Sapienza
Università di Roma
Dipartimento di Informatica

CONCURRENT SYSTEMS
LECTURE 9

Prof. Daniele Gorla

Universal Object

Given objects of type T and an object of type Z, is it possible to wait-free implement Z by using only objects of type T and atomic R/W registers?

The **type** of an object is

1. The set of all possible *values for states* of objects of that type
2. A set of *operations* for manipulating the object, each provided with a *specification*, i.e. a description of the conditions under which the operation can be invoked and the effect of the invocation

Here, we focus on types whose operations are

- *Total* : all operations can be invoked in any state of the object
- *Sequentially specified*: given the initial state of an obj, the behaviour depends only by the sequence of operations, where the output to every op. invocation only depends on the input arguments and the invocations preceding it.

→ formally, $\delta(s, op(args)) = \{\langle s_1,res_1\rangle,\dots, \langle s_k,res_k\rangle\}$

→ it is *deterministic* whenever k=1, for every s and every op(args)

An object of type T_U is **universal** if every other object can be wait-free implemented by using only onjects of type T_U and atomic R/W registers.

Consensus object

A **consensus object** is a *one-shot object* (i.e., an object such that any process can access it at most once) whose type has only one operation propose(v) such that:

1. (*Validity*) The returned value (also called *the decided value*) is one of the arguments of the propose (i.e., a *proposed value*) in one invocation done by a process (also called a *participant*)
2. (*Integrity*) every process decides at most once
3. (*Agreement*) The decided value is the same for all processes
4. (*Wait-freedom*) every invocation of propose by a correct process terminates

Conceptually, we can implement a consensus object by a register X, initialized at ⊥, for which the propose operation is atomically defined as

```
propose(v)     :=     if X = ⊥ then X ← v
                      return X
```

Universality of consensus holds as follows:

- Given an object O of type Z
- Each participant runs a local copy of O, all initialized at the same value
- Create a total order on the operations on O, by using consensus objects
- Force all processes to follow this order to locally simulate O

→ all local copies are consistent

First attempt (not wait-free)

Let us first concentrate on obj's with deterministic specifications (we will see how to handle non-determinism later on)

Process pi that wants to invoke operation op of obj Z with arguments args locally runs:

```
result_i ← ⊥
invoc_i ← ⟨op(args) , i⟩
wait  result_i ≠ ⊥
return result_i
```

Every process locally runs also a simulation of Z (stored in the local var z_i) s.t.

- z_i is initialized at the initial value of Z
- every process performs on z_i all the operations performed on Z by all processes, in the same order

→ need of consensus

First attempt (not wait-free)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let CONS be an unbounded array of consensus ojects and π_1 and π_2 respectively denote the first and the second element of a given pair (aka the first and second *projection*)

The local simulation of Z by pi is

```
k ← 0
zᵢ ← Z.init()
while true
   if invocᵢ ≠ ⊥ then
        k++
        execᵢ ← CONS[k].propose(invocᵢ)
        ⟨zᵢ , res⟩ ← δ(zᵢ , π₁(execᵢ))
        if π₂(execᵢ) = i then
             invocᵢ ← ⊥
             resultᵢ ← res
```

This solution is non-blocking but not wait-free

A wait-free construction

LAST_OP[1..n] : array of SWMR atomic R/W registers containing pairs init at ⟨⊥,0⟩ ∀i

$last\_sn_i$[1..n] : local array of the last op by pj executed by pi init at 0 ∀i,j

op(arg) by pi on Z

```
result_i ← ⊥
LAST_OP[i] ← ⟨op(args),
              last_sn_i[i]+1⟩
wait result_i ≠ ⊥
return result_i
```

local simulation of Z by pi

```
k ← 0
z_i ← Z.init()
while true
   invoc_i ← ε
   ∀ j = 1..n
      if π_2(LAST_OP[j])>last_sn_i[j]
      then invoc_i.append(
                    ⟨π_1(LAST_OP[j]),j⟩ )
   if invoc_i ≠ ε then
      k++
      exec_i←CONS[k].propose(invoc_i)
      for r=1 to |exec_i|
         ⟨z_i,res⟩ ← δ(z_i,π_1(exec_i[r]))
         j ← π_2(exec_i[r])
         last_sn_i[j]++
         if i=j then result_i←res
```

Lemma1: the construction is wait free

Proof:
Let pi invoke Z.op(par); to prove the Lemma, we need to show that eventually $result_i \neq \perp$

It suffices to prove that ∃ k s.t. CONS[k].propose(-) returns a list that contains ⟨“op(par)”,i⟩ and pi participates to the consensus:

a) Since pi increases the seq.numb. of LAST_OP[i], eventually all lists proposed contain that element forever
b) Eventually, pi will always find $invoc_i \neq \varepsilon$ and partecipates with an increasing sequence number k_i
c) Let k be the first consensus where (a) holds
d) By the properties of consensus, $exec_i$ contains ⟨“op(par)”,i⟩

Lemma2: all operations invoked by processes are executed exactly once.

Proof:

As shown before, every invocation is executed; we have to show that this cannot happen more than once.

If pi has inserted ⟨“op(par)”,sn⟩ in LAST_OP[i], it cannot invoke another operation until ⟨“op(par)”,sn⟩ appears in a list chosen by the consensus

→ the seq.numb. of LAST_OP[i] can increase only after ⟨“op(par)”,sn⟩ has been executed

Let k be the minimum such that CONS[k] contains ⟨“op(par)”,i⟩

Every pj that partecipate in the k-th consensus increases $last\_sn_j[i]$ before calculating a new proposal $invoc_j$

→ if π_2(LAST_OP[i]) is not changed, then the guard of the IF is false and ⟨“op(par)”,i⟩ is not appended in $invoc_j$

→ otherwise, we have a new invocation from pi (but this can only happen after that ⟨“op(par)”,sn⟩ has been executed)

Lemma3: the local copies si of all correct processes behave the same and comply with the sequential specification of Z.

Proof:

- The processes use CONS in the same order (CONS[1], CONS[2], …)
- Every consensus object returns the same list to all processes
- All processes scan the returned list from head to tail
  - → they apply the same sequence of op.'s to their local copies (that started from the same initial value)
  - → everything works because of determinism!

REMARK: bounded wait freedom does NOT hold:

- → if the background process suspends for a long time, when it wakes up it has an unbounded number of agreed lists to locally execute
- → this may arbitrarily delay an operation issued before the sleep

Solution for non-deterministic spec.'s

If the specifications of Z's operations are *non-deterministic*, then δ does not return one single possible pair after one invocation, but a set of possible choices.

How to force every process to run the very same sequence of operations on their local simulations?

1. Brute force solution: for every pair ⟨s , op(args)⟩, fix a priori one element of δ(⟨s , op(args)⟩) to be chosen

   → «cancelling» non-determinism

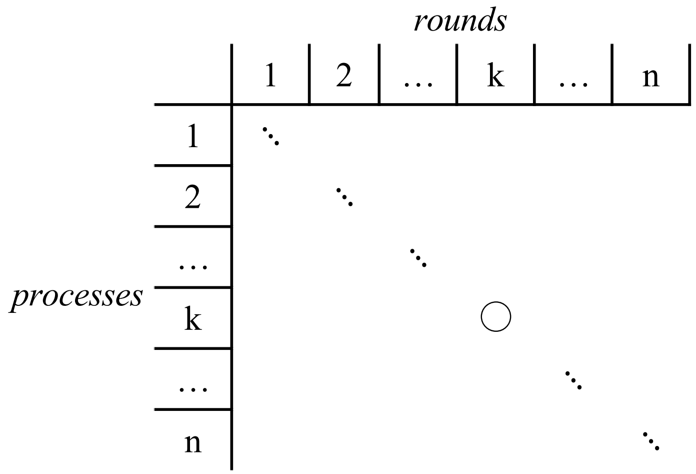
2. Additional consensus objects, one for every element of every list

3. Reuse the same consensus object: for all k, CONS[k] not only chooses the list of invocations, but also the final state of every invocation

   → the proposals should also pre-calculate the next state and propose one

Binary vs Multivalued Consensus

Binary consensus: just 2 possible proposals (say, 0 and 1)



<u>Multivalued consensus (with unbounded values)</u>

IDEA:

- we have n processes and n binary consensus rounds;
- at round k, all processes propose 1 if p_k has proposed something, 0 otherwise.
- If the decided value is 1, then decide p_k's proposal; otherwise, go to the next round.

```
PROP[1..n] array of n proposals, all init at ⊥
BC[1..n] array of n binary consensus objects
mv_propose(i,v) :=
   PROP[i] ← v
   for k=1 to n do
         if PROP[k] ≠ ⊥ then res ← BC[k].propose(1)
                        else res ← BC[k].propose(0)
         if res = 1 then return PROP[k]
   wait forever
```

Binary vs Multivalued Consensus

Validity, Agreement, Integrity, Termination:

- Let px the first process that writes a proposal
- every pi that participates to the consensus reads the other proposals after that it has written PROP[i]

  → all participants start their for loops after that px has written its proposal
- every pi that participates to the consensus finds PROP[x] $\neq \perp$ in their for loop
- BC[x] only receives proposals equal to 1
- Because of validity of binary consensus, BC[x] returns 1
- every pi that participates to the consensus receives 1 at most in its x-th iteration of the for
- Let y ($\leq$ x) be the first index such that BC[y] returns 1

  → BC[z] = 0 for all z < y
- Since all participating processes invoke the binary consensus in the same order, they all decide the value proposed by py and terminate

Binary vs Bounded Multivalued Consensus

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Multivalued consensus (with bounded values)

Let k be the number of possible proposals and h = $\lceil log_2 k \rceil$ be the number of bits needed to binary represent them (this value is known to all processes).

→ IDEA: decide bit by bit the final outcome

```
PROP[1..n][1..h] array of n h-bits proposals, all init at ⊥
BC[1..h] array of h binary consensus objects

bmv_propose(i,v) :=
   PROP[i] ← binary_repr_h(v)
   for k=1 to h do
        P ← {PROP[j][k] | PROP[j]≠⊥ ∧ PROP[j][1..k-1]=res[1..k-1]}
        let b be an element of P
        res[k] ← BC[k].propose(b)
   return value(res)
```

Binary vs Bounded Multivalued Consensus

- *Wait freedom*: trivial
- *Integrity*: trivial
- *Agreement*: by agreement of the h binary consensus objects
- *Validity*: for all k, we prove that, if dec is the decided value, then there exists a j such that pj is partecipating (i.e., PROP[j] $\neq \perp$) and dec[1..k] = PROP[j][1..k]

→ By construction, P contains the k-th bits of the proposals whose first (k-1) bits coincide with the first k-1 bits decided so far:

for every b $\in$ P, there exists a j such that PROP[j] $\neq \perp$,

PROP[j][1..k-1] = dec[1..k-1] and PROP[j][k] = b

→ whatever b $\in$ P is selected in the k-th binary consensus, there exists a j such that PROP[j] $\neq \perp$ and PROP[j][1..k] = dec[1..k]

→ by taking k = h, we can conclude.