```
──────────────────── MODULE syncCon3 ────────────────────
Synchronized consensus
EXTENDS Integers, Sequences, FiniteSets, TLC
CONSTANTS N, FAILNUM
ASSUME N ≤ 5 ∧ 0 ≤ FAILNUM ∧ FAILNUM ≤ 4
Nodes ≜ 1 . . N

--algorithm syncCon3
{ variable FailNum = FAILNUM,   Initialization block
           up = [n ∈ Nodes ↦ TRUE];   nodes are up
           pt = [n ∈ Nodes ↦ 0];   nodes are at round 0
           t = [n ∈ Nodes ↦ FALSE];   nodes are not terminated
           d = [n ∈ Nodes ↦ −1];   nodes are not decided
           mb = [n ∈ Nodes ↦ {}];   nodes have mailbox as emptyset

   define {
   SetMin(S) ≜ CHOOSE i ∈ S : ∀ j ∈ S : i ≤ j
   }

   macro MaybeFail( ) {
      if ( FailNum > 0 ∧ up[self] )
        { either
          { up[self] := FALSE; FailNum := FailNum − 1; }   Node may fail
            or skip; } ;   or not
   }

   fair process ( n ∈ Nodes )
   variable pmb = {}, Q = {};
   {
P: while ( up[self] ∧ ¬t[self] ) {
     if ( d[self] = −1 ) d[self] := self;   vote is set
     Q := Nodes;
PS:  while ( up[self] ∧ Q ≠ {} ) {   send vote to mb[p] one by one; this node can fail in between
         with ( p ∈ Q ) {
           mb[p] := mb[p] ∪ {d[self]};   skip for attacking generals impossibility
           Q := Q \ {p};
           MaybeFail() ;
          } ;
      } ;   end_while
      if ( up[self] ) pt[self] := pt[self] + 1;   move to next round
PR:   await (up[self] ∧ (∀ k ∈ Nodes : up[k] ⇒ pt[k] = pt[self]));   wait for others to move
      d[self] := SetMin(mb[self]);
      if ( pmb = mb[self] ) t[self] := TRUE;
      pmb := mb[self];
      mb[self] := {};
   } ;   end_if
```

```
  }  process
 }
  \* PR label critical for nonblocking;
 \* Remove up in PR label, to show the FLP result with asynchronous rounds!
  BEGIN TRANSLATION
VARIABLES FailNum, up, pt, t, d, mb, pc

  define statement
SetMin(S) ≜ CHOOSE i ∈ S : ∀ j ∈ S : i ≤ j

VARIABLES pmb, Q

vars ≜ ⟨FailNum, up, pt, t, d, mb, pc, pmb, Q⟩

ProcSet ≜ (Nodes)

Init ≜  Global variables
        ∧ FailNum = FAILNUM
        ∧ up = [n  ∈ Nodes ↦ TRUE]
        ∧ pt  = [n  ∈ Nodes ↦ 0]
        ∧ t  = [n ∈ Nodes ↦ FALSE]
        ∧ d = [n ∈ Nodes ↦  − 1]
        ∧ mb = [n ∈ Nodes ↦ {}]
         Process n
        ∧ pmb = [self ∈ Nodes ↦ {}]
        ∧ Q = [self ∈ Nodes ↦ {}]
        ∧ pc = [self ∈ ProcSet ↦ "P"]

P(self) ≜  ∧ pc[self] = "P"
           ∧ IF up[self] ∧ ¬t[self]
                 THEN ∧ IF d[self] = − 1
                             THEN ∧ d′ = [d EXCEPT ![self] = self]
                             ELSE ∧ TRUE
                                  ∧ d′ = d
                      ∧ Q′ = [Q EXCEPT ![self] = Nodes]
                      ∧ pc′ = [pc EXCEPT ![self] = "PS"]
                 ELSE ∧ pc′ = [pc EXCEPT ![self] = "Done"]
                      ∧ UNCHANGED ⟨d, Q⟩
           ∧ UNCHANGED ⟨FailNum, up, pt, t, mb, pmb⟩

PS(self) ≜  ∧ pc[self] = "PS"
            ∧ IF up[self] ∧ Q[self] ≠ {}
                  THEN ∧ ∃ p ∈ Q[self] :
                            ∧ mb′ = [mb EXCEPT ![p] = mb[p] ∪ {d[self]}]
                            ∧ Q′ = [Q EXCEPT ![self] = Q[self] \ {p}]
                            ∧ IF FailNum > 0 ∧ up[self]
                                  THEN ∧ ∨ ∧ up′ = [up EXCEPT ![self] = FALSE]
```

```
                                                   ∧ FailNum′ = FailNum − 1
                                                 ∨ ∧ TRUE
                                                   ∧ UNCHANGED ⟨FailNum, up⟩
                                        ELSE  ∧ TRUE
                                              ∧ UNCHANGED ⟨FailNum, up⟩
                             ∧ pc′ = [pc EXCEPT ![self] = "PS"]
                             ∧ pt′ = pt
                    ELSE  ∧ IF up[self]
                                THEN ∧ pt′ = [pt EXCEPT ![self] = pt[self] + 1]
                                ELSE  ∧ TRUE
                                      ∧ pt′ = pt
                          ∧ pc′ = [pc EXCEPT ![self] = "PR"]
                          ∧ UNCHANGED ⟨FailNum, up, mb, Q⟩
                 ∧ UNCHANGED ⟨t, d, pmb⟩

PR(self) ≜ ∧ pc[self] = "PR"
           ∧ (up[self] ∧ (∀ k ∈ Nodes : up[k] ⇒ pt[k] = pt[self]))
           ∧ d′ = [d EXCEPT ![self] = SetMin(mb[self])]
           ∧ IF pmb[self] = mb[self]
                THEN ∧ t′ = [t EXCEPT ![self] = TRUE]
                ELSE ∧ TRUE
                     ∧ t′ = t
           ∧ pmb′ = [pmb EXCEPT ![self] = mb[self]]
           ∧ mb′ = [mb EXCEPT ![self] = {}]
           ∧ pc′  = [pc EXCEPT ![self]  = "P"]
           ∧ UNCHANGED ⟨FailNum, up, pt, Q⟩

n(self) ≜ P(self) ∨ PS(self) ∨ PR(self)

Next ≜ (∃ self ∈ Nodes : n(self))
         ∨ Disjunct to prevent deadlock on termination
           ((∀ self ∈ ProcSet : pc[self] = "Done") ∧ UNCHANGED vars)

Spec ≜ ∧ Init ∧ □[Next]_vars
       ∧ ∀ self ∈ Nodes : WF_vars(n(self))

Termination ≜ ◇(∀ self ∈ ProcSet : pc[self] = "Done")

 END TRANSLATION

Agreement ≜ ∀ i, j ∈ Nodes : t[i] ∧ t[j] ⇒ (d[i] = d[j] ∧ d[i] ≠ −1)
NoTerm ≜ ¬∀ i    ∈ Nodes : up[i] ⇒ t[i]
SyncTerm ≜ ∀ i, j ∈ Nodes : t[i] ∧ t[j] ⇒ pt[i] = pt[j]
Term ≜ ◇∀ i ∈ Nodes : up[i] ⇒ t[i]
 Remember  ≜ □ [ (∀ j ∈ Nodes: v′[p] ≥ v[p]) ]_vars
```

Agreement. Two correct processes can not commit to different decision variables. $(i,j{:}ti$ tj $:di = dj)$ Validity (Nontriviality). If all initial values are equal, correct processes must decide on that value. $(k{::}\ (i{::}\ vi = k))\ \ (i\ {:}ti\ {:}di = vi)$ Termination. The system eventually terminates. true $(i{::}\ ti)$

Synchronous consensus Every process broadcasts (to all other processes, including itself) its initial value vi. In a synchronous network, this can be done in a single "round" of messages. After this round, each process decides on the minimum value it received. If no faults occur, this algorithm is correct. In the presence of a crash fault, however, a problem can arise. In particular, a problem may occur if a process crashes during a round. When this happens, some processes may have received its (low) initial value, but others may not have.

To address this concern, consider this simplifying assumption: say that at most 1 process can crash. How can we modify the algorithm to handle such a failure? Answer: by using 2 rounds. In $1st$ round, processes broadcast their own initial value. In $2nd$ round, processes broadcast the minimum value they heard. Each process then decides on the min value among all the sets of values it received in $2nd$ round. If the one crash occurs during the first round, the second round ensures that all processes have the same set of values from which to decide. Else, if the one crash occurs during the second round, the first round must have completed without a crash and hence all processes have the same set of values from which to decide.

The key observation is that if no crash occurs during a round, all processes have the same set of values from which to decide and they correctly decide on the same minimum value. Thus, to tolerate multiple crashes, say f , the protocol is modified to have $f + 1$ rounds of synchronous communication. Of course, this requires knowing f , an upper bound on the number of possible crash faults.