MODULE *Sailfish*

Specification of the signature-free Sailfish++ consensus algorithm at a high level of abstraction.

We use a number of abstractions and simplifiying assumptions in order to expose the high-level principles of the algorithm clearly and in order to make model-checking of interesting configuations tractable:

1) Nodes read and write a global DAG. When a node transitions into a new round, it is provided with an arbitrary quorum of vertices from the previous round (except that, after GST, some additional assumptions apply).

2) We do not model timeouts. Instead, we assume that, every round r after GST, each correct node votes for the previous leader.

3) Byzantine nodes are allowed to create new DAG vertices arbitrarily, but only one per round.

4) We do not explicitly model committing based on $2f + 1$ first RBC messages.

5) There are no weak edges.

EXTENDS $Digraph$, $Integers$, $FiniteSets$, $Sequences$

CONSTANTS
$\quad N$ The set of all nodes
, F The set of *Byzantine* nodes
, R The set of rounds
, $IsQuorum(\_)$ Whether a set is a quorum (*i.e.* cardinality $\geq n$-f)
, $IsBlocking(\_)$ Whether a set is a blocking set (*i.e.* cardinality $\geq f + 1$)
, $Leader(\_)$ operator mapping each round to its leader
, GST the first round in which the system is synchronous

ASSUME $\exists\, n \in R : R = 1 \,..\, n$ useful rounds start at 1

For our purpose of checking safety and liveness of *Sailfish*, we do not need to model blocks of transactions. Instead, DAG vertices just consist of a node and a round.

$V \triangleq N \times R$ the set of possible DAG vertices
$Node(v) \triangleq v[1]$
$Round(v) \triangleq v[2]$

Next we define how we order DAG vertices when we commit a leader vertice

$LeaderVertice(r) \triangleq \langle Leader(r),\ r\rangle$

RECURSIVE $OrderSet(\_)$ arbitrarily order the members of the set S
$OrderSet(S) \triangleq$ IF $S = \{\}$ THEN $\langle\rangle$ ELSE
$\quad$ LET $e \triangleq$ CHOOSE $e \in S$: TRUE
$\quad$ IN $Append(OrderSet(S \setminus \{e\}),\ e)$

NOTE: CHOOSE is deterministic in TLA+,
i.e. CHOOSE $e \in S$: TRUE is always the same e if S is the same

```
RECURSIVE CollectLeaders(_, _, _)
CollectLeaders(vs, r, dag) ≜ IF vs = {} THEN ⟨⟩ ELSE
    LET children ≜ UNION {Children(v, dag) : v ∈ vs}
    IN  IF LeaderVertice(r) ∈ vs
          THEN Append(
              CollectLeaders(Children(LeaderVertice(r), dag), r − 1, dag),
              LeaderVertice(r))
          ELSE  CollectLeaders(children, r − 1, dag)

RECURSIVE OrderVertices(_, _)
OrderVertices(dag, leaderVertices) ≜
    IF leaderVertices = ⟨⟩ THEN ⟨⟩ ELSE
    LET l ≜ Head(leaderVertices)
        toOrder ≜ Descendants({l}, dag)
        prefix ≜ OrderSet(toOrder)
        remainingVertices ≜ Vertices(dag) \ (toOrder ∪ {l})
        remainingEdges ≜ {e ∈ Edges(dag) : {e[1], e[2]} ⊆ remainingVertices}
        remainingDAG ≜ ⟨remainingVertices, remainingEdges⟩
    IN  prefix ∘ ⟨l⟩ ∘ OrderVertices(remainingDAG, Tail(leaderVertices))

CommitLeader(v, dag) ≜
    LET leaderVertices ≜ CollectLeaders({v}, Round(v), dag)
    IN   OrderVertices(dag, leaderVertices)
```

Now we specify the algorithm in the *PlusCal* language.

```
--algorithm Sailfish{
  variables
      vs = {},   the vertices of the DAG
      es = {} ;  the edges of the DAG
  define {
      dag ≜ ⟨vs, es⟩
      NoLeaderVoteQuorum(r, delivered, add) ≜
          LET NoLeaderVote ≜ {v ∈ delivered : LeaderVertice(r − 1) ∉ Children(v, dag)}
          IN   IsQuorum({Node(v) : v ∈ NoLeaderVote} ∪ add)
   }
  process ( correctNode ∈ N \ F )
      variables
          round = 0,  current round; 0 means the node has not started execution
          log = ⟨⟩ ;  delivered log
  {
l0:    while ( TRUE ) {
          if ( round = 0 ) {  start the first round r = 1
              round := 1 ;
              vs := vs ∪ {⟨self, 1⟩}
           }
```

```
        else {   start a round r > 1
            with ( r ∈ {r ∈ R : r > round} )
            with ( delivered ∈ SUBSET {v ∈ vs : Round(v) = r − 1} ) {
                await IsQuorum({Node(v) : v ∈ delivered}) ;
                await LeaderVertice(r − 1) ∈ delivered ⇒
                          ∨ LeaderVertice(r − 2) ∈ Children(LeaderVertice(r − 1), dag)
                          ∨ NoLeaderVoteQuorum(r − 1, delivered, {}) ;
                if ( Leader(r) = self )
                    await   ∨ LeaderVertice(r − 1) ∈ delivered
                            ∨ NoLeaderVoteQuorum(r, delivered, {self}) ;
                round := r ;
                with ( newV = ⟨self, round⟩ ) {
                    vs := vs ∪ {newV} ;
                    es := es ∪ {⟨newV, pv⟩ : pv ∈ delivered} ;
                 } ;
                 commit if there is a quorum of votes for the leader of r − 2:
                if ( round > 1 )
                    with ( votesForLeader = {pv ∈ delivered : ⟨pv, LeaderVertice(round − 2)⟩ ∈ es} )
                    if ( IsBlocking({Node(pv) : pv ∈ votesForLeader}) )
                        log := CommitLeader(LeaderVertice(round − 2), dag)
             }
          }
       }
    }
```

Next comes our model of *Byzantine* nodes. Because the real protocol disseminates *DAG* vertices using reliable broadcast, *Byzantine* nodes cannot equivocate and cannot deviate much from the protocol (lest their messages be ignored).

```
    process ( byzantineNode ∈ F )
    {
l0:     while ( TRUE ) {
            with ( r ∈ R )
            with ( newV = ⟨self, r⟩ ) {
                when newV ∉ vs ;   no equivocation
                if ( r = 1 )
                     vs := vs ∪ {newV}
                else
                with ( delivered ∈ SUBSET {v ∈ vs : Round(v) = r − 1} ) {
                    await IsQuorum({Node(v) : v ∈ delivered}) ;   ignored otherwise
                    vs := vs ∪ {newV} ;
                    es := es ∪ {⟨newV, pv⟩ : pv ∈ delivered}
                 }
             }
         }
     }
 }
```

Next we define the safety and liveness properties

$Compatible(s1, s2) \triangleq$ whether the sequence $s1$ is a prefix of the sequence $s2$, or vice versa

$$\begin{array}{l} \text{LET } Min(n1, n2) \triangleq \text{IF } n1 \geq n2 \text{ THEN } n2 \text{ ELSE } n1 \text{IN} \\ \quad \forall i \in 1 \,..\, Min(Len(s1), Len(s2)) : s1[i] = s2[i] \end{array}$$

$$Agreement \triangleq \forall n1, n2 \in N \setminus F : Compatible(log[n1], log[n2])$$

$$\begin{array}{l} Liveness \triangleq \forall r \in R : r \geq GST \wedge Leader(r) \notin F \Rightarrow \exists B \in \text{SUBSET } (N \setminus F) : \\ \quad \wedge IsBlocking(B) \\ \quad \wedge \forall n \in B : round[n] \geq r + 2 \Rightarrow \exists i \in \text{DOMAIN } log[n] : log[n][i] = LeaderVertice(r) \end{array}$$

$$\begin{array}{l} Liveness \triangleq \forall r \in R : r \geq GST \wedge Leader(r) \notin F \Rightarrow \\ \quad \forall n \in N \setminus F : round[n] \geq r + 2 \Rightarrow \\ \qquad \exists i \in \text{DOMAIN } log[n] : log[n][i] = LeaderVertice(r) \end{array}$$

Finally we make a few auxiliary definitions used for model-checking with TLC

Basic typing invariant:

$$\begin{array}{l} TypeOK \triangleq \\ \quad \wedge \ \forall v \in vs : Node(v) \in N \wedge Round(v) \in Nat \setminus \{0\} \\ \quad \wedge \ \forall e \in es : \\ \qquad \wedge \ e = \langle e[1], e[2] \rangle \\ \qquad \wedge \ \{e[1], e[2]\} \subseteq vs \\ \qquad \wedge \ Round(e[1]) > Round(e[2]) \\ \quad \wedge \ \forall n \in N \setminus F : round[n] \in Nat \end{array}$$

Synchrony assumption: for each round r from GST onwards, if the leader of r is correct then every correct node votes for the round-r leader vertex in round
$r + 1$

$$\begin{array}{l} Synchrony \triangleq \forall r \in R : r \geq GST \wedge Leader(r) \notin F \Rightarrow \\ \quad \forall n \in N \setminus F : \\ \qquad \text{LET } v \triangleq \langle n, r + 1 \rangle \\ \qquad \text{IN } \quad \wedge \ v \in vs \\ \qquad\qquad \wedge \ \vee \ r = 1 \\ \qquad\qquad\quad \vee \ LeaderVertice(r - 1) \in Children(LeaderVertice(r), dag) \\ \qquad\qquad\quad \vee \ NoLeaderVoteQuorum(r, \{v2 \in vs : Round(v2) = r + 1\}, \{\}) \\ \qquad\qquad \Rightarrow LeaderVertice(r) \in Children(v, dag) \end{array}$$

We add the synchrony assumption to the specification

$$\begin{array}{l} SyncNext \triangleq (\exists\, self \in N \setminus F : correctNode(self) \wedge Synchrony') \\ \qquad\qquad \vee (\exists\, self \in F : byzantineNode(self)) \\ SyncSpec \triangleq Init \wedge \Box[SyncNext]_{vars} \end{array}$$

Next we define a constraint to stop the model-checker.

$$\begin{array}{l} Max(S) \triangleq \text{CHOOSE } x \in S : \forall y \in S : y \leq x \\ StateConstraint \triangleq \forall n \in N \setminus F : round[n] \in 0 \,..\, Max(R) \end{array}$$

Finally, we give some properties we expect to be violated (useful to get the model-checker to print interesting executions).

$$
\begin{array}{l}
\mathit{Falsy1} \;\triangleq\; \neg(\\
\quad \forall\, n \in N \setminus F : \mathit{round}[n] = \mathit{Max}(R) \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Falsy2} \;\triangleq\; \neg(\\
\quad \exists\, n \in N \setminus F : \mathit{Len}(\mathit{log}[n]) > 1 \\
)
\end{array}
$$