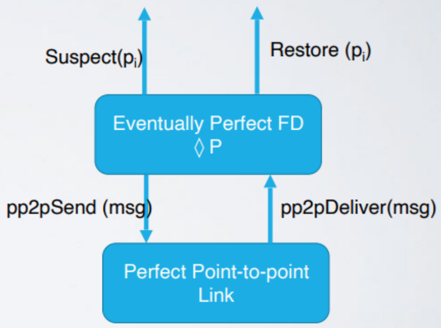
5. TIME

EVENTUALLY PERFECT FAILURE DETECTORS $\Diamond P$

> **Algorithm**:
>
> ```
> Implements:
>     EventuallyPerfectFailureDetector, instance ◇P.
>
> Uses:
>     PerfectPointToPointLinks, instance pl.
>
> upon event ⟨ ◇P, Init ⟩ do
>     alive := Π;
>     suspected := ∅;
>     delay := Δ;
>     starttimer(delay);
>
> upon event ⟨ Timeout ⟩ do
>     if alive ∩ suspected ≠ ∅ then
>         delay := delay + Δ;
>     forall p ∈ Π do
>         if (p ∉ alive) ∧ (p ∉ suspected) then
>             suspected := suspected ∪ {p};
>             trigger ⟨ ◇P, Suspect | p ⟩;
>         else if (p ∈ alive) ∧ (p ∈ suspected) then
>             suspected := suspected \ {p};
>             trigger ⟨ ◇P, Restore | p ⟩;
>         trigger ⟨ pl, Send | p, [HEARTBEATREQUEST] ⟩;
>     alive := ∅;
>     starttimer(delay);
>
> upon event ⟨ pl, Deliver | q, [HEARTBEATREQUEST] ⟩ do
>     trigger ⟨ pl, Send | q, [HEARTBEATREPLY] ⟩;
>
> upon event ⟨ pl, Deliver | p, [HEARTBEATREPLY] ⟩ do
>     alive := alive ∪ {p};
> ```
>
> 
>

Strong Completeness: if a process crashes, it will stop to send messages. Therefore the process will be suspected by any correct process and no process will revise the judgement.

Eventual Strong Accuracy: after time t the system becomes synchronous. After that time a message sent by a correct process p to another one q will be delivered within a bounded time (the time is MAX_{DELAY} unknown to us). If p was wrongly suspected by q, then q will revise its suspicious. Moreover q increases its delay, if the new delay is less than MAX_{DELAY}, eventually q suspects again p and then it corrects again its delay. After a finite number of errors the delay of q will be greater than MAX_{DELAY}.

Is possible that the deelay is big enough that can happen a situation where a process p_2 communicates at process p_1 that he's dead but the message is recived after the detection of p_2 dead line from p_1.
Solution: add brenches T_x.

Property of Suspected:

> **Lemma**: take two correct processes, p_1 and p_2 and let $suspected_1$ and $suspected_2$ be the respective sets. There is a time t (stabilization time), after which $suspected_1 = suspected_2$.

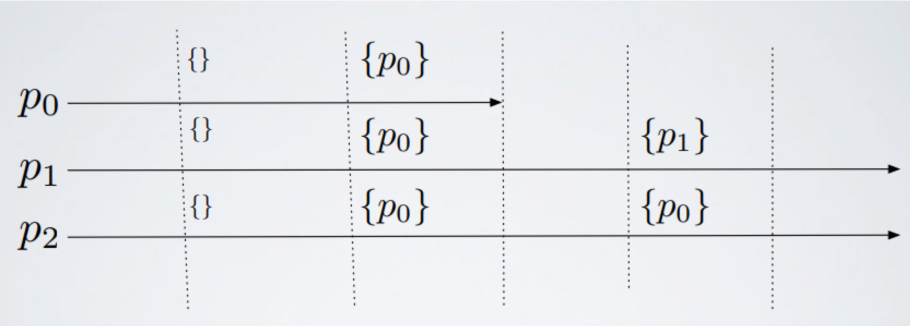
Proof (by contraddiction): suppose that exists a p ina $suspected_1$ but not $suspected_2$:

- if p is correct this violates the **eventual strong accuracy**.
- if p crashed this violates the **strong completeness**.

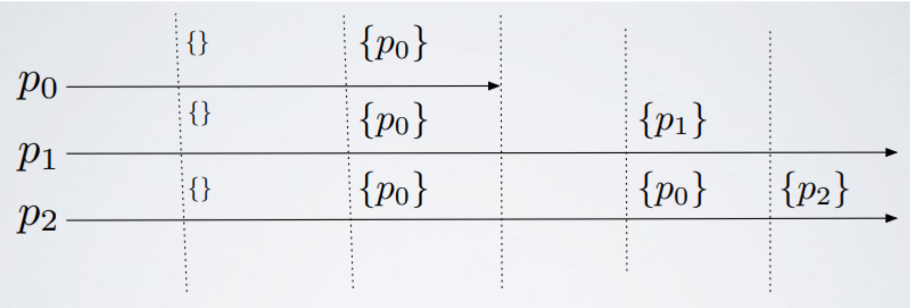
Leader Election

Sometimes, we may be interested in knowing one process that is alive instead of monitoring failures. In this case we can use a different oracle (called **leader election module**) that reports a process that is alive.

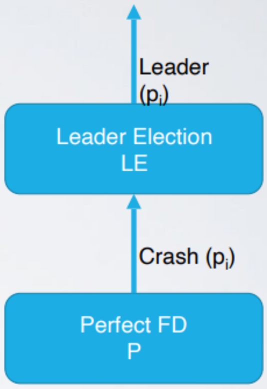
- **Name**: LeaderElection, **istance** le.
- **Indication**: $\langle le, Leader \mid p\rangle$: indicates that process p **is elected as a leader**.
- **Properties**:
  - **LE1** (eventual detection): either there is no correct process, or some correct process is eventually elected as the leader (liveness property).
  - **LE2** (accuracy): if a process is leader, then all previously elected leaders have crashed



When p_0 dies it's okay because before he dies everyone know that he's the leader, but in p_2 after p_0 dead, the leader isn't changed. (broke the eventual detection property).



In this case the accuracy propery is broken because p_2 changes the leader even if p_1 is still alive.

> **Algorithm**:
> **Implements:**
> LeaderElection, **instance** *le*.
>
> **Uses:**
> PerfectFailureDetector, **instance** $\mathcal{P}$.
>
> **upon event** $\langle$ *le*, *Init* $\rangle$ **do**
> $suspected := \emptyset$;
> $leader := \bot$;
>
> **upon event** $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
> $suspected := suspected \cup \{p\}$;
>
> **upon** $leader \neq \text{maxrank}(\Pi \setminus suspected)$ **do**
> $leader := \text{maxrank}(\Pi \setminus suspected)$;
> **trigger** $\langle$ *le*, Leader | *leader* $\rangle$;
>
> 
>
>
> $(\Pi \setminus suspected)$ is the subset of alive processes.

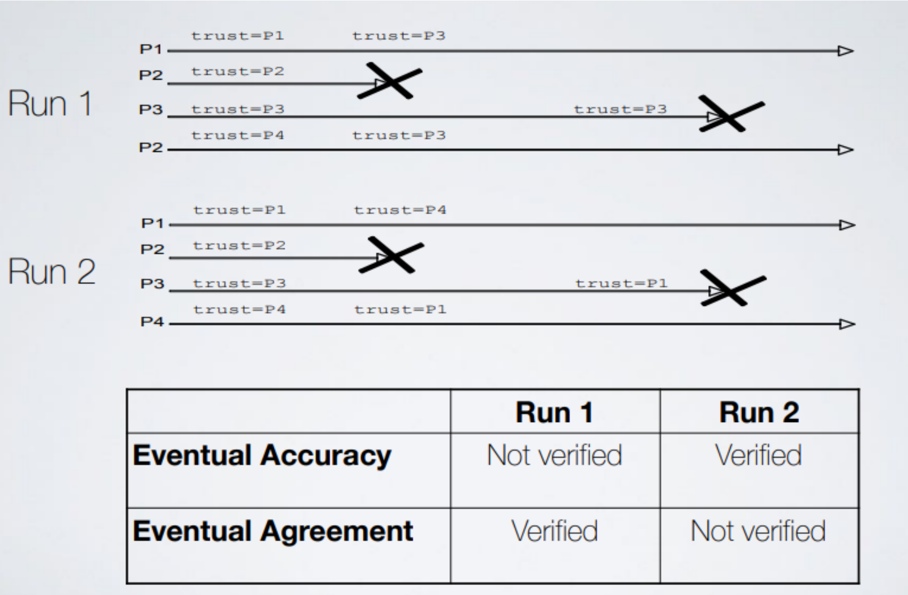
Correctness:

- **Eventual detection**: from the strong completeness of P (perfect failure).
- **Accuracy**: from the strong accuracy of P and the total order on the ranks (IDs) of processes.

Eventual Leader Election Ω

- **Name**: $EventualLeaderDetector$, **istance** Ω.
- **Indication**: $\langle \Omega, Trust \mid p \rangle$: indicates that process p is trusted to be leader.
- **Properties**:
  - **ELD1** (eventual accuracy): there is a time after which every correct process trusts some correct process.
  - **ELD2** (eventual agreement): there ia time after which no two correct processes trust different correct processes.
  Both of properties are **liveness**.

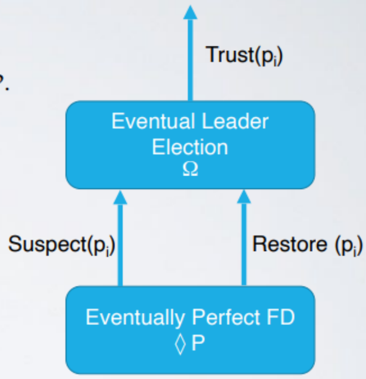
Ω ensures that **eventually** correct processes will elect the same correct process as their leader.



	Run 1	Run 2
Eventual Accuracy	Not verified	Verified
Eventual Agreement	Verified	Not verified

We can build an eventual leader election using **crash-stop process abstraction**:

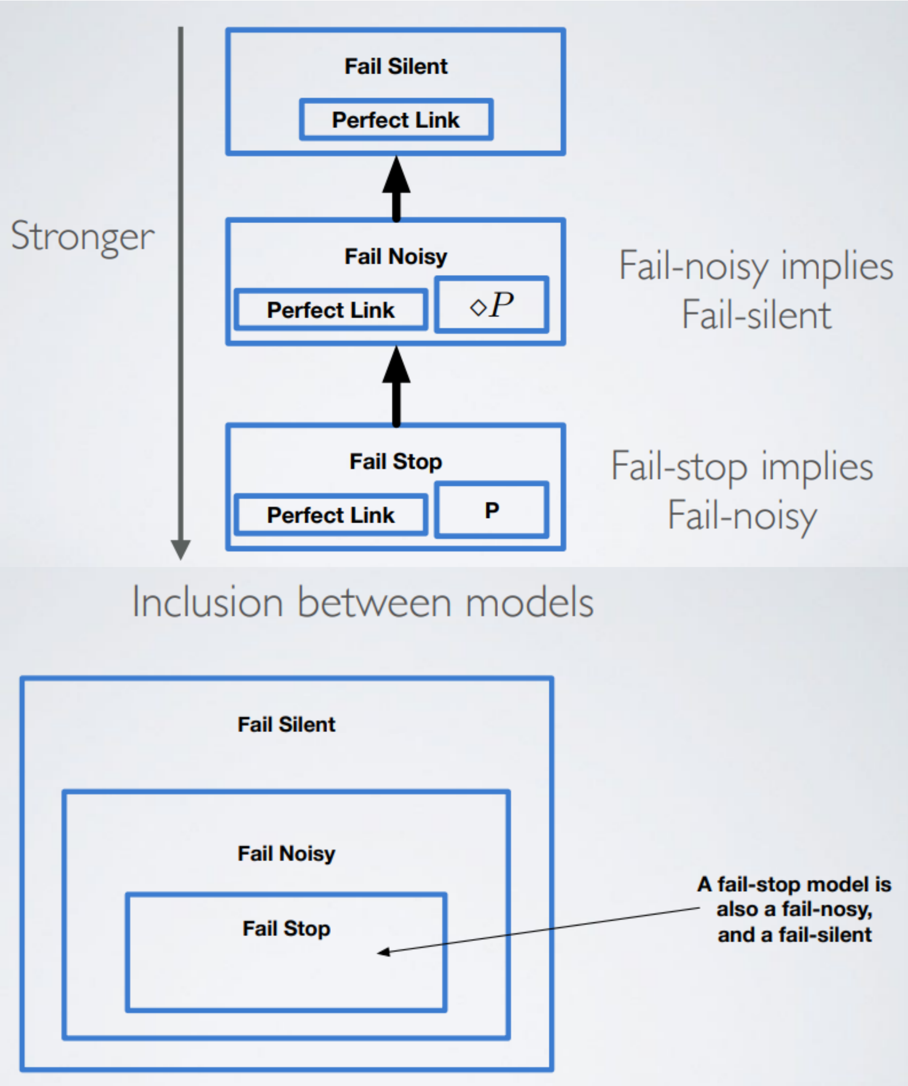
- Obtained directly by $\Diamond P$ by using a deterministic rule on processes that are not suspected by $\Diamond P$.
- Trust the process with the highest identifier among all processes that are not suspected by $\Diamond P$.

> **Algorithm**:
>
> **Implements:**
> EventualLeaderDetector, **instance** Ω.
>
> **Uses:**
> EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.
>
> **upon event** $\langle$ *Ω, Init* $\rangle$ **do**
> *suspected* := $\emptyset$;
> *leader* := $\perp$;
>
> **upon event** $\langle$ *$\Diamond\mathcal{P}$, Suspect* | *p* $\rangle$ **do**
> *suspected* := *suspected* $\cup$ $\{p\}$;
>
> **upon event** $\langle$ *$\Diamond\mathcal{P}$, Restore* | *p* $\rangle$ **do**
> *suspected* := *suspected* $\setminus$ $\{p\}$;
>
> **upon** *leader* $\neq$ maxrank(Π $\setminus$ *suspected*) **do**
> *leader* := maxrank(Π $\setminus$ *suspected*);
> **trigger** $\langle$ *Ω, Trust* | *leader* $\rangle$;
>
> 
>

Proof:

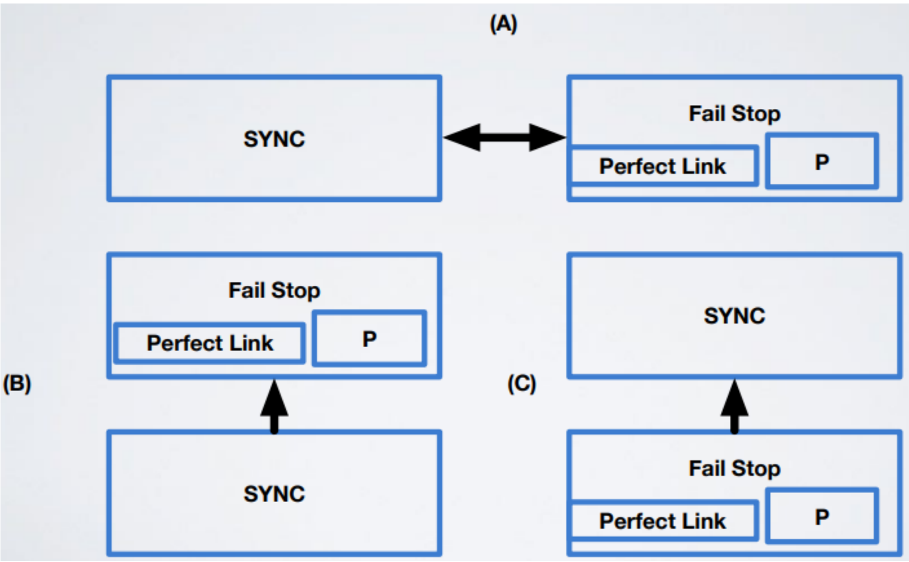
- **ELD1** (Eventual Accuracy): by the strong completeness of the FD we have that eventually suspected set contains all the crashed processes. Thus $\Pi \setminus suspected$ contains only correct processes (or its empty).
- **ELD2** (Eventual Agreement): for any pair of correct processes, their suspected sets eventually stabilises to the same content (by the property of the FD). If the set are equals $\Pi \setminus suspected$ returns the same ID on both processes.

THREE MODELS



We can also say that the set of problems solvable in fail-stop includes the problems solvable in other models.

Relationship between Fail-Stop and SYNC:



SYNC is stronger than Fail-Stop (Correct answer is B !!!!possibile domanda esame!!!!).
Problems we can solve with fail-stop is strictly contained in subset of problems we can solve

with SYNC (SYNC can solve problems based on **time** like clock synchronization):

APPLICATION OF FAILURE DETECTOR AND LEADER ELECTOR

Using P to make **Lamport's Mutual Exclusion fault tolerant**.
Events:

- *Request*: from upper layer - requests access to Critical Section (CS).
- *Grant*: to upper layer - grant the access to CS.
- *Release*: from upper layer - release the CS.
  Properties:
- **Mutual Exclusions**: at any time t, at most one non-crashed process is inside the CS.
- **Liveness**: if a correct process p requests access, then it eventually enters the CS.
- **Fairness**: if a correct process p requests access before a process q, then q cannot access the CS before p.

ORIGINAL ALGORITHM

Critical points:
1) if you crashed you cannot release
2) If you crashed you cannot ack

```
upon event INIT
    Requests = Acks = ∅
    scalar_clock = 0
    my_req = ⊥
    Π = {p0, p1, ..., pn−1}                          ▷ Set of all processes

▷ Request access to CS from upper layer
upon event REQUEST
    scalar_clock = scalar_clock + 1
    my_req = (REQ, ts =< i, scalar_clock >)
    for all pj ∈ Π do
        SEND FIFOPERFECTLINK(pj, req_msg)   ▷ Send a REQ containing my ID (i) and ts (scalar_clock) to all p ∈ Π

▷ Release CS from upper layer
upon event RELEASE
    Requests = Requests \ {req_msg}
    scalar_clock = scalar_clock + 1
    for all pj ∈ Π do
        SEND FIFOPERFECTLINK(pj, (RLS, ts =< i, scalar_clock >))

▷ ts(x) < ts(y) when scalar_clock of x is less than the one of y, or they are equal and the id that sent x is less
  than the id that sent y
upon event ∄req ∈ Requests : ts(req) < ts(my_req) ∧ ∀p ∈ Π : ∃m ∈ Acks|ts(m) > ts(my_req)
    trigger event GRANTED

upon event DELIVER MESSAGE(m)
    scalar_clock = max(clock(m), scalar_clock) + 1
    if m is a REQ then
        Request_set = Request_set ∪ {m}
        scalar_clock = scalar_clock + 1
        SEND FIFOPERFECTLINK(sender(m), (ACK, ts =< i, scalar_clock >))
    else if m is a ACK then
        Acks = Acks ∪ {m}
    else if m is a RLS ∧∃req ∈ Request_set : sender(req)=sender(m) then
        Requests = Requests \ {req}
```

Patch 1:

```
upon event INIT
    Requests = Acks = ∅
    scalar_clock = 0
    my_req = ⊥
    Π = {p0, p1, ..., pn−1}                          ▷ Set of all processes
    Crashed = {}

▷ Crash handler, the crash event is generated by a perfect failure detector P
upon event CRASH(process pj)
    Crashed = Crashed ∪ {pj}
    for all req ∈ Requests such that pj is the sender of req do
        remove req from Requests

▷ ts(x) < ts(y) when scalar_clock of x is less than the one of y, or they are equal and the id that sent x is less
  than the id that sent y
upon event ∄req ∈ Requests : ts(req) < ts(my_req) ∧ ∀p ∈ (Π \ Crashed) : ∃m ∈ Acks|ts(m) > ts(my_req)
    trigger event GRANTED
```

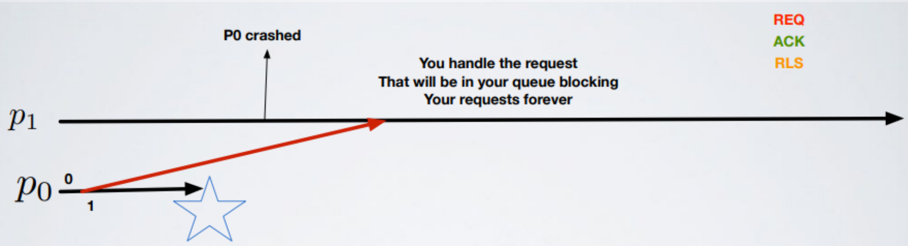
If the system receive a request from p_j after the crash detection we have a deadlock.

Patch 2:

upon event INIT
  $Requests = Acks = \emptyset$
  $scalar\_clock = 0$
  $my\_req = \perp$
  $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$ ▷ Set of all processes
  $Crashed = \{\}$

▷ Crash handler, the crash event is generated by a perfect failure detector P
upon event CRASH(process p_j)
  $Crashed = Crashed \cup \{p_j\}$
  for all $req \in Requests$ such that p_j is the sender of req **do**
    remove req from $Requests$

▷ $ts(x) < ts(y)$ when $scalar\_clock$ of x is less than the one of y, or they are equal and the id that sent x is less than the id that sent y
upon event $\nexists req \in Requests : ts(req) < ts(my\_req) \land \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
  trigger event GRANTED



If a process crashes I put a ban on all the events related to him.

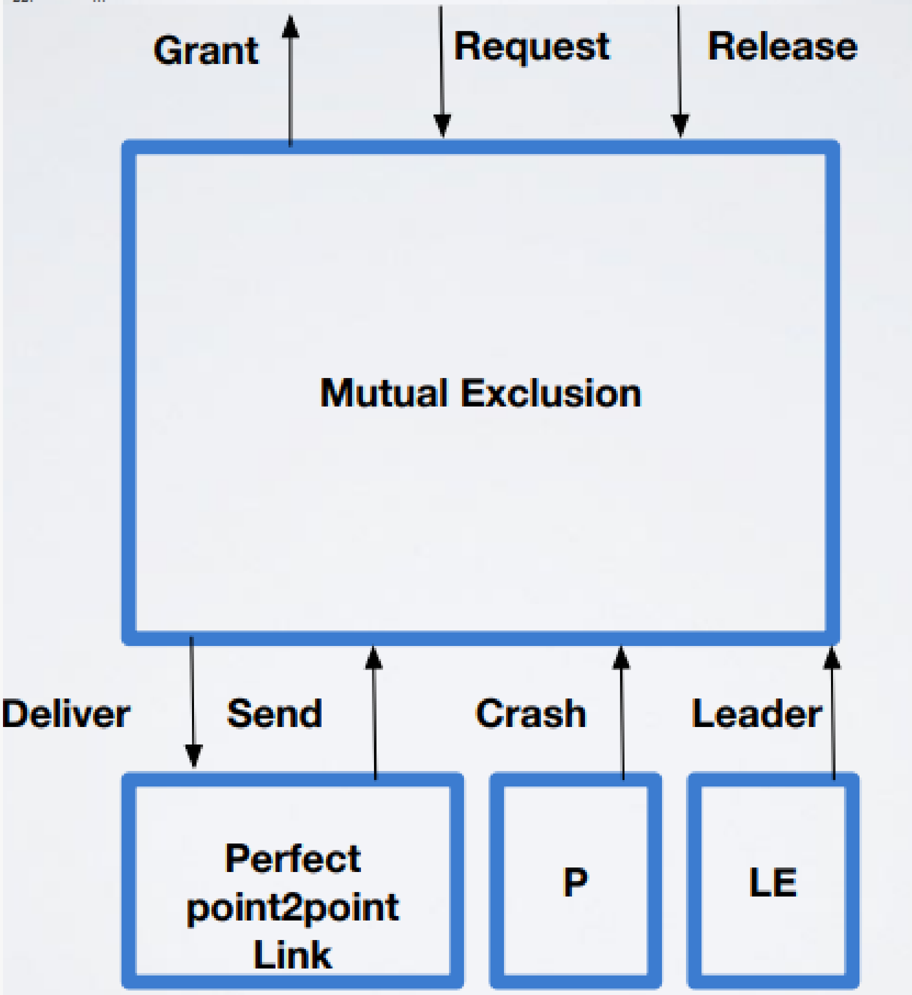
Full Algorithm with Patches:

```
upon event INIT
    Requests = Acks = ∅
    scalar_clock = 0
    my_req = ⊥
    Π = {p_0, p_1, ..., p_{n-1}}                                  ▷ Set of all processes
    Crashed = {}

▷ Crash handler, the crash event is generated by a perfect failure detector P
upon event CRASH(process p_j)
    Crashed = Crashed ∪ {p_j}
    for all req ∈ Requests such that p_j is the sender of req do
        remove req from Requests

▷ ts(x) < ts(y) when scalar_clock of x is less than the one of y, or they are equal and the id that sent x is less
  than the id that sent y
upon event ∄req ∈ Requests : ts(req) < ts(my_req) ∧ ∀p ∈ (Π \ Crashed) : ∃m ∈ Acks|ts(m) > ts(my_req)
    trigger event GRANTED

upon event DELIVER MESSAGE(m)
    if m is a REQ and the sender of m is not in Crashed then
        scalar_clock = max(clock(m), scalar_clock) + 1
        Request_set = Request_set ∪ {m}
        scalar_clock = scalar_clock + 1
        SEND FIFOPERFECTLINK(sender(m), (ACK, ts =< i, scalar_clock >))
    else if m is a ACK then
        ...
```



Exercise: write an algorithm with those properties:

- Use LE to elect a leader.
- Ask the leader for CS with a request message.
- The leader allows access to CS using FIFO order on requests.
- When done release CS using a release message.
- If the leader detects a crash p:
  - If p is not in CS, it removes the pending request of p (if any).
  - If p is in CS, it acts as p released the CS.

- *Problem*: what to do when a new leader is elected? The old leader was the only one to know who was in CS?