Object Implementation Out of Faulty Base Objects
Lecture Notes

1 Introduction

In the previous lectures, we assume that the base objects are always correct and they never fail. In those lectures, only processes could crash. Now, we want to see what will happen if the base objects become faulty. There are two types of object failures:

**Responsive.** The object only fails once; but when it fails, it fails forever. If a process calls an operation on a responsive failed object, it will return a specified value ($\bot$) and announce the process that it is faulty.

**Non-responsive.** In this type of failure, if a process calls an operation on a non-responsive failed object, it will never reply to that process. So, in the asynchronous model, it is impossible to distinguish a non-responsive failed object from a slow object.

Then, in this lecture, we show how to implement non-faulty objects out of faulty base objects.

2 Algorithm 1: SWMR register from responsive base objects

In this section we show how to implement a failure-free SWMR register out of $(t+1)$ SWMR base responsive failure-prone registers. ($t$ is the maximum number of failed base registers that the algorithm can tolerate.)

**Solution.** SWMR register defines two operations: read and write. In the following subsections, we show how to implement these two operations.

2.1 Write operation

As mentioned earlier SWMR register is implemented out of $(t+1)$ base registers. For write operation, we write to all $(t+1)$ base registers and return OK. We do not know which one of the base registers will fail; We only know that the maximum number of the failed registers is $t$.

```plaintext
Write(v)
For j=1 to (t+1) do
Reg[i].write(v);
return(ok)
```

In this method, when we write to $(t+1)$ registers, we do not wait to receive ok or $\bot$ from those base registers. We only know that at least one base register will have the new value.

2.2 Read operation

For implementing the read operation, we start reading from the last base register and go downward. As soon as we received a returned value ($\neq \bot$), we stop reading and return that value.
Read(v)
For j=(t+1) to 1 do
  v := Reg[j].read();
  if v ≠⊥ then return(v)

If we start reading from the first base register and we go upward, the implemented SWMR register is not atomic. 

**Reason.** Assume we have one slow write operation which is written in the first base register and start writing to the second base register; concurrent with this write, two sequential read operations come. The first read will read from the first base register and return the new value. If before the second read starts, the first base register fails; The second read will read from the second base register and return old value. So, the implemented SWMR register will not be atomic.

The suggested algorithm does not work for not-responsive base register failures. Because in read algorithm, each read from a base register must wait until receiving a reply from that register. If that register becomes faulty, it never replies and so the implemented SWMR register will not be wait-free.

3 Algorithm 2: SWSR register from non-responsive base objects

In this section we show how to implement a failure-free SWSR register out of (2t+1) SWSR base non-responsive failure-prone registers. (t is the maximum number of failed base registers that the algorithm can tolerate.)

**Solution.** SWSR register defines two operations: read and write. In the following subsections we show how to implement these two operations.

3.1 Write operation

For implementing write operation, we use timestamp wSeq. When we want to write value v, we increment the timestamp wSeq and write concurrently the pair-value (wSeq, v) to all (2t+1) base registers. Then, we wait until receiving at least (t+1) ok from these base registers. In the worst case t base registers will be failed and so we will receive (t+1) ok. Then we return ok. When we return ok, we are sure that the majority of base registers are received new value.

```
Init: seq := 1
Write(v)
  wSeq := wSeq + 1;
  For j = 1 to (2t+1) do ||
    Reg[j].write(wSeq,v);
  ▷ wait until a majority of oks are returned <
  return(ok)
```

3.2 Read operation

For implementing read operation, we first initialize the read pair-value to (-1,⊥). Then we read concurrently from all (2t+1) base registers and wait until receiving at least (t+1) reply from these base registers. Then the value v with the highest timestamp s will be returned.

```
Init: (sn,val) := (-1,⊥);
Read()
  For j = 1 to (2t+1) do ||
    (s,v) := Reg[j].read();
    (sn,val) := (s,v) with the highest s from majority, including (sn,val)
  return(val)
```
Note. This algorithm could not be used for implementing SWMR register, because the implemented SWMR register will not be atomic.

Reason. Assume we have 3 base registers and all of them are correct and a write operation comes and starts writing to all of these 3 base registers. If one of these base registers is fast and the other two registers are slow. After a while, the fast base register will have the new value, but two slow base registers still have the old value. Now assume two sequential read operations come. The first read receives the answer from one slow base register and fast base register and so returns the new value. But, the second read receives reply from two slow base registers and so returns the old value. It means that the implemented SWMR register is not atomic. If readers could also write the value they read, the modified algorithm would be atomic.

4 Algorithm 3: C&S object from responsive base objects

In this section we show how to implement a failure-free C&S object out of (t+1) C&S base responsive failure-prone objects. (t is the maximum number of failed base registers that the algorithm can tolerate.)

```
C&S(v)
    r := v;
    For j = 1 to (t+1) do
        temp := CS[j].C&S(r);
        if temp ≠ ⊥ then r := temp;
    return(r)
```