

Solutions to Exercise 6

Problem 1. Let P be the problem of implementing C&S using base C&S objects, one of which can be non-responsive, and registers (non-faulty). Let Q be the problem of implementing consensus using registers in a system of $n > 1$ processes, one of which can crash (we know this problem to be impossible). We perform our proof by contradiction: assume there exists an algorithm A that solves P using k C&S objects, in a system of n processes (one of which can crash). If we find an algorithm B that solves problem Q , using A we have reached a contradiction.

From non-faulty C&S to consensus: We implement consensus in a system of $N = \max(k, n)$ processes, one of which can crash. A process p_i that proposes a value, writes the value in a register $R[i]$ and waits until a decided value is written in register D :

initially: $D = \perp, R[1, \dots, N] = \perp$

upon $propose_i(v)$ **do**

$R[i] \leftarrow v$
 wait until $D \neq \perp$
return D

Each of the n processes then runs the following task in parallel and uses the hypothetical correct C&S object implemented using algorithm A .

parallel task $Cons_i$

wait until some value $v \neq \perp$ is written in some register $R[j]$
 use algorithm A to call $CAS(\perp, v)$ on the non-faulty C&S object
 $D \leftarrow$ value returned by the CAS

From registers to non-responsive C&S: Each of n processes emulates one base C&S object. The processes share a 2-dimensional array CS of registers. When process i wants to invoke the CAS operation of C&S object x it invokes the following:

upon $CAS_x(oldval, newval)_i$ **do**

$CS[x][i] \leftarrow (\text{invocation}, oldval, newval)$
 wait until $CS[x][i] = (\text{response}, retval)$
return $retval$

Since one of the processes can fail, its corresponding C&S object becomes non-responsive. Each process i reads invocations from locations $CS[i][*]$ and applies them:

```

parallel task  $C_i$ 
  initially:  $q = \perp$  (local variable)
  while true do
    for  $j \leftarrow 1$  to  $n$  do
       $(type, oldval, newval) \leftarrow CS[i][j]$ 
      if  $type = \text{invocation}$  then
        if  $q = oldval$  then  $q \leftarrow newval$ 
       $CS[i][j] \leftarrow (\text{response}, q)$ 

```

Problem 2. We use $2t + 1$ base registers, so that always majority is correct. Read/write from/to majority of registers.

uses: $R[1, \dots, 2t + 1]$ – SWMR registers t of which can be non-responsive

```

upon  $write_1(v)$  do
   $ts \leftarrow ts + 1$ 
  invoke  $write_1(ts, v)$  on all  $R[1, \dots, 2t + 1]$ 
  wait for  $t + 1$  responses

upon  $read_i$  do
  invoke  $read_i(v)$  on all  $R[1, \dots, 2t + 1]$ 
  wait for  $t + 1$  responses
  return the value  $v$  with the highest timestamp  $ts$ 

```

The presented algorithm implements a regular SWMR register. However, a regular register can be transformed into an atomic one (see the lecture slides about register transformations).

Problem 3. The following algorithm solves the problem:

uses: C_0, C_1 – counters

```

upon  $propose(v)$  do
  while true do
     $(x_0, x_1) \leftarrow readCounters()$ 
    if  $x_0 > x_1$  then  $v \leftarrow 0$ 
    else if  $x_1 > x_0$  then  $v \leftarrow 1$ 
    if  $|x_0 - x_1| \geq n$  then return  $v$ 
     $C_v.inc()$ 

```

The $readCounters$ procedure atomically reads both counters C_0 and C_1 . It can be implemented as follows:

```

upon readCounters() do
  while true do
     $x_0 \leftarrow C_0.read()$ 
     $x_1 \leftarrow C_1.read()$ 
     $x'_0 \leftarrow C_0.read()$ 
    if  $x_0 = x'_0$  then return  $(x_0, x_1)$ 

```

Problem 4. The answer is yes. To justify this, we show linearizability and termination still hold. For linearizability, we need only to justify the return value of the replaced condition. Consider the first scan s which returns on this condition. (The “first” scan refers to when the scan starts.) Since the timestamp τ of the snapshot ret returned by s is no less than ts (which is obtained at the beginning of s), therefore the $wInc$ procedure which returns τ (denoted by $wInc_1$) cannot end before the $wInc$ procedure which returns ts (denoted by $wInc_2$) starts, by the property of the weak counter. In other words, $wInc_1$ ends no earlier than $wInc_2$ starts. Thus the call of scan (denoted by s_{ret}) inside the update which writes ret ends no earlier than s starts. I.e., two scans s and s_{ret} are concurrent. As a result, s can be linearized at the same point as s_{ret} . Since s_{ret} returns a linearizable value, then s also returns a linearizable value. We can extend the reasoning to infinity by induction. For termination, it is easy to see that now the implementation has more chances to return, and therefore must satisfy termination (as the original implementation satisfies termination).