Lecture notes

Johann Conus

Universal constructions

We saw that registers cannot be used to implement some objects, such as consensus. Although registers can be used to build objects weaker than consensus, weak renaming for example, the help of other objects, such as queues, make possible the construction of more powerful objects like consensus. We would like to find a way to build any type of object from a particular one without "assembling" multiple types of objects and without limitations. Such an object would be able to build any object.

So can we find a way to implement any object?
The idea is that we define an object T as being *universal* if, together with registers, it can be used to implement a wait-free, linearizable object of any type in a system of more than two processes. Queues on their own are not universal objects as we will explain that later.

Universal objects

An algorithm that can construct any wait-free object using a universal object is called a universal construction.

What is a universal object? We saw that registers cannot be solely used to implement some objects, such as consensus. But the use of other objects, queues in this case, make it possible.

We also saw that a register cannot implement queues. If it was able to do so, then it could also build consensus. Since it is not the case, we can conclude that registers are not universal objects.

Now, we claim that Compare & Swap (which is equivalent to consensus) is a universal object. It means that if we put Compare & Swap objects in a machine, we can build any object.

To see why Compare & Swap is a universal object, we introduce the notion of consensus number which every object has. A consensus number is the size of the system in which the object type can solve consensus. As we assume consensus as being universal, it is the size of the system in which the object type can be considered as being universal. Here is a table of some objects with their associated consensus number:

object	*number*
$compare\&swap$	∞
$test\&set$	2
$fetch\&increment$	2
$queue$	2
$register$	1

Table 1: Consensus number table

Objects like registers and queues are universal whithin a specific limited system. Queue is universal only in a system of two or less processes. However, it cannot guarantee universality in a larger system. This limitation of size makes the use of such objects not really appropriate to build other objects. We want to be able to build other objects independently of the size of the system. But we can see that Compare & Swap is universal no matter how many processes are in the system. So if we have $C\&S$, we can build any object.

For example, if we have a given deterministic object that each process has a copy of, we can guarantee wait-freedom because each process can execute some operation locally on his object and thus does not need to wait for some other process. But linearizability is not guaranteed. Linearizability can be achieved by making processes communicate between each other in order to ensure consistency. They communicate their values through registers and use consensus like a synchronization mechanism. When a consensus has decided, we know that all values since then are consistent on all processes.

To guarantee such a construction, all processes should do the operations togeter on all their copies, but in the same order. To guarantee this order, we need consensus.

How can we do that?

The idea is that processes use registers to "shout": a process informs other ones about its own operation by writing a value in a register so that others processes can read it. Then the consensus is used to determine which operations to execute. In other terms, each process proposes a set of operations and consensus decides which set of operations to execute in order to have all processes execute operations in the same order. What if a set has been missed? Propose again to next consensus. Every process will eventually get the same set of operations.

Universal algorithm

The algorithm works as follow:
We assume that every process has his own copy of some determinstic object so that wait-freedom is guaranteed. In order to communicate, the processes share an array of n SWMR registers ***Lreq***. It is used to inform all other processes about which requests need to be performed. They share a list ***Lcons*** of consensus objects aswell. Consensus is used to determine a total ordering of all requests.
Furthermore, we can identify the consensus integer: consensus object keeps a number in memory. When it has decided, this number is incremented and processes can know which decisions have already been made. Practically, this is just the index of the last consensus used in the array.
The algorithm combines the shared registers ***Lreq*** and the shared consensus ***Lcons*** to ensure that:

1. Every request invoked by a correct process is performed and as a result is eventually returned (wait-freedom)

2. Requests are executed in the same total order for all processes (linearizability)

3. This total order reflects the real-time order, i.e. the linearization point is within the interval of the operation

Every process also uses two local data structures:

LInv (list of invoked requests): What a process learns from other processes. This is the list of jobs that the process has to perform.

LPerf (list of performed requests): What a process has already performed. This is the list of jobs that the process has already executed.

LInv-LPerf = list of pending requests of the process (which need a result)
A process p_i execute three tasks in parallel:

1. Whenever a process has a new request of his own, it adds it to ***Lreq[i]***. It "shouts".

2. It periodically adds the new requests of every ***Lreq[j]*** into ***LInv***. It "listens to others".

3. While (***LInv-LPerf***) is not empty, it proposes this non-empty set (the requests that p_i still need to execute) to a new ***Lcons*** (thus incrementing the consensus integer). When ***Lcons*** decides, p_i performs the set of requests decided that are not in ***LPerf*** (p_i does not execute requests that already have been performed). Then for each request executed this way, p_i returns the result of the request if it is in ***Lreq[i]*** and put it in ***LPerf***.

Processes keep "shouting" what jobs they still need to perform, in this manner there is no erasure and so no risk of information loss.

Here is a pseudo-code version of the algorithm (note that a request consist of an operation specifier and arguments on the object):

```
global variables:
LCons = 0;
LReq[n] = 0;
Local execution:
LPerf=0;
LInv=0;
 execute in parallel:
task 1
  LReq[i].enqeue(newrequest);

task 2
  for int j = 0 upto n do
    LInv.concatenate(LReq[j]);
  end

task 3
  while LInv-Perf ≠ ∅ do
    LCons.propose(LInv-Perf);
    Req = LCons.decide();
    foreach request ∈ Req do
      result = object.execute(request);
      if request ∈ LReq[i] then
        LPerf.enqueue(request);
        return result;
      end
    end
  end
```

Consensus is used to ensure total ordering: when a set of requests is decided all processes execute these requests in the same order. If some request is still pending for some process, it will be proposed to the next consensus until it has been decided in some set and thus executed by all processes. In the end, all processes will eventually execute the same set of requests in the same order.

When order is decided, all processes can use the same deterministic objects. We know that this way, they all will perform their jobs in the same way as it is deterministic.

Non-deterministic objects

What if the object is non-deterministic?

We lose linearizability because it is not the same locally.

What should we do? Basically, there are two strategies:

1. transform the system into a deterministic one, but this is "cheating"
2. do a consensus on result

The second solution is better. We do not have to agree only on requests but on states. Such a state includes requests, replies and update (update is the new state). It is a quite heavy solution but it preserves non-determinism.

Construction is done the same way as the one for deterministic objects, except that processes first execute their own request locally and store the result (as it is a non-deterministic object, two processes may get different results for a same request). Then processes propose a set of state (including request, result they computed and new state) instead of only the request. Consensus will make them agree on one result so that all processes get consistent results.

Here is the modified version the algorithm:

```
global variables:
LCons = 0;
LReq[n] = 0;
Local execution:
LPerf=0;
LInv=0;
 execute in parallel:
task 1
    (newresult, newupdate) = object.execute(newrequest);
    LReq[i].enqeue(newrequest,newresult,newupdate);

task 2
    for int j = 0 upto n do
        LInv.concatenate(LReq[j]);
    end

task 3
    while LInv-Perf ≠ ∅ do
        LCons.propose(request,result,update);
        (req,res,up) = LCons.decide();
        object.update(up);
        if req ∈ LReq[i] then
            LPerf.enqueue(req,res,up);
            return res;
        end
    end
```