─────────────────── MODULE $ConsensusUpdate$ ───────────────────

EXTENDS $Integers$, $Sequences$, TLC

CONSTANTS

    $Names$,   a set

    $Participants$,   an array of participants, in their order in the state channel

    $NULL$

ASSUME

  $\wedge\ Len(Participants) > 1$

$NumParticipants \triangleq Len(Participants)$

$Types \triangleq [$

    $WAITING \mapsto \text{"WAITING"},$

    $SENT \quad\ \mapsto \text{"SENT"},$

    $SUCCESS \mapsto \text{"SUCCESS"},$

    $FAILURE \mapsto \text{"FAILURE"}$

$]$

$Status \triangleq [$

  $OK \qquad\ \mapsto \text{"OK"},$

  $ABORT \quad\ \mapsto \text{"ABORT"},$

  $SUCCESS \quad \mapsto \text{"SUCCESS"}$

$]$

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$Running(state) \triangleq state.type \in \{Types.WAITING,\ Types.SENT\}$

$Terminated(state) \triangleq \neg Running(state)$

--algorithm $consensus\_update$

For the moment, we assume that participants only send commitments forward.

Since a read message is then discarded, it's enough to just store one.

variables $msg = NULL$

define

 Arrays are 1-indexed, while the % operator returns a number between 0 and $NumParticipants$.

 This explains the following slightly complicated expression

$mover(turnNumber) \triangleq 1 + ((turnNumber - 1)\%NumParticipants)$

$safeToSend(state) \quad \triangleq$

    $\wedge\ state.type = Types.WAITING$

    $\wedge\ \vee\ state.ourIndex = state.turnNumber\%NumParticipants$

       $\vee\ \wedge\ msg \neq NULL$

          $\wedge\ msg.status = Status.OK$

          $\wedge\ state.ourIndex = mover(msg.turnNumber)$

$target(turnNumber) \triangleq Participants[mover(turnNumber)]$

end define ;

macro $sendVote(turnNumber, votesRequired)$

begin

assert $votesRequired > 0$;

1

```
state := [
    type ↦ Types.SENT,
    turnNumber ↦ turnNumber,
    ourIndex     ↦ state.ourIndex
] ;
msg := [
    to ↦ target(state.turnNumber),
    turnNumber        ↦ state.turnNumber,
    votesRequired     ↦ votesRequired,
    status            ↦ Status.OK
]
end macro ;


macro returnSuccess()
begin
state := [type ↦ Types.SUCCESS] @@ state ;
msg := [
    to      ↦ target(state.turnNumber),
    status ↦ Status.SUCCESS
]
end macro ;

macro returnFailure(turnNumber)
begin
state := [
    type ↦ Types.FAILURE,
    turnNumber ↦ turnNumber
] @@ state ;
msg := [
    to ↦ target(state.ourIndex + 1),
    status ↦ Status.ABORT
] ;
end macro ;

macro vote(turnNumber, votesRequired)
begin
if votesRequired = 0 then returnSuccess()
 else sendVote(turnNumber, votesRequired)
end if ; end macro ;

macro waitForUpdate(turnNumber)
begin
state := [
    turnNumber ↦ turnNumber,
    type          ↦ Types.WAITING,
    ourIndex     ↦ state.ourIndex
```

```
] ;
msg := NULL ;
end macro ;

macro voteOrreturnFailure(turnNumber, votesRequired)
begin
  If the participant agrees with the allocation, they vote
either vote(turnNumber, votesRequired)
  Otherwise, they return FAILURE
or returnFailure(turnNumber)
end either ; end macro ;
```

Calling this a fair process prevents the process from stuttering forever. It's always considered to be valid to take a step where your state variables don't change, which could be the case if some unrelated protocols end up in an infinite loop, for instance. However, we want to check that IF A: wallets always eventually take some valid action THEN B: wallets always eventually terminate the consensus-update protocol Calling the process fair ensures that A is true, and therefore the model checks that under the assumption A, B is also true.

```
fair process consensusUpdate ∈ DOMAIN Participants
variables
  state = [
    turnNumber ↦ 1,
    ourIndex ↦ self,
    type ↦ Types.WAITING
  ],
  me = Participants[self]

begin
```
  Each participant either sends a message if it's currently safe to do so, or else it reads a message for the participant, updates their state accordingly, and sends a message if it's then safe. These actions are currently assumed to be atomic, and are therefore assigned to a single label, *ReachConsensus*

```
  ReachConsensus:
    while Running(state) do
      if safeToSend(state) ∧ msg = NULL then
        either returnFailure(state.turnNumber)  If the commitment is not valid
        or
            if    state.type = Types.WAITING then vote(state.turnNumber + 1, NumParticipants − 1) ;
             elsif state.type = Types.SENT      then returnFailure(state.turnNumber) ;
             else assert FALSE
            end if   ;
        end either ;
      else
          await msg ≠ NULL ∧ msg.to = me ;
          if msg.status = Status.OK then
              If the commitment received is not valid, return FAILURE
```

either $returnFailure(state.turnNumber)$
or if $msg.turnNumber > state.turnNumber$ **then**

First, update our state based on the incoming message

if $msg.votesRequired = 0$ **then** $returnSuccess()$
elsif $safeToSend(state)$ **then**
if $state.type = Types.SENT$ **then** $returnFailure(msg.turnNumber)$
elsif $state.type = Types.WAITING$ **then** $voteOrreturnFailure(msg.turnNumber + 1, n$
else assert FALSE ;
end if ;
else $waitForUpdate(msg.turnNumber)$
end if ;
end if ; **end either** ;
elsif $msg.status = Status.ABORT$ **then** $returnFailure(state.turnNumber)$
elsif $msg.status = Status.SUCCESS$ **then** $returnSuccess()$
else assert FALSE
end if ;
end if ;
end while ;
end process ;

end algorithm ;

VARIABLES msg, pc

define statement
$mover(turnNumber) \triangleq 1 + ((turnNumber - 1)\%NumParticipants)$
$safeToSend(state) \triangleq$
$\quad \wedge state.type = Types.WAITING$
$\quad \wedge \vee state.ourIndex = state.turnNumber\%NumParticipants$
$\qquad \vee \wedge msg \neq NULL$
$\qquad\quad \wedge msg.status = Status.OK$
$\qquad\quad \wedge state.ourIndex = mover(msg.turnNumber)$
$target(turnNumber) \triangleq Participants[mover(turnNumber)]$

VARIABLES $state$, me

$vars \triangleq \langle msg, pc, state, me \rangle$

$ProcSet \triangleq (\text{DOMAIN } Participants)$

$Init \triangleq$ *Global variables*
$\quad \wedge msg = NULL$
Process consensusUpdate
$\quad \wedge state = [self \in \text{DOMAIN } Participants \mapsto \quad [$

4

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad turnNumber \mapsto 1,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ourIndex \mapsto self,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad type \mapsto Types.WAITING\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad]]\\
\qquad\qquad \wedge\, me = [self \in \text{DOMAIN } Participants \mapsto Participants[self]]\\
\qquad\qquad \wedge\, pc\ = [self \in ProcSet \mapsto \text{"ReachConsensus"}]
\end{array}
$$

$$
\begin{array}{ll}
ReachConsensus(self) \;\triangleq\;& \wedge\, pc[self] = \text{"ReachConsensus"}\\
& \wedge\, \text{IF } Running(state[self])\\
&\qquad \text{THEN }\ \wedge \text{ IF } safeToSend(state[self]) \wedge msg = NULL\\
&\qquad\qquad\qquad \text{THEN }\ \wedge \vee \wedge state' = [state \text{ EXCEPT } ![self] = \qquad\qquad [\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad type \mapsto Types.F\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad turnNumber \mapsto\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad]\,@@\, state[self]]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, msg' = \qquad\quad [\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad to \mapsto target(state'[self].ourIndex + 1),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad status \mapsto Status.ABORT\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad]\\
&\qquad\qquad\qquad\qquad\quad \vee\ \wedge \text{ IF } state[self].type = Types.WAITING\\
&\qquad\qquad\qquad\qquad\qquad\quad \text{THEN }\ \wedge \text{ IF } (NumParticipants - 1) = 0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN }\ \wedge state' = [state \text{ EXCEPT } ![s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, msg' = \qquad\quad [\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad to \qquad \mapsto target\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad status \mapsto Status\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE }\ \wedge Assert((NumParticipants\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{"Failure of assertio}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, state' = [state \text{ EXCEPT } ![s\\
\\
\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, msg' = \qquad\qquad [\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad to \mapsto target(sta\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad turnNumber\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad votesRequired\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad status\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad]\\
&\qquad\qquad\qquad\qquad\qquad \text{ELSE }\ \wedge \text{ IF } state[self].type = Types.SENT\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN }\ \wedge state' = [state \text{ EXCEPT } ![s\\
\\
\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, msg' = \qquad\qquad [\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad to \mapsto target(sta
\end{array}
$$

$$status \mapsto Status$$
$$]$$

ELSE $\quad \wedge Assert(\text{FALSE},$

$\quad\quad\quad\quad$ "Failure of assertio

$\quad\quad \wedge$ UNCHANGED $\langle msg,$

$\quad\quad\quad\quad\quad\quad\quad\quad state\rangle$

ELSE $\quad \wedge msg \neq NULL \wedge msg.to = me[self]$

$\quad\quad \wedge$ IF $msg.status = Status.OK$

$\quad\quad\quad$ THEN $\quad \wedge \vee \wedge state' = [state \text{ EXCEPT } ![self] =$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad typ$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad tur$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad] @@ st$

$\quad\quad\quad\quad\quad \wedge msg' = \quad\quad [$

$\quad\quad\quad\quad\quad\quad to \mapsto target(state'[self].ourI$

$\quad\quad\quad\quad\quad\quad status \mapsto Status.ABORT$

$\quad\quad\quad\quad\quad]$

$\quad\quad\quad\quad \vee \wedge$ IF $msg.turnNumber > state[self].turnl$

$\quad\quad\quad\quad\quad THEN \quad \wedge$ IF $msg.votesRequired = 0$

$\quad\quad\quad\quad\quad\quad\quad THEN \quad \wedge state' = [state$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge msg' =$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad to$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad sta$

$\quad\quad\quad\quad\quad\quad\quad]$

$\quad\quad\quad\quad\quad\quad ELSE \quad \wedge$ IF $safeToSend$

$\quad\quad\quad\quad\quad\quad\quad THEN \quad \wedge I$

ELSE $\quad \wedge s$

$\wedge r$

ELSE $\quad \wedge$ TRUE
$\wedge$ UNCHANGED $\langle msg,$
$state \rangle$

ELSE $\quad \wedge$ IF $msg.status = Status.ABORT$
THEN $\wedge state' = [state$ EXCEPT $![self] =$

$\wedge msg' = \qquad [$
$to \mapsto target(state'[se$
$status \mapsto Status.ABC$
$]$

ELSE $\quad \wedge$ IF $msg.status = Status.SUCCE$
THEN $\wedge state' = [state$ EXC
$\wedge msg' = \qquad [$
$to \qquad \mapsto$
$status \mapsto$
$]$

ELSE $\quad \wedge Assert(\text{FALSE},$

7

$$\qquad\qquad \wedge\, pc' = [pc \text{ EXCEPT } ![self] = \text{"ReachConsensus"}]$$
$$\text{ELSE}\quad \wedge\, pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}]$$
$$\wedge \text{ UNCHANGED } \langle msg,\, state \rangle$$
$$\wedge\, me' = me$$

$consensusUpdate(self) \;\triangleq\; ReachConsensus(self)$

Allow infinite stuttering to prevent deadlock on termination.
$Terminating \;\triangleq\; \wedge\, \forall\, self \in ProcSet : pc[self] = \text{"Done"}$
$\qquad\qquad\quad\; \wedge \text{ UNCHANGED } vars$

$Next \;\triangleq\; (\exists\, self \in \text{DOMAIN } Participants : consensusUpdate(self))$
$\qquad\quad\; \vee\; Terminating$

$Spec \;\triangleq\; \wedge\, Init \wedge \square[Next]_{vars}$
$\qquad\quad\; \wedge\, \forall\, self \in \text{DOMAIN } Participants : \text{WF}_{vars}(consensusUpdate(self))$

$Termination \;\triangleq\; \Diamond(\forall\, self \in ProcSet : pc[self] = \text{"Done"})$

END TRANSLATION

$AllowedMessages \;\triangleq\;$
$\quad [$
$\qquad turnNumber : Nat,$
$\qquad votesRequired : 0\,..\,(NumParticipants - 1),$
$\qquad to : Names,$
$\qquad status : \{Status.OK\}$
$\quad]$
$\quad \cup \{NULL\}$
$\quad \cup [$
$\qquad to : Names,$
$\qquad status : \{Status.ABORT,\, Status.SUCCESS\}$
$\quad]$

$States \;\triangleq\; \{\}$
$\quad \cup\; [turnNumber : Nat,\, ourIndex : \text{DOMAIN } Participants,\, type : Range(Types)]$

Safety properties

$TypeOK \;\triangleq\;$
The following two conditions specify the format of each message and participant state.
$\quad \wedge\, state \in [\text{DOMAIN } Participants \to States]$
$\quad \wedge\, msg \in AllowedMessages$

TODO : *Get TurnNumberDoesNotDecrease and StaysTerminated*
For some reason, $state[p].turnNumber$ is not valid
$TurnNumberDoesNotDecrease \triangleq$
   $\wedge \forall\, p \in \text{DOMAIN } Participants : state[p].turnNumber' \geq state[p].turnNumber$

Once a process has terminated, its state does not change.
$StaysTerminated \triangleq \forall\, p \in \text{DOMAIN } Participants : (Terminated(state[p]) \Rightarrow (state'[p] = state[p]))$

Liveness properties

The protocol always terminates consistently across all processes.
TODO: Is this actually feasible, or actually what we want?
For example, perhaps the last person to vote agrees, and sends a message reaching consensus.
Their process terminates in the $SUCCESS$ state, but for whatever reason their
commitment was invalid, and the other processes therefore terminate in $FAILURE$.
$ProtocolTerminates \triangleq$
    $\vee \; \wedge (\forall\, p \in \text{DOMAIN } Participants : \Diamond\Box(state[p].type = Types.SUCCESS))$
       $\wedge \text{ TRUE}$ *TODO*: In this case, should we specify that they reach the same turn number?
    $\vee (\forall\, p \in \text{DOMAIN } Participants : \Diamond\Box(state[p].type = Types.FAILURE))$

The value of msg should eventually always be $NULL$
$MessagesAreRead \triangleq \Diamond\Box(msg = NULL)$

* Modification History
* Last modified *Mon Aug* 12 14:42:54 *MDT* 2019 by *andrewstewart*
* Created *Tue Aug* 06 14:38:11 *MDT* 2019 by *andrewstewart*