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* * Replaced by *recorder.nClientRequest*
Value $\triangleq$ *Nat*

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 *clientReuqest* * Start from 0, and increases monotonically when *LeaderProcessRequest* performed. To avoid existing two requests with same value.
Update: Remove it to *recorder.nClientRequest*.

Variable used for recording critical data,
to constrain state space or update values.

VARIABLE *recorder* Consists: members of *Parameters* and *pc*, values.
Form is record:
[*pc*, *nTransaction*, *maxEpoch*, *nTimeout*, *nClientRequest*]

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

$$electionVars \triangleq electionVarsL$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$ViolationSet \triangleq \{$"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
$\land\ electionMsgs \in [Server \to [Server \to Seq(ElectionMsg)]]$
$\land\ recvQueue \in [Server \to Seq(ElectionMsg)]$
$\land\ leadingVoteSet \in [Server \to \text{SUBSET}\ Server]$
$\land\ receiveVotes \in [Server \to [Server \to ElectionVote]]$
$\land\ currentVote \in [Server \to\ Vote]$
$\land\ outOfElection \in [Server \to [Server \to ElectionVote]]$
$\land\ lastProcessed\ \ \in [Server \to LastItem]$
$\land\ history \in [Server \to Seq(HistoryItem)]$
$\land\ state \in [Server \to ElectionState]$
$\land\ waitNotmsg \in [Server \to \text{BOOLEAN}\ ]$
$\land\ currentEpoch \in [Server \to Nat]$
$\land\ logicalClock\quad \in [Server \to Nat]$

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

$$ToZxid(z) \triangleq [epoch \mapsto z[1],\ counter \mapsto z[2]]$$

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

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

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

Actions about recorder

$$GetParameter(p) \triangleq \text{IF } p \in \text{DOMAIN } Parameters \text{ THEN } Parameters[p] \text{ ELSE } 0$$
$$GetRecorder(p) \triangleq \text{IF } p \in \text{DOMAIN } recorder \text{ THEN } recorder[p] \text{ ELSE } 0$$

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

$$RecorderIncTimeout \triangleq RecorderIncHelper(\text{“nTimeout”})$$
$$RecorderGetTimeout \triangleq RecorderGetHelper(\text{“nTimeout”})$$
$$RecorderSetTransactionNum(pc) \triangleq (\text{“nTransaction”} :>$$
$$\quad \text{IF } pc[1] = \text{“LeaderProcessRequest”} \text{ THEN}$$
$$\quad\quad \text{LET } s \triangleq \text{CHOOSE } i \in Server :$$
$$\quad\quad\quad \forall\, j \in Server : Len(history'[i]) \geq Len(history'[j])$$
$$\quad\quad \text{IN } Len(history'[s])$$
$$\quad \text{ELSE } recorder[\text{“nTransaction”}])$$
$$RecorderSetMaxEpoch(pc) \triangleq (\text{“maxEpoch”} :>$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \text{IF } pc[1] = \text{“LeaderProcessFOLLOWERINFO”} \text{ THEN} \\
\qquad\qquad\qquad\qquad\qquad \text{LET } s \triangleq \text{CHOOSE } i \in Server : \\
\qquad\qquad\qquad\qquad\qquad\qquad \forall\, j \in Server : acceptedEpoch'[i] \geq acceptedEpoch'[j] \\
\qquad\qquad\qquad\qquad\qquad \text{IN} \quad acceptedEpoch'[s] \\
\qquad\qquad\qquad\qquad \text{ELSE} \quad recorder[\text{“maxEpoch”}]) \\
RecorderSetRequests(pc) \triangleq (\text{“nClientRequest”} :> \\
\qquad\qquad\qquad\qquad \text{IF } pc[1] = \text{“LeaderProcessRequest”} \text{ THEN} \\
\qquad\qquad\qquad\qquad\qquad recorder[\text{“nClientRequest”}] + 1 \\
\qquad\qquad\qquad\qquad \text{ELSE} \quad recorder[\text{“nClientRequest”}]) \\
RecorderSetPc(pc) \triangleq (\text{“pc”} :> pc) \\
RecorderSetFailure(pc) \triangleq \text{CASE } pc[1] = \text{“Timeout”} \quad \to RecorderIncTimeout \\
\qquad\qquad\qquad\qquad\qquad\quad \square \quad pc[1] = \text{“LeaderTimeout”} \quad \to RecorderIncTimeout \\
\qquad\qquad\qquad\qquad\qquad\quad \square \quad pc[1] = \text{“FollowerTimeout”} \to RecorderIncTimeout \\
\qquad\qquad\qquad\qquad\qquad\quad \square \quad \text{OTHER} \qquad\qquad\qquad\quad \to RecorderGetTimeout
\end{array}
$$

$$
\begin{array}{l}
UpdateRecorder(pc) \triangleq recorder' = RecorderSetFailure(pc) \quad @@\, RecorderSetTransactionNum(pc) \\
\qquad\qquad\qquad\qquad\qquad @@\, RecorderSetMaxEpoch(pc) \;\; @@\, RecorderSetPc(pc) \\
\qquad\qquad\qquad\qquad\qquad @@\, RecorderSetRequests(pc) \;\; @@\, recorder \\
UnchangeRecorder \triangleq \text{UNCHANGED } recorder
\end{array}
$$

$$
\begin{array}{l}
CheckParameterHelper(n,\, p,\, Comp(\_,\, \_)) \triangleq \text{IF } p \in \text{DOMAIN } Parameters \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } Comp(n,\, Parameters[p]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE TRUE} \\
CheckParameterLimit(n,\, p) \triangleq CheckParameterHelper(n,\, p,\, \text{LAMBDA } i,\, j : i < j)
\end{array}
$$

$$
\begin{array}{lll}
CheckTimeout & \triangleq & CheckParameterLimit(recorder.nTimeout, \quad \text{“MaxTimeoutFailures”}) \\
CheckTransactionNum & \triangleq & CheckParameterLimit(recorder.nTransaction, \text{“MaxTransactionNum”}) \\
CheckEpoch & \triangleq & CheckParameterLimit(recorder.maxEpoch, \quad \text{“MaxEpoch”})
\end{array}
$$

$$
CheckStateConstraints \triangleq CheckTimeout \wedge CheckTransactionNum \wedge CheckEpoch
$$

Actions about network

$$
\begin{array}{lll}
PendingFOLLOWERINFO(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = FOLLOWERINFO \\
PendingLEADERINFO(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = LEADERINFO \\
PendingACKEPOCH(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = ACKEPOCH \\
PendingNEWLEADER(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = NEWLEADER \\
PendingACKLD(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = ACKLD \\
PendingUPTODATE(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = UPTODATE \\
PendingPROPOSAL(i,\, j) & \triangleq & \wedge\, msgs[j][i] \neq \langle\rangle \\
 & & \wedge\, msgs[j][i][1].mtype = PROPOSAL
\end{array}
$$

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

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

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

$Send(i,\, j,\, m) \triangleq msgs' = [msgs \text{ EXCEPT } ![i][j] = Append(msgs[i][j],\, m)]$

$SendPackets(i,\, j,\, ms) \triangleq msgs' = [msgs \text{ EXCEPT } ![i][j] = msgs[i][j] \circ ms]$

$DiscardAndSendPackets(i,\, j,\, ms) \triangleq msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i][j] = msgs[i][j] \circ ms]$

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

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

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

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

$DiscardAndBroadcast(i,\, j,\, m) \triangleq$
$msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i] = [v \in Server \mapsto \text{IF } \land\, v \in forwarding[i]$
$\land\, v \neq i$
$\text{THEN } Append(msgs[i][v],\, m)$
$\text{ELSE } msgs[i][v]]]$

Leader broadcasts $LEADERINFO$ to all other servers in $connectingFollowers$.

$DiscardAndBroadcastLEADERINFO(i,\, j,\, m) \triangleq$
$msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i] = [v \in Server \mapsto \text{IF } \land\, v \in connecting'[i]$
$\land\, v \in learners[i]$
$\land\, v \neq i$
$\text{THEN } Append(msgs[i][v],\, m)$
$\text{ELSE } msgs[i][v]]]$

Leader broadcasts $UPTODATE$ to all other servers in $newLeaderProposal$.

$DiscardAndBroadcastUPTODATE(i,\, j,\, m) \triangleq$
$msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i] = [v \in Server \mapsto \text{IF } \land\, v \in ackldRecv'[i]$
$\land\, v \in learners[i]$
$\land\, v \neq i$
$\text{THEN } Append(msgs[i][v],\, m)$
$\text{ELSE } msgs[i][v]]]$

Combination of $Send$ and $Discard$ − discard head of $msgs[j][i]$ and add m into $msgs$.

$Reply(i,\, j,\, m) \triangleq msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i][j] = Append(msgs[i][j],\, m)]$

Shuffle input buffer.

$Clean(i,\, j) \triangleq msgs' = [msgs \text{ EXCEPT } ![j][i] = \langle\rangle,\, ![i][j] = \langle\rangle]$

$$CleanInputBuffer(i) \triangleq msgs' = [s \in Server \mapsto [v \in Server \mapsto \text{IF } v = i \text{ THEN } \langle\rangle \text{ ELSE } msgs[s][v]]]$$

$$CleanInputBufferInCluster(S) \triangleq msgs' = [s \in Server \mapsto [v \in Server \mapsto \text{IF } v \in S \text{ THEN } \langle\rangle \text{ ELSE } msgs[s][v]]]$$

Define initial values for all variables

$$\begin{aligned}
InitServerVars \triangleq\ & \wedge InitServerVarsL \\
& \wedge zabState = [s \in Server \mapsto ELECTION] \\
& \wedge acceptedEpoch = [s \in Server \mapsto 0] \\
& \wedge lastCommitted = [s \in Server \mapsto [index \mapsto 0, \\
& \qquad\qquad zxid \mapsto \langle 0, 0\rangle]] \\
& \wedge initialHistory = [s \in Server \mapsto \langle\rangle]
\end{aligned}$$

$$\begin{aligned}
InitLeaderVars \triangleq\ & \wedge InitLeaderVarsL \\
& \wedge learners = [s \in Server \mapsto \{\}] \\
& \wedge connecting = [s \in Server \mapsto \{\}] \\
& \wedge electing = [s \in Server \mapsto \{\}] \\
& \wedge ackldRecv = [s \in Server \mapsto \{\}] \\
& \wedge forwarding = [s \in Server \mapsto \{\}] \\
& \wedge tempMaxEpoch = [s \in Server \mapsto 0]
\end{aligned}$$

$$InitElectionVars \triangleq InitElectionVarsL$$

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

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

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

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

$$Init \triangleq \wedge InitServerVars$$

$\land$ *InitLeaderVars*
$\land$ *InitElectionVars*
$\land$ *InitFollowerVars*
$\land$ *InitVerifyVars*
$\land$ *InitMsgVars*
$\land$ *InitRecorder*

$ZabTurnToLeading(i) \triangleq$
$\land\ zabState' = [zabState \text{ EXCEPT } ![i] = DISCOVERY]$
$\land\ learners' = [learners \text{ EXCEPT } ![i] = \{i\}]$
$\land\ connecting' = [connecting \text{ EXCEPT } ![i] = \{i\}]$
$\land\ electing' = [electing \text{ EXCEPT } ![i] = [sid \mapsto i,$
$peerLastZxid \mapsto \langle -1, -1\rangle,$
$inQuorum \mapsto \text{TRUE}]]$
$\land\ ackldRecv' = [ackldRecv \text{ EXCEPT } ![i] = \{i\}]$
$\land\ forwarding' = [forwarding \text{ EXCEPT } ![i] = \{\}]$
$\land\ initialHistory' = [initialHistory \text{ EXCEPT } ![i] = history'[i]]$
$\land\ tempMaxEpoch' = [tempMaxEpoch \text{ EXCEPT } ![i] = acceptedEpoch[i] + 1]$

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

Fast Leader Election

$FLEReceiveNotmsg(i, j) \triangleq$
$\land\ ReceiveNotmsg(i, j)$
$\land$ UNCHANGED $\langle zabState, acceptedEpoch, lastCommitted, learners, connecting,$
$initialHistory, electing, ackldRecv, forwarding, tempMaxEpoch,$
$followerVars, verifyVars, msgs\rangle$
$\land\ UpdateRecorder(\langle$"FLEReceiveNotmsg", $i, j\rangle)$

$FLENotmsgTimeout(i) \triangleq$
$\land\ NotmsgTimeout(i)$
$\land$ UNCHANGED $\langle zabState, acceptedEpoch, lastCommitted, learners, connecting,$
$initialHistory, electing, ackldRecv, forwarding, tempMaxEpoch,$
$followerVars, verifyVars, msgs\rangle$
$\land\ UpdateRecorder(\langle$"FLENotmsgTimeout", $i\rangle)$

$FLEHandleNotmsg(i) \triangleq$
$\land\ HandleNotmsg(i)$
$\land$ LET $newState \triangleq state'[i]$
IN
$\lor\ \land\ newState = LEADING$
$\land\ ZabTurnToLeading(i)$

```
                    ∧ UNCHANGED packetsSync
                ∨ ∧ newState = FOLLOWING
                  ∧ ZabTurnToFollowing(i)
                  ∧ UNCHANGED ⟨learners, connecting, electing, ackldRecv,
                               forwarding, tempMaxEpoch⟩
                ∨ ∧ newState = LOOKING
                  ∧ UNCHANGED ⟨zabState, learners, connecting, electing, ackldRecv,
                               forwarding, tempMaxEpoch, packetsSync, initialHistory⟩
            ∧ UNCHANGED ⟨lastCommitted, acceptedEpoch, leaderAddr, verifyVars, msgs⟩
            ∧ UpdateRecorder(⟨"FLEHandleNotmsg", i⟩)

    On the premise that ReceiveVotes.HasQuorums = TRUE,
    corresponding to logic in line 1050 − 1055 in FastLeaderElection.
FLEWaitNewNotmsg(i) ≜
            ∧ WaitNewNotmsg(i)
            ∧ UNCHANGED ⟨zabState, acceptedEpoch, lastCommitted, learners, connecting,
                         electing, ackldRecv, forwarding, tempMaxEpoch, initialHistory,
                         followerVars, verifyVars, msgs⟩
            ∧ UpdateRecorder(⟨"FLEWaitNewNotmsg", i⟩)

    On the premise that ReceiveVotes.HasQuorums = TRUE,
    corresponding to logic in line 1061 − 1066 in FastLeaderElection.
FLEWaitNewNotmsgEnd(i) ≜
            ∧ WaitNewNotmsgEnd(i)
            ∧ LET newState ≜ state'[i]
              IN
                ∨ ∧ newState = LEADING
                  ∧ ZabTurnToLeading(i)
                  ∧ UNCHANGED packetsSync
                ∨ ∧ newState = FOLLOWING
                  ∧ ZabTurnToFollowing(i)
                  ∧ UNCHANGED ⟨learners, connecting, electing, ackldRecv, forwarding,
                               tempMaxEpoch⟩
                ∨ ∧ newState = LOOKING
                  ∧ PrintT("Note: New state is LOOKING in FLEWaitNewNotmsgEnd," ∘
                        " which should not happen.")
                  ∧ UNCHANGED ⟨zabState, learners, connecting, electing, ackldRecv,
                               forwarding, tempMaxEpoch, initialHistory, packetsSync⟩
            ∧ UNCHANGED ⟨lastCommitted, acceptedEpoch, leaderAddr, verifyVars, msgs⟩
            ∧ UpdateRecorder(⟨"FLEWaitNewNotmsgEnd", i⟩)
```

```
InitialVotes ≜ [vote    ↦ InitialVote,
                round   ↦ 0,
                state   ↦ LOOKING,
                version ↦ 0]
```

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

$$
\begin{aligned}
&ZabTimeoutInCluster(S) \triangleq \\
&\quad \land state' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } LOOKING \text{ ELSE } state[s]] \\
&\quad \land lastProcessed' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } InitLastProcessed(s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } lastProcessed[s]] \\
&\quad \land logicalClock' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } logicalClock[s] + 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } logicalClock[s]] \\
&\quad \land currentVote' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN} \\
&\qquad\qquad\qquad\qquad\qquad\qquad [proposedLeader \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ proposedZxid \mapsto lastProcessed'[s].zxid, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ proposedEpoch \mapsto currentEpoch[s]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } currentVote[s]] \\
&\quad \land receiveVotes' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } [v \in Server \mapsto InitialVotes] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } receiveVotes[s]] \\
&\quad \land outOfElection' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } [v \in Server \mapsto InitialVotes] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } outOfElection[s]] \\
&\quad \land recvQueue' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } \langle [mtype \mapsto NONE] \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } recvQueue[s]] \\
&\quad \land waitNotmsg' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN FALSE ELSE } waitNotmsg[s]] \\
&\quad \land leadingVoteSet' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } \{\} \text{ ELSE } leadingVoteSet[s]] \\
&\quad \land \text{UNCHANGED } \langle electionMsgs, currentEpoch, history \rangle \\
&\quad \land zabState' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } ELECTION \text{ ELSE } zabState[s]] \\
&\quad \land leaderAddr' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } NullPoint \text{ ELSE } leaderAddr[s]] \\
&\quad \land CleanInputBufferInCluster(S)
\end{aligned}
$$

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

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

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

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

See *removeLearnerHandler* for details.

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

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

$$
\begin{aligned}
&FollowerTimeout(i) \triangleq \\
&\quad \land\ CheckTimeout \quad \text{test restrictions of } timeout\_1 \\
&\quad \land\ IsFollower(i) \\
&\quad \land\ HasNoLeader(i) \\
&\quad \land\ FollowerShutdown(i) \\
&\quad \land\ CleanInputBuffer(i) \\
&\quad \land\ \text{UNCHANGED } \langle acceptedEpoch,\ lastCommitted,\ learners,\ connecting,\ electing, \\
&\qquad\qquad\qquad ackldRecv,\ forwarding,\ tempMaxEpoch,\ initialHistory, \\
&\qquad\qquad\qquad verifyVars,\ packetsSync\rangle \\
&\quad \land\ UpdateRecorder(\langle \text{“FollowerTimeout”},\ i\rangle)
\end{aligned}
$$

Leader loses support from a quorum and turns to *LOOKING*.

$$
\begin{aligned}
&LeaderTimeout(i) \triangleq \\
&\quad \land\ CheckTimeout \quad \text{test restrictions of } timeout\_2 \\
&\quad \land\ IsLeader(i) \\
&\quad \land\ \neg IsQuorum(learners[i]) \\
&\quad \land\ LeaderShutdown(i) \\
&\quad \land\ \text{UNCHANGED } \langle acceptedEpoch,\ lastCommitted,\ connecting,\ electing,\ ackldRecv, \\
&\qquad\qquad\qquad tempMaxEpoch,\ initialHistory,\ verifyVars,\ packetsSync\rangle \\
&\quad \land\ UpdateRecorder(\langle \text{“LeaderTimeout”},\ i\rangle)
\end{aligned}
$$

Timeout between leader and follower.

$$
\begin{aligned}
&Timeout(i,\ j) \triangleq \\
&\quad \land\ CheckTimeout \quad \text{test restrictions of } timeout\_3 \\
&\quad \land\ IsLeader(i) \quad \land\ IsMyLearner(i,\ j) \\
&\quad \land\ IsFollower(j) \land\ IsMyLeader(j,\ i) \\
&\quad \text{The action of leader } i. \\
&\quad \land\ RemoveLearner(i,\ j) \\
&\quad \text{The action of follower } j. \\
&\quad \land\ FollowerShutdown(j) \\
&\quad \land\ Clean(i,\ j) \\
&\quad \land\ \text{UNCHANGED } \langle acceptedEpoch,\ lastCommitted,\ connecting,\ ackldRecv, \\
&\qquad\qquad\qquad tempMaxEpoch,\ initialHistory,\ verifyVars,\ packetsSync\rangle \\
&\quad \land\ UpdateRecorder(\langle \text{“Timeout”},\ i,\ j\rangle)
\end{aligned}
$$

Establish connection between leader and follower, containing actions like *addLearnerHandler*, *findLeader*, *connectToLeader*.

$ConnectAndFollowerSendFOLLOWERINFO(i, j) \triangleq$
$\land\ IsLeader(i) \land \neg IsMyLearner(i, j)$
$\land\ IsFollower(j) \land HasNoLeader(j) \land MyVote(j) = i$
$\land\ learners' = [learners \text{ EXCEPT } ![i] = learners[i] \cup \{j\}]$
$\land\ leaderAddr' = [leaderAddr \text{ EXCEPT } ![j] = i]$
$\land\ Send(j, leaderAddr'[j], [mtype \mapsto FOLLOWERINFO,$
$mzxid \mapsto \langle acceptedEpoch[j], 0\rangle])$
$\land$ UNCHANGED $\langle serverVars, electionVars, leadingVoteSet, connecting,$
$electing, ackldRecv, forwarding, tempMaxEpoch,$
$verifyVars, electionMsgs, packetsSync\rangle$
$\land\ UpdateRecorder(\langle$"ConnectAndFollowerSendFOLLOWERINFO", $i, j\rangle)$

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

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 $violatedInvariants$
$\lor\ \land\ zabState[i] \neq DISCOVERY$
$\land\ PrintT($"Exception: waitingFotNewEpoch true," $\circ$
" while zabState not DISCOVERY.")
$\land\ violatedInvariants' = [violatedInvariants \text{ EXCEPT } !.stateInconsistent = \text{TRUE}]$
$\land\ tempMaxEpoch' = [tempMaxEpoch \text{ EXCEPT } ![i] = \text{IF } lastAcceptedEpoch \geq tempMaxEpoch[i]$
THEN $lastAcceptedEpoch + 1$
ELSE @]
$\land\ connecting' = [connecting \text{ EXCEPT } ![i] = @ \cup \{j\}]$
$\land\ \lor\ \land\ WaitingForNewEpochTurnToFalse(i)$
$\land\ acceptedEpoch' = [acceptedEpoch \text{ EXCEPT } ![i] = tempMaxEpoch'[i]]$
$\land$ LET $newLeaderZxid \triangleq \langle acceptedEpoch'[i], 0\rangle$
$m \triangleq [mtype \mapsto LEADERINFO,$
$mzxid \mapsto newLeaderZxid]$

```
                IN  DiscardAndBroadcastLEADERINFO(i, j, m)
            ∨ ∧ ¬ WaitingForNewEpochTurnToFalse(i)
              ∧ Discard(j, i)
              ∧ UNCHANGED acceptedEpoch
        ∨   2. has broadcast LEADERINFO
          ∧ ¬ WaitingForNewEpoch(i)
          ∧ Reply(i, j, [mtype ↦ LEADERINFO,
                         mzxid ↦ ⟨acceptedEpoch[i], 0⟩])
          ∧ UNCHANGED ⟨tempMaxEpoch, connecting, acceptedEpoch, violatedInvariants⟩
    ∧ UNCHANGED ⟨state, currentEpoch, lastProcessed, zabState, history, lastCommitted,
                 followerVars, electionVars, initialHistory, leadingVoteSet, learners,
                 electing, ackldRecv, forwarding, proposalMsgsLog, epochLeader,
                 electionMsgs⟩
    ∧ UpdateRecorder(⟨"LeaderProcessFOLLOWERINFO", i, j⟩)
```

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.

```
FollowerProcessLEADERINFO(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingLEADERINFO(i, j)
    ∧ LET msg       ≜ msgs[j][i][1]
          newEpoch  ≜ msg.mzxid[1]
          infoOk    ≜ IsMyLeader(i, j)
          epochOk   ≜ newEpoch ≥ acceptedEpoch[i]
          stateOk   ≜ zabState[i] = DISCOVERY
      IN  ∧ infoOk
          ∧ ∨ 1. Normal case
              ∧ epochOk
              ∧ ∨ ∧ stateOk
                  ∧ ∨ ∧ newEpoch > acceptedEpoch[i]
                      ∧ acceptedEpoch' = [acceptedEpoch EXCEPT ![i] = newEpoch]
                      ∧ LET epochBytes ≜ currentEpoch[i]
                            m ≜ [mtype  ↦ ACKEPOCH,
                                 mzxid  ↦ lastProcessed[i].zxid,
                                 mepoch ↦ epochBytes]
                        IN  Reply(i, j, m)
                    ∨ ∧ newEpoch = acceptedEpoch[i]
                      ∧ LET m ≜ [mtype  ↦ ACKEPOCH,
                                 mzxid  ↦ lastProcessed[i].zxid,
                                 mepoch ↦ −1]
                        IN  Reply(i, j, m)
                      ∧ UNCHANGED acceptedEpoch
                  ∧ zabState' = [zabState EXCEPT ![i] = SYNCHRONIZATION]
                  ∧ UNCHANGED violatedInvariants
```

```
                    ∨ ∧ ¬stateOk
                      ∧ PrintT("Exception: Follower receives LEADERINFO," ∘
                        " whileZabState not DISCOVERY.")
                      ∧ violatedInvariants' = [violatedInvariants EXCEPT !.stateInconsistent = TRUE]
                      ∧ Discard(j, i)
                      ∧ UNCHANGED ⟨acceptedEpoch, zabState⟩
                  ∧ UNCHANGED ⟨varsL, leaderAddr, learners, forwarding, electing⟩
                ∨ 2. Abnormal case - go back to election
                  ∧ ¬epochOk
                  ∧ FollowerShutdown(i)
                  ∧ Clean(i, leaderAddr[i])
                  ∧ RemoveLearner(leaderAddr[i], i)
                  ∧ UNCHANGED ⟨acceptedEpoch, violatedInvariants⟩
            ∧ UNCHANGED ⟨history, lastCommitted, connecting, ackldRecv, tempMaxEpoch,
                         initialHistory, proposalMsgsLog, epochLeader, packetsSync⟩
            ∧ UpdateRecorder(⟨"FollowerProcessLEADERINFO", i, j⟩)
```

```
RECURSIVE UpdateAckSidHelper(_, _, _, _)
UpdateAckSidHelper(his, cur, end, target) ≜
    IF cur > end THEN his
     ELSE LET curTxn ≜ [zxid   ↦ his[1].zxid,
                        value  ↦ his[1].value,
                        ackSid ↦ IF target ∈ his[1].ackSid THEN his[1].ackSid
                                 ELSE his[1].ackSid ∪ {target},
                        epoch  ↦ his[1].epoch]
          IN ⟨curTxn⟩ ∘ 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) ≜
    IF Len(his) = 0 ∨ 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 ¬Len(his): exists and appears just once.
RECURSIVE ZxidToIndexHepler(_, _, _, _)
ZxidToIndexHepler(his, zxid, cur, appeared) ≜
    IF cur > Len(his) THEN cur
     ELSE IF TxnZxidEqual(his[cur], zxid)
          THEN CASE appeared = TRUE → −1
                 □ OTHER           → Minimum({cur,
                                              ZxidToIndexHepler(his, zxid, cur + 1, TRUE)})
          ELSE ZxidToIndexHepler(his, zxid, cur + 1, appeared)

ZxidToIndex(his, zxid) ≜ IF ZxidEqual(zxid, ⟨0, 0⟩) THEN 0
```

```
                    ELSE IF Len(his) = 0 THEN 1
                         ELSE LET len ≜ Len(his)IN
                                IF ∃ idx ∈ 1 .. len : TxnZxidEqual(his[idx], zxid)
                                 THEN ZxidToIndexHepler(his, zxid, 1, FALSE)
                                 ELSE len + 1
```

Find index *idx* which meets:
history[*idx*].*zxid* ≤ *zxid* < *history*[*idx* + 1].*zxid*

```
RECURSIVE IndexOfZxidHelper(_, _, _, _)
IndexOfZxidHelper(his, zxid, cur, end) ≜
    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) ≜ 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.

$$
\begin{array}{l}
StartForwarding(i,\ j,\ lastSeenZxid,\ lastSeenIndex,\ mode,\ needRemoveHead) \triangleq \\
\quad \land \text{LET } lastCommittedIndex \triangleq \text{IF } zabState[i] = BROADCAST \\
\qquad\qquad\qquad\qquad \text{THEN } lastCommitted[i].index \\
\qquad\qquad\qquad\qquad \text{ELSE } Len(initialHistory[i]) \\
\qquad\quad lastProposedIndex \triangleq Len(history[i]) \\
\qquad\quad queue\_origin \triangleq \text{IF } lastSeenIndex \geq lastProposedIndex \\
\qquad\qquad\qquad\qquad \text{THEN } \langle\rangle \\
\qquad\qquad\qquad\qquad \text{ELSE } queuePackets(\langle\rangle,\ history[i], \\
\qquad\qquad\qquad\qquad\qquad\quad lastSeenIndex + 1,\ lastCommittedIndex, \\
\qquad\qquad\qquad\qquad\qquad\quad lastProposedIndex) \\
\qquad\quad set\_forChecking \triangleq \text{IF } lastSeenIndex \geq lastProposedIndex \\
\qquad\qquad\qquad\qquad\quad \text{THEN } \{\} \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } setPacketsForChecking(\{\},\ i, \\
\qquad\qquad\qquad\qquad\qquad\qquad acceptedEpoch[i],\ history[i], \\
\qquad\qquad\qquad\qquad\qquad\qquad lastSeenIndex + 1,\ lastProposedIndex) \\
\qquad\quad m\_trunc \triangleq [mtype \mapsto TRUNC,\ mtruncZxid \mapsto lastSeenZxid] \\
\qquad\quad m\_diff \triangleq [mtype \mapsto DIFF,\ \ mzxid \mapsto lastSeenZxid] \\
\qquad\quad newLeaderZxid \triangleq \langle acceptedEpoch[i],\ 0\rangle \\
\qquad\quad m\_newleader \triangleq [mtype \mapsto NEWLEADER, \\
\qquad\qquad\qquad\qquad mzxid \mapsto newLeaderZxid] \\
\qquad\quad queue\_toSend \triangleq \text{CASE } mode = TRUNC \to (\langle m\_trunc\rangle \circ queue\_origin) \circ \langle m\_newleader\rangle \\
\qquad\qquad\qquad\qquad\quad \square \quad \text{OTHER} \qquad\quad \to (\langle m\_diff\rangle \ \circ queue\_origin) \circ \langle m\_newleader\rangle \\
\quad \text{IN} \quad \land \lor \land needRemoveHead \\
\qquad\qquad\quad \land DiscardAndSendPackets(i,\ j,\ queue\_toSend) \\
\qquad\qquad \lor \land \neg needRemoveHead \\
\qquad\qquad\quad \land SendPackets(i,\ j,\ queue\_toSend) \\
\qquad\quad \land proposalMsgsLog' = proposalMsgsLog \cup set\_forChecking \\
\quad \land forwarding' = [forwarding \text{ EXCEPT } ![i] = @ \cup \{j\}] \\
\quad \land history' = [history \text{ EXCEPT } ![i] = UpdateAckSid(@,\ lastSeenIndex,\ j)]
\end{array}
$$

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

$$
\begin{array}{l}
SyncFollower(i,\ j,\ peerLastZxid,\ needRemoveHead) \triangleq \\
\quad \text{LET } IsPeerNewEpochZxid \triangleq peerLastZxid[2] = 0 \\
\qquad lastProcessedZxid \triangleq lastProcessed[i].zxid \\
\qquad maxCommittedLog \triangleq \text{IF } zabState[i] = BROADCAST \\
\qquad\qquad\qquad\qquad \text{THEN } lastCommitted[i].zxid \\
\qquad\qquad\qquad\qquad \text{ELSE LET } totalLen \triangleq Len(initialHistory[i]) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{IN IF } totalLen = 0 \text{ THEN } \langle 0,\ 0\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } history[i][totalLen].zxid
\end{array}
$$

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
```

$\land\ ZxidCompare(ss1.lastZxid,\ ss2.lastZxid)$

electionFinished in Leader

$ElectionFinished(i,\ set) \triangleq \land\ i \in set$
$\land\ IsQuorum(set)$

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*).

$UpdateElecting(oldSet,\ sid,\ peerLastZxid,\ inQuorum) \triangleq$
LET $sid\_electing \triangleq \{s.sid : s \in oldSet\}$
IN IF $sid \in sid\_electing$
THEN LET $old\_info \triangleq$ CHOOSE $info \in oldSet : info.sid = sid$
$follower\_info \triangleq$
$[sid \mapsto sid,$
$peerLastZxid \mapsto peerLastZxid,$
$inQuorum \mapsto (inQuorum \lor old\_info.inQuorum)]$
IN $(oldSet \setminus \{old\_info\}) \cup \{follower\_info\}$
ELSE LET $follower\_info \triangleq$
$[sid \mapsto sid,$
$peerLastZxid \mapsto peerLastZxid,$
$inQuorum \mapsto inQuorum]$
IN $oldSet \cup \{follower\_info\}$

$LeaderTurnToSynchronization(i) \triangleq$
$\land\ currentEpoch' = [currentEpoch$ EXCEPT $![i] = acceptedEpoch[i]]$
$\land\ zabState' = [zabState$ 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.

$LeaderProcessACKEPOCH(i,\ j) \triangleq$
$\land\ IsLeader(i)$
$\land\ PendingACKEPOCH(i,\ j)$
$\land$ LET $msg \triangleq msgs[j][i][1]$
$infoOk \triangleq IsMyLearner(i,\ j)$
$leaderStateSummary \triangleq [currentEpoch \mapsto currentEpoch[i],$
$lastZxid \mapsto lastProcessed[i].zxid]$
$followerStateSummary \triangleq [currentEpoch \mapsto msg.mepoch,$
$lastZxid \mapsto msg.mzxid]$
$logOk \triangleq$ whether follower is no more up-to-date than leader
$\neg IsMoreRecentThan(followerStateSummary,\ leaderStateSummary)$
$electing\_quorum \triangleq \{e \in electing[i] : e.inQuorum =$ TRUE$\}$
$sid\_electing \triangleq \{s.sid : s \in electing\_quorum\}$

```
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) ≜ i ∈ ackldRecv'[i] ∧ IsQuorum(ackldRecv'[i])

UpdateElectionVote(i, epoch) ≜ UpdateProposal(i, currentVote[i].proposedLeader,
                                              currentVote[i].proposedZxid, epoch)

 See startZkServer in Leader for details.
StartZkServer(i) ≜
    LET latest ≜ LastProposed(i)
    IN  ∧ lastCommitted' = [lastCommitted EXCEPT ![i] = latest]
        ∧ lastProcessed' = [lastProcessed EXCEPT ![i] = latest]
        ∧ UpdateElectionVote(i, acceptedEpoch[i])

LeaderTurnToBroadcast(i) ≜
    ∧ StartZkServer(i)
    ∧ 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) ≜
    ∧ IsLeader(i)
    ∧ PendingACKLD(i, j)
    ∧ LET msg     ≜ msgs[j][i][1]
          infoOk  ≜ IsMyLearner(i, j)
          match   ≜ ZxidEqual(msg.mzxid, ⟨acceptedEpoch[i], 0⟩)
          currentZxid ≜ ⟨acceptedEpoch[i], 0⟩
          m_uptodate ≜ [mtype ↦ UPTODATE,
                        mzxid ↦ currentZxid]  not important
      IN  ∧ infoOk
          ∧ ∨  just reply UPTODATE.
            ∧ QuorumFormed(i)
```

```
                    ∧ Reply(i, j, m_uptodate)
                    ∧ UNCHANGED ⟨ackldRecv, zabState, lastCommitted, lastProcessed,
                                 currentVote, violatedInvariants⟩
                ∨ ∧ ¬QuorumFormed(i)
                  ∧ ∨ ∧ match
                      ∧ ackldRecv' = [ackldRecv EXCEPT ![i] = @ ∪ {j}]
                      ∧ ∨ jump out of waitForNewLeaderAck, and do startZkServer,
                          setZabState, and reply UPTODATE.
                        ∧ QuorumFormedTurnToTrue(i)
                        ∧ LeaderTurnToBroadcast(i)
                        ∧ DiscardAndBroadcastUPTODATE(i, j, m_uptodate)
                        ∨ still wait in waitForNewLeaderAck.
                        ∧ ¬QuorumFormedTurnToTrue(i)
                        ∧ Discard(j, i)
                        ∧ UNCHANGED ⟨zabState, lastCommitted, lastProcessed, currentVote⟩
                      ∧ 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  ∧ infoOk
              Here we ignore ack of UPTODATE.
              ∧ UpdateElectionVote(i, acceptedEpoch[i])
              ∧ FollowerLogRequestInBatches(i, j, ms_ack, packetsNotCommitted)
              ∧ FollowerCommitInBatches(i)
              ∧ packetsSync' = [packetsSync EXCEPT ![i].notCommitted = ⟨⟩,
                                                   ![i].committed = ⟨⟩]
              ∧ zabState' = [zabState EXCEPT ![i] = BROADCAST]
        ∧ UNCHANGED ⟨state, currentEpoch, acceptedEpoch, logicalClock,
              receiveVotes, outOfElection, recvQueue, waitNotmsg, leaderVars,
              initialHistory, leaderAddr, epochLeader, proposalMsgsLog, electionMsgs⟩
        ∧ UpdateRecorder(⟨"FollowerProcessUPTODATE", i, j⟩)
```

```
IncZxid(s, zxid) ≜ IF currentEpoch[s] = zxid[1] THEN ⟨zxid[1], zxid[2] + 1⟩
                   ELSE ⟨currentEpoch[s], 1⟩
```

Leader receives client propose and broadcasts *PROPOSAL*. See *processRequest* in *ProposalRequestProcessor* and propose in Leader for details. Since *prosalProcessor.processRequest* → *syncProcessor.processRequest* → *ackProcessor.processRequest* → *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) ≜
        ∧ CheckTransactionNum  test restrictions of transaction num
        ∧ IsLeader(i)
        ∧ zabState[i] = BROADCAST
        ∧ LET request_value ≜ GetRecorder("nClientRequest")  unique value
```

```
        newTxn ≜ [zxid    ↦ IncZxid(i, LastProposed(i).zxid),
                  value   ↦ request_value,
                  ackSid  ↦ {i},   assume we have done leader.processAck
                  epoch   ↦ acceptedEpoch[i]]
        m_proposal ≜ [mtype ↦ PROPOSAL,
                      mzxid ↦ newTxn.zxid,
                      mdata ↦ request_value]
        m_proposal_for_checking ≜ [source ↦ i,
                                   epoch  ↦ acceptedEpoch[i],
                                   zxid   ↦ newTxn.zxid,
                                   data   ↦ request_value]
    IN  ∧ history' = [history EXCEPT ![i] = Append(@, newTxn)]
        ∧ Broadcast(i, m_proposal)
        ∧ proposalMsgsLog' = proposalMsgsLog ∪ {m_proposal_for_checking}
  ∧ UNCHANGED ⟨state, currentEpoch, lastProcessed, zabState, acceptedEpoch,
               lastCommitted, electionVars, leaderVars, followerVars, initialHistory,
               epochLeader, violatedInvariants, electionMsgs⟩
  ∧ UpdateRecorder(⟨"LeaderProcessRequest", i⟩)

Follower processes PROPOSAL in BROADCAST. See processPacket in Follower for details.
FollowerProcessPROPOSAL(i, j) ≜
  ∧ IsFollower(i)
  ∧ PendingPROPOSAL(i, j)
  ∧ zabState[i] = BROADCAST
  ∧ 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  ↦ {},
                  epoch   ↦ acceptedEpoch[i]]
        m_ack  ≜ [mtype ↦ ACK,
                  mzxid ↦ msg.mzxid]
    IN  ∧ infoOk
        ∧ ∧ ∨ ∧ isNext
              ∧ UNCHANGED violatedInvariants
            ∨ ∧ ¬isNext
              ∧ PrintT("Exception: Follower receives PROPOSAL, while" ∘
                  " the transaction is not the next.")
              ∧ violatedInvariants' = [violatedInvariants EXCEPT
                      !.proposalInconsistent = TRUE]
        ∧ history' = [history EXCEPT ![i] = Append(@, newTxn)]
        ∧ Reply(i, j, m_ack)
  ∧ UNCHANGED ⟨state, currentEpoch, lastProcessed, zabState, acceptedEpoch,
               lastCommitted, electionVars, leaderVars, followerVars, initialHistory,
```

```
                epochLeader, proposalMsgsLog, electionMsgs⟩
        ∧ UpdateRecorder(⟨"FollowerProcessPROPOSAL", i, j⟩)

 See outstandingProposals in Leader
OutstandingProposals(i) ≜ IF zabState[i] ≠ BROADCAST THEN ⟨⟩
                          ELSE SubSeq(history[i], lastCommitted[i].index + 1,
                                      Len(history[i]))

LastAckIndexFromFollower(i, j) ≜
    LET set_index ≜ {idx ∈ 1 .. Len(history[i]) : j ∈ history[i][idx].ackSid}
    IN  Maximum(set_index)

 See commit in Leader for details.
LeaderCommit(s, follower, index, zxid) ≜
    ∧ lastCommitted' = [lastCommitted EXCEPT ![s] = [index ↦ index,
                                                     zxid  ↦ zxid]]
    ∧ LET m_commit ≜ [mtype ↦ COMMIT,
                      mzxid ↦ zxid]
      IN  DiscardAndBroadcast(s, follower, m_commit)

 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,
```

$electionVars$, $leaderVars$, $initialHistory$, $followerVars$,
$proposalMsgsLog$, $epochLeader$, $electionMsgs\rangle$
$\land\ UpdateRecorder(\langle$"FollowerProcessCOMMIT", i, $j\rangle)$

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

$FilterNonexistentMessage(i) \triangleq$
$\land \exists j \in Server \setminus \{i\} : \land msgs[j][i] \neq \langle\rangle$
$\land$ LET $msg \triangleq msgs[j][i][1]$
IN
$\lor \land IsLeader(i)$
$\land$ LET $infoOk \triangleq IsMyLearner(i, j)$
IN
$\lor msg.mtype = LEADERINFO$
$\lor msg.mtype = NEWLEADER$
$\lor msg.mtype = UPTODATE$
$\lor msg.mtype = PROPOSAL$
$\lor msg.mtype = COMMIT$
$\lor \land \neg infoOk$
$\land \lor msg.mtype = FOLLOWERINFO$
$\lor msg.mtype = ACKEPOCH$
$\lor msg.mtype = ACKLD$
$\lor msg.mtype = ACK$
$\lor \land IsFollower(i)$
$\land$ LET $infoOk \triangleq IsMyLeader(i, j)$
IN
$\lor msg.mtype = FOLLOWERINFO$
$\lor msg.mtype = ACKEPOCH$
$\lor msg.mtype = ACKLD$
$\lor msg.mtype = ACK$
$\lor \land \neg infoOk$
$\land \lor msg.mtype = LEADERINFO$
$\lor msg.mtype = NEWLEADER$
$\lor msg.mtype = UPTODATE$
$\lor msg.mtype = PROPOSAL$
$\lor msg.mtype = COMMIT$
$\lor IsLooking(i)$
$\land Discard(j, i)$
$\land violatedInvariants' = [violatedInvariants$ EXCEPT $!.messageIllegal =$ TRUE$]$
$\land$ UNCHANGED $\langle serverVars, electionVars, leaderVars,$
$followerVars, proposalMsgsLog, epochLeader, electionMsgs\rangle$
$\land UnchangeRecorder$

Defines how the variables may transition.

$Next \triangleq$

FLE modlue

$\lor \exists\, i, j \in Server : FLEReceiveNotmsg(i, j)$
$\lor \exists\, i \in Server : \quad FLENotmsgTimeout(i)$
$\lor \exists\, i \in Server : \quad FLEHandleNotmsg(i)$
$\lor \exists\, i \in Server : \quad FLEWaitNewNotmsg(i)$
$\lor \exists\, i \in Server : \quad FLEWaitNewNotmsgEnd(i)$

Some conditions like failure, network delay

$\lor \exists\, i \in Server : \quad FollowerTimeout(i)$
$\lor \exists\, i \in Server : \quad LeaderTimeout(i)$
$\lor \exists\, i, j \in Server : Timeout(i, j)$

Zab module - Discovery and Synchronization part

$\lor \exists\, i, j \in Server : ConnectAndFollowerSendFOLLOWERINFO(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessFOLLOWERINFO(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessLEADERINFO(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACKEPOCH(i, j)$
$\lor \exists\, i \in Server : \quad LeaderSyncFollower(i)$
$\lor \exists\, i, j \in Server : FollowerProcessSyncMessage(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessPROPOSALInSync(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessCOMMITInSync(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessNEWLEADER(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACKLD(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessUPTODATE(i, j)$

Zab module – *Broadcast* part

$\lor \exists\, i \in Server : \quad LeaderProcessRequest(i)$
$\lor \exists\, i, j \in Server : FollowerProcessPROPOSAL(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACK(i, j)$ *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.

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

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

$$
\begin{array}{l}
Integrity \triangleq \forall\, i \in Server : \\
\quad \land\ IsFollower(i) \\
\quad \land\ lastCommitted[i].index > 0 \\
\quad \Rightarrow \forall\, idx \in 1\,..\,lastCommitted[i].index : \exists\, proposal \in proposalMsgsLog : \\
\qquad \text{LET } txn\_proposal \triangleq [zxid \mapsto proposal.zxid, \\
\qquad\qquad\qquad\qquad\quad value \mapsto proposal.data] \\
\qquad \text{IN} \quad TxnEqual(history[i][idx],\ txn\_proposal)
\end{array}
$$

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

$$
\begin{array}{l}
Agreement \triangleq \forall\, i, j \in Server : \\
\quad \land\ IsFollower(i) \land lastCommitted[i].index > 0 \\
\quad \land\ IsFollower(j) \land lastCommitted[j].index > 0 \\
\quad \Rightarrow \\
\quad \forall\, idx1 \in 1\,..\,lastCommitted[i].index,\ idx2 \in 1\,..\,lastCommitted[j].index : \\
\qquad \lor \exists\, idx\_j \in 1\,..\,lastCommitted[j].index : \\
\qquad\quad TxnEqual(history[j][idx\_j],\ history[i][idx1]) \\
\qquad \lor \exists\, idx\_i \in 1\,..\,lastCommitted[i].index : \\
\qquad\quad TxnEqual(history[i][idx\_i],\ history[j][idx2])
\end{array}
$$

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

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

Local primary order: If a primary broadcasts a before it broadcasts b, then a follower that

delivers b must also deliver a before b.

$$
\begin{aligned}
LocalPrimaryOrder \triangleq\ & \text{LET } p\_set(i,\ e) \triangleq \{p \in proposalMsgsLog : \land\ p.source = i \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\ p.epoch\ = e\} \\
& \qquad txn\_set(i,\ e) \triangleq \{[zxid\ \mapsto p.zxid, \\
& \qquad\qquad\qquad\qquad\qquad\qquad value \mapsto p.data] : p \in p\_set(i,\ e)\} \\
& \text{IN}\quad \forall\, i \in Server : \forall\, e \in 1\,..\,currentEpoch[i] : \\
& \quad \lor\ Cardinality(txn\_set(i,\ e)) < 2 \\
& \quad \lor\ \land\ Cardinality(txn\_set(i,\ e)) \geq 2 \\
& \qquad \land \exists\, txn1,\ txn2 \in txn\_set(i,\ e) : \\
& \qquad\ \lor\ TxnEqual(txn1,\ txn2) \\
& \qquad\ \lor\ \land \neg TxnEqual(txn1,\ txn2) \\
& \qquad\qquad \land \text{LET } TxnPre \triangleq \text{IF } ZxidCompare(txn1.zxid,\ txn2.zxid) \text{ THEN } txn2 \text{ ELSE} \\
& \qquad\qquad\qquad\ TxnNext \triangleq \text{IF } ZxidCompare(txn1.zxid,\ txn2.zxid) \text{ THEN } txn1 \text{ ELSE} \\
& \qquad\qquad\quad \text{IN}\quad \forall\, j \in Server : \land\ lastCommitted[j].index \geq 2 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \land \exists\, idx \in 1\,..\,lastCommitted[j].index : \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad TxnEqual(history[j][idx],\ TxnNext) \\
& \qquad\qquad\qquad\qquad \Rightarrow \exists\, idx2 \in 1\,..\,lastCommitted[j].index : \\
& \qquad\qquad\qquad\qquad\quad \land\ TxnEqual(history[j][idx2],\ TxnNext) \\
& \qquad\qquad\qquad\qquad\quad \land\ idx2 > 1 \\
& \qquad\qquad\qquad\qquad\quad \land \exists\, idx1 \in 1\,..\,(idx2 - 1) : \\
& \qquad\qquad\qquad\qquad\qquad\ TxnEqual(history[j][idx1],\ TxnPre)
\end{aligned}
$$

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.

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

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.

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

* Modification History
* Last modified *Mon Nov* 22 22:23:20 *CST* 2021 by Dell
* Created Sat *Oct* 23 16:05:04 *CST* 2021 by Dell