MODULE *VDFConsensus*

EXTENDS *FiniteSets*, *Integers*, *Utils*, *TLC*

CONSTANTS
```
    P     the set of processes
,   B     the set of malicious processes
,   tAdv  the time it takes for a malicious process to produce a message
,   tWB   the time it takes for a well-behaved process to produce a message
```

ASSUME $B \subseteq P$ malicious processes are a subset of all processes
$W \triangleq P \setminus B$ the set of well-behaved processes

$Tick \triangleq Nat$ a tick is a real-time clock tick
$Round \triangleq Nat$ a round is just a tag on a message

Processes build a *DAG* of messages. The message-production rate of well-behaved processes is of 1 message per *tWB* ticks, and that of malicious processes is of 1 message per *tAdv* ticks. We require that, collectively, well-behaved processes produce messages at a rate strictly higher than that of malicious processes.

ASSUME $Cardinality(W) * tAdv > Cardinality(B) * tWB$

A message consists of a unique *ID*, a round number, and a coffer containing the *IDs* of a set of predecessor messages:
$MessageID \triangleq P \times Nat$
$Message \triangleq [id : MessageID, round : Round, coffer : \text{SUBSET } MessageID]$
$sender(m) \triangleq m.id[1]$

A strongly consistent chain is a subset of the messages in the *DAG* that potentially has some dangling pointers (*i.e.* messages that have predecessors not in the chain) and that satisfies the following recursive predicate:

* Any set of messages which all have a round of 0 is a strongly consistent chain.

* A set of messages C with some non-zero rounds and maximal round r is a strongly consistent chain when, with *Tip* being the set of messages in the chain that have round r and *Pred* being the set of messages in the chain with round $r-1$, *Pred* is a strict majority of the set of predecessors of each message in *Tip* and $C \setminus Tip$ is a consistent chain.

The max round of a set of messages is the maximal round of its messages:
$MaxRound(M) \triangleq MaxInteger(\{m.round : m \in M\})$

```
StrongglyConsistentChain(M) ≜
```

```
$StronglyConsistentChain(M) \triangleq$
    $\land M \neq \{\}$
    $\land \lor MaxRound(M) = 0$
       $\lor \forall r \in 1 \,..\, MaxRound(M) :$
          LET $Tip \triangleq \{m \in M : m.round = r\}$
              $Pred \triangleq \{m \in M : m.round = r - 1\}$
          IN  $\land Tip \neq \{\}$
              $\land \forall m \in Tip :$
                  $\land \{m2.id : m2 \in Pred\} \subseteq m.coffer$
```

```
            ∧ $2 * Cardinality(Pred) > Cardinality(m.coffer)$
```

A weaker version of the above:

```
$ConsistentChain(M) \triangleq$
    ∧ $M \neq \{\}$
    ∧ ∨ $MaxRound(M) = 0$
      ∨ $\forall r \in 1 \,..\, MaxRound(M) :$
          LET $Tip \triangleq \{m \in M : m.round = r\}$
              $Pred \triangleq \{m \in M : m.round = r - 1\}$
          IN  ∧ $Tip \neq \{\}$
              ∧ $\forall m \in Tip : \exists Maj \in$ SUBSET $Pred :$
                  ∧ $\{m2.id : m2 \in Maj\} \subseteq m.coffer$
                  ∧ $2 * Cardinality(Maj) > Cardinality(m.coffer)$
```

```
$Chains(M, r) \triangleq \{C \in$ SUBSET $M :$
    ∧ $\forall m \in C : m.round \leq r$
    ∧ $\forall r2 \in 0 \,..\, r : \exists m \in C : m.round = r2\}$
```

The set of all consistent chains that can be found in M:

```
$ConsistentChains(M, r) \triangleq$
    $\{C \in Chains(M, r) : ConsistentChain(C)\}$
```

The set of all strongly consistent chains that can be found in M:

```
$StronglyConsistentChains(M, r) \triangleq$
    $\{C \in Chains(M, r) : StronglyConsistentChain(C)\}$
```

We can rank the chains by wheight, *i.e.* just their cardinality, or we can consider the maximal or minimal one in the subset order:

```
$HeaviestConsistentChains(M, r) \triangleq MaxCardinalitySets(ConsistentChains(M, r))$
$HeaviestStronglyConsistentChains(M, r) \triangleq MaxCardinalitySets(StronglyConsistentChains(M, r))$
$MinimalStronglyConsistentChains(M, r) \triangleq MinimalSets(StronglyConsistentChains(M, r))$
$MaximalStronglyConsistentChains(M, r) \triangleq MaximalSets(StronglyConsistentChains(M, r))$
```

Two chains are disjoint when there is a round smaller than their max round in which they share less than a strict majority of their messages.

```
$DisjointChains(C1, C2) \triangleq$
    LET $rmax \triangleq MaxRound(C1 \cup C2)$
    IN  $\exists r \in 0 \,..\, (rmax - 1) :$
          LET $I \triangleq \{m \in C1 : m.round = r\} \cap \{m \in C2 : m.round = r\}$
          IN  ∨ $2 * Cardinality(I) < Cardinality(C1)$
              ∨ $2 * Cardinality(I) < Cardinality(C2)$
```

Acceptance rule

```
$AcceptedMessages(M, r) \triangleq \{m \in M :$
    ∧ $m.round = r - 1$
    ∧ LET $CCs \triangleq MaximalStronglyConsistentChains(M, r - 1)$ IN        This looks promising!
    $\land$ $\exists\, C \in CCs : m \in C$
    $\land$ $\forall\, C1,\ C2 \in CCs :$
            $\land$ $m \in C1$
            $\land$ $m \notin C2$
            $\land$ $DisjointChains(C1,\ C2)$
            $\Rightarrow$ $Cardinality(C2) \leq Cardinality(C1)\}$
```

M does not have dangling edges:

$Closed(M) \triangleq \forall\, m \in M : \forall\, i \in m.coffer : \exists\, m2 \in M : m2.id = i$

Now we specify the algorithm

```
--algorithm $Algo${
  variables
      $messages = \{\}$ ;
      $tick = 0$ ;
      $phase =$ "start" ;   each tick has two phases: "start" and "end"
      $donePhase = [p \in P \mapsto$ "end"$]$ ;
      $pendingMessage = [p \in P \mapsto \langle\rangle]$ ;   message we're computing the $VDF$ on
      $messageCount = [p \in P \mapsto 0]$ ;   used to generate unique message $IDs$
  define {
      $currentRound \triangleq tick \div tWB$   round of well-behaved processes
      $wellBehavedMessages \triangleq \{m \in messages : sender(m) \in P \setminus B\}$
       possible sets of messages received by a well-behaved process:
      $receivedMsgsSets \triangleq$
           ignore messages from future rounds:
          LET $msgs \triangleq \{m \in messages : m.round < currentRound\}$IN
          $\{M \in$ SUBSET $msgs$ :
                 don't use a set of messages that has dangling edges (messages in coffers that are missing):
                $\land$ $Closed(M)$
                $\land$ $wellBehavedMessages \subseteq M\}$
   }
  macro $sendMessage(\ m\ )$ {
      $messages := messages \cup \{m\}$
   }
  process ( $clock \in \{$"clock"$\}$ ) {
tick:  while ( TRUE ) {
          await $\forall\, p \in P : donePhase[p] = phase$ ;
          if ( $phase =$ "start" )
              $phase :=$ "end"
          else {
              $phase :=$ "start" ;
              $tick := tick + 1$
           }
       }
```

```
    }
    process ( $proc \in P \setminus B$ )   a well-behaved process
    {
l1:    while ( TRUE ) {
          await $phase =$ "start" ;
          if ( $tick \% tWB = 0$ ) {
               Start the $VDF$ computation for the next message:
             with ( $msgs \in receivedMsgsSets$ )
             with ( $predMsgs = AcceptedMessages(msgs, currentRound)$ ) {
                $pendingMessage[self] := [$
                   $id \mapsto \langle self, messageCount[self] + 1\rangle,$
                   $round \mapsto currentRound,$
                   $coffer \mapsto \{m.id : m \in predMsgs\}]$ ;
                $messageCount[self] := messageCount[self] + 1$
             }
          } ;
          $donePhase[self] :=$ "start" ;
l2:       await $phase =$ "end" ;
          if ( $tick \% tWB = tWB - 1$ )
               it's the end of the $tWB$ period, the $VDF$ has been computed
             $sendMessage(pendingMessage[self])$ ;
          $donePhase[self] :=$ "end" ;
       }
    }
    process ( $byz \in B$ )   a malicious process
    {
lb1:   while ( TRUE ) {
          await $phase =$ "start" ;
           when $currentRound < 2$; \ * $TODO$ temporary hack
          if ( $tick \% tAdv = 0$ ) {
               Start the $VDF$ computation for the next message:
             with ( $maxRound = Max(\{m.round : m \in messages\} \cup \{0\}, \leq)$ )
             with ( $rnd \in \{maxRound, maxRound + 1\}$ )
             with ( $predMsgs \in$ SUBSET $\{m \in messages : m.round = rnd - 1\}$ ) {
                when $rnd > 0 \Rightarrow predMsgs \neq \{\}$ ;
                $pendingMessage[self] := [$
                   $id \mapsto \langle self, messageCount[self] + 1\rangle,$
                   $round \mapsto rnd,$
                   $coffer \mapsto \{m.id : m \in predMsgs\}]$ ;
                $messageCount[self] := messageCount[self] + 1$
             }
          } ;
          $donePhase[self] :=$ "start" ;
lb2:      await $phase =$ "end" ;
          if ( $tick \% tAdv = tAdv - 1$ )
```

```
            $sendMessage(pendingMessage[self])$ ;
        $donePhase[self] :=$ "end" ;
      } ;
    }
}
```

Invariant describing the type of the variables:

```
$TypeOK \triangleq$
    $\land\ messages \in \text{SUBSET}\ Message$
    $\land\ pendingMessage \in [P \rightarrow Message \cup \{\langle\rangle\}]$
    $\land\ tick \in Tick$
    $\land\ phase \in \{\text{"start"}, \text{"end"}\}$
    $\land\ donePhase \in [P \rightarrow \{\text{"start"}, \text{"end"}\}]$
    $\land\ messageCount \in [P \rightarrow Nat]$
```

The main property we want to establish is that, each round, for each message m of a well-behaved process, the messages of well-behaved processes from the previous round are all in m's coffer and consist of a strict majority of m's coffer.

```
$Safety \triangleq \forall p \in P \setminus B : \text{LET}\ m \triangleq pendingMessage[p]\ \text{IN}$
    $\land\ m \neq \langle\rangle$
    $\land\ m.round > 0$
    $\Rightarrow$
    $\land\ \forall m2 \in wellBehavedMessages : m2.round = m.round - 1 \Rightarrow m2.id \in m.coffer$
    $\land\ \text{LET}\ M \triangleq \{m2 \in wellBehavedMessages : m2.round = m.round - 1\}$
        $\text{IN}\quad 2 * Cardinality(M) > Cardinality(m.coffer)$
```

helper definition:

```
$messageWithID(id) \triangleq \text{CHOOSE}\ m \in messages : m.id = id$
```

Basic well-formedness properties:

```
$Inv1 \triangleq \forall m \in messages :$
    $\land\ \forall m2 \in messages : m \neq m2 \Rightarrow m.id \neq m2.id$
    $\land\ \forall id \in m.coffer :$
        $\land\ \exists m2 \in messages : m2.id = id$
        $\land\ messageWithID(id).round = m.round - 1$
```