

Specification of the *Sailfish* consensus algorithm at a high level of abstraction.

We use a number of abstractions and simplifying assumptions in order to expose the high-level principles of the algorithm clearly and in order to make model-checking of interesting configurations 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 (including 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 possible views of the *DAG*, for every two committed vertices (one in each view), there is a path in the *DAG* from the one with the higher round to the one with the lower round.
- 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  $r$

$\{ VQ \in \text{SUBSET } vs : \text{LET } NQ \triangleq \{ Node(v) : v \in VQ \} \text{IN}$

$\wedge NQ \in \text{Quorum}$

$\wedge \forall v \in VQ :$

$\wedge Round(v) = r$

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):

$\wedge \vee \neg(r > 0 \wedge v = LeaderVertice(r) \wedge \langle v, LeaderVertice(r-1) \rangle \notin es)$

$\vee \exists VQ2 \in \text{SUBSET } VQ :$

$\wedge VQ2 \in \text{Quorum}$

$\wedge \forall v2 \in VQ2 : \langle v2, LeaderVertice(r-1) \rangle \notin es \}$

}

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 (  $VQ \in ValidVertexQuorums(round - 1)$  ) {
          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
               $\vee LeaderVertice(round - 1) \in VQ$ 
               $\vee \exists Q \in Quorum : \forall n \in Q \setminus \{self\} : \text{LET } vn \triangleq \langle n, round \rangle \text{ IN}$ 
                 $\wedge vn \in vs$ 
                 $\wedge \langle vn, LeaderVertice(round - 1) \rangle \notin es$ ;
            } ;
           $es := es \cup \{\langle v, pv \rangle : pv \in VQ\}$ ;  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 (  $vq \in ValidVertexQuorums(round_- - 1)$  ) {
           $es := es \cup \{\langle v, pv \rangle : pv \in vq\}$ 

```

```

    }
  } or skip ;
  go to the next round:
  round_ := round_ + 1
}
}
}

```

Next we define the safety and liveness properties

$Committed(v, view) \triangleq$  view intended to be a sub-DAG of the DAG  $es$   
 $\wedge v \in view$   
 $\wedge Node(v) = Leader(Round(v))$   
 $\wedge \exists Bl \in Blocking : Bl \subseteq \{Node(pv) : pv \in Parents(v, es) \cap view\}$   
 $\wedge \vee Round(v) = 0$   
 $\vee LeaderVertice(Round(v) - 1) \in Children(v, es)$   
 $\vee \exists Q \in Quorum : \forall n \in Q : LET vn \triangleq \langle n, Round(v) \rangle IN$   
 $\wedge vn \in view$   
 $\wedge \langle vn, LeaderVertice(Round(v) - 1) \rangle \notin es$

$Safety \triangleq \forall v1, v2 \in vs : \forall view1, view2 \in SUBSET vs :$   
 $\wedge SubDAG(view1, es)$   
 $\wedge SubDAG(view2, es)$   
 $\wedge Committed(v1, view1)$   
 $\wedge Committed(v2, view2)$   
 $\wedge Round(v1) \leq Round(v2)$   
 $\Rightarrow Reachable(v2, v1, es)$

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

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

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

The round of a node, whether *Byzantine* or not  
 $Round\_ (n) \triangleq IF n \in F THEN round\_ [n] ELSE round[n]$

Basic typing invariant:  
 $TypeOK \triangleq$   
 $\wedge \forall v \in vs : Node(v) \in N \wedge Round(v) \in Nat$   
 $\wedge \forall e \in es :$

$$\begin{aligned}
& \wedge e = \langle e[1], e[2] \rangle \\
& \wedge \{e[1], e[2]\} \subseteq vs \\
& \wedge Round(e[1]) > Round(e[2]) \\
& \wedge \forall n \in N : Round\_ (n) \in Nat
\end{aligned}$$

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:

$$\wedge (Round\_ (n) > 0 \Rightarrow \forall n2 \in N : Round\_ (n2) \geq Round\_ (n))$$

wait for all nodes with lower index to leave the round (leader index is always last):

$$\wedge \forall n2 \in N : NodeIndexLeaderLast(n2, Round\_ (n)) < NodeIndexLeaderLast(n, Round\_ (n)) \Rightarrow Round\_ (n) > Round\_ (n2)$$

$$SeqNext \triangleq (\exists self \in N \setminus F : SeqConstraints(self) \wedge correctNode(self)) \vee (\exists self \in F : SeqConstraints(self) \wedge byzantineNode(self))$$

$$SeqSpec \triangleq Init \wedge \Box [SeqNext]_{vars}$$

Example assignment of leaders to rounds:

$$ModLeader(r) \triangleq NodeSeq[(r \% Cardinality(N)) + 1]$$

Constraint to stop the model checker:

$$StateConstraint \triangleq$$

$$LET Max(S) \triangleq CHOOSE x \in S : \forall y \in S : y \leq x IN$$

$$\forall n \in N : Round\_ (n) \in 0 \dots (Max(R) + 1)$$

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

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

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

\_\_\_\_\_