINFO*, *LEADERINFO*, *ACKEPOCH*, *NEWLEADER*, *ACKLD*, *UPTODATE*, *PROPOSAL*, *ACK*, *COMMIT*

NOTE: In production, there is no message type *ACKLD*. Server judges if counter of *ACK* is 0 to distinguish one *ACK* represents *ACKLD* or not. Here we divide *ACK* into *ACKLD* and *ACK*, to enhance readability of spec.

[*MaxTimeoutFailures*, *MaxTransactionNum*, *MaxEpoch*]
CONSTANT *Parameters*

TODO: Here we can add more constraints to decrease space, like restart, partition.

$MAXEPOCH \triangleq 10$

Variables in annotations mean variables defined in *FastLeaderElection*.
Variables that all servers use.

VARIABLES
- *zabState*, Current phase of server, in {*ELECTION*, *DISCOVERY*, *SYNCHRONIZATION*, *BROADCAST*}.
- *acceptedEpoch*, Epoch of the last *LEADERINFO* packet accepted, namely $f.p$ in paper.
- *lastCommitted*, Maximum index and *zxid* known to be committed, namely 'lastCommitted' in Leader. Starts from 0, and increases monotonically before restarting.
- *initialHistory* history that server initially has before election.
- state, * State of server, in {*LOOKING*, *FOLLOWING*, *LEADING*}.
- *currentEpoch*, * Epoch of the last *NEWLEADER* packet accepted, namely $f.a$ in paper.
- *lastProcessed*, * Index and *zxid* of the last processed *txn*.
- *history* * History of servers: sequence of transactions, containing: *zxid*, value, *ackSid*, epoch.

leader : [*committedRequests* + *toBeApplied*] [*outstandingProposals*]
follower: [*committedRequests*] [*pendingTxns*]

Variables only used for leader.

VARIABLES *learners*, Set of servers leader connects, namely 'learners' in Leader.

connecting, Set of learners leader has received *FOLLOWERINFO* from, namely 'connectingFollowers' in Leader.

electing, Set of learners leader has received *ACKEPOCH* from, namely 'electingFollowers' in Leader. Set of record [*sid*, *peerLastZxid*, *inQuorum*]. And $peerLastZxid = \langle -1, -1 \rangle$ means has done *syncFollower* with this *sid*. *inQuorum* = TRUE means in code it is one element in 'electingFollowers'.

ackldRecv, Set of learners leader has received *ACK* of *NEWLEADER* from, namely 'newLeaderProposal' in Leader.

forwarding, Set of learners that are synced with leader, namely 'forwardingFollowers' in Leader.

tempMaxEpoch ({*Maximum epoch in FOLLOWEINFO*} + 1) that leader has received from learners, namely 'epoch' in Leader.

leadingVoteSet \ ∗ Set of voters that follow leader.

Variables only used for follower.

VARIABLES *leaderAddr*, If follower has connected with leader. If follower lost connection, then null.

packetsSync packets of *PROPOSAL* and *COMMIT* from leader, namely 'packetsNotCommitted' and 'packetsCommitted' in *SyncWithLeader* in Learner.

Variables about network channel.

VARIABLE *msgs* Simulates network channel. *msgs*[*i*][*j*] means the input buffer of server *j* from server *i*.

electionMsgs \ ∗ Network channel in *FLE* module.

Variables only used in verifying properties.

VARIABLES *epochLeader*, Set of leaders in every epoch.

proposalMsgsLog, Set of all broadcast messages.

violatedInvariants Check whether there are conditions

contrary to the facts.

Variables only used for looking.

VARIABLE *currentVote*, * *Info* of current vote, namely 'currentVote'
* in *QuorumPeer*.
logicalClock, * Election instance, namely 'logicalClock'
* in *FastLeaderElection*.
receiveVotes, * Votes from current *FLE* round, namely
* 'recvset' in *FastLeaderElection*.
outOfElection, * Votes from previous and current *FLE* round,
* namely 'outofelection' in *FastLeaderElection*.
recvQueue, * Queue of received notifications or timeout
* signals.
waitNotmsg * Whether waiting for new *not.See* line 1050
* in *FastLeaderElection* for details.
VARIABLE *idTable* * For mapping *Server* to *Integers*,
to compare ids between servers.
Update: we have transformed *idTable* from variable to function.

Variable used for recording data to constrain state space.
VARIABLE *recorder* Consists: members of *Parameters* and *pc*.

$$serverVars \triangleq \langle state, currentEpoch, lastProcessed, zabState, acceptedEpoch, history, lastCommitted, initialHistory\rangle$$

$$electionVars \triangleq electionVarsL$$

$$leaderVars \triangleq \langle leadingVoteSet, learners, connecting, electing, ackldRecv, forwarding, tempMaxEpoch\rangle$$

$$followerVars \triangleq \langle leaderAddr, packetsSync\rangle$$

$$verifyVars \triangleq \langle proposalMsgsLog, epochLeader, violatedInvariants\rangle$$

$$msgVars \triangleq \langle msgs, electionMsgs\rangle$$

$$vars \triangleq \langle serverVars, electionVars, leaderVars, followerVars, verifyVars, msgVars, recorder\rangle$$

$$ServersIncNullPoint \triangleq Server \cup \{NullPoint\}$$

$$Zxid \triangleq Seq(Nat \cup \{-1\})$$

$$HistoryItem \triangleq [zxid : Zxid, value : Value, ackSid : \text{SUBSET } Server, epoch : Nat]$$

$Proposal \triangleq$
$[source : Server,$
$epoch : Nat,$
$zxid : Zxid,$
$data : Value]$

$LastItem \triangleq$
$[index : Nat,\ zxid : Zxid]$

$SyncPackets \triangleq$
$[notCommitted : Seq(HistoryItem),$
$committed : Seq(Zxid)]$

$Message \triangleq$
$[mtype : \{FOLLOWERINFO\},\ mzxid : Zxid] \cup$
$[mtype : \{LEADERINFO\},\ mzxid : Zxid] \cup$
$[mtype : \{ACKEPOCH\},\ mzxid : Zxid,\ mepoch : Nat \cup \{-1\}] \cup$
$[mtype : \{DIFF\},\ mzxid : Zxid] \cup$
$[mtype : \{TRUNC\},\ mtruncZxid : Zxid] \cup$
$[mtype : \{PROPOSAL\},\ mzxid : Zxid,\ mdata : Value] \cup$
$[mtype : \{COMMIT\},\ mzxid : Zxid] \cup$
$[mtype : \{NEWLEADER\},\ mzxid : Zxid] \cup$
$[mtype : \{ACKLD\},\ mzxid : Zxid] \cup$
$[mtype : \{ACK\},\ mzxid : Zxid] \cup$
$[mtype : \{UPTODATE\},\ mzxid : Zxid]$

$ElectionState \triangleq \{LOOKING,\ FOLLOWING,\ LEADING\}$

$ZabState \triangleq \{ELECTION,\ DISCOVERY,\ SYNCHRONIZATION,\ BROADCAST\}$

$ViolationSet \triangleq \{$"stateInconsistent", "proposalInconsistent",
"commitInconsistent", "ackInconsistent",
"messageIllegal"$\}$

$Electing \triangleq [sid : Server,$
$peerLastZxid : Zxid,$
$inQuorum : \text{BOOLEAN}\ ]$

$Vote \triangleq$
$[proposedLeader : ServersIncNullPoint,$
$proposedZxid : Zxid,$
$proposedEpoch : Nat]$

$ElectionVote \triangleq$
$[vote : Vote,\ round : Nat,\ state : ElectionState,\ version : Nat]$

$ElectionMsg \triangleq$
$[mtype : \{NOTIFICATION\},$

$msource : Server,$
$mstate \ : ElectionState,$
$mround : Nat,$
$mvote : Vote] \cup$
$[mtype : \{NONE\}]$

$TypeOK \triangleq$
$\land \quad zabState \in [Server \to ZabState]$
$\land \quad acceptedEpoch \in [Server \to Nat]$
$\land \quad lastCommitted \in [Server \to LastItem]$
$\land \quad learners \in [Server \to \text{SUBSET } Server]$
$\land \quad connecting \in [Server \to \text{SUBSET } ServersIncNullPoint]$
$\land \quad electing \in [Server \to \text{SUBSET } Electing]$
$\land \quad ackldRecv \in [Server \to \text{SUBSET } ServersIncNullPoint]$
$\land \quad forwarding \in [Server \to \text{SUBSET } Server]$
$\land \quad initialHistory \in [Server \to Seq(HistoryItem)]$
$\land \quad tempMaxEpoch \in [Server \to Nat]$
$\land \quad leaderAddr \in [Server \to ServersIncNullPoint]$
$\land \quad packetsSync \in [Server \to SyncPackets]$
$\land \quad proposalMsgsLog \in \text{SUBSET } Proposal$
$\land \quad epochLeader \in [1 \,..\, MAXEPOCH \to \text{SUBSET } Server]$
$\land \quad violatedInvariants \in [ViolationSet \to \text{BOOLEAN }]$
$\land \quad msgs \in [Server \to [Server \to Seq(Message)]]$
Fast Leader Election
$\land\ electionMsgs \in [Server \to [Server \to Seq(ElectionMsg)]]$
$\land\ recvQueue \in [Server \to Seq(ElectionMsg)]$
$\land\ leadingVoteSet \in [Server \to \text{SUBSET } Server]$
$\land\ receiveVotes \in [Server \to [Server \to ElectionVote]]$
$\land\ currentVote \in [Server \to Vote]$
$\land\ outOfElection \in [Server \to [Server \to ElectionVote]]$
$\land\ lastProcessed \ \in [Server \to LastItem]$
$\land\ history \in [Server \to Seq(HistoryItem)]$
$\land\ state \in [Server \to ElectionState]$
$\land\ waitNotmsg \in [Server \to \text{BOOLEAN }]$
$\land\ currentEpoch \in [Server \to Nat]$
$\land\ logicalClock \ \in [Server \to Nat]$

Return the maximum value from the set S
$Maximum(S) \triangleq \text{IF } S = \{\} \text{ THEN } -1$
$\text{ELSE CHOOSE } n \in S : \forall\, m \in S : n \geq m$

Return the minimum value from the set S
$Minimum(S) \triangleq \text{IF } S = \{\} \text{ THEN } -1$
$\text{ELSE CHOOSE } n \in S : \forall\, m \in S : n \leq m$

Check server state

$IsLeader(s) \triangleq state[s] = LEADING$
$IsFollower(s) \triangleq state[s] = FOLLOWING$
$IsLooking(s) \triangleq state[s] = LOOKING$

$IsMyLearner(i, j) \triangleq j \in learners[i]$
$IsMyLeader(i, j) \triangleq leaderAddr[i] = j$
$HasNoLeader(i) \triangleq leaderAddr[i] = NullPoint$
$HasLeader(i) \triangleq leaderAddr[i] \neq NullPoint$
$MyVote(i) \triangleq currentVote[i].proposedLeader$

Check if s is a quorum
$IsQuorum(s) \triangleq s \in Quorums$

Check *zxid* state
$ToZxid(z) \triangleq [epoch \mapsto z[1], counter \mapsto z[2]]$

$TxnZxidEqual(txn, z) \triangleq txn.zxid[1] = z[1] \land txn.zxid[2] = z[2]$

$TxnEqual(txn1, txn2) \triangleq ZxidEqual(txn1.zxid, txn2.zxid)$

$EpochPrecedeInTxn(txn1, txn2) \triangleq txn1.zxid[1] < txn2.zxid[1]$

Actions about recorder
$GetParameter(p) \triangleq$ IF $p \in$ DOMAIN $Parameters$ THEN $Parameters[p]$ ELSE 0

$RecorderGetHelper(m) \triangleq (m :> recorder[m])$
$RecorderIncHelper(m) \triangleq (m :> recorder[m] + 1)$

$RecorderIncTimeout \triangleq RecorderIncHelper(\text{“nTimeout”})$
$RecorderGetTimeout \triangleq RecorderGetHelper(\text{“nTimeout”})$
$RecorderSetTransactionNum(pc) \triangleq$ (“nTransaction” :>
IF $pc[1] =$ “LeaderProcessRequest” THEN
LET $s \triangleq$ CHOOSE $i \in Server$:
$\forall j \in Server : Len(history'[i]) \geq Len(history'[j])$
IN $Len(history'[s])$
ELSE $recorder[\text{“nTransaction”}]$)
$RecorderSetMaxEpoch(pc) \triangleq$ (“maxEpoch” :>
IF $pc[1] =$ “LeaderProcessFOLLOWERINFO” THEN
LET $s \triangleq$ CHOOSE $i \in Server$:
$\forall j \in Server : acceptedEpoch'[i] \geq acceptedEpoch'[j]$
IN $acceptedEpoch'[s]$
ELSE $recorder[\text{“maxEpoch”}]$)
$RecorderSetPc(pc) \triangleq (\text{“pc”} :> pc)$
$RecorderSetFailure(pc) \triangleq$ CASE $pc[1] =$ “Timeout” $\rightarrow RecorderIncTimeout$
$\square$ $pc[1] =$ “LeaderTimeout” $\rightarrow RecorderIncTimeout$
$\square$ $pc[1] =$ “FollowerTimeout” $\rightarrow RecorderIncTimeout$
$\square$ OTHER $\rightarrow RecorderGetTimeout$

$UpdateRecorder(pc) \triangleq recorder' = RecorderSetFailure(pc)$ @@ $RecorderSetTransactionNum(pc)$
@@ $RecorderSetMaxEpoch(pc)$ @@ $RecorderSetPc(pc)$ @@ $recorder$

$UnchangeRecorder \triangleq$ UNCHANGED $recorder$

$CheckParameterHelper(n, p, Comp(\_, \_)) \triangleq$ IF $p \in$ DOMAIN $Parameters$
THEN $Comp(n, Parameters[p])$
ELSE TRUE

$CheckParameterLimit(n, p) \triangleq CheckParameterHelper(n, p,$ LAMBDA $i, j : i < j)$

$CheckTimeout \triangleq CheckParameterLimit(recorder.nTimeout,$ "MaxTimeoutFailures")
$CheckTransactionNum \triangleq CheckParameterLimit(recorder.nTransaction,$ "MaxTransactionNum")
$CheckEpoch \triangleq CheckParameterLimit(recorder.maxEpoch,$ "MaxEpoch")

$CheckStateConstraints \triangleq CheckTimeout \land CheckTransactionNum \land CheckEpoch$

Actions about network

$PendingFOLLOWERINFO(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = FOLLOWERINFO$

$PendingLEADERINFO(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = LEADERINFO$

$PendingACKEPOCH(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = ACKEPOCH$

$PendingNEWLEADER(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = NEWLEADER$

$PendingACKLD(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = ACKLD$

$PendingUPTODATE(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = UPTODATE$

$PendingPROPOSAL(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = PROPOSAL$

$PendingACK(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = ACK$

$PendingCOMMIT(i, j) \triangleq \land msgs[j][i] \neq \langle\rangle$
$\land msgs[j][i][1].mtype = COMMIT$

Add a message to $msgs$ − add a message m to $msgs$.

$Send(i, j, m) \triangleq msgs' = [msgs$ EXCEPT $![i][j] = Append(msgs[i][j], m)]$
$SendPackets(i, j, ms) \triangleq msgs' = [msgs$ EXCEPT $![i][j] = msgs[i][j] \circ ms]$
$DiscardAndSendPackets(i, j, ms) \triangleq msgs' = [msgs$ EXCEPT $![j][i] = Tail(msgs[j][i]),$
$![i][j] = msgs[i][j] \circ ms]$

Remove a message from $msgs$ − discard head of $msgs$.

$Discard(i, j) \triangleq msgs' =$ IF $msgs[i][j] \neq \langle\rangle$ THEN $[msgs$ EXCEPT $![i][j] = Tail(msgs[i][j])]$
ELSE $msgs$

Leader broadcasts a $message(PROPOSAL/COMMIT)$ to all other servers in $forwardingFollowers$.

$Broadcast(i, m) \triangleq msgs' = [msgs$ EXCEPT $![i] = [v \in Server \mapsto$ IF $\land v \in forwarding[i]$
$\land v \neq i$
THEN $Append(msgs[i][v], m)$

```
                                                              ELSE  msgs[i][v]]]
DiscardAndBroadcast(i, j, m) ≜
        msgs' = [msgs EXCEPT ![j][i] = Tail(msgs[j][i]),
                             ![i] = [v ∈ Server ↦ IF  ∧ v ∈ forwarding[i]
                                                      ∧ v ≠ i
                                                    THEN Append(msgs[i][v], m)
                                                    ELSE msgs[i][v]]]
 Leader broadcasts LEADERINFO to all other servers in connectingFollowers.
DiscardAndBroadcastLEADERINFO(i, j, m) ≜
        msgs' = [msgs EXCEPT ![j][i] = Tail(msgs[j][i]),
                             ![i] = [v ∈ Server ↦ IF  ∧ v ∈ connecting'[i]
                                                      ∧ v ∈ learners[i]
                                                      ∧ v ≠ i
                                                    THEN Append(msgs[i][v], m)
                                                    ELSE msgs[i][v]]]
 Leader broadcasts UPTODATE to all other servers in newLeaderProposal.
DiscardAndBroadcastUPTODATE(i, j, m) ≜
        msgs' = [msgs EXCEPT ![j][i] = Tail(msgs[j][i]),
                             ![i] = [v ∈ Server ↦ IF  ∧ v ∈ ackldRecv'[i]
                                                      ∧ v ∈ learners[i]
                                                      ∧ v ≠ i
                                                    THEN Append(msgs[i][v], m)
                                                    ELSE msgs[i][v]]]
 Combination of Send and Discard − discard head of msgs[j][i] and add m into msgs.
Reply(i, j, m) ≜  msgs' = [msgs EXCEPT ![j][i] = Tail(msgs[j][i]),
                                       ![i][j] = Append(msgs[i][j], m)]

 Shuffle input buffer.
Clean(i, j) ≜  msgs' = [msgs EXCEPT ![j][i] = ⟨⟩, ![i][j] = ⟨⟩]
CleanInputBuffer(i) ≜  msgs' = [s ∈ Server ↦ [v ∈ Server ↦ IF v = i THEN ⟨⟩
                                                            ELSE msgs[s][v]]]
CleanInputBufferInCluster(S) ≜  msgs' = [s ∈ Server ↦
                                          [v ∈ Server ↦ IF v ∈ S THEN ⟨⟩
                                                        ELSE msgs[s][v]]]
```

```
 Define initial values for all variables
InitServerVars ≜  ∧ InitServerVarsL
                  ∧ zabState       = [s ∈ Server ↦ ELECTION]
                  ∧ acceptedEpoch  = [s ∈ Server ↦ 0]
                  ∧ lastCommitted  = [s ∈ Server ↦ [index ↦ 0,
                                                    zxid  ↦ ⟨0, 0⟩]]
                  ∧ initialHistory = [s ∈ Server ↦ ⟨⟩]

InitLeaderVars ≜  ∧ InitLeaderVarsL
                  ∧ learners       = [s ∈ Server ↦ {}]
```

$$
\begin{array}{ll}
\land\ connecting & = [s \in Server \mapsto \{\}] \\
\land\ electing & = [s \in Server \mapsto \{\}] \\
\land\ ackldRecv & = [s \in Server \mapsto \{\}] \\
\land\ forwarding & = [s \in Server \mapsto \{\}] \\
\land\ tempMaxEpoch & = [s \in Server \mapsto 0]
\end{array}
$$

$$InitElectionVars \triangleq InitElectionVarsL$$

$$
\begin{array}{ll}
InitFollowerVars \triangleq & \land\ leaderAddr = [s \in Server \mapsto NullPoint] \\
& \land\ packetsSync = [s \in Server \mapsto \\
& \qquad [notCommitted \mapsto \langle\rangle, \\
& \qquad\ committed \mapsto \langle\rangle]]
\end{array}
$$

$$
\begin{array}{ll}
InitVerifyVars \triangleq & \land\ proposalMsgsLog = \{\} \\
& \land\ epochLeader = [i \in 1\,..\,MAXEPOCH \mapsto \{\}] \\
& \land\ violatedInvariants = [stateInconsistent \mapsto \text{FALSE}, \\
& \qquad proposalInconsistent \mapsto \text{FALSE}, \\
& \qquad commitInconsistent \mapsto \text{FALSE}, \\
& \qquad ackInconsistent \mapsto \text{FALSE}, \\
& \qquad messageIllegal \mapsto \text{FALSE}]
\end{array}
$$

$$
\begin{array}{ll}
InitMsgVars \triangleq & \land\ msgs = [s \in Server \mapsto [v \in Server \mapsto \langle\rangle]] \\
& \land\ electionMsgs = [s \in Server \mapsto [v \in Server \mapsto \langle\rangle]]
\end{array}
$$

$$
\begin{array}{ll}
InitRecorder \triangleq & recorder = [nTimeout \mapsto 0, \\
& \qquad nTransaction \mapsto 0, \\
& \qquad maxEpoch \mapsto 0, \\
& \qquad pc \mapsto \langle \text{“Init”} \rangle]
\end{array}
$$

$$
\begin{array}{ll}
Init \triangleq & \land\ InitServerVars \\
& \land\ InitLeaderVars \\
& \land\ InitElectionVars \\
& \land\ InitFollowerVars \\
& \land\ InitVerifyVars \\
& \land\ InitMsgVars \\
& \land\ InitRecorder
\end{array}
$$

$$
\begin{array}{ll}
ZabTurnToLeading(i) \triangleq & \\
\quad \land\ zabState' & = [zabState \ \text{EXCEPT}\ ![i] = DISCOVERY] \\
\quad \land\ learners' & = [learners \ \text{EXCEPT}\ ![i] = \{i\}] \\
\quad \land\ connecting' & = [connecting \ \text{EXCEPT}\ ![i] = \{i\}] \\
\quad \land\ electing' & = [electing \ \text{EXCEPT}\ ![i] = \{[sid \mapsto i, \\
& \qquad\qquad\qquad\qquad peerLastZxid \mapsto \langle -1, -1\rangle, \\
& \qquad\qquad\qquad\qquad inQuorum \mapsto \text{TRUE}]\}] \\
\quad \land\ ackldRecv' & = [ackldRecv \ \text{EXCEPT}\ ![i] = \{i\}] \\
\quad \land\ forwarding' & = [forwarding \ \text{EXCEPT}\ ![i] = \{\}] \\
\quad \land\ initialHistory' & = [initialHistory \ \text{EXCEPT}\ ![i] = history'[i]]
\end{array}
$$

$\land\ tempMaxEpoch' = [tempMaxEpoch \quad \text{EXCEPT}\ ![i] = acceptedEpoch[i] + 1]$

$ZabTurnToFollowing(i) \triangleq$
$\land\ zabState' = [zabState\ \text{EXCEPT}\ ![i] = DISCOVERY]$
$\land\ initialHistory' = [initialHistory\ \text{EXCEPT}\ ![i] = history'[i]]$
$\land\ packetsSync' = [packetsSync\ \text{EXCEPT}\ ![i].notCommitted = \langle\rangle,$
$\qquad ![i].committed = \langle\rangle]$

Fast Leader Election
$FLEReceiveNotmsg(i, j) \triangleq$
$\land\ ReceiveNotmsg(i, j)$
$\land\ \text{UNCHANGED}\ \langle zabState, acceptedEpoch, lastCommitted, learners, connecting,$
$\qquad initialHistory, electing, ackldRecv, forwarding, tempMaxEpoch,$
$\qquad followerVars, verifyVars, msgs\rangle$
$\land\ UpdateRecorder(\langle \text{“FLEReceiveNotmsg”}, i, j\rangle)$

$FLENotmsgTimeout(i) \triangleq$
$\land\ NotmsgTimeout(i)$
$\land\ \text{UNCHANGED}\ \langle zabState, acceptedEpoch, lastCommitted, learners, connecting,$
$\qquad initialHistory, electing, ackldRecv, forwarding, tempMaxEpoch,$
$\qquad followerVars, verifyVars, msgs\rangle$
$\land\ UpdateRecorder(\langle \text{“FLENotmsgTimeout”}, i\rangle)$

$FLEHandleNotmsg(i) \triangleq$
$\land\ HandleNotmsg(i)$
$\land\ \text{LET}\ newState \triangleq state'[i]$
$\quad \text{IN}$
$\quad \lor\ \land\ newState = LEADING$
$\qquad \land\ ZabTurnToLeading(i)$
$\qquad \land\ \text{UNCHANGED}\ packetsSync$
$\quad \lor\ \land\ newState = FOLLOWING$
$\qquad \land\ ZabTurnToFollowing(i)$
$\qquad \land\ \text{UNCHANGED}\ \langle learners, connecting, electing, ackldRecv,$
$\qquad\qquad forwarding, tempMaxEpoch\rangle$
$\quad \lor\ \land\ newState = LOOKING$
$\qquad \land\ \text{UNCHANGED}\ \langle zabState, learners, connecting, electing, ackldRecv,$
$\qquad\qquad forwarding, tempMaxEpoch, packetsSync, initialHistory\rangle$
$\land\ \text{UNCHANGED}\ \langle lastCommitted, acceptedEpoch, leaderAddr, verifyVars, msgs\rangle$
$\land\ UpdateRecorder(\langle \text{“FLEHandleNotmsg”}, i\rangle)$

On the premise that $ReceiveVotes.HasQuorums = \text{TRUE}$,
corresponding to logic in line $1050 - 1055$ in $FastLeaderElection$.
$FLEWaitNewNotmsg(i) \triangleq$
$\land\ WaitNewNotmsg(i)$
$\land\ \text{UNCHANGED}\ \langle zabState, acceptedEpoch, lastCommitted, learners, connecting,$
$\qquad electing, ackldRecv, forwarding, tempMaxEpoch, initialHistory,$

$followerVars,\ verifyVars,\ msgs\rangle$
$\land\ UpdateRecorder(\langle$"FLEWaitNewNotmsg", $i\rangle)$

On the premise that $ReceiveVotes.HasQuorums = \text{TRUE}$,
corresponding to logic in line $1061 - 1066$ in *FastLeaderElection*.

$FLEWaitNewNotmsgEnd(i) \triangleq$
$\land\ WaitNewNotmsgEnd(i)$
$\land$ LET $newState \triangleq state'[i]$
IN
$\lor\ \land\ newState = LEADING$
$\land\ ZabTurnToLeading(i)$
$\land$ UNCHANGED $packetsSync$
$\lor\ \land\ newState = FOLLOWING$
$\land\ ZabTurnToFollowing(i)$
$\land$ UNCHANGED $\langle learners,\ connecting,\ electing,\ ackldRecv,\ forwarding,$
$tempMaxEpoch\rangle$
$\lor\ \land\ newState = LOOKING$
$\land\ PrintT($"Note: New state is LOOKING in FLEWaitNewNotmsgEnd," $\circ$
" which should not happen.")
$\land$ UNCHANGED $\langle zabState,\ learners,\ connecting,\ electing,\ ackldRecv,$
$forwarding,\ tempMaxEpoch,\ initialHistory,\ packetsSync\rangle$
$\land$ UNCHANGED $\langle lastCommitted,\ acceptedEpoch,\ leaderAddr,\ verifyVars,\ msgs\rangle$
$\land\ UpdateRecorder(\langle$"FLEWaitNewNotmsgEnd", $i\rangle)$

$InitialVotes \triangleq [vote \mapsto InitialVote,$
$round \mapsto 0,$
$state \mapsto LOOKING,$
$version \mapsto 0]$

Equals to for every server in S, performing action *ZabTimeout*.

$ZabTimeoutInCluster(S) \triangleq$
$\land\ state' = [s \in Server \mapsto$ IF $s \in S$ THEN $LOOKING$ ELSE $state[s]]$
$\land\ lastProcessed' = [s \in Server \mapsto$ IF $s \in S$ THEN $InitLastProcessed(s)$
ELSE $lastProcessed[s]]$
$\land\ logicalClock' = [s \in Server \mapsto$ IF $s \in S$ THEN $logicalClock[s] + 1$
ELSE $logicalClock[s]]$
$\land\ currentVote' = [s \in Server \mapsto$ IF $s \in S$ THEN
$[proposedLeader \mapsto s,$
$proposedZxid \mapsto lastProcessed'[s].zxid,$
$proposedEpoch \mapsto currentEpoch[s]]$
ELSE $currentVote[s]]$
$\land\ receiveVotes' = [s \in Server \mapsto$ IF $s \in S$ THEN $[v \in Server \mapsto InitialVotes]$
ELSE $receiveVotes[s]]$
$\land\ outOfElection' = [s \in Server \mapsto$ IF $s \in S$ THEN $[v \in Server \mapsto InitialVotes]$
ELSE $outOfElection[s]]$

$\land$ $recvQueue' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } \langle [mtype \mapsto NONE] \rangle$
ELSE $recvQueue[s]]$
$\land$ $waitNotmsg' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN FALSE ELSE } waitNotmsg[s]]$
$\land$ $leadingVoteSet' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } \{\} \text{ ELSE } leadingVoteSet[s]]$
$\land$ UNCHANGED $\langle electionMsgs, currentEpoch, history \rangle$
$\land$ $zabState' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } ELECTION \text{ ELSE } zabState[s]]$
$\land$ $leaderAddr' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } NullPoint \text{ ELSE } leaderAddr[s]]$
$\land$ $CleanInputBufferInCluster(S)$

Describe how a server transitions from *LEADING/FOLLOWING* to *LOOKING*.

$FollowerShutdown(i) \triangleq$
$\land$ $ZabTimeout(i)$
$\land$ $zabState' = [zabState \text{ EXCEPT } ![i] = ELECTION]$
$\land$ $leaderAddr' = [leaderAddr \text{ EXCEPT } ![i] = NullPoint]$
$\land$ $CleanInputBuffer(i)$

$LeaderShutdown(i) \triangleq$
$\land$ LET $cluster \triangleq \{i\} \cup learners[i]$
IN $ZabTimeoutInCluster(cluster)$
$\land$ $learners' = [learners \text{ EXCEPT } ![i] = \{\}]$
$\land$ $forwarding' = [forwarding \text{ EXCEPT } ![i] = \{\}]$

$RemoveElecting(set, sid) \triangleq$
LET $sid\_electing \triangleq \{s.sid : s \in set\}$
IN IF $sid \notin sid\_electing$ THEN set
ELSE LET $info \triangleq \text{CHOOSE } s \in set : s.sid = sid$
$new\_info \triangleq [sid \mapsto sid,$
$peerLastZxid \mapsto \langle -1, -1 \rangle,$
$inQuorum \mapsto info.inQuorum]$
IN $(set \setminus \{info\}) \cup \{new\_info\}$

See *removeLearnerHandler* for details.

$RemoveLearner(i, j) \triangleq$
$\land$ $learners' = [learners \text{ EXCEPT } ![i] = @ \setminus \{j\}]$
$\land$ $forwarding' = [forwarding \text{ EXCEPT } ![i] = \text{IF } j \in forwarding[i]$
THEN $@ \setminus \{j\}$ ELSE $@]$
$\land$ $electing' = [electing \text{ EXCEPT } ![i] = RemoveElecting(@, j)]$

Follower connecting to leader fails and truns to *LOOKING*.

$FollowerTimeout(i) \triangleq$
$\land$ $CheckTimeout$ test restrictions of *timeout_1*
$\land$ $IsFollower(i)$
$\land$ $HasNoLeader(i)$
$\land$ $FollowerShutdown(i)$
$\land$ $CleanInputBuffer(i)$
$\land$ UNCHANGED $\langle acceptedEpoch, lastCommitted, learners, connecting, electing,$

$ackldRecv$, $forwarding$, $tempMaxEpoch$, $initialHistory$,
$verifyVars$, $packetsSync\rangle$
$\land$ $UpdateRecorder(\langle$"FollowerTimeout", $i\rangle)$

Leader loses support from a quorum and turns to $LOOKING$.
$LeaderTimeout(i) \triangleq$
$\land$ $CheckTimeout$ test restrictions of $timeout\_2$
$\land$ $IsLeader(i)$
$\land$ $\neg IsQuorum(learners[i])$
$\land$ $LeaderShutdown(i)$
$\land$ UNCHANGED $\langle acceptedEpoch, lastCommitted, connecting, electing, ackldRecv,$
$tempMaxEpoch, initialHistory, verifyVars, packetsSync\rangle$
$\land$ $UpdateRecorder(\langle$"LeaderTimeout", $i\rangle)$

Timeout between leader and follower.
$Timeout(i, j) \triangleq$
$\land$ $CheckTimeout$ test restrictions of $timeout\_3$
$\land$ $IsLeader(i)$ $\land$ $IsMyLearner(i, j)$
$\land$ $IsFollower(j)$ $\land$ $IsMyLeader(j, i)$
The action of leader i.
$\land$ $RemoveLearner(i, j)$
The action of follower j.
$\land$ $FollowerShutdown(j)$
$\land$ $Clean(i, j)$
$\land$ UNCHANGED $\langle acceptedEpoch, lastCommitted, connecting, ackldRecv,$
$tempMaxEpoch, initialHistory, verifyVars, packetsSync\rangle$
$\land$ $UpdateRecorder(\langle$"Timeout", $i, j\rangle)$

Establish connection between leader and follower, containing actions like $addLearnerHandler$, $findLeader$, $connectToLeader$.
$ConnectAndFollowerSendFOLLOWERINFO(i, j) \triangleq$
$\land$ $IsLeader(i) \land \neg IsMyLearner(i, j)$
$\land$ $IsFollower(j) \land HasNoLeader(j) \land MyVote(j) = i$
$\land$ $learners' = [learners$ EXCEPT $![i] = learners[i] \cup \{j\}]$
$\land$ $leaderAddr' = [leaderAddr$ EXCEPT $![j] = i]$
$\land$ $Send(j, leaderAddr'[j], [mtype \mapsto FOLLOWERINFO,$
$mzxid \mapsto \langle acceptedEpoch[j], 0\rangle])$
$\land$ UNCHANGED $\langle serverVars, electionVars, leadingVoteSet, connecting,$
$electing, ackldRecv, forwarding, tempMaxEpoch,$
$verifyVars, electionMsgs, packetsSync\rangle$
$\land$ $UpdateRecorder(\langle$"ConnectAndFollowerSendFOLLOWERINFO", $i, j\rangle)$

$waitingForNewEpoch$ in Leader
$WaitingForNewEpoch(i) \triangleq (i \in connecting[i] \land IsQuorum(connecting[i])) =$ FALSE
$WaitingForNewEpochTurnToFalse(i) \triangleq \land i \in connecting'[i]$

$\land\ IsQuorum(connecting'[i])$

Leader waits for receiving *FOLLOWERINFO* from a quorum including itself, and chooses a new epoch e' as its own epoch and broadcasts *LEADERINFO*. See *getEpochToPropose* in Leader for details.

$LeaderProcessFOLLOWERINFO(i,\ j)\ \triangleq$
$\land\ CheckEpoch$ test restrictions of max epoch
$\land\ IsLeader(i)$
$\land\ PendingFOLLOWERINFO(i,\ j)$
$\land$ LET $msg\ \triangleq\ msgs[j][i][1]$
$infoOk\ \triangleq\ IsMyLearner(i,\ j)$
$lastAcceptedEpoch\ \triangleq\ msg.mzxid[1]$
IN
$\land\ infoOk$
$\land\ \lor$ 1. has not broadcast *LEADERINFO*
$\land\ WaitingForNewEpoch(i)$
$\land\ \lor\ \land\ zabState[i] = DISCOVERY$
$\land$ UNCHANGED $violatedInvariants$
$\lor\ \land\ zabState[i] \neq DISCOVERY$
$\land\ PrintT(\text{“Exception: waitingFotNewEpoch true,” } \circ$
$\text{“ while zabState not DISCOVERY.”})$
$\land\ violatedInvariants' = [violatedInvariants$ EXCEPT $!.stateInconsistent = \text{TRUE}]$
$\land\ tempMaxEpoch' = [tempMaxEpoch$ EXCEPT $![i] = \text{IF } lastAcceptedEpoch \geq tempMaxEpoch[i]$
THEN $lastAcceptedEpoch + 1$
ELSE $@]$
$\land\ connecting'\ = [connecting$ EXCEPT $![i]\ = @ \cup \{j\}]$
$\land\ \lor\ \land\ WaitingForNewEpochTurnToFalse(i)$
$\land\ acceptedEpoch' = [acceptedEpoch$ EXCEPT $![i] = tempMaxEpoch'[i]]$
$\land$ LET $newLeaderZxid\ \triangleq\ \langle acceptedEpoch'[i],\ 0\rangle$
$m\ \triangleq\ [mtype \mapsto LEADERINFO,$
$mzxid \mapsto newLeaderZxid]$
IN $DiscardAndBroadcastLEADERINFO(i,\ j,\ m)$
$\lor\ \land\ \neg WaitingForNewEpochTurnToFalse(i)$
$\land\ Discard(j,\ i)$
$\land$ UNCHANGED $acceptedEpoch$
$\lor$ 2. has broadcast *LEADERINFO*
$\land\ \neg WaitingForNewEpoch(i)$
$\land\ Reply(i,\ j,\ [mtype \mapsto LEADERINFO,$
$mzxid \mapsto \langle acceptedEpoch[i],\ 0\rangle])$
$\land$ UNCHANGED $\langle tempMaxEpoch,\ connecting,\ acceptedEpoch,\ violatedInvariants\rangle$
$\land$ UNCHANGED $\langle state,\ currentEpoch,\ lastProcessed,\ zabState,\ history,\ lastCommitted,$
$followerVars,\ electionVars,\ initialHistory,\ leadingVoteSet,\ learners,$
$electing,\ ackldRecv,\ forwarding,\ proposalMsgsLog,\ epochLeader,$
$electionMsgs\rangle$
$\land\ UpdateRecorder(\langle \text{“LeaderProcessFOLLOWERINFO”},\ i,\ j\rangle)$

Follower receives *LEADERINFO*. If *newEpoch* $\geq$ *acceptedEpoch*, then follower updates *acceptedEpoch* and sends *ACKEPOCH* back, containing *currentEpoch* and *lastProcessedZxid*. After this, *zabState* turns to *SYNC*. See *registerWithLeader* in Learner for details.

$$
\begin{array}{l}
FollowerProcessLEADERINFO(i, j) \triangleq \\
\quad \land\ IsFollower(i) \\
\quad \land\ PendingLEADERINFO(i, j) \\
\quad \land\ \text{LET}\ msg \triangleq msgs[j][i][1] \\
\qquad\quad newEpoch \triangleq msg.mzxid[1] \\
\qquad\quad infoOk \triangleq IsMyLeader(i, j) \\
\qquad\quad epochOk \triangleq newEpoch \geq acceptedEpoch[i] \\
\qquad\quad stateOk \triangleq zabState[i] = DISCOVERY \\
\qquad \text{IN}\ \land\ infoOk \\
\qquad\quad \land\ \lor\ \text{1. Normal case} \\
\qquad\qquad\quad \land\ epochOk \\
\qquad\qquad\quad \land\ \lor\ \land\ stateOk \\
\qquad\qquad\qquad\qquad \land\ \lor\ \land\ newEpoch > acceptedEpoch[i] \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ acceptedEpoch' = [acceptedEpoch\ \text{EXCEPT}\ ![i] = newEpoch] \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \text{LET}\ epochBytes \triangleq currentEpoch[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad m \triangleq [mtype \mapsto ACKEPOCH, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad mzxid \mapsto lastProcessed[i].zxid, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad mepoch \mapsto epochBytes] \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \text{IN}\ Reply(i, j, m) \\
\qquad\qquad\qquad\qquad\quad \lor\ \land\ newEpoch = acceptedEpoch[i] \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \text{LET}\ m \triangleq [mtype \mapsto ACKEPOCH, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad mzxid \mapsto lastProcessed[i].zxid, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad mepoch \mapsto -1] \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \text{IN}\ Reply(i, j, m) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \text{UNCHANGED}\ acceptedEpoch \\
\qquad\qquad\qquad\qquad \land\ zabState' = [zabState\ \text{EXCEPT}\ ![i] = SYNCHRONIZATION] \\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED}\ violatedInvariants \\
\qquad\qquad\qquad\quad \lor\ \land\ \neg stateOk \\
\qquad\qquad\qquad\qquad \land\ PrintT(\text{“Exception: Follower receives LEADERINFO,”} \circ \\
\qquad\qquad\qquad\qquad\quad \text{“ whileZabState not DISCOVERY.”}) \\
\qquad\qquad\qquad\qquad \land\ violatedInvariants' = [violatedInvariants\ \text{EXCEPT}\ !.stateInconsistent = \text{TRUE}] \\
\qquad\qquad\qquad\qquad \land\ Discard(j, i) \\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED}\ \langle acceptedEpoch, zabState \rangle \\
\qquad\qquad\quad \land\ \text{UNCHANGED}\ \langle varsL, leaderAddr, learners, forwarding, electing \rangle \\
\qquad\qquad \lor\ \text{2. Abnormal case - go back to election} \\
\qquad\qquad\quad \land\ \neg epochOk \\
\qquad\qquad\quad \land\ FollowerShutdown(i) \\
\qquad\qquad\quad \land\ Clean(i, leaderAddr[i]) \\
\qquad\qquad\quad \land\ RemoveLearner(leaderAddr[i], i) \\
\qquad\qquad\quad \land\ \text{UNCHANGED}\ \langle acceptedEpoch, violatedInvariants \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle history, lastCommitted, connecting, ackldRecv, tempMaxEpoch, \\
\qquad\qquad\qquad initialHistory, proposalMsgsLog, epochLeader, packetsSync \rangle
\end{array}
$$

$\wedge$ *UpdateRecorder*($\langle$ "FollowerProcessLEADERINFO", i, $j\rangle$)

RECURSIVE *UpdateAckSidHelper*(_, _, _, _)
UpdateAckSidHelper(*his*, *cur*, *end*, *target*) $\triangleq$
IF *cur* $>$ *end* THEN *his*
ELSE LET *curTxn* $\triangleq$ [*zxid* $\mapsto$ *his*[1].*zxid*,
value $\mapsto$ *his*[1].*value*,
ackSid $\mapsto$ IF *target* $\in$ *his*[1].*ackSid* THEN *his*[1].*ackSid*
ELSE *his*[1].*ackSid* $\cup$ {*target*},
epoch $\mapsto$ *his*[1].*epoch*]
IN $\langle$*curTxn*$\rangle$ $\circ$ *UpdateAckSidHelper*(*Tail*(*his*), *cur* + 1, *end*, *target*)

There originally existed one bug in *LeaderProcessACK* when *monotonicallyInc* = FALSE, and it is we did not add *ackSid* of history in *SYNC*. So we update *ackSid* in *syncFollower*.
UpdateAckSid(*his*, *lastSeenIndex*, *target*) $\triangleq$
IF *Len*(*his*) = 0 $\vee$ *lastSeenIndex* = 0 THEN *his*
ELSE *UpdateAckSidHelper*(*his*, 1, *Minimum*({*Len*(*his*), *lastSeenIndex*}), *target*)

return -1: this *zxid* appears at least twice; *Len*(*his*) + 1: does not exist;
1 $\neg$*Len*(*his*): exists and appears just once.
RECURSIVE *ZxidToIndexHepler*(_, _, _, _)
ZxidToIndexHepler(*his*, *zxid*, *cur*, *appeared*) $\triangleq$
IF *cur* $>$ *Len*(*his*) THEN *cur*
ELSE IF *TxnZxidEqual*(*his*[*cur*], *zxid*)
THEN CASE *appeared* = TRUE $\rightarrow$ -1
$\square$ OTHER $\rightarrow$ *Minimum*({*cur*,
ZxidToIndexHepler(*his*, *zxid*, *cur* + 1, TRUE)})
ELSE *ZxidToIndexHepler*(*his*, *zxid*, *cur* + 1, *appeared*)

ZxidToIndex(*his*, *zxid*) $\triangleq$ IF *ZxidEqual*(*zxid*, $\langle$0, 0$\rangle$) THEN 0
ELSE IF *Len*(*his*) = 0 THEN 1
ELSE LET *len* $\triangleq$ *Len*(*his*)IN
IF $\exists$ *idx* $\in$ 1 .. *len* : *TxnZxidEqual*(*his*[*idx*], *zxid*)
THEN *ZxidToIndexHepler*(*his*, *zxid*, 1, FALSE)
ELSE *len* + 1

Find index *idx* which meets:
history[*idx*].*zxid* $\leq$ *zxid* $<$ *history*[*idx* + 1].*zxid*
RECURSIVE *IndexOfZxidHelper*(_, _, _, _)
IndexOfZxidHelper(*his*, *zxid*, *cur*, *end*) $\triangleq$
IF *cur* $>$ *end* THEN *end*
ELSE IF *ZxidCompare*(*his*[*cur*].*zxid*, *zxid*) THEN *cur* $-$ 1
ELSE *IndexOfZxidHelper*(*his*, *zxid*, *cur* + 1, *end*)

IndexOfZxid(*his*, *zxid*) $\triangleq$ IF *Len*(*his*) = 0 THEN 0

```
                ELSE LET idx ≜ ZxidToIndex(his, zxid)
                         len ≜ Len(his)
                     IN
                     IF idx ≤ len THEN idx
                      ELSE IndexOfZxidHelper(his, zxid, 1, len)

RECURSIVE queuePackets(_, _, _, _, _)
queuePackets(queue, his, cur, committed, end) ≜
    IF cur > end THEN queue
     ELSE CASE cur > committed →
               LET m_proposal ≜ [mtype ↦ PROPOSAL,
                                 mzxid ↦ his[cur].zxid,
                                 mdata ↦ his[cur].value]
               IN queuePackets(Append(queue, m_proposal), his, cur + 1, committed, end)
          □ cur ≤ committed →
               LET m_proposal ≜ [mtype ↦ PROPOSAL,
                                 mzxid ↦ his[cur].zxid,
                                 mdata ↦ his[cur].value]
                   m_commit   ≜ [mtype ↦ COMMIT,
                                 mzxid ↦ his[cur].zxid]
                   newQueue   ≜ queue ∘ ⟨m_proposal, m_commit⟩
               IN queuePackets(newQueue, his, cur + 1, committed, end)

RECURSIVE setPacketsForChecking(_, _, _, _, _, _)
setPacketsForChecking(set, src, ep, his, cur, end) ≜
    IF cur > end THEN set
     ELSE LET m_proposal ≜ [source ↦ src,
                            epoch  ↦ ep,
                            zxid   ↦ his[cur].zxid,
                            data   ↦ his[cur].value]
          IN setPacketsForChecking((set ∪ {m_proposal}), src, ep, his, cur + 1, end)
```

See *queueCommittedProposals* in *LearnerHandler* and *startForwarding* in Leader for details. For proposals in *committedLog* and *toBeApplied*, send $< PROPOSAL, COMMIT >$. For proposals in *outstandingProposals*, send *PROPOSAL* only.

```
StartForwarding(i, j, lastSeenZxid, lastSeenIndex, mode, needRemoveHead) ≜
    ∧ LET lastCommittedIndex ≜ IF zabState[i] = BROADCAST
                               THEN lastCommitted[i].index
                               ELSE Len(initialHistory[i])
          lastProposedIndex  ≜ Len(history[i])
          queue_origin ≜ IF lastSeenIndex ≥ lastProposedIndex
                         THEN ⟨⟩
                         ELSE queuePackets(⟨⟩, history[i],
                                           lastSeenIndex + 1, lastCommittedIndex,
                                           lastProposedIndex)
          set_forChecking ≜ IF lastSeenIndex ≥ lastProposedIndex
```

```
                                THEN {}
                                ELSE setPacketsForChecking({}, i,
                                       acceptedEpoch[i], history[i],
                                       lastSeenIndex + 1, lastProposedIndex)
          m_trunc ≜ [mtype ↦ TRUNC, mtruncZxid ↦ lastSeenZxid]
          m_diff  ≜ [mtype ↦ DIFF,  mzxid ↦ lastSeenZxid]
          newLeaderZxid ≜ ⟨acceptedEpoch[i], 0⟩
          m_newleader ≜ [mtype ↦ NEWLEADER,
                         mzxid ↦ newLeaderZxid]
          queue_toSend ≜ CASE mode = TRUNC → (⟨m_trunc⟩ ∘ queue_origin) ∘ ⟨m_newleader⟩
                         □    OTHER       → (⟨m_diff⟩  ∘ queue_origin) ∘ ⟨m_newleader⟩
   IN   ∧ ∨ ∧ needRemoveHead
            ∧ DiscardAndSendPackets(i, j, queue_toSend)
          ∨ ∧ ¬needRemoveHead
            ∧ SendPackets(i, j, queue_toSend)
        ∧ proposalMsgsLog' = proposalMsgsLog ∪ set_forChecking
  ∧ forwarding' = [forwarding EXCEPT ![i] = @ ∪ {j}]
  ∧ history' = [history EXCEPT ![i] = UpdateAckSid(@, lastSeenIndex, j)]
```

Leader syncs with follower using *DIFF*/*TRUNC*/*PROPOSAL*/*COMMIT* ... See *syncFollower* in *LearnerHandler* for details.

```
SyncFollower(i, j, peerLastZxid, needRemoveHead) ≜
  LET IsPeerNewEpochZxid ≜ peerLastZxid[2] = 0
      lastProcessedZxid  ≜ lastProcessed[i].zxid
      maxCommittedLog    ≜ IF zabState[i] = BROADCAST
                           THEN lastCommitted[i].zxid
                           ELSE LET totalLen ≜ Len(initialHistory[i])
                                IN  IF totalLen = 0 THEN ⟨0, 0⟩
                                    ELSE history[i][totalLen].zxid
```

Hypothesis: 1. *minCommittedLog* : *zxid* of head of history, so no SNAP.
2. *maxCommittedLog* = *lastCommitted*, to compress state space.
3. merge *queueCommittedProposals*,*startForwarding* and sending *NEWLEADER* into *StartForwarding*.

```
   IN   ∨ case1. peerLastZxid = lastProcessedZxid
          DIFF + StartForwarding(lastProcessedZxid)
        ∧ ZxidEqual(peerLastZxid, lastProcessedZxid)
        ∧ StartForwarding(i, j, peerLastZxid, lastProcessed[i].index,
                          DIFF, needRemoveHead)
      ∨ ∧ ¬ZxidEqual(peerLastZxid, lastProcessedZxid)
        ∧ ∨ case2. peerLastZxid > maxCommittedLog
            TRUNC + StartForwarding(maxCommittedLog)
          ∧ ZxidCompare(peerLastZxid, maxCommittedLog)
          ∧ LET maxCommittedIndex ≜ IF zabState[i] = BROADCAST
                                    THEN lastCommitted[i].index
```

```
                                                ELSE  Len(initialHistory[i])
            IN  StartForwarding(i, j, maxCommittedLog, maxCommittedIndex,
                                TRUNC, needRemoveHead)
        ∨  case3. minCommittedLog ≤ peerLastZxid ≤ maxCommittedLog
           ∧ ¬ZxidCompare(peerLastZxid, maxCommittedLog)
           ∧ LET lastSeenIndex ≜ ZxidToIndex(history[i], peerLastZxid)
                 exist ≜ ∧ lastSeenIndex ≥ 0
                         ∧ lastSeenIndex ≤ Len(history[i])
                 lastIndex ≜ IF exist THEN lastSeenIndex
                             ELSE IndexOfZxid(history[i], peerLastZxid)
                  Maximum zxid that < peerLastZxid
                 lastZxid ≜ IF exist THEN peerLastZxid
                            ELSE IF lastIndex = 0 THEN ⟨0, 0⟩
                                 ELSE history[i][lastIndex].zxid
             IN
             ∨  case 3.1. peerLastZxid exists in history
                        DIFF + StartForwarding
                ∧ exist
                ∧ StartForwarding(i, j, peerLastZxid, lastSeenIndex,
                                  DIFF, needRemoveHead)
             ∨  case 3.2. peerLastZxid does not exist in history
                        TRUNC + StartForwarding
                ∧ ¬exist
                ∧ StartForwarding(i, j, lastZxid, lastIndex,
                                  TRUNC, needRemoveHead)
      we will not have case 4 where peerLastZxid < minCommittedLog, because
      minCommittedLog default value is 1 in our spec.

 compare state summary of two servers
IsMoreRecentThan(ss1, ss2) ≜ ∨ ss1.currentEpoch > ss2.currentEpoch
                             ∨ ∧ ss1.currentEpoch = ss2.currentEpoch
                               ∧ ZxidCompare(ss1.lastZxid, ss2.lastZxid)

 electionFinished in Leader
ElectionFinished(i, set) ≜ ∧ i ∈ set
                           ∧ IsQuorum(set)

 There may exist some follower shuts down and connects again, while
 it has sent ACKEPOCH or updated currentEpoch last time. This means
 sid of this follower has existed in elecingFollower but its info
 is old. So we need to make sure each sid in electingFollower is
 unique and latest(newest).
UpdateElecting(oldSet, sid, peerLastZxid, inQuorum) ≜
    LET sid_electing ≜ {s.sid : s ∈ oldSet}
    IN  IF sid ∈ sid_electing
        THEN LET old_info ≜ CHOOSE info ∈ oldSet : info.sid = sid
```

```
                    follower_info ≜
                                [sid            ↦ sid,
                                 peerLastZxid   ↦ peerLastZxid,
                                 inQuorum       ↦ (inQuorum ∨ old_info.inQuorum)]
              IN    (oldSet \ {old_info}) ∪ {follower_info}
          ELSE  LET follower_info ≜
                                [sid            ↦ sid,
                                 peerLastZxid   ↦ peerLastZxid,
                                 inQuorum       ↦ inQuorum]
              IN    oldSet ∪ {follower_info}

LeaderTurnToSynchronization(i) ≜
    ∧ currentEpoch' = [currentEpoch EXCEPT ![i] = acceptedEpoch[i]]
    ∧ zabState'     = [zabState     EXCEPT ![i] = SYNCHRONIZATION]
```

Leader waits for receiving *ACKEPOPCH* from a quorum, and check whether it has most recent state summary from them. After this, leader's *zabState* turns to *SYNCHRONIZATION*. See *waitForEpochAck* in Leader for details.

```
LeaderProcessACKEPOCH(i, j) ≜
    ∧ IsLeader(i)
    ∧ PendingACKEPOCH(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
          leaderStateSummary   ≜ [currentEpoch ↦ currentEpoch[i],
                                  lastZxid     ↦ lastProcessed[i].zxid]
          followerStateSummary ≜ [currentEpoch ↦ msg.mepoch,
                                  lastZxid     ↦ msg.mzxid]
          logOk ≜  whether follower is no more up-to-date than leader
                   ¬IsMoreRecentThan(followerStateSummary, leaderStateSummary)
          electing_quorum ≜ {e ∈ electing[i] : e.inQuorum = TRUE}
          sid_electing ≜ {s.sid : s ∈ electing_quorum}
      IN  ∧ infoOk
          ∧ ∨  electionFinished = true, jump ouf of waitForEpochAck.
               Different from code, here we still need to record info
               into electing, to help us perform syncFollower afterwards.
               Since electing already meets quorum, it does not break
               consistency between code and spec.
            ∧ ElectionFinished(i, sid_electing)
            ∧ electing' = [electing EXCEPT ![i] = UpdateElecting(@, j, msg.mzxid, FALSE)]
            ∧ Discard(j, i)
            ∧ UNCHANGED ⟨varsL, zabState, forwarding, leaderAddr,
                         learners, epochLeader, violatedInvariants⟩
            ∨ ∧ ¬ElectionFinished(i, sid_electing)
              ∧ ∨ ∧ zabState[i] = DISCOVERY
                  ∧ UNCHANGED violatedInvariants
                ∨ ∧ zabState[i] ≠ DISCOVERY
```

```
                    ∧ PrintT("Exception: electionFinished false," ∘
                      " while zabState not DISCOVERY.")
                    ∧ violatedInvariants' = [violatedInvariants EXCEPT
                                             !.stateInconsistent = TRUE]
                ∧ ∨ ∧ followerStateSummary.currentEpoch = −1
                    ∧ electing' = [electing EXCEPT ![i]      = UpdateElecting(@, j,
                                                               msg.mzxid, FALSE)]
                    ∧ Discard(j, i)
                    ∧ UNCHANGED ⟨varsL, zabState, forwarding, leaderAddr,
                                 learners, epochLeader⟩
                  ∨ ∧ followerStateSummary.currentEpoch > −1
                    ∧ ∨ normal follower
                      ∧ logOk
                      ∧ electing' = [electing EXCEPT ![i] =
                                     UpdateElecting(@, j, msg.mzxid, TRUE)]
                      ∧ LET new_electing_quorum ≜ {e ∈ electing'[i] : e.inQuorum = TRUE}
                            new_sid_electing ≜ {s.sid : s ∈ new_electing_quorum}
                        IN
                        ∨ electionFinished = true, jump out of waitForEpochAck,
                          update currentEpoch and zabState.
                          ∧ ElectionFinished(i, new_sid_electing)
                          ∧ LeaderTurnToSynchronization(i)
                          ∧ LET newLeaderEpoch ≜ acceptedEpoch[i]
                            IN  epochLeader' = [epochLeader EXCEPT ![newLeaderEpoch]
                                                = @ ∪ {i}] for checking invariants
                        ∨ there still exists electionFinished = false.
                          ∧ ¬ElectionFinished(i, new_sid_electing)
                          ∧ UNCHANGED ⟨currentEpoch, zabState, epochLeader⟩
                      ∧ Discard(j, i)
                      ∧ UNCHANGED ⟨state, lastProcessed, electionVars, leadingVoteSet,
                                   electionMsgs, leaderAddr, learners, history, forwarding⟩
                    ∨ Exists follower more recent than leader
                      ∧ ¬logOk
                      ∧ LeaderShutdown(i)
                      ∧ UNCHANGED ⟨electing, epochLeader⟩
            ∧ UNCHANGED ⟨acceptedEpoch, lastCommitted, connecting, ackldRecv,
                         tempMaxEpoch, initialHistory, packetsSync, proposalMsgsLog⟩
            ∧ UpdateRecorder(⟨"LeaderProcessACKEPOCH", i, j⟩)

Strip syncFollower from LeaderProcessACKEPOCH.
Only when electionFinished = true and there exists some
learnerHandler has not perform syncFollower, this
action will be called.
LeaderSyncFollower(i) ≜
    ∧ IsLeader(i)
```

```
        ∧ LET electing_quorum ≜ {e ∈ electing[i] : e.inQuorum = TRUE}
              electionFinished ≜ ElectionFinished(i, {s.sid : s ∈ electing_quorum})
              toSync ≜ {s ∈ electing[i] : ∧ ¬ZxidEqual(s.peerLastZxid, ⟨−1, −1⟩)
                                          ∧ s.sid ∈ learners[i]}
              canSync ≜ toSync ≠ {}
          IN
          ∧ electionFinished
          ∧ canSync
          ∧ LET chosen ≜ CHOOSE s ∈ toSync : TRUE
                newChosen ≜ [sid           ↦ chosen.sid,
                             peerLastZxid ↦ ⟨−1, −1⟩,   ⟨−1, −1⟩ means has handled.
                             inQuorum     ↦ chosen.inQuorum]
            IN  ∧ SyncFollower(i, chosen.sid, chosen.peerLastZxid, FALSE)
                ∧ electing' = [electing EXCEPT ![i] = (@ \ {chosen}) ∪ {newChosen}]
        ∧ UNCHANGED ⟨state, currentEpoch, lastProcessed, zabState, acceptedEpoch,
                     lastCommitted, initialHistory, electionVars, leadingVoteSet,
                     learners, connecting, ackldRecv, tempMaxEpoch, followerVars,
                     epochLeader, violatedInvariants, electionMsgs⟩
        ∧ UpdateRecorder(⟨"LeaderSyncFollower", i⟩)

TruncateLog(his, index) ≜ IF index ≤ 0 THEN ⟨⟩
                          ELSE SubSeq(his, 1, index)
```

Follower receives *DIFF*/*TRUNC*, and then may receives *PROPOSAL*,*COMMIT*,*NEWLEADER*, and *UPTODATE*. See *syncWithLeader* in Learner for details.

```
FollowerProcessSyncMessage(i, j) ≜
        ∧ IsFollower(i)
        ∧ msgs[j][i] ≠ ⟨⟩
        ∧ msgs[j][i][1].mtype = DIFF ∨ msgs[j][i][1].mtype = TRUNC
        ∧ LET msg ≜ msgs[j][i][1]
              infoOk ≜ IsMyLeader(i, j)
              stateOk ≜ zabState[i] = SYNCHRONIZATION
          IN  ∧ infoOk
              ∧ ∨   Follower should receive packets in SYNC.
                    ∧ ¬stateOk
                    ∧ PrintT("Exception: Follower receives DIFF/TRUNC," ∘
                             " whileZabState not SYNCHRONIZATION.")
                    ∧ violatedInvariants' = [violatedInvariants EXCEPT !.stateInconsistent = TRUE]
                    ∧ UNCHANGED ⟨history, initialHistory, lastProcessed, lastCommitted⟩
                ∨ ∧ stateOk
                  ∧ ∨ ∧ msg.mtype = DIFF
                      ∧ UNCHANGED ⟨history, initialHistory, lastProcessed, lastCommitted,
                                   violatedInvariants⟩
                    ∨ ∧ msg.mtype = TRUNC
                      ∧ LET truncZxid ≜ msg.mtruncZxid
```

```
                truncIndex ≜ ZxidToIndex(history[i], truncZxid)
            IN
             ∨ ∧ truncIndex > Len(history[i])
               ∧ PrintT("Exception: TRUNC error.")
               ∧ violatedInvariants' = [violatedInvariants EXCEPT
                                          !.proposalInconsistent = TRUE]
               ∧ UNCHANGED ⟨history, initialHistory, lastProcessed, lastCommitted⟩
             ∨ ∧ truncIndex ≤ Len(history[i])
               ∧ history' = [history EXCEPT
                                ![i] = TruncateLog(history[i], truncIndex)]
               ∧ initialHistory' = [initialHistory EXCEPT ![i] = history'[i]]
               ∧ lastProcessed'  = [lastProcessed EXCEPT
                                      ![i] = [index ↦ truncIndex,
                                              zxid  ↦ truncZxid]]
               ∧ lastCommitted' = [lastCommitted EXCEPT
                                      ![i] = [index ↦ truncIndex,
                                              zxid  ↦ truncZxid]]
               ∧ UNCHANGED violatedInvariants
    ∧ Discard(j, i)
    ∧ UNCHANGED ⟨state, currentEpoch, zabState, acceptedEpoch, electionVars,
                 leaderVars, tempMaxEpoch, followerVars,
                 proposalMsgsLog, epochLeader, electionMsgs⟩
    ∧ UpdateRecorder(⟨"FollowerProcessSyncMessage", i, j⟩)

See lastProposed in Leader for details.
LastProposed(i) ≜ IF Len(history[i]) = 0 THEN [index ↦ 0,
                                               zxid  ↦ ⟨0, 0⟩]
                  ELSE
                  LET lastIndex ≜ Len(history[i])
                      entry     ≜ history[i][lastIndex]
                  IN  [index ↦ lastIndex,
                       zxid  ↦ entry.zxid]

See lastQueued in Learner for details.
LastQueued(i) ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                THEN LastProposed(i)
                ELSE condition: IsFollower(i) ∧ zabState = SYNCHRONIZATION
                     LET packetsInSync ≜ packetsSync[i].notCommitted
                         lenSync   ≜ Len(packetsInSync)
                         totalLen  ≜ Len(history[i]) + lenSync
                     IN  IF lenSync = 0 THEN LastProposed(i)
                         ELSE [index ↦ totalLen,
                               zxid  ↦ packetsInSync[lenSync].zxid]

IsNextZxid(curZxid, nextZxid) ≜
          ∨ first PROPOSAL in this epoch
```

```
                    ∧ nextZxid[2] = 1
                    ∧ curZxid[1] < nextZxid[1]
                ∨  not first PROPOSAL in this epoch
                    ∧ nextZxid[2] > 1
                    ∧ curZxid[1] = nextZxid[1]
                    ∧ curZxid[2] + 1 = nextZxid[2]

FollowerProcessPROPOSALInSync(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingPROPOSAL(i, j)
    ∧ zabState[i] = SYNCHRONIZATION
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLeader(i, j)
          isNext ≜ IsNextZxid(LastQueued(i).zxid, msg.mzxid)
          newTxn ≜ [zxid    ↦ msg.mzxid,
                    value   ↦ msg.mdata,
                    ackSid  ↦ {},      follower do not consider ackSid
                    epoch   ↦ acceptedEpoch[i]]  epoch of this round
      IN  ∧ infoOk
          ∧ ∨ ∧ isNext
              ∧ packetsSync' = [packetsSync EXCEPT ![i].notCommitted
                                  = Append(packetsSync[i].notCommitted, newTxn)]
            ∨ ∧ ¬isNext
              ∧ PrintT("Warn: Follower receives PROPOSAL," ∘
                  " while zxid != lastQueued + 1.")
              ∧ UNCHANGED packetsSync
    logRequest → SyncRequestProcessor → SendAckRequestProcessor →  reply ack
    So here we do not need to send ack to leader.
    ∧ Discard(j, i)
    ∧ UNCHANGED ⟨serverVars, electionVars, leaderVars, leaderAddr,
                 verifyVars, electionMsgs⟩
    ∧ UpdateRecorder(⟨"FollowerProcessPROPOSALInSync", i, j⟩)

RECURSIVE IndexOfFirstTxnWithEpoch(_, _, _, _)
IndexOfFirstTxnWithEpoch(his, epoch, cur, end) ≜
    IF cur > end THEN cur
    ELSE IF his[cur].epoch = epoch THEN cur
         ELSE IndexOfFirstTxnWithEpoch(his, epoch, cur + 1, end)

LastCommitted(i) ≜ IF zabState[i] = BROADCAST THEN lastCommitted[i]
                   ELSE CASE IsLeader(i)  →
                             LET lastInitialIndex ≜ Len(initialHistory[i])
                             IN  IF lastInitialIndex = 0 THEN [index ↦ 0,
                                                               zxid  ↦ ⟨0, 0⟩]
                                 ELSE [index ↦ lastInitialIndex,
                                       zxid  ↦ history[i][lastInitialIndex].zxid]
```

```
                    □ IsFollower(i) →
                      LET completeHis ≜ history[i] ∘ packetsSync[i].notCommitted
                          packetsCommitted ≜ packetsSync[i].committed
                          lenCommitted ≜ Len(packetsCommitted)
                      IN  IF lenCommitted = 0  return last one in initial history
                          THEN LET lastInitialIndex ≜ Len(initialHistory[i])
                                   IN  IF lastInitialIndex = 0
                                       THEN [index ↦ 0,
                                             zxid  ↦ ⟨0, 0⟩]
                                       ELSE [index ↦ lastInitialIndex,
                                             zxid  ↦ completeHis[lastInitialIndex].zxid]
                          ELSE                  return tail of packetsCommitted
                               LET committedIndex ≜ ZxidToIndex(completeHis,
                                                                packetsCommitted[lenCommitted])
                                   IN  [index ↦ committedIndex,
                                        zxid  ↦ packetsCommitted[lenCommitted]]
                    □ OTHER → lastCommitted[i]

TxnWithIndex(i, idx) ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                       THEN history[i][idx]
                       ELSE LET completeHis ≜ history[i] ∘ packetsSync[i].notCommitted
                                IN  completeHis[idx]

To simplify specification, we assume snapshotNeeded = false and writeToTxnLog = true. So
here we just call packetsCommitted.add.
FollowerProcessCOMMITInSync(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingCOMMIT(i, j)
    ∧ zabState[i] = SYNCHRONIZATION
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLeader(i, j)
          committedIndex ≜ LastCommitted(i).index + 1
          exist ≜ ∧ committedIndex ≤ LastQueued(i).index
                  ∧ IsNextZxid(LastCommitted(i).zxid, msg.mzxid)
          match ≜ ZxidEqual(msg.mzxid, TxnWithIndex(i, committedIndex).zxid)
      IN  ∧ infoOk
          ∧ ∨ ∧ exist
              ∧ ∨ ∧ match
                  ∧ packetsSync′ = [packetsSync EXCEPT ![i].committed
                                      = Append(packetsSync[i].committed, msg.mzxid)]
                  ∧ UNCHANGED violatedInvariants
                ∨ ∧ ¬match
                  ∧ PrintT("Warn: Follower receives COMMIT," ∘
                      " but zxid not the next committed zxid in COMMIT.")
                  ∧ violatedInvariants′ = [violatedInvariants EXCEPT
```

```
                                        !.commitInconsistent = TRUE]
                                ∧ UNCHANGED packetsSync
                        ∨ ∧ ¬exist
                          ∧ PrintT("Warn: Follower receives COMMIT," ∘
                              " but no packets with its zxid exists.")
                          ∧ violatedInvariants' = [violatedInvariants EXCEPT
                                  !.commitInconsistent = TRUE]
                          ∧ UNCHANGED packetsSync
        ∧ Discard(j, i)
        ∧ UNCHANGED ⟨serverVars, electionVars, leaderVars,
                     leaderAddr, epochLeader, proposalMsgsLog, electionMsgs⟩
        ∧ UpdateRecorder(⟨"FollowerProcessCOMMITInSync", i, j⟩)

RECURSIVE ACKInBatches(_, _)
ACKInBatches(queue, packets) ≜
    IF packets = ⟨⟩ THEN queue
     ELSE LET head ≜ packets[1]
              newPackets ≜ Tail(packets)
              m_ack ≜ [mtype ↦ ACK,
                       mzxid ↦ head.zxid]
          IN ACKInBatches(Append(queue, m_ack), newPackets)

Update currentEpoch, and logRequest every packets in packetsNotCommitted and clear it. As
syncProcessor will be called in logRequest, we have to reply acks here.
FollowerProcessNEWLEADER(i, j) ≜
        ∧ IsFollower(i)
        ∧ PendingNEWLEADER(i, j)
        ∧ LET msg ≜ msgs[j][i][1]
              infoOk ≜ IsMyLeader(i, j)
              packetsInSync ≜ packetsSync[i].notCommitted
              m_ackld ≜ [mtype ↦ ACKLD,
                         mzxid ↦ msg.mzxid]
              ms_ack ≜ ACKInBatches(⟨⟩, packetsInSync)
              queue_toSend ≜ ⟨m_ackld⟩ ∘ ms_ack  send ACK − NEWLEADER first.
          IN ∧ infoOk
             ∧ currentEpoch' = [currentEpoch EXCEPT ![i] = acceptedEpoch[i]]
             ∧ history'      = [history      EXCEPT ![i] = @ ∘ packetsInSync]
             ∧ packetsSync'  = [packetsSync  EXCEPT ![i].notCommitted = ⟨⟩]
             ∧ DiscardAndSendPackets(i, j, queue_toSend)
        ∧ UNCHANGED ⟨state, lastProcessed, zabState, acceptedEpoch, lastCommitted,
                     electionVars, leaderVars, initialHistory, leaderAddr, verifyVars,
                     electionMsgs⟩
        ∧ UpdateRecorder(⟨"FollowerProcessNEWLEADER", i, j⟩)

quorumFormed in Leader
QuorumFormed(i) ≜ i ∈ ackldRecv[i] ∧ IsQuorum(ackldRecv[i])
```

$QuorumFormedTurnToTrue(i) \triangleq i \in ackldRecv'[i] \land IsQuorum(ackldRecv'[i])$

$UpdateElectionVote(i,\ epoch) \triangleq UpdateProposal(i,\ currentVote[i].proposedLeader,$
$currentVote[i].proposedZxid,\ epoch)$

See *startZkServer* in Leader for details.

$StartZkServer(i) \triangleq$
LET $latest \triangleq LastProposed(i)$
IN $\land\ lastCommitted' = [lastCommitted$ EXCEPT $![i] = latest]$
$\land\ lastProcessed' = [lastProcessed$ EXCEPT $![i] = latest]$
$\land\ UpdateElectionVote(i,\ acceptedEpoch[i])$

$LeaderTurnToBroadcast(i) \triangleq$
$\land\ StartZkServer(i)$
$\land\ zabState' = [zabState$ EXCEPT $![i] = BROADCAST]$

Leader waits for receiving quorum of *ACK* whose lower bits of *zxid* is 0, and broadcasts *UPTODATE*. See *waitForNewLeaderAck* for details.

$LeaderProcessACKLD(i,\ j) \triangleq$
$\land\ IsLeader(i)$
$\land\ PendingACKLD(i,\ j)$
$\land$ LET $msg \triangleq msgs[j][i][1]$
$infoOk \triangleq IsMyLearner(i,\ j)$
$match \triangleq ZxidEqual(msg.mzxid,\ \langle acceptedEpoch[i],\ 0\rangle)$
$currentZxid \triangleq \langle acceptedEpoch[i],\ 0\rangle$
$m\_uptodate \triangleq [mtype \mapsto UPTODATE,$
$mzxid \mapsto currentZxid]$ not important
IN $\land\ infoOk$
$\land\ \lor$ just reply *UPTODATE*.
$\land\ QuorumFormed(i)$
$\land\ Reply(i,\ j,\ m\_uptodate)$
$\land$ UNCHANGED $\langle ackldRecv,\ zabState,\ lastCommitted,\ lastProcessed,$
$currentVote,\ violatedInvariants\rangle$
$\lor\ \land \neg QuorumFormed(i)$
$\land\ \lor\ \land\ match$
$\land\ ackldRecv' = [ackldRecv$ EXCEPT $![i] = @ \cup \{j\}]$
$\land\ \lor$ jump out of *waitForNewLeaderAck*, and do *startZkServer*,
setZabState, and reply *UPTODATE*.
$\land\ QuorumFormedTurnToTrue(i)$
$\land\ LeaderTurnToBroadcast(i)$
$\land\ DiscardAndBroadcastUPTODATE(i,\ j,\ m\_uptodate)$
$\lor$ still wait in *waitForNewLeaderAck*.
$\land \neg QuorumFormedTurnToTrue(i)$
$\land\ Discard(j,\ i)$
$\land$ UNCHANGED $\langle zabState,\ lastCommitted,\ lastProcessed,\ currentVote\rangle$
$\land$ UNCHANGED $violatedInvariants$

```
                    ∨ ∧ ¬match
                      ∧ PrintT("Exception: NEWLEADER ACK is from a different epoch. ")
                      ∧ violatedInvariants' = [violatedInvariants EXCEPT
                                  !.ackInconsistent = TRUE]
                      ∧ Discard(j, i)
                      ∧ UNCHANGED ⟨ackldRecv, zabState, lastCommitted,
                                   lastProcessed, currentVote⟩
          ∧ UNCHANGED ⟨state, currentEpoch, acceptedEpoch, history, logicalClock, receiveVotes,
                       outOfElection, recvQueue, waitNotmsg, leadingVoteSet, learners, connecting,
                       electing, forwarding, tempMaxEpoch, initialHistory, followerVars,
                       proposalMsgsLog, epochLeader, electionMsgs⟩
          ∧ UpdateRecorder(⟨"LeaderProcessACKLD", i, j⟩)

TxnsWithPreviousEpoch(i) ≜
        LET completeHis ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                          THEN history[i]
                          ELSE history[i] ∘ packetsSync[i].notCommitted
            end   ≜ Len(completeHis)
            first ≜ IndexOfFirstTxnWithEpoch(completeHis, acceptedEpoch[i], 1, end)
        IN  IF first > end THEN completeHis
            ELSE SubSeq(completeHis, 1, first − 1)

TxnsRcvWithCurEpoch(i) ≜
        LET completeHis ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                          THEN history[i]
                          ELSE history[i] ∘ packetsSync[i].notCommitted
            end   ≜ Len(completeHis)
            first ≜ IndexOfFirstTxnWithEpoch(completeHis, acceptedEpoch[i], 1, end)
        IN  IF first > end THEN ⟨⟩
            ELSE SubSeq(completeHis, first, end)  completeHis[first : end]

Txns received in current epoch but not committed.
See pendingTxns in FollowerZooKeeper for details.
PendingTxns(i) ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                 THEN SubSeq(history[i], lastCommitted[i].index + 1, Len(history[i]))
                 ELSE LET packetsCommitted ≜ packetsSync[i].committed
                          completeHis ≜ history[i] ∘ packetsSync[i].notCommitted
                      IN  IF Len(packetsCommitted) = 0
                          THEN SubSeq(completeHis, Len(initialHistory[i])     + 1, Len(completeHis))
                          ELSE SubSeq(completeHis, LastCommitted(i).index + 1, Len(completeHis))

CommittedTxns(i) ≜ IF ¬IsFollower(i) ∨ zabState[i] ≠ SYNCHRONIZATION
                   THEN SubSeq(history[i], 1, lastCommitted[i].index)
                   ELSE LET packetsCommitted ≜ packetsSync[i].committed
                            completeHis ≜ history[i] ∘ packetsSync[i].notCommitted
                        IN  IF Len(packetsCommitted) = 0 THEN initialHistory[i]
```

```
                                ELSE  SubSeq(completeHis, 1, LastCommitted(i).index)

 Each zxid of packetsCommitted equals to zxid of
 corresponding txn in txns.
RECURSIVE TxnsAndCommittedMatch(_, _)
TxnsAndCommittedMatch(txns, packetsCommitted) ≜
    LET len1 ≜ Len(txns)
        len2 ≜ Len(packetsCommitted)
    IN  IF len2 = 0 THEN TRUE
        ELSE IF len1 < len2 THEN FALSE
             ELSE ∧ ZxidEqual(txns[len1].zxid, packetsCommitted[len2])
                  ∧ TxnsAndCommittedMatch(SubSeq(txns, 1, len1 − 1),
                                          SubSeq(packetsCommitted, 1, len2 − 1))

FollowerLogRequestInBatches(i, leader, ms_ack, packetsNotCommitted) ≜
    ∧ history′ = [history EXCEPT ![i] = @ ∘ packetsNotCommitted]
    ∧ DiscardAndSendPackets(i, leader, ms_ack)

 Since commit will call commitProcessor.commit, which will finally
 update lastProcessed, we update it here atomically.
FollowerCommitInBatches(i) ≜
    LET committedTxns ≜ CommittedTxns(i)
        packetsCommitted ≜ packetsSync[i].committed
        match ≜ TxnsAndCommittedMatch(committedTxns, packetsCommitted)
    IN
    ∨ ∧ match
      ∧ lastCommitted′ = [lastCommitted EXCEPT ![i] = LastCommitted(i)]
      ∧ lastProcessed′  = [lastProcessed EXCEPT ![i]  = lastCommitted′[i]]
      ∧ UNCHANGED violatedInvariants
    ∨ ∧ ¬match
      ∧ PrintT("Warn: Committing zxid withou see txn. /" ∘
          "Committing zxid != pending txn zxid.")
      ∧ violatedInvariants′ = [violatedInvariants EXCEPT
                !.commitInconsistent = TRUE]
      ∧ UNCHANGED ⟨lastCommitted, lastProcessed⟩

 Follower jump out of outerLoop here, and log the stuff that came in between snapshot and
 uptodate, which means calling logRequest and commit to clear packetsNotCommitted and
 packetsCommitted.
FollowerProcessUPTODATE(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingUPTODATE(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLeader(i, j)
          packetsNotCommitted ≜ packetsSync[i].notCommitted
          ms_ack ≜ ACKInBatches(⟨⟩, packetsNotCommitted)
```

IN $\land$ *infoOk*

Here we ignore ack of *UPTODATE*.

$\land$ *UpdateElectionVote*(i, *acceptedEpoch*[i])

$\land$ *FollowerLogRequestInBatches*(i, j, *ms_ack*, *packetsNotCommitted*)

$\land$ *FollowerCommitInBatches*(i)

$\land$ $packetsSync' = [packetsSync$ EXCEPT $![i].notCommitted = \langle\rangle,$
$![i].committed = \langle\rangle]$

$\land$ $zabState' = [zabState$ EXCEPT $![i] = BROADCAST]$

$\land$ UNCHANGED $\langle state, currentEpoch, acceptedEpoch, logicalClock,$
receiveVotes, *outOfElection*, *recvQueue*, *waitNotmsg*, *leaderVars*,
initialHistory, *leaderAddr*, *epochLeader*, *proposalMsgsLog*, *electionMsgs*$\rangle$

$\land$ *UpdateRecorder*($\langle$"FollowerProcessUPTODATE", i, $j\rangle$)

IncZxid(s, *zxid*) $\triangleq$ IF $currentEpoch[s] = zxid[1]$ THEN $\langle zxid[1], zxid[2] + 1\rangle$
ELSE $\langle currentEpoch[s], 1\rangle$

Leader receives client propose and broadcasts *PROPOSAL*. See *processRequest* in *ProposalRequestProcessor* and propose in Leader for details. Since *prosalProcessor.processRequest* $\rightarrow$ *syncProcessor.processRequest* $\rightarrow$ *ackProcessor.processRequest* $\rightarrow$ *leader.processAck*, we initially set *txn.ackSid* $= \{i\}$, assuming we have done *leader.processAck*. Note: In production, any server in traffic can receive requests and
forward it to leader if necessary. We choose to let leader be the sole one who can receive write requests, to simplify spec and keep correctness at the same time.

LeaderProcessRequest(i, v) $\triangleq$

$\land$ *CheckTransactionNum* test restrictions of transaction num

$\land$ *IsLeader*(i)

$\land$ $zabState[i] = BROADCAST$

$\land$ LET $newTxn \triangleq [zxid \mapsto IncZxid(i, LastProposed(i).zxid),$
$value \mapsto v,$
$ackSid \mapsto \{i\},$ assume we have done *leader.processAck*
$epoch \mapsto acceptedEpoch[i]]$
$m\_proposal \triangleq [mtype \mapsto PROPOSAL,$
$mzxid \mapsto newTxn.zxid,$
$mdata \mapsto v]$
$m\_proposal\_for\_checking \triangleq [source \mapsto i,$
$epoch \mapsto acceptedEpoch[i],$
$zxid \mapsto newTxn.zxid,$
$data \mapsto v]$

IN $\land$ $history' = [history$ EXCEPT $![i] = Append(@, newTxn)]$

$\land$ *Broadcast*(i, *m_proposal*)

$\land$ $proposalMsgsLog' = proposalMsgsLog \cup \{m\_proposal\_for\_checking\}$

$\land$ UNCHANGED $\langle state, currentEpoch, lastProcessed, zabState, acceptedEpoch,$
lastCommitted, *electionVars*, *leaderVars*, *followerVars*, *initialHistory*,
epochLeader, *violatedInvariants*, *electionMsgs*$\rangle$

$\land$ *UpdateRecorder*($\langle$"LeaderProcessRequest", i, $v\rangle$)

Follower processes *PROPOSAL* in *BROADCAST*. See *processPacket* in Follower for details.

$$
\begin{array}{l}
FollowerProcessPROPOSAL(i,\, j) \triangleq \\
\quad \land\ IsFollower(i) \\
\quad \land\ PendingPROPOSAL(i,\, j) \\
\quad \land\ zabState[i] = BROADCAST \\
\quad \land\ \text{LET}\ msg \triangleq msgs[j][i][1] \\
\qquad\quad infoOk \triangleq IsMyLeader(i,\, j) \\
\qquad\quad isNext \triangleq IsNextZxid(LastQueued(i).zxid,\, msg.mzxid) \\
\qquad\quad newTxn \triangleq [zxid \mapsto msg.mzxid, \\
\qquad\qquad\qquad\ value \mapsto msg.mdata, \\
\qquad\qquad\qquad\ ackSid \mapsto \{\}, \\
\qquad\qquad\qquad\ epoch \mapsto acceptedEpoch[i]] \\
\qquad\quad m\_ack \triangleq [mtype \mapsto ACK, \\
\qquad\qquad\qquad\ mzxid \mapsto msg.mzxid] \\
\qquad \text{IN}\ \land\ infoOk \\
\qquad\quad \land\ \lor\ \land\ isNext \\
\qquad\qquad\qquad\ \land\ \text{UNCHANGED}\ violatedInvariants \\
\qquad\qquad\quad \lor\ \land\ \neg isNext \\
\qquad\qquad\qquad\ \land\ PrintT(\text{“Exception: Follower receives PROPOSAL, while”} \circ \\
\qquad\qquad\qquad\qquad \text{“ the transaction is not the next.”}) \\
\qquad\qquad\qquad\ \land\ violatedInvariants' = [violatedInvariants\ \text{EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad !.proposalInconsistent = \text{TRUE}] \\
\qquad\quad \land\ history' = [history\ \text{EXCEPT}\ ![i] = Append(@,\, newTxn)] \\
\qquad\quad \land\ Reply(i,\, j,\, m\_ack) \\
\quad \land\ \text{UNCHANGED}\ \langle state,\, currentEpoch,\, lastProcessed,\, zabState,\, acceptedEpoch, \\
\qquad\quad lastCommitted,\, electionVars,\, leaderVars,\, followerVars,\, initialHistory, \\
\qquad\quad epochLeader,\, proposalMsgsLog,\, electionMsgs\rangle \\
\quad \land\ UpdateRecorder(\langle \text{“FollowerProcessPROPOSAL”},\, i,\, j\rangle)
\end{array}
$$

See *outstandingProposals* in Leader

$$
\begin{array}{l}
OutstandingProposals(i) \triangleq \text{IF}\ zabState[i] \neq BROADCAST\ \text{THEN}\ \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ SubSeq(history[i],\, lastCommitted[i].index + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Len(history[i]))
\end{array}
$$

$$
\begin{array}{l}
LastAckIndexFromFollower(i,\, j) \triangleq \\
\quad \text{LET}\ set\_index \triangleq \{idx \in 1\,..\,Len(history[i]) : j \in history[i][idx].ackSid\} \\
\quad \text{IN}\ \ Maximum(set\_index)
\end{array}
$$

See commit in Leader for details.

$$
\begin{array}{l}
LeaderCommit(s,\, follower,\, index,\, zxid) \triangleq \\
\quad \land\ lastCommitted' = [lastCommitted\ \text{EXCEPT}\ ![s] = [index \mapsto index, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad zxid \mapsto zxid]] \\
\quad \land\ \text{LET}\ m\_commit \triangleq [mtype \mapsto COMMIT, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ mzxid \mapsto zxid] \\
\qquad \text{IN}\ \ DiscardAndBroadcast(s,\, follower,\, m\_commit)
\end{array}
$$

```
Try to commit one operation, called by LeaderProcessAck.
See tryToCommit in Leader for details.
commitProcessor.commit → processWrite → toBeApplied.processRequest
 → finalProcessor.processRequest, finally processTxn will be implemented
and lastProcessed will be updated. So we update it here.
LeaderTryToCommit(s, index, zxid, newTxn, follower) ≜
    LET allTxnsBeforeCommitted ≜ lastCommitted[s].index ≥ index − 1
                Only when all proposals before zxid has been committed,
                this proposal can be permitted to be committed.
        hasAllQuorums ≜ IsQuorum(newTxn.ackSid)
                In order to be committed, a proposal must be accepted
                by a quorum.
        ordered ≜ lastCommitted[s].index + 1 = index
                Commit proposals in order.
    IN  ∨ ∧ Current conditions do not satisfy committing the proposal.
          ∨ ¬allTxnsBeforeCommitted
          ∨ ¬hasAllQuorums
        ∧ Discard(follower, s)
        ∧ UNCHANGED ⟨violatedInvariants, lastCommitted, lastProcessed⟩
        ∨ ∧ allTxnsBeforeCommitted
          ∧ hasAllQuorums
          ∧ ∨ ∧ ¬ordered
                ∧ PrintT("Warn: Committing zxid " ∘ ToString(zxid) ∘ " not first.")
                ∧ violatedInvariants′ = [violatedInvariants EXCEPT
                        !.commitInconsistent = TRUE]
            ∨ ∧ ordered
              ∧ UNCHANGED violatedInvariants
          ∧ LeaderCommit(s, follower, index, zxid)
          ∧ lastProcessed′ = [lastProcessed EXCEPT ![s] = [index ↦ index,
                                                           zxid  ↦ zxid]]

Leader Keeps a count of acks for a particular proposal, and try to commit the proposal. See
case Leader.ACK in LearnerHandler, processRequest in AckRequestProcessor, and processAck
in Leader for details.
LeaderProcessACK(i, j) ≜
    ∧ IsLeader(i)
    ∧ PendingACK(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
          outstanding ≜ LastCommitted(i).index < LastProposed(i).index
                  outstandingProposals not null
          hasCommitted ≜ ¬ZxidCompare(msg.mzxid, LastCommitted(i).zxid)
                  namely, lastCommitted ≥ zxid
          index ≜ ZxidToIndex(history[i], msg.mzxid)
          exist ≜ index ≥ 1 ∧ index ≤ LastProposed(i).index
```

```
                    the proposal exists in history
        ackIndex ≜ LastAckIndexFromFollower(i, j)
        monotonicallyInc ≜ ∨ ackIndex = −1
                           ∨ ackIndex + 1 = index
                TCP makes everytime ackIndex should just increase by 1
IN      ∧ infoOk
        ∧ ∨ ∧ exist
              ∧ monotonicallyInc
              ∧ LET txn ≜ history[i][index]
                    txnAfterAddAck ≜ [zxid    ↦ txn.zxid,
                                      value   ↦ txn.value,
                                      ackSid  ↦ txn.ackSid ∪ {j},
                                      epoch   ↦ txn.epoch]
                IN  p.addAck(sid)
                ∧   history′ = [history EXCEPT ![i][index] = txnAfterAddAck]
                ∧   ∨ ∧ Note: outstanding is 0.
                      / proposal has already been committed.
                      ∨ ¬outstanding
                      ∨ hasCommitted
                    ∧ Discard(j, i)
                    ∧ UNCHANGED ⟨violatedInvariants, lastCommitted, lastProcessed⟩
                  ∨ ∧ outstanding
                    ∧ ¬hasCommitted
                    ∧ LeaderTryToCommit(i, index, msg.mzxid, txnAfterAddAck, j)
          ∨ ∧ ∨ ¬exist
              ∨ ¬monotonicallyInc
            ∧ PrintT("Exception: No such zxid. " ∘
                  " / ackIndex doesn't inc monotonically.")
            ∧ violatedInvariants′ = [violatedInvariants
                  EXCEPT !.ackInconsistent = TRUE]
            ∧ Discard(j, i)
            ∧ UNCHANGED ⟨history, lastCommitted, lastProcessed⟩
      ∧ UNCHANGED ⟨state, currentEpoch, zabState, acceptedEpoch, electionVars,
                   leaderVars, initialHistory, followerVars, proposalMsgsLog, epochLeader,
                   electionMsgs⟩
      ∧ UpdateRecorder(⟨"LeaderProcessACK", i, j⟩)

Follower processes COMMIT in BROADCAST. See processPacket in Follower for details.
FollowerProcessCOMMIT(i, j) ≜
      ∧ IsFollower(i)
      ∧ PendingCOMMIT(i, j)
      ∧ zabState[i] = BROADCAST
      ∧ LET msg ≜ msgs[j][i][1]
            infoOk ≜ IsMyLeader(i, j)
            pendingTxns ≜ PendingTxns(i)
```

```
        noPending ≜ Len(pendingTxns) = 0
    IN
    ∧ infoOk
    ∧ ∨ ∧ noPending
          ∧ PrintT("Warn: Committing zxid without seeing txn.")
          ∧ UNCHANGED ⟨lastCommitted, lastProcessed, violatedInvariants⟩
      ∨ ∧ ¬noPending
          ∧ LET firstElementZxid ≜ pendingTxns[1].zxid
                match ≜ ZxidEqual(firstElementZxid, msg.mzxid)
            IN
              ∨ ∧ ¬match
                ∧ PrintT("Exception: Committing zxid not equals" ∘
                         " next pending txn zxid.")
                ∧ violatedInvariants' = [violatedInvariants EXCEPT
                        !.commitInconsistent = TRUE]
                ∧ UNCHANGED ⟨lastCommitted, lastProcessed⟩
              ∨ ∧ match
                ∧ lastCommitted' = [lastCommitted EXCEPT
                        ![i] = [index ↦ lastCommitted[i].index + 1,
                                zxid  ↦ firstElementZxid]]
                ∧ lastProcessed' = [lastProcessed EXCEPT
                        ![i] = [index ↦ lastCommitted[i].index + 1,
                                zxid  ↦ firstElementZxid]]
                ∧ UNCHANGED violatedInvariants
    ∧ Discard(j, i)
    ∧ UNCHANGED ⟨state, currentEpoch, zabState, acceptedEpoch, history,
                 electionVars, leaderVars, initialHistory, followerVars,
                 proposalMsgsLog, epochLeader, electionMsgs⟩
    ∧ UpdateRecorder(⟨"FollowerProcessCOMMIT", i, j⟩)
```

Used to discard some messages which should not exist in network channel. This action should not be triggered.

```
FilterNonexistentMessage(i) ≜
    ∧ ∃ j ∈ Server \ {i} : ∧ msgs[j][i] ≠ ⟨⟩
                           ∧ LET msg ≜ msgs[j][i][1]
                             IN
                               ∨ ∧ IsLeader(i)
                                 ∧ LET infoOk ≜ IsMyLearner(i, j)
                                   IN
                                    ∨ msg.mtype = LEADERINFO
                                    ∨ msg.mtype = NEWLEADER
                                    ∨ msg.mtype = UPTODATE
                                    ∨ msg.mtype = PROPOSAL
                                    ∨ msg.mtype = COMMIT
                                    ∨ ∧ ¬infoOk
```

$$
\begin{array}{l}
\land \lor msg.mtype = FOLLOWERINFO \\
\quad \lor msg.mtype = ACKEPOCH \\
\quad \lor msg.mtype = ACKLD \\
\quad \lor msg.mtype = ACK \\
\lor \land IsFollower(i) \\
\quad \land \text{LET } infoOk \triangleq IsMyLeader(i, j) \\
\quad \text{IN} \\
\quad \lor msg.mtype = FOLLOWERINFO \\
\quad \lor msg.mtype = ACKEPOCH \\
\quad \lor msg.mtype = ACKLD \\
\quad \lor msg.mtype = ACK \\
\quad \lor \land \neg infoOk \\
\quad \land \lor msg.mtype = LEADERINFO \\
\quad \lor msg.mtype = NEWLEADER \\
\quad \lor msg.mtype = UPTODATE \\
\quad \lor msg.mtype = PROPOSAL \\
\quad \lor msg.mtype = COMMIT \\
\lor IsLooking(i) \\
\land Discard(j, i) \\
\land violatedInvariants' = [violatedInvariants \text{ EXCEPT } !.messageIllegal = \text{TRUE}] \\
\land \text{UNCHANGED } \langle serverVars, electionVars, leaderVars, \\
\qquad followerVars, proposalMsgsLog, epochLeader, electionMsgs \rangle \\
\land UnchangeRecorder
\end{array}
$$

Defines how the variables may transition.

$$
\begin{array}{l}
Next \triangleq \\
\quad \text{FLE modlue} \\
\quad \lor \exists i, j \in Server : FLEReceiveNotmsg(i, j) \\
\quad \lor \exists i \in Server : \quad FLENotmsgTimeout(i) \\
\quad \lor \exists i \in Server : \quad FLEHandleNotmsg(i) \\
\quad \lor \exists i \in Server : \quad FLEWaitNewNotmsg(i) \\
\quad \lor \exists i \in Server : \quad FLEWaitNewNotmsgEnd(i) \\
\quad \text{Some conditions like failure, network delay} \\
\quad \lor \exists i \in Server : \quad FollowerTimeout(i) \\
\quad \lor \exists i \in Server : \quad LeaderTimeout(i) \\
\quad \lor \exists i, j \in Server : Timeout(i, j) \\
\quad \text{Zab module - Discovery and Synchronization part} \\
\quad \lor \exists i, j \in Server : ConnectAndFollowerSendFOLLOWERINFO(i, j) \\
\quad \lor \exists i, j \in Server : LeaderProcessFOLLOWERINFO(i, j) \\
\quad \lor \exists i, j \in Server : FollowerProcessLEADERINFO(i, j) \\
\quad \lor \exists i, j \in Server : LeaderProcessACKEPOCH(i, j) \\
\quad \lor \exists i \in Server : \quad LeaderSyncFollower(i) \\
\quad \lor \exists i, j \in Server : FollowerProcessSyncMessage(i, j) \\
\quad \lor \exists i, j \in Server : FollowerProcessPROPOSALInSync(i, j) \\
\quad \lor \exists i, j \in Server : FollowerProcessCOMMITInSync(i, j)
\end{array}
$$

$\lor \exists\, i, j \in Server : FollowerProcessNEWLEADER(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACKLD(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessUPTODATE(i, j)$

Zab module – *Broadcast* part

$\lor \exists\, i \in Server,\, v \in Value : LeaderProcessRequest(i, v)$
$\lor \exists\, i, j \in Server : FollowerProcessPROPOSAL(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACK(i, j)$ *Sync + Broadcast*
$\lor \exists\, i, j \in Server : FollowerProcessCOMMIT(i, j)$

An action used to judge whether there are redundant messages in network

$\lor \exists\, i \in Server : \quad FilterNonexistentMessage(i)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

Define safety properties of *Zab* 1.0 protocol.

$ShouldNotBeTriggered \triangleq \forall\, p \in \text{DOMAIN } violatedInvariants : violatedInvariants[p] = \text{FALSE}$

There is most one established leader for a certain epoch.

$$\begin{aligned} Leadership1 \triangleq\ & \forall\, i, j \in Server : \\ & \land IsLeader(i) \land zabState[i] \in \{SYNCHRONIZATION,\ BROADCAST\} \\ & \land IsLeader(j) \land zabState[j] \in \{SYNCHRONIZATION,\ BROADCAST\} \\ & \land acceptedEpoch[i] = acceptedEpoch[j] \\ & \Rightarrow i = j \end{aligned}$$

$Leadership2 \triangleq \forall\, epoch \in 1 \,..\, MAXEPOCH : Cardinality(epochLeader[epoch]) \leq 1$

PrefixConsistency: The prefix that have been committed
in history in any process is the same.

$$\begin{aligned} PrefixConsistency \triangleq\ & \forall\, i, j \in Server : \\ & \text{LET } smaller \triangleq Minimum(\{lastCommitted[i].index,\ lastCommitted[j].index\}) \\ & \text{IN} \quad \lor smaller = 0 \\ & \qquad \lor \land smaller > 0 \\ & \qquad\quad \land \forall\, index \in 1 \,..\, smaller : \\ & \qquad\qquad TxnEqual(history[i][index],\ history[j][index]) \end{aligned}$$

Integrity: If some follower delivers one transaction, then some primary has broadcast it.

$$\begin{aligned} Integrity \triangleq\ & \forall\, i \in Server : \\ & \land IsFollower(i) \\ & \land lastCommitted[i].index > 0 \\ & \Rightarrow \forall\, idx \in 1 \,..\, lastCommitted[i].index : \exists\, proposal \in proposalMsgsLog : \\ & \qquad ZxidEqual(history[i][idx].zxid,\ proposal.zxid) \end{aligned}$$

Agreement: If some follower f delivers transaction a and some follower f' delivers transaction b,
then f' delivers a or f delivers b.

$$\begin{aligned} Agreement \triangleq\ & \forall\, i, j \in Server : \\ & \land IsFollower(i) \land lastCommitted[i].index > 0 \\ & \land IsFollower(j) \land lastCommitted[j].index > 0 \end{aligned}$$

$$
\begin{array}{l}
\Rightarrow \\
\forall\, idx1 \in 1\,..\,lastCommitted[i].index,\ idx2 \in 1\,..\,lastCommitted[j].index : \\
\quad \lor \exists\, idx\_j \in 1\,..\,lastCommitted[j].index : \\
\qquad TxnEqual(history[j][idx\_j],\ history[i][idx1]) \\
\quad \lor \exists\, idx\_i \in 1\,..\,lastCommitted[i].index : \\
\qquad TxnEqual(history[i][idx\_i],\ history[j][idx2])
\end{array}
$$

Total order: If some follower delivers a before b, then any process that delivers b must also deliver a and deliver a before b.

$$
\begin{array}{l}
TotalOrder \triangleq\ \forall\, i,\ j \in Server : \\
\quad \text{LET}\ committed1 \triangleq\ lastCommitted[i].index \\
\qquad\ \ committed2 \triangleq\ lastCommitted[j].index \\
\quad \text{IN}\ \ committed1 \geq 2 \land committed2 \geq 2 \\
\qquad \Rightarrow \forall\, idx\_i1 \in 1\,..\,(committed1 - 1) : \forall\, idx\_i2 \in (idx\_i1 + 1)\,..\,committed1 : \\
\qquad\ \ \text{LET}\ logOk \triangleq\ \exists\, idx \in 1\,..\,committed2 : \\
\qquad\qquad\qquad TxnEqual(history[i][idx\_i2],\ history[j][idx]) \\
\qquad\ \ \text{IN}\ \ \lor \neg logOk \\
\qquad\qquad \lor\ \land logOk \\
\qquad\qquad\quad \land \exists\, idx\_j2 \in 1\,..\,committed2 : \\
\qquad\qquad\qquad \land\ TxnEqual(history[i][idx\_i2],\ history[j][idx\_j2]) \\
\qquad\qquad\qquad \land \exists\, idx\_j1 \in 1\,..\,(idx\_j2 - 1) : \\
\qquad\qquad\qquad\qquad TxnEqual(history[i][idx\_i1],\ history[j][idx\_j1])
\end{array}
$$

Local primary order: If a primary broadcasts a before it broadcasts b, then a follower that delivers b must also deliver a before b.

$$
\begin{array}{l}
LocalPrimaryOrder \triangleq\ \text{LET}\ p\_set(i,\ e) \triangleq\ \{p \in proposalMsgsLog :\ \land p.source = i \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land p.epoch\ \ = e\} \\
\qquad\qquad zxid\_set(i,\ e) \triangleq\ \{p.zxid : p \in p\_set(i,\ e)\} \\
\qquad\ \ \text{IN}\ \ \forall\, i \in Server : \forall\, e \in 1\,..\,currentEpoch[i] : \\
\qquad\qquad \lor\ Cardinality(zxid\_set(i,\ e)) < 2 \\
\qquad\qquad \lor\ \land\ Cardinality(zxid\_set(i,\ e)) \geq 2 \\
\qquad\qquad\quad \land \exists\, zxid1,\ zxid2 \in zxid\_set(i,\ e) : \\
\qquad\qquad\qquad \lor\ ZxidEqual(zxid1,\ zxid2) \\
\qquad\qquad\qquad \lor\ \land \neg ZxidEqual(zxid1,\ zxid2) \\
\qquad\qquad\qquad\quad \land\ \text{LET}\ zxidPre \triangleq\ \text{IF}\ ZxidCompare(zxid1,\ zxid2)\ \text{THEN}\ zxid2\ \text{ELSE}\ zxid1 \\
\qquad\qquad\qquad\qquad\qquad\ \ zxidNext \triangleq\ \text{IF}\ ZxidCompare(zxid1,\ zxid2)\ \text{THEN}\ zxid1\ \text{ELSE}\ zxid2 \\
\qquad\qquad\qquad\qquad\quad \text{IN}\ \ \forall\, j \in Server :\ \land lastCommitted[j].index \geq 2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land \exists\, idx \in 1\,..\,lastCommitted[j].index : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ZxidEqual(history[j][idx].zxid,\ zxidNext) \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \exists\, idx2 \in 1\,..\,lastCommitted[j].index : \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ ZxidEqual(history[j][idx2].zxid,\ zxidNext) \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ idx2 > 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \land \exists\, idx1 \in 1\,..\,(idx2 - 1) : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ZxidEqual(history[j][idx1].zxid,\ zxidPre)
\end{array}
$$

Global primary order: A follower f delivers both a with epoch e and b with epoch e', and $e < e'$,

then f must deliver a before b.

$$\begin{aligned}
GlobalPrimaryOrder \triangleq\ & \forall\, i \in Server : lastCommitted[i].index \geq 2 \\
& \Rightarrow \forall\, idx1,\ idx2 \in 1\,..\,lastCommitted[i].index : \\
& \quad \lor\ \neg EpochPrecedeInTxn(history[i][idx1],\ history[i][idx2]) \\
& \quad \lor\ \land\ EpochPrecedeInTxn(history[i][idx1],\ history[i][idx2]) \\
& \qquad \land\ idx1 < idx2
\end{aligned}$$

Primary integrity: If primary p broadcasts a and some follower f delivers b such that b has epoch smaller than epoch of p, then p must deliver b before it broadcasts a.

$$\begin{aligned}
PrimaryIntegrity \triangleq\ & \forall\, i,\ j \in Server : \land\ IsLeader(i) \quad \land\ IsMyLearner(i,\ j) \\
& \qquad \land\ IsFollower(j) \land\ IsMyLeader(j,\ i) \\
& \qquad \land\ zabState[i] = BROADCAST \\
& \qquad \land\ zabState[j] = BROADCAST \\
& \qquad \land\ lastCommitted[j].index \geq 1 \\
& \Rightarrow \forall\, idx_j \in 1\,..\,lastCommitted[j].index : \\
& \quad \lor\ history[j][idx_j].zxid[1] \geq currentEpoch[i] \\
& \quad \lor\ \land\ history[j][idx_j].zxid[1] < currentEpoch[i] \\
& \qquad \land\ \exists\, idx_i \in 1\,..\,lastCommitted[i].index : \\
& \qquad\ TxnEqual(history[i][idx_i],\ history[j][idx_j])
\end{aligned}$$

\ * Modification History
\ * Last modified Sun *Nov* 21 21:41:22 *CST* 2021 by Dell
\ * Created Sat *Oct* 23 16:05:04 *CST* 2021 by Dell