MODULE *Zab*

This is the formal specification for the *Zab* consensus algorithm, in *DSN'*2011, which means *Zab pre* − 1.0.

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

The set of servers

CONSTANT *Server*

States of server

CONSTANTS *LOOKING*, *FOLLOWING*, *LEADING*

Zab states of server

CONSTANTS *ELECTION*, *DISCOVERY*, *SYNCHRONIZATION*, *BROADCAST*

Message types

CONSTANTS *CEPOCH*, *NEWEPOCH*, *ACKEPOCH*, *NEWLEADER*, *ACKLD*, *COMMITLD*, *PROPOSE*, *A*

[*MaxTimeoutFailures*, *MaxTransactionNum*, *MaxEpoch*, *MaxRestarts*]

CONSTANT *Parameters*

$MAXEPOCH \triangleq 10$

$NullPoint \triangleq \text{CHOOSE } p : p \notin Server$

$Quorums \triangleq \{Q \in \text{SUBSET } Server : Cardinality(Q) * 2 > Cardinality(Server)\}$

Variables that all servers use.

VARIABLES *state*, State of server, in {*LOOKING*, *FOLLOWING*, *LEADING*}.

zabState, Current phase of server, in {*ELECTION*, *DISCOVERY*, *SYNCHRONIZATION*, *BROADCAST*}.

acceptedEpoch, Epoch of the last *LEADERINFO* packet accepted, namely $f.p$ in paper.

currentEpoch, Epoch of the last *NEWLEADER* packet accepted, namely $f.a$ in paper.

history, History of servers: sequence of transactions, containing: [*zxid*, value, *ackSid*, epoch].

lastCommitted Maximum index and *zxid* known to be committed, namely 'lastCommitted' in Leader. Starts from 0, and increases monotonically before restarting.

Variables only used for leader.

VARIABLES *learners*, Set of servers leader connects.

cepochRecv, Set of learners leader has received *CEPOCH* from. Set of record [*sid*, connected, epoch], where epoch means $f.p$ from followers.

ackeRecv, Set of learners leader has received *ACKEPOCH* from. Set of record [*sid*, connected, *peerLastEpoch*, *peerHistory*], to record $f.a$ and $h(f)$ from followers.

ackldRecv, Set of learners leader has received *ACKLD* from. Set of record [*sid*, connected].

sendCounter Count of *txns* leader has broadcast.

Variables only used for follower.

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

Variable representing oracle of leader.

VARIABLE *leaderOracle* Current oracle.

Variables about network channel.

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

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.

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*, *nRestart*, *nClientRequest*]

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

$leaderVars \triangleq \langle learners, cepochRecv, ackeRecv, ackldRecv, sendCounter\rangle$

$followerVars \triangleq leaderAddr$

$electionVars \triangleq leaderOracle$

$msgVars \triangleq msgs$

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

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

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

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

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

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

FALSE: $zxid1 \leq zxid2$; TRUE: $zxid1 > zxid2$
$$ZxidCompare(zxid1, zxid2) \triangleq \lor\ zxid1[1] > zxid2[1] \lor\ \land\ zxid1[1] = zxid2[1] \land\ zxid1[2] > zxid2[2]$$

$ZxidEqual(zxid1, zxid2) \triangleq zxid1[1] = zxid2[1] \land zxid1[2] = zxid2[2]$

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

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

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

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

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

$RecorderIncTimeout \triangleq RecorderIncHelper(\text{“nTimeout”})$
$RecorderGetTimeout \triangleq RecorderGetHelper(\text{“nTimeout”})$
$RecorderIncRestart \triangleq RecorderIncHelper(\text{“nRestart”})$
$RecorderGetRestart \triangleq RecorderGetHelper(\text{“nRestart”})$

$$\begin{aligned}RecorderSetTransactionNum(pc) \triangleq\ & (\text{“nTransaction”} :> \\ & \text{IF } pc[1] = \text{“LeaderProcessRequest”} \text{ THEN} \\ & \quad \text{LET } s \triangleq \text{CHOOSE } i \in Server : \\ & \qquad \forall j \in Server : Len(history'[i]) \geq Len(history'[j]) \\ & \quad \text{IN } Len(history'[s]) \\ & \text{ELSE } recorder[\text{“nTransaction”}])\end{aligned}$$

$$\begin{aligned}RecorderSetMaxEpoch(pc) \triangleq\ & (\text{“maxEpoch”} :> \\ & \text{IF } pc[1] = \text{“LeaderProcessCEPOCH”} \text{ THEN} \\ & \quad \text{LET } s \triangleq \text{CHOOSE } i \in Server : \\ & \qquad \forall j \in Server : acceptedEpoch'[i] \geq acceptedEpoch'[j]\end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \text{IN} \quad acceptedEpoch'[s] \\
\qquad\qquad\qquad\quad \text{ELSE} \quad recorder[\text{“maxEpoch”}]) \\
RecorderSetRequests(pc) \triangleq (\text{“nClientRequest”} :> \\
\qquad\qquad\qquad \text{IF } pc[1] = \text{“LeaderProcessRequest”} \text{ THEN} \\
\qquad\qquad\qquad\quad recorder[\text{“nClientRequest”}] + 1 \\
\qquad\qquad\qquad \text{ELSE} \quad recorder[\text{“nClientRequest”}]) \\
RecorderSetPc(pc) \triangleq (\text{“pc”} :> pc) \\
RecorderSetFailure(pc) \triangleq \text{CASE } pc[1] = \text{“Timeout”} \quad \to RecorderIncTimeout \mathbin{@@} RecorderGetRestart \\
\qquad\qquad\qquad\qquad\quad \square \quad pc[1] = \text{“LeaderTimeout”} \quad \to RecorderIncTimeout \mathbin{@@} RecorderGetRestart \\
\qquad\qquad\qquad\qquad\quad \square \quad pc[1] = \text{“FollowerTimeout”} \to RecorderIncTimeout \mathbin{@@} RecorderGetRestart \\
\qquad\qquad\qquad\qquad\quad \square \quad pc[1] = \text{“Restart”} \quad \to RecorderIncTimeout \mathbin{@@} RecorderIncRestart \\
\qquad\qquad\qquad\qquad\quad \square \quad \text{OTHER} \quad \to RecorderGetTimeout \mathbin{@@} RecorderGetRestart
\end{array}
$$

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

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

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

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

Actions about network

$$
\begin{array}{ll}
PendingCEPOCH(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = CEPOCH \\
PendingNEWEPOCH(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = NEWEPOCH \\
PendingACKEPOCH(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = ACKEPOCH \\
PendingNEWLEADER(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = NEWLEADER \\
PendingACKLD(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = ACKLD \\
PendingCOMMITLD(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = COMMITLD \\
PendingPROPOSE(i,\, j) & \triangleq \land msgs[j][i] \neq \langle\rangle \\
 & \qquad \land msgs[j][i][1].mtype = PROPOSE
\end{array}
$$

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

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

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

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

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

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

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.

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

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

Leader broadcasts a message *PROPOSE* to all other servers in Q.
Note: In paper, Q is fuzzy. We think servers who leader broadcasts *NEWLEADER* to should receive every *PROPOSE*. So we consider *ackeRecv* as Q.
Since we let $ackeRecv = Q$, there may exist some follower receiving *COMMIT* before *COMMITLD*, and *zxid* in *COMMIT* later than *zxid* in *COMMITLD*. To avoid this situation, if $f \in ackeRecv$ but $\notin ackldRecv$, f should not receive *COMMIT* until $f \in ackldRecv$ and receives *COMMITLD*.

$Broadcast(i,\, m) \triangleq$
LET $ackeRecv\_quorum \triangleq \{a \in ackeRecv[i] : a.connected = \text{TRUE}\}$
$sid\_ackeRecv \triangleq \{a.sid : a \in ackeRecv\_quorum\}$
IN $msgs' = [msgs \text{ EXCEPT } ![i] = [v \in Server \mapsto \text{IF } \land\, v \in sid\_ackeRecv$
$\land\, v \in learners[i]$
$\land\, v \neq i$
THEN $Append(msgs[i][v],\, m)$
ELSE $msgs[i][v]]]$

Since leader decides to broadcasts message *COMMIT* when processing *ACK*, so we need to discard *ACK* and broadcast *COMMIT*.
Here Q is *ackldRecv*, because we assume that f should not receive *COMMIT* until f receives *COMMITLD*.

$DiscardAndBroadcast(i,\, j,\, m) \triangleq$
LET $ackldRecv\_quorum \triangleq \{a \in ackldRecv[i] : a.connected = \text{TRUE}\}$
$sid\_ackldRecv \triangleq \{a.sid : a \quad \in ackldRecv\_quorum\}$
IN $msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]),$
$![i] = [v \in Server \mapsto \text{IF } \land\, v \in sid\_ackldRecv$
$\land\, v \in learners[i]$
$\land\, v \neq i$
THEN $Append(msgs[i][v],\, m)$
ELSE $msgs[i][v]]]$

Leader broadcasts *LEADERINFO* to all other servers in *cepochRecv*.

$$
\begin{array}{l}
DiscardAndBroadcastNEWEPOCH(i,\, j,\, m) \triangleq \\
\quad \text{LET } new\_cepochRecv\_quorum \triangleq \{c \in cepochRecv'[i] : c.connected = \text{TRUE}\} \\
\qquad new\_sid\_cepochRecv \triangleq \{c.sid : c \in new\_cepochRecv\_quorum\} \\
\quad \text{IN } msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]), \\
\qquad\qquad\qquad\qquad ![i] = [v \in Server \mapsto \text{IF } \land v \in new\_sid\_cepochRecv \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \in learners[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \neq i \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Append(msgs[i][v],\, m) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } msgs[i][v]]]
\end{array}
$$

Leader broadcasts *NEWLEADER* to all other servers in *ackeRecv*.

$$
\begin{array}{l}
DiscardAndBroadcastNEWLEADER(i,\, j,\, m) \triangleq \\
\quad \text{LET } new\_ackeRecv\_quorum \triangleq \{a \in ackeRecv'[i] : a.connected = \text{TRUE}\} \\
\qquad new\_sid\_ackeRecv \triangleq \{a.sid : a \in new\_ackeRecv\_quorum\} \\
\quad \text{IN } msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]), \\
\qquad\qquad\qquad\qquad ![i] = [v \in Server \mapsto \text{IF } \land v \in new\_sid\_ackeRecv \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \in learners[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \neq i \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Append(msgs[i][v],\, m) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } msgs[i][v]]]
\end{array}
$$

Leader broadcasts *COMMITLD* to all other servers in *ackldRecv*.

$$
\begin{array}{l}
DiscardAndBroadcastCOMMITLD(i,\, j,\, m) \triangleq \\
\quad \text{LET } new\_ackldRecv\_quorum \triangleq \{a \in ackldRecv'[i] : a.connected = \text{TRUE}\} \\
\qquad new\_sid\_ackldRecv \triangleq \{a.sid : a \in new\_ackldRecv\_quorum\} \\
\quad \text{IN } msgs' = [msgs \text{ EXCEPT } ![j][i] = Tail(msgs[j][i]), \\
\qquad\qquad\qquad\qquad ![i] = [v \in Server \mapsto \text{IF } \land v \in new\_sid\_ackldRecv \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \in learners[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \neq i \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Append(msgs[i][v],\, m) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } msgs[i][v]]]
\end{array}
$$

Define initial values for all variables

$$
\begin{array}{lll}
InitServerVars \triangleq & \land state & = [s \in Server \mapsto LOOKING] \\
& \land zabState & = [s \in Server \mapsto ELECTION] \\
& \land acceptedEpoch & = [s \in Server \mapsto 0] \\
& \land currentEpoch & = [s \in Server \mapsto 0] \\
& \land history & = [s \in Server \mapsto \langle\rangle] \\
& \land lastCommitted & = [s \in Server \mapsto [index \mapsto 0, \\
& & \qquad\qquad\qquad zxid \mapsto \langle 0,\, 0\rangle]]
\end{array}
$$

$$
\begin{array}{lll}
InitLeaderVars \triangleq & \land learners & = [s \in Server \mapsto \{\}] \\
& \land cepochRecv & = [s \in Server \mapsto \{\}] \\
& \land ackeRecv & = [s \in Server \mapsto \{\}] \\
& \land ackldRecv & = [s \in Server \mapsto \{\}] \\
& \land sendCounter & = [s \in Server \mapsto 0]
\end{array}
$$

$$InitFollowerVars \triangleq leaderAddr = [s \in Server \mapsto NullPoint]$$

$$InitElectionVars \triangleq leaderOracle = NullPoint$$

$$InitMsgVars \triangleq msgs = [s \in Server \mapsto [v \in Server \mapsto \langle\rangle]]$$

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

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

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

Utils in state switching

$$\begin{array}{ll} FollowerShutdown(i) \triangleq & \\ \quad \land\ state' & = [state \text{ EXCEPT } ![i] = LOOKING] \\ \quad \land\ zabState' & = [zabState \text{ EXCEPT } ![i] = ELECTION] \\ \quad \land\ leaderAddr' & = [leaderAddr \text{ EXCEPT } ![i] = NullPoint] \end{array}$$

$$\begin{array}{ll} LeaderShutdown(i) \triangleq & \\ \quad \land \text{ LET } S \triangleq learners[i] & \\ \quad \text{ IN } \land\ state' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } LOOKING \text{ ELSE } state[s]] & \\ \qquad \land\ zabState' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } ELECTION \text{ ELSE } zabState[s]] & \\ \qquad \land\ leaderAddr' = [s \in Server \mapsto \text{IF } s \in S \text{ THEN } NullPoint \text{ ELSE } leaderAddr[s]] & \\ \qquad \land\ CleanInputBuffer(S) & \\ \quad \land\ learners' & = [learners \text{ EXCEPT } ![i] = \{\}] \end{array}$$

$$\begin{array}{ll} SwitchToFollower(i) \triangleq & \\ \quad \land\ state' = [state \text{ EXCEPT } ![i] = FOLLOWING] & \\ \quad \land\ zabState' = [zabState \text{ EXCEPT } ![i] = DISCOVERY] & \end{array}$$

$$
\begin{array}{l}
SwitchToLeader(i) \triangleq \\
\quad \land state' = [state \text{ EXCEPT } ![i] = LEADING] \\
\quad \land zabState' = [zabState \text{ EXCEPT } ![i] = DISCOVERY] \\
\quad \land learners' = [learners \text{ EXCEPT } ![i] = \{i\}] \\
\quad \land cepochRecv' = [cepochRecv \text{ EXCEPT } ![i] = \{[sid \mapsto i, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad connected \mapsto \text{TRUE}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad epoch \mapsto acceptedEpoch[i]]\}] \\
\quad \land ackeRecv' = [ackeRecv \text{ EXCEPT } ![i] = \{[sid \mapsto i, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad connected \mapsto \text{TRUE}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad peerLastEpoch \mapsto currentEpoch[i], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad peerHistory \mapsto history[i]]\}] \\
\quad \land ackldRecv' = [ackldRecv \text{ EXCEPT } ![i] = \{[sid \mapsto i, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad connected \mapsto \text{TRUE}]\}] \\
\quad \land sendCounter' = [sendCounter \text{ EXCEPT } ![i] = 0]
\end{array}
$$

$$
\begin{array}{l}
RemoveCepochRecv(set, sid) \triangleq \\
\quad \text{LET } sid\_cepochRecv \triangleq \{s.sid : s \in set\} \\
\quad \text{IN IF } sid \notin sid\_cepochRecv \text{ THEN } set \\
\qquad \text{ELSE LET } info \triangleq \text{CHOOSE } s \in set : s.sid = sid \\
\qquad\qquad new\_info \triangleq [sid \mapsto sid, \\
\qquad\qquad\qquad\qquad\quad connected \mapsto \text{FALSE}, \\
\qquad\qquad\qquad\qquad\quad epoch \mapsto info.epoch] \\
\qquad\qquad \text{IN } (set \setminus \{info\}) \cup \{new\_info\}
\end{array}
$$

$$
\begin{array}{l}
RemoveAckeRecv(set, sid) \triangleq \\
\quad \text{LET } sid\_ackeRecv \triangleq \{s.sid : s \in set\} \\
\quad \text{IN IF } sid \notin sid\_ackeRecv \text{ THEN } set \\
\qquad \text{ELSE LET } info \triangleq \text{CHOOSE } s \in set : s.sid = sid \\
\qquad\qquad new\_info \triangleq [sid \mapsto sid, \\
\qquad\qquad\qquad\qquad\quad connected \mapsto \text{FALSE}, \\
\qquad\qquad\qquad\qquad\quad peerLastEpoch \mapsto info.peerLastEpoch, \\
\qquad\qquad\qquad\qquad\quad peerHistory \mapsto info.peerHistory] \\
\qquad\qquad \text{IN } (set \setminus \{info\}) \cup \{new\_info\}
\end{array}
$$

$$
\begin{array}{l}
RemoveAckldRecv(set, sid) \triangleq \\
\quad \text{LET } sid\_ackldRecv \triangleq \{s.sid : s \in set\} \\
\quad \text{IN IF } sid \notin sid\_ackldRecv \text{ THEN } set \\
\qquad \text{ELSE LET } info \triangleq \text{CHOOSE } s \in set : s.sid = sid \\
\qquad\qquad new\_info \triangleq [sid \mapsto sid, \\
\qquad\qquad\qquad\qquad\quad connected \mapsto \text{FALSE}] \\
\qquad\qquad \text{IN } (set \setminus \{info\}) \cup \{new\_info\}
\end{array}
$$

$$
\begin{array}{l}
RemoveLearner(i, j) \triangleq \\
\quad \land learners' = [learners \text{ EXCEPT } ![i] = @ \setminus \{j\}] \\
\quad \land cepochRecv' = [cepochRecv \text{ EXCEPT } ![i] = RemoveCepochRecv(@, j)] \\
\quad \land ackeRecv' = [ackeRecv \text{ EXCEPT } ![i] = RemoveAckeRecv(@, j)]
\end{array}
$$

$\land\ ackldRecv' = [ackldRecv \ \text{EXCEPT}\ ![i] = RemoveAckldRecv(@, j)]$

Actions of abnormal situations and election

$UpdateLeader(i) \triangleq$
$\land\ IsLooking(i)$
$\land\ leaderOracle \neq i$
$\land\ leaderOracle' = i$
$\land\ SwitchToLeader(i)$
$\land$ UNCHANGED $\langle acceptedEpoch, currentEpoch, history, lastCommitted, followerVars, verifyVars, msgVars\rangle$
$\land\ UpdateRecorder(\langle$"UpdateLeader", $i\rangle)$

$FollowLeader(i) \triangleq$
$\land\ IsLooking(i)$
$\land\ leaderOracle \neq NullPoint$
$\land\ \lor\ \land\ leaderOracle = i$
$\land\ SwitchToLeader(i)$
$\lor\ \land\ leaderOracle \neq i$
$\land\ SwitchToFollower(i)$
$\land$ UNCHANGED $leaderVars$
$\land$ UNCHANGED $\langle acceptedEpoch, currentEpoch, history, lastCommitted, electionVars, followerVars, verifyVars, msgVars\rangle$
$\land\ UpdateRecorder(\langle$"FollowLeader", $i\rangle)$

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 $\langle acceptedEpoch, currentEpoch, history, lastCommitted, leaderVars, electionVars, verifyVars\rangle$
$\land\ UpdateRecorder(\langle$"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 $\langle acceptedEpoch, currentEpoch, history, lastCommitted, cepochRecv, ackeRecv, ackldRecv, sendCounter, electionVars, verifyVars\rangle$
$\land\ UpdateRecorder(\langle$"LeaderTimeout", $i\rangle)$

```
Timeout between leader and follower.
Timeout(i, j) ≜
        ∧ CheckTimeout  test restrictions of timeout_3
        ∧ IsLeader(i)    ∧ IsMyLearner(i, j)
        ∧ IsFollower(j) ∧ IsMyLeader(j, i)
        ∧ RemoveLearner(i, j)
        ∧ FollowerShutdown(j)
        ∧ Clean(i, j)
        ∧ UNCHANGED ⟨acceptedEpoch, currentEpoch, history, lastCommitted,
                sendCounter, electionVars, verifyVars⟩
        ∧ UpdateRecorder(⟨"Timeout", i, j⟩)

Restart(i) ≜
        ∧ CheckRestart   test restrictions of restart
        ∧ ∨ ∧ IsLooking(i)
            ∧ UNCHANGED ⟨state, zabState, learners, followerVars, msgVars,
                  cepochRecv, ackeRecv, ackldRecv⟩
          ∨ ∧ IsFollower(i)
            ∧ LET connectedWithLeader ≜ HasLeader(i)
              IN   ∨ ∧ connectedWithLeader
                     ∧ LET leader ≜ leaderAddr[i]
                       IN
                        ∧ FollowerShutdown(i)
                        ∧ RemoveLearner(leader, i)
                        ∧ Clean(leader, i)
                   ∨ ∧ ¬connectedWithLeader
                     ∧ FollowerShutdown(i)
                     ∧ CleanInputBuffer({i})
                     ∧ UNCHANGED ⟨learners, cepochRecv, ackeRecv, ackldRecv⟩
          ∨ ∧ IsLeader(i)
            ∧ LeaderShutdown(i)
            ∧ UNCHANGED ⟨cepochRecv, ackeRecv, ackldRecv⟩
        ∧ lastCommitted' = [lastCommitted EXCEPT ![i] = [index ↦ 0,
                                                         zxid  ↦ ⟨0, 0⟩]]
        ∧ UNCHANGED ⟨acceptedEpoch, currentEpoch, history,
                sendCounter, leaderOracle, verifyVars⟩
        ∧ UpdateRecorder(⟨"Restart", i⟩)
```

```
Establish connection between leader and follower.
ConnectAndFollowerSendCEPOCH(i, j) ≜
        ∧ IsLeader(i) ∧ ¬IsMyLearner(i, j)
        ∧ IsFollower(j) ∧ HasNoLeader(j) ∧ leaderOracle = i
        ∧ learners'    = [learners   EXCEPT ![i]  = @ ∪ {j}]
        ∧ leaderAddr' = [leaderAddr EXCEPT ![j] = i]
        ∧ Send(j, i, [mtype  ↦ CEPOCH,
```

```
                    mepoch ↦ acceptedEpoch[j]])  contains f.p
        ∧ UNCHANGED ⟨serverVars, electionVars, verifyVars, cepochRecv,
                     ackeRecv, ackldRecv, sendCounter⟩
        ∧ UpdateRecorder(⟨"ConnectAndFollowerSendCEPOCH", i, j⟩)

CepochRecvQuorumFormed(i) ≜ LET sid_cepochRecv ≜ {c.sid : c ∈ cepochRecv[i]}
                            IN  IsQuorum(sid_cepochRecv)
CepochRecvBecomeQuorum(i) ≜ LET sid_cepochRecv ≜ {c.sid : c ∈ cepochRecv'[i]}
                            IN  IsQuorum(sid_cepochRecv)

UpdateCepochRecv(oldSet, sid, peerEpoch) ≜
    LET sid_set ≜ {s.sid : s ∈ oldSet}
    IN  IF sid ∈ sid_set
        THEN LET old_info ≜ CHOOSE info ∈ oldSet : info.sid = sid
                 new_info ≜ [sid       ↦ sid,
                             connected ↦ TRUE,
                             epoch     ↦ peerEpoch]
             IN  (oldSet \ {old_info}) ∪ {new_info}
        ELSE LET follower_info ≜ [sid       ↦ sid,
                                  connected ↦ TRUE,
                                  epoch     ↦ peerEpoch]
             IN  oldSet ∪ {follower_info}

 Determine new e' in this round from a quorum of CEPOCH.
DetermineNewEpoch(i) ≜
    LET epoch_cepochRecv ≜ {c.epoch : c ∈ cepochRecv'[i]}
    IN  Maximum(epoch_cepochRecv) + 1

 Leader waits for receiving FOLLOWERINFO from a quorum including itself, and chooses a
 new epoch e' as its own epoch and broadcasts NEWEPOCH.
LeaderProcessCEPOCH(i, j) ≜
    ∧ CheckEpoch   test restrictions of max epoch
    ∧ IsLeader(i)
    ∧ PendingCEPOCH(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
      IN  ∧ infoOk
          ∧ ∨  1. has not broadcast NEWEPOCH
               ∧ ¬CepochRecvQuorumFormed(i)
               ∧ ∨ ∧ zabState[i] = DISCOVERY
                   ∧ UNCHANGED violatedInvariants
                 ∨ ∧ zabState[i] ≠ DISCOVERY
                   ∧ PrintT("Exception: CepochRecvQuorumFormed false," ∘
                       " while zabState not DISCOVERY.")
                   ∧ violatedInvariants' = [violatedInvariants
                                            EXCEPT !.stateInconsistent = TRUE]
```

```
                    ∧ cepochRecv' = [cepochRecv EXCEPT ![i] = UpdateCepochRecv(@, j, msg.mepoch)]
                    ∧ ∨ 1.1. cepochRecv becomes quorum,
                      then determine e' and broadcasts NEWEPOCH in Q.
                        ∧ CepochRecvBecomeQuorum(i)
                        ∧ acceptedEpoch' = [acceptedEpoch EXCEPT ![i] = DetermineNewEpoch(i)]
                        ∧ LET m ≜ [mtype   ↦ NEWEPOCH,
                                   mepoch  ↦ acceptedEpoch'[i]]
                          IN  DiscardAndBroadcastNEWEPOCH(i, j, m)
                      ∨ 1.2. cepochRecv still not quorum.
                        ∧ ¬CepochRecvBecomeQuorum(i)
                        ∧ Discard(j, i)
                        ∧ UNCHANGED acceptedEpoch
                  ∨ 2. has broadcast NEWEPOCH
                    ∧ CepochRecvQuorumFormed(i)
                    ∧ cepochRecv' = [cepochRecv EXCEPT ![i] = UpdateCepochRecv(@, j, msg.mepoch)]
                    ∧ Reply(i, j, [mtype   ↦ NEWEPOCH,
                                   mepoch  ↦ acceptedEpoch[i]])
                    ∧ UNCHANGED ⟨violatedInvariants, acceptedEpoch⟩
            ∧ UNCHANGED ⟨state, zabState, currentEpoch, history, lastCommitted, learners,
                         ackeRecv, ackldRecv, sendCounter, followerVars,
                         electionVars, proposalMsgsLog, epochLeader⟩
            ∧ UpdateRecorder(⟨"LeaderProcessCEPOCH", i, j⟩)

Follower receives LEADERINFO. If newEpoch ≥ acceptedEpoch, then follower updates
acceptedEpoch and sends ACKEPOCH back, containing currentEpoch and history. After this,
zabState turns to SYNC.

FollowerProcessNEWEPOCH(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingNEWEPOCH(i, j)
    ∧ LET msg       ≜ msgs[j][i][1]
          infoOk    ≜ IsMyLeader(i, j)
          stateOk   ≜ zabState[i] = DISCOVERY
          epochOk   ≜ msg.mepoch ≥ acceptedEpoch[i]
      IN  ∧ infoOk
          ∧ ∨ 1. Normal case
              ∧ epochOk
              ∧ ∨ ∧ stateOk
                  ∧ acceptedEpoch' = [acceptedEpoch EXCEPT ![i] = msg.mepoch]
                  ∧ LET m ≜ [mtype    ↦ ACKEPOCH,
                             mepoch   ↦ currentEpoch[i],
                             mhistory ↦ history[i]]
                    IN  Reply(i, j, m)
                  ∧ zabState' = [zabState EXCEPT ![i] = SYNCHRONIZATION]
                  ∧ UNCHANGED violatedInvariants
                ∨ ∧ ¬stateOk
```

```
                        ∧ PrintT("Exception: Follower receives NEWEPOCH," ∘
                            " whileZabState not DISCOVERY.")
                        ∧ violatedInvariants' = [violatedInvariants
                                    EXCEPT !.stateInconsistent = TRUE]
                        ∧ Discard(j, i)
                        ∧ UNCHANGED ⟨acceptedEpoch, zabState⟩
                    ∧ UNCHANGED ⟨followerVars, learners, cepochRecv, ackeRecv,
                          ackldRecv, state⟩
                ∨  2. Abnormal case - go back to election
                    ∧ ¬epochOk
                    ∧ FollowerShutdown(i)
                    ∧ LET leader ≜ leaderAddr[i]
                      IN  ∧ Clean(i, leader)
                          ∧ RemoveLearner(leader, i)
                    ∧ UNCHANGED ⟨acceptedEpoch, violatedInvariants⟩
            ∧ UNCHANGED ⟨currentEpoch, history, lastCommitted, sendCounter,
                    electionVars, proposalMsgsLog, epochLeader⟩
            ∧ UpdateRecorder(⟨"FollowerProcessNEWEPOCH", i, j⟩)

AckeRecvQuorumFormed(i) ≜ LET sid_ackeRecv ≜ {a.sid : a ∈ ackeRecv[i]}
                          IN  IsQuorum(sid_ackeRecv)
AckeRecvBecomeQuorum(i) ≜ LET sid_ackeRecv ≜ {a.sid : a ∈ ackeRecv'[i]}
                          IN  IsQuorum(sid_ackeRecv)

UpdateAckeRecv(oldSet, sid, peerEpoch, peerHistory) ≜
    LET sid_set ≜ {s.sid : s ∈ oldSet}
        follower_info ≜ [sid           ↦ sid,
                         connected     ↦ TRUE,
                         peerLastEpoch ↦ peerEpoch,
                         peerHistory   ↦ peerHistory]
    IN  IF sid ∈ sid_set
        THEN LET old_info ≜ CHOOSE info ∈ oldSet : info.sid = sid
             IN  (oldSet \ {old_info}) ∪ {follower_info}
        ELSE oldSet ∪ {follower_info}

 for checking invariants
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)
```

```
LastZxidOfHistory(his) ≜ IF Len(his) = 0 THEN ⟨0, 0⟩
                         ELSE his[Len(his)].zxid

TRUE: f1.a > f2.a or (f1.a = fa.a and f1.zxid ≥ f2.zxid)
MoreResentOrEqual(ss1, ss2) ≜ ∨ ss1.currentEpoch > ss2.currentEpoch
                              ∨ ∧ ss1.currentEpoch = ss2.currentEpoch
                                ∧ ¬ZxidCompare(ss2.lastZxid, ss1.lastZxid)

Determine initial history Ie' in this round from a quorum of ACKEPOCH.
DetermineInitialHistory(i) ≜
    LET set ≜ ackeRecv'[i]
        ss_set ≜ {[sid          ↦ a.sid,
                   currentEpoch ↦ a.peerLastEpoch,
                   lastZxid     ↦ LastZxidOfHistory(a.peerHistory)]
                   : a ∈ set}
        selected ≜ CHOOSE ss ∈ ss_set :
                     ∀ ss1 ∈ (ss_set \ {ss}) : MoreResentOrEqual(ss, ss1)
        info ≜ CHOOSE f ∈ set : f.sid = selected.sid
    IN  info.peerHistory

RECURSIVE InitAcksidHelper(_, _)
InitAcksidHelper(txns, src) ≜ IF Len(txns) = 0 THEN ⟨⟩
                              ELSE LET oldTxn ≜ txns[1]
                                       newTxn ≜ [zxid  ↦ oldTxn.zxid,
                                                 value ↦ oldTxn.value,
                                                 ackSid ↦ {src},
                                                 epoch ↦ oldTxn.epoch]
                                   IN ⟨newTxn⟩ ∘ InitAcksidHelper(Tail(txns), src)

Atomically let all txns in initial history contain self's acks.
InitAcksid(i, his) ≜ InitAcksidHelper(his, i)

Leader waits for receiving ACKEPOPCH from a quorum, and determines initialHistory according to history of whom has most recent state summary from them. After this, leader's zabState turns to SYNCHRONIZATION.
LeaderProcessACKEPOCH(i, j) ≜
    ∧ IsLeader(i)
    ∧ PendingACKEPOCH(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
      IN  ∧ infoOk
          ∧ ∨ 1. has broadcast NEWLEADER
              ∧ AckeRecvQuorumFormed(i)
              ∧ ackeRecv' = [ackeRecv EXCEPT ![i] = UpdateAckeRecv(@, j,
                                                    msg.mepoch, msg.mhistory)]
              ∧ LET toSend ≜ history[i] contains (Ie', Be')
                    m ≜ [mtype ↦ NEWLEADER,
```

```
                    mepoch   ↦ acceptedEpoch[i],
                    mhistory ↦ toSend]
            set_forChecking ≜ SetPacketsForChecking({}, i,
                    acceptedEpoch[i], toSend, 1, Len(toSend))
        IN
         ∧ Reply(i, j, m)
         ∧ proposalMsgsLog' = proposalMsgsLog ∪ set_forChecking
       ∧ UNCHANGED ⟨violatedInvariants, currentEpoch, history,
                    zabState, epochLeader⟩
   ∨  2. has not broadcast NEWLEADER
       ∧ ¬AckeRecvQuorumFormed(i)
       ∧ ∨ ∧ zabState[i] = DISCOVERY
           ∧ UNCHANGED violatedInvariants
         ∨ ∧ zabState[i] ≠ DISCOVERY
           ∧ PrintT("Exception: AckeRecvQuorumFormed false," ∘
             " while zabState not DISCOVERY.")
           ∧ violatedInvariants' = [violatedInvariants EXCEPT
                                    !.stateInconsistent = TRUE]
       ∧ ackeRecv' = [ackeRecv EXCEPT ![i] = UpdateAckeRecv(@, j,
                          msg.mepoch, msg.mhistory)]
       ∧ ∨  2.1. ackeRecv becomes quorum, determine Ie'
            and broacasts NEWLEADER in Q. (l.1.2 + l.2.1)
           ∧ AckeRecvBecomeQuorum(i)
           ∧  Update f.a
             LET newLeaderEpoch ≜ acceptedEpoch[i]IN
              ∧ currentEpoch' = [currentEpoch EXCEPT ![i] = newLeaderEpoch]
              ∧ epochLeader' = [epochLeader EXCEPT ![newLeaderEpoch]
                                = @ ∪ {i}]  for checking invariants
           ∧  Determine initial history Ie'
             LET initialHistory ≜ DetermineInitialHistory(i)IN
             history' = [history EXCEPT ![i] = InitAcksid(i, initialHistory)]
           ∧  Update zabState
             zabState' = [zabState EXCEPT ![i] = SYNCHRONIZATION]
           ∧  Broadcast NEWLEADER with (e', Ie')
             LET toSend ≜ history'[i]
                 m ≜ [mtype     ↦ NEWLEADER,
                      mepoch    ↦ acceptedEpoch[i],
                      mhistory  ↦ toSend]
                 set_forChecking ≜ SetPacketsForChecking({}, i,
                      acceptedEpoch[i], toSend, 1, Len(toSend))
             IN
              ∧ DiscardAndBroadcastNEWLEADER(i, j, m)
              ∧ proposalMsgsLog' = proposalMsgsLog ∪ set_forChecking
         ∨  2.2. ackeRecv still not quorum.
           ∧ ¬AckeRecvBecomeQuorum(i)
```

$\land\ Discard(j,\ i)$
$\land$ UNCHANGED $\langle currentEpoch,\ history,\ zabState,$
$proposalMsgsLog,\ epochLeader\rangle$
$\land$ UNCHANGED $\langle state,\ acceptedEpoch,\ lastCommitted,\ learners,\ cepochRecv,\ ackldRecv,$
$sendCounter,\ followerVars,\ electionVars\rangle$
$\land\ UpdateRecorder(\langle$"LeaderProcessACKEPOCH"$,\ i,\ j\rangle)$

Follower receives *NEWLEADER*. Update $f.a$ and history.

$FollowerProcessNEWLEADER(i,\ j)\ \triangleq$
$\land\ IsFollower(i)$
$\land\ PendingNEWLEADER(i,\ j)$
$\land$ LET $msg\ \triangleq\ msgs[j][i][1]$
$infoOk\ \triangleq\ IsMyLeader(i,\ j)$
$epochOk\ \triangleq\ acceptedEpoch[i] = msg.mepoch$
$stateOk\ \triangleq\ zabState[i] = SYNCHRONIZATION$
IN $\land\ infoOk$
$\land\ \lor$ 1. $f.p$ not equals e', starts a new iteration.
$\land\ \neg epochOk$
$\land\ FollowerShutdown(i)$
$\land$ LET $leader\ \triangleq\ leaderAddr[i]$
IN $\land\ Clean(i,\ leader)$
$\land\ RemoveLearner(leader,\ i)$
$\land$ UNCHANGED $\langle violatedInvariants,\ currentEpoch,\ history\rangle$
$\lor$ 2. $f.p$ equals e'.
$\land\ epochOk$
$\land\ \lor\ \land\ stateOk$
$\land$ UNCHANGED $violatedInvariants$
$\lor\ \land\ \neg stateOk$
$\land\ PrintT($"Exception: Follower receives NEWLEADER," $\circ$
" whileZabState not SYNCHRONIZATION.")
$\land\ violatedInvariants' = [violatedInvariants$
EXCEPT $!.stateInconsistent = $ TRUE$]$
$\land\ currentEpoch' = [currentEpoch$ EXCEPT $![i] = acceptedEpoch[i]]$
$\land\ history' = [history$ EXCEPT $![i] = msg.mhistory]$ no need to care *ackSid*
$\land$ LET $m\ \triangleq\ [mtype \mapsto ACKLD,$
$mzxid \mapsto LastZxidOfHistory(history'[i])]$
IN $Reply(i,\ j,\ m)$
$\land$ UNCHANGED $\langle followerVars,\ state,\ zabState,\ learners,\ cepochRecv,$
$ackeRecv,\ ackldRecv\rangle$
$\land$ UNCHANGED $\langle acceptedEpoch,\ lastCommitted,\ sendCounter,\ electionVars,$
$proposalMsgsLog,\ epochLeader\rangle$
$\land\ UpdateRecorder(\langle$"FollowerProcessNEWLEADER"$,\ i,\ j\rangle)$

$AckldRecvQuorumFormed(i)\ \triangleq$ LET $sid\_ackldRecv\ \triangleq\ \{a.sid : a \in ackldRecv[i]\}$
IN $IsQuorum(sid\_ackldRecv)$

```
AckldRecvBecomeQuorum(i) ≜ LET sid_ackldRecv ≜ {a.sid : a ∈ ackldRecv'[i]}
                           IN  IsQuorum(sid_ackldRecv)

UpdateAckldRecv(oldSet, sid) ≜
    LET sid_set ≜ {s.sid : s ∈ oldSet}
        follower_info ≜ [sid       ↦ sid,
                         connected ↦ TRUE]
    IN  IF sid ∈ sid_set
        THEN LET old_info ≜ CHOOSE info ∈ oldSet : info.sid = sid
             IN  (oldSet \ {old_info}) ∪ {follower_info}
        ELSE oldSet ∪ {follower_info}

LastZxid(i) ≜ LastZxidOfHistory(history[i])

RECURSIVE UpdateAcksidHelper(_, _, _)
UpdateAcksidHelper(txns, target, endZxid) ≜
    IF Len(txns) = 0 THEN ⟨⟩
    ELSE LET oldTxn ≜ txns[1]
         IN  IF ZxidCompare(oldTxn.zxid, endZxid) THEN txns
             ELSE LET newTxn ≜ [zxid    ↦ oldTxn.zxid,
                                value   ↦ oldTxn.value,
                                ackSid  ↦ IF target ∈ oldTxn.ackSid
                                          THEN oldTxn.ackSid
                                          ELSE oldTxn.ackSid ∪ {target},
                                epoch   ↦ oldTxn.epoch]
                  IN  ⟨newTxn⟩ ∘ UpdateAcksidHelper(Tail(txns), target, endZxid)

Atomically add ackSid of one learner according to zxid in ACKLD.
UpdateAcksid(his, target, endZxid) ≜ UpdateAcksidHelper(his, target, endZxid)

Leader waits for receiving ACKLD from a quorum including itself, and broadcasts COMMITLD
and turns to BROADCAST.
LeaderProcessACKLD(i, j) ≜
    ∧ IsLeader(i)
    ∧ PendingACKLD(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
      IN  ∧ infoOk
          ∧ ∨ 1. has not broadcast COMMITLD
              ∧ ¬AckldRecvQuorumFormed(i)
              ∧ ∨ ∧ zabState[i] = SYNCHRONIZATION
                  ∧ UNCHANGED violatedInvariants
                ∨ ∧ zabState[i] ≠ SYNCHRONIZATION
                  ∧ PrintT("Exception: AckldRecvQuorumFormed false," ∘
                        " while zabState not SYNCHRONIZATION.")
                  ∧ violatedInvariants' = [violatedInvariants
```

```
                                EXCEPT !.stateInconsistent = TRUE]
                    ∧ ackldRecv' = [ackldRecv EXCEPT ![i] = UpdateAckldRecv(@, j)]
                    ∧ history' = [history EXCEPT ![i] = UpdateAcksid(@, j, msg.mzxid)]
                    ∧ ∨ 1.1. ackldRecv becomes quorum,
                      then broadcasts COMMITLD and turns to BROADCAST.
                        ∧ AckldRecvBecomeQuorum(i)
                        ∧ lastCommitted' = [lastCommitted EXCEPT
                                                ![i] = [index ↦ Len(history[i]),
                                                        zxid  ↦ LastZxid(i)]]
                        ∧ zabState' = [zabState EXCEPT ![i] = BROADCAST]
                        ∧ LET m ≜ [mtype ↦ COMMITLD,
                                   mzxid ↦ LastZxid(i)]
                          IN  DiscardAndBroadcastCOMMITLD(i, j, m)
                      ∨ 1.2. ackldRecv still not quorum.
                        ∧ ¬AckldRecvBecomeQuorum(i)
                        ∧ Discard(j, i)
                        ∧ UNCHANGED ⟨zabState, lastCommitted⟩
                  ∨ 2. has broadcast COMMITLD
                    ∧ AckldRecvQuorumFormed(i)
                    ∧ ∨ ∧ zabState[i] = BROADCAST
                        ∧ UNCHANGED violatedInvariants
                      ∨ ∧ zabState[i] ≠ BROADCAST
                        ∧ PrintT("Exception: AckldRecvQuorumFormed true," ∘
                              " while zabState not BROADCAST.")
                        ∧ violatedInvariants' = [violatedInvariants
                                EXCEPT !.stateInconsistent = TRUE]
                    ∧ ackldRecv' = [ackldRecv EXCEPT ![i] = UpdateAckldRecv(@, j)]
                    ∧ history' = [history EXCEPT ![i] = UpdateAcksid(@, j, msg.mzxid)]
                    ∧ Reply(i, j, [mtype ↦ COMMITLD,
                                   mzxid ↦ lastCommitted[i].zxid])
                    ∧ UNCHANGED ⟨zabState, lastCommitted⟩
            ∧ UNCHANGED ⟨state, acceptedEpoch, currentEpoch, learners, cepochRecv, ackeRecv,
                         sendCounter, followerVars, electionVars, proposalMsgsLog, epochLeader⟩
            ∧ UpdateRecorder(⟨"LeaderProcessACKLD", i, j⟩)

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)
```

return -1: this *zxid* appears at least twice. $Len(his) + 1$: does not exist.

```
\* 1 - Len(his): exists and appears just once.
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

\* Follower receives COMMITLD. Commit all txns.
FollowerProcessCOMMITLD(i, j) ≜
    ∧ IsFollower(i)
    ∧ PendingCOMMITLD(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLeader(i, j)
          index ≜ IF ZxidEqual(msg.mzxid, ⟨0, 0⟩) THEN 0
                  ELSE ZxidToIndex(history[i], msg.mzxid)
          logOk ≜ index ≥ 0 ∧ index ≤ Len(history[i])
      IN  ∧ infoOk
          ∧ ∨ ∧ logOk
              ∧ UNCHANGED violatedInvariants
            ∨ ∧ ¬logOk
              ∧ PrintT("Exception: zxid in COMMITLD not exists in history.")
              ∧ violatedInvariants' = [violatedInvariants
                                       EXCEPT !.proposalInconsistent = TRUE]
          ∧ lastCommitted' = [lastCommitted EXCEPT ![i] = [index ↦ index,
                                                           zxid  ↦ msg.mzxid]]
          ∧ zabState' = [zabState EXCEPT ![i] = BROADCAST]
          ∧ Discard(j, i)
    ∧ UNCHANGED ⟨state, acceptedEpoch, currentEpoch, history, leaderVars,
                 followerVars, electionVars, proposalMsgsLog, epochLeader⟩
    ∧ UpdateRecorder(⟨"FollowerProcessCOMMITLD", i, j⟩)

----------

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

\* Leader receives client request. 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, LastZxid(i)),
                    value ↦ request_value,
```

```
                                ackSid ↦ {i},
                                epoch  ↦ currentEpoch[i]]
        IN    history' = [history EXCEPT ![i] = Append(@, newTxn)]
    ∧ UNCHANGED ⟨state, zabState, acceptedEpoch, currentEpoch, lastCommitted,
                 leaderVars, followerVars, electionVars, msgVars, verifyVars⟩
    ∧ UpdateRecorder(⟨"LeaderProcessRequest", i⟩)

Latest counter existing in history.
CurrentCounter(i) ≜ IF LastZxid(i)[1] = currentEpoch[i] THEN LastZxid(i)[2]
                    ELSE 0

Leader broadcasts PROPOSE when sendCounter < currentCounter.
LeaderBroadcastPROPOSE(i) ≜
    ∧ IsLeader(i)
    ∧ zabState[i] = BROADCAST
    ∧ sendCounter[i] < CurrentCounter(i)  there exists proposal to be sent
    ∧ LET toSendCounter ≜ sendCounter[i] + 1
          toSendZxid ≜ ⟨currentEpoch[i], toSendCounter⟩
          toSendIndex ≜ ZxidToIndex(history[i], toSendZxid)
          toSendTxn ≜ history[i][toSendIndex]
          m_proposal ≜ [mtype ↦ PROPOSE,
                        mzxid ↦ toSendTxn.zxid,
                        mdata ↦ toSendTxn.value]
          m_proposal_forChecking ≜ [source ↦ i,
                                    epoch  ↦ currentEpoch[i],
                                    zxid   ↦ toSendTxn.zxid,
                                    data   ↦ toSendTxn.value]
      IN  ∧ sendCounter' = [sendCounter EXCEPT ![i] = toSendCounter]
          ∧ Broadcast(i, m_proposal)
          ∧ proposalMsgsLog' = proposalMsgsLog ∪ {m_proposal_forChecking}
    ∧ UNCHANGED ⟨serverVars, learners, cepochRecv, ackeRecv, ackldRecv,
          followerVars, electionVars, epochLeader, violatedInvariants⟩
    ∧ UpdateRecorder(⟨"LeaderBroadcastPROPOSE", i⟩)

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]

Follower processes PROPOSE, saves it in history and replies ACK.
FollowerProcessPROPOSE(i, j) ≜
    ∧ IsFollower(i)
```

```
∧ PendingPROPOSE(i, j)
∧ LET msg ≜ msgs[j][i][1]
      infoOk ≜ IsMyLeader(i, j)
      isNext ≜ IsNextZxid(LastZxid(i), msg.mzxid)
      newTxn ≜ [zxid  ↦ msg.mzxid,
                value ↦ msg.mdata,
                ackSid ↦ {},
                epoch ↦ currentEpoch[i]]
      m_ack ≜ [mtype ↦ ACK,
               mzxid ↦ msg.mzxid]
  IN  ∧ infoOk
      ∧ ∨ ∧ isNext
          ∧ history' = [history EXCEPT ![i] = Append(@, newTxn)]
          ∧ Reply(i, j, m_ack)
          ∧ UNCHANGED violatedInvariants
        ∨ ∧ ¬isNext
          ∧ LET index ≜ ZxidToIndex(history[i], msg.mzxid)
                exist ≜ index > 0 ∧ index ≤ Len(history[i])
            IN  ∨ ∧ exist
                  ∧ UNCHANGED violatedInvariants
                ∨ ∧ ¬exist
                  ∧ PrintT("Exception: Follower receives PROPOSE, while" ∘
                        " txn is neither the next nor exists in history.")
                  ∧ violatedInvariants' = [violatedInvariants EXCEPT
                        !.proposalInconsistent = TRUE]
          ∧ Discard(j, i)
          ∧ UNCHANGED history
∧ UNCHANGED ⟨state, zabState, acceptedEpoch, currentEpoch, lastCommitted,
             leaderVars, followerVars, electionVars, proposalMsgsLog, epochLeader⟩
∧ UpdateRecorder(⟨"FollowerProcessPROPOSE", i, j⟩)

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⟩
```

```
        ∨ ∧ allTxnsBeforeCommitted
          ∧ hasAllQuorums
          ∧ ∨ ∧ ¬ordered
              ∧ PrintT("Warn: Committing zxid " ∘ ToString(zxid) ∘ " not first.")
              ∧ violatedInvariants' = [violatedInvariants EXCEPT
                    !.commitInconsistent = TRUE]
            ∨ ∧ ordered
              ∧ UNCHANGED violatedInvariants
          ∧ lastCommitted' = [lastCommitted EXCEPT ![s] = [index ↦ index,
                                                          zxid  ↦ zxid]]
          ∧ LET m_commit ≜ [mtype ↦ COMMIT,
                           mzxid ↦ zxid]
            IN  DiscardAndBroadcast(s, follower, m_commit)

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

Leader Keeps a count of acks for a particular proposal, and try to commit the proposal. If committed, COMMIT of proposal will be broadcast.
LeaderProcessACK(i, j) ≜
    ∧ IsLeader(i)
    ∧ PendingACK(i, j)
    ∧ LET msg ≜ msgs[j][i][1]
          infoOk ≜ IsMyLearner(i, j)
          index ≜ ZxidToIndex(history[i], msg.mzxid)
          exist ≜ index ≥ 1 ∧ index ≤ Len(history[i])  proposal exists in history
          outstanding ≜ lastCommitted[i].index < Len(history[i])  outstanding not null
          hasCommitted ≜ ¬ZxidCompare(msg.mzxid, lastCommitted[i].zxid)
          ackIndex ≜ LastAckIndexFromFollower(i, j)
          monotonicallyInc ≜ ∨ ackIndex = −1
                             ∨ ackIndex + 1 = index
      IN  ∧ infoOk
          ∧ ∨ ∧ exist
              ∧ monotonicallyInc
              ∧ LET txn ≜ history[i][index]
                    txnAfterAddAck ≜ [zxid   ↦ txn.zxid,
                                      value  ↦ txn.value,
                                      ackSid ↦ txn.ackSid ∪ {j},
                                      epoch  ↦ txn.epoch]
                IN
                ∧ history' = [history EXCEPT ![i][index] = txnAfterAddAck]
                ∧ ∨ ∧  Note: outstanding is 0.
                       / proposal has already been committed.
                  ∨ ¬outstanding
```

```
                                    ∨ hasCommitted
                                 ∧ Discard(j, i)
                                 ∧ UNCHANGED ⟨violatedInvariants, lastCommitted⟩
                              ∨ ∧ 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⟩
                  ∧ UNCHANGED ⟨state, zabState, acceptedEpoch, currentEpoch, leaderVars,
                               followerVars, electionVars, proposalMsgsLog, epochLeader⟩
                  ∧ UpdateRecorder(⟨"LeaderProcessACK", i, j⟩)

Follower processes COMMIT.
FollowerProcessCOMMIT(i, j) ≜
        ∧ IsFollower(i)
        ∧ PendingCOMMIT(i, j)
        ∧ LET msg ≜ msgs[j][i][1]
              infoOk ≜ IsMyLeader(i, j)
              pending ≜ lastCommitted[i].index < Len(history[i])
          IN  ∧ infoOk
              ∧ ∨ ∧ ¬pending
                  ∧ PrintT("Warn: Committing zxid without seeing txn.")
                  ∧ UNCHANGED ⟨lastCommitted, violatedInvariants⟩
                ∨ ∧ pending
                  ∧ LET firstElement ≜ history[i][lastCommitted[i].index + 1]
                        match ≜ ZxidEqual(firstElement.zxid, msg.mzxid)
                    IN
                     ∨ ∧ ¬match
                       ∧ PrintT("Exception: Committing zxid not equals" ∘
                             " next pending txn zxid.")
                       ∧ violatedInvariants' = [violatedInvariants EXCEPT
                             !.commitInconsistent = TRUE]
                       ∧ UNCHANGED lastCommitted
                     ∨ ∧ match
                       ∧ lastCommitted' = [lastCommitted EXCEPT ![i] =
                                           [index ↦ lastCommitted[i].index + 1,
                                            zxid  ↦ firstElement.zxid]]
                       ∧ UNCHANGED violatedInvariants
        ∧ Discard(j, i)
```

$\land$ UNCHANGED $\langle state, zabState, acceptedEpoch, currentEpoch, history, leaderVars, followerVars, electionVars, proposalMsgsLog, epochLeader\rangle$
$\land$ $UpdateRecorder(\langle$"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
$\lor msg.mtype = NEWEPOCH$
$\lor msg.mtype = NEWLEADER$
$\lor msg.mtype = COMMITLD$
$\lor msg.mtype = PROPOSE$
$\lor msg.mtype = COMMIT$
$\lor \land \neg infoOk$
$\land \lor msg.mtype = CEPOCH$
$\lor msg.mtype = ACKEPOCH$
$\lor msg.mtype = ACKLD$
$\lor msg.mtype = ACK$
$\lor \land IsFollower(i)$
$\land$ LET $infoOk \triangleq IsMyLeader(i, j)$
IN
$\lor msg.mtype = CEPOCH$
$\lor msg.mtype = ACKEPOCH$
$\lor msg.mtype = ACKLD$
$\lor msg.mtype = ACK$
$\lor \land \neg infoOk$
$\land \lor msg.mtype = NEWEPOCH$
$\lor msg.mtype = NEWLEADER$
$\lor msg.mtype = COMMITLD$
$\lor msg.mtype = PROPOSE$
$\lor msg.mtype = COMMIT$
$\lor IsLooking(i)$
$\land Discard(j, i)$
$\land violatedInvariants' = [violatedInvariants$ EXCEPT $!.messageIllegal =$ TRUE$]$
$\land$ UNCHANGED $\langle serverVars, leaderVars, followerVars, electionVars, proposalMsgsLog, epochLeader\rangle$
$\land UnchangeRecorder$

Defines how the variables may transition.
$Next \triangleq$

Election
$\lor \exists\, i \in Server : \quad UpdateLeader(i)$
$\lor \exists\, i \in Server : \quad FollowLeader(i)$
Abnormal situations like failure, network disconnection
$\lor \exists\, i \in Server : \quad FollowerTimeout(i)$
$\lor \exists\, i \in Server : \quad LeaderTimeout(i)$
$\lor \exists\, i, j \in Server : Timeout(i, j)$
$\lor \exists\, i \in Server : \quad Restart(i)$
Zab module - Discovery and Synchronization part
$\lor \exists\, i, j \in Server : ConnectAndFollowerSendCEPOCH(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessCEPOCH(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessNEWEPOCH(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACKEPOCH(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessNEWLEADER(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACKLD(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessCOMMITLD(i, j)$
Zab module – *Broadcast* part
$\lor \exists\, i \in Server : \quad LeaderProcessRequest(i)$
$\lor \exists\, i \in Server : \quad LeaderBroadcastPROPOSE(i)$
$\lor \exists\, i, j \in Server : FollowerProcessPROPOSE(i, j)$
$\lor \exists\, i, j \in Server : LeaderProcessACK(i, j)$
$\lor \exists\, i, j \in Server : FollowerProcessCOMMIT(i, j)$
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*.

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

There is most one established leader for a certain epoch.

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

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

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

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

$$\land \forall\, index \in 1\,..\,smaller :$$
$$TxnEqual(history[i][index],\, history[j][index])$$

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

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

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

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

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

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

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

$$LocalPrimaryOrder \triangleq \text{LET } p\_set(i,\, e) \triangleq \{p \in proposalMsgsLog :\ \land p.source = i$$
$$\land\ p.epoch = e\}$$
$$txn\_set(i,\, e) \triangleq \{[zxid \mapsto p.zxid,$$
$$value \mapsto p.data] : p \in p\_set(i,\, e)\}$$

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

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.

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

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.

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

\ * Modification History
\ * Last modified Sat *Dec* 11 22:31:08 *CST* 2021 by Dell
\ * Created *Thu Dec* 02 20:49:23 *CST* 2021 by Dell