1 Consensus

We consider a shared-memory system, where a process might crash, by not taking any more steps. If it is not crashed, it is non-faulty.

Each process $p_i$ has a local variable $x_i$ which holds its input, and a local variable $y_i$ which will hold its output and is initialized to $\perp$. An algorithm for consensus has to satisfy:

**Agreement** In every execution, if $y_i$ and $y_j$ are not null then $y_i = y_j$, for every non-faulty processes $p_i, p_j$ (all the non-faulty processes decide the same value).

**Validity** In every execution, if there is a value $v$ such that $x_i = v$ for every process $p_i$, then if $y_j \neq \perp$ for a non-faulty process $p_j$, then $y_j = v$ (if all the inputs are $v$ then $v$ is the only possible decision value).

These are safety properties. The validity prevents a trivial solution of always deciding upon a predetermined default value. In case of binary consensus (inputs are 0 or 1) this validity condition is equivalent to requiring that the decision value is the input of some process.

A trivial solution to this problem is an algorithm in which no process ever decides. Therefore, we add a liveness condition:

**Termination** In every admissible execution, for every non-faulty process $p_i$, $y_i$ eventually gets a value other than null (every non-faulty process eventually decides).
2 Solo-Termination

There is no solution to the above consensus problem, therefore we will consider a relaxed termination requirement:

**Solo-termination** In every admissible execution, if there is a suffix in which only \( p_i \) takes steps, then \( p_i \) eventually decides.

We show an algorithm for solving consensus with solo-termination. It is based on a wrapper for the **safe-phase** procedure.

The **safe-phase** procedure satisfies the following conditions:

1. Agreement - If an invocation of **safe-phase** returns \( v \neq \perp \), then any other invocation of **safe-phase** returns either \( v \) or \( \perp \).
2. If an invocation of **safe-phase** returns a value \( v \neq \perp \) then \( v \) was an argument to this or a previous invocation of **safe-phase** (perhaps by another process).
3. Conditional Termination - If an invocation of **safe-phase** has a phase \( r \) which is larger than any phase of an invocation that starts before this one ends, then it returns a value \( v \neq \perp \).

Proof of the **safe-phase** properties can be found in the course book, chapter 17.

We show how together with the wrapper they imply the consensus requirements.

1. Agreement - Follows from the **safe-phase** agreement, since any value written to \( y \) must be returned from a **safe-phase** invocation.
2. Validity - Follows from the **safe-phase** validity, since any value written to \( y \) must be returned from a **safe-phase** invocation, therefore sent as an argument for a **safe-phase** invocation, therefore the input of some process.
3. Solo-Termination - If a process runs alone in a suffix of an execution, then this suffix has an invocation of **safe-phase** with phase \( r \) that is larger than any phase of an invocation that starts before this one ends, therefore by the conditional termination of **safe-phase** this invocation returns \( v \neq \perp \) and the process decides.
Algorithm 1 safe-phase procedure for processor $p_i$.

procedure safe-phase(integer $r$, value $x$)

    // Stage 1: choose that value with largest tag
    1: $R_i, phase := r$  // other fields of $R_i$ are written with their current values
    2: $maxPhase := 0$
    3: $chosenVal := x$

    // copy $R_j$ to a local variable
    4: for $j := 0$ to $n - 1$ do
    5:     $other := R_j$

    // if other.phase > $r$ then return $\perp$
    6:     if $other.phase > r$ then return $\perp$

    // if other.val $\neq \perp$ then
    7:     if $other.val \neq \perp$ then

    // maxPhase := other.tag
    8:     if $other.tag > maxPhase$ then
    9:         $maxPhase := other.tag$

    // chosenVal := other.val
    10:     $chosenVal := other.val$

    // Stage 2: check that no other processor started a larger phase
    11: $R_i := (r, chosenVal, r)$

    // if $R_j.phase > r$ then return $\perp$
    12: for $j := 0$ to $n - 1$ do
    13:     if $R_j.phase > r$ then return $\perp$

    // return $chosenVal$
    14: return $chosenVal$

Algorithm 2 wrapper code for $p_i$.

Initially $r = i$, $x_i = input$

1: while true
2:     $ans = safe-phase(r, x_i)$
3:     if $ans \neq \perp$ then $y_i = ans$
4:     $r = r + n$