# MODULE *Sailfish*

Specification of the *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*. Each round, each node gets to see an arbitrary quorum of vertices from the previous round (and, after *GST*, this quorum must include all correct vertices).

2) We do not model timeouts. Instead, before *GST*, nodes can non-deterministically increase their round number (inluding skipping rounds entirely); after *GST*, correct nodes can only increment their round number and only do so after acting upon a superset of the correct vertices of the previous round.

3) We do not model the *DAG* ordering procedure. Instead, we check that for every two committed vertices, there is a path in the *DAG* from the one with the higher round to the one with the lower round. Moreover, we define committed vertices using the global *DAG* and it is plausible that local *DAG* views would contain fewer committed vertices; so there is a potential for missing safety or liveness violations because of this.

4) We model *Byzantine* nodes explicitly by assigning them an algorithm. This algorithm should allow *Byzantine* nodes to do the worst possible, but there is no 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 too hard.

5) We do model committing based on $2f+1$ first *RBC* messages.

This version of the algorithm does not use "*no_vote*" messages.

EXTENDS *DomainModel*

CONSTANT

*GST* the first synchronous round (all later rounds are synchronous)

```
--algorithm Sailfish{
  variables
      $vs = \{\}$,   the vertices of the DAG
      $es = \{\}$;   the edges of the DAG
  define {
      $LeaderVertice(r) \triangleq \langle Leader(r), r \rangle$
      $ValidVerticeQuorums(r) \triangleq$   Quorums of valid vertices of round r
          $\{VQ \in \text{SUBSET } vs : \text{LET } NQ \triangleq \{Node(v) : v \in VQ\} \text{IN}$
                $\land\ NQ \in Quorum$
                $\land\ \forall v \in VQ :$
                    $\land\ Round(v) = r$
                    the leader vertice, if included, must be valid (i.e. if it does not point
                    to the previous leader vertice, then a quorum of votes must justify that):
                    $\land\ \lor\ \neg(r > 0 \land v = LeaderVertice(r) \land \langle v, LeaderVertice(r-1) \rangle \notin es)$
                        $\lor\ \exists VQ2 \in \text{SUBSET } VQ :$
                            $\land\ VQ2 \in Quorum$
                            $\land\ \forall v2 \in VQ2 : \langle v2, LeaderVertice(r-1) \rangle \notin es\}$
```

```
   }
  process ( $correctNode \in N \setminus F$ )
     variables $round = 0$ ;   current round
  {
l0:    while ( TRUE )
       either with ( $v = \langle self, round \rangle$ ) {
           add a new vertex to the $DAG$ and go to the next round
         $vs := vs \cup \{v\}$ ;
         if ( $round > 0$ )
         with ( $VQ \in ValidVerticeQuorums(round - 1)$ ) {
             from $GST$ onwards, each node receives all correct vertices of the previous round:
           when $round \geq GST \Rightarrow (N \setminus F) \subseteq \{Node(v2) : v2 \in VQ\}$ ;
           if ( $Leader(round) = self$ ) {
               we must either include the previous leader vertice,
               or a quorum of vertices not voting for the previous leader vertice
             when
                 $\lor\ LeaderVertice(round - 1) \in VQ$
                 $\lor\ \exists\, Q \in Quorum : \forall\, n \quad \in Q \setminus \{self\}$ : LET $vn \triangleq \langle n, round \rangle$IN
                    $\land\ vn \in vs$
                    $\land\ \langle vn, LeaderVertice(round - 1)\rangle \notin es$ ;
            } ;
           $es := es \cup \{\langle v, pv \rangle : pv \in VQ\}$ ;   add the edges
          } ;
         $round := round + 1$
       }
       or with ( $r \in \{r \in R : r > round\}$ ) {
           go to a higher round
         when $r \leq GST$ ;   from $GST$ onwards, correct nodes do not skip rounds
         $round := r$
       }
  }
```

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 ( $byzantineNode \in F$ )
     variables $round\_ = 0$ ;
  {
l0:    while ( TRUE ) {
           maybe add a vertices to the $DAG$:
         either with ( $v = \langle self, round\_ \rangle$ ) {
           $vs := vs \cup \{v\}$ ;
           if ( $round\_ > 0$ )
```

```
            with ( $vq \in ValidVerticeQuorums(round\_ - 1)$ ) {
                $es := es \cup \{\langle v, pv \rangle : pv \in vq\}$
             }
        } or skip ;
         go to the next round:
        $round\_ := round\_ + 1$
      }
    }
 }
```

Next we define the safety and liveness properties

```
$Committed(v) \triangleq$
    $\land\ v \in vs$
    $\land\ Node(v) = Leader(Round(v))$
    $\land\ \exists Bl \in Blocking : Bl \subseteq \{Node(pv) : pv \in Parents(v, es)\}$
    $\land\ \lor\ Round(v) = 0$
        $\lor\ LeaderVertice(Round(v) - 1) \in Children(v, es)$
        $\lor\ \exists Q \in Quorum : \forall n \in Q : \text{LET } vn \triangleq \langle n, Round(v) \rangle \text{IN}$
            $\land\ vn \in vs$
            $\land\ \langle vn, LeaderVertice(Round(v) - 1) \rangle \notin es$
```

```
$Safety \triangleq \forall v1, v2 \in vs :$
    $\land\ Committed(v1)$
    $\land\ Committed(v2)$
    $\land\ Round(v1) \leq Round(v2)$
    $\Rightarrow Reachable(v2, v1, es)$
```

```
$Liveness \triangleq \forall r \in R :$
    $\land\ r \geq GST$
    $\land\ Leader(r) \notin F$
     all correct $round - (r + 1)$ vertices are created:
    $\land\ \forall n \in N \setminus F : round[n] > r + 1$
    $\Rightarrow Committed(LeaderVertice(r))$
```

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

$$Quorum1 \triangleq \{Q \in \text{SUBSET } N : Cardinality(Q) \geq Cardinality(N) - Cardinality(F)\}$$
$$Blocking1 \triangleq \{Q \in \text{SUBSET } N : Cardinality(Q) > Cardinality(F)\}$$

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:

```
$TypeOK \triangleq$
    $\land\ \forall v \in vs : Node(v) \in N \land Round(v) \in Nat$
    $\land\ \forall e \in es :$
```

```
        $\land\ \ e = \langle e[1],\ e[2] \rangle$
        $\land\ \ \{e[1],\ e[2]\} \subseteq vs$
        $\land\ \ Round(e[1]) > Round(e[2])$
    $\land\ \ \forall\, n \in N : Round\_(n) \in Nat$
```

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, due to its use of other nodes's vertices, its action does not commute to the left of the actions of other nodes.

```
$SeqConstraints(n)\ \triangleq$
     wait for all nodes to finish previous rounds:
    $\land\, (Round\_(n) > 0 \Rightarrow \forall\, n2 \in N : Round\_(n2) \geq Round\_(n))$
     wait for all nodes with lower index to leave the round (leader index is always last):
    $\land\, \forall\, n2 \in N : NodeIndexLeaderLast(n2,\ Round\_(n)) < NodeIndexLeaderLast(n,\ Round\_(n)) \Rightarrow Round$

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

 Example assignment of leaders to rounds:
$ModLeader(r)\ \triangleq\ NodeSeq[(r \% Cardinality(N)) + 1]$

 Constraint to stop the model checker:
$StateConstraint\ \triangleq$
    LET $Max(S)\ \triangleq$ CHOOSE $x \in S : \forall\, y \in S : y \leq x$IN
        $\forall\, n \in N : Round\_(n) \in 0\, ..\, (Max(R) + 1)$

 Some properties we expect to be violated (useful to get the model-checker to print interesting executions):

$Falsy1\ \triangleq\ \neg($
    $\land\ Committed(\langle Leader(1),\ 1 \rangle)$
$)$

$Falsy2\ \triangleq\ \neg($
    $\land\ Committed(\langle Leader(0),\ 0 \rangle)$
    $\land\ \neg Committed(\langle Leader(1),\ 1 \rangle)$
    $\land\ \neg Committed(\langle Leader(2),\ 2 \rangle)$
    $\land\ Committed(\langle Leader(3),\ 3 \rangle)$
$)$
```