message passing

message passing:

- send/receive messages with asynchronous communication (each message sent by a process is eventually received by a correct process)
- process may crash (stop its execution)
- process p is correct if it doesn't crash (makes an infinity of steps)

Shared memory versus message passing

simulating message passing memory:

P sends the kth message m to q —> M[p][q][k]=m

P receives a message from q

Init k=1; M[p][q][*]= $\perp$

Repeat forever

If M[p][q][k] $\neq \perp$ then receive (m); k++;

Message Passing versus Shared memory

simulating shared memory: with a majority of correct processes, atomic registers may be implemented in asynchronous message passing models.

(Message-Passing and shared memory models are « equivalent » -with a majority of correct processes)

Simulating shared registers in message passing

simulation of a single-writer single-reader regular register

- assume a majority of correct processes

For the writer

to write(v)

$seq := seq + 1$

send (W, v, seq) to all

wait until receiving $\lfloor n/2 \rfloor + 1$ messages (ACK, seq)

For the reader

to read()

send(R,rseq) to all

wait until receiving $\lfloor n/2 \rfloor + 1$ messages $(V, v, rseq)$

return val such that $(V, Val, S, rseq)$ has been received

and S is the max of sequence number of received V messages

For all processes

when (W, v, s) is received

if $s > seq$ then

$val := v; seq := s$

send (ACK, s)

when (R, r) is received

send $(V, val, seq, rseq)$ to all

A majority of correct processes is needed

partition argument:

- if $n \leq 2t$ then we can partition the set of processes in two set S_1 and S_2 such that $|S_1| \geq t$ and $|S_2| \geq t$.
  - Run A_1: all processes in S_1 are correct and all processes in S_2 are initially dead, p_0 invokes a write(1), at some time t_0 the write terminates
  - Run A_2: all processes in S_2 are correct and all processes in S_1 are initially dead, p_1 S_2 invokes a read(), at time t_0 +1 the read terminates at time t_1
  - Run B: « merge » of A_1 and A_2 but no process crash. write(1) terminates before the read() and the read return 0

contradiction

Consensus…
circumvent Impossibility

Remember…

- n asynchronous processes p_0, p_1,…,p_{n-1}
- processes can fail by crashing:
  - a process that fails is a faulty process, a process that never fails is a correct process (it makes infinitely many steps)
  - t-resilient system: up to t processes can crash
- processes communicate via atomic registers (or any objects that can be constructed from atomic registers)

Impossibility of Consensus

- Thm: Using read-write registers, with asynchronous processes, it is impossible to solve deterministically the consensus t-resilient even for t=1

BUT: the consensus is universel (for objects)

Circumventing consensus impossibility

« Strengthening » the model

- communication primitive: using more « powerful » objects than read-write register
- synchrony assumptions: round model, partial synchrony

Relaxing properties

- *safety* (agreement with probability 1, epsilon agreement),
- *liveness* (safe agreement, obstruction free, termination with probability 1)

relaxing liveness : safe agreement

- Safety: Agreement + Validity
- Liveness : If every participant takes enough steps then every correct process decides
- a process *participates* if it takes at least one step

More precisely:

- propose(v) returns the decided value (or doesn't terminate)
  - every decided value was proposed (validity)
  - no two different values are decided (agreement)
  - Termination
    - every correct process eventually decides (terminates) if some process decides or if every participant takes enough steps
  -

Safe agreement for n processes

Shared variables

$A[0, ..., n-1]$ atomic snapshot object init $\perp$
$B[0, ..., n-1]$ atomic snapshot object init $\perp$

Upon *propose*(v) **by process** p_i

A.update(v)
U:= A.snapshot()
B.update(U)
repeat
V:=B.snapshot()
until $\forall j : U[j] \neq \perp \Rightarrow V[j] \neq \perp$
Let X be the vector in V with the smallest number of non $\perp$ value
decide on the smallest non $\perp$ value in X

proof...

- Liveness : clear
- Validity: all non $\perp$ values in vector of B have been previoulsy proposed
- Agreement : Consider the process p_i that wrote the smallest snapshot S to B (B.update(S))
  - for all j that takes a snapshot V of B, we have:
  - p_j has taken previously a snapshot U of A (code), and $S \subseteq U$ (snapshot containment).
  - $S[i] \neq \perp$ (code), then $U[i] \neq \perp$. So p_j waits until p_i updates B by S.

Relaxing liveness: obstruction freedom

- (Binary consensus)
- Safety: Agreement + Validity
- Liveness:
  - if alone a process terminates
  - *Obstruction free termination:* if at some point of the execution a process takes (enough) steps alone then it decides

Shared variables

A infinite array of atomic snapshot object init $\perp$
decide init $\perp$

Upon *propose*(v) **by process** p_i

Obstruction Free

prop:=v
for r:=0 to ∞
A[2r].update(prop)
U:= A[2r].snapshot()
if all values non $\perp$ in U are equal to prop
then report:=prop
else report:=?
A[2r+1].update(report)
V:= A[2r+1].snapshot()
Let L be the non $\perp$ in V
if $|L| = 1$ then
if $L = \{1\}$ then decide:=1
if $L = \{0\}$ then decide:=0
else
if $L = \{1, ?\}$ then prop:=1
if $L = \{0, ?\}$ then prop:= 0
if decide $\neq \perp$ then return decide

- Validity (trivial)
- **Agreement?**

agreement

U_i and U_j at round r for processes p_i and p_j,
and let x_i and x_j be the non $\perp$ values of U_i and U_j

We may have (snapshot inclusion):

- $x_i = \{1\}$ and ($x_j = \{1\}$ or $x_j = \{1, 0\}$)
- $x_i = \{0\}$ and ($x_j = \{0\}$ or $x_j = \{1, 0\}$)

it is impossible to get $x_i = \{0\}$ and $x_j = \{1\}$

So either:

- ($report_i = 1$ and ($report_j = 1$ or ?)) or
- ($report_i = 0$ and ($report_j = 0$ or ?)) or
- ($report_i =$? and ($report_j = 1$ or $report_j = 0$ or ?))

it is impossible to get i and j such that $report_i = 1$ and $report_j = \{0\}$

In the same way:
V_i and V_j at round r for processes p_i and p_j,
and let x_i and x_j be the non $\perp$ values of V_i and V_j

We may have:

- if $x_i = \{1\}$ then $(\forall j : (x_j = \{1\}$ or $x_j = \{1, ?\}))$
- if $x_i = \{1, ?\}$ then $(\forall j : (x_j = \{1\}$ or $x_j = \{1, ?\})$ or $(\forall j : (x_j = \{?\}$ or $x_j = \{1, ?\}))$
- if $x_i = \{?\}$ and $(\forall j : (x_j = \{?\}$ or $x_j = \{1, ?\})$ or $(\forall j : (x_j = \{?\}$ or $x_j = \{0, ?\}))$

it is impossible to get $x_i = \{0\}$ and $x_j = \{1\}$

If a process decides 1 then the other processes either decide 1 or have prop=1 (and don't decide 0), at the next round all processes have prop=1 and they decide 1.

If a process decides 0 then the other processes either decide 0 or have prop=0 (and don't decide 1) at the next round all processes have prop=0 and they decide 0.

liveness

- Liveness : if from the beginning of a round a process is alone, it decides at the end of a round : obstruction-free

Relaxing liveness: terminaison with probability 1

- Binary consensus
- Each process is equipped with a random number generator $P(1)=p_1>0$ and $P(0)=p_0>0$
- Safety: Agreement+ Validity
- Liveness: termination with probability 1

Shared variables

A infinite array of atomic snapshot object init $\perp$

decide init $\perp$

Upon *propose*(v) **by process** p_i

prop:=v

for r:=0 to ∞

A[2r].update(prop)

U:= A[2r].snapshot()

if all values non $\perp$ in U are equal to prop

then report:=prop

else report:=?

A[2r+1].update(report)

V:= A[2r+1].snapshot()

Let L be the non $\perp$ in V

if $|L| = 1$ then

if $L = \{1\}$ then decide:=1

if $L = \{0\}$ then decide:=0

if $L = \{?\}$ then prop:= r.n.g

else

if $L = \{1, ?\}$ then prop:=1

if $L = \{0, ?\}$ then prop:= 0

if decide $\neq \perp$ then return decide

- Safety as previously
- Liveness :
  - if some process decides at some round r then all processes decide by the end of round r+1 (as before)
  - if all processes have L={x,?} or L={x} at the end of the round every process proposes or decides x : then they all decide x by the next round
  - if some processes have L={x,?} and the others have L={?} at the end of the round
    - With probability at least equal to $(p_x)^n$ (>0) all processes choose x by r.n.g. and propose x for the next round and decide.

Leader election oracle Ω

At every process and each time Ω outputs a process identifier in $leader_i$

Eventually, the same correct process is output at every correct process

- There is a correct process q and a time after which all correct processes p_i have $leader_i$=q

Ω failure detector

(May be done in a system where after some time, there exists k, there exists a correct process that takes at least one step each k steps of every process: limited asynchrony)

Shared variables

A infinite array of atomic snapshot object init $\perp$
decide init $\perp$

Upon $propose(v)$ **by process** p_i

prop:=v
for r:=0 to ∞
wait until leader=p_i or decide $\neq \perp$
A[2r].update(prop)
U:= A[2r].snapshot()
if all values non $\perp$ in U are equal to prop
then report:=prop
else report:=?
A[2r+1].update(report)
V:= A[2r+1].snapshot()
Let L be the non $\perp$ in V
if $|L| = 1$ then
if $L = \{1\}$ then decide:=1
if $L = \{0\}$ then decide:=0
else
if $L = \{1, ?\}$ then prop:=1
if $L = \{0, ?\}$ then prop:= 0
if decide $\neq \perp$ then return decide

Strengthening synchrony assumptions:

- Binary consensus
- Leader election (failure detector Ω)
  - Each process have a local variable leader
    - There is a correct process q and a time after which all correct processes p_i have $leader_i$=q
- *May be done in a system where after some time, there exists k, there exists a correct process that takes at least one step each k steps of every process*

Failure Detector Chandra&Toueg

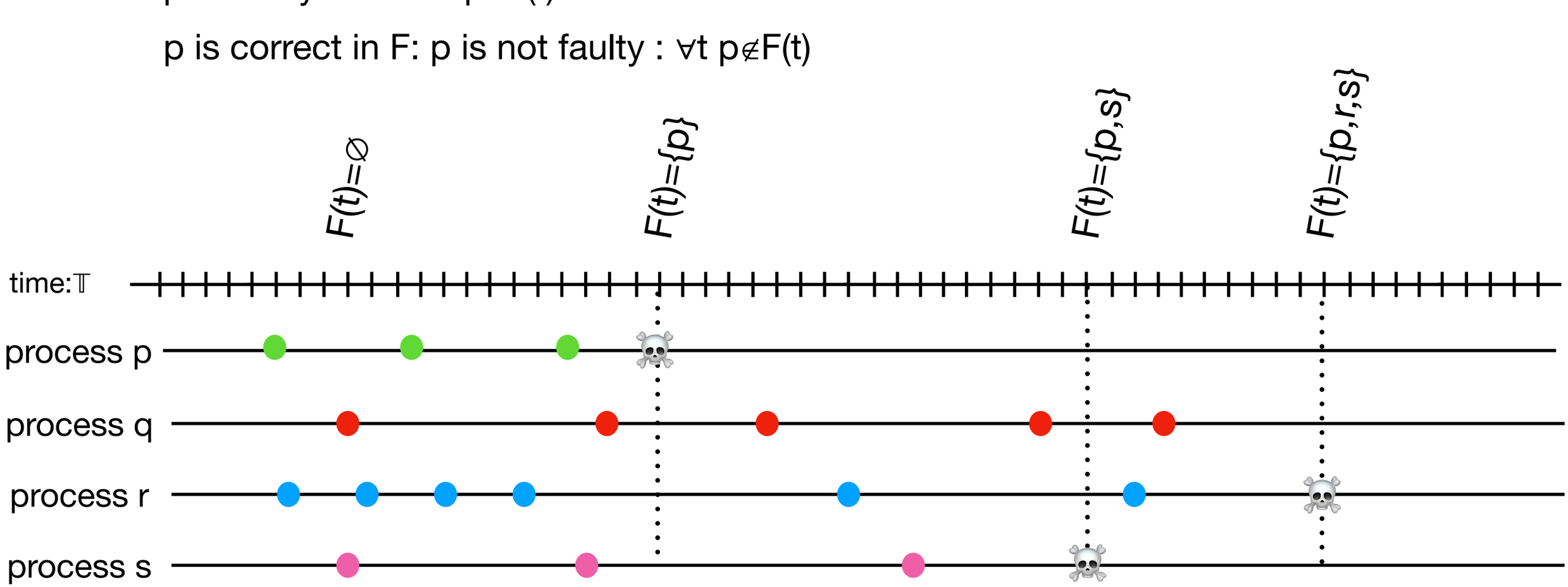
- PODC 91 - J. ACM 96
- Distributed oracles that give hints on the failure pattern (e.g set of suspected processes)
- A FD is basically defined by:
  - a completeness property: actual detection of failure
  - an accuracy property: restrict the mistake that a FD can make

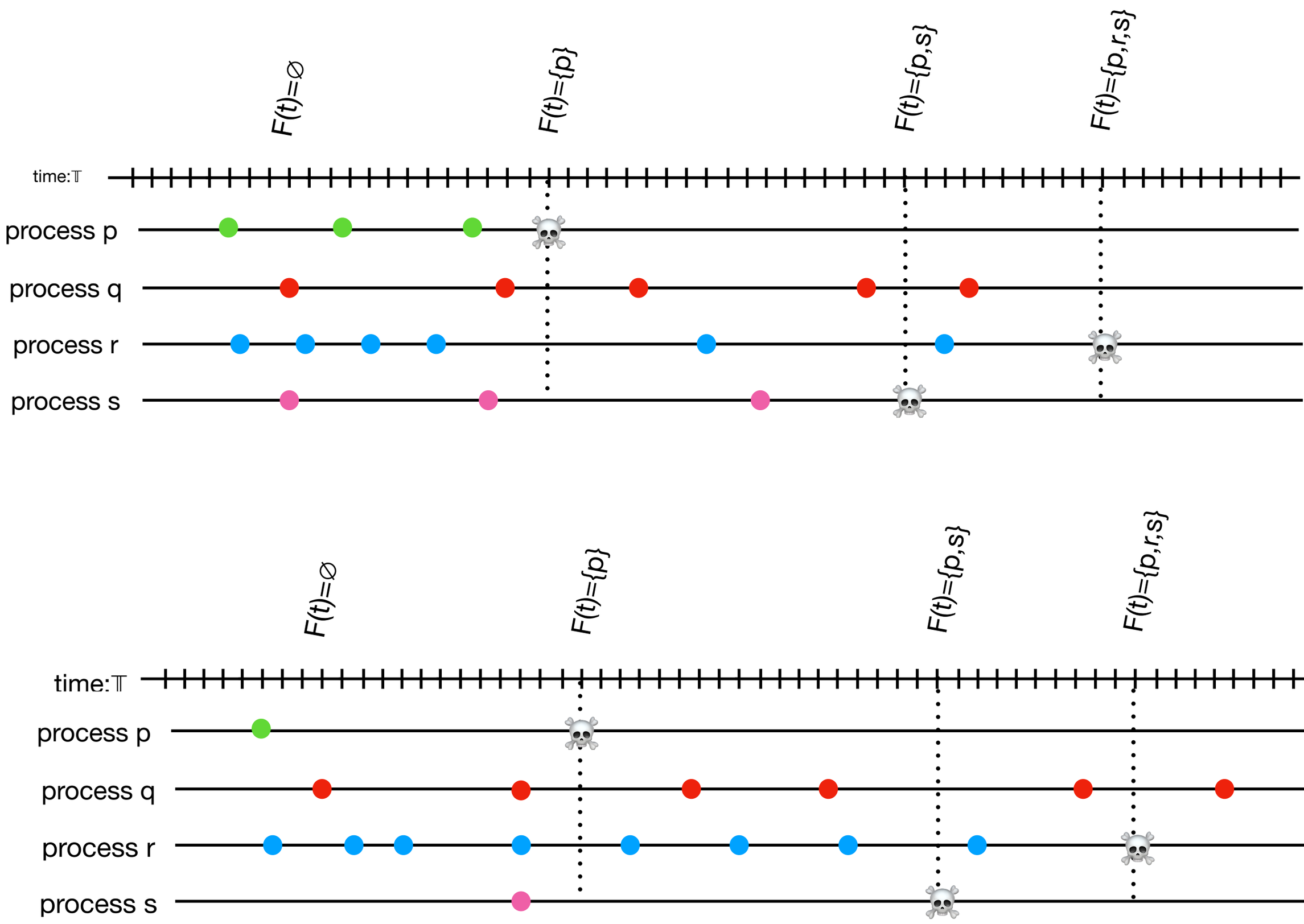
Failures pattern

- failures and failure pattern:
  - at time t:p is faulty (dead, crashed: p stops making steps) or is alive (p eventually makes steps)
  - *failure pattern* (schéma de pannes): F(t) is the set of faulty processes at time t; (F(t) ⊆ F(t+1))

    p is faulty in F: ∃ t p∈F(t)

    p is correct in F: p is not faulty : ∀t p∉F(t)





another schedule for the same failure pattern⚠

Failure detector[CT96]

- *failure detector:* oracle that may be invoked locally by processes depending only on failure pattern (not on the schedule!). At each step a process may invoke its failure detector module and gets an answer

- depending only on the failure pattern: failure detector $\mathscr{X}$ is defined for each failure pattern F by « histories » H

    - H history of $\mathscr{X}$ for the failure pattern F: H(p,t) is the output of $\mathscr{X}$ for process p at time t (if p invokes failure detector $\mathscr{X}$ at time t, p gets the answer H(p,t))

Failure detectors

Examples of failure detectors with lists of suspected processes:

- $\mathscr{P}$ perfect FD: completeness + strong accuracy
- $\diamond\,\mathscr{P}$: completeness + eventual strong accuracy
- $\diamond\,\mathscr{S}$: completeness + eventual weak accuracy
- Ω: outputs at each process one id of process (the expected leader):
  eventually Ω(p,t) is forever the same correct process for all processes (leader election)

$\mathscr{P}$ enables the consensus (synchronous rounds)

$\diamond\,\mathscr{P}$, $\diamond\,\mathscr{S}$, Ω enable the consensus (in shared memory or with a majority of correct processes in message passing)

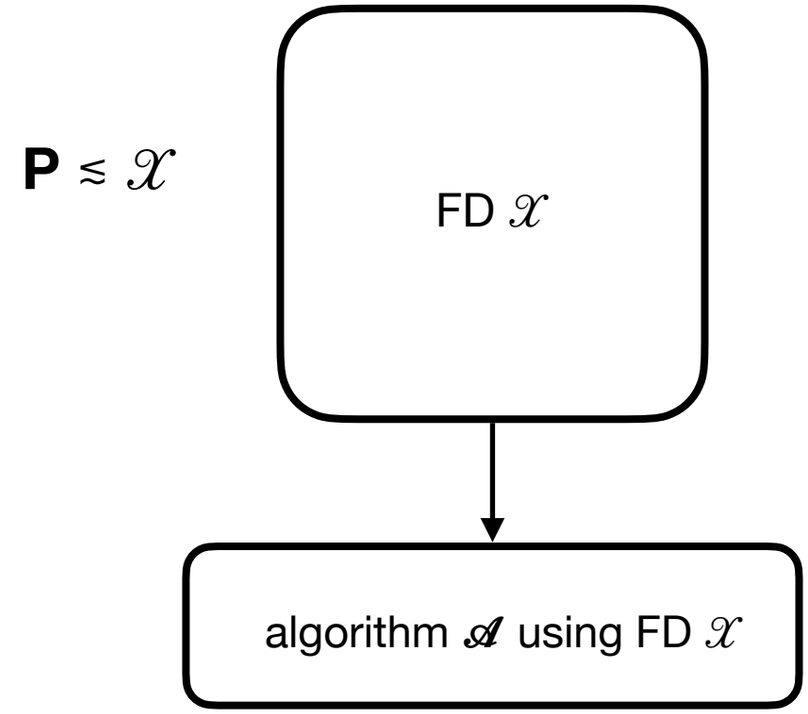
Comparing failure detectors

Remarks:

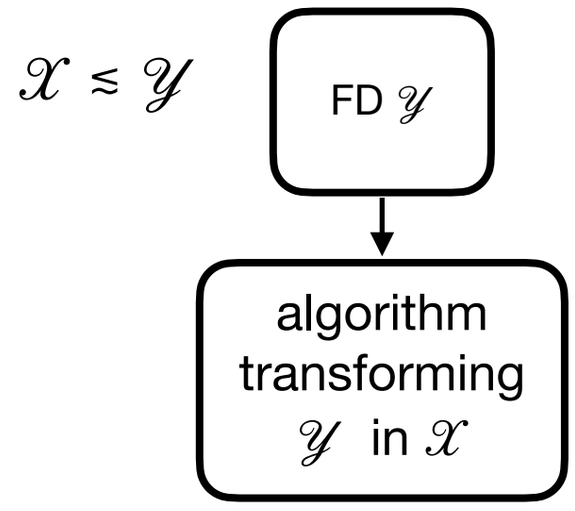
- environment: set of of failure pattern e.g. less than m faulty processes (here implicitly any number of faulty processes)
- shared memory versus message passing: depending on environments, solvability is not (exactly) the same for message passing and shared memory

Comparing failure detectors

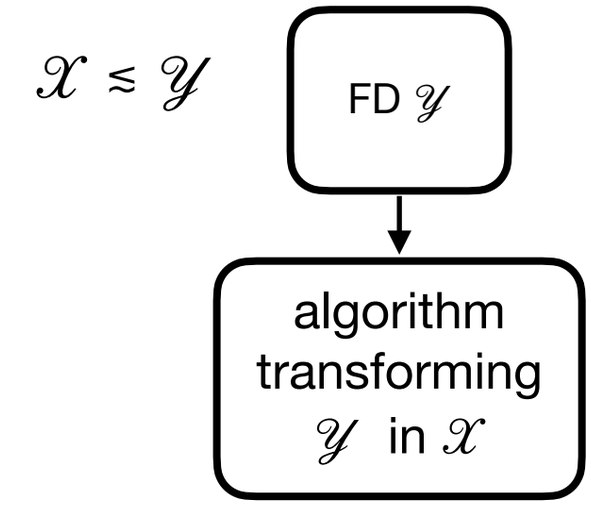
- Problem P is solvable with FD $\mathscr{X}$ iff there is a distributed algorithm using FD $\mathscr{X}$ that solves P



- implementing a FD $\mathscr{X}$ may be (with more formal definitions) considered as a problem

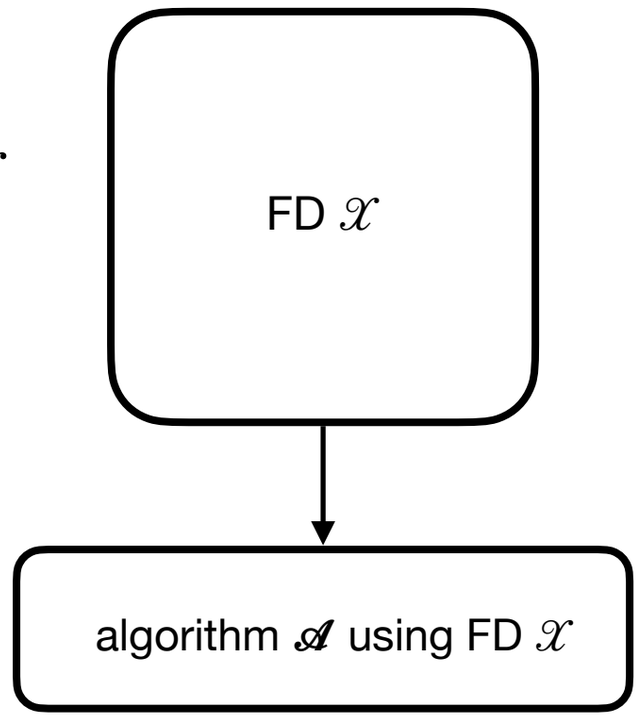


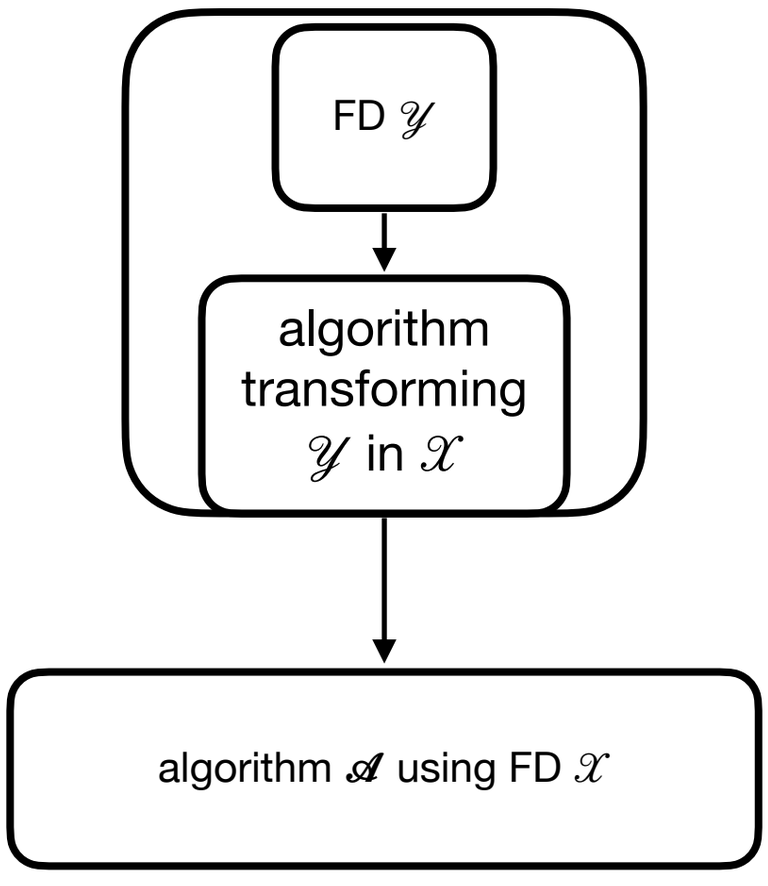
$\mathcal{X} \preceq \mathcal{Y}$



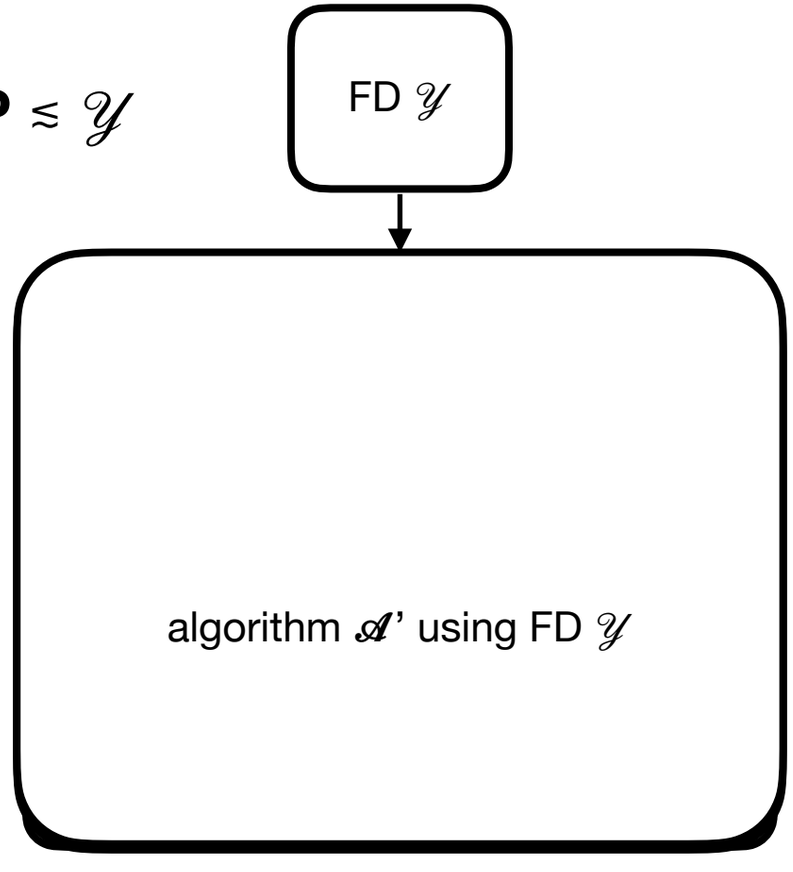
and

$\mathbf{P} \preceq \mathcal{X}$





$\mathbf{P} \preceq \mathcal{Y}$



Comparing failure detectors

$\mathscr{A} \precsim \mathscr{B}$ ($\mathscr{A}$ is weaker than $\mathscr{B}$) : there is a (distributed) algorithm with failure detector $\mathscr{B}$ that implements $\mathscr{A}$.

- $\mathscr{B}$ provides at least as much information about failures than $\mathscr{A}$
- (intuitively: less information about failures in $\mathscr{A}$ than in $\mathscr{B}$)

$\mathscr{A} \prec \mathscr{B}$ ($\mathscr{A}$ is strictly weaker than $\mathscr{B}$) if $\mathscr{A} \precsim \mathscr{B}$ and not $\mathscr{B} \precsim \mathscr{A}$

if $\mathscr{A} \precsim \mathscr{B}$ and $\mathscr{B} \precsim \mathscr{C}$ then $\mathscr{A} \precsim \mathscr{C}$ and (with more formal definitions) $\mathscr{A} \precsim \mathscr{A}$

- $\precsim$ is a pre-order.

Weakest failure detector

As $\preceq$ is a pre-order, failure detectors can be compared and given some set S of failure detectors $\mathscr{M}$ is the weakest for S iff for all $\mathscr{X} \in$S $\mathscr{M} \preceq \mathscr{X}$

(from any $\mathscr{X}$, $\mathscr{X}$ contains at least as much information about failures than $\mathscr{M}$)

given a « problem » P, failure detector $\mathscr{WK}$ is a weakest failure detector for P iff

1. *$\mathscr{WK}$ can be used to solve P*
2. *for all $\mathscr{X}$ that can be used to solve P: $\mathscr{WK} \preceq \mathscr{X}$*

$\mathscr{WK}$ encapsulates the minimum information about failures to solve problem P

Comparing problems with failure detectors

Given a « problem » P (solvable by a FD) there is a weakest failure detector to solve P [JT2008]

from the hierarchy of FD to hierarchy of problems:

- Given problems P and Q: $P \lesssim_{FD} Q$ iff Q solvable with some FD $\mathscr{X}$ then P is also solvable with $\mathscr{X}$
- $P \lesssim_{FD} Q$ iff the WFD(P) $\lesssim$ WFD(Q)

Example: $\diamond \mathcal{S}$ and Ω

- Ω: outputs at each process one id of process (the expected leader):
  eventually $\Omega(p,t)$ is forever the same correct process for all processes (leader election)
- $\diamond \mathcal{S}$: completeness + eventual weak accuracy

Ω >= <>S: clear

<>S >= Ω :

```
each process p :V[p] : array of size n
(code of p)
each k steps :
    Q=query the failure detector <>S
    for all processes q :if q in Q then V[p][q] is incremented
    compute V[p][q] = max V[x][q]
    output of Ω = q such that V[p][q] is the smallest element of V[p][*]
```

quorum failure detector Σ

- *Majority*: ensures the quorum property i.e. given any two sets of messages received from a majority of processes at least one comes from the same process

- Σ quorum failure detector.

  history H $\in \Sigma$: H(p,t) list of processes (trusted)

  - *intersection:* every two lists intersect

    for all p, q for all t, t' H(p,t)∩H(q,t')
  - *completeness:* there is some time after which for all p H(p,t) ⊆ Correct(F)

Majority and Σ

With a majority of correct prowesses:

Output=Π (set of all processes)

repeat forever

send(ARE_YOU_ALIVE,r) to all

wait until receive (I_AM_ALIVE,r) from a majority

Σ={q| a message (I_AM_ALIVE,r) received from q}

r:=r+1

||

when receive (ARE_YOU_ALIVE,r) from q

send (I_AM_ALIVE,r) to q

completeness:
let t s.t. after time t no new process crashes and all messages from crashed processes are received;
p receives only messages from processes that are alive

intersection: two majorities intersect !

remark: if a majority of processes are correct then the « wait » always terminate (and all correct processes progress)

with a majority of correct processes Σ is implementable

Σ weakest FD for registers

We have to prove:

1. With Σ it is possible to implement registers.

   done!

2. Consider an implementation of registers with a failure detector $\mathscr{D}$, we have to prove that $\Sigma \lesssim \mathscr{D}$ (we can extract Σ from $\mathscr{D}$)

Σ is the weakest failure detector to implement registers in message passing

Consensus = Ω + CA

Ω enables to solve consensus

Shared:

D[1,…,∞], regular registers, initially T

CA_1,CA_2,… a series of commit-adopt instances

Upon propose(v) by process p_i:

v_i := v

r := 0

repeat forever

r++

$(c,v_i):=CA_r(v_i)$ // r-th instance of commit-adopt

if c=true then

$D[r]:=v_i$ // let the others learn your value

return v_i

repeat

if Ω outputs p_i then

$D[r]:= v_i$ // advertise your value if leader

until D[r]=v' where v'≠T //wait until the leader writes its value

v_i := v' //adopt the leader's value

Weakest failure detector to solve consensus

- failure detector Ω : outputs at each correct process a leader (process id) such that eventually all correct processes agree forever in the same leader and this leader is a correct process.
  For all histories H of Ω:
  ∃ leader ∈ Correct(F) ∃ t ∀t'>t ∀p ∈ Correct(F) H(p,t')=leader

Main result: with a majority of correct processes (or in shared memory) Ω is the weakest failure detector to solve the consensus[CHT16]

Extension: $\Omega x \Sigma$ is the weakest failure detector to solve the consensus in message passing [DFG10]

to get consensus we need a leader!

Solving problem P (practical point of view)

- Given some problem P that is not solvable in asynchronous models

  - From a problem P, find the weakest failure detector WFD(P) for P

  - Find the « best » model in which WFD(P) can be implemented + a « good » implementation of WFD(P)

  - With the « good » implementation of WFD(P) and a (good) algorithm using FD: WFD(P) solves P

Implementing failure detectors

- Implementing Ω:
  - « good » partially synchronous model:
    - link p to q is *eventually timely*: there exists delta, there exists a time after which all messages from p to q are received in less than delta
    - if there is a (correct) process p such that all links from p are eventually timely then Ω may be implemented (source)
    - (then in such systems consensus is possible)

implementing Ω

process p:
  sends regularly message(alive) to all

  when no (alive) message from q since $timeout_p[q]$
    $timeout_p[q]= timeout_p[q] + 1$
    $counter_p[q]=counter_p[q] + 1$
    reset timer for alive message from q
    send to all ($counter_p$)

when receives (counter)
  $counter_p=max(counter_p, counter)$
  send to all ($counter_p$)

$leader_p=min\{r|\ counter_p(r)=min\ \{counter_p(x)|x\in \Pi\}\}$

Consider a source s:
  at some time for all (alive) processes $timeout_p[s]$ is greater than communication delay from s
then $counter_p[s]$ stops increasing
eventually
$counter_p[s]=counter_q[s]$

q is not a source $counter_p[q]$ increases forever for all p