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) We model *Byzantine* nodes explicitly by assigning them an algorithm too. This algorithm should allow the worst attacks possible, but, while the author thinks this is true, there is no formal guarantee that this is the case. A more realistic model would allow *Byzantine* nodes to send completely arbitrary messages at any time, but this would make model-checking with TLC too hard.

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
  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
  LET $e \triangleq$ CHOOSE $e \in S$: TRUE
  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

$\text{RECURSIVE } \mathit{CollectLeaders}(\_, \_, \_)$
$\mathit{CollectLeaders}(\mathit{vs}, r, \mathit{dag}) \triangleq \text{IF } \mathit{vs} = \{\} \text{ THEN } \langle\rangle \text{ ELSE}$
$\quad \text{LET } \mathit{children} \triangleq \text{UNION } \{\mathit{Children}(v, \mathit{dag}) : v \in \mathit{vs}\}$
$\quad \text{IN} \quad \text{IF } \mathit{LeaderVertice}(r) \in \mathit{vs}$
$\qquad\quad \text{THEN } \mathit{Append}(\mathit{CollectLeaders}(\mathit{Children}(\mathit{LeaderVertice}(r), \mathit{dag}), r-1, \mathit{dag}), \mathit{LeaderVertice}(r))$
$\qquad\quad \text{ELSE } \mathit{CollectLeaders}(\mathit{children}, r-1, \mathit{dag})$

$\text{RECURSIVE } \mathit{OrderVertices}(\_, \_)$
$\mathit{OrderVertices}(\mathit{dag}, \mathit{leaderVertices}) \triangleq$
$\quad \text{IF } \mathit{leaderVertices} = \langle\rangle \text{ THEN } \langle\rangle \text{ ELSE}$
$\quad \text{LET } l \triangleq \mathit{Head}(\mathit{leaderVertices})$
$\qquad \mathit{toOrder} \triangleq \mathit{Descendants}(\{l\}, \mathit{dag})$
$\qquad \mathit{prefix} \triangleq \mathit{OrderSet}(\mathit{toOrder})$
$\qquad \mathit{remainingVertices} \triangleq \mathit{Vertices}(\mathit{dag}) \setminus (\mathit{toOrder} \cup \{l\})$
$\qquad \mathit{remainingEdges} \triangleq \{e \in \mathit{Edges}(\mathit{dag}) : \{e[1], e[2]\} \subseteq \mathit{remainingVertices}\}$
$\qquad \mathit{remainingDAG} \triangleq \langle \mathit{remainingVertices}, \mathit{remainingEdges} \rangle$
$\quad \text{IN} \quad \mathit{prefix} \circ \langle l \rangle \circ \mathit{OrderVertices}(\mathit{remainingDAG}, \mathit{Tail}(\mathit{leaderVertices}))$

$\mathit{CommitLeader}(v, \mathit{dag}) \triangleq$
$\quad \text{LET } \mathit{leaderVertices} \triangleq \mathit{CollectLeaders}(\{v\}, \mathit{Round}(v), \mathit{dag})$
$\quad \text{IN} \quad \mathit{OrderVertices}(\mathit{dag}, \mathit{leaderVertices})$

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

--algorithm *Sailfish* **{**
  variables
    $\mathit{vs} = \{\}$, the vertices of the *DAG*
    $\mathit{es} = \{\}$ **;** the edges of the *DAG*
  define {
    $\mathit{dag} \triangleq \langle \mathit{vs}, \mathit{es} \rangle$
  }
  process ($\mathit{correctNode} \in N \setminus F$ **)**
    variables
      $\mathit{round} = 0$, current round; 0 means the node has not started execution
      $\mathit{log} = \langle\rangle$ **;** delivered *log*
  {
$l0$:   **while (** TRUE **) {** keep starting new rounds
      $\mathit{round} := \mathit{round} + 1$ **;**
      with ($\mathit{newV} = \langle \mathit{self}, \mathit{round} \rangle$ **) {**
        if ($\mathit{round} = 1$ **)**
          $\mathit{vs} := \mathit{vs} \cup \{\mathit{newV}\}$
        else with ($\mathit{delivered} \in \text{SUBSET } \{v \in \mathit{vs} : \mathit{Round}(v) = \mathit{round} - 1\}$ **) {**
          await $\mathit{IsQuorum}(\{\mathit{Node}(v) : v \in \mathit{delivered}\})$ **;**
          await if delivered, leader vertice must be valid:

```
                    ∨ LeaderVertice(round − 1) ∉ delivered
                    ∨ round − 1 = 1
                    ∨ LeaderVertice(round − 2) ∈ Children(LeaderVertice(round − 1), dag)
                    ∨ LET NoVote ≜ {v ∈ delivered : LeaderVertice(round − 2) ∉ Children(v, dag)}
                      IN  IsQuorum({Node(v) : v ∈ NoVote}) ;
                if ( Leader(round) = self ) {
                    we must either include the previous leader vertice,
                    or we must witness a quorum of vertices not voting for the previous leader
                    await
                        ∨ LeaderVertice(round − 1) ∈ delivered
                        ∨ ∃ Q ∈ SUBSET delivered :
                            ∧ IsQuorum(Q)
                            ∧ ∀ n ∈ Q \ {self} : LET vn ≜ ⟨n, round⟩IN
                                ∧ vn ∈ vs
                                ∧ ⟨vn, LeaderVertice(round − 1)⟩ ∉ es ;
                 } ;
                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). Also note that creating a round-r vertice commutes to the left of actions of rounds greater than r and to the right of actions of rounds smaller than R, so we can, without loss of generality, schedule *Byzantine* nodes in the same “round-by-round” manner as other nodes.

```
    process ( byzantineNode ∈ F )
        variables round_ = 0 ;
    {
l0:     while ( TRUE ) {
            round_ := round_ + 1 ;
             maybe add a vertices to the DAG:
            either with ( newV = ⟨self, round_⟩ ) {
                if ( round_ = 1 )
                    vs := vs ∪ {newV}
                else
                with ( delivered ∈ SUBSET {v ∈ vs : Round(v) = round_ − 1} ) {
                    await IsQuorum({Node(v) : v ∈ delivered}) ;
```

$$vs := vs \cup \{newV\};$$
$$es := es \cup \{\langle newV, pv\rangle : pv \in delivered\}$$

}

} or skip ;

}

}

}

Next we define the safety and liveness properties

$$
\begin{array}{l}
Compatible(s1, s2) \triangleq \text{ whether the sequence } s1 \text{ is a prefix of the sequence } s2\text{, or vice versa} \\
\quad \text{LET } Min(n1, n2) \triangleq \text{IF } n1 \geq n2 \text{ THEN } n2 \text{ ELSE } n1 \text{ IN} \\
\qquad \forall i \in 1 \,..\, Min(Len(s1), Len(s2)) : s1[i] = s2[i]
\end{array}
$$

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

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

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

The round of a node, whether *Byzantine* or not

$$Round\_(n) \triangleq \text{IF } n \in F \text{ THEN } round\_[n] \text{ ELSE } round[n]$$

Basic typing invariant:

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

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 vertices in round
$r + 1$

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

Sequentialization constraints, which enforce a particular ordering of the actions. Because of how actions commute, the set of reachable states remains unchanged. Essentially, we schedule all nodes "round-by-round" and in lock-steps, with the leader last. This speeds up model-checking a lot.

Note that we must always schedule the leader last because, because of its relying on other nodes's vertices, its action does not commute to the left of the actions of other nodes.

An arbitrary ordering of the nodes, with the round leader last:

$NodeSeqLeaderLast(r) \triangleq \text{CHOOSE } s \in [1 \,..\, Cardinality(N) \to N] :$
$\quad \land \; s[Cardinality(N)] = Leader(r)$
$\quad \land \; \forall i, j \in 1 \,..\, Cardinality(N) : i \neq j \Rightarrow s[i] \neq s[j]$
$NodeIndexLeaderLast(n, r) \triangleq \text{CHOOSE } i \in 1 \,..\, Cardinality(N) : NodeSeqLeaderLast(r)[i] = n$

$SeqConstraints(n) \triangleq$
wait for all nodes to be at least in the round:
$\quad \land \forall n2 \in N : Round\_(n2) \geq Round\_(n)$
wait for all nodes with lower index to leave the round (leader index is always last):
$\quad \land \forall n2 \in N : NodeIndexLeaderLast(n2, Round\_(n)) < NodeIndexLeaderLast(n, Round\_(n))$
$\qquad \Rightarrow Round\_(n2) > Round\_(n)$

We add the sequentialization constraints and the synchrony assumption to the specification

$SeqNext \triangleq (\exists self \in N \setminus F : SeqConstraints(self) \land correctNode(self) \land Synchrony')$
$\qquad\qquad \lor (\exists self \in F : SeqConstraints(self) \land byzantineNode(self))$
$SeqSpec \triangleq Init \land \Box[SeqNext]_{vars}$

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

$Max(S) \triangleq \text{CHOOSE } x \in S : \forall y \in S : y \leq x$
$StateConstraint \triangleq \forall n \in N : Round\_(n) \in 0 \,..\, Max(R)$

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

$Falsy1 \triangleq \neg($
$\quad \forall n \in N : Round\_(n) = Max(R)$
$)$

$Falsy2 \triangleq \neg($
$\quad \exists n \in N \setminus F : Len(log[n]) > 1$
$)$