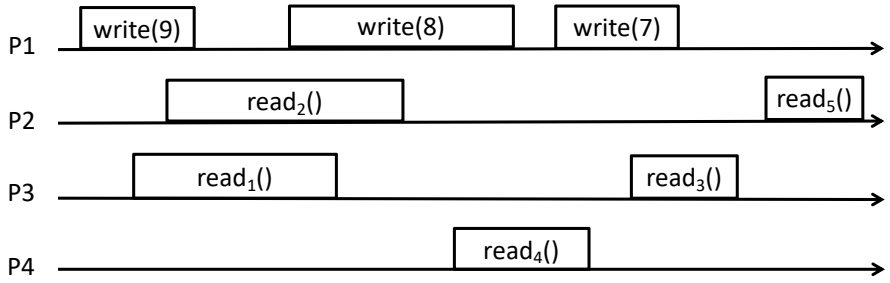
Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2023/2024

Week 8 – Exercises
November 15th, 2023

Ex 1: Consider the partial execution depicted in the Figure



Answer to the following questions:

1. Define ALL the values that can be returned by read operations (Rx) assuming that the run refers to a regular register.
2. Define ALL the values that can be returned by read operations (Rx) assuming that the run refers to an atomic register.
3. Assign to each read operations (Rx) a return value that makes the execution linearizable.

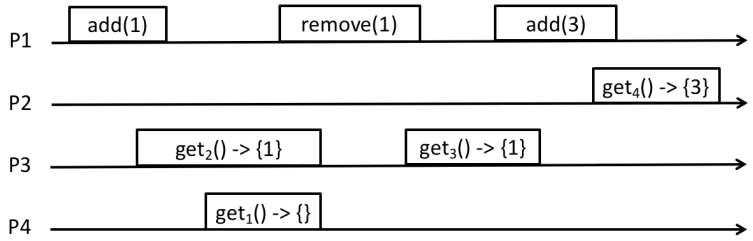
Ex 2: Consider a distributed system composed of *n* processes $\prod=\{p_1, p_2 \ldots p_n\}$ with unique identifiers that exchange messages through perfect point-to-point links. $\prod$ is not known to the processes. Processes are connected through a directed ring (i.e., each process p_i can exchange messages only with processes and $p_{i+1(\text{mod } n)}$). Processes may crash and each process is equipped with a local perfect oracle (having the interface *new_next(p)*) reporting a new neighbor when the previous one is failing.
Write the pseudo-code of an algorithm implementing a Leader Election primitive at every process p_i.

Ex 3: Consider a set object that can be accessed by a set of processes. Processes may invoke the following operations on the object:

- add(v): it adds the value v in to the set
- remove(v) it removes the value v from the set
- get(): it returns the content of the set.

Informally, every get() operation returns all the values that have been added before its invocation and that have not been removed by any remove().
For the sake of simplicity, assume that a value can be added/removed just once in the execution.

Consider the distributed execution depicted in the Figure



Answer to the following questions:

1. Is the proposed execution linearizable? Motivate your answer with examples.
2. Consider now the following property: “every get() operation returns all the values that have been added before its invocation and that have not been removed by any remove(). If an add(v)/remove(v) operation is concurrent with the get, the value v may or may be not returned by the get()”.
   Provide an execution that satisfy get validity and that is not linearizable.

Ex 4: Consider the algorithm shown in the Figure

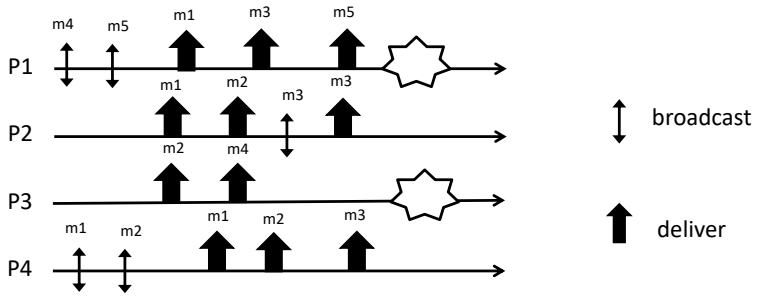
```
upon event ⟨ Init ⟩ do
      delivered := ∅; pending := ∅; correct := Π;
      forall m do ack[m] := ∅;

upon event ⟨ urb, Broadcast | m ⟩ do
      pending := pending ∪ {(self, m)};
      trigger ⟨ beb, Broadcast | [DATA, self, m] ⟩;

upon event ⟨ beb, Deliver | p, [DATA, s, m] ⟩ do
      ack[m] := ack[m] ∪ {p};
      if (s, m)∉ pending then
         pending := pending ∪ {(s, m)};
         trigger ⟨ beb, Broadcast | [DATA, s, m] ⟩;
```

```
upon event ⟨◊P,Suspect |p⟩do
      correct := correct \ {p};

upon event ⟨◊P,Restore |p⟩do
      correct := correct ∪ {p};

function candeliver(m) returns Boolean is
      return (correct ⊆ ack[m]);

upon exists (s, m) ∈ pending such that
candeliver(m) do
      delivered := delivered ∪ {m};
      trigger ⟨ urb, Deliver | s, m ⟩;
```

Assuming that the algorithm is using a Best Effort Broadcast primitive and an Eventually Perfect Failure Detector ◊P discuss if the following properties are satisfied or not and motivate your answer

- *Validity*: If a correct process p broadcasts a message m, then p eventually delivers m.
- *No duplication*: No message is delivered more than once.
- *No creation*: If a process delivers a message m with sender s, then m was previously broadcast by process s.
- *Uniform agreement*: If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

Ex 5: Consider the execution depicted in the Figure

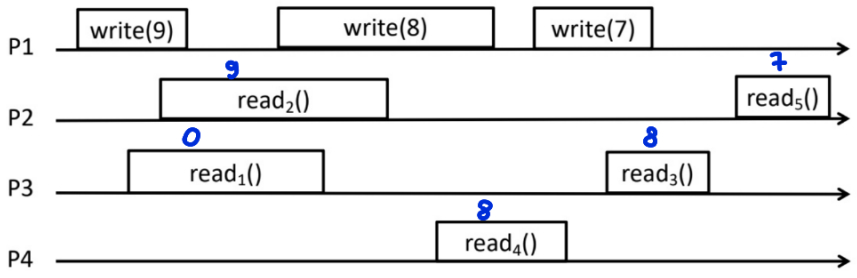


Answer to the following questions:

1. Which is the strongest TO specification satisfied by the proposed run? Motivate your answer.
2. Does the proposed execution satisfy Causal order Broadcast, FIFO Order Broadcast or none of them?
3. Modify the execution in order to satisfy TO(UA, WUTO) but not TO(UA, SUTO).
4. Modify the execution in order to satisfy TO(NUA, WNUTO) but not TO(UA, WNUTO).

NOTE: In order to solve point 3 and point 4 you can only add messages and/or failures.

Ex 1: Consider the partial execution depicted in the Figure



Answer to the following questions:

1. Define ALL the values that can be returned by read operations (Rx) assuming that the run refers to a regular register.
2. Define ALL the values that can be returned by read operations (Rx) assuming that the run refers to an atomic register.
3. Assign to each read operations (Rx) a return value that makes the execution linearizable.

1) Regular register ⇒ R_1 : 0,9,8
R_2 : 0,9,8
R_3 : 8,7
R_4 : 9,8,7
R_5 : 7

WRITE:
9 8 7

2) Atomic register ⇒ R_1 : 0,9,8 } concurrency
R_2 : 0,9,8 }
R_3 : 8,7 or 7
R_4 : 9,8,7 or 8,7
R_5 : 7

R_4 depends on R_1 and R_2: if R_2 has (0,9) then R_4 can have (9,8,7), otherwise (8,7). (R_1 the same, the two processes are concurrent)

R_3 depends on R_4: if R_4 has (9,8) then R_3 can have (8,7) otherwise (3).

3) Linearizable

$R_1 = 0$ $R_2 = 9$ $R_3 = 8$ $R_4 = 8$ $R_5 = 7$

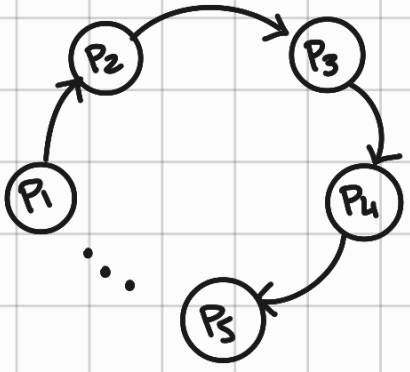
Ex 2: Consider a distributed system composed of n processes $\prod=\{p_1, p_2 \dots p_n\}$ with unique identifiers that exchange messages through perfect point-to-point links. $\prod$ is not known to the processes. Processes are connected through a directed ring (i.e., each process p_i can exchange messages only with processes and $p_{i+1(\text{mod } n)}$). Processes may crash and each process is equipped with a local perfect oracle (having the interface *new_next(p)*) reporting a new neighbor when the previous one is failing.
Write the pseudo-code of an algorithm implementing a Leader Election primitive at every process p_i.

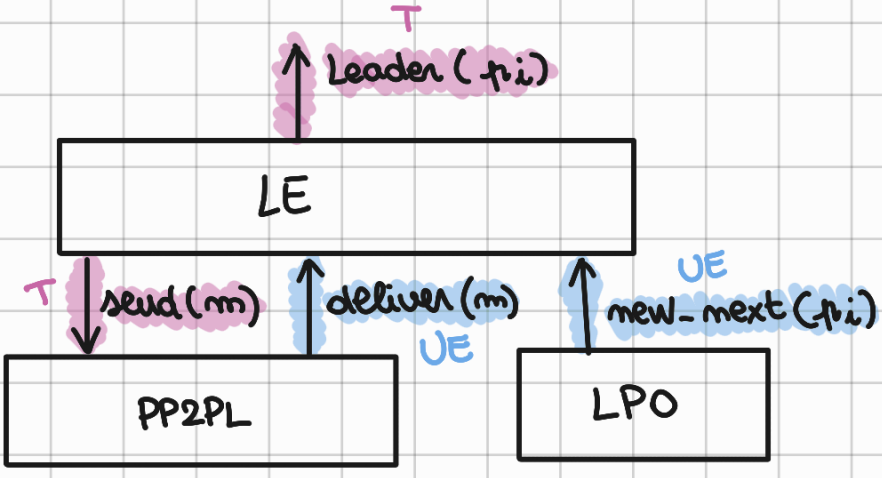
PP2PL

RING

LOCAL PERFECT ORACLE

new_next(p)





upon event < le, INIT > do
    leader := ⊥
    next := $p_{i+1 \text{ (mod N)}}$

upon event < lpo, new_next(p) > do
    if leader > p AND my_id > p
        leader = my_id
        next = p
        trigger leader(leader)

```
            trigger < pp2pl, pp2pl_send ({newLeader}, leader)
            to next
        else
            next = p
            trigger < pp2pl, pp2pl_send ({newLeader}, leader)

upon event < pp2pl, pp2pl_deliver ({newLeader}, l) > do
        if leader ≠ l
            leader = l
            trigger leader (leader)
            trigger < pp2pl, pp2pl_send ({newLeader}, leader)
            to next
```

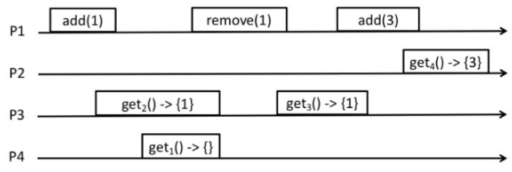
Correzione Farina:
2) Salve, ci sono alcuni problemi nella soluzione. Viene detto nel testo che pigreco non è noto, ma nell'init viene dichiarata una variabile alive (mai utilizzata) con valore iniziale uguale a pigreco. Come si avvia il protocollo se non avviene un crash? Cosa succede al suo protocollo se due processi non collegati crashano?

Ex 3: Consider a set object that can be accessed by a set of processes. Processes may invoke the following operations on the object:

- add(v): it adds the value v in to the set
- remove(v) it removes the value v from the set
- get(): it returns the content of the set.

Informally, every get() operation returns all the values that have been added before its invocation and that have not been removed by any remove().
For the sake of simplicity, assume that a value can be added/removed just once in the execution.

Consider the distributed execution depicted in the Figure



Answer to the following questions:

1. Is the proposed execution linearizable? Motivate your answer with examples.
2. Consider now the following property: "every get() operation returns all the values that have been added before its invocation and that have not been removed by any remove(). If an add(v)/remove(v) operation is concurrent with the get, the value v may or may be not returned by the get()".
   Provide an execution that satisfy get validity and that is not linearizable.

1) add(1), $get_2() \to \{1\}$, remove(1), $get_1() \to \{\}$, $get_3() \to \{1\}$, add(3), $get_4() \to \{3\}$

We have a problem with the $get_3() \to \{1\}$ because it return 1 after the remove and after get_1. So, it is not correct, respect to the sequencial specification of the set.
We can try to put the get_3 before the remove but it violates the relationship between get_1 and get_2.
We can conclude that the execution is not linearizable.

2) add(1), $get_2() \to \{1\}$, remove(1), $get_1() \to \{\}$, $get_3() \to \{1\}$, add(3), $get_4() \to \{3\}$

Ex 4: Consider the algorithm shown in the Figure

```
upon event ⟨ Init ⟩ do
    delivered := ∅; pending := ∅; correct := Π;
    forall m do ack[m] := ∅;

upon event ⟨ urb, Broadcast | m ⟩ do
    pending := pending ∪ {(self, m)};
    trigger ⟨ beb, Broadcast | [DATA, self, m] ⟩;

upon event ⟨ beb, Deliver | p, [DATA, s, m] ⟩ do
    ack[m] := ack[m] ∪ {p};
    if (s, m)∉ pending then
        pending := pending ∪ {(s, m)};
        trigger ⟨ beb, Broadcast | [DATA, s, m] ⟩;

upon event ⟨◊P,Suspect |p⟩do
    correct := correct \ {p};

upon event ⟨◊P,Restore |p⟩do
    correct := correct ∪ {p};

function candeliver(m) returns Boolean is
    return (correct ⊆ ack[m]);

upon exists (s, m) ∈ pending such that
candeliver(m) do
    delivered := delivered ∪ {m};
    trigger ⟨ urb, Deliver | s, m ⟩;
```

Assuming that the algorithm is using a Best Effort Broadcast primitive and an Eventually Perfect Failure Detector ◊P discuss if the following properties are satisfied or not and motivate your answer

- *Validity*: If a correct process p broadcasts a message m, then p eventually delivers m.
- *No duplication*: No message is delivered more than once.
- *No creation*: If a process delivers a message m with sender s, then m was previously broadcast by process s.
- *Uniform agreement*: If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

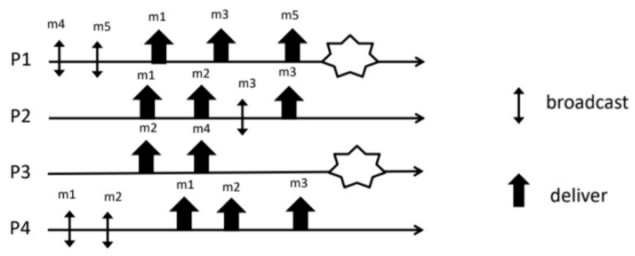
1) Validity: is satisfied because when we broadcast a message, the p is added in pending (store the msg locally). Then the msg is delivered only when all correct process sent the ack. Althought can happen that a correct process is wrongly suspected as "crashed" and the msg is delivered.

2) No duplication: is not satisfied because there isn't the "m ∉ delivered" in the last "upon exist...". In the case of suspect / restore can happen that a process wrongly suspected is restored and the deliver = TRUE, so the msg is delivered again.

3) No creation: yes, we use the pending's set to store locally the msgs broadcasted and only if $(s, m) \in$ pending we can deliver.

4) Uniform agreement: is not satisfied, because in the suspect / restore part there can be processes that, wrongly suspected, have different deliver respect to correct processes.

Ex 5: Consider the execution depicted in the Figure

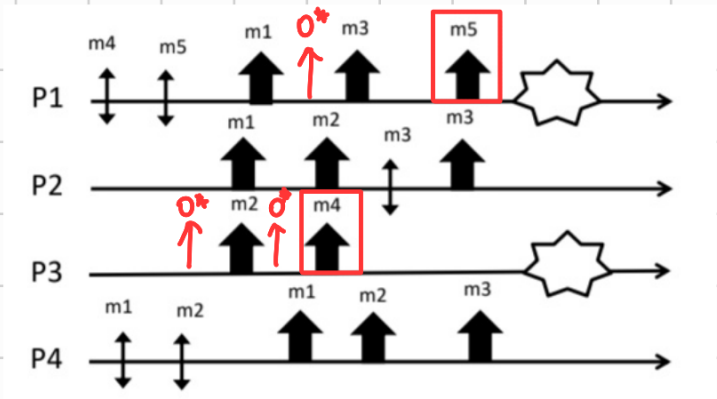


Answer to the following questions:

1. Which is the strongest TO specification satisfied by the proposed run? Motivate your answer.
2. Does the proposed execution satisfy Causal order Broadcast, FIFO Order Broadcast or none of them?
3. Modify the execution in order to satisfy TO(UA, WUTO) but not TO(UA, SUTO).
4. Modify the execution in order to satisfy TO(NUA, WNUTO) but not TO(UA, WNUTO).

NOTE: In order to solve point 3 and point 4 you can only add messages and/or failures.

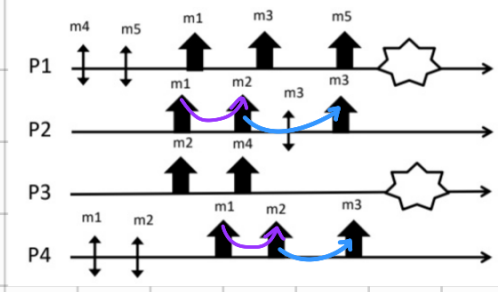
1)



O*: omission
□: new delivery

TO(NUA, WUTO) because the faulty processes are not a subset of the correct (in p_1 there is m_5; in p_3 there is m_4), also we have some omission (> 1) and the order is not strong.

2)



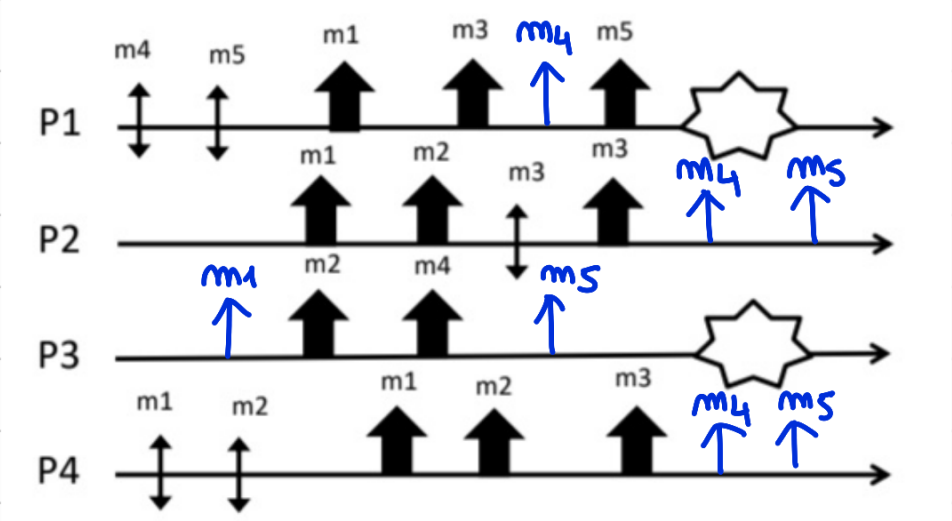
FIFO: $m_4 \rightarrow m_5$

$m_1 \rightarrow m_2$

Local: $m_2 \rightarrow m_3$

In the correct process FIFO and causal is respected. In fact we have $m_1 \rightarrow m_2 \rightarrow m_3$ in p_2 and p_4 that are correct.

3) TO (UA, WUTO), mot TO(UA, SUTO)



1 0 3 4 5

1 2 3 4 5

1 2 0 4 5

1 2 3 4 5

4) TO(NUA, WNUTO), mot TO(UA, WNUTO)