Distributed Systems
Master of Science in Engineering in Computer Science

AA 2018/2019

LECTURE 4: FAILURE DETECTION AND LEADER ELECTION

Recap on Timing Assumptions

Synchronous

- timing assumptions are <u>explicit</u> either on
  - Bounds on process executions and communication channels, or
  - Existence of a common global clock, or
  - Both

Asynchronous

- there are no timing assumptions

Recap on Timing Assumptions

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

Two choices:

1. Put assumption on the system model (including links and processes)
2. Create a 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

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 stronger are the timing assumption, 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)

Perfect Failure detectors (P)

System model

- synchronous system
- crash failures

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

Perfect failure detectors (P) Specification

Module 2.6: Interface and properties of the perfect failure detector

Module:

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

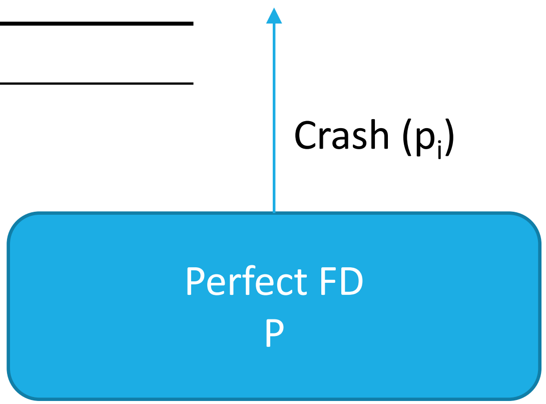
Events:

Indication: $\langle\, \mathcal{P}, \mathit{Crash} \mid p\, \rangle$: 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** $\mathcal{P}$.

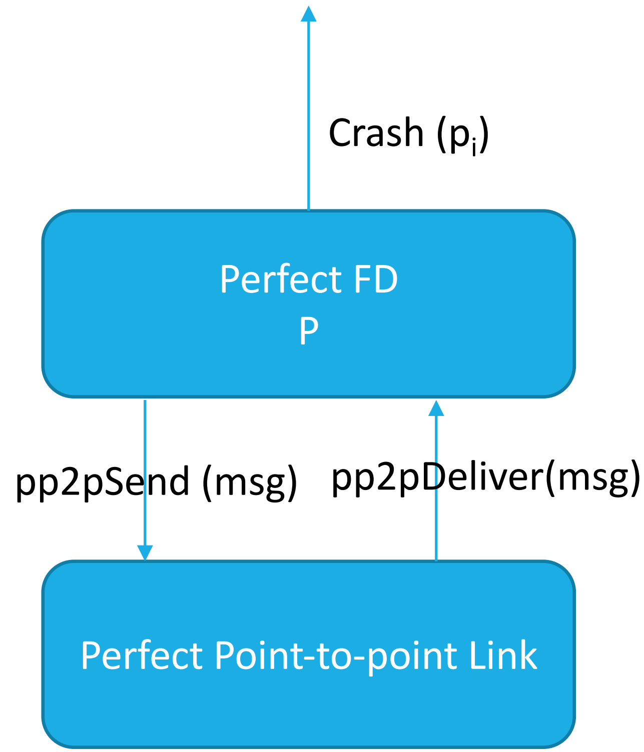
Uses:
PerfectPointToPointLinks, **instance** *pl*.

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

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 $\langle\, \mathit{pl}, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}] \,\rangle$;
  $\mathit{alive} := \emptyset$;
  $\mathit{starttimer}(\Delta)$;

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

- To prove the correctness we have to prove that both Strong Completeness and Strong Accuracy are satisfied

- What if links are fair loss?

- What if we select a timeout too long?

- What if we select a timeout too short?

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 mistake in that periods assuming correct processes as crashed.

The notion of detection becomes suspicious

Eventually perfect failure detectors (◊P) Specification

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

Module:

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

Events:

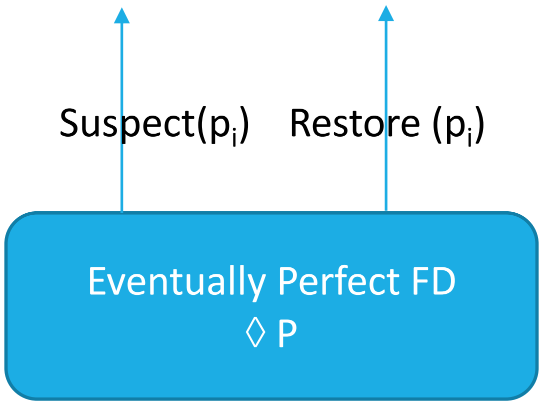
Indication: $\langle\, \Diamond\mathcal{P}, \mathit{Suspect} \mid p \,\rangle$: Notifies that process p is suspected to have crashed.

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.



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 reverse its judgment as soon as it receives a message from q (updating also the timeout value)
- If q has actually crashed, p does not change its judgment anymore.

Eventually perfect failure detectors (◊P) Implementation

Algorithm 2.7: Increasing Timeout

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

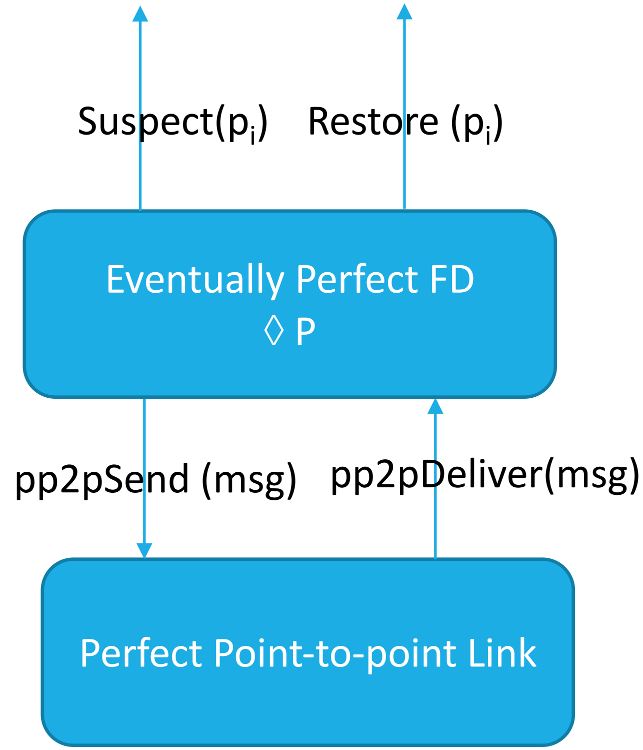
Uses:
PerfectPointToPointLinks, **instance** *pl*.

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

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 $\langle\, \mathit{pl}, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}]\,\rangle$;
  $\mathit{alive} := \emptyset$;
  $\mathit{starttimer}(\mathit{delay})$;

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. If p was wrongly suspected by q, then q will revise its suspicious.

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*.

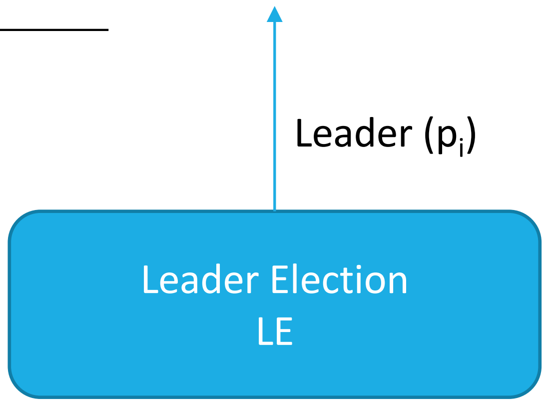
Events:

Indication: $\langle$ *le*, *Leader* $\mid p \rangle$: 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

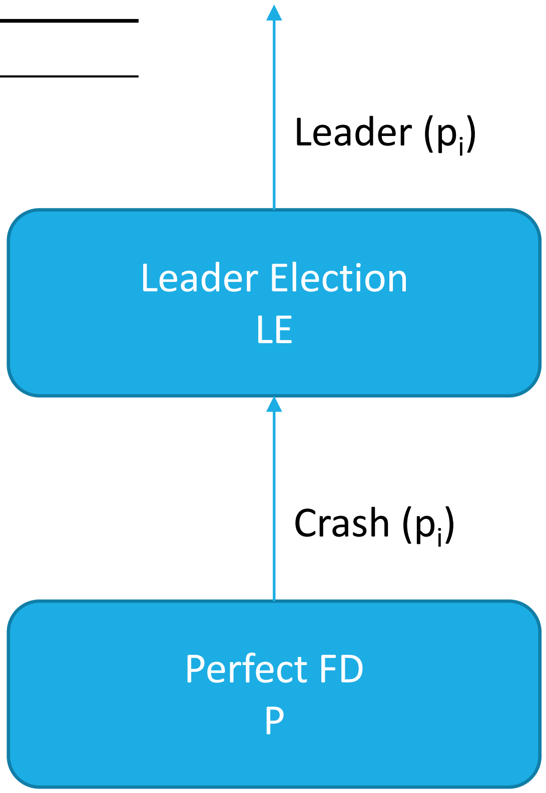
Implements:
LeaderElection, **instance** *le*.

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

upon event $\langle$ *le*, *Init* $\rangle$ **do**
$suspected := \emptyset$;
$leader := \perp$;

upon event $\langle$ $\mathcal{P}$, *Crash* | 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* $\rangle$;



Correctness

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.

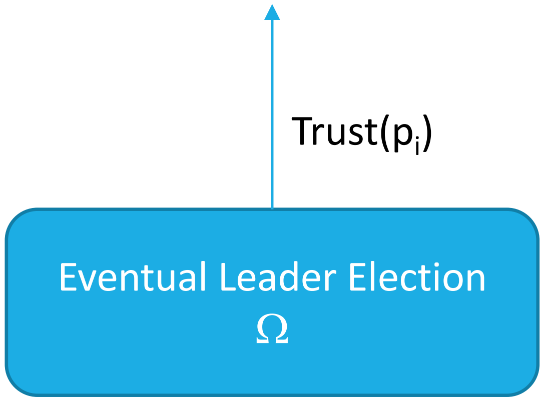
Events:

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



Observation on Ω

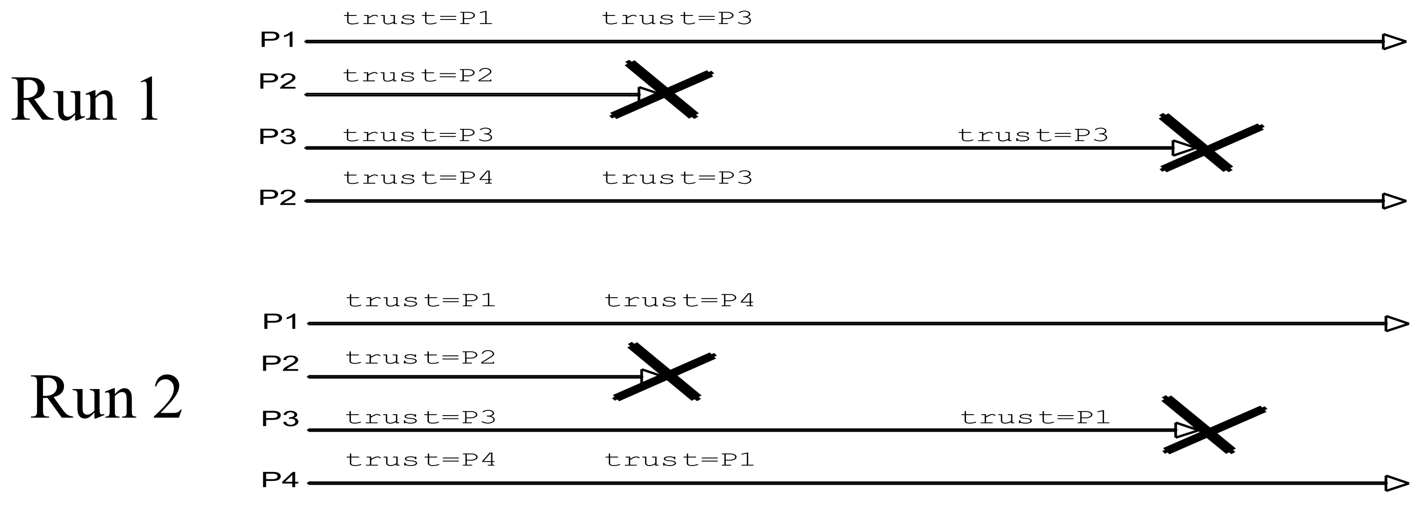
Ω ensures that *eventually* correct processes will elect the same correct process as their leader

Ω does not guarantee 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
Eventual Accuracy	Not verified	Verified
Eventual Agreement	Verified	Not verified

Eventual leader election (Ω)

Using Crash-stop process abstraction

- Obtained directly by <>P by using a deterministic rule on processes that are not suspected by <>P
- trust the process with the highest identifier among all processes that are not suspected by <>P

Assume the existence of a correct process (otherwise Ω cannot be built)

Ω Implementation

Algorithm 2.8: Monarchical Eventual Leader Detection

Implements:
EventualLeaderDetector, **instance** Ω.

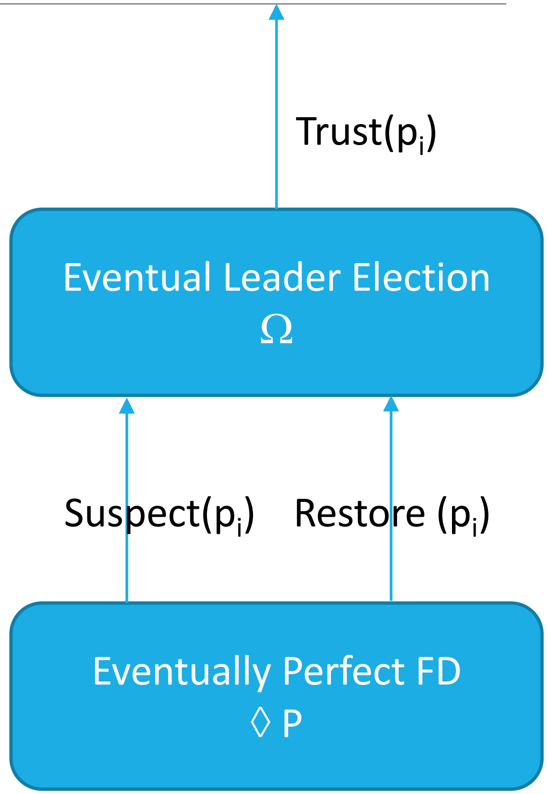
Uses:
EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

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* $\rangle$;



Eventual leader election (Ω)

System model

- Crash-Recovery
- Partial synchrony

Under this assumption, a correct process means:

1. A process that does not crash or
2. A process that crashes, eventually recovers and never crashes again

Ω With crash-recovery, fair lossy links and timeouts

Algorithm 2.9: Elect Lower Epoch

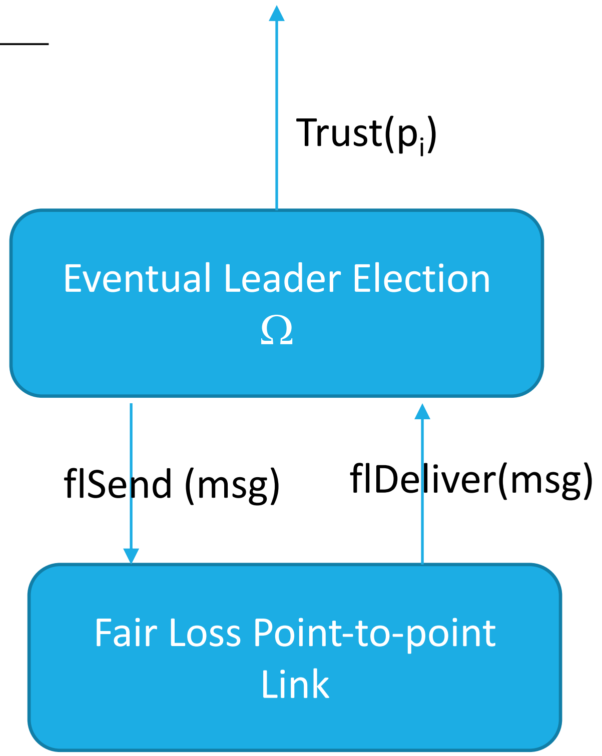
Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event $\langle$ Ω, *Init* $\rangle$ **do**
epoch := 0;
store(*epoch*);
candidates := $\emptyset$;
trigger $\langle$ Ω, *Recovery* $\rangle$;

keeps track of how many times the process crashed and recovered

upon event $\langle$ Ω, *Recovery* $\rangle$ **do**
leader := maxrank(Π);
trigger $\langle$ Ω, *Trust* | *leader* $\rangle$;
delay := Δ;
retrieve(*epoch*);
epoch := *epoch* + 1;
store(*epoch*);
forall $p \in \Pi$ **do**
trigger $\langle$ *fll*, *Send* | p, [HEARTBEAT, *epoch*] $\rangle$;
candidates := $\emptyset$;
starttimer(*delay*);



Ω With crash-recovery, fair lossy links and timeouts

Algorithm 2.9: Elect Lower Epoch

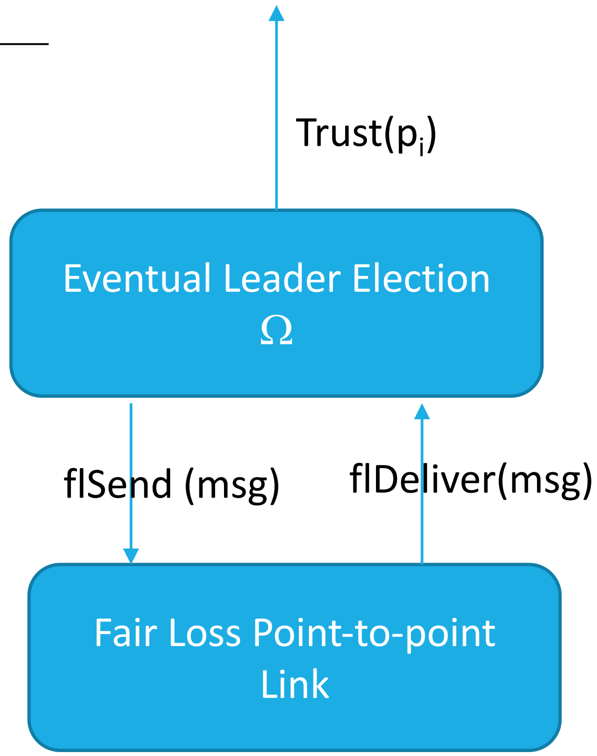
Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event $\langle$ *Timeout* $\rangle$ **do**
  newleader := *select*(*candidates*);
  if *newleader* $\neq$ *leader* **then**
    delay := *delay* + Δ;
    leader := *newleader*;
    trigger $\langle$ Ω, *Trust* | *leader* $\rangle$;
  forall $p \in \Pi$ **do**
    trigger $\langle$ *fll*, *Send* | p, [HEARTBEAT, *epoch*] $\rangle$;
  candidates := $\emptyset$;
  starttimer(*delay*);

upon event $\langle$ *fll*, *Deliver* | q, [HEARTBEAT, *ep*] $\rangle$ **do**
  if exists $(s, e) \in$ *candidates* such that $s = q \wedge e < ep$ **then**
    candidates := *candidates* $\setminus \{(q, e)\}$;
  candidates := *candidates* $\cup$ (q, ep);

deterministic function returning one process among all candidates (i.e., process with the lowest epoch number and among the ones with the same epoch number the one with the lowest identifier)



References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 2 – from Section 2.6.1 to Section 2.6.5