TIME IN DISTRIBUTED SYSTEMS-III

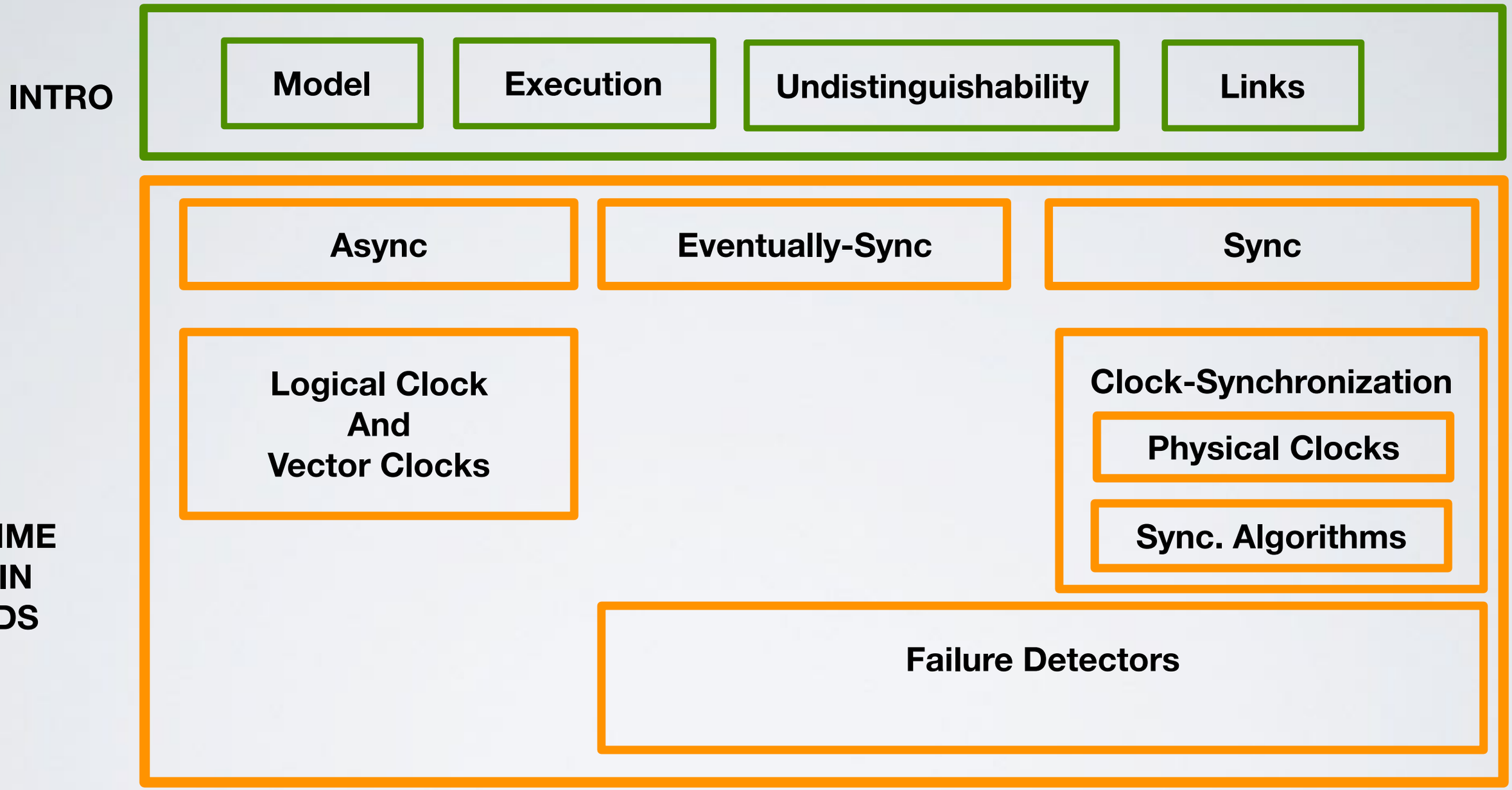
DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

WHERE WE ARE


INTRO
Model
Execution
Undistinguishability
Links
TIME
IN
DS
Async
Eventually-Sync
Sync
Logical Clock
And
Vector Clocks
Clock-Synchronization
Physical Clocks
Sync. Algorithms
Failure Detectors

RECAP ON TIMING ASSUMPTIONS

Synchronous

- timing assumptions are explicit on
  - Bounds on process executions and communication channels, or
  - Existence of a common global clock (apart from the synchronisation error that we neglect)

Asynchronous

- There are no timing assumptions
- The only ordering is given by the happened-before relationship
- We can build scalar clock or vector clocks.

RECAP ON TIMING ASSUMPTIONS

Eventual synchrony requires **abstract timing assumptions** (after an unknown time *t* the system becomes synchronous)

Two choices either:

- 1) Put assumption on the system model (including links and processes)
- 2) Create separate abstractions that encapsulates those timing assumptions

Note: **manipulating time inside a protocol/algorithm is complex** and the correctness proof may become very involved and sometimes prone to errors

FAILURE DETECTOR ABSTRACTION

A software module to be used together with process and link abstractions

It encapsulates timing assumptions of a either partially synchronous or fully synchronous system

The purpose of this module is to detect processes that crashed

NB: **Failure Detectors** only **work** for **Crash Failures**

FAILURE DETECTOR ABSTRACTION

The **stronger the timing assumption** are, the **more accurate** the information provided by a failure detector will be.

Described by two properties:

- **Accuracy** (informally is the ability to avoid mistakes in the detection)
- **Completeness** (informally is ability to detect all failures)

- They are based on the idea of **`Pinging"**

PERFECT FAILURE DETECTORS (P)

System model

- **Synchronous system**
- **Crash failures**
- **Perfect Synchronous Point2point**

Perfect Synchronous Point2point:

Perfect Point2point + synchronous delivery:

If a message m is sent at time t, m is received by at most t+max_delay.

PERFECT FAILURE DETECTORS (P)

Using its **clock** and the **bounds of the synchrony model**, a process can infer if another process has crashed

In asynch?

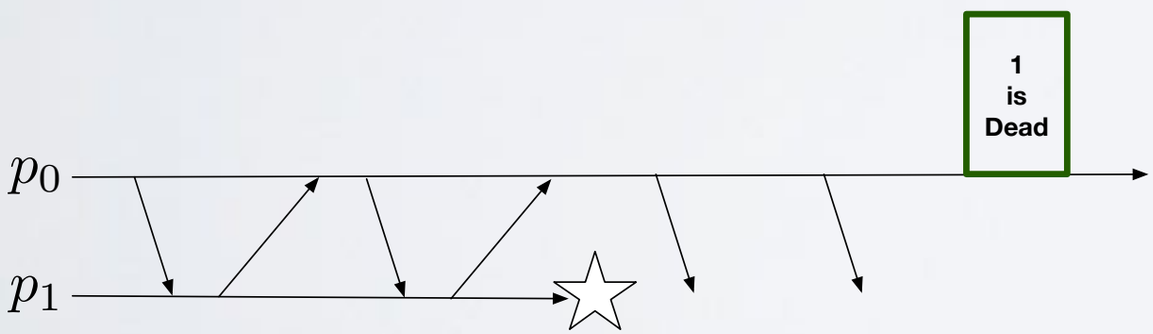
PERFECT FAILURE DETECTORS (P)

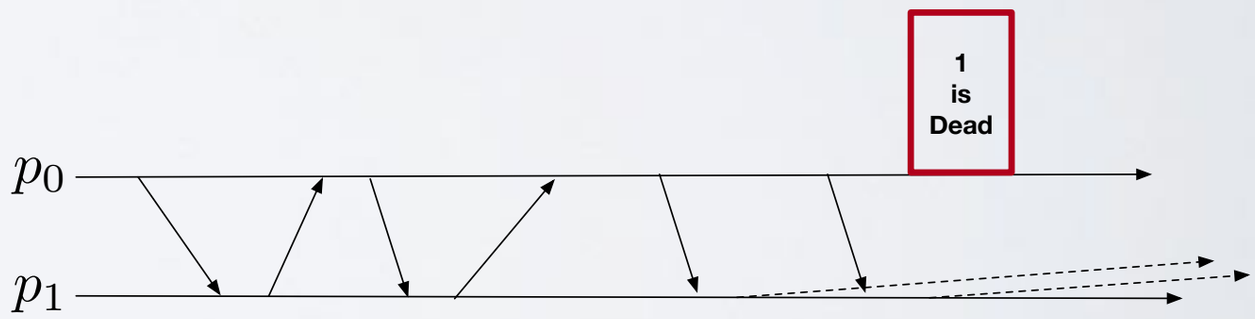
System model

- synchronous system
- crash failures

Using its clock and the bounds of the synchrony model, a process can infer if another process has crashed

In asynch?



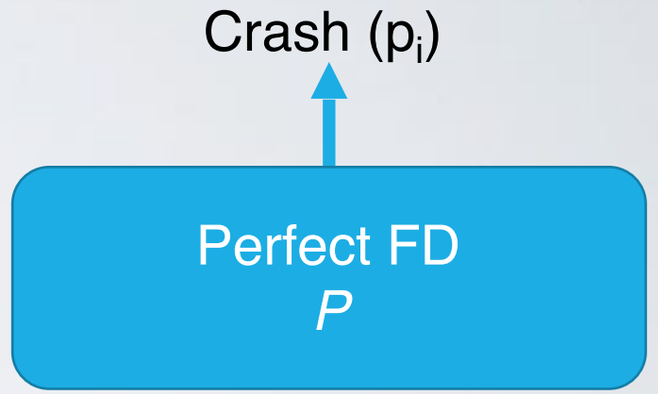


PERFECT FAILURE DETECTORS (P) SPECIFICATION

Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.



Events:

Indication: $\langle\ \mathcal{P}, Crash \mid p\ \rangle$: Detects that process p has crashed.

Properties:

PFD1: *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

PERFECT FAILURE DETECTORS (P) IMPLEMENTATION

Algorithm 2.5: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

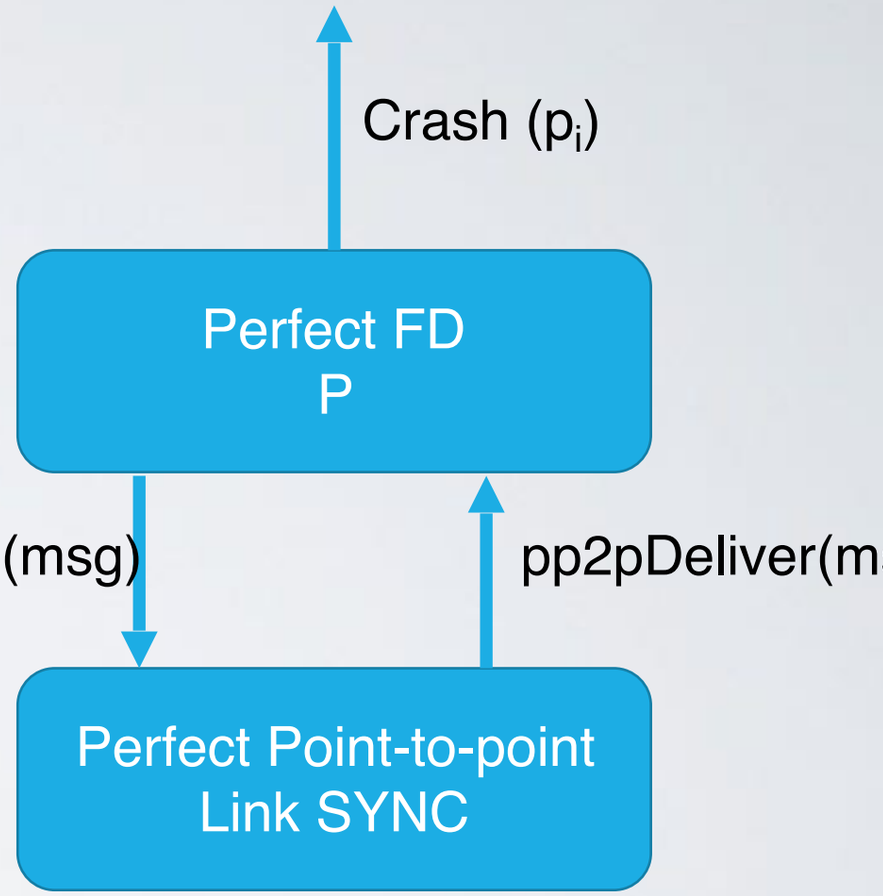
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P}, Init\ \rangle$ **do**
alive := Π;
detected := $\emptyset$;
starttimer(Δ);

upon event $\langle\ Timeout\ \rangle$ **do**
forall $p \in \Pi$ **do**
if $(p \notin alive) \wedge (p \notin detected)$ **then**
$detected := detected \cup \{p\}$;
trigger $\langle\ \mathcal{P}, Crash \mid p\ \rangle$;
trigger $\langle\ pl, Send \mid p, [\text{HEARTBEATREQUEST}]\ \rangle$;
alive := $\emptyset$;
starttimer(Δ);

upon event $\langle\ pl, Deliver \mid q, [\text{HEARTBEATREQUEST}]\ \rangle$ **do**
trigger $\langle\ pl, Send \mid q, [\text{HEARTBEATREPLY}]\ \rangle$;

upon event $\langle\ pl, Deliver \mid p, [\text{HEARTBEATREPLY}]\ \rangle$ **do**
alive := $alive \cup \{p\}$;



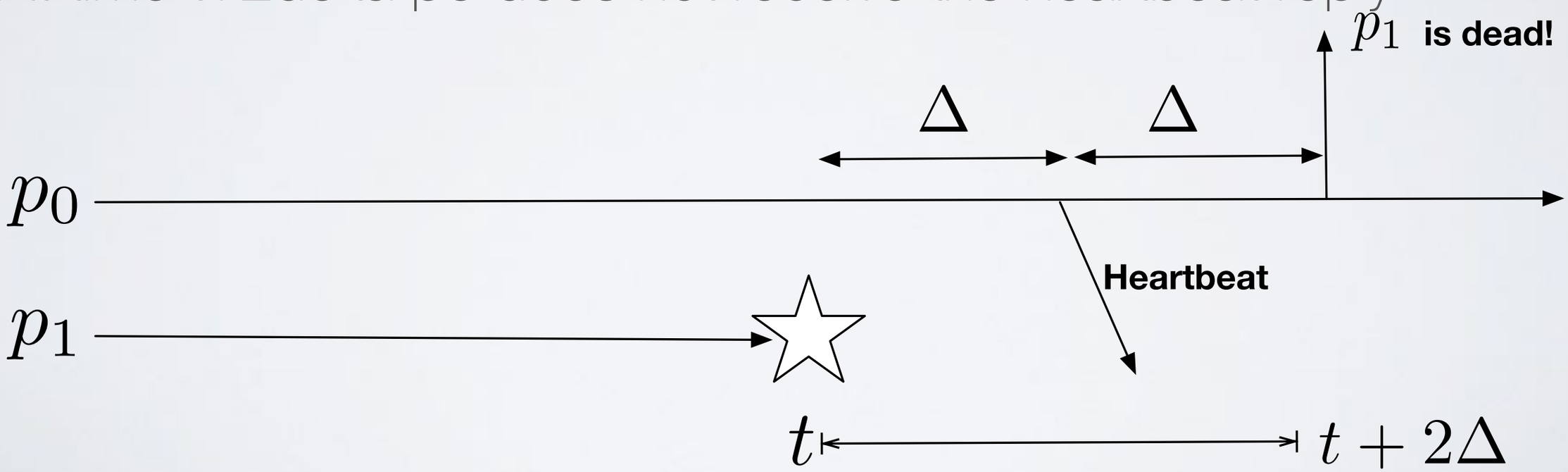
CORRECTNESS

Strong completeness:

Assume p1 crashes at time t, then p0 detects it at most by time t+2*delta.

1) At time t+delta (at most) p0 sends an heartbeat request

2) At time t+2delta p0 does not receive the heartbeat reply



CORRECTNESS

Strong accuracy:

$\text{Assuming } 2 \times max\_delay \leq \Delta$

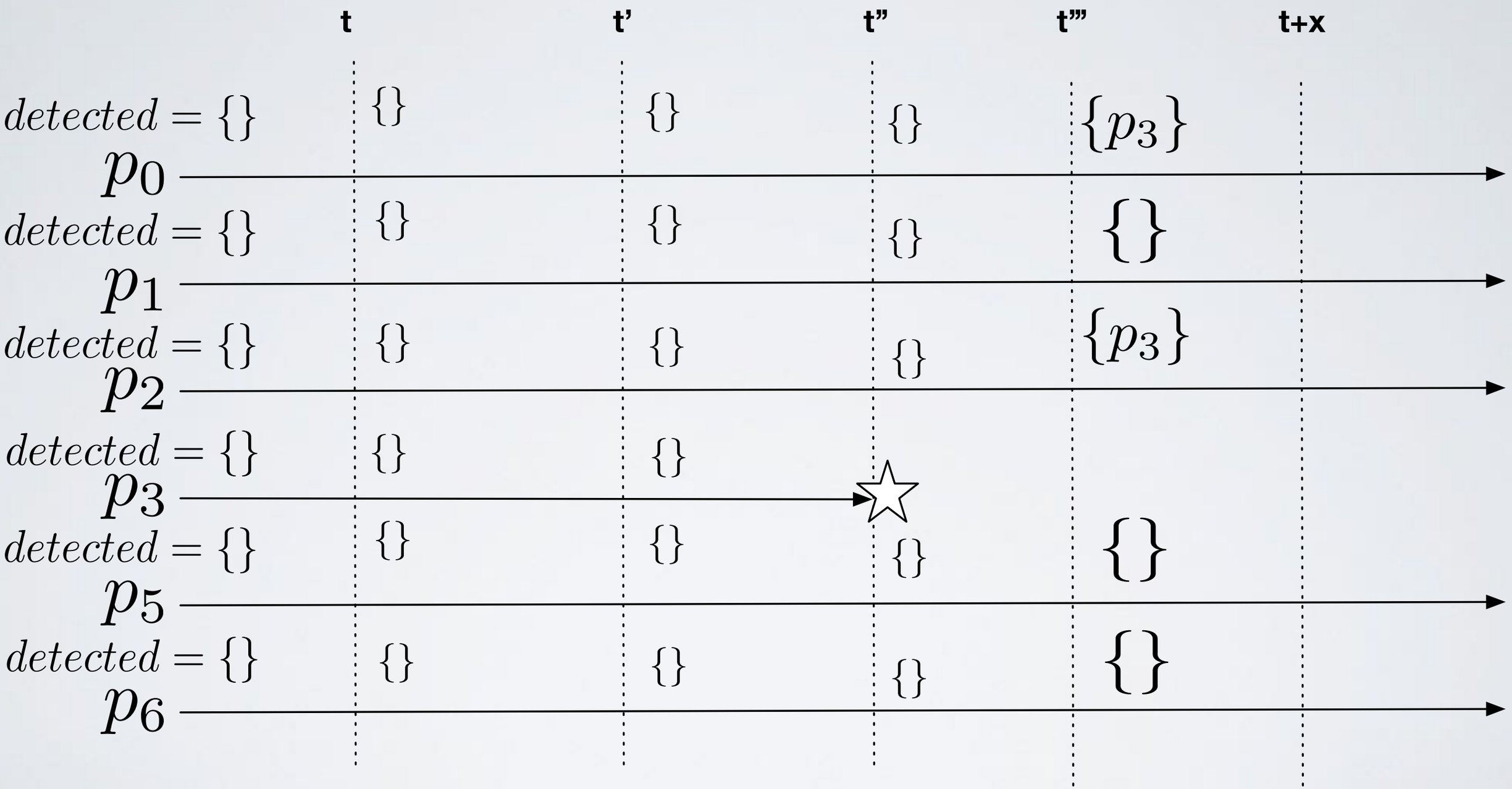
Proof. By contradiction.

If p is detected faulty but it was alive, then either:

- My hearthbeatREQ did not reached it by the timeout.
- Its hearthbeatREPL did not reached me by timeout.

Both contradicts the assumption.

GIVEN P IS IT POSSIBLE?



POSSIBLE? YES

Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:

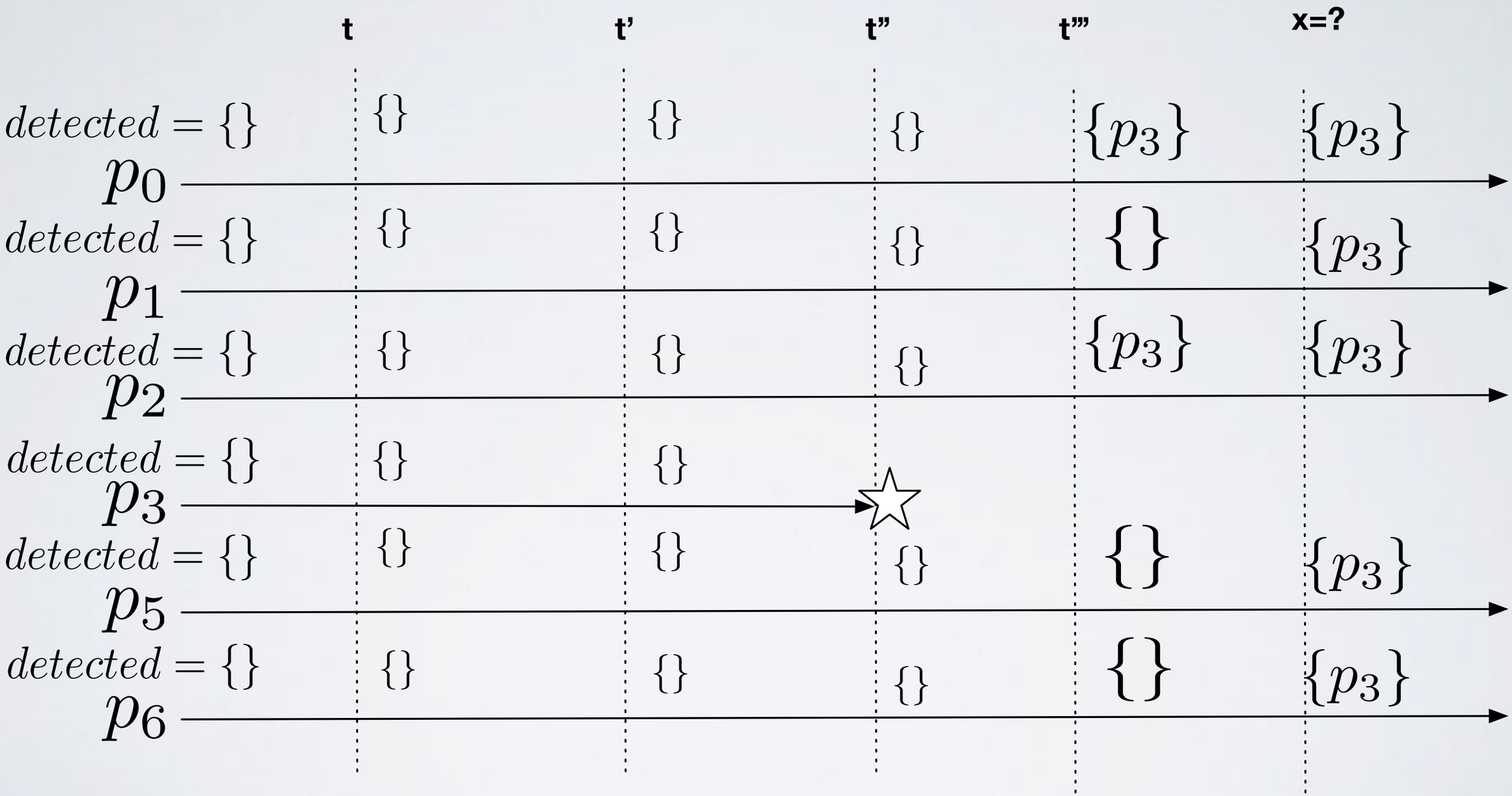
Indication: $\langle\ \mathcal{P}, Crash \mid p\ \rangle$: Detects that process p has crashed.

Properties:

PFD1: *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

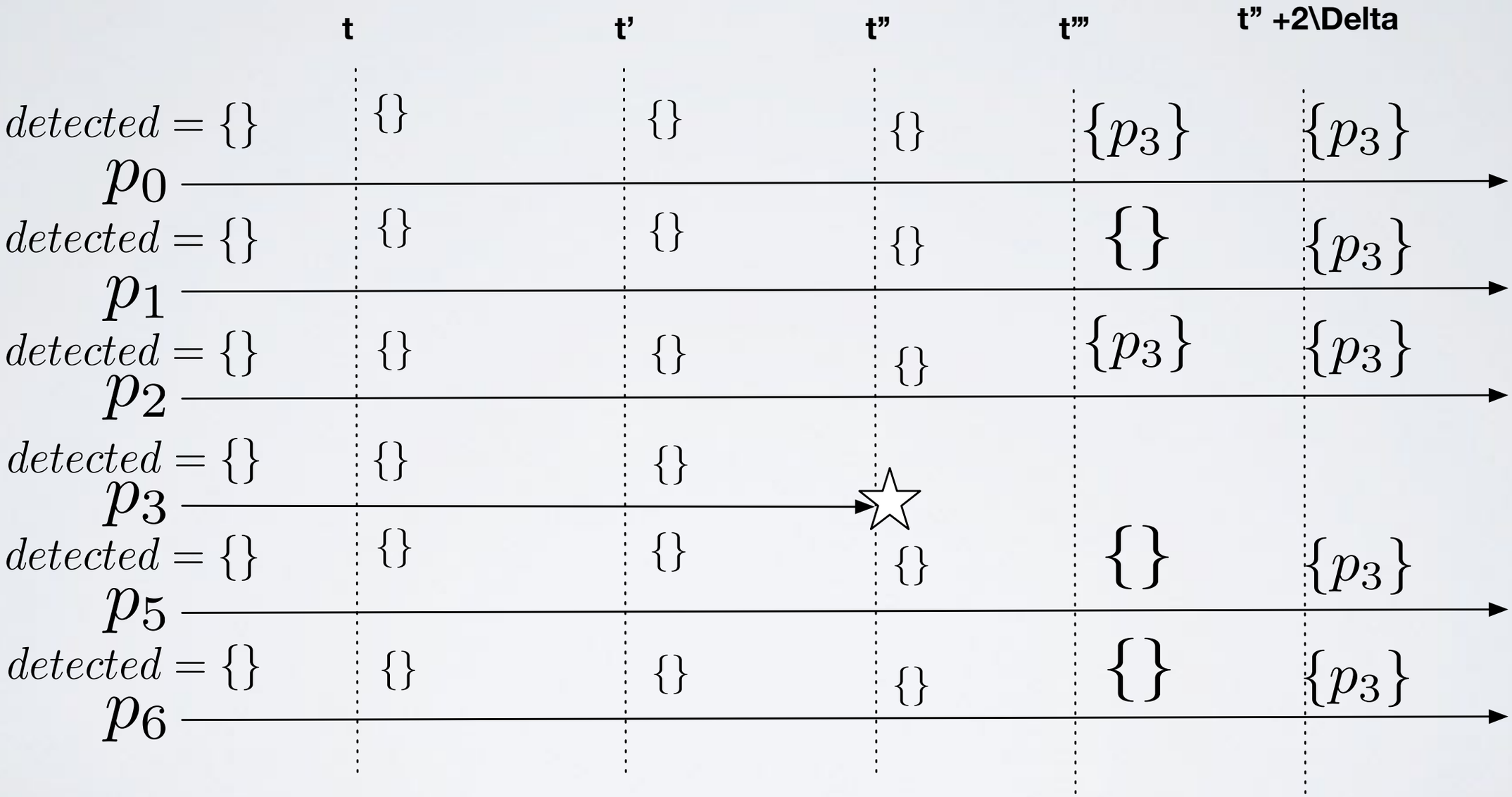
PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

WHEN?


t
t'
t"
t'"
x=?
$detected = \{\}$
p_0
$detected = \{\}$
p_1
$detected = \{\}$
p_2
$detected = \{\}$
p_3
$detected = \{\}$
p_5
$detected = \{\}$
p_6
$\{\}$
$\{p_3\}$

WHEN? IMPLEMENTATION

The exact latency is given by the algorithm, the specification Just says ``eventually”



POSSIBLE?

Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:

Indication: $\langle\ \mathcal{P}, Crash \mid p\ \rangle$: Detects that process p has crashed.

Properties:

PFD1: *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

Consider Exclude on Timeout, if at time t I do not have process p in my detected set can I state that p is alive at time t?

Consider Exclude on Timeout, if at time t I do not have process p in my detected set can I state that p is alive at time t? NO

No Algorithm can say what is happening at exact time t!
You only know the world by messages (delay d) so you only know the world at time $t-d$.

No Failure Detector can say, if someone is alive at a time t

IMPROVING P

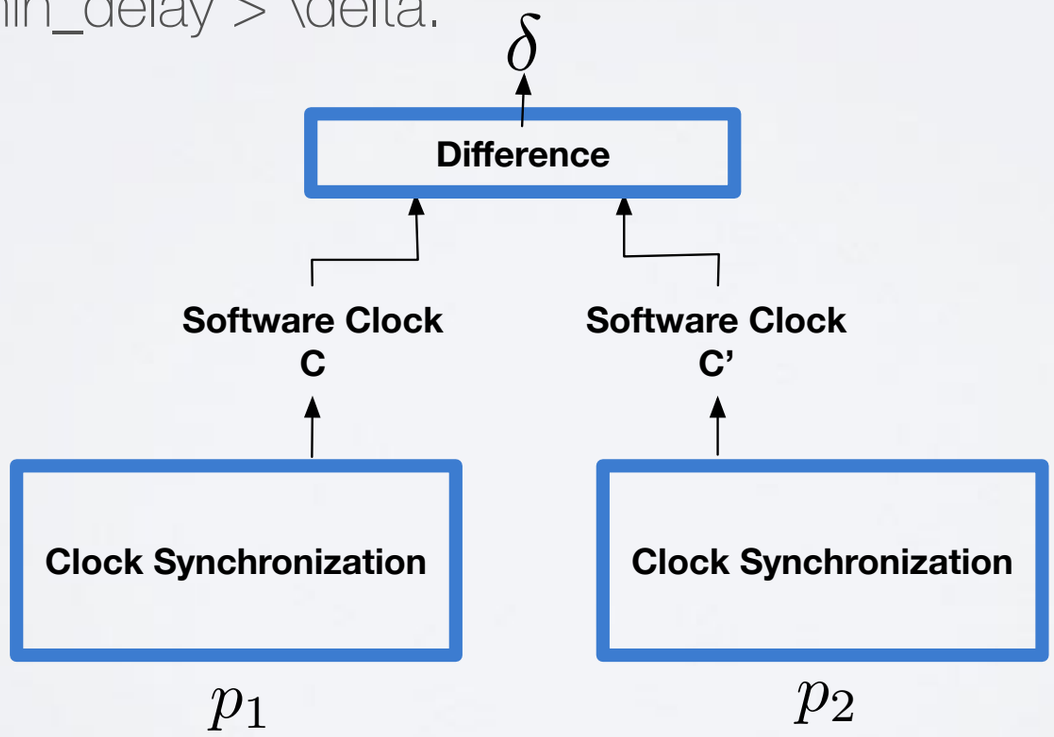
Can we cut the messages by half?

IMPROVING P

Can we cut the messages by half?

``[…] In synchronous networks, …, silences are expressive […]" - N. Santoro - Design and analysis of distributed algorithms, Wiley 2008.

Suppose to have clock synchroniser with global skew \delta, and synchronous link with max delay=max_delay, and min_delay > \delta.



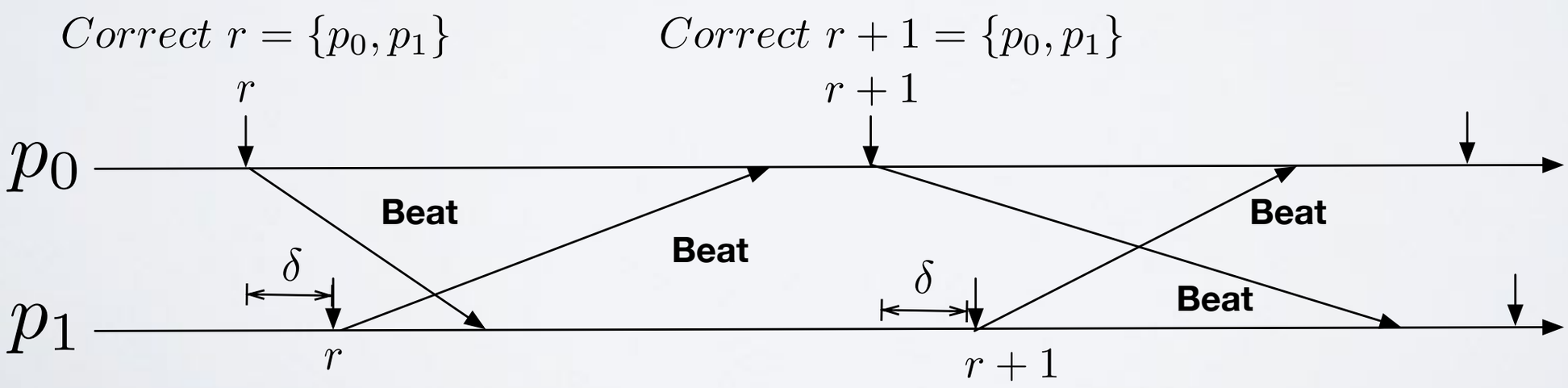
ROUND-BASED P

Algorithm 3 Round-Based, Failure Detector $\mathcal{P}$ on process p_i

1: **upon event** INIT
2: C_v ▷ Synchronizer's clock
3: T ▷ Round length: $T > 2 \cdot (max\_delay + \delta)$
4: $Corrects = \{p_0, p_1, p_2, \ldots, p_n\}$
5: $RoundAlive = Corrects$
6: $r = -1$

7: ▷ Actions to execute at each new round
8: **upon event** $C_v = kT$ FOR SOME $k \in \mathbb{N}^+$
9: $r = r + 1$
10: **for all** $p_j \in Corrects \setminus RoundAlive$ **do**
11: $Corrects = Corrects \setminus \{p_j\}$
12: TRIGGER $\langle Crash | p_j \rangle$
13: $RoundAlive = \{p_i\}$ ▷ Myself
14: **for all** $p_j \in Corrects$ **do**
15: SEND$(< p_i, BEAT >)$

16: **upon event** DELIVERY$(\langle p_j, BEAT \rangle)$
17: $RoundAlive = RoundAlive \cup \{p_j\}$



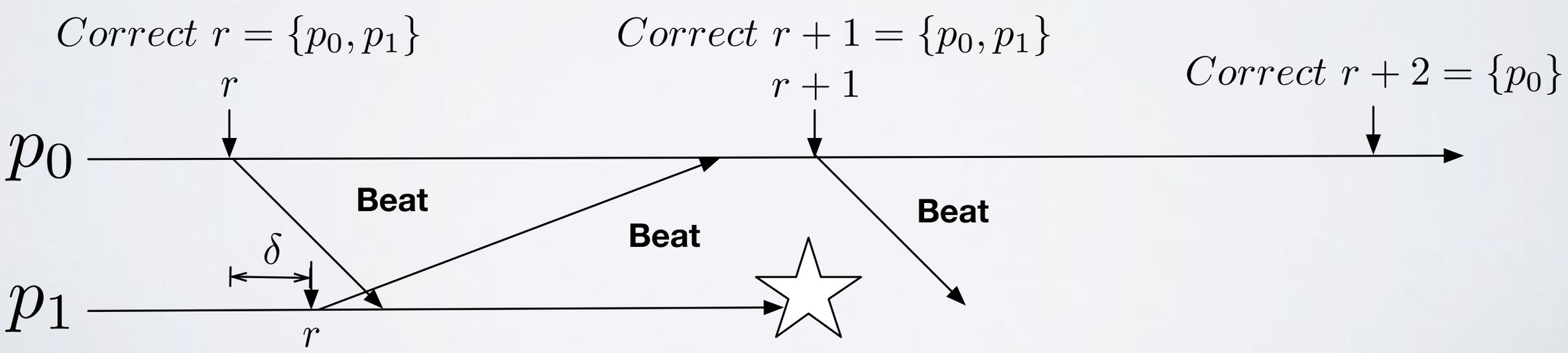
ROUND-BASED P

Algorithm 3 Round-Based, Failure Detector $\mathcal{P}$ on process p_i

1: **upon event** INIT
2: C_v ▷ Synchronizer's clock
3: T ▷ Round length: $T > 2 \cdot (max\_delay + \delta)$
4: $Corrects = \{p_0, p_1, p_2, \ldots, p_n\}$
5: $RoundAlive = Corrects$
6: $r = -1$

7: ▷ Actions to execute at each new round
8: **upon event** $C_v = kT$ FOR SOME $k \in \mathbb{N}^+$
9: $r = r + 1$
10: **for all** $p_j \in Corrects \setminus RoundAlive$ **do**
11: $Corrects = Corrects \setminus \{p_j\}$
12: TRIGGER $\langle Crash | p_j \rangle$
13: $RoundAlive = \{p_i\}$ ▷ Myself
14: **for all** $p_j \in Corrects$ **do**
15: SEND$(< p_i, BEAT >)$

16: **upon event** DELIVERY$(\langle p_j, BEAT \rangle)$
17: $RoundAlive = RoundAlive \cup \{p_j\}$



ROUND-BASED P

FACT 1: If I send a BEAT I am alive.

FACT 2: If I don't send a BEAT, I am dead.

We have to show that: If I send a BEAT to p when my round number shows r, then the beat reaches p when its round number shows r.

By the algorithm, I send my BEAT as soon as my round is r.

By the bounded skew of clock synch. If time t is the first time at which my round number is r, then p will be at round r by at most $t + \delta$

By the round length, p stays in round r at least until

$$t - \delta + T = t - \delta + 2\delta + 2max\_delay > t + \delta + max\_delay$$

By the delay of the link, my BEAT reaches p in the interval $(t + \delta, t + max\_delay]$

ROUND-BASED P

FACT 1: If I send a BEAT I am alive.

FACT 2: If I don't send a BEAT, I am dead.

We have that: If I send a BEAT to p when my round number shows r, then the beat reaches p when its round number shows r.

Strong completeness: If p dies, I will not receive the BEAT and I will signal it.

Strong accuracy: If I do not receive the expected BEAT from p then,
The only possible reason is that p did not send it.
Therefore p is dead.

HOMEWORK

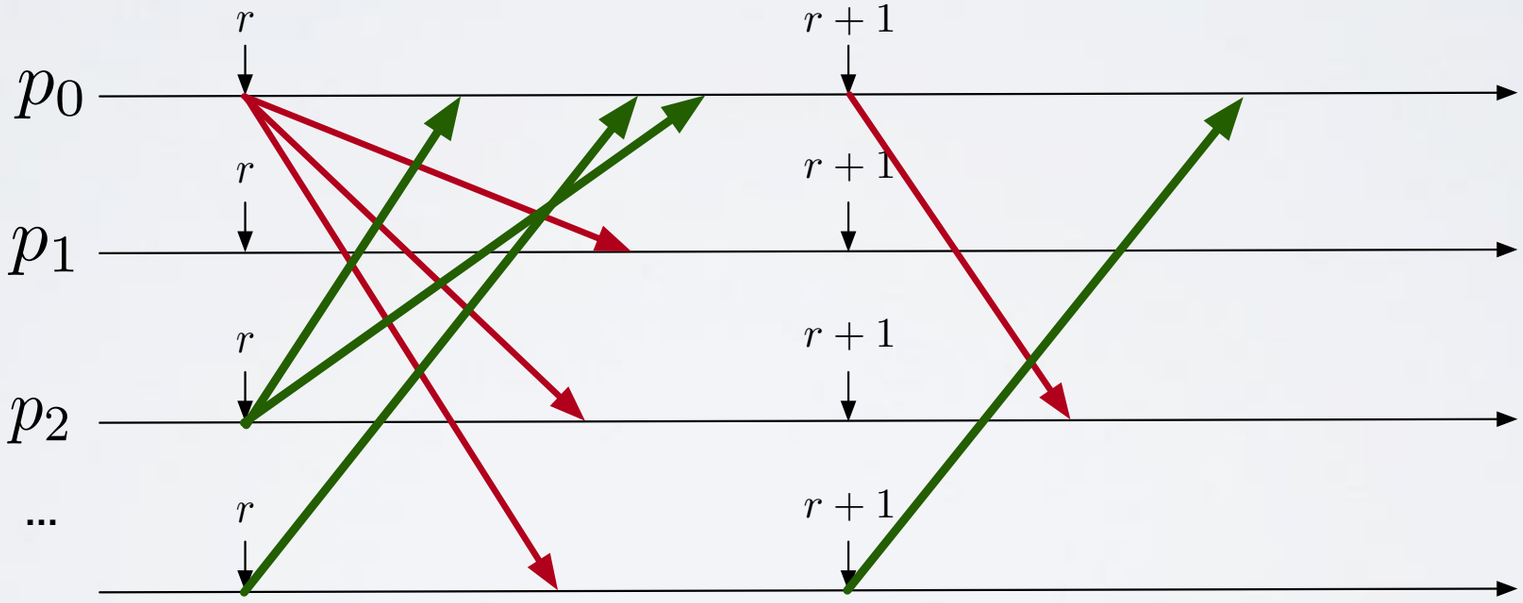
Minimum T for round-based P?

SYNCHRONOUS = ROUND-BASED

An alternative way to see synchronous systems is to imagine time divided in logical slots, the so called **round**.

Synchrony assumptions are abstracted by assuming:

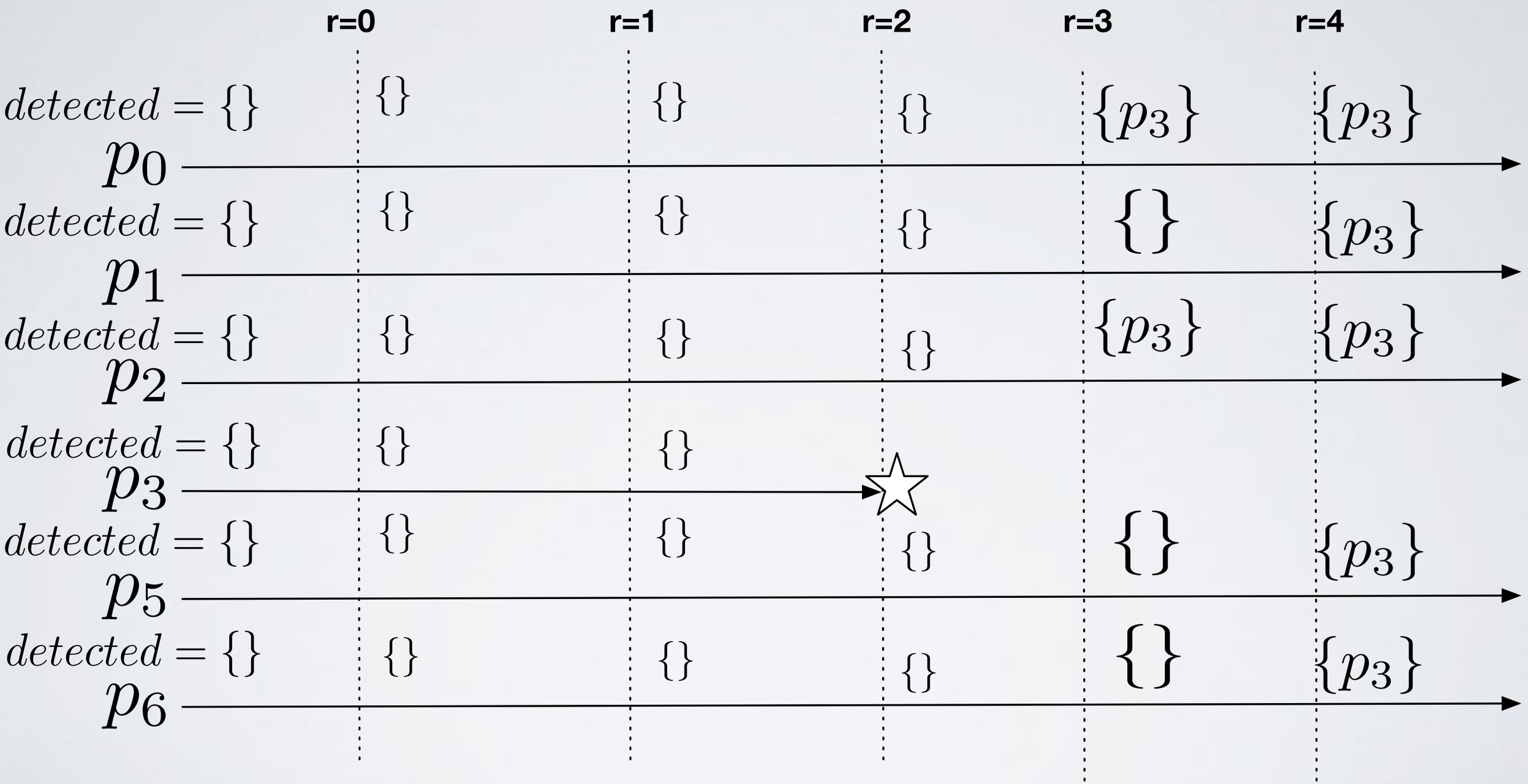
- Processes switch rounds exactly at the same time.
- If I a correct process send a message to a set of processes at the beginning of round r, the messages will reach all correct processes in the set by the end of round r.



Rounds are a powerful tool for the designer of distributed systems!
They are the standard measure of time complexity in synchronous algorithms

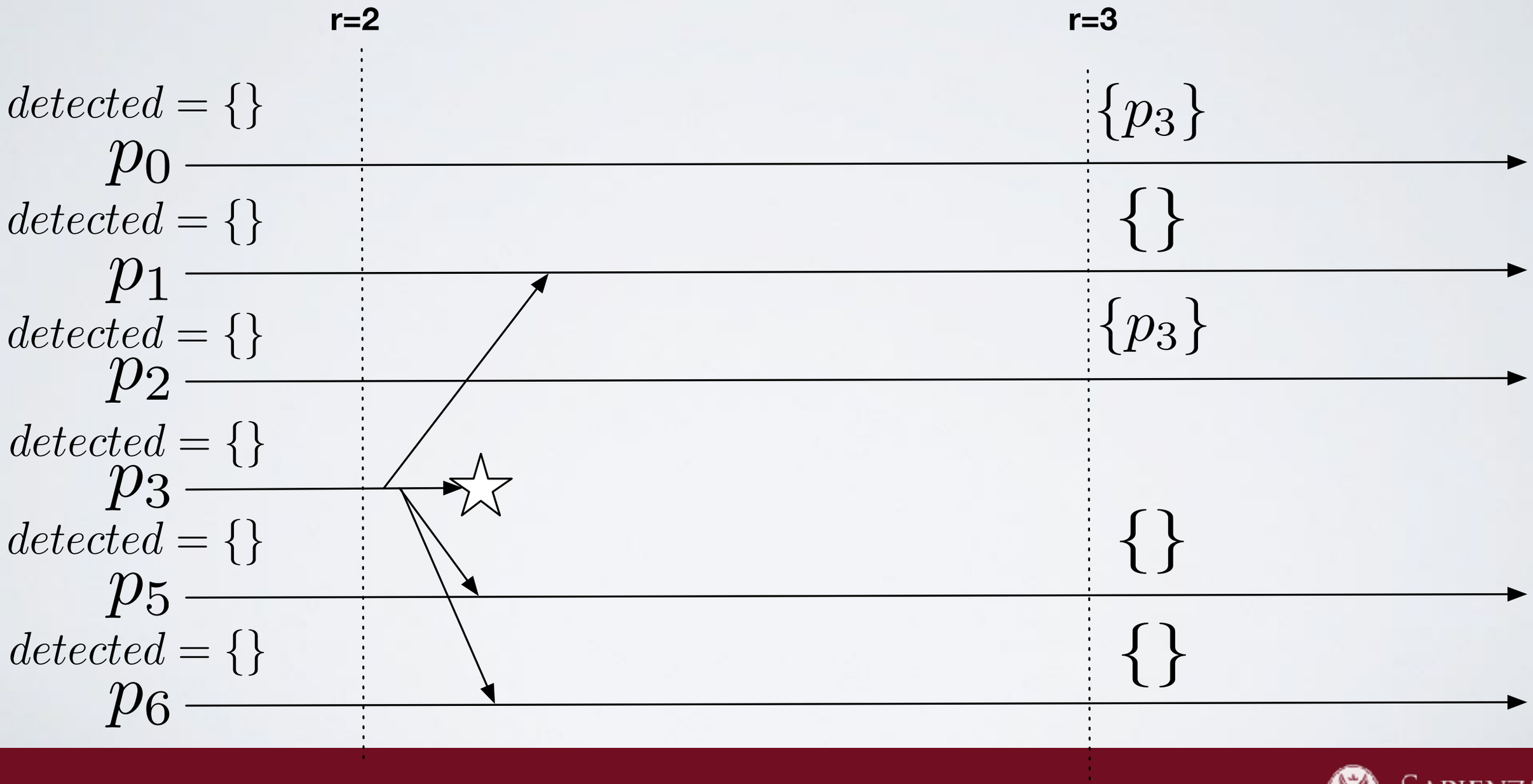
Optional read: The splendors and miseries of rounds, Adam Shimi, Sept. 2019.

ROUND-BASED P, POSSIBLE?



ROUND-BASED P, POSSIBLE?

14: **for all** $p_j \in Corrects$ **do**
15: SEND($< p_i, BEAT >$)



TRADE-OFF

What is the Advantage of decreasing \delta?

Algorithm 2.5: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P}, \mathit{Init}\ \rangle$ **do**
$\mathit{alive} := \Pi$;
$\mathit{detected} := \emptyset$;
starttimer(Δ);

upon event $\langle\ \mathit{Timeout}\ \rangle$ **do**
forall $p \in \Pi$ **do**
if $(p \notin \mathit{alive}) \wedge (p \notin \mathit{detected})$ **then**
$\mathit{detected} := \mathit{detected} \cup \{p\}$;
trigger $\langle\ \mathcal{P}, \mathit{Crash} \mid p\ \rangle$;
trigger $\langle\ pl, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}]\ \rangle$;
$\mathit{alive} := \emptyset$;
starttimer(Δ);

upon event $\langle\ pl, \mathit{Deliver} \mid q, [\text{HEARTBEATREQUEST}]\ \rangle$ **do**
trigger $\langle\ pl, \mathit{Send} \mid q, [\text{HEARTBEATREPLY}]\ \rangle$;

upon event $\langle\ pl, \mathit{Deliver} \mid p, [\text{HEARTBEATREPLY}]\ \rangle$ **do**
$\mathit{alive} := \mathit{alive} \cup \{p\}$;

EVENTUALLY PERFECT FAILURE DETECTORS (◊P)

System model

- Partial synchrony
- Crash failures
- Perfect point-to-point links

There is a (unknown) time *t* after that crashes can be accurately detected

Before *t* the systems behaves as an asynchronous one

- The failure detector makes mistakes in that periods assuming correct processes as crashed.
- The notion of detection becomes suspicion

EVENTUALLY PERFECT FAILURE DETECTORS (◊P)

Module 2.8: Interface and properties of the eventually perfect failure detector

Module:

Name: EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

Events:

Indication: $\langle\ \Diamond\mathcal{P}, \mathit{Suspect} \mid p\ \rangle$: Notifies that process p is suspected to have crashed.

Indication: $\langle\ \Diamond\mathcal{P}, \mathit{Restore} \mid p\ \rangle$: Notifies that process p is not suspected anymore.

Properties:

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

EVENTUALLY PERFECT FAILURE DETECTORS (◊P)

Module 2.8: Interface and properties of the eventually perfect failure detector

Module:

Name: EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

Events:

Indication: $\langle\ \diamond\mathcal{P}, \mathit{Suspect} \mid p\ \rangle$: Notifies that process p is suspected to have crashed.

Indi ... more.

EPFD1 and EPFD2 ensures that the set of suspected Processes stabilises on a correct process

Properti

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

BASIC CONSTRUCTIONS RULES OF AN EVENTUALLY PERFECT FD

- Use timeouts to suspect processes that did not sent expected messages
- A suspect may be wrong. A process *p* may suspect another one *q* because the chosen timeout was too short
- ◊P is ready to change its judgment as soon as it receives a message from *q* (updating also the timeout value)
- If *q* has actually crashed, *p* will not change its judgment anymore.

EVENTUALLY PERFECT FAILURE DETECTORS (◊P)

Algorithm 2.7: Increasing Timeout

Implements:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

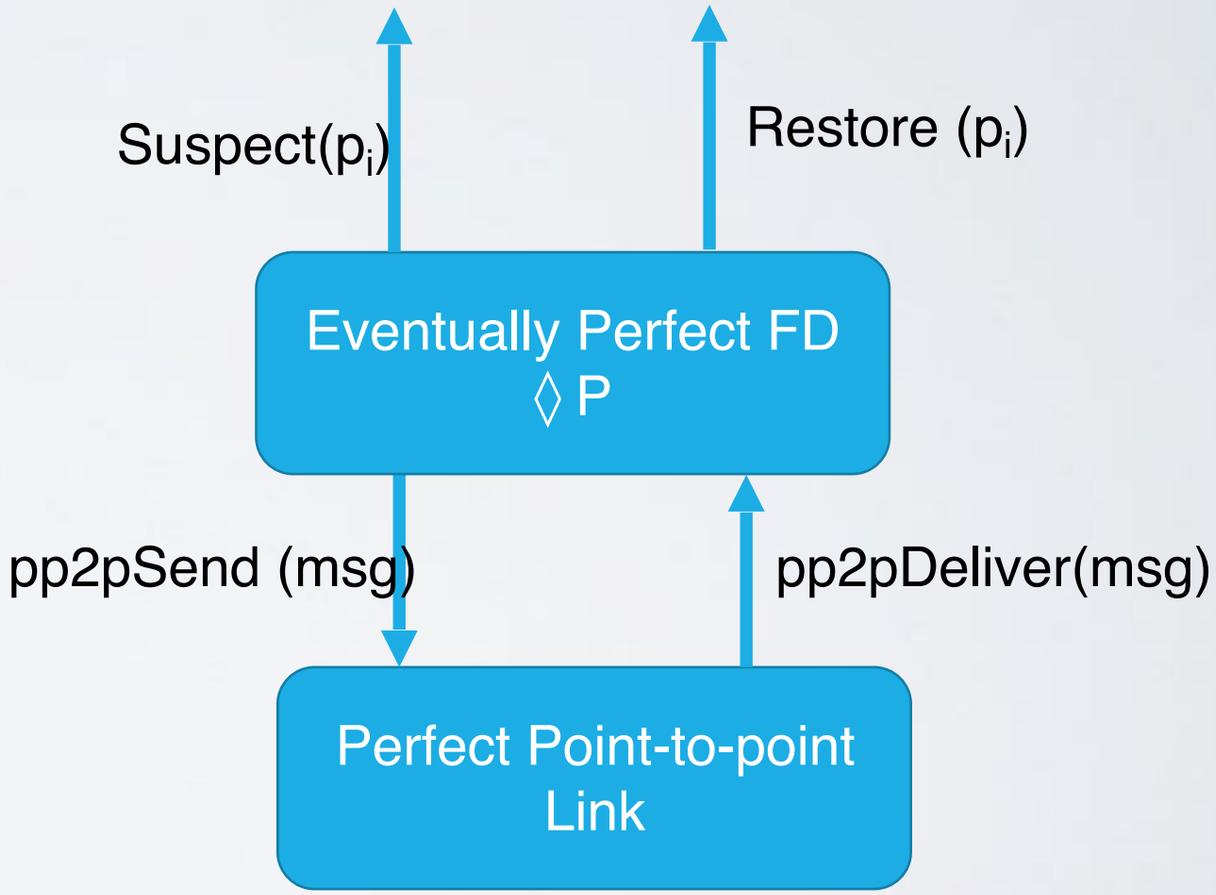
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \diamond\mathcal{P}, \mathit{Init}\ \rangle$ **do**
alive := Π;
suspected := $\emptyset$;
delay := Δ;
starttimer(*delay*);

upon event $\langle\ \mathit{Timeout}\ \rangle$ **do**
if $\mathit{alive} \cap \mathit{suspected} \neq \emptyset$ **then**
$\mathit{delay} := \mathit{delay} + \Delta$;
forall $p \in \Pi$ **do**
if $(p \notin \mathit{alive}) \wedge (p \notin \mathit{suspected})$ **then**
$\mathit{suspected} := \mathit{suspected} \cup \{p\}$;
trigger $\langle\ \diamond\mathcal{P}, \mathit{Suspect} \mid p\ \rangle$;
else if $(p \in \mathit{alive}) \wedge (p \in \mathit{suspected})$ **then**
$\mathit{suspected} := \mathit{suspected} \setminus \{p\}$;
trigger $\langle\ \diamond\mathcal{P}, \mathit{Restore} \mid p\ \rangle$;
trigger $\langle\ \mathit{pl}, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}]\ \rangle$;
alive := $\emptyset$;
starttimer(*delay*);

upon event $\langle\ \mathit{pl}, \mathit{Deliver} \mid q, [\text{HEARTBEATREQUEST}]\ \rangle$ **do**
trigger $\langle\ \mathit{pl}, \mathit{Send} \mid q, [\text{HEARTBEATREPLY}]\ \rangle$;

upon event $\langle\ \mathit{pl}, \mathit{Deliver} \mid p, [\text{HEARTBEATREPLY}]\ \rangle$ **do**
$\mathit{alive} := \mathit{alive} \cup \{p\}$;



CORRECTNESS

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. i.e., 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.

PROPERTY OF SUSPECTED

Lemma: Take two correct processes, p1 and p2 and let suspected1 and suspected2 be the respective sets. There is a time t (stabilization time), after which suspected1=suspected2.

Proof. By contradiction, suppose exists a p in suspected1 but not suspected2.

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

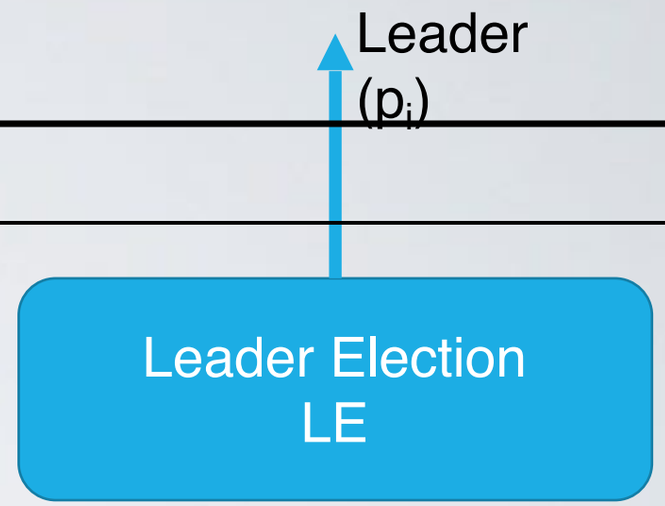
AN ALTERNATIVE

Sometimes, we may be interested in knowing one process that is alive instead of monitoring failures

- E.g., Need of a coordinator

We can use a different oracle (called *leader election* module) that reports a process that is alive

LEADER ELECTION SPECIFICATION



Module 2.7: Interface and properties of leader election

Module:

Name: LeaderElection, **instance** *le*.

Events:

Indication: ⟨ *le*, *Leader* | *p* ⟩: Indicates that process *p* is elected as leader.

Properties:

LE1: *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

LE2: *Accuracy:* If a process is leader, then all previously elected leaders have crashed.

LEADER ELECTION IMPLEMENTATION

Algorithm 2.6: Monarchical Leader Election

Implements:

LeaderElection, **instance** *le*.

Uses:

PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle\ le, Init\ \rangle$ **do**

$suspected := \emptyset$;

$leader := \perp$;

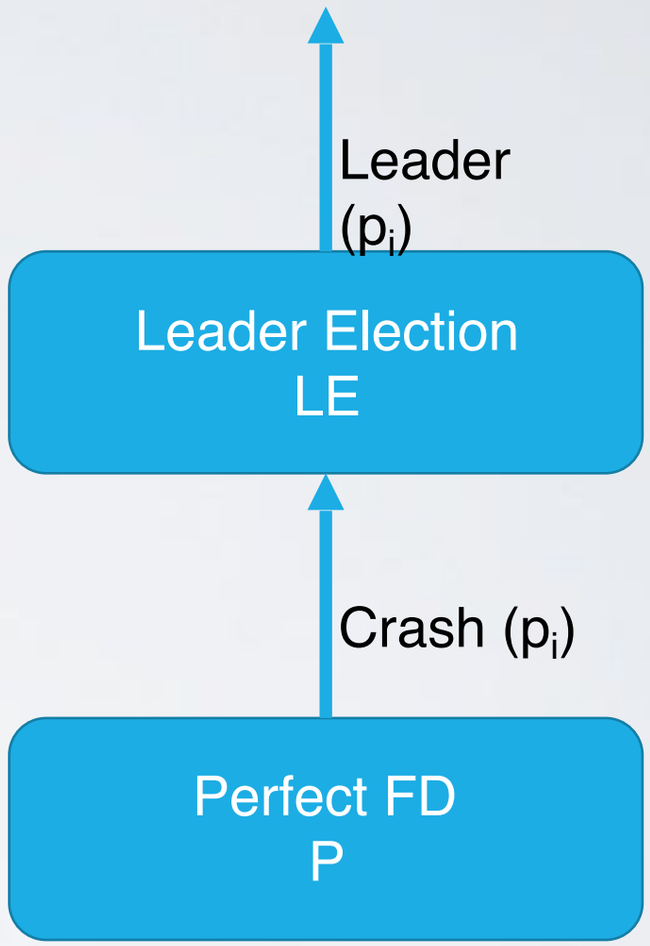
upon event $\langle\ \mathcal{P}, Crash \mid p\ \rangle$ **do**

$suspected := suspected \cup \{p\}$;

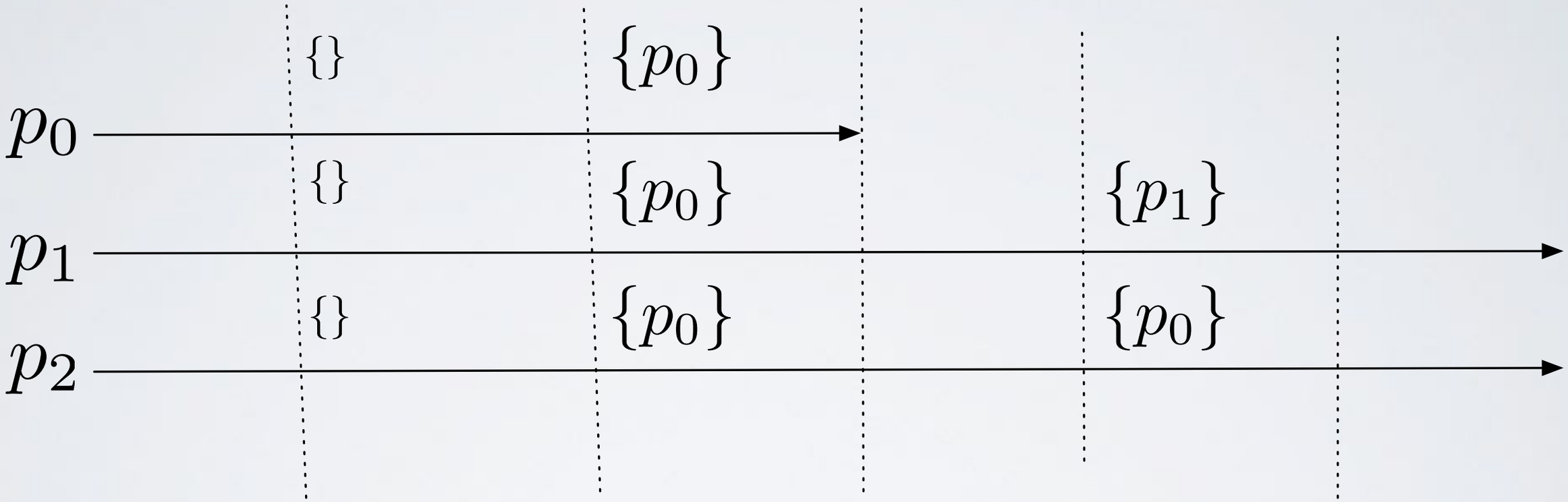
upon $leader \neq maxrank(\Pi \setminus suspected)$ **do**

$leader := maxrank(\Pi \setminus suspected)$;

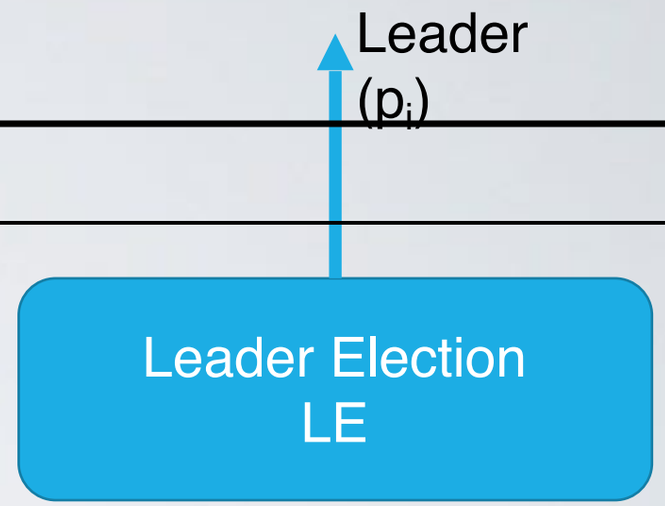
trigger $\langle\ le, Leader \mid leader\ \rangle$;



POSSIBLE?



LEADER ELECTION SPECIFICATION



Module 2.7: Interface and properties of leader election

Module:

Name: LeaderElection, **instance** *le*.

Events:

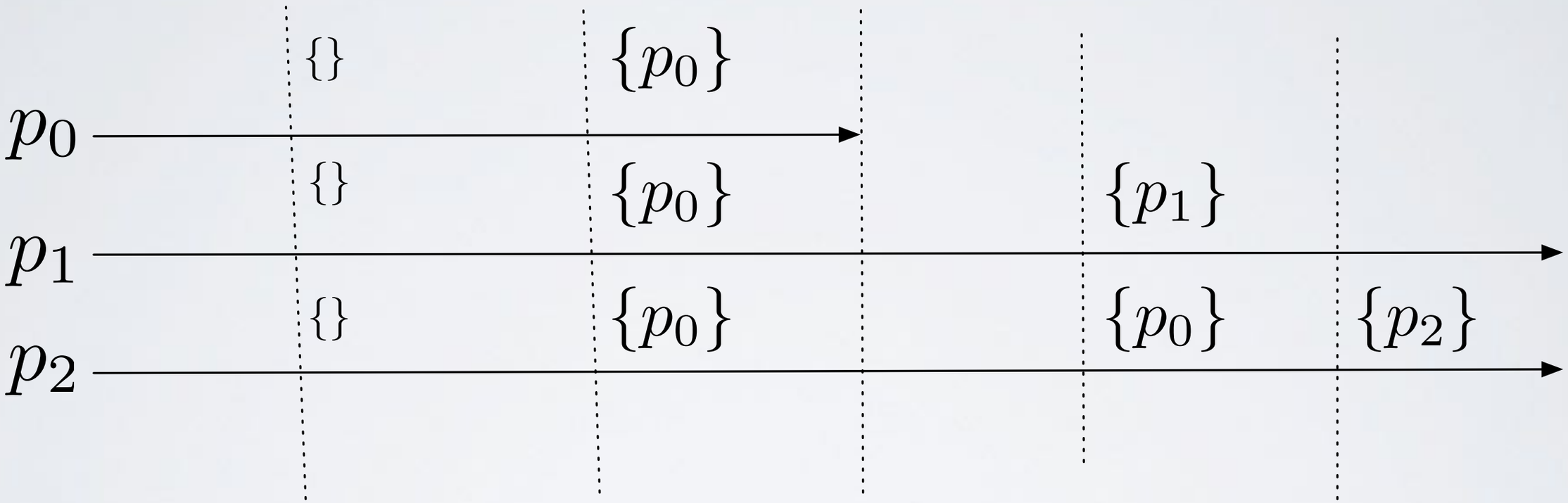
Indication: ⟨ *le*, *Leader* | *p* ⟩: Indicates that process *p* is elected as leader.

Properties:

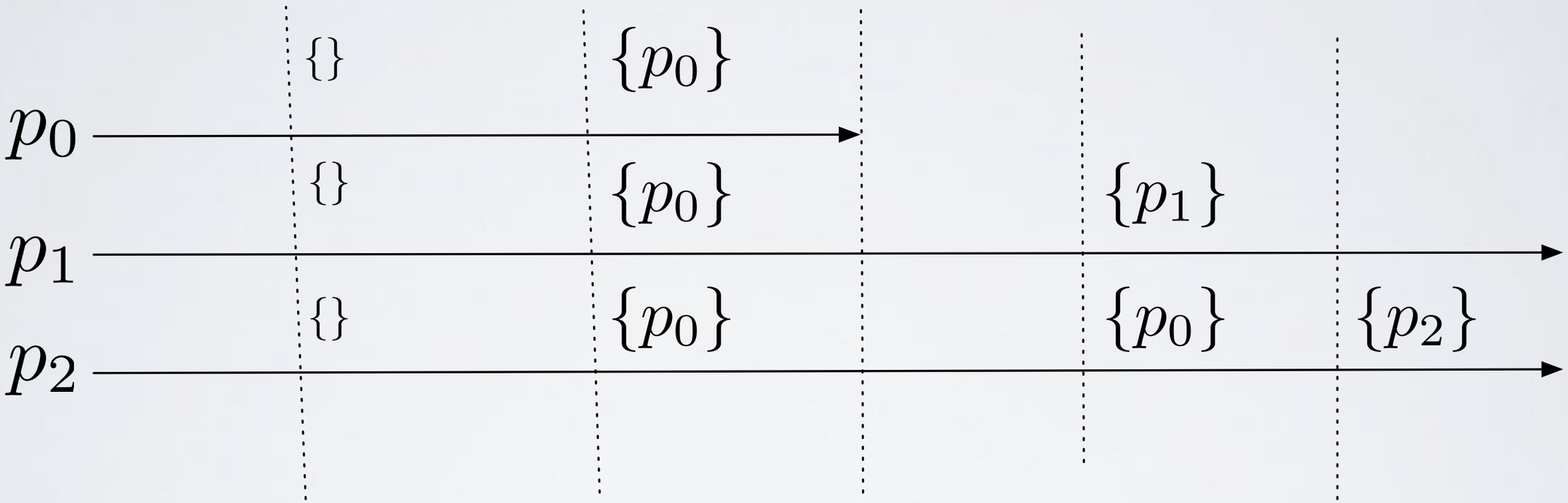
LE1: *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

LE2: *Accuracy:* If a process is leader, then all previously elected leaders have crashed.

POSSIBLE?



POSSIBLE?



No! P1 is alive

CORRECTNESS

Eventual detection: from the strong completeness of P

Accuracy: from the strong accuracy of P and the total order on the ranks (IDs) of processes.

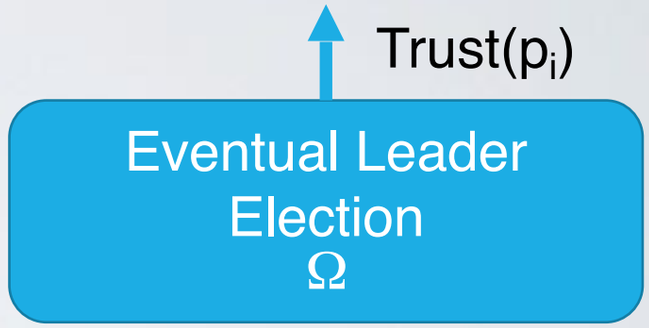
What if the Failure detector is not perfect?

EVENTUAL LEADER ELECTION (Ω)

Module 2.9: Interface and properties of the eventual leader detector

Module:

Name: EventualLeaderDetector, **instance** Ω.



Events:

Indication: $\langle\ \Omega,\ Trust\ |\ p\ \rangle$: 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 is a time after which no two correct processes trust different correct processes.

OBSERVATION ON Ω

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

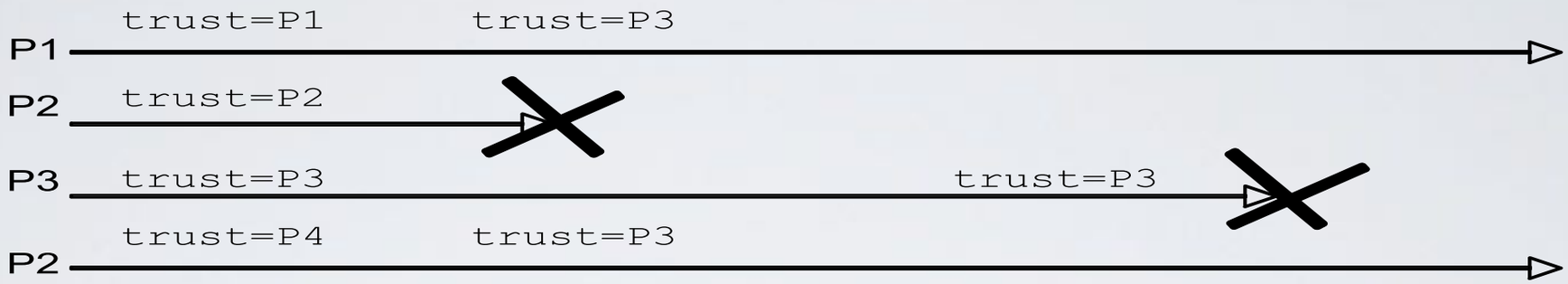
With Ω we can have that:

- Leaders change in an arbitrary manner and for an arbitrary period of time
- many leaders might be elected during the same period of time without having crashed

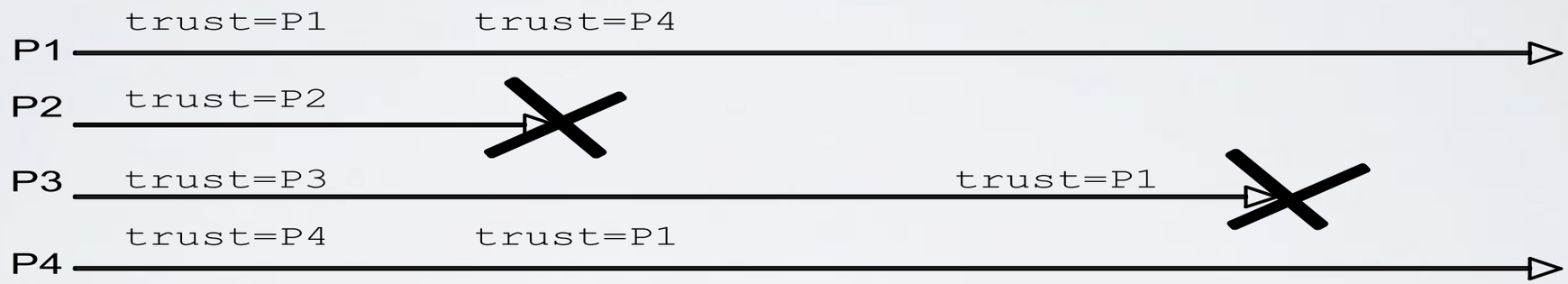
Once a unique leader is determined, and does not change again, we say that the leader has ***stabilized***

STUDY OF PROPERTIES

Run 1



Run 2



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

EVENTUAL LEADER ELECTION (Ω)

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 $\Diamond P$

Ω IMPLEMENTATION

Algorithm 2.8: Monarchical Eventual Leader Detection

Implements:

EventualLeaderDetector, **instance** Ω.

Uses:

EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

upon event $\langle\ \Omega, Init\ \rangle$ **do**

$suspected := \emptyset$;

$leader := \perp$;

upon event $\langle\ \Diamond\mathcal{P}, Suspect \mid p\ \rangle$ **do**

$suspected := suspected \cup \{p\}$;

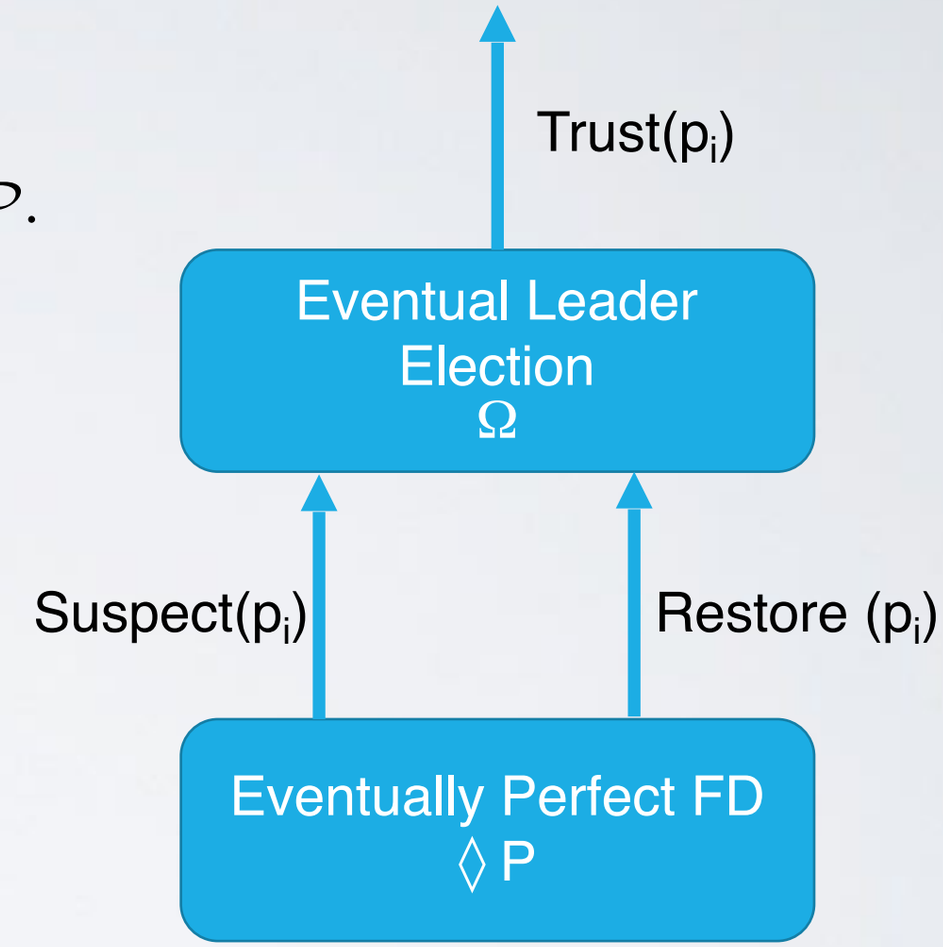
upon event $\langle\ \Diamond\mathcal{P}, Restore \mid p\ \rangle$ **do**

$suspected := suspected \setminus \{p\}$;

upon $leader \neq maxrank(\Pi \setminus suspected)$ **do**

$leader := maxrank(\Pi \setminus suspected)$;

trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;



PROOF OF Ω

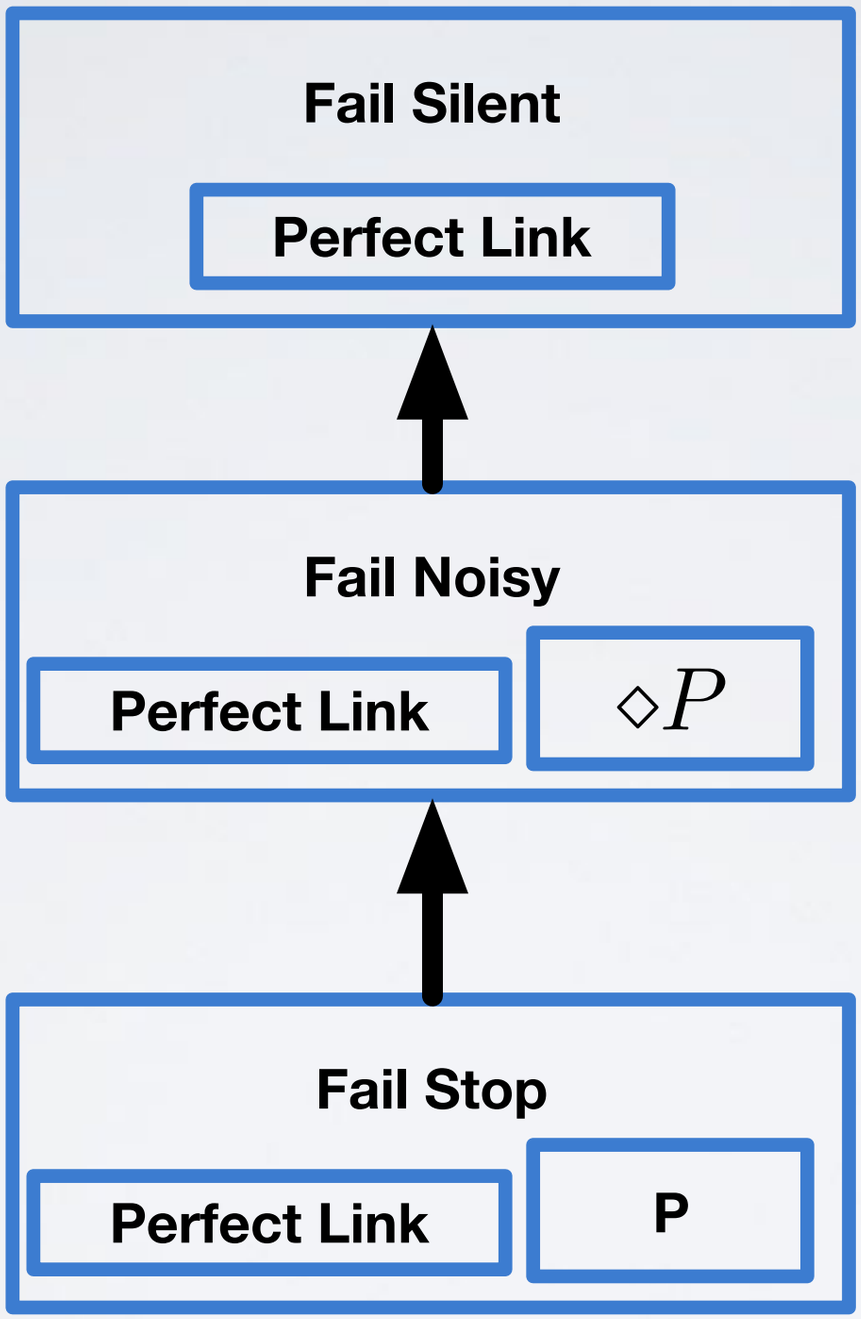
Proof.

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

Stronger

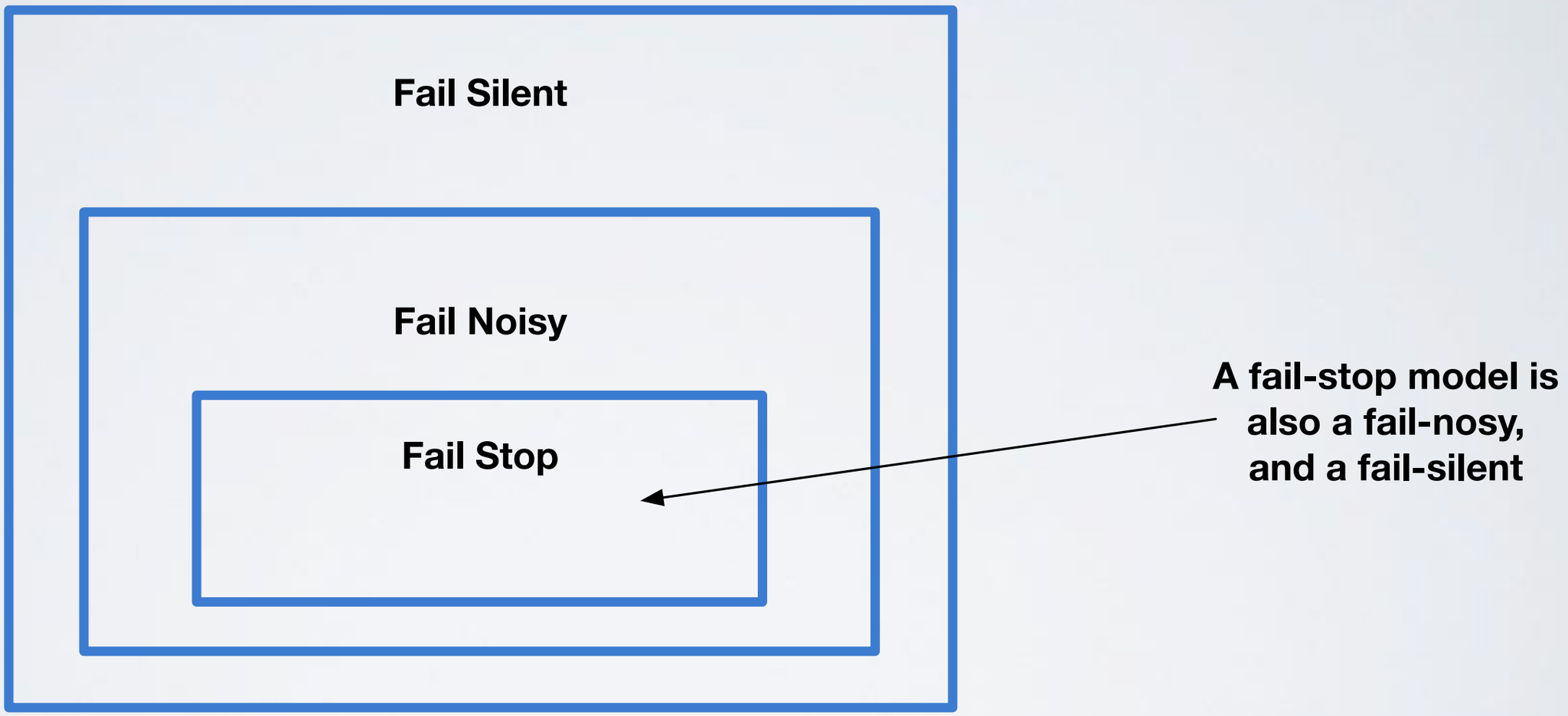


Fail-noisy implies Fail-silent

Fail-stop implies Fail-noisy

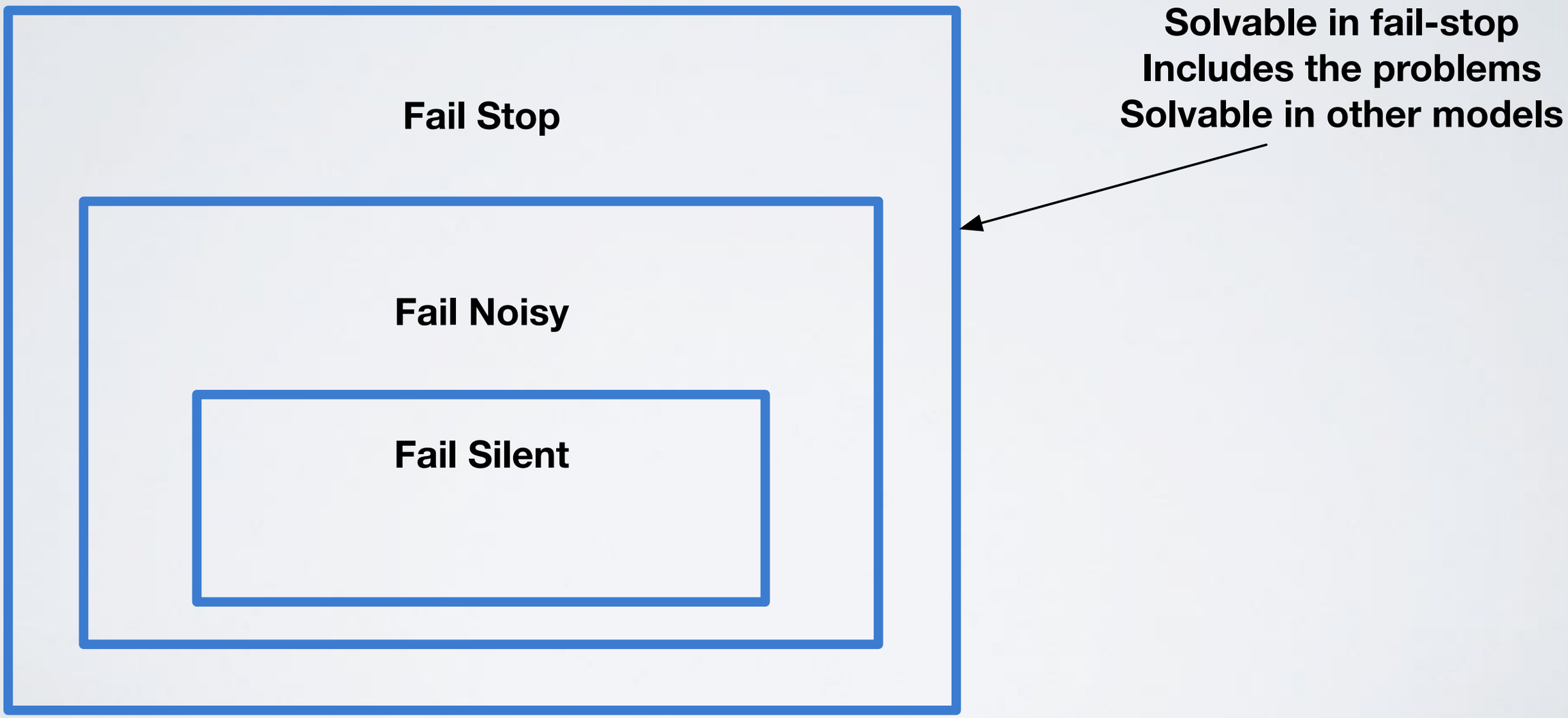
THREE MODELS

Inclusion between models



THREE MODELS

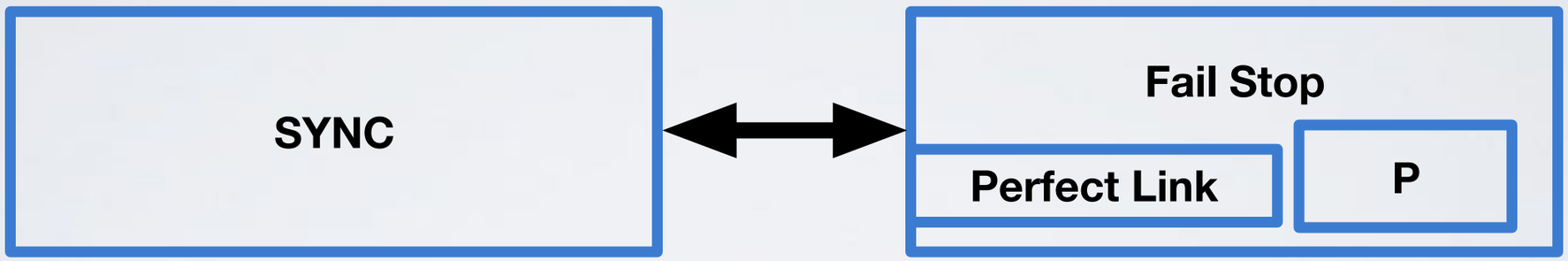
Set of solvable problems



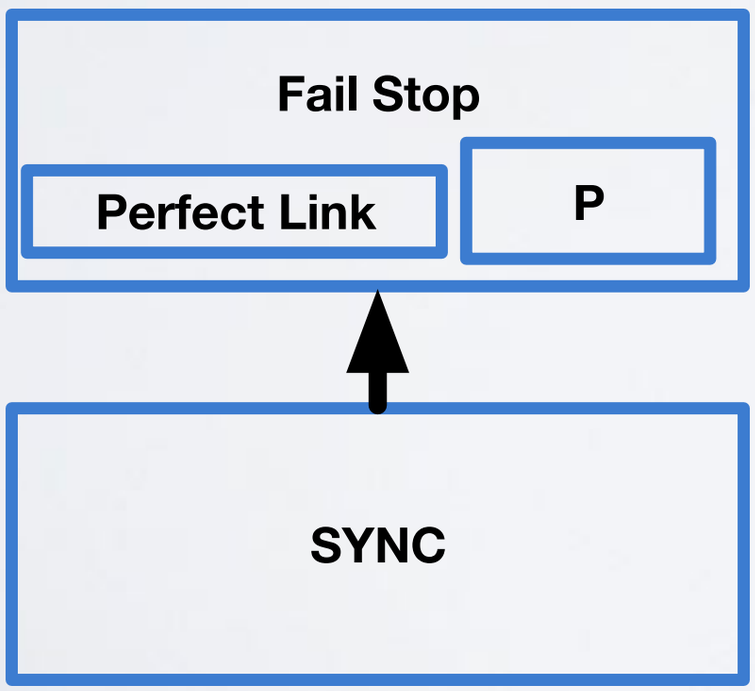
FAIL-STOP VS SYNC

(A), (B) or (C)?

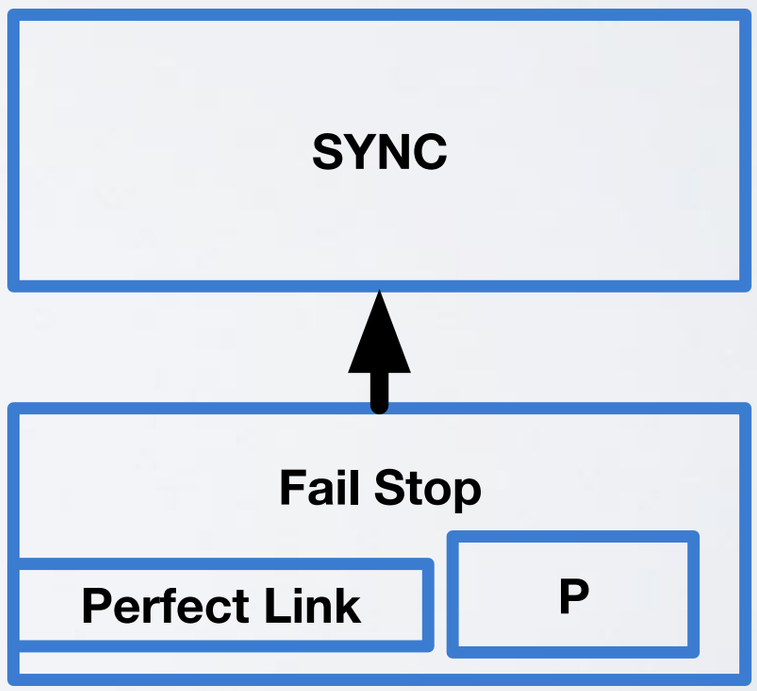
(A)



(B)

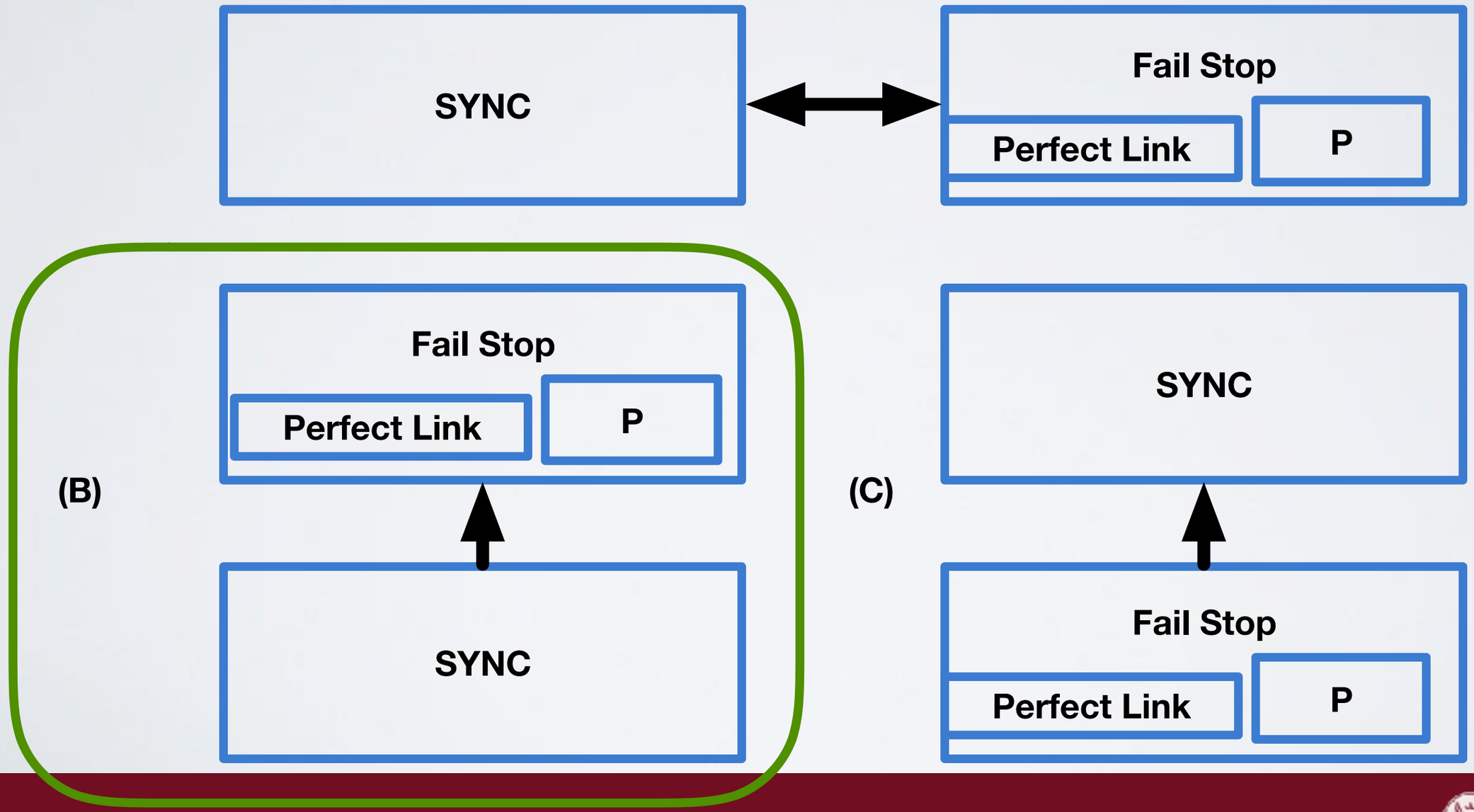


(C)



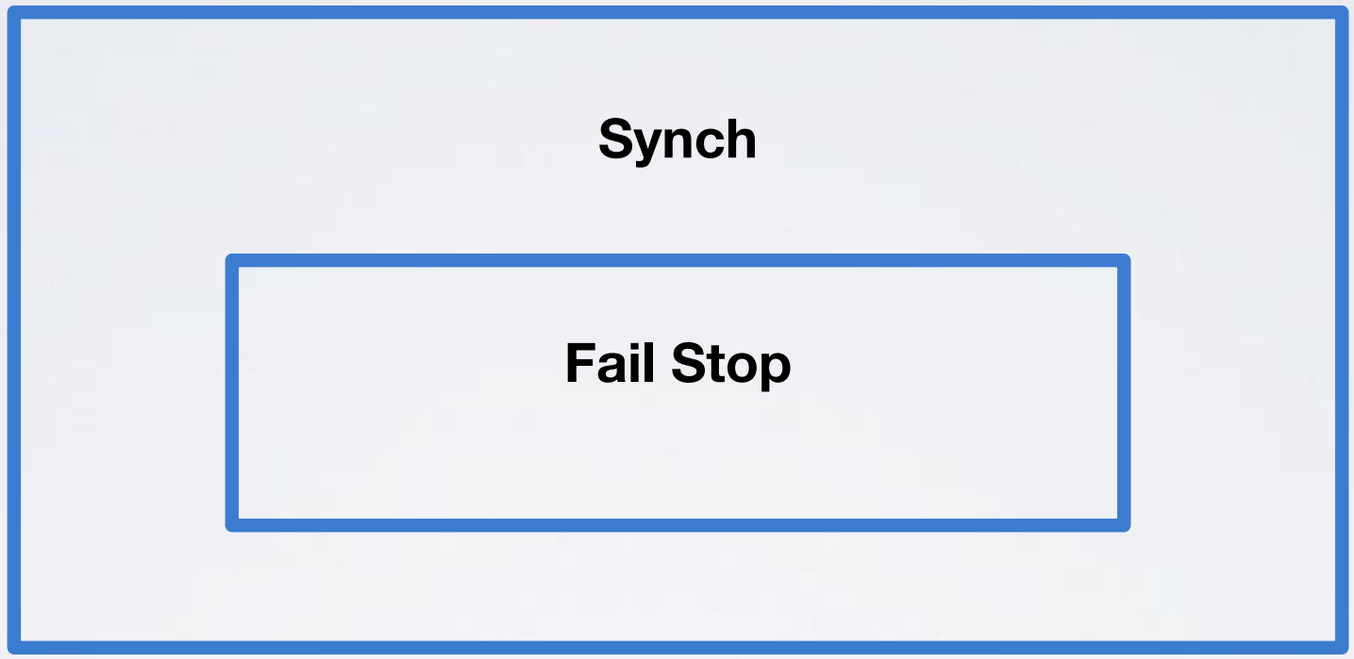
FAIL-STOP VS SYNC

(A), (B) or (C)? (B)!



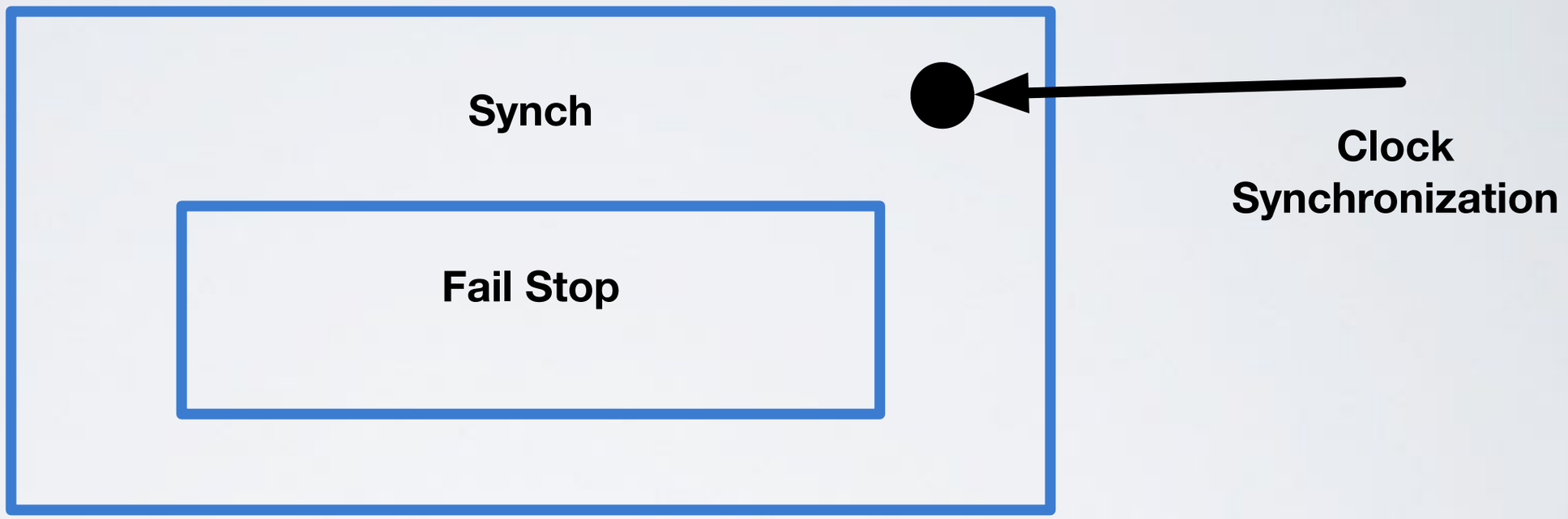
SYNC IS STRONGER THAN FAIL-STOP

Set of solvable problems



SYNC IS STRONGER THAN FAIL-STOP

Set of solvable problems



APPLICATIONS OF FD AND LE

Objective: using P to make Lamport's ME fault tolerant

Fault-tolerant Mutual Exclusion

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.

Algorithm 1 Lamport's ME Algorithm on process p_i - MSGS are REQ, ACK, RLS

ORIGINAL ALGORITHM

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

6: ▷ Request access to CS from upper layer
7: **upon event** REQUEST
8: $scalar\_clock = scalar\_clock + 1$
9: $my\_req = (REQ, ts =< i, scalar\_clock >)$
10: **for all** $p_j \in \Pi$ **do**
11: SEND FIFOPERFECTLINK(p_j, req_msg) ▷ Send a REQ containing my ID (i) and ts (scalar_clock) to all $p \in \Pi$

12: ▷ Release CS from upper layer
13: **upon event** RELEASE
14: $Requests = Requests \setminus \{req\_msg\}$
15: $scalar\_clock = scalar\_clock + 1$
16: **for all** $p_j \in \Pi$ **do**
17: SEND FIFOPERFECTLINK$(p_j, (RLS, ts =< i, scalar_clock >))$

18: ▷ $ts(x) < ts(y)$ 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
19: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in \Pi : \exists m \in Acks | ts(m) > ts(my\_req)$
20: **trigger event** GRANTED

21: **upon event** DELIVER MESSAGE(m)
22: $scalar\_clock = max(clock(m), scalar\_clock) + 1$
23: **if** m is a REQ **then**
24: $Request\_set = Request\_set \cup \{m\}$
25: $scalar\_clock = scalar\_clock + 1$
26: SEND FIFOPERFECTLINK$(sender(m), (ACK, ts =< i, scalar_clock >))$
27: **else if** m is a ACK **then**
28: $Acks = Acks \cup \{m\}$
29: **else if** m is a RLS $\wedge \exists req \in Request\_set$: sender(req)=sender(m) **then**
30: $Requests = Requests \setminus \{req\}$

Algorithm 1 Lamport's ME Algorithm on process p_i - MSGS are REQ, ACK, RLS

ORIGINAL ALGORITHM

Critical points:

1) if you crashed you cannot release

2) If you crashed you cannot ack

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

6: ▷ Request access to CS from upper layer
7: **upon event** REQUEST
8: $scalar\_clock = scalar\_clock + 1$
9: $my\_req = (REQ, ts =< i, scalar\_clock >)$
10: **for all** $p_j \in \Pi$ **do**
11: SEND FIFOPERFECTLINK(p_j, req_msg) ▷ Send a REQ containing my ID (i) and ts (scalar_clock) to all $p \in \Pi$

12: ▷ Release CS from upper layer
13: **upon event** RELEASE
14: $Requests = Requests \setminus \{req\_msg\}$
15: $scalar\_clock = scalar\_clock + 1$
16: **for all** $p_j \in \Pi$ **do**
17: SEND FIFOPERFECTLINK$(p_j, (RLS, ts =< i, scalar_clock >))$

18: ▷ $ts(x) < ts(y)$ 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
19: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in \Pi : \exists m \in Acks | ts(m) > ts(my\_req)$
20: **trigger event** GRANTED

21: **upon event** DELIVER MESSAGE(m)
22: $scalar\_clock = max(clock(m), scalar\_clock) + 1$
23: **if** m is a REQ **then**
24: $Request\_set = Request\_set \cup \{m\}$
25: $scalar\_clock = scalar\_clock + 1$
26: SEND FIFOPERFECTLINK$(sender(m), (ACK, ts =< i, scalar_clock >))$
27: **else if** m is a ACK **then**
28: $Acks = Acks \cup \{m\}$
29: **else if** m is a RLS $\wedge \exists req \in Request\_set$: sender(req)=sender(m) **then**
30: $Requests = Requests \setminus \{req\}$

PATCH

Algorithm 15 Lamport's ME Algorithm on process p_i - Fault-Tolerant Patch

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

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

12: ▷ $ts(x) < ts(y)$ 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
13: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
14: **trigger event** GRANTED

PATCH - UNCORRECT

Algorithm 15 Lamport's ME Algorithm on process p_i - Fault-Tolerant Patch

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

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

12: ▷ $ts(x) < ts(y)$ 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
13: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
14: **trigger event** GRANTED

It is enough?

PATCH

Algorithm 15 Lamport's ME Algorithm on process p_i - Fault-Tolerant Patch

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 $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
    trigger event GRANTED

What happens if you receive a req from pj after the crash detection?

PATCH

Algorithm 15 Lamport's ME Algorithm on process p_i - Fault-Tolerant Patch

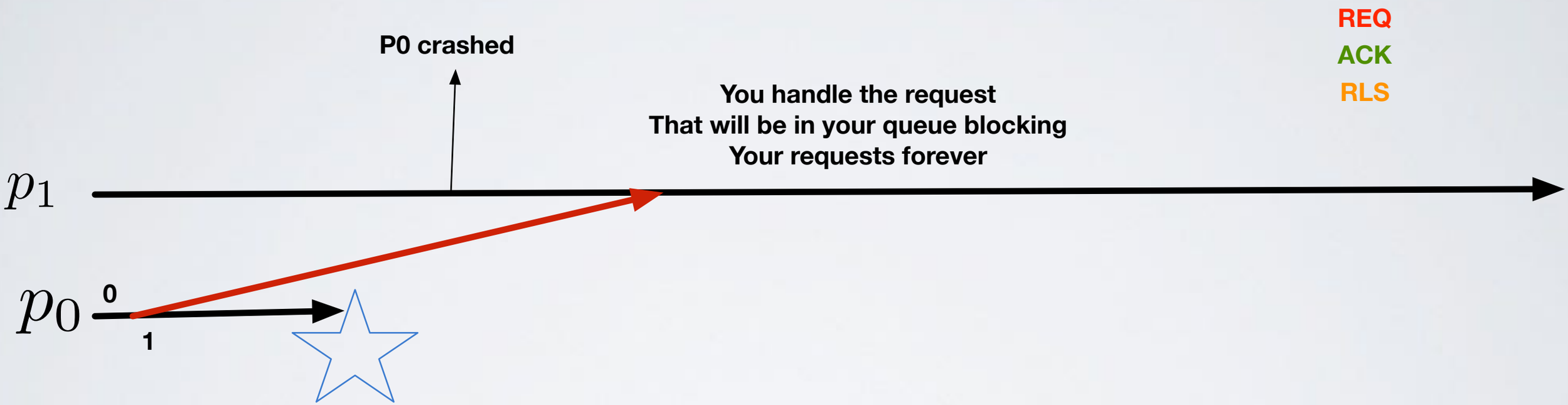
upon event INIT
    $Requests = Acks = \emptyset$
    $scalar\_clock = 0$
    $my\_req = \perp$
    $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$ ▷ Set of all processes
    $Crashed = \{\}$

▷ 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 $\nexists req \in Requests : ts(req) < ts(my\_req) \wedge \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
    trigger event GRANTED

What happens if you receive a req from pj after the crash detection?

PATCH


REQ
ACK
RLS
P0 crashed
You handle the request
That will be in your queue blocking
Your requests forever
p_1
p_0
0
1

PATCH - CORRECT

Algorithm 15 Lamport's ME Algorithm on process p_i - Fault-Tolerant Patch

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

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

12: ▷ $ts(x) < ts(y)$ 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
13: **upon event** $\nexists req \in Requests : ts(req) < ts(my\_req) \land \forall p \in (\Pi \setminus Crashed) : \exists m \in Acks | ts(m) > ts(my\_req)$
14: **trigger event** GRANTED

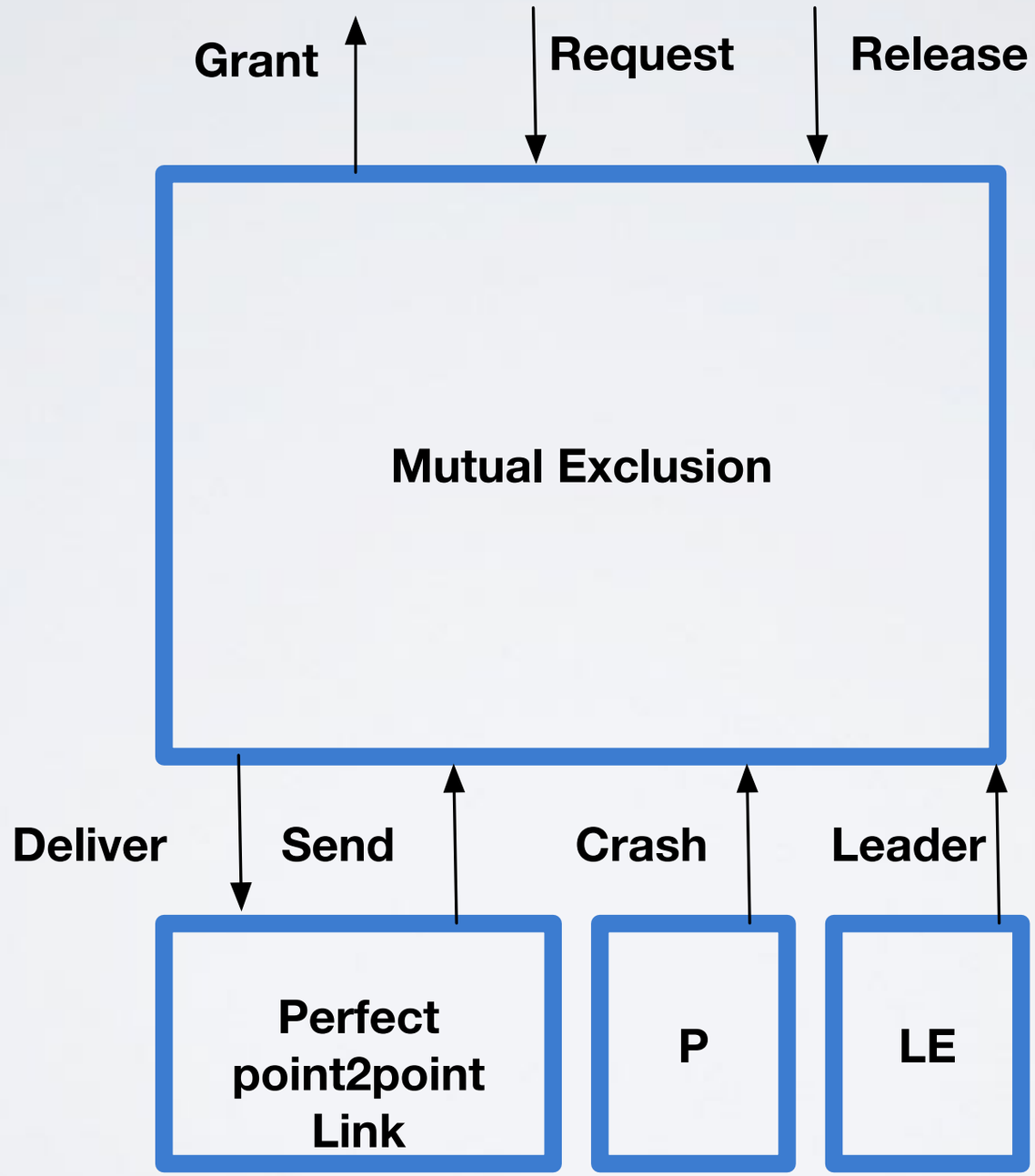
15: **upon event** DELIVER MESSAGE(m)
16: **if** m is a REQ and the sender of m is not in $Crashed$ **then**
17: $scalar\_clock = max(clock(m), scalar\_clock) + 1$
18: $Request\_set = Request\_set \cup \{m\}$
19: $scalar\_clock = scalar\_clock + 1$
20: SEND FIFOPERFECTLINK$(sender(m), (ACK, ts =< i, scalar_clock >))$
21: **else if** m is a ACK **then**
22: ...

ANOTHER ME ALGORITHM

Objective: A simpler algorithm using LE and FD P.

USING FD AND LE

Fault-tolerant Mutual Exclusion



USING FD AND LE

IDEAS:

- 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,

USING FD AND LE

- Problem: What to do when a new leader is elected? The old leader was the only one to know who was in CS,
  - If the process that was in CS crashed, is fine.
  - If it is alive, then it will answer to a message.
- The new leader starts an new_leader phase:
  - It interrogates any process that has not been detected as faulty to know if is in CS, and it updates its information accordingly.
- After the new_leader phase, the new elected leader satisfies new requests as explained before.
- When a new leader is elected the processes sends the old requests (if not satisfied) to the new leader.

HOMEWORKS

- Write code and proofs of the Fault Tolerant ME algorithm explained before
- Main Book pg 67 ex 2.4
- Main Book pg 67 ex 2.6
- Main Book pg 67 ex 2.7
- Main Book pg 67 ex 2.9
- Main Book pg 68 ex 2.10
- *Suppose to be in a synchronous system with an unknown maximum channel delay. You want to create an eventually perfect P. What is a good strategy to increase, and decrease the internal timeout in such a way to:
  - 1) Stabilise to perfect P in the fastest way.
  - 2) Have a minimum latency when suspecting an actual crash.

HOMEWORKS

- Describe two problems that can be solved in sync but not in fail-stop:
  - ?
  - ?
- Describe one problem that can be solved in fail-stop but not in sync:
  - ?
-

HOMEWORKS

- Describe two problems that can be solved in sync but not in fail-stop:
  - Clock-Synchronisation
  - Timed-delivery (if p send m to q, then q receives m after at most x)
- Describe one problem that can be solved in fail-stop but not in sync:
  - It does not exits. Problems(Fail-stop) \subset Problems(Sync)

HOMEWORKS

- Do you think that is possible to solve fault-tolerant mutual exclusion in a fail-silent system? (Write your considerations, they do not have to be a formal proof)
- Do you think that is possible to solve fault-tolerant mutual exclusion in a fail-noisy system? (Write your considerations, they do not have to be a formal proof)

HOMEWORKS

Exercise 2.8: *Suppose an algorithm A implements a distributed programming abstraction M using a failure detector $\mathcal{D}$ that is assumed to be eventually perfect. Can A violate a safety property of M if $\mathcal{D}$ is not eventually perfect, for example, when $\mathcal{D}$ permanently outputs the empty set?*