\ * To fix: heartbeat

MODULE *TunableMongoDB_Repl*

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

constants and variables

CONSTANTS *Client*, *Server*, the set of clients and servers
Key, *Value*, the set of keys and values
Nil, model value, place holder
PtStop max physical time

VARIABLES *Primary*, Primary node
Secondary, secondary nodes
Oplog, *oplog*[*s*]: *oplog* at *server*[*s*]
Store, *store*[*s*]: data stored at *server*[*s*]
Ct, *Ct*[*s*]: cluster time at node *s*
Ot, *Ot*[*s*]: the last applied operation time at server *s*
ServerMsg, *ServerMsg*[*s*]: the channel of heartbeat msgs at server *s*
Pt, *Pt*[*s*]: physical time at server *s*
Cp, *Cp*[*s*]: majority commit point at server *s*
State, *State*[*s*]: the latest *Ot* of all servers that server *s* knows
CurrentTerm, *CurrentTerm*[*s*]: current election term at server *s*
→ updated in *update_position*, heartbeat and replicate
ReadyToServe, equal to 0 before any primary is elected
SyncSource *SyncSource*[*s*]: sync source of server node *s*

---

group related *vars* to optimize code

$electionVars \triangleq \langle Primary, Secondary \rangle$ *vars* that are related to election
$storageVars \triangleq \langle Oplog, Store \rangle$ *vars* that are related to storage
$messageVar \triangleq \langle ServerMsg \rangle$ var that is related to message
$serverVars \triangleq \langle Ot, SyncSource \rangle$ *vars* that each server node holds for itself
$learnableVars \triangleq \langle Ct, State, Cp, CurrentTerm \rangle$ *vars* that must learn from msgs
$timeVar \triangleq \langle Pt \rangle$ var that is used for timing
$functionalVar \triangleq \langle ReadyToServe \rangle$ var that is used for some extra function

---

ASSUME $Cardinality(Client) \geq 1$ at least one clinet
ASSUME $Cardinality(Server) \geq 2$ at least one primary and one secondary
ASSUME $Cardinality(Key) \geq 1$ at least one object
ASSUME $Cardinality(Value) \geq 2$ at least two values to update

Helpers

$HLCLt(x, y) \triangleq$ IF $x.p < y.p$
THEN TRUE
ELSE IF $x.p = y.p$

```
                THEN IF x.l < y.l
                        THEN TRUE
                     ELSE FALSE
                ELSE FALSE

HLCMin(x, y) ≜ IF HLCLt(x, y) THEN x ELSE y
HLCMax(x, y) ≜ IF HLCLt(x, y) THEN y ELSE x
HLCType ≜ [p : Nat, l : Nat]
Min(x, y) ≜ IF x < y THEN x ELSE y
Max(x, y) ≜ IF x > y THEN x ELSE y

vars ≜ ⟨Primary, Secondary, Oplog, Store, Ct, Ot, messageVar,
        Pt, Cp, State, CurrentTerm, ReadyToServe, SyncSource⟩

RECURSIVE CreateState(_, _)  init state
CreateState(len, seq) ≜
  IF len = 0 THEN seq
   ELSE CreateState(len − 1, Append(seq, [p ↦ 0, l ↦ 0]))

LogTerm(i, index) ≜ IF index = 0 THEN 0 ELSE Oplog[i][index].term
LastTerm(i) ≜ CurrentTerm[i]

 Is node i ahead of node j
NotBehind(i, j) ≜ ∨ LastTerm(i) > LastTerm(j)
                  ∨ ∧ LastTerm(i) = LastTerm(j)
                    ∧ Len(Oplog[i]) ≥ Len(Oplog[j])

IsMajority(servers) ≜ Cardinality(servers) ∗ 2 > Cardinality(Server)

 Return the maximum value from a set, or undefined if the set is empty.
MaxVal(s) ≜ CHOOSE x ∈ s : ∀ y ∈ s : x ≥ y
HLCMinSet(s) ≜ CHOOSE x ∈ s : ∀ y ∈ s : ¬HLCLt(y, x)

 clock

MaxPt ≜ LET x ≜ CHOOSE s ∈ Server : ∀ s1 ∈ Server \ {s} :
                                      Pt[s] ≥ Pt[s1]
        IN  Pt[x]

Tick(s) ≜ Ct′ = IF Ct[s].p ≥ Pt[s]
                THEN [Ct EXCEPT ![s] = [p ↦ @.p, l ↦ @.l + 1]]
                ELSE [Ct EXCEPT ![s] = [p ↦ Pt[s], l ↦ 0]]

 heartbeat
 Only Primary node sends heartbeat once advance pt
BroadcastHeartbeat(s) ≜
  LET msg ≜ [type ↦ "heartbeat", s ↦ s, aot ↦ Ot[s],
             ct ↦ Ct[s], cp  ↦ Cp[s], term ↦ CurrentTerm[s]]
```

```
    IN  ServerMsg' = [x ∈ Server ↦ IF x = s THEN ServerMsg[x]
                                    ELSE Append(ServerMsg[x], msg)]

 Can node i sync from node j?
CanSyncFrom(i, j) ≜
    ∧ Len(Oplog[i]) < Len(Oplog[j])
    ∧ LastTerm(i) = LogTerm(j, Len(Oplog[i]))

 Oplog entries needed to replicate from j to i
ReplicateOplog(i, j) ≜
    LET len_i ≜ Len(Oplog[i])
        len_j ≜ Len(Oplog[j])
    IN  IF i ∈ Secondary ∧ len_i < len_j
                THEN SubSeq(Oplog[j], len_i + 1, len_j)
                ELSE ⟨⟩

 Can node i rollback its log based on j's log
CanRollback(i, j) ≜ ∧ Len(Oplog[i]) > 0
                    ∧ Len(Oplog[j]) > 0
                    ∧ CurrentTerm[i] < CurrentTerm[j]
                    ∧
                       ∨ Len(Oplog[i]) > Len(Oplog[j])
                       ∨ ∧ Len(Oplog[i]) ≤ Len(Oplog[j])
                         ∧ CurrentTerm[i] ≠ LogTerm(j, Len(Oplog[i]))

 Returns the highest common index between two divergent logs.
 If there is no common index between the logs, returns 0.
RollbackCommonPoint(i, j) ≜
    LET commonIndices ≜ {k ∈ DOMAIN Oplog[i] :
                            ∧ k ≤ Len(Oplog[j])
                            ∧ Oplog[i][k] = Oplog[j][k]}IN
        IF commonIndices = {} THEN 0 ELSE MaxVal(commonIndices)

 The set of all quorums. This just calculates simple majorities, but the only
 important property is that every quorum overlaps with every other.
Quorum ≜ {i ∈ SUBSET (Server) : Cardinality(i) * 2 > Cardinality(Server)}

QuorumAgreeInSameTerm(states) ≜
    LET quorums ≜ {Q ∈ Quorum :
                    Make sure all nodes in quorum have actually applied some entries.
                   ∧ ∨ ∀ s ∈ Q : states[s].p > 0
                     ∨ ∧ ∀ s ∈ Q : states[s].p = 0
                       ∧ ∀ s ∈ Q : states[s].l > 0
                    Make sure every applied entry in quorum has the same term.
                   ∧ ∀ s, t ∈ Q :
                     s ≠ t ⇒ states[s].term = states[s].term}
    IN  quorums
```

Init Part

$InitPrimary \triangleq Primary = \{\text{CHOOSE } s \in Server : \text{TRUE}\}$
$InitSecondary \triangleq Secondary = Server \setminus Primary$
$InitOplog \triangleq Oplog = [s \in Server \mapsto \langle\rangle]$
$InitStore \triangleq Store = [n \in Server \cup Client \mapsto [k \in Key \mapsto Nil]]$
$InitCt \triangleq Ct = [n \in Server \cup Client \mapsto [p \mapsto 0, l \mapsto 0]]$
$InitOt \triangleq Ot = [n \in Server \cup Client \mapsto [p \mapsto 0, l \mapsto 0]]$
$InitServerMsg \triangleq ServerMsg = [s \in Server \mapsto \langle\rangle]$
$InitPt \triangleq Pt = [s \in Server \mapsto 1]$
$InitCp \triangleq Cp = [n \in Server \cup Client \mapsto [p \mapsto 0, l \mapsto 0]]$
$InitState \triangleq State = [s \in Server \mapsto [s0 \in Server \mapsto [p \mapsto 0, l \mapsto 0, term \mapsto 0]]]$
$InitCurrentTerm \triangleq CurrentTerm = [s \in Server \mapsto 0]$
$InitReadyToServe \triangleq ReadyToServe = 0$
$InitSyncSource \triangleq SyncSource = [s \in Server \mapsto Nil]$

$Init \triangleq$
$\quad \land InitPrimary \land InitSecondary \land InitOplog \land InitStore \land InitCt$
$\quad \land InitOt \land InitPt \land InitCp$
$\quad \land InitServerMsg$
$\quad \land InitState \land InitCurrentTerm \land InitReadyToServe$
$\quad \land InitSyncSource$

Next *State* Actions
Replication Protocol: possible actions

$TurnOnReadyToServe \triangleq$
$\quad \land ReadyToServe = 0$
$\quad \land \exists\, s \in Primary :$
$\quad\quad \land CurrentTerm' = [CurrentTerm \text{ EXCEPT } ![s] = CurrentTerm[s] + 1]$
$\quad\quad \land CurrentTerm' = [s \in Server \mapsto 1]$
$\quad\quad \land ReadyToServe' = ReadyToServe + 1$
$\quad \land \text{UNCHANGED } \langle electionVars, storageVars, serverVars, Ct, messageVar, timeVar, Cp, State\rangle$

$Stepdown \triangleq$
$\quad \land ReadyToServe > 0$
$\quad \land \exists\, s \in Primary :$
$\quad\quad \land Primary' = Primary \setminus \{s\}$
$\quad\quad \land Secondary' = Secondary \cup \{s\}$
$\quad \land \text{UNCHANGED } \langle storageVars, serverVars, Ct, messageVar, timeVar, Cp, State, CurrentTerm, functionalV$

Todo: *Stepdown* when receiving a higher term heartbeat

There are majority nodes agree to elect node i to become primary
$ElectPrimary \triangleq$

```
    ∧ ReadyToServe > 0
    ∧ ∃ i ∈ Server : ∃ majorNodes ∈ SUBSET (Server) :
        ∧ ∀ j ∈ majorNodes : ∧ NotBehind(i, j)
                             ∧ CurrentTerm[i] ≥ CurrentTerm[j]
        ∧ IsMajority(majorNodes)
        voted nodes for i cannot be primary anymore
        ∧ Primary' = LET possiblePrimary ≜ Primary \ majorNodes
                     IN  possiblePrimary ∪ {i}
        add voted nodes into secondaries
        ∧ Secondary' = LET possibleSecondary ≜ Secondary ∪ majorNodes
                       IN  possibleSecondary \ {i}
        ∧ CurrentTerm' = [index ∈ Server ↦ IF index ∈ (majorNodes ∪ {i})
                                           THEN CurrentTerm[i] + 1
                                           ELSE CurrentTerm[index]]
         A primary node do not have any sync source
        ∧ SyncSource' = [SyncSource EXCEPT ![i] = Nil]
    ∧ UNCHANGED ⟨storageVars, Ct, Ot, messageVar, timeVar, Cp, State, functionalVar⟩

AdvanceCp ≜
    ∧ ReadyToServe > 0
    ∧ ∃ s ∈ Primary :
      LET newCp ≜
          LET quorumAgree ≜ QuorumAgreeInSameTerm(State[s])
          IN  IF Cardinality(quorumAgree) > 0
              THEN LET QuorumSet ≜ CHOOSE i ∈ quorumAgree : TRUE
                       serverInQuorum ≜ CHOOSE j ∈ QuorumSet : TRUE
                       termOfQuorum ≜ State[s][serverInQuorum].term
                       StateSet ≜ {[p ↦ State[s][j].p, l ↦ State[s][j].l] : j ∈ QuorumSet}
                       newCommitPoint ≜ HLCMinSet(StateSet)
                       oldCommitPoint ≜ [p ↦ Cp[s].p, l ↦ Cp[s].l]
                    newCp must be greater than current Cp for primary to advance it
                   IN  IF termOfQuorum = CurrentTerm[s] ∧ HLCLt(oldCommitPoint, newCommit
                       THEN [p ↦ newCommitPoint.p, l ↦ newCommitPoint.l, term ↦ termOf
                       ELSE Cp[s]
              ELSE Cp[s]
      IN  Cp' = [Cp EXCEPT ![s] = newCp]
    ∧ UNCHANGED ⟨electionVars, storageVars, serverVars, Ct, messageVar, timeVar, State, CurrentTerm, fu

 heartbeatoplogOtstore
ServerTakeHeartbeat ≜
    ∧ ReadyToServe > 0
    ∧ ∃ s ∈ Server :
        ∧ Len(ServerMsg[s]) ≠ 0   message channel is not empty
        ∧ ServerMsg[s][1].type = "heartbeat"
        ∧ CurrentTerm[s] = ServerMsg[s][1].term
```

```
        ∧ Ct' = [Ct EXCEPT ![s] = HLCMax(Ct[s], ServerMsg[s][1].ct)]
        ∧ State' =
            LET newState ≜ [
                    p ↦ ServerMsg[s][1].aot.p,
                    l ↦ ServerMsg[s][1].aot.l,
                    term ↦ ServerMsg[s][1].term
                ]
            IN  LET SubHbState ≜ State[s]
                    hb ≜ [SubHbState EXCEPT ![ServerMsg[s][1].s] = newState]
                IN  [State EXCEPT ![s] = hb]
        ∧ Cp' = LET newcp ≜
                  primary node: compute new mcp
                    IF s ∈ Primary THEN
                        LET quorumAgree ≜ QuorumAgreeInSameTerm(State[s])IN
                            IF Cardinality(quorumAgree) > 0
                                THEN LET QuorumSet ≜ CHOOSE i ∈ quorumAgree : TRUE
                                         serverInQuorum ≜ CHOOSE j ∈ QuorumSet : TRUE
                                         termOfQuorum ≜ State[s][serverInQuorum].term
                                         StateSet ≜ {[p ↦ State[s][j].p, l ↦ State[s][j].l] : j ∈ QuorumSet
                                         newCommitPoint ≜ HLCMinSet(StateSet)
                                     IN  IF termOfQuorum = CurrentTerm[s]
                                             THEN
                                                 [p ↦ newCommitPoint.p, l ↦ newCommitPoint.l, term ↦
                                             ELSE  Cp[s]
                                ELSE  Cp[s]
                  secondary node: update mcp
                    ELSE  IF LET msgCP ≜ [p ↦ ServerMsg[s][1].cp.p, l ↦ ServerMsg[s][1].cp.l]IN
                              ∧ ¬HLCLt(msgCP, Cp[s])
                              ∧ ¬HLCLt(Ot[s], msgCP)
                              The term of cp must equal to the CurrentTerm of that node to advance it
                              ∧ ServerMsg[s][1].term = CurrentTerm[s]
                          THEN ServerMsg[s][1].cp
                          ELSE  Cp[s]
                IN  [Cp EXCEPT ![s] = newcp]
        ∧ ServerMsg' = [ServerMsg EXCEPT ![s] = Tail(@)]
        ∧ CurrentTerm' = [CurrentTerm EXCEPT ![s] = Max(CurrentTerm[s], ServerMsg[s][1].term)]
    ∧ UNCHANGED ⟨electionVars, storageVars, serverVars, timeVar, functionalVar⟩

ServerTakeUpdatePosition ≜
    ∧ ReadyToServe > 0
    ∧ ∃ s ∈ Server :
        ∧ Len(ServerMsg[s]) ≠ 0   message channel is not empty
        ∧ ServerMsg[s][1].type = "update_position"
        ∧ Ct' = [Ct EXCEPT ![s] = HLCMax(Ct[s], ServerMsg[s][1].ct)]  update ct accordingly
        ∧ State' =
```

```
                LET newState ≜ [
                        p ↦ ServerMsg[s][1].aot.p,
                        l ↦ ServerMsg[s][1].aot.l,
                        term ↦ ServerMsg[s][1].term
                    ]
                IN  LET SubHbState ≜ State[s]
                        hb ≜ [SubHbState EXCEPT ![ServerMsg[s][1].s] = newState]
                    IN  [State EXCEPT ![s] = hb]
            ∧ Cp' = LET newcp ≜
                      primary node: compute new mcp
                      IF s ∈ Primary THEN
                          LET quorumAgree ≜ QuorumAgreeInSameTerm(State[s])IN
                              IF Cardinality(quorumAgree) > 0
                                  THEN LET QuorumSet ≜ CHOOSE i ∈ quorumAgree : TRUE
                                           serverInQuorum ≜ CHOOSE j ∈ QuorumSet : TRUE
                                           termOfQuorum ≜ State[s][serverInQuorum].term
                                           StateSet ≜ {[p ↦ State[s][j].p, l ↦ State[s][j].l] : j ∈ QuorumSet
                                           newCommitPoint ≜ HLCMinSet(StateSet)
                                       IN  IF termOfQuorum = CurrentTerm[s]
                                               THEN
                                                   [p ↦ newCommitPoint.p, l ↦ newCommitPoint.l, term ↦
                                             ELSE  Cp[s]
                                  ELSE  Cp[s]
                      secondary node: update mcp
                          ELSE  IF LET msgCP ≜ [p ↦ ServerMsg[s][1].cp.p, l ↦ ServerMsg[s][1].cp.l]IN
                                    ∧ ¬HLCLt(msgCP, Cp[s])
                                    ∧ ¬HLCLt(Ot[s], msgCP)
                                THEN ServerMsg[s][1].cp
                                ELSE Cp[s]
                    IN  [Cp EXCEPT ![s] = newcp]
            ∧ CurrentTerm' = [CurrentTerm EXCEPT ![s] = Max(CurrentTerm[s], ServerMsg[s][1].term)]
            ∧ ServerMsg' = LET newServerMsg ≜ [ServerMsg EXCEPT ![s] = Tail(@)]
                           IN  (LET appendMsg ≜ [type ↦ "update_position", s ↦ ServerMsg[s][1].s, aot ↦ Se
                                                ct ↦ ServerMsg[s][1].ct, cp ↦ ServerMsg[s][1].cp, term ↦ ServerM
                                IN  (LET newMsg ≜ IF s ∈ Primary ∨ SyncSource[s] = Nil
                                                  THEN newServerMsg If s is primary, accept the msg, else f
                                                  ELSE [newServerMsg EXCEPT ![SyncSource[s]] = Append
                                     IN  newMsg))
            ∧ UNCHANGED ⟨electionVars, storageVars, serverVars, timeVar, functionalVar⟩

NTPSync ≜  simplify NTP protocal
    ∧ ReadyToServe > 0
    ∧ Pt' = [s ∈ Server ↦ MaxPt]
    ∧ UNCHANGED ⟨electionVars, storageVars, serverVars, learnableVars, messageVar, functionalVar⟩
```

$AdvancePt \triangleq$
$\land\ ReadyToServe > 0$
$\land\ \exists\, s \in Server :$
$\land\ s \in Primary$ for simplicity
$\land\ Pt[s] \leq PtStop$
$\land\ Pt' = [Pt \text{ EXCEPT } ![s] = @ + 1]$ advance physical time
$\land\ BroadcastHeartbeat(s)$ broadcast heartbeat periodly
$\land$ UNCHANGED $\langle electionVars,\ storageVars,\ serverVars,\ learnableVars,\ functionalVar \rangle$

Replicate oplog from node j to node i, and update related structures accordingly
$Replicate \triangleq$
$\land\ ReadyToServe > 0$
$\land\ \exists\, i, j \in Server :$
$\land\ CanSyncFrom(i, j)$ i can sync from j only need not to rollback
$\land\ i \in Secondary$
$\land\ ReplicateOplog(i, j) \neq \langle\rangle$
$\land\ Oplog' = [Oplog \text{ EXCEPT } ![i] = @ \circ ReplicateOplog(i, j)]$
$\land\ Store' = [Store \text{ EXCEPT } ![i] = Store[j]]$
$\land\ Ct' = [Ct \text{ EXCEPT } ![i] = HLCMax(Ct[i],\ Ct[j])]$ update $Ct[i]$
$\land\ Ot' = [Ot \text{ EXCEPT } ![i] = HLCMax(Ot[i],\ Ot[j])]$ update $Ot[i]$
$\land\ Cp' = [Cp \text{ EXCEPT } ![i] = HLCMax(Cp[i],\ Cp[j])]$ update $Cp[i]$
$\land\ CurrentTerm' = [CurrentTerm \text{ EXCEPT } ![i] = Max(CurrentTerm[i],\ CurrentTerm[j])]$ update $Curren$
$\land\ State' =$
LET $newState \triangleq$ [
$p \mapsto Ot[j].p,$
$l \mapsto Ot[j].l,$
$term \mapsto CurrentTerm[j]$
]
IN LET $SubHbState \triangleq State[i]$
$hb \triangleq [SubHbState \text{ EXCEPT } ![j] = newState]$
IN $[State \text{ EXCEPT } ![i] = hb]$ update j's state i knows
$\land$ LET $msg \triangleq [type \mapsto$ "update_position", $s \mapsto i,\ aot \mapsto Ot'[i],\ ct \mapsto Ct'[i],\ cp \mapsto Cp'[i],\ term \mapsto Curr$
IN $ServerMsg' = [ServerMsg \text{ EXCEPT } ![j] = Append(ServerMsg[j],\ msg)]$
$\land\ SyncSource' = [SyncSource \text{ EXCEPT } ![i] = j]$
$\land$ UNCHANGED $\langle electionVars,\ timeVar,\ functionalVar \rangle$

Rollback i's oplog and recover it to j's state
Recover to j's state immediately to prevent internal client request
$RollbackAndRecover \triangleq$
$\land\ ReadyToServe > 0$
$\land\ \exists\, i, j \in Server :$
$\land\ i \in Secondary$
$\land\ CanRollback(i, j)$
$\land$ LET $cmp \triangleq RollbackCommonPoint(i, j)$ IN
LET $commonLog \triangleq SubSeq(Oplog[i],\ 1,\ cmp)$

```
                appendLog  ≜  SubSeq(Oplog[j], cmp + 1, Len(Oplog[j]))
            IN  Oplog' = [Oplog EXCEPT ![i] = commonLog ∘ appendLog]
        ∧ CurrentTerm' = [CurrentTerm EXCEPT ![i] = Max(CurrentTerm[i], CurrentTerm[j])]  update Curren
        ∧ Store' = [Store EXCEPT ![i] = Store[j]]
        ∧ Ct' = [Ct EXCEPT ![i] = HLCMax(Ct[i], Ct[j])]  update Ct[i]
        ∧ Ot' = [Ot EXCEPT ![i] = HLCMax(Ot[i], Ot[j])]  update Ot[i]
        ∧ Cp' = [Cp EXCEPT ![i] = HLCMax(Cp[i], Cp[j])]  update Cp[i]
        ∧ State' =
            LET newStatei ≜ [
                    p ↦ Ot'[i].p,
                    l ↦ Ot'[j].l,
                    term ↦ CurrentTerm'[i]
                ]
                newStatej ≜ [
                    p ↦ Ot[j].p,
                    l ↦ Ot[j].l,
                    term ↦ CurrentTerm[j]
                ]
            IN  LET SubHbState ≜ State[i]
                    hb ≜ [SubHbState EXCEPT ![i] = newStatei]  update i's self state (used in mcp computation
                    hb1 ≜ [hb EXCEPT ![j] = newStatej]  update j's state i knows
                IN  [State EXCEPT ![i] = hb1]
        ∧ LET msg ≜ [type ↦ "update_position", s ↦ i, aot ↦ Ot'[i], ct ↦ Ct'[i], cp ↦ Cp'[i], term ↦ Curr
            IN  ServerMsg' = [ServerMsg EXCEPT ![j] = Append(ServerMsg[j], msg)]
        ∧ SyncSource' = [SyncSource EXCEPT ![i] = j]
    ∧ UNCHANGED ⟨electionVars, timeVar, functionalVar⟩

ClientRequest ≜
    ∧ ReadyToServe > 0
    ∧ ∃ s ∈ Server, k ∈ Key, v ∈ Value :
        ∧ s ∈ Primary
        ∧ Tick(s)
        ∧ Ot' = [Ot EXCEPT ![s] = Ct'[s]]
        ∧ Store' = [Store EXCEPT ![s][k] = v]
        ∧ Oplog' = LET entry ≜ [k ↦ k, v ↦ v, ot ↦ Ot'[s], term ↦ CurrentTerm[s]]
                       newLog ≜ Append(Oplog[s], entry)
                   IN  [Oplog EXCEPT ![s] = newLog]
        ∧ State' =
            LET newState ≜ [
                    p ↦ Ot'[s].p,
                    l ↦ Ot'[s].l,
                    term ↦ CurrentTerm[s]
                ]
            IN  LET SubHbState ≜ State[s]
                    hb ≜ [SubHbState EXCEPT ![s] = newState]
```

$\quad$ IN $\quad [State$ EXCEPT $![s] = hb]$ update i's state
$\wedge$ UNCHANGED $\langle electionVars,\ messageVar,\ timeVar,\ Cp,\ CurrentTerm,\ functionalVar,\ SyncSource\rangle$

Next state for all configurations

$$
\begin{aligned}
Next \triangleq\ & \vee Replicate \\
& \vee AdvancePt \\
& \vee AdvanceCp \\
& \vee ServerTakeHeartbeat \\
& \vee ServerTakeUpdatePosition \\
& \vee Stepdown \\
& \vee RollbackAndRecover \\
& \vee TurnOnReadyToServe \\
& \vee ElectPrimary \\
& \vee ClientRequest \\
& \vee NTPSync
\end{aligned}
$$

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

Properties to check?

$$
\begin{aligned}
& IsLogPrefix(i,\ j) \triangleq \\
& \quad \wedge Len(Oplog[i]) \leq Len(Oplog[j]) \\
& \quad \wedge Oplog[i] = SubSeq(Oplog[j],\ 1,\ Len(Oplog[i]))
\end{aligned}
$$

If two logs have the same last log entry term, then one is a prefix of the other (from Will

$$
\begin{aligned}
& LastTermsEquivalentImplyPrefixes \triangleq \\
& \quad \forall\, i,\ j \in Server : \\
& \qquad LogTerm(i,\ Len(Oplog[i])) = LogTerm(j,\ Len(Oplog[j])) \Rightarrow \\
& \qquad IsLogPrefix(i,\ j) \vee IsLogPrefix(j,\ i)
\end{aligned}
$$

Check whether terms are incremented monotoniclly (from Will

$$
\begin{aligned}
& TermsMonotonic \triangleq \\
& \quad \Box[\forall\, s \in Server : CurrentTerm'[s] \geq CurrentTerm[s]]_{vars}
\end{aligned}
$$

Check the log in *Primary* node is append only (from Will

$$
\begin{aligned}
& PrimaryAppendOnly \triangleq \\
& \quad \Box[\forall\, s \in Server : s \in Primary \Rightarrow Len(Oplog'[s]) \geq Len(Oplog[s])]_{vars}
\end{aligned}
$$

Never rollback oplog before common point (from Will & Raft *Mongo*

$$
\begin{aligned}
& NeverRollbackCommonPoint \triangleq \\
& \quad \exists\, i,\ j \in Server : CanRollback(i,\ j) \Rightarrow \\
& \qquad \text{LET } commonPoint \triangleq RollbackCommonPoint(i,\ j) \\
& \qquad\quad lastOplog \triangleq Oplog[i][commonPoint] \\
& \qquad \text{IN } \quad HLCLt(Cp[i],\ lastOplog.ot)
\end{aligned}
$$

Eventually log correctness (from Will

$$
\begin{aligned}
& EventuallyLogsConverge \triangleq \Diamond\Box[\forall\, s,\ t \in Server : s \neq t \Rightarrow Oplog[s] = Oplog[t]]_{vars} \\
& EventuallyLogsNonEmpty \triangleq \Diamond(\exists\, s \in Server : Len(Oplog[s]) > 0)
\end{aligned}
$$

(from *RaftMongo*

$$TwoPrimariesInSameTerm \triangleq$$
$$\exists\, i,\, j \in Server :$$
$$\wedge\ i \neq j$$
$$\wedge\ CurrentTerm[i] = CurrentTerm[j]$$
$$\wedge\ i \in Primary$$
$$\wedge\ j \in Primary$$

$$NoTwoPrimariesInSameTerm \triangleq \neg TwoPrimariesInSameTerm$$

Check if there is any cycle of sync source path (from *RaftMongo Sync*

$$SyncSourceCycleTwoNode \triangleq$$
$$\exists\, s,\, t \in Server :$$
$$\wedge\ s \neq t$$
$$\wedge\ SyncSource[s] = t$$
$$\wedge\ SyncSource[t] = s$$

$$BoundedSeq(s,\, n) \triangleq [1 \,..\, n \to s]$$

$$SyncSourcePaths \triangleq$$
$$\{p \in BoundedSeq(Server,\, Cardinality(Server)) :$$
$$\forall\, i \in 1 \,..\, (Len(p) - 1) : SyncSource[p[i]] = p[i+1]\}$$

$$SyncSourcePath(i,\, j) \triangleq$$
$$\exists\, p \in SyncSourcePaths :$$
$$\wedge\ Len(p) > 1$$
$$\wedge\ p[1] = i$$
$$\wedge\ p[Len(p)] = j$$

$$SyncSourceCycle \triangleq$$
$$\exists\, s \in Server : SyncSourcePath(s,\, s)$$

$$NonTrivialSyncCycle \triangleq SyncSourceCycle \wedge \neg SyncSourceCycleTwoNode$$
$$NoNonTrivialSyncCycle \triangleq \neg NonTrivialSyncCycle$$

* Modification History
* Last modified *Wed* May 11 22:55:31 *CST* 2022 by *dh*
* Created *Mon Apr* 18 11:38:53 *CST* 2022 by *dh*