Concurrent Algorithms November 20, 2018

Solutions to Exercise 6

Problem 1. We perform our proof by contradiction. Assume there exists an algorithm B that implements a C&S object using base C&S objects one of which can be non-responsive.

Figure 1 presents the idea behind our proof. We show that we can implement C&S objects one of which can be non-responsive just by using read-write registers (A). Furthermore, we show that we can get consensus out of using a single reliable C&S object (C). Therefore, if an algorithm B existed such that we could implement a reliable C&S object out of C&S objects where one can be non-responsive, we could also implement consensus out of read-write registers, hence a contradiction.

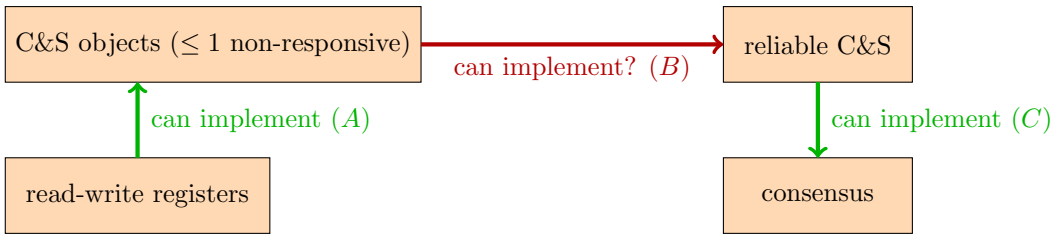

Figure 1: Idea behind the proof: If we could implement (B), we could get consensus out of resigters, a contradiction.

We consider a model with n processes where at most one process can crash. Furthermore, we consider that in every infinite execution, every non-crashed process takes an infinite number of steps. Notice that in such a model, we cannot devise an algorithm where consensus is guaranteed to terminate (see the "The Limitations of Registers" lecture) just by using read-write registers.

We present implementations for algorithms (A) and (B) below.

(A) 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**
  $CS[x][i] \leftarrow (\texttt{invocation}, oldval, newval)$
  wait until $CS[x][i] = (\texttt{response}, retval)$
  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 $type =$ `invocation` **then**

if $q = oldval$ **then** $q \leftarrow newval$

$CS[i][j] \leftarrow (\texttt{response}, q)$

Notice, that the presented algorithm does not provide wait-free implementations of $C\&S$ objects. However, in an infinite execution, all except the potentially non-responsive $C\&S$ will respond back.

(C) From non-faulty (i.e., reliable) C&S to consensus: 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: $D = \perp$, $R[1, \ldots, N] = \perp$

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 reliable C&S object.

parallel task $Cons_i$

wait until some value $v \neq \perp$ is written in some register $R[j]$

call $CAS(\perp, v)$ on the reliable C&S object

$D \leftarrow$ value returned by the CAS

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

uses: $R[1, \ldots, 2t+1]$ – 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 $R[1, \ldots, 2t+1]$

wait for $t + 1$ responses

upon $read_i$ **do**

invoke $read_i(v)$ on all $R[1, \ldots, 2t+1]$

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 trans-

formed 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).