MODULE *Sailfish*

TODO: add skipping rounds

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

$\quad\text{IN}\quad Append(OrderSet(S \setminus \{e\}),\ e)$

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

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

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

$$
\begin{array}{l}
CommitLeader(v,\ dag) \triangleq \\
\quad \text{LET}\ leaderVertices \triangleq CollectLeaders(\{v\},\ Round(v),\ dag) \\
\quad \text{IN}\quad OrderVertices(dag,\ leaderVertices)
\end{array}
$$

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

$$
\begin{array}{l}
\ \ \textbf{--algorithm}\ Sailfish\textbf{\{} \\
\quad \textbf{variables} \\
\qquad vs = \{\},\quad \text{the vertices of the } DAG \\
\qquad es = \{\}\,\textbf{;}\quad \text{the edges of the } DAG \\
\quad \textbf{define \{} \\
\qquad dag \triangleq \langle vs,\ es\rangle \\
\qquad NoVoteQuorum(r,\ delivered) \triangleq \\
\qquad\quad \text{LET}\ NoVote \triangleq \{v \in delivered : LeaderVertice(r-1) \notin Children(v,\ dag)\} \\
\qquad\quad \text{IN}\quad IsQuorum(\{Node(v) : v \in NoVote\}) \\
\quad\ \textbf{\}} \\
\quad \textbf{process (}\ correctNode \in N \setminus F\ \textbf{)} \\
\qquad \textbf{variables} \\
\qquad\quad round = 0,\quad \text{current round; 0 means the node has not started execution} \\
\qquad\quad log = \langle\rangle\,\textbf{;}\quad \text{delivered } log \\
\quad \textbf{\{} \\
l0:\qquad \textbf{while (}\ \text{TRUE}\ \textbf{) \{}
\end{array}
$$

```
        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)
                        ∨ NoVoteQuorum(r − 1, delivered) ;
                if ( Leader(r) = self )
                    await   ∨ LeaderVertice(r − 1) ∈ delivered
                            ∨ NoVoteQuorum(r, delivered) ;
                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). 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} ) {
```

$\quad\quad\quad\quad\quad\quad$ **await** $IsQuorum(\{Node(v) : v \in delivered\})$ **;**
$\quad\quad\quad\quad\quad\quad$ $vs := vs \cup \{newV\}$ **;**
$\quad\quad\quad\quad\quad\quad$ $es := es \cup \{\langle newV, pv\rangle : pv \in delivered\}$
$\quad\quad\quad\quad\quad$ **}**
$\quad\quad\quad\quad$ **} or skip ;**
$\quad\quad\quad$ **}**
$\quad\quad$ **}**
}

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

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

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

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

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:

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

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 vertex in round
$r + 1$

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

We add the synchrony assumption to the specification

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

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).

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

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