EX.2

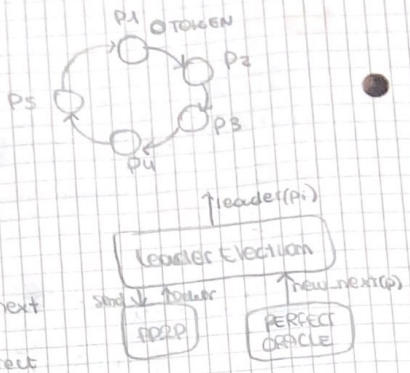
INIT
ID ← identifier
next = $p_{i+1 (mod n)}$
leader = ⊥
correct = ∅
IF ID = ⊥ !
pp2pSend ([ALIVE, self, correct) to next

UPON EVENT pp2p Deliver([ALIVE, s]) DO
IF s ∉ correct
correct = correct ∪ {s}
TRIGGER pp2pSend ([ALIVE], self, correct) to next

UPON EVENT new_next(p) DO ← IF ID(p-1) ∈ correct
correct = correct \ {ID(p-1)}
TRIGGER pp2p Send ([CRASH], p-1) to next

UPON EVENT pp2p Deliver ([CRASH], q)
IF q ∈ correct
correct = correct \ {q}
TRIGGER <pp2pSend([CRASH], q) to next

UPON leader ≠ max(rank(correct))
leader = max rank (correct)
TRIGGER Elect(leader)



EX.2

IDEA:
when the write has
round in the ring, it

INIT
value = null
writing = FALSE
next = ID(pi+1)

UPON EVENT <
IF (writing = FA
writing =
TRIGGER

UPON EVENT <
value = v
IF (! writ
TRIGGER
ELSE
TRIGGE

UPON EVENT
TRIGGER

WEEK 9

EX.1

1. Regular Register

$r_2() \to 0,1,2$
$r_1() \to 1,2$
$r_4() \to 1,2$
$r_3() \to 1,2,3$
$r_5() \to 2,3$

2. Atomic Register

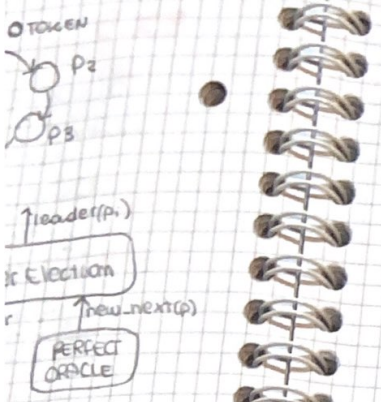
$r_2() \to 0,1,2$
$r_1() \to 1,2$
$r_4() \to 1,2$
$r_3() \to$ IF $r_2() \to 2$ THEN $r_3() \to 2,3$
OTHERWISE $r_3() \to 1,2,3$
$r_5() \to 2,3$

3.

$S = w(1), r_2(2), r_4() \to 1, w(2), r_1() \to 2, w(3), r_3() \to 3, r_5() \to 3$

(not legal)

NOT LINEARIZABLE



EX.2

IDEA:

when the write has completed round in the ring, it can return

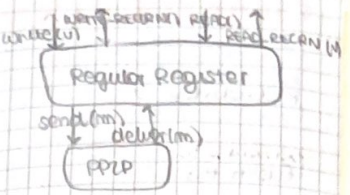
INIT

value = NULL
WRITEing = FALSE
next = ID(p+1)

UPON EVENT < rr, Write | v > DO value = v
  IF (!WRITEing) = FALSE
    WRITEing = TRUE
    TRIGGER <app2p | Send | [WRITE], value, > to next

UPON EVENT < app2p | Deliver | [WRITE], v, >
  value = v
  IF (!WRITEing)
    TRIGGER <app2p | Send | [WRITE], value > to next
  ELSE
    TRIGGER <nr, WriteReturn>

UPON EVENT < rr | Read >
  TRIGGER <ReadReturn | value >



EXERCISE 1

PER FARE rb-deliver UN QUALSIASI PROCESSO DEVE RICEVERE UNA COPIA DEL MESSAGGIO, QUINDI IL TEMPO ATTESA DI DELIVERY SI PUO' STIMARE COME IL TEMPO CHE IMPIEGA UN QUALSIASI PROCESSO A FAR DELIVERY DI UN MESSAGGIO.

Quanti messaggi riceve ogni processo → 1 msg ogni 20s, quindi $\frac{1}{20}$ = 0.05 msg/s. I processi sono 10 quindi il rate di esecuzione è $10 \cdot 0.05$. Nel protocollo eager, ogni messaggio viene ripropagato da ogni processo. Quindi l'arrival rate a cui ogni processo è soggetto è $\lambda = 0.05 \cdot 10 \cdot 10 = 5$ req/sec.

Ogni processo è in grado di gestire $\mu = 10$ mes/s.

Quindi

$$R = \frac{1}{\mu - \lambda} = \frac{1}{10 - 5} = 0,2\ s$$

(stability condition ou $\lambda \leq \mu$)

EXERCISE 2

2) Entrambi i protocolli richiedono che un messaggio faccia il giro dell'intero ring. Nel caso di Unring chi esegue la get genera un solo messaggio, nel caso di Biring sono 2.

$\lambda_{uni} = 1 \cdot 10 = 10$ req/sec

$\lambda_{bi} = 2 \cdot 10 = 20$ req/sec

$\mu = 10$ msg/s

Siccome un messaggio deve attraversare 10 nodi, risultato finale è $R_i \cdot 10$

1) $A_{ser} = \prod A_i = \left(\frac{MTBF}{MTBF + MTTR}\right)^{10} = 0,1$

EXERCISE 3

Entro il tempo Δ ogni processo deve essere in grado di contattare tutti i processi e ricevere risposta. Siccome tutti i processi stanno eseguendo if failure detector, ogni processo deve essere in grado in $\Delta/2$ di ricevere tanti messaggi quanti sono i processi. Assumiamo una distr. esponenziale.

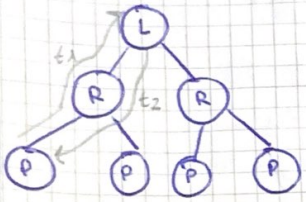
$$\frac{\Delta}{2} = 1 - (10 - 30 \cdot \frac{\Delta}{2})$$

WEEK 10

EX.1

1. T The performance of a system is mainly measured by the time taken to perform the service the rate at which the service is performed and the resource consumed.
2. F The response time depend end of arrival rate and service rate $R = \frac{1}{\lambda - \mu}$
3. T
4. F In M/M/k we consider the probability of an arrival
5. F Since we assume that the faults occurences and repairs are independent among them other.

EX.2



$\lambda_p = 10$ req/min
$R_L = 0.5$ s
60 msg/min

2) $A_{TOT} = A_R \times A_S \times A_P = 0.9 \cdot 0.9 \cdot 1 =$ 0,81

1) To evaluate the expected time that elapses between a request generated by a process P and the receipt of the response, we consider:

- Generation time $\frac{1}{10} = 0,1$ s
- Transmission time (req → R → L → R → P) since each process can send/receive 60 msg/min, the average time to transmit a message is 1/60 = 0,016 min
- Computation Time: 0.5/60 = 0,008 min
- Relay Time: 0,016 min

$T = \frac{1}{10} + 3 \cdot \frac{1}{60} + \frac{0.5}{60} + 2 \cdot \frac{1}{60} = 0,18$ min

EX.3

$\lambda = 10$ req/s
$k = 4$
$\mu = 5$ req/s

$R = \frac{1}{k\mu + \lambda} = \frac{1}{20 - 10} = 0,1$ s expected response time

WEEK 11

EX. 1

Validity = GUARANTEED. Since the byzantine can not compromise the reliable delivery

No duplication = NOT GUARANTEED. The byzantine can send a message already delivered, and the other correct never check if the message was already delivered, so they can deliver it dgaain.

No creation = NOT GUARANTEED. The byzantine can create whatever message

EX. 1

Validity. Not Satisfied. BYZANTINE p BROADCAST [DATA, q, m', 1] IN SUCH A WAY WHEN q RECEIVES IT, q DELIVERS m' AND next[q] = 2. NOW q broadcast m and q will recieve [DATA, q, m, 1] and q will never deliver m because next[q] != 1

No Duplication. Not Satisfied. Byzantine can deliver more than once the same message.

No creation. Not Satisfied. See validity, q delivers m' with sender q but m' was not broadcast by q.

Agreement. Not Satisfied. Byzantine p broadcasts through REB [DATA, p, m1, sn] and [DATA, p, m2, sn] (same sn). Suppose two corrects q_1, q_2 have next[p] = sn. It can happen that q_1 recieves [DATA, p, m1, sn] and delivers m1 and next[p] = sn + 1 so q_1 never deliver m2; q_2 recieve [DATA, p, m2, sn] and delivers m2 and so q_2 will never be able to deliver m1.

FIFO Delivery. Not Satisfied. Byzantine broadcasts m1 and then broadcasts m2 both with inverted sn. In such way processes will deliver them in inverted order.

EX. 2

Validity. Satisfied. Guaranteed by the validity of BEB, since the byzantine can not send different values to different processes.

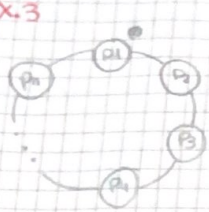
No duplication. Not Satisfied. Byzantine can deliver a message more than once.

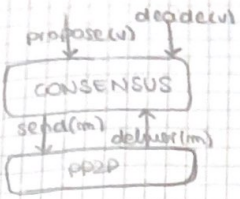
No creation. Not Satisfied. Byzantine can broadcast [DATA, q, m]. When a process deliver it with sender q, but q never broadcast m.

Agreement. Satisfied. If the sender is byzantine then we know that for the symmetric behaviour of the byzantine all corrects will recieve the same message.

EX. 3

1.





INIT

next = $p_{(i+1) \bmod n}$

n ← n. of processes
decision = ⊥
proposalSet = ∅

IF next == p1
TRIGGER <pp2p, Send | [TOKEN]>

UPON EVENT <pp2p, Deliver | [TOKEN]>
TRIGGER <Con, Propose | v>

UPON EVENT <Con, Propose | v> DO
proposalSet = proposalSet ∪ {v}
TRIGGER <pp2p, Send | [PROPOSE, proposalSet]> TO next
TRIGGER <pp2p, Send | [TOKEN]> TO next

UPON EVENT <pp2p, Deliver | [PROPOSE, pSet]> DO
proposalSet = pSet

IF #(proposalSet) == n
decision = min(proposalSet)
TRIGGER <Con, Decide | decision>

next

IDEA:
Who have the token can propose. When all the proposal has been traversed during the processes can decide

2. If the processes may crash, the problem is that when the token is passed to next it remains blocked since the next process never pass the token.
In order to solve this issue we need an oracle that provide a new-next(p) to each process in order to be sure that the token is passed to the next correct.

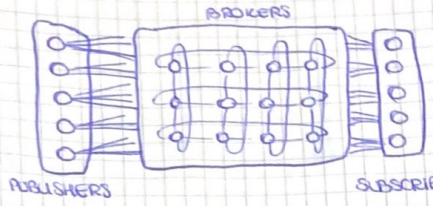
3. Terminatum. Not Satisfied. Byzantine can never send the token and the ring is broken

Validity. Not satified. Byzantine can change the proposalSet, in such a way processes may decide fake value.

Integrity. Not Satisfied. Byzantine may decide twice

Agreement. Not Satisfied. Byzantine can change the message in such a way previous correct decide a value and next other value

EX.4



k-connected

1. Subscription-flooding dissemination
   ↳ each subscription is copied on every brokers, in order to build complete subscription tables.

Code for each broker

INIT

```
publishers = getPublishers()
subscribers = getSubscribers()
brokers = getBrokers()
subscriptions [] = {}
messages = {}

UPON EVENT pp2pDeliver ([SUB,s,p]) from q
   IF (s not in subscriptions[p])
      subscription[p] = subscription[p] ∪ {s}
   FOR b in Brokers \ {q}
      TRIGGER pp2pSend([SUB, s, p]) to b

UPON EVENT pp2pDeliver ([PUB, m, p]) from q
   IF (m not in messages)
      messages = messages ∪ {m}
      FOR s in subscription
         IF (m match subscription[s])
            TRIGGER pp2pSend([PUB, m, p]) to s
      FOR b in Brokers \ {q}
         TRIGGER pp2pSend([PUB, m, p]) to b
```

Code for publisher

INIT

```
Brokers = getBrokers()

UPON EVENT pp2pSend(m)
   FOR b in Brokers
      TRIGGER pp2pSend([PUB, m, self)] to b
```

Code for each subscriber

INIT

```
messages = {}
Brokers = getBrokers()

UPON EVENT Subscribe(s)
   FOR b in Brokers
      TRIGGER pp2pSend([SUB, s, self]) to b

UPON EVENT pp2pDeliver ([PUB, m, p]) from q
   IF (m not in messages)
      messages = messages ∪ {m}
      TRIGGER Deliver(m) FROM p.
```

2. The network
   the algorith

3. The idea
   the mode
   subscrip
   neighbou
   dis, aint
   The algo

EX.5

Specificatio

= L
Safety

FOR GLOB
Communi
MUST exi
For k
HKUO w

2. The network is k-connected, so the number of crashes that the algorithm can tolerate is $k-1$

3. The idea can be that of attaching to each message/subscription the nodes that have relayed it. A neighbor of the source of the subscription/message can consider it as trusted; while a non neighbour waits for $f+1$ copies of the subscription/message with disjoint paths.
The algorithm works if and only if $k \geq 2f+1$.

EX. 5

Specification of Reliable Communication are:

Safety = IF A CORRECT PROCESS p DELIVER A CONTENT c FROM q, THEN c WAS PREVIOUSLY SENT BY q. (no dup. and no creation)

Liveness = IF p IS A CORRECT PROCESS AND SENS A CONTENT c TO A CORRECT PROCESS q, THEN q EVENTUALLY DELIVERS c.

FOR Globally Bounded Model the correctness condition of reliable communication is that at least $f+1$ node-disjoint path must exists. In case of authenticated link we need node-connectivity $\geq 2f$.
For locally bounded instead the sufficient condition is that $\forall i \in V$ with $k \geq 2f+1$, necessary condition $k=f+1$.