Dependable Distributed Systems
Exercise week 2
October 6th, 2022

Exercise 1

With reference to the synchronization of physical clocks, provide the definition of internal and external clock synchronization. In addition, consider a system composed of two processes p_1 and p_2 and one UTC server p_S. Let us assume that:

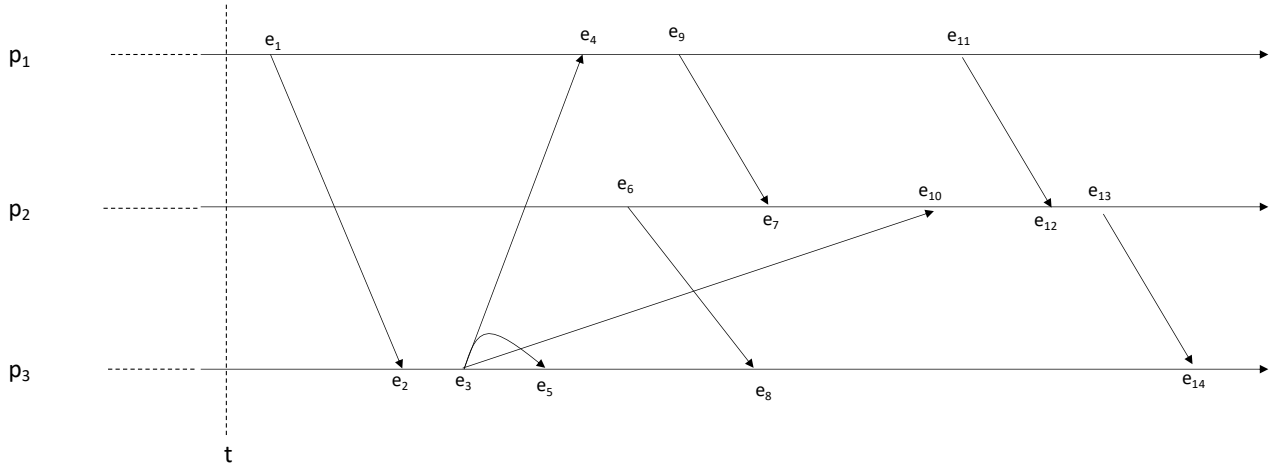
- p_1 and p_2 communicate with p_S by using perfect point-to-point links
- the maximum latency of the channel between p_S and p_1 is 1 ms
- the maximum latency of the channel between ps and p_2 is 2 ms.

In addition, let us assume that p_1 and p_2 start a clocks synchronization procedure at a certain time t by running the Christian algorithm. Answer to the following questions:

1. how much is the accuracy bound D_{ext} of the external synchronization obtained by p_1 and p_2 at the end of the synchronization?
2. Is the current system internally synchronized? If yes, determine the internal synchronization bound D_{int} obtained at the end of the procedure.

Exercise 2

Let us consider the execution history depicted in the figure

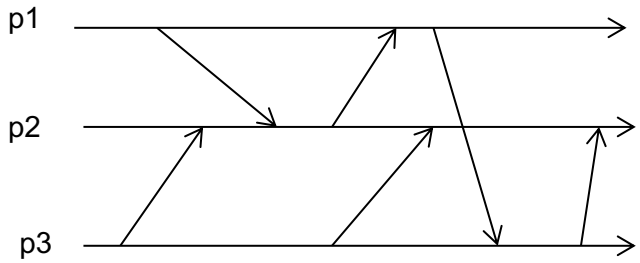


Given the run depicted in the figure state the truthfulness of the following sentences:			
a	According to the happened-before relation, $e_5 \rightarrow e_7$	T	F
b	According to the happened-before relation, $e_4 \parallel e_5$	T	F
c	Let CK_i be the variable storing the scalar logical clock of process p_i. Let us assume that at time t, $CK_i = 0$ for each process p_i. The logical clock CK_2 associated to e_6 is strictly larger than the logical clock CK_1 associated to e_1	T	F
d	Let CK_i be the variable storing the scalar logical clock of process p_i. If at time t $CK_1 = 0$ then the logical clock associated to e_8 is $CK_2=3$	T	F
e	Let CK_i be the variable storing the vector logical clock of process p_i. If at time t $CK_1 = [0, 0, 0]$ then the logical clock associated to e_8 is $CK_2=[3, 0, 3]$	T	F

For each point, provide a justification for your answer

Exercise 3

Describe timestamping techniques based on scalar logical clocks and vector logical clocks. In addition, considering the execution reported in Figure, answer to the following questions:



1. Apply the scalar clock timestamping technique to the execution assigning a timestamp to each event
2. Apply the vector clock timestamping technique to the execution assigning a timestamp to each event
3. List all pairs of concurrent events in the proposed execution

Exercise 4

Consider an asynchronous message passing system that uses vectors clock to implement some causal consistency check. The message passing system is composed by 4 processes with IDs 1 to 4, and, as usual, the ID is used as displacement in the vector clock (i.e., the locations are (p1, p2, p3, p4)).
Vector clocks are updated increasing before send. Processes communicate by point2point links. You start debugging process p1 (the process with id 1) in the middle of the algorithm execution. You see the following stream of messages exiting and entering the ethernet card of process p2:·

- Time 00:00 – EXITING: Send Message [MSG CONTENT] Vector Clock: (1, 0, 0,0)
- Time 00:05 – ENTERING: Rcvd Message [MSG CONTENT] Vector Clock: (1, 2, 3,1)

1. Draw an execution that justifies the vectors clocks you are seeing. Is such an execution unique?
2. There exists an execution that justifies the vector clocks and where there exists at least a process that does not send any message? Justify your answer.

Exercise 5

Let us consider a distributed system composed of N processes p_1, p_2, … pn each one having a unique integer identifier. Processes are arranged in line topology as in the following figure

Let us assume that there are no failures in the system (i.e., processes are always correct) and that topology links are implemented trough perfect point-to-point links.

Write the pseudo-code of a distributed algorithm that is able to build the abstraction of a perfect point-to-point link between any pair of processes (also between those that are not directly connected).

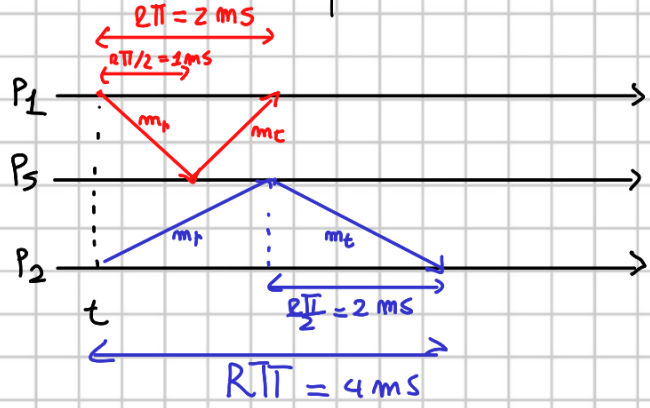
Exercise 1

With reference to the synchronization of physical clocks, we talk about internal synchronization when all the processes synchronize their clocks C_i between them and clocks are internally synchronized in a time interval Δ if $|C_i(t) - C_j(t)| < D$ for $i,j = 1 \cdots N$ and for all time t in Δ, where $D > 0$ is the synchronization bound and $C_i - C_j$ the clocks at processes p_i and p_j. We have external synchronization when processes synchronize their clock C_i with an authoritative external source S (UTC) and clocks C_i are externally synchronized with a time source S if for each time interval Δ : $|S(t) - C_i(t)| < D$.

info:

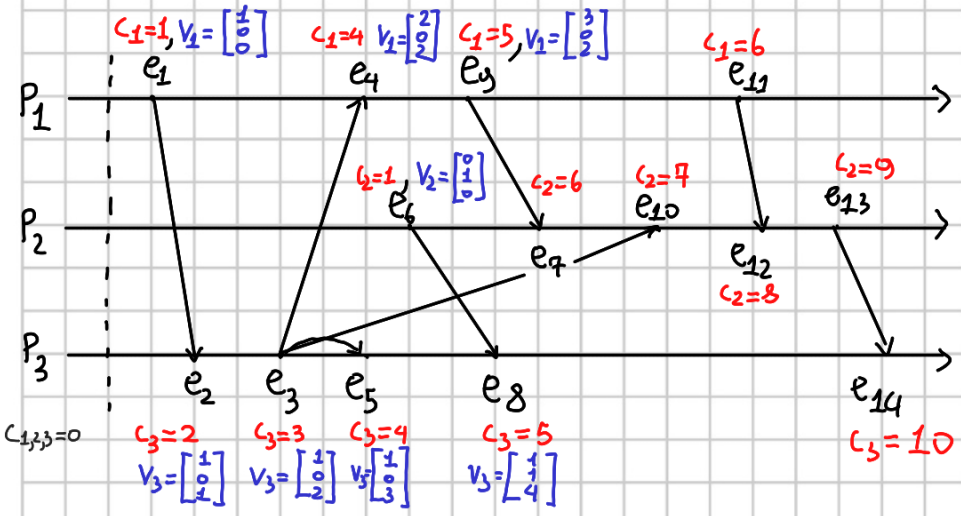
- there are P_1, P_2, P_S
- $P_1 \longleftrightarrow P_S$ 1 ms
- $P_2 \longleftrightarrow P_S$ 2 ms

Simulation of Christian algorithm:



1. Accuracy bound D_{ext} of the external syncronization obtained by P_1 is $D_{ext_1} = \pm \frac{RTT_1}{2} = \pm 1$ ms, while $D_{ext_2} = \pm \frac{RTT_2}{2} = \pm 2$ ms
2. Given that the system is externally syncronized is also internally synchronized in a bound $2 \cdot D_{ext}$ that is $D_{int} = \pm 4$ ms. We choose ± 4 ms and not ± 2 ms because we choose the greatest D_{ext} for consider the worst case scenario, that ± 2 ms do not cover. (Ex.: P_2 forward of D_2 (+2 ms))

Exercise 2



a. False because the events $e_5 \parallel e_7$, not soddisfy ony of the 3 condition of the happened-before.

b. True because V_1 associated to e_4 is $\begin{bmatrix}2\\0\\2\end{bmatrix}$ and V_3 associated to e_5 is $\begin{bmatrix}1\\0\\3\end{bmatrix}$ ($2>1$, $0=0$ but $2<3$)

C. False because the timestamp of $e_1=1$ and timestamp of $e_6=1$, they are equal.

d. False because $c_3=5$ associated to e_8.

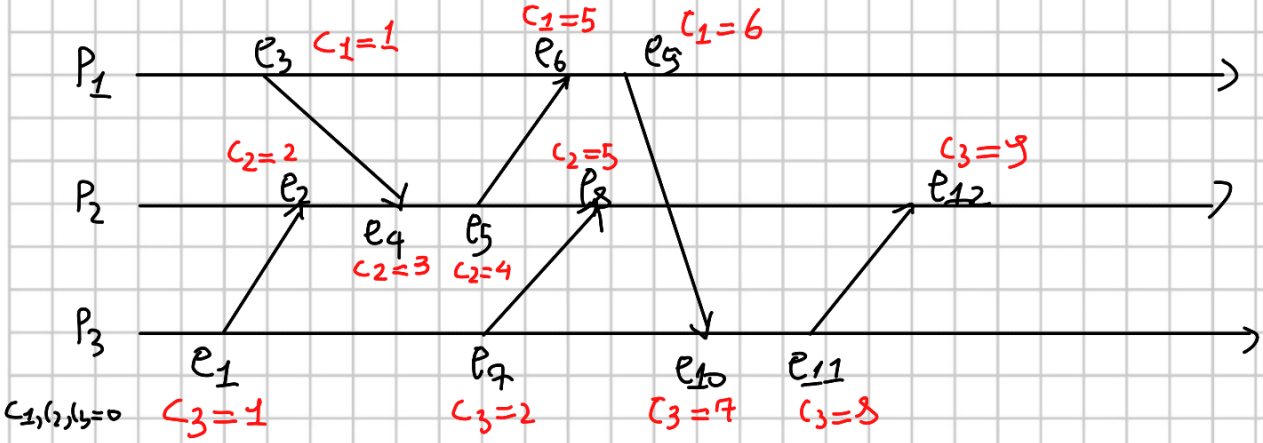
e. False because $V_3=\begin{bmatrix}1\\1\\4\end{bmatrix}$ associated to e_8.

Exercise 3

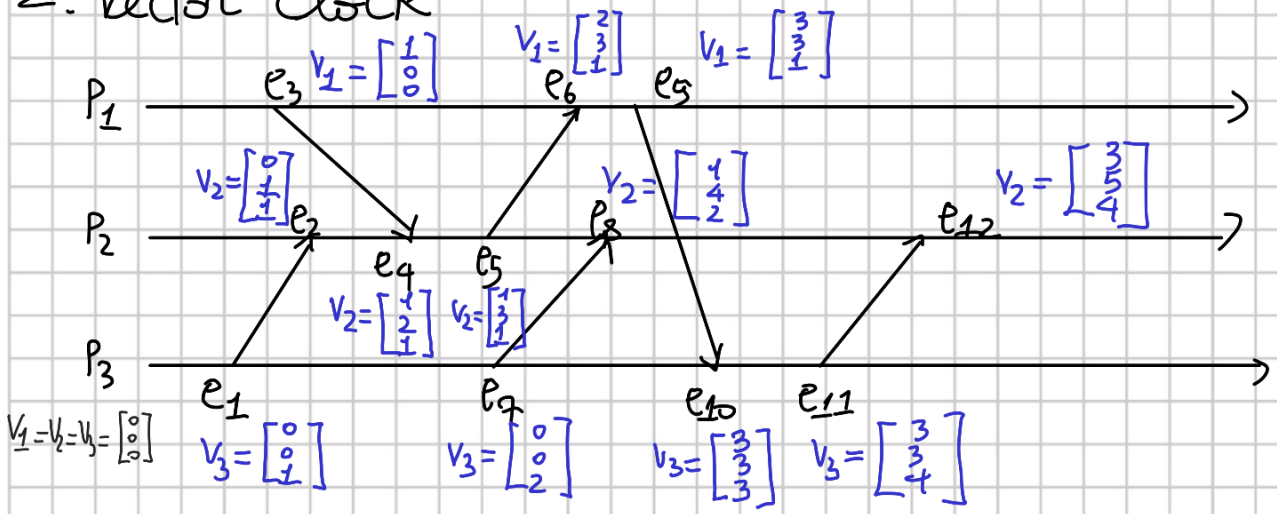
In Scalar logical clocks technique each process P_i initializes its logical clock $c_i=0$, P_i increase c_i of 1 when it generates an event. When P_i send a message m increase c_i of 1 and timestamp m with $t_s=c_i$, while when P_i recive a message m update $c_i=\max(t_s,i)$

In Vector clocks technique for a set of n processes the vector clock in composed by an array of n integers. Each process P_i mantain a vector clock V_i and timestamps events by mean of its vector clock.

1. Scalar clock



2. Vector clock



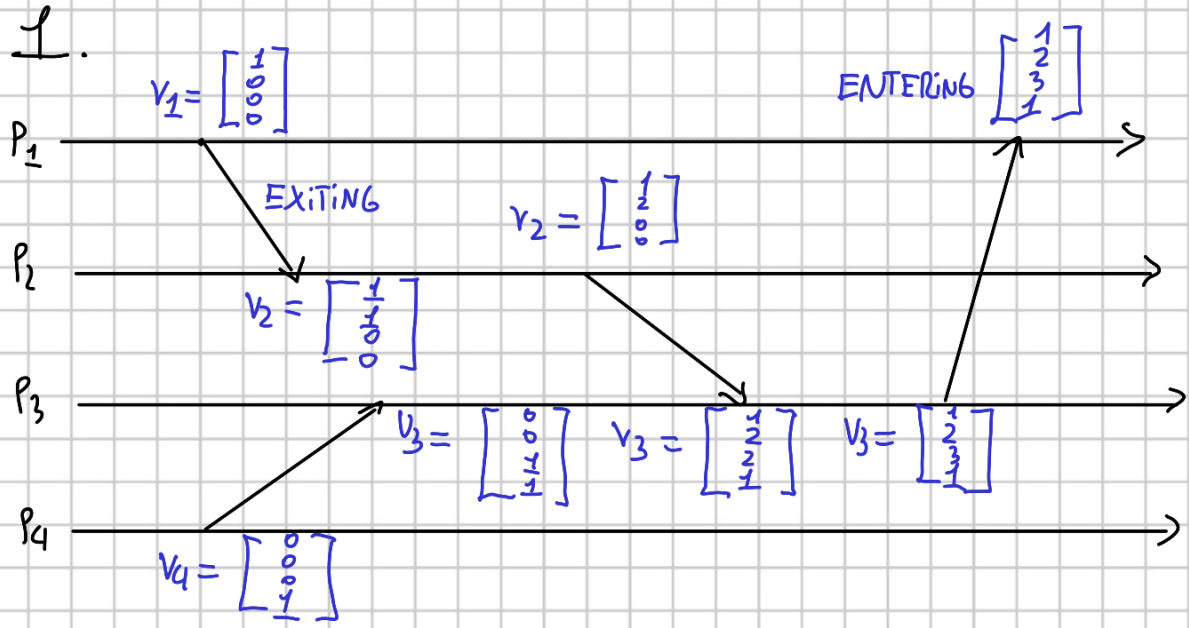
3. We can see the concurrent event using the vector clocks:

$e_1 \parallel e_3$, $e_2 \parallel e_3$, $e_2 \parallel e_7$, $e_5 \parallel e_7$, $e_4 \parallel e_7$, $e_3 \parallel e_4$, $e_9 \parallel e_7$, $e_6 \parallel e_7$, $e_8 \parallel e_5$, $e_8 \parallel e_{10}$, $e_8 \parallel e_{11}$

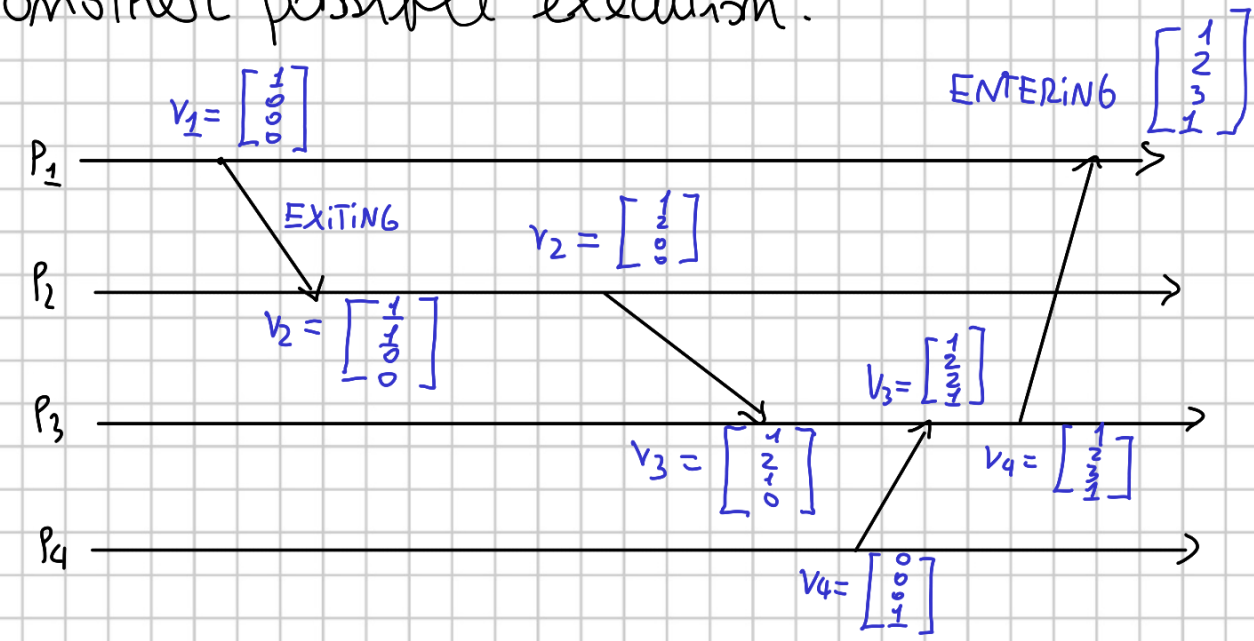
Exercise 4

4 processes: P_1, P_2, P_3, P_4 each with a vector clock V_i

1.

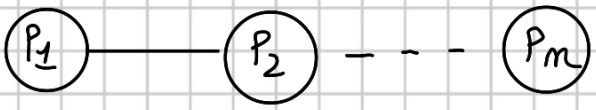


this execution is not unique because we show another possible execution:

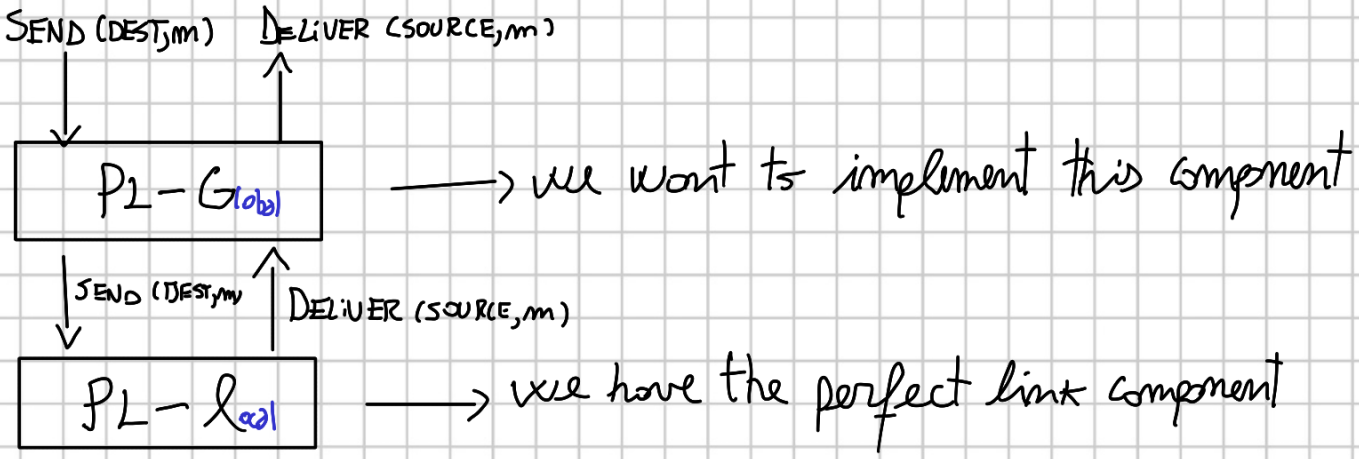


2. its not possible that a process send any message, because in V_1 at time 00:05 enter a vector with all data of array at least 1 which implies that every process send at least a message.

Exercise 5



there are n processes, each with an identifier $i = 1, \dots, N$. each message is for another process $j = 1, \dots, N$. Each process can send the message only to the near process, that are, for a process P_i, P_{i+1} and P_{i-1}. We are using perfect link that guarantee that if the endpoints are correct the message will recived. We want implementing a perfect line P2P given that each process sends a message from the opposite side it delivered it.
We have two event handler, one for send and one for deliver, we indicate with p the sending process and with q the reciver process.
these are the components and the events:



→ we want to implement this component

→ we have the perfect link component

algotithm:

implements: PerfectP2P PL-G

uses: P2P-link PL-l

every process have an ID unique.

upon event $\langle$ PL-G, SEND | q, m $\rangle$ do :
(event; component; parameters)

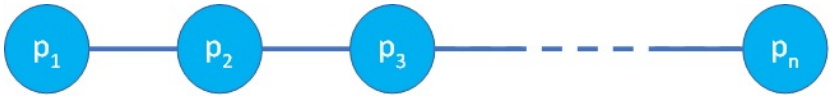
```
if q > ID then
  trigger < PL-L, SEND | ID+1, <ID, q, m> >
else if q < ID then
  trigger < PL-L, SEND | ID-1, <ID, q, m> >
else
  trigger < PL-L, SEND | ID, <ID, q, m>>
```

Upon event $\langle$ PL-L, DELIVER | s, $\langle$ p, q, m $\rangle\rangle$ do:

```
  if q == ID then
    trigger < PL-G, DELIVER | p, m >
  else if q > ID then
    trigger < PL-L, SEND | ID+1, <p, q, m>>
  else
    trigger < PL-L, SEND | ID-1, <p, q, m>>
```

Exercise week 2 - October 6th, 2022 - Exercise 5. Commented Solution

Let us consider a distributed system composed of N processes $p_1, p_2, \ldots p_n$ each one having a unique integer identifier. Processes are arranged in line topology as in the following figure



Let us assume that there are no failures in the system (i.e., processes are always correct) and that topology links are implemented through perfect point-to-point links.

Write the pseudo-code of a distributed algorithm that is able to build the abstraction of a perfect point-to-point link between any pair of processes (also between those that are not directly connected).

General Comments

The text says that processes have access to a perfect point-to-point link primitive, let us refer to this component with $\mathcal{PL}_l$. This primitive allows every process to exchange messages with at most two other processes, the neighbor processes in the line.

Let us recall the events and properties of a perfect point-to-point primitive.

Module:

Name: PerfectPointToPointLinks, **instance** *pl*.

Events:

Request: $\langle$ *pl*, *Send* | q, m $\rangle$: Requests to send message m to process q.

Indication: $\langle$ *pl*, *Deliver* | p, m $\rangle$: Delivers message m sent by process p.

Properties:

PL1: *Reliable delivery:* If a correct process p sends a message m to a correct process q, then q eventually delivers m.

PL2: *No duplication:* No message is delivered by a process more than once.

PL3: *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by process p.

Every process can thus trigger *Send* events and catch *Deliver* events of the $\mathcal{PL}_l$ component.

The exercise requests to define a protocol a link abstraction, let us name it $\mathcal{PL}_g$, that implements a perfect point-to-point link between all pairs of processes. In the initial setting, every process can exchange messages only with at most two processes, namely the neighbor processes on the line. It follows that if two processes are not linked by $\mathcal{PL}_l$, such as processes p_1 and p_4, they cannot exchange messages in the current setting. The target of the exercise is the definition of a protocol that implements a communication primitive guaranteeing all the properties of a perfect point-to-point link between all pairs of processes.

Protocol Idea

Each process is associated with an integer identifier and it is placed in a topology (the line) such that all the processes with a lower identifier are placed on one side (from the prospective of a selected process) and all processes with a higher identifier are located on the other side. Furthermore, the processes are placed in order on the line with respect to their identifiers. It follows that if process p_1 wants to communicate with process p_4, it can send its messages to process p_2, that will relay such messages to p_3, which finally forwards them to p_4. More in detail, if the destination of a message is a process with a higher identifier with respect to a selected process, then it is sufficient to forward the message to the neighbors with greater ID till reaching the destination process, and vice-versa for a destination with a lower identifier.

Pseudo-code

Algorithm 1 $\mathcal{PL}_g$

```
1: procedure INIT
2: |  ID ← unique integer identifier

3: upon event ⟨PL_g, Send | dest, m⟩ do
4: |  if dest > ID then
5: |  |  trigger ⟨PL_l, Send | ID + 1, ⟨ID, dest, m⟩⟩
6: |  else if dest < ID then
7: |  |  trigger ⟨PL_l, Send | ID − 1, ⟨ID, dest, m⟩⟩⟩
8: |  else
9: |  |  trigger ⟨PL_l, Send | ID, ⟨ID, dest, m⟩⟩⟩

10: upon event ⟨PL_l, Deliver | q, ⟨source, dest, m⟩⟩ do
11: |  if dest == ID then
12: |  |  trigger ⟨PL_g, Deliver | source, m⟩
13: |  else if dest > ID then
14: |  |  trigger ⟨PL_l, Send | ID + 1, ⟨source, dest, m⟩⟩
15: |  else
16: |  |  trigger ⟨PL_l, Send | ID − 1, ⟨source, dest, m⟩⟩
```

Pseudo-code Comments

The first upon block (lines 3-9) rules how a process must act when the *Send* operation of the global perfect point-to-point primitive we are developing, $\mathcal{PL}_g$, is triggered. It is a tiny procedure that simply starts the propagation of a new message, $\langle source, dest, m\rangle$, over the proper link, $\mathcal{PL}_l$. Again, a content m (let use this word to distinguish m from $\langle source, dest, m\rangle$) targeted to a process with an higher identifier (*dest*) is reachable (on the line) by relaying the content to the neighbor with higher ID and vice-versa.

The second upon block rules how a process that has received a message $\langle source, dest, m\rangle$ from the primitive $\mathcal{PL}_l$ (namely the local perfect point-to-point link) must act. This procedure compares the *dest* field contained inside the received message with the process identifier. If the process is the destination of a content, then it triggers the *Deliver* operation of the primitive we are developing (content m was targeted to this process). Otherwise, it continues the propagation of the message $\langle source, dest, m\rangle$ over the proper perfect point-to-point link. Notice that we need to forward the information about the source ID *source* and the destination id *dest* insider the message, otherwise a process (not linked with the source process) receiving a content m from $\mathcal{PL}_l$ cannot assert whether the content is targeted to it-self and which process was its source.

Informal Correctness Proofs

We briefly check the correctness of the provided pseudo-code. The $\mathcal{PL}_g$ primitive we defined must handle two events, $\langle \mathcal{PL}_g, Send \mid dest, m \rangle$ and $\langle \mathcal{PL}_g, Deliver \mid source, m \rangle$. Furthermore, it must guarantee the *reliable delivery*, *no duplication* and *No creation* property of a perfect point-to-point link abstraction.

Reliable delivery: if a $\langle \mathcal{PL}_g, Send \mid dest, m \rangle$ event is generated (from the application layer), then the first upon procedure reacts to this event by starting the propagation of the message $\langle source, dest, m \rangle$ over the proper perfect point-to-point link, triggering a *Send* event on the $\mathcal{PL}_l$ primitive. The $\mathcal{PL}_l$ abstraction guarantees all the perfect point-to-point link properties between two linked processes. Therefore, every *Send* event of the $\mathcal{PL}_l$ component generates a *Deliver* event on the destination process (e.g. a $\langle \mathcal{PL}_l, Send \mid p_2, \langle source, dest, m \rangle \rangle$ event on process p_1 generates the $\langle \mathcal{PL}_l, Deliver \mid p_1, \langle source, dest, m \rangle \rangle$ event on process p_2). The second upon procedure guarantees the propagation of the message $\langle source, dest, m \rangle$ till process $p_i, i = dest$. Only process $p_i, i = dest$ triggers the $\langle \mathcal{PL}_g, Deliver \mid source, m \rangle$ event. The propagation policy eventually guarantees the occurrence of this event.

No duplication: $\mathcal{PL}_l$ guarantees no duplication of the messages diffused by the primitive. For a single $\langle \mathcal{PL}_g, Send \mid dest, m \rangle$ event a single message $\langle source, dest, m \rangle$ is generated and propagated over the line using $\mathcal{PL}_l$, furthermore every process relays at most once a single $\langle source, dest, m \rangle$ message.

No creation: $\langle \mathcal{PL}_g, Deliver \mid source, m \rangle$ event can be triggered only from the reception of a message $\langle source, dest, m \rangle$. The message $\langle source, dest, m \rangle$ is initially generated by the processes only managing a $\langle \mathcal{PL}_g, Send \mid dest, m \rangle$ event.

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

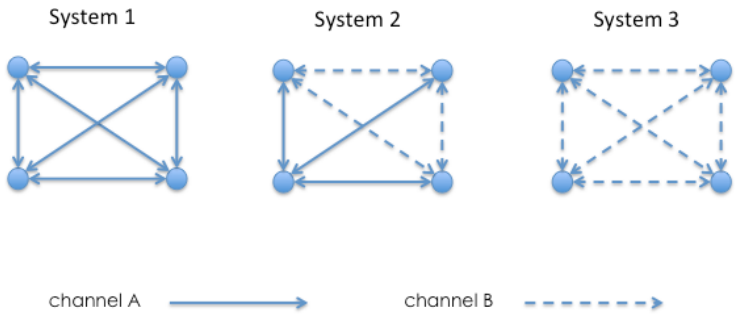
AA 2022/2023

Lecture 8 – Exercises

Ex 1: Let channel A and channel B be two different types of point-to-point channels satisfying the following properties:

- channel A: if a correct process p_i sends a message *m* to a correct process p_j at time *t*, then *m* is delivered by p_j by time $t+\delta$.
- channel B: if a correct process p_i sends a message *m* to a correct process p_j at time *t*, then *m* is delivered by p_j with probability p_{cons} (p_{cons} <1).

Let us consider the following systems composed by 4 processes p_1, p_2, p_3 and p_4 connected trough channels A and channels B.



Assuming that each process p_i is aware of the type of channel connecting it to any other process, answer to the following questions:

1. is it possible to design an algorithm implementing a perfect failure detector in system 2 if only processes having an outgoing channel of type B can fail by crash? true
2. is it possible to design an algorithm implementing a perfect failure detector in system 2 if any process can fail by crash? false
3. is it possible to design an algorithm implementing a perfect failure detector in system 3? false

For each point, if an algorithm exists write its pseudo-code, otherwise show the impossibility.

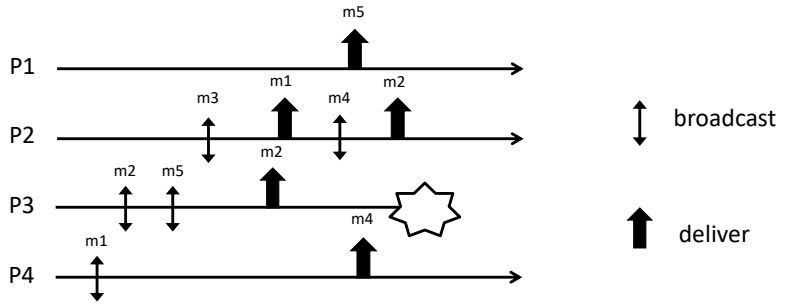
Ex 2: Consider a distributed system composed by n processes $\{p_1,p_2,\ldots,p_n\}$ that communicate by exchanging messages on top of a line topology, where p_1 and p_n are respectively the first and the last process of the network.
Initially, each process knows only its left neighbour and its right neighbour (if they exist) and stores the respective identifiers in two local variables LEFT and RIGHT.
Processes may fail by crashing, but they are equipped with a perfect oracle that notifies at each process the new neighbour (when one of the two fails) through the following primitives:

- **Left_neighbour(p_x)**: process p_x is the new left neighbour of p_i
- **Right_neighbour(p_x)**: process p_x is the new right neighbour of p_i

Both the events may return a NULL value in case p_i becomes the first or the last process of the line.
Each process can communicate only with its neighbours.

Write the pseudo-code of an algorithm implementing a Leader Election primitive assuming that channels connecting two neighbour processes are perfect.

Ex 3: Consider the partial execution depicted in the Figure



Answer to the following questions:

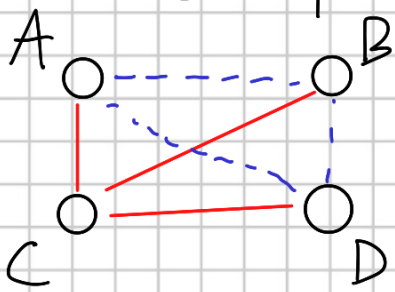
1. Complete the execution in order to have a run satisfying Uniform Reliable Broadcast.
2. Complete the execution in order to have a run satisfying Regular Reliable Broadcast but not Uniform Reliable Broadcast.
3. Complete the execution in order to have a run satisfying Best Effort Broadcast but not Regular Reliable Broadcast.

NOTE: In order to solve the exercise you can add broadcast, deliver and crash events but you cannot remove anything from the run.

Ex 4: Consider a distributed system composed by n processes $\{p_1,p_2,\ldots,p_n\}$ identified through unique integer identifiers. Processes may communicate using perfect point-to-point links. Links are available for any pair of processes.
Processes may fail by crash and each process has access to a perfect failure detector.
Modify the algorithms implementing a distributed mutual exclusion abstraction discussed during the lectures to allow them to tolerate crash failures.

Exercise 1

1. Yes is possible because all processes are linked with channel A that guarantee that the message sent by a process arrive to the destination.



Channel A

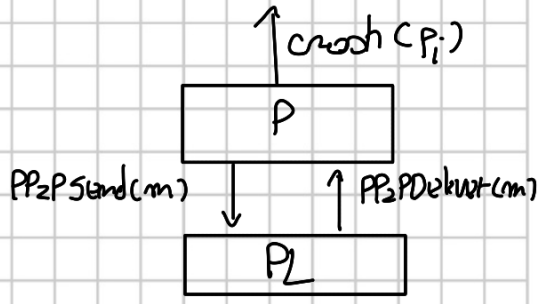
Channel B

each have unique ID

Foundamental is C that can not crash, because otherwise we can't guarantee the specification of a perfect failure detector, in particular the accuracy. All messages of A, B and D must pass to C, we can write this pseudo code:

implements: PerfectFailureDetector, P

Uses: PerfectP2PLinks, PL



upon event $\langle P, init \rangle$ do
  alive := Π; $\{P_1, P_2, P_3, P_4\}$
  detected := $\emptyset$;
  $timer_1 = \delta$
  starttimer($timer_1$)

→ we use two timer

upon event timeout $timer_1$ do
  forall $P_j \in \Pi$ do
    trigger $\langle PL, Send \mid LIST, alive_i \rangle$ to P_j
  starttimer($timer_2$) → $timer_2$ is equal to $timer_1$

Upon event $\langle PL, Deliver \mid LIST, l \rangle$ from P_j
  $alive_i = alive_i \cup l$ → all processes receive alive list of C because of channel A

when timeout timer$_2$ do
  For every $p_j \in \Pi$ do
    if $p_j \notin alive_i$ and $p_j \notin detected_i$:
      $detected_i = detected_i \cup \{p_j\}$
      trigger $\langle P, crash \mid p_j \rangle$
    trigger $\langle PL, send \mid HEARTBEAT \rangle$ to p_j;
  $alive_i = \emptyset$
  start timer$_2$ (timer$_2$)

search for crashed processes

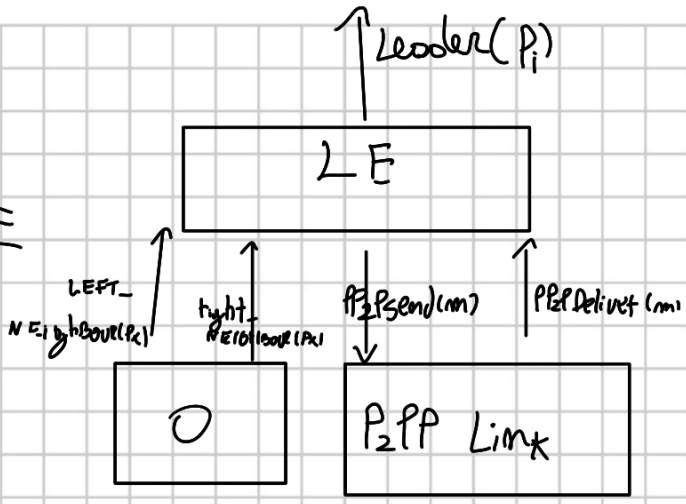
Upon event $\langle PL, Deliver \mid HeartBeat \rangle$ From p_j;
  $alive_i = alive_i \cup \{p_j\}$ → all update alive$_i$ list, in particular c will recive all correct HB request

In this way in the first timer$_1$ the processes A, B, C recive the alive processes known by C, that is always correct because have chA. in the second timer with heartBeat C add who is alive to the list and all work until C is the only that don't fail.

2. False, given that from exercise 1 we learn that without the processes C this algorithm don't work, if C crash the comunication between processes use channels A, that don't guarantee accuracy of PerfectFailureDetector, the system become asynchronous.

3. False the channels B are required for the PerfectFailureDetector, system 3 have only channel A, can't gustantee accuracy

Exercise 2



implements: LeaderElection, LE

Uses: Otacle O, PP2P link PL

Upon event < LE, Init > do
  suspected = ∅
  leader = P_1 ⟶ Process at beggining at start is the leader
  LEFT = get_left()
  RIGHT = get_right

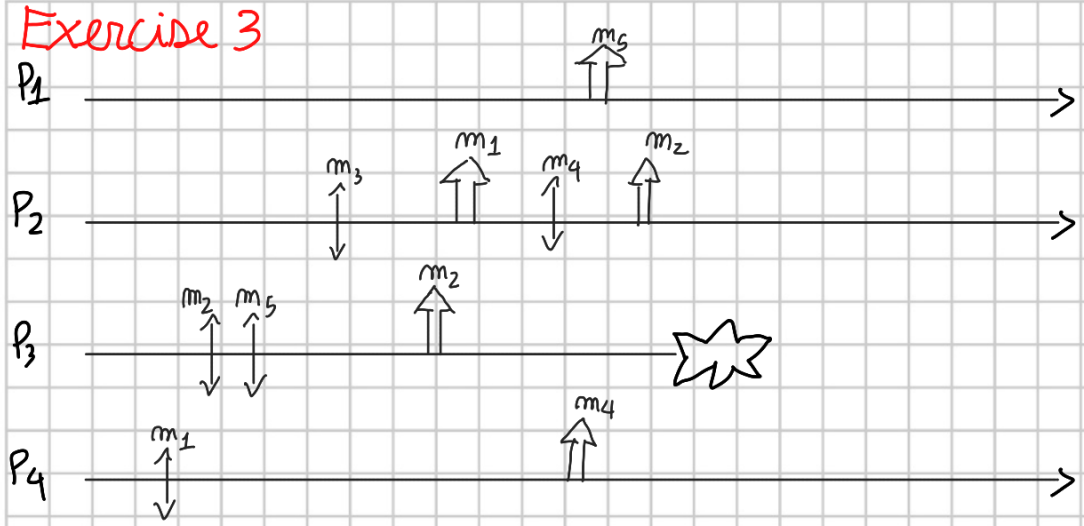
Upon event < O, LEFT_NEIGHBOUR | P_x > do
  left = P_x
  if left = null ⟶ must be that leader failed, i am the new leader
    leader = self
    trigger < PL, PP2Psend | NEW_LEADER, leader > to right

Upon event < O, RIGHT_NEIGHBOUR | P_x > do
  right = P_x ⟶ we simply update, the problem is on left

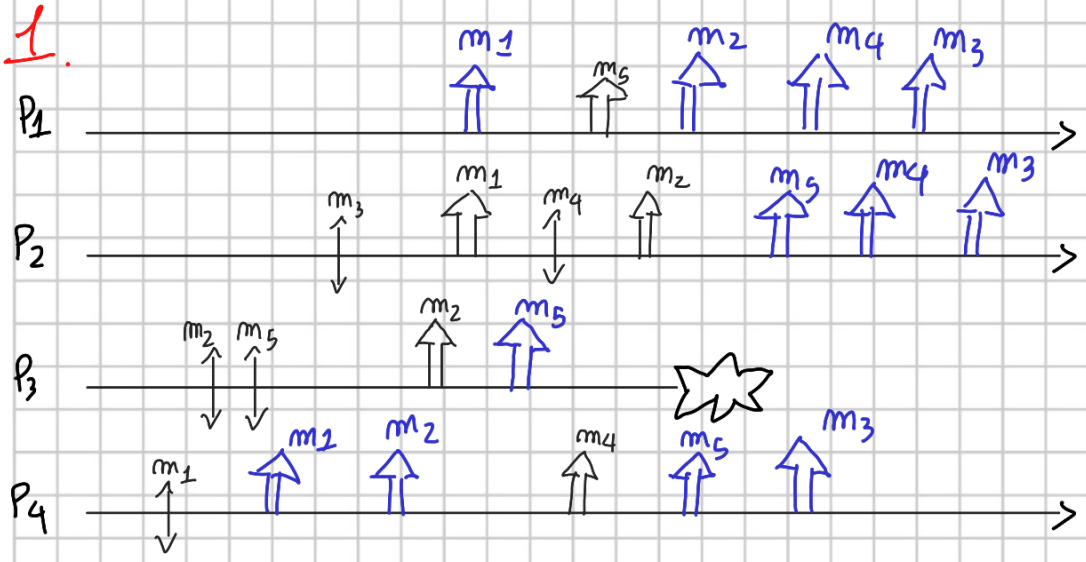
Upon event < PL, PP2PDeliver | NEW_LEADER, l > from left
  leader = l
  trigger < LE, leader | leader >
  trigger < P2, PP2Psend | NEW_LEADER, leader > to right

idea is to set initial leader the process at start, when recive the left neighbour from otacle we check that is null, that implies leader is crashed, and in the case add that process as new leader and propagate the news.

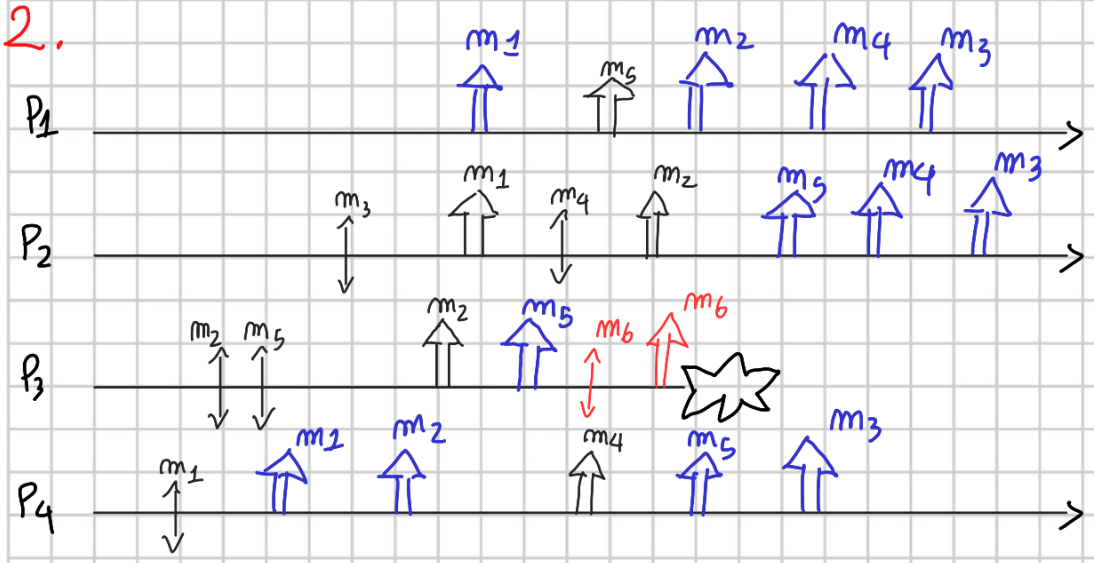
Exercise 3



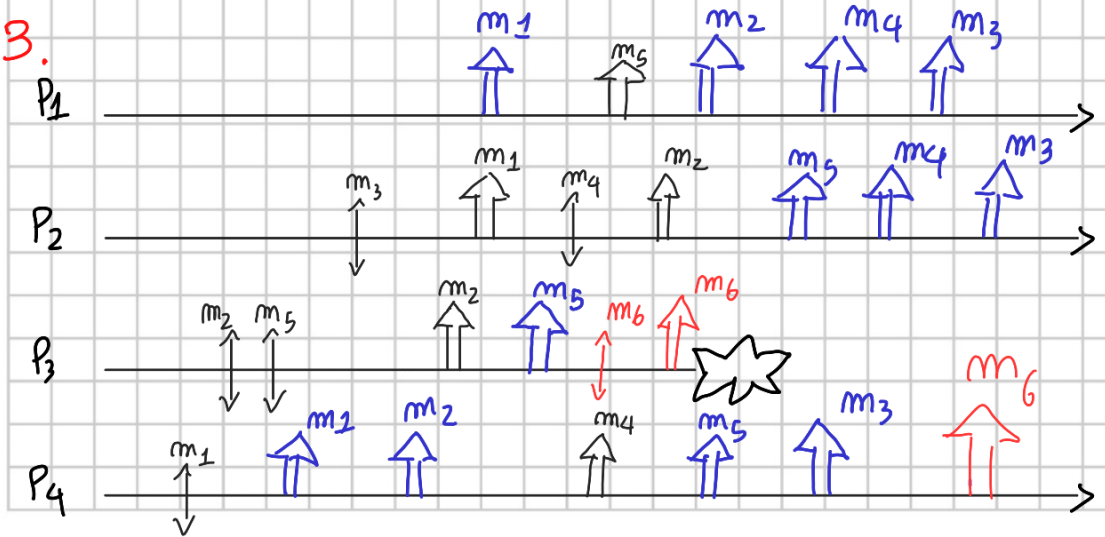
1.



2.



3.



Exercise 4

We modify the Ricart-Agrawala's Algorithm for mutual exclusion such that tolerate crash failures. Given that processes uses PP2P Links, all processes are linked each other, and have a Perfect Failure Detector

uses: PP2P Links, PL
Perfect Failure Detector, P

init:
replies = 0
state = NCS
Q = ∅
Last_Req = 0
Num = 0

begin
1. State = Requesting;
2. Num = num + 1; LastReq = num;
3. ∀ i = 1, ..., N send REQUEST(num) to p_i, $i \neq ID$
4. wait until replies = n − 1
5. state = CS
6. CS
7. ∀ r ∈ Q send REPLY to r
8. Q = ∅; state = NCS; replies = 0

upon receipt REQUEST(t) from p_j
IF state = CS or (state = Requesting and {Last_Req, i} < {t, j})
insert in Q {t, j}
else
send REPLY to p_j
num = max(t, num)

upon receipt REPLY from p_j
replies = replies + 1

upon receipt CRASH(p) ⇝ PerfectFailureDetector notify that P crashed
if p in Q
remove p from Q
n = n − 1 ⇝ we have minus processes

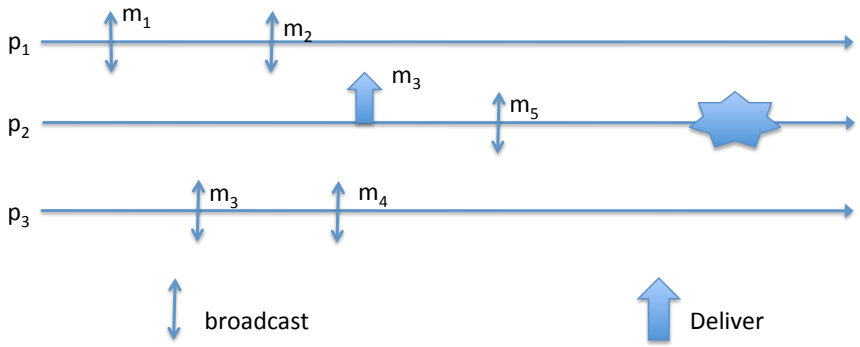
Simply we manage the crash event in the way that if crashed process is in Queue for enter in CS then we delete it. Also we decrease n as the process don't wait for another reply for enter in CS.

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2022/2023

Lecture 11 – Exercises
October 20th, 2022

Ex 1: Consider the partial execution depicted in the following figure:



1. Complete the execution in order to obtain a run satisfying *Best Effort Broadcast* but *not Reliable Broadcast.*
2. Complete the execution in order to obtain a run satisfying *Regular Reliable Broadcast* but *not Uniform Reliable Broadcast.*
3. Complete the execution in order to obtain a run satisfying *Uniform Reliable Broadcast.*

Ex 2: Consider a distributed system composed by n processes $\{p_1,p_2,\ldots,p_n\}$. Each process is connected to all the others trough fair-loss point-to-point links and has access to a perfect failure detector.

Write the pseudo-code of an algorithm implementing a Uniform Reliable Broadcast primitive.

Additionally, answer to the following questions:

1. Is it possible to provide a quiescent implementation of the Uniform Reliable Broadcast primitive?
2. Given the system model described here, is it possible to provide an implementation that uses only data structure with finite size?

Ex 3: Consider a distributed system composed by N servers $\{s_1, s_2, \ldots s_n\}$ and M clients $\{c_1, c_2, \ldots c_m\}$.
Each client c_i runs its algorithm and it can request to servers the execution of a particular task T_i. Servers will execute the task T_i and, after that, a notification will be sent to c_i that T_i has been completed.
The Figure shows the code executed by a generic client c_i.

Operation executeTask (T_i)	Upon pp2pdeliver (TASK_COMPLETED, T_i) from s_j
1. For each $s_i \in \{s_1, s_2, \ldots s_n\}$ 2. pp2psend (TASK_REQ, T_i, c_i) to s_i;	1. **trigger** completedTask (T_i);

Write the pseudo-code of an algorithm, executed by servers, able to allocate tasks assuming that:

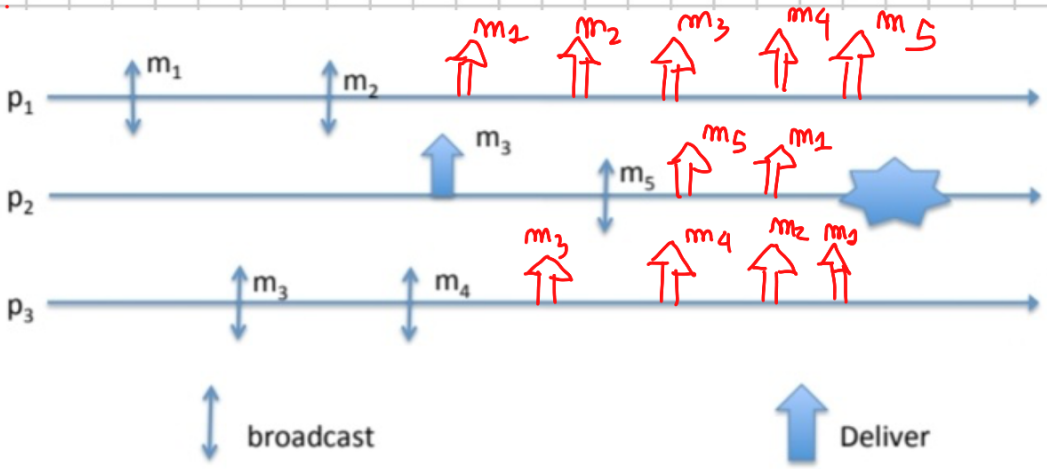
- Once clients ask for a task execution, they remain blocked until the task is not terminated.
- Any two clients c_i and c_j can concurrently require the execution of two different tasks T_i and T_j;
- Each task is univocally identified by the pair (T_i, c_i);
- Each server can manage at most one task at every time;
- At most N-1 servers can crash;
- Servers can use a uniform consensus primitive;
- Servers can use a failure detector P;
- Servers communicate through a uniform reliable broadcast primitive.

Note that, if a server crashes while executing a task, such task needs to be re-allocated and re-processed by a different server.

Ex 4: Consider a distributed system formed by n processes p_1, p_2, …, p_n connected along a ring i.e., a process p_i is initially connected to a process $p_{(i+1) \bmod n}$ through a unidirectional perfect point-to-point link.
Write the pseudo-code of a distributed algorithm implementing a consensus primitive.

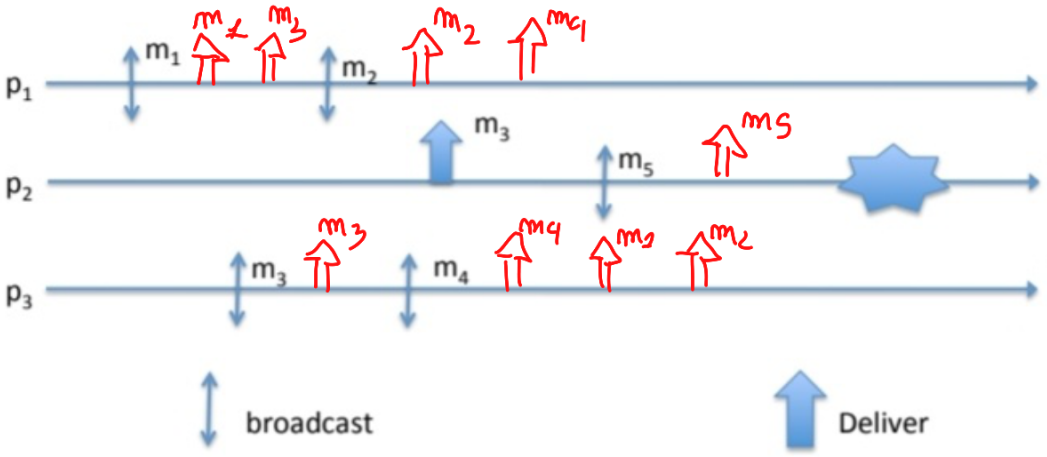
Exercise 1

1.



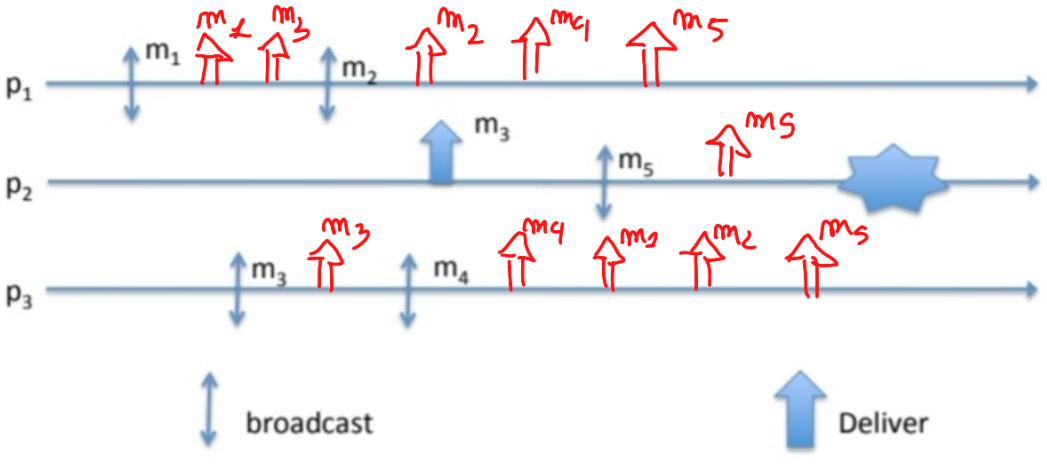
Best effort Broadcast

2.



REgular reliable Broadcast

3.



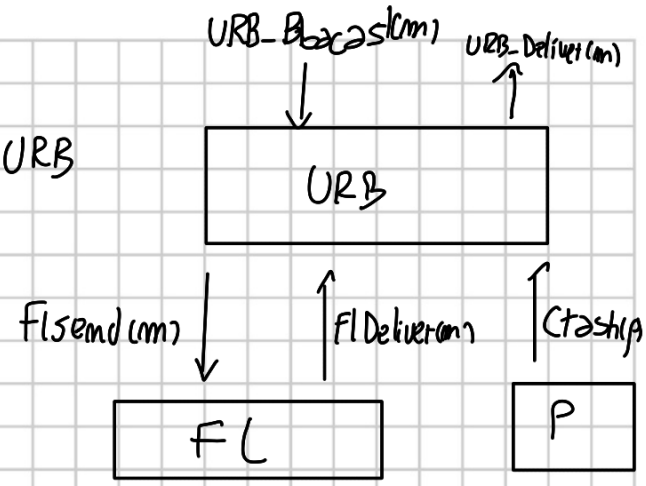
Uniform reliable Broadcast

Exercise 2

implements: Uniform Reliable Broadcast, URB

Uses: Fair-loss-point-to-point, FL

Perfect Failure Detector, P



init:

timer = Δ, starttimer(timer)

delivered = ∅

pending = ∅

correct = Π

forall m do ack[m] = ∅

upon event URB_Broadcast(m) do

pending = pending ∪ {(self, m)}

forall p_i ∈ correct do trigger FLsend(m) to p_i

Upon event FLDeliver(m) from p_j

ack[m] = ack[m] ∪ {p_j}

if (p_j, m) ∉ pending and m ∉ delivered

pending = pending ∪ {(p_j, m)}

Forall p_i ∈ correct do trigger FLsend(m) to p_i

Upon event Crash(p) do

correct = correct \ {p}

Function candeliver(m) returns Boolean is

return (correct ⊆ ack[m]);

Upon exist (S, m) ∈ pending such that candeliver(m)

∧ m ∉ delivered do

delivered = delivered ∪ {m}

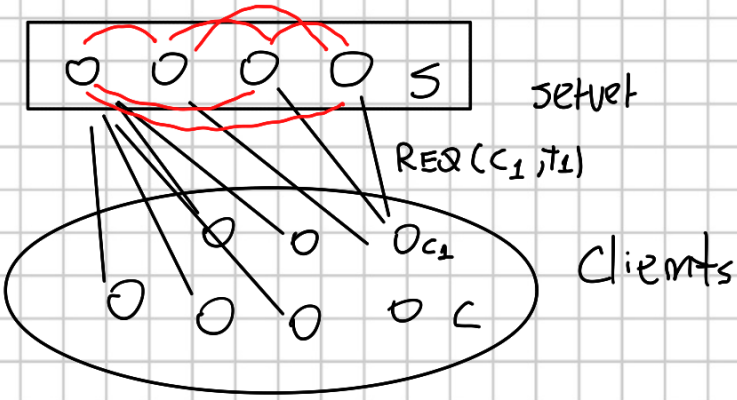
trigger URB_Deliver(m)

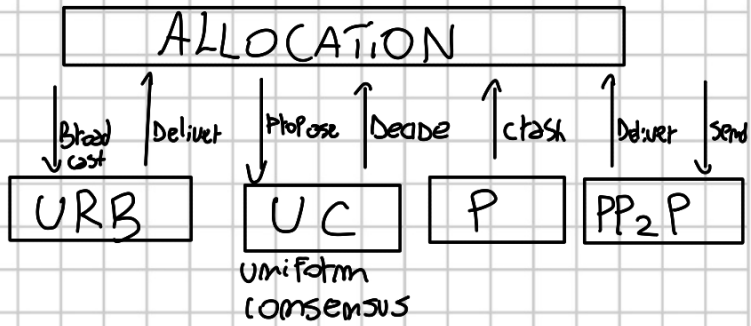
Upon timer expired do
  for all $p \in$ pending and $p \notin$ delivered do
    trigger fl send(m) to p
  start timer(timer)

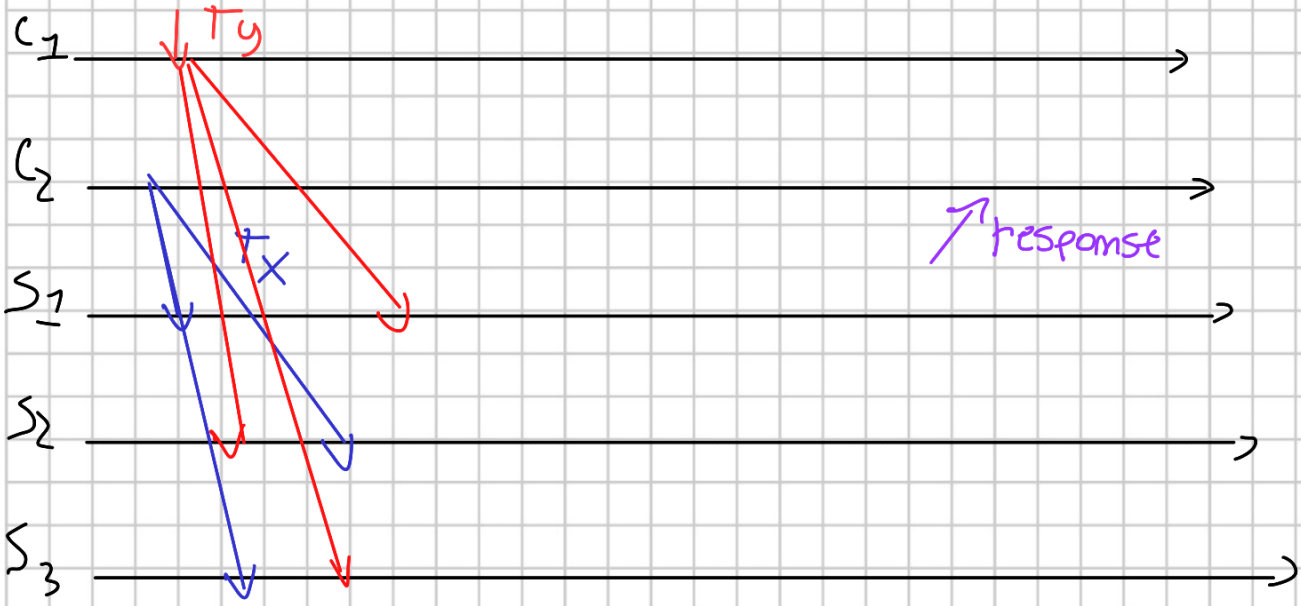
1 it is possible to have a quiescient implementation of PP2P link sending the ack, avoid to teltasmitting forever

2. if the system continuosly Broadcast the messages, you can not avoid to use an infinite data structure, if at some point you wait them you can bound the memory. You can erase something only when you are sure no more messages concerning the delivered messages.

Exercise 3







init:

pending = ∅

busy = false

current_allocation = null

upon event $\langle PL, PP_2P Deliver \mid TASK\_REQ, T_i, C_i \rangle$ from C_i

if $\langle T_i, C_i \rangle$ is not in current_allocation do

pending = pending $\cup \{\langle T_i, C_i \rangle\}$

when pending is not empty do

IF not busy do

candidate = select_from(pending)

trigger $\langle UC, Propose \mid \langle self, candidate \rangle \rangle$

else

trigger $\langle UC, Propose \mid \langle self, null \rangle \rangle$

upon event $\langle UC, Decide \mid \langle id, Task \rangle \rangle$

current_allocation = current_allocation $\cup \{\langle id, task \rangle\}$

IF id == self and task != null do

busy = true

execute(T_i)

busy = false

trigger $\langle PL, Pp2Psend \mid TASK\_COMPLETED, T_i \rangle$ to c_i

trigger $\langle URB, URBbroadcast \mid TASK\_COMPLETED, T_i, c_i \rangle$

upon event $\langle URB, URBDeliver \mid TASK\_COMPLETED, T_i, c_i \rangle$

current_allocation = current_allocation $\setminus \{\langle T_i, c_i \rangle\}$

Pending = pending $\setminus \{\langle T_i, c_i \rangle\}$

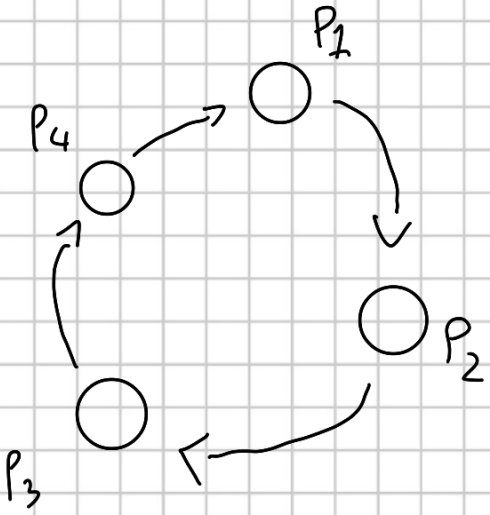
upon event $\langle P, ctask \mid s \rangle$

IF there exist <id, task> in current_allocation such that id == s do

current_allocation = current_allocation $\setminus \{\langle id, task \rangle\}$

pending = pending $\cup \{Task\}$

Exercise 4



Use a token as a collector, you add your proposal and pass it to the next, when come back to you all proposal are collected!
TOKEN initially is a empty list

init
  state = wait
  next = $P_{(i+1) \bmod N}$
  IF self = P_0
    TOKEN = TOKEN $\cup \{V\}$ ← value proposed by P_i
    trigger PP2PSend (TOKEN) to next

Upon event PP2PDeliver (TOKEN)
  if size (TOKEN) == N → all proposal recived
    decision = min (TOKEN) ⇝ min value of TOKEN list
    trigger PP2Psend (decision) to next
    trigger Decide (decision)
  else
    TOKEN = TOKEN $\cup \{V\}$
    trigger PP2Psend (TOKEN) to next

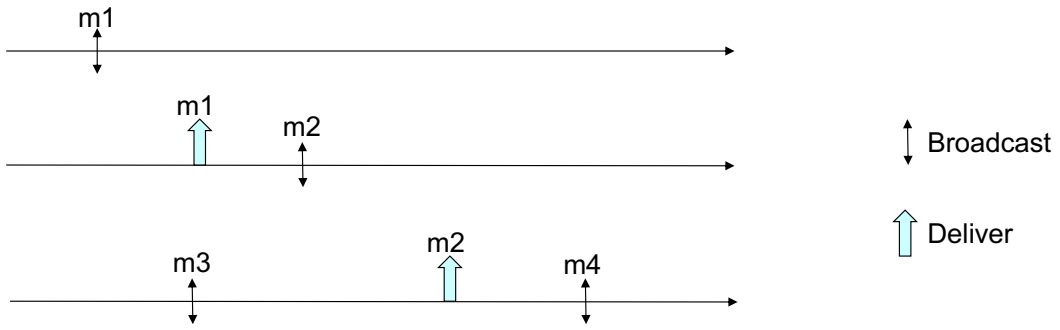
Upon event PP2PDeliver (decision)
  if state == wait
    state = done
    trigger PP2Psend (decision) to next
    trigger Decide (decision)

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

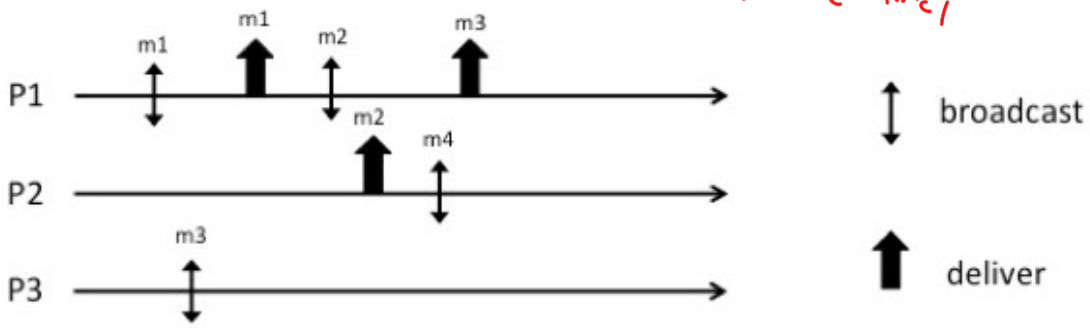
AA 2022/2023

Lecture 13 – Exercises
October 26th, 2022

Ex 1: Given the partial execution in Figure, provide all the delivery sequences such that both total order and causal order are satisfied



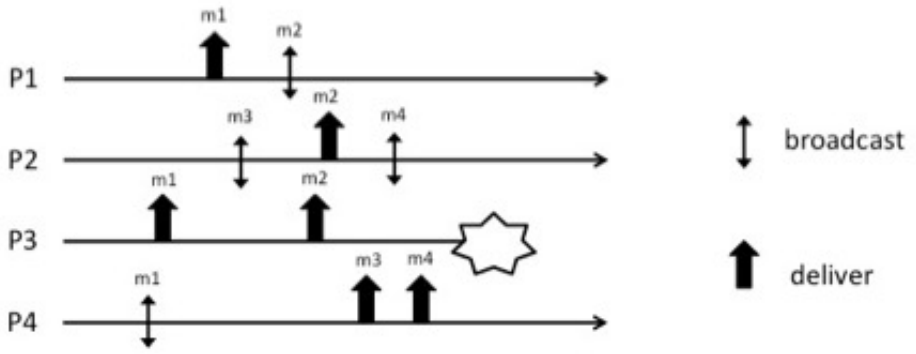
Ex 2: Let us consider the following partial execution



Answer the following points:

1. Provide all the possible sequences satisfying Causal Order
2. Complete the execution to have a run satisfying FIFO order but not causal order

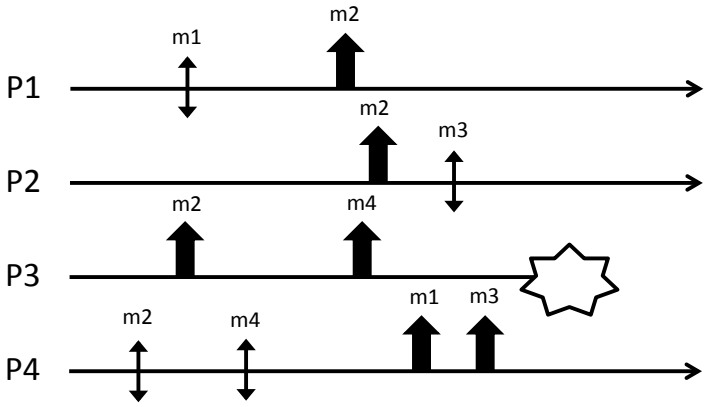
Ex 3: Let us consider the following partial execution



Answer the following points:

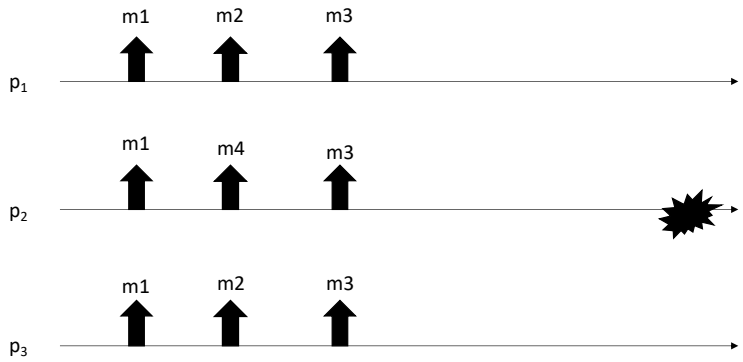
1. Provide the list al all the possible delivery sequences that satisfy both Total Order and Causal Order
2. Complete the history (by adding the missing delivery events) to satisfy Total Order but not Causal Order
3. Complete the history (by adding the missing delivery events) to satisfy FIFO Order but not Causal Order nor Total Order

Ex 4: Consider the message pattern shown in the Figure below and answer to the following questions:



1. Complete the execution in order to have a run satisfying Reliable Broadcast but not Uniform Reliable Broadcast.
2. Provide all the delivery sequences satisfying causal order and total order.
3. Provide all the delivery sequences violating causal order and satisfying TO(UA, WNUTO) but not satisfying TO(UA, SUTO)

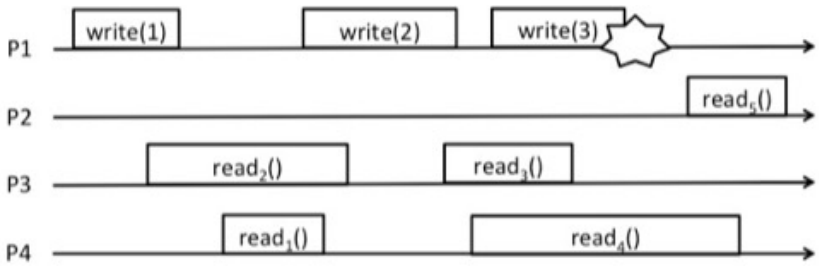
Ex 5: Consider the partial execution in the following figure



Given the run depicted in the figure state the truthfulness of the following sentences:			
a	The strongest agreement property satisfied is UA	T	F
b	The NUA agreement property is violated	T	F
c	The strongest ordering property satisfied is SUTO	T	F
d	The WUTO ordering property is satisfied	T	F
e	The SNUTO ordering property is violated	T	F
f	Let us assume we can add only one more delivery to p1 and p3, it is not possible to get a run satisfying TO(NUA, SUTO)	T	F
g	If p2 is not going to deliver m4 then the strongest specification satisfied by the resulting execution is TO(UA, SUTO)	T	F
h	Let us assume we can add only one more delivery to p1 and p3, it is possible to get a run satisfying TO(UA, WNUTO) but not satisfying TO(UA, WUTO)	T	F
i	If p2 is not faulty, the NUA agreement property is satisfied	T	F
l	If p2 is not faulty, the SUTO ordering property is satisfied	T	F

For each point, provide a justification for your answer

Ex 6: Consider the execution depicted in the following figure and answer the questions



1. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to a regular register.
2. Define ALL the values that can be returned by read operations (Rx) assuming the run refers to an atomic register.

Ex 7: Consider a distributed systems composed by a set of *n* processes p_1, p_2,… p_n. Processes have a unique identifier and are structured as a binary tree topology. Messages are exchanged between processes over the edges of the tree which act like perfect point-to-point links. Each process p_i has stored the identifiers of its neighbors

into the local variables FATHER, R_CHILD e L_CHILD representing respectively the father of p_i, the right child and the left child (if they exists).
Assuming that processes are not going to fail, write the pseudo-code of an algorithm satisfying the following specification:

Events:

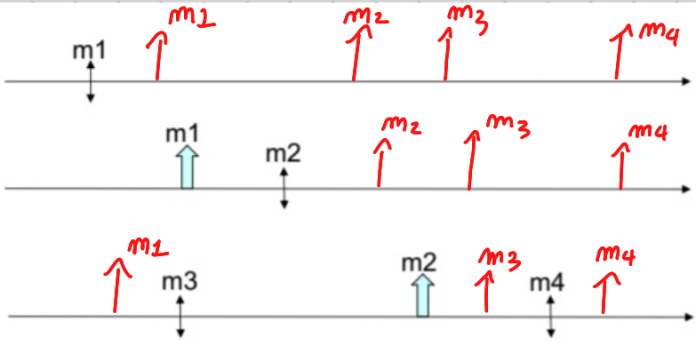
- **Request:** ⟨ *tob*, Broadcast | m ⟩: Broadcasts a message m to all processes.
- **Indication:** ⟨ *tob*, Deliver | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

- *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*.
- *Agreement:* If a message *m* is delivered by some correct process, then *m* is eventually delivered by every correct process.
- *Total order:* Let m_1 and m_2 be any two messages and suppose *p* and *q* are any two correct processes that deliver m_1 and m_2. If *p* delivers m_1 before m_2, then *q* delivers m_1 before m_2.

Exercise 1

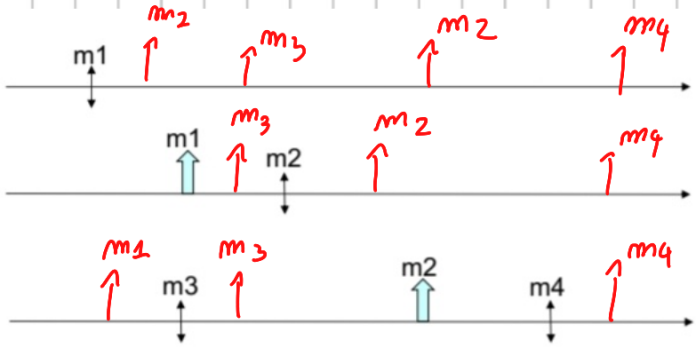
$m_1 \rightarrow m_2$ and $m_2 \rightarrow m_4$ (local order) $m_3 \rightarrow m_4$ (Fifo order)

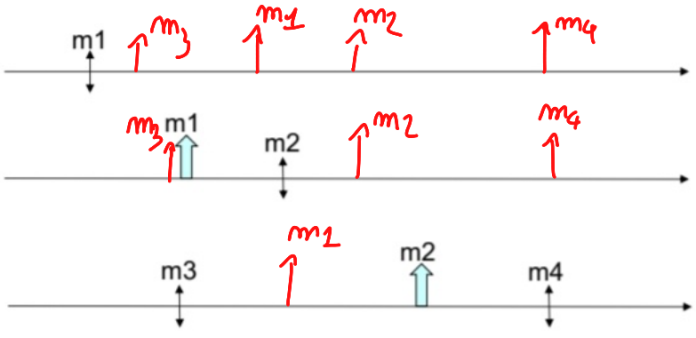


m_1, m_2, m_3, m_4

m_1, m_3, m_2, m_4

m_3, m_1, m_2, m_4





Exercise 2

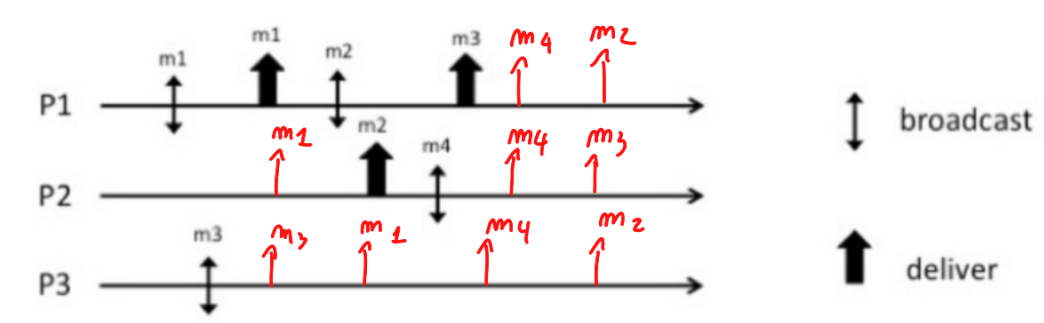
$m_1 \longrightarrow m_2 \longrightarrow m_4$ (local order + Fifo order)

1. sequence of deliveries:

We have four case:

- (m_1, m_2, m_3, m_4)
- (m_1, m_3, m_2, m_4)
- (m_1, m_2, m_4, m_3)
- (m_3, m_1, m_2, m_4)

2.



FIFO Broadcast but not Causal

Exercise 3

Causal order = Fifo order + local order : $m_1 \rightarrow m_2 \rightarrow m_4$

$m_3 \rightarrow m_4$

1. Total order and causal order

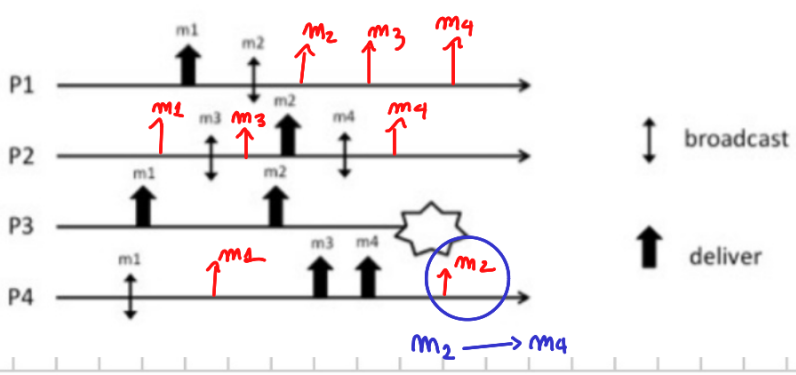
- m_1, m_2, m_3, m_4
- m_1, m_3, m_2, m_4

because of delivery of m_1 in P_1 is before of broadcast of m_3 in P_2 we can't have a sequence that start with m_3

2.

not exist an execution that guarantee Total order but not causal order given the delivery and add only the missing delivery

3.



Fifo order but not causal or total