Consensus et détection de fautes

Pierre Sens

Projet Delys(LIP6/Inria)

Pierre.Sens@lip6.fr

http://lip6.fr/Pierre.Sens/

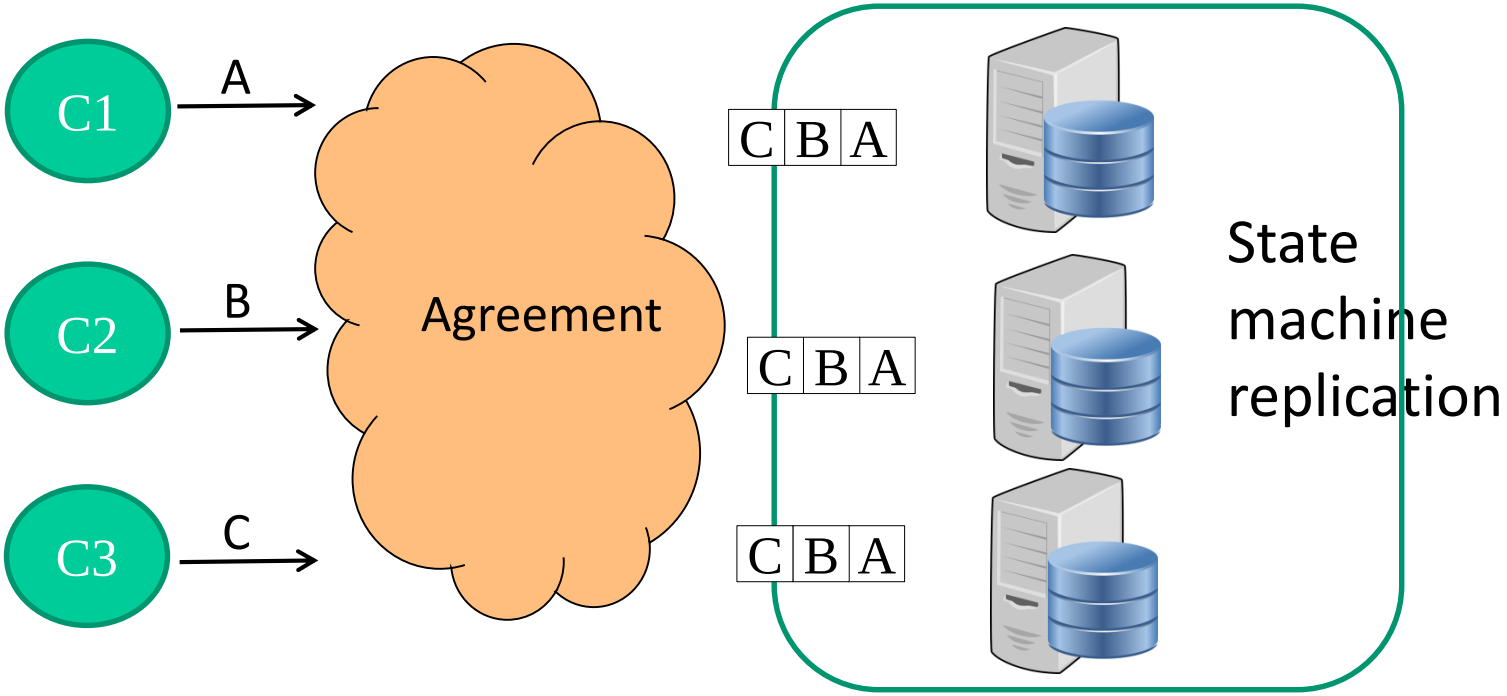
Algorithmes répartis tolérants les fautes

- Construction d'applications fiables
  - Problème complexe
  - Agencement de primitives fiables
- Définition de primitives
  - Consensus, diffusion atomique, gestion de groupe, …

Le consensus = dénominateur commun

Problème d'accord

- Abstraction fondamentale pour construire des services fiables



Accord sur l'ordre des opérations

Spécification du consensus (1)

- Permettre l'accord entre processus

Initialement 1 valeur initiale par processus

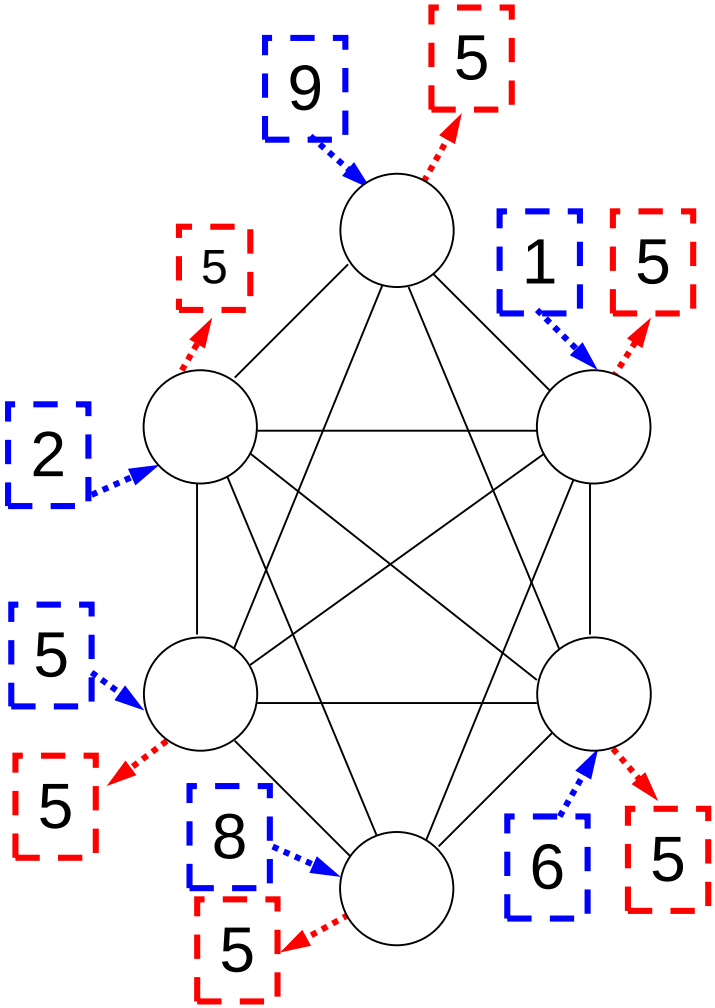
Finalement Tous les processus corrects décident une même valeur

Validité: si un processus décide *v* alors *v* est une valeur proposée

Terminaison : tous les processus corrects décident finalement

Cohérence (agreement) : deux processus corrects ne peuvent décider différemment

[Intégrité : un processus doit décider au plus une fois]



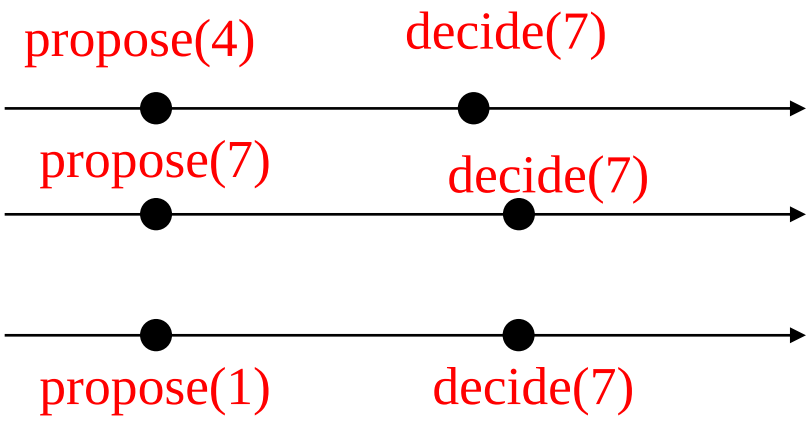
Spécification du consensus (2)

Consensus uniforme :

- Validité: si un processus décide *v* alors *v* est une valeur proposée
- Terminaison : tous les processus corrects décident finalement
- Cohérence uniforme (uniform agreement) : deux processus ~~corrects~~ ne peuvent décider différemment

Spécification du consensus (3)

- Deux primitives:
  - propose (v): le processus appelant propose une valeur initiale v
  - decide(v): le processus appelant décide v



Définition et modèle (1) : Processus

2 types de processus:

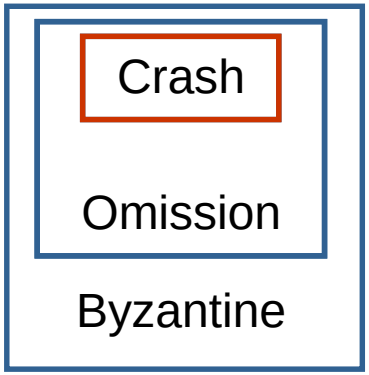
- **correct** : ne défaille pas pendant toute la durée de l'exécution
- **fautif** : pas correct

- Interconnexion :
  - $\Pi = \{p1, p2, \ldots, p_N\}$ – N processus communiquent par passage de messages
  - Graphe complet

Définition et modèle (2) : Types de fautes

Processus :

- **Franche** (*crash*) : le processus fautif n'émet plus ni ne reçoit de message de façon *permanente* = silence sur défaillance - *fail-silent*, variante *fail-stop* (faute visible)
- **Omission** : Transitoire
- **Temporaire** : Trop tôt ou trop tard
- **Byzantin** : malveillance



Canaux :

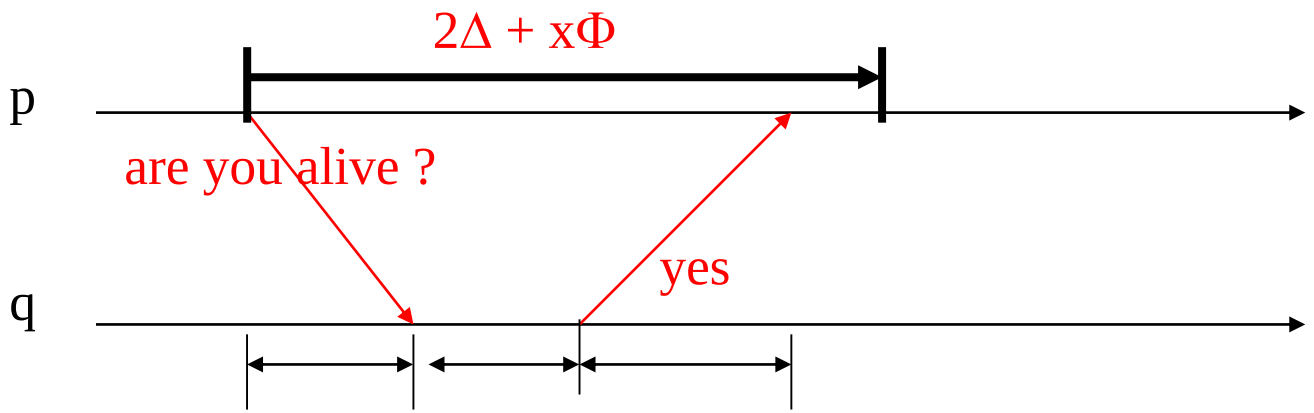
- **Fiable** (*reliable*): si p execute send (m) vers q et *q est correct*, alors q recevra m
- **Quasi-fiable** (*quasi-reliable*): si p execute send (m) vers q et *p et q sont corrects*, alors q recevra m
- **Equitable** (*fair-lossy*) : si un processus correct envoie un message m à q une infinité de fois, alors q recevra m

Définition et modèle (3) : Modèles temporels

- Hypothèse sur les vitesses de transmission et de traitement des messages
- Modèle synchrone :
  - Borne Δ sur le temps de transmission : Si un processus p envoie un message vers q à l'instant t, alors q recoit le message avant $t+\Delta$.
  - Borne Φ sur la vitesse relative des processus : Si le processus le plus rapide prend x unités de temps pour un traitement, alors le processus le plus lent ne peut pas prendre plus $x\Phi$ temps pour faire le même traitement

Modèle temporel : système synchrone

Permet une détection parfaite



Modèle temporel : système asynchrone

- Pas de borne sur les délais de transmission
- Pas de borne sur les vitesses relatives des processus

=> Impossible de distinguer entre un processus lent et un processus « crashé »

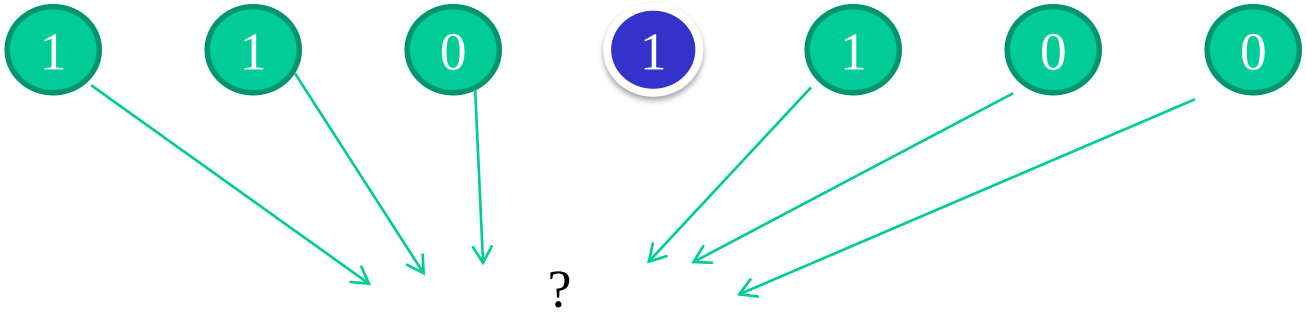
Modèles temporel : systèmes partiellement synchrones

(Dwork, Lynch, Stockmeier, 1988):

- Modèles intermédiaires entre synchrone et asynchrone (32 modèles)
- Bornes Δ et Φ du modèle synchrone :
  1. Existent mais sont **inconnues**, ou
  2. Sont connues mais **ont lieu à partir d'un temps T** appelé GST : global stabilization time
- Avant GST, le système est instable (pas de bornes)
- Après GST, le système est stable (bornes)
- GST est inconnu

Impossibilité de Fischer, Lynch et Paterson [FLP 85]

- Impossible de résoudre le consensus de façon déterministe
  - Asynchrone
  - Réseau fiable
  - 1 seul crash
- Idée :
  - Impossible de différencier un processus défaillant d'un processus lent
  - La décision peut dépendre d'un seul vote

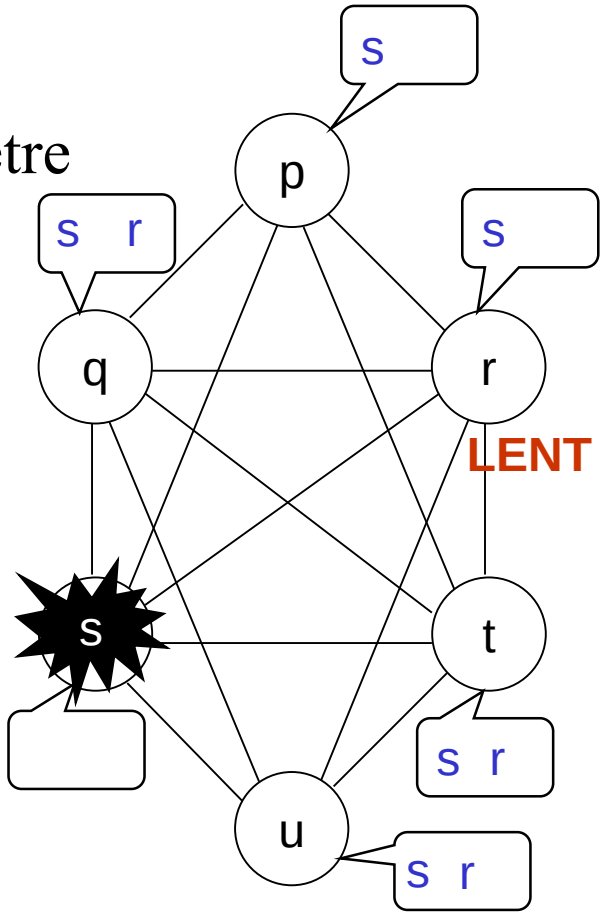


Contourner FLP 85

- **Changer le problème**
  - *k*-agreement [Cha90]
    - Plusieurs valeurs peuvent être décidées
- **Systèmes partiellement synchrones** [DDS87]
  - Les bornes sont non connues, valables uniquement à partir d'un moment
  - Variantes : alternance de bonne et mauvaise périodes
  - Borne restreinte à certains nœuds :
    - 1 bi-source (ultime) : Il existe (ultimement) une borne sur les liens entrants et sortants de la source
    - 1 source (ultime) : Il existe (ultimement) une borne sur les liens sortants
  - Algorithmique dépendante du système
- **Consensus « imparfait »**
  - Consensus probabiliste [BO83] : Des processus peuvent ne pas terminer
  - Paxos [Lamport 89] : Hypothèse très faible, terminaison non assurée
- **Les détecteurs de défaillances non fiables** [CHT96]
  - Algorithmique en asynchrone (indépendante du système)
  - Hypothèses plus facilement utilisables

Détecteur de défaillances non fiables [CHT 96]

- Introduit en 1991
- Oracle local sur chaque nœud
- Fournit une liste des processus suspectés d'être défaillants
- Informations non fiables
  - Possibilité de fausses suspicions



Complétude : un processus défaillant doit être détecté comme défaillant

Justesse : un processus correct ne doit pas être considéré comme défaillant

Qualités des détecteurs

- Complétude (completeness)
  - **forte** : Il existe un instant à partir duquel tout processus défaillant est suspecté par *tous* les processus corrects
  - **Faible** : Il existe un instant à partir duquel tout processus défaillant est suspecté par *un* processus corrects
- Justesse (accuracy) :
  - **Forte** : aucun processus correct n'est suspecté
  - **Faible** : il existe au moins un processus correct qui n'est jamais suspecté
  - **Finalement forte** : *il existe un instant à partir duquel* tout processus correct n'est plus suspecté par aucun processus correct
  - **Finalement faible** : *il existe un instant à partir duquel* au moins un processus correct n'est suspecté par aucun processus correct

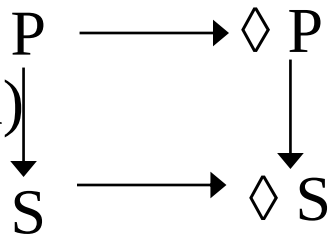
Classes de détecteurs

Hypothèses : pannes franches, communication fiable, réseau asynchrone

	Justesse			
	Forte	Faible	Finalement forte	Finalement faible
Complétude forte	P	S	◊ P	◊S
Complétude faible	Q	W	◊ Q	◊ W

- Complétudes forte et faible sont équivalentes
  (on peut construire une complétude forte à partir d'une faible)
  => 4 classes : P, S, <>P, <>S

- Force des détecteurs (⟶ = implication)



Les complétudes forte et faible sont équivalentes

1. La complétude faible est incluse dans la forte
2. Construction de la complétude forte à partir de la faible

|| *Task 1:* **repeat forever**
$\{p$ *queries its local failure detector module* $\mathcal{D}_p\}$
$suspects_p \leftarrow \mathcal{D}_p$
send $(p, suspects_p)$ to all

|| *Task 2:* **when** receive $(q, suspects_q)$ for some q
$output_p \leftarrow (output_p \cup suspects_q) - \{q\}$

→ Uniquement 4 classes de FD : P, S, ◊P, ◊S

Détecteur et consensus

- Consensus résoluble avec ◊S
- ◊S le plus faible détecteur pour résoudre le consensus (minimalité) avec **une majorité de processus corrects**
- FLP => Impossible à implémenter en asynchrone

Leader ultime

- Détecteur de défaillances Ω

Ω : un détecteur de défaillances dont la sortie est un unique processus supposé être correct

q est la sortie de Ω à l'instant t:

p fait confiance à q à l'instant t

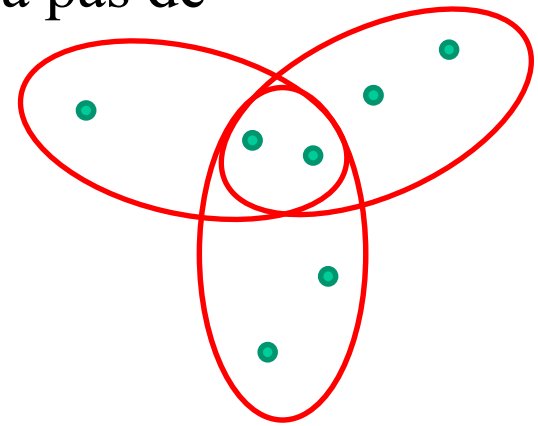
Ω assure :

un jour tous les processus corrects feront confiance au *même* processus *correct*.

Ω et <> S équivalent

Détecteur minimum avec n-1 fautes

- Détecteur de fautes $\sum$
  - Propose une liste de processus corrects
    - *Intersection* : Il y a toujours une intersection non vide dans les listes proposées par $\sum$
    - *Complétude ultime* : ultimement, il n'y a pas de processus fautif dans les listes

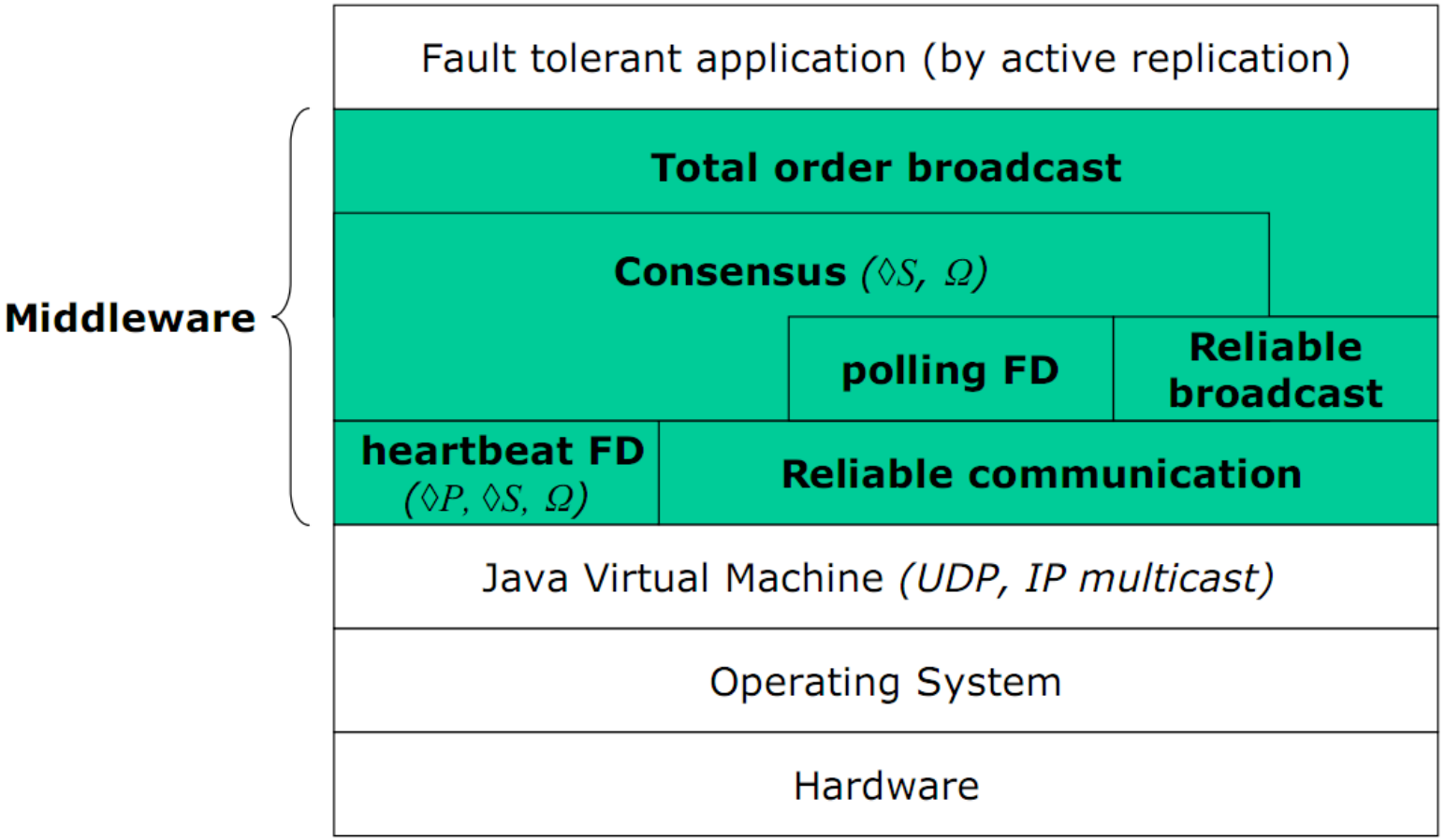


- (Ω , $\sum$) le plus faible détecteur pour réaliser le consensus avec n-1 fautes

FD mininum

Problems / Models	**Consensus**	k-set agreement	set agreement	Eventual consistency
Shared memory	Ω [LH94]	k-anti-Ω [GK09]	anti-Ω [Z10]	
Message passing	(Ω,Σ) [DFG10]	?	$\mathcal{L}$ [DFGT08]	Ω [DKGPS15]

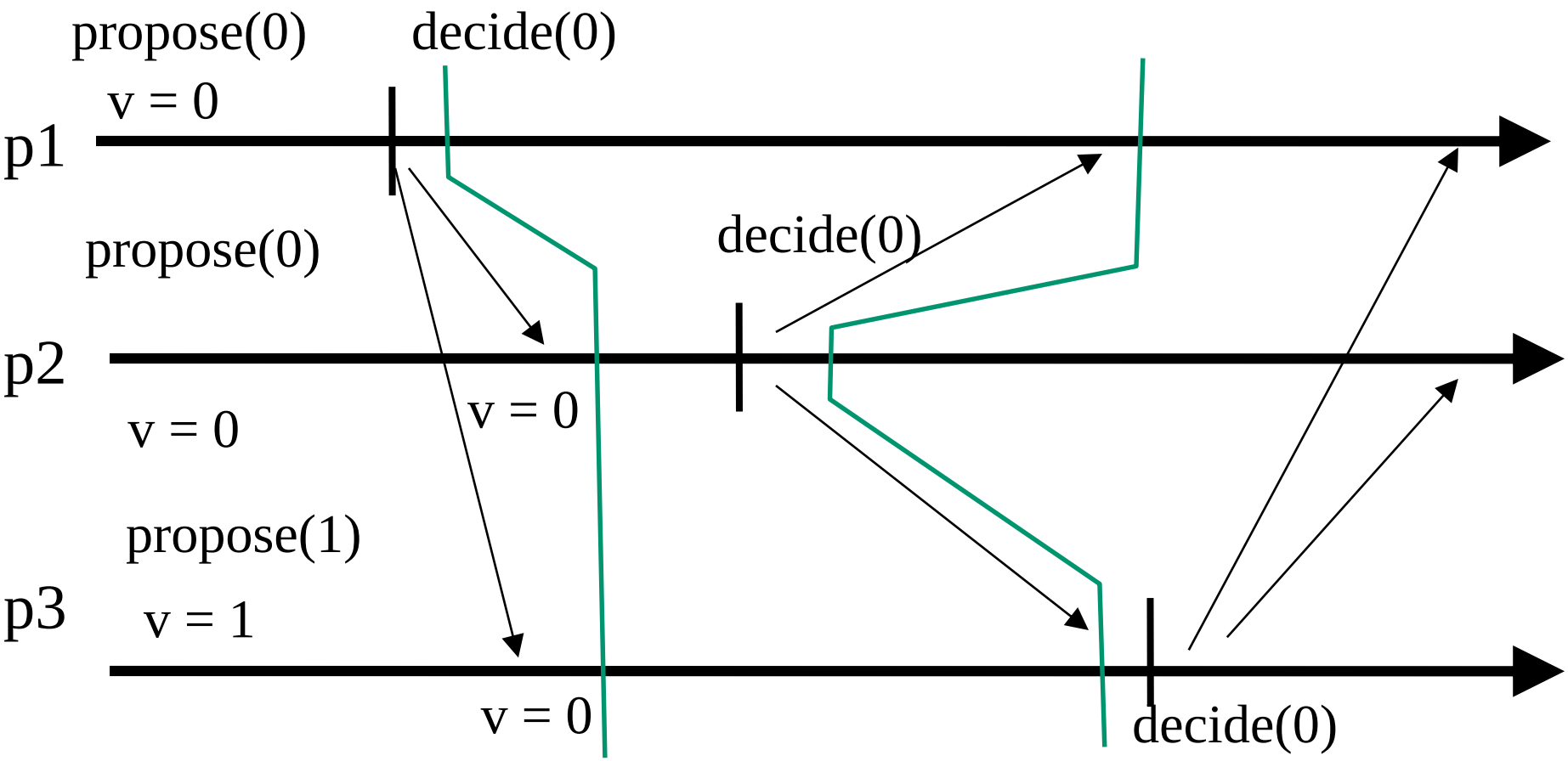
Fault-tolerant Architecture



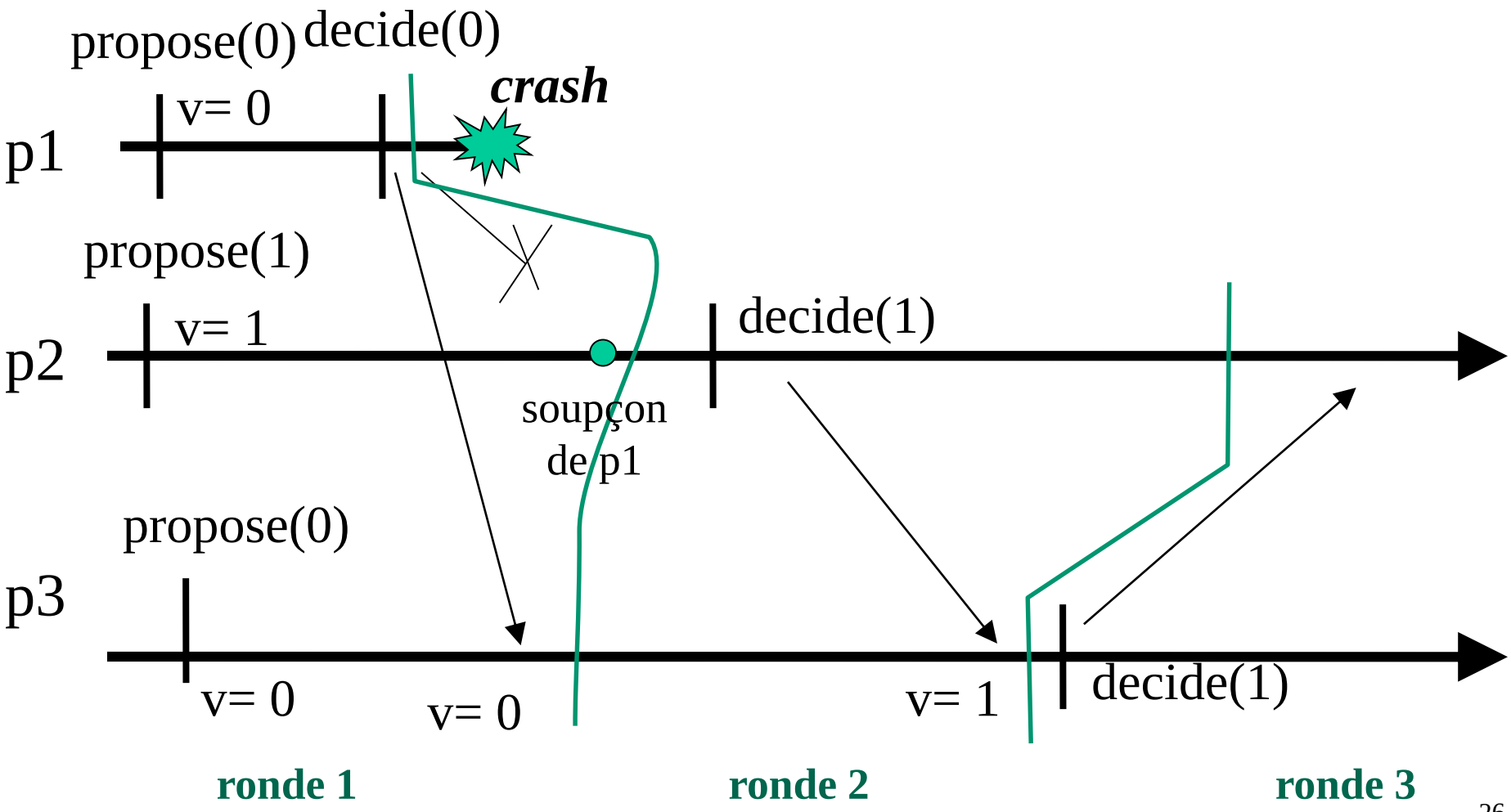
Consensus avec P – Algorithme 1

- Fondé sur des rondes et un leader
- Les processus exécutent des rondes de manière incrémentale (1 à *n*)
- Dans chaque ronde : le processus dont l'id correspond au numéro de ronde est le leader (id leader = id ronde % N)
- Le leader choisit sa valeur courante, la décide et la diffuse à tous.
- Les "non" leader (id node != id ronde % N) attendent :
  - (a) la réception du message du leader pour choisir sa valeur
  - (b) la suspicion du leader
- En *n* rondes tous les processus ont décidé (tous ont été leaders)

Algorithme 1 : Exemple



Algorithme 1 : Exemple (2)



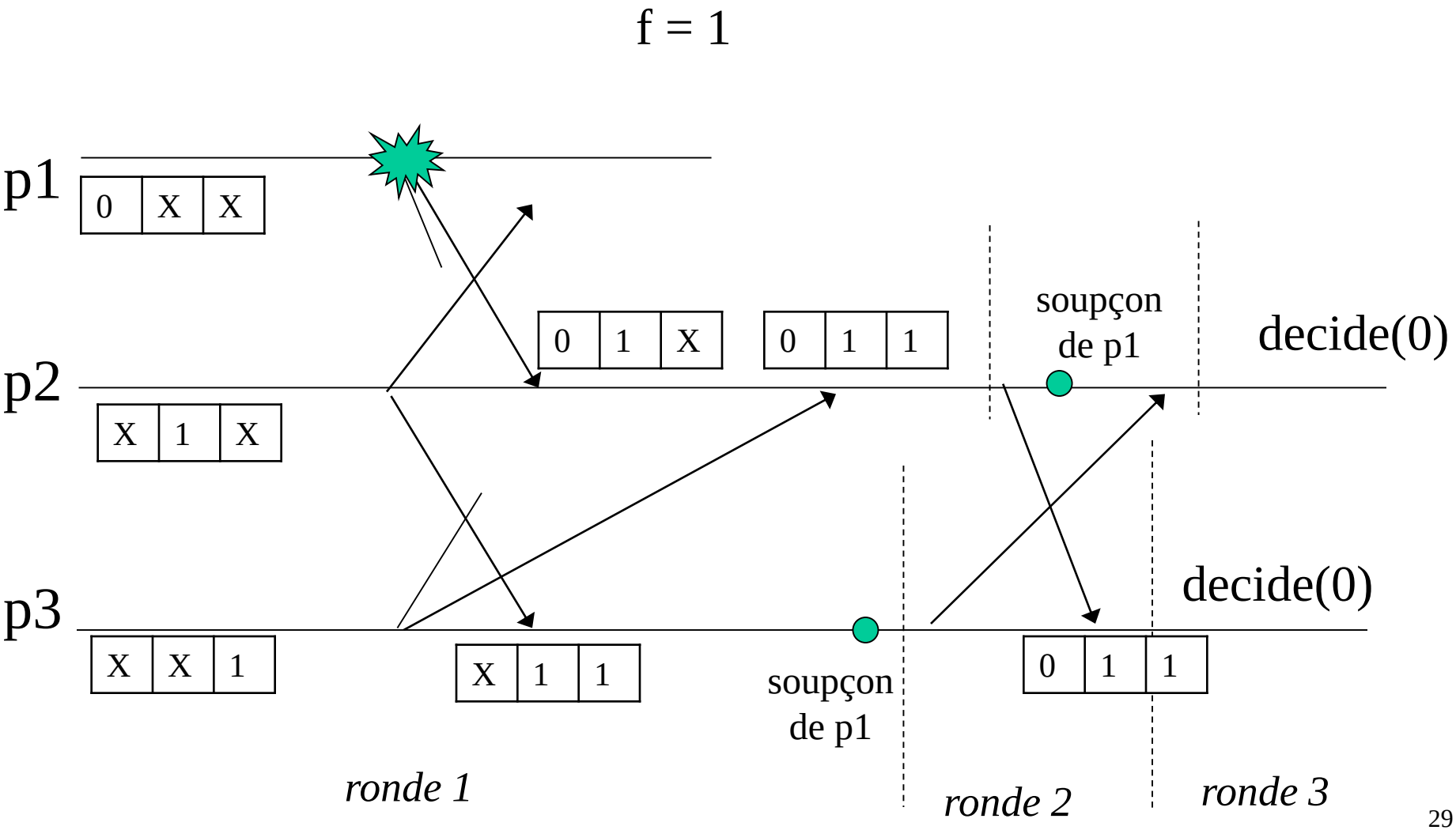
Autres consensus sur P

- Attendre les N rondes pour décider [CT 91, DLS 88]
- Attendre f+1 rondes pour décider : Algorithme 2

Consensus avec P : Algorithme 2

- f = nombre maximum de fautes tolérées
- Chaque processus P_i maintient un vecteur V_i pour stocker les valeurs proposées
- $f+1$ rondes :
  - Chaque processus P_i diffuse V_i de façon incrémentale
  - Attendre la réception des vecteurs de tous les processus non suspectés
- Après $f+1$ rondes (tours) :
  - P_i choisit et décide **la première valeur non vide** de son vecteur

Algorithme 2 : Exemple



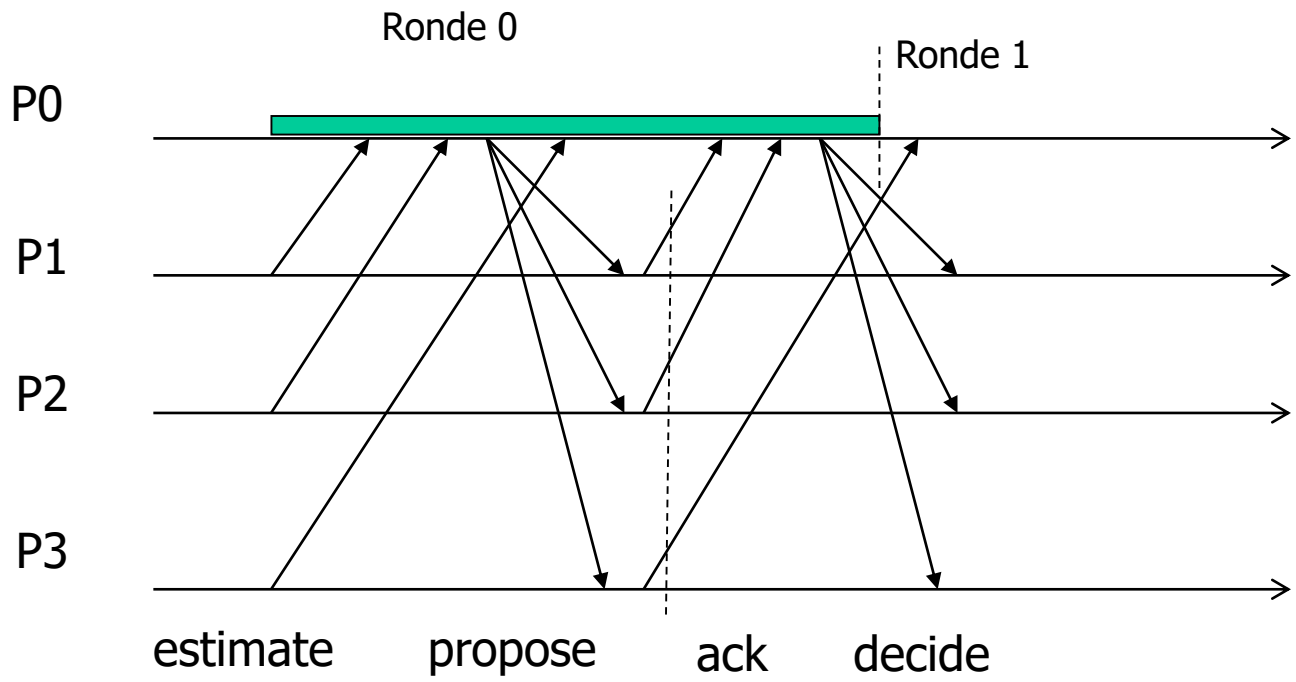
Consensus avec <>S

- Algorithme du coordinateur tournant [CHT 96]
- $f < n/2$ crashs
- Processus numérotés *1, 2, …, n*
- Exécution de *rondes asynchrones*
- Ronde r, coordinateur = processus $(r \bmod n) + 1$
- Le coordinateur c :
  - Impose sa valeur v
  - v est choisie si c n'est pas suspecté

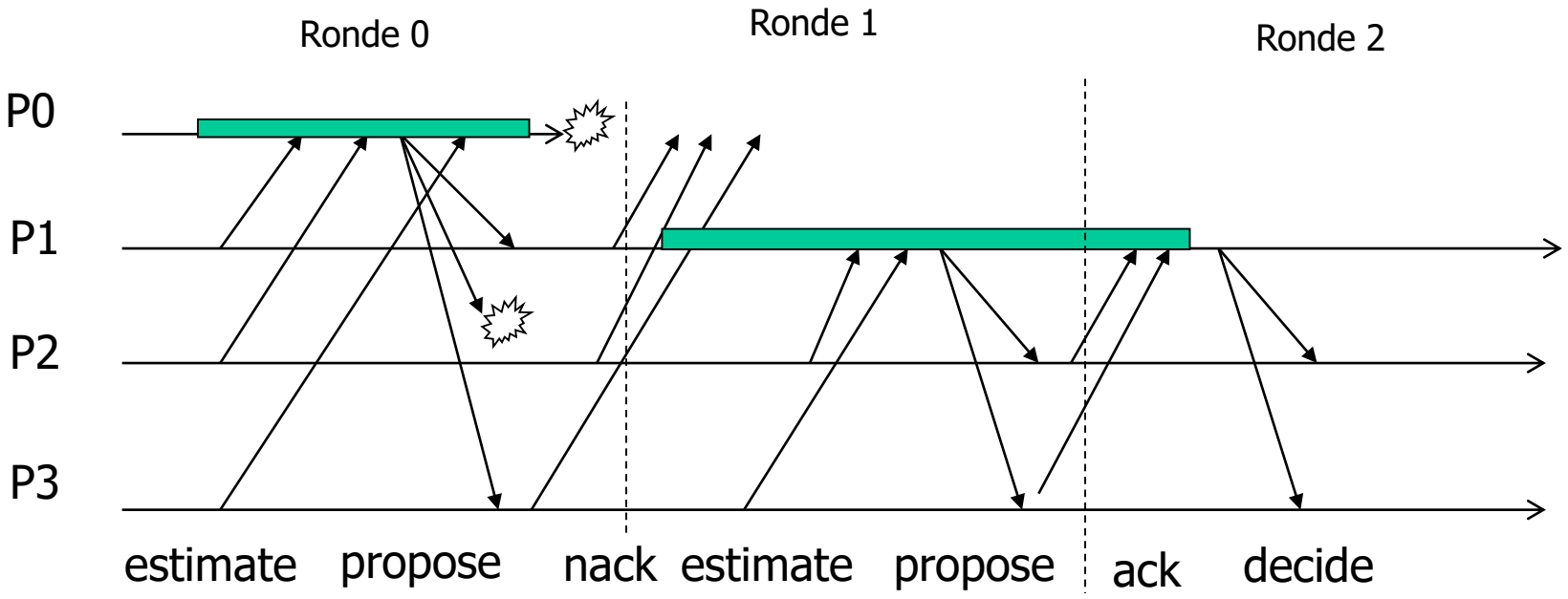
Consensus avec <>S (2)

- 4 phases par ronde
- **Phase 1** : chaque processus envoie au coordinateur sa valeur courante estampillée par la ronde de sa dernière mise à jour.
- **Phase 2** : le coordinateur réunit une majorité de valeurs,
  valeur estimée = une valeur parmi les plus à jour,
  diffusion de la valeur estimée
- **Phase 3** : Pour chaque processus correct :
  - Réception de la valeur estimée : renvoyer ack au coordinateur, mise à jour de la valeur courante
  - Soupçon du coordinateur : renvoyer nack
- **Phase 4** :
  - Coordinateur reçoit une majorité de réponses (ack-nack) :
    si majorité de ack
    valeur finale = valeur estimé + diffusion fiable valeur
    A la réception tous les processus décident la valeur reçue
  - Si pas de majorité : changement de ronde

Coordinateur tournant : Exemple

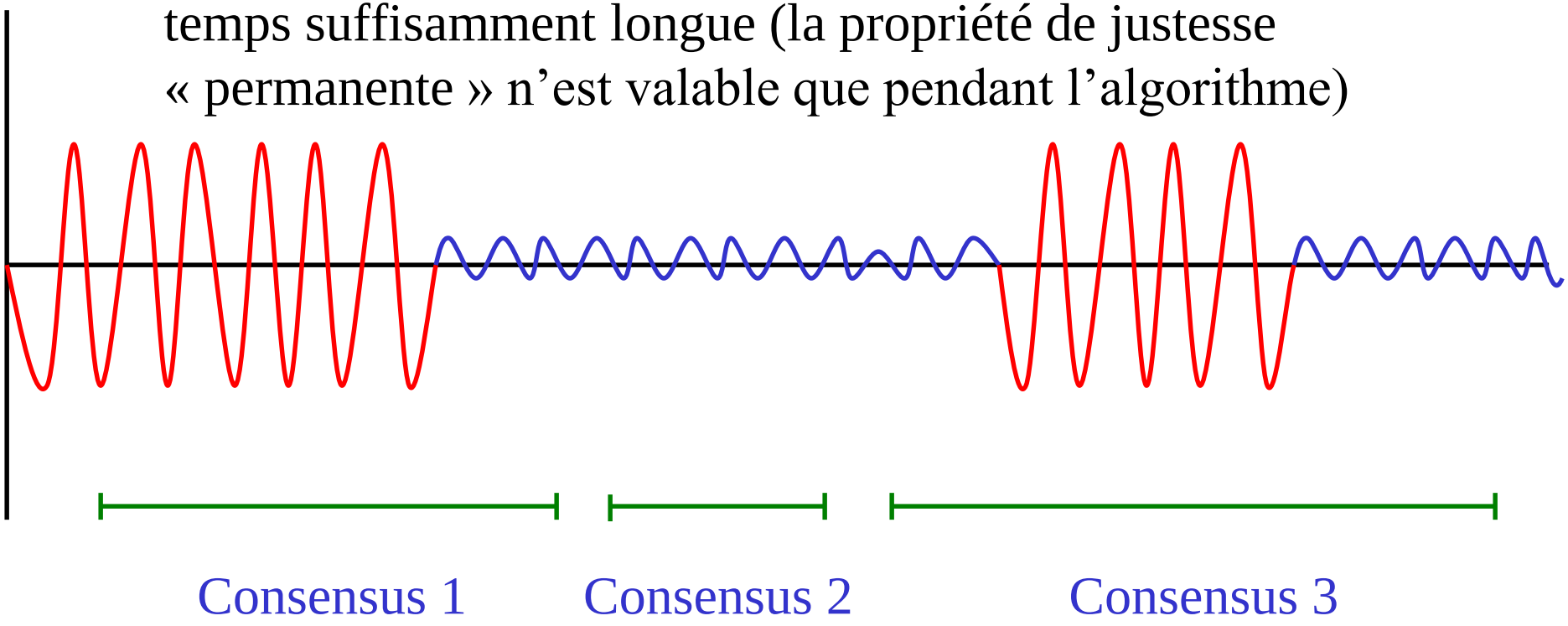


Coordinateur tournant : Exemple (2)



Comportement en cas d'instabilité

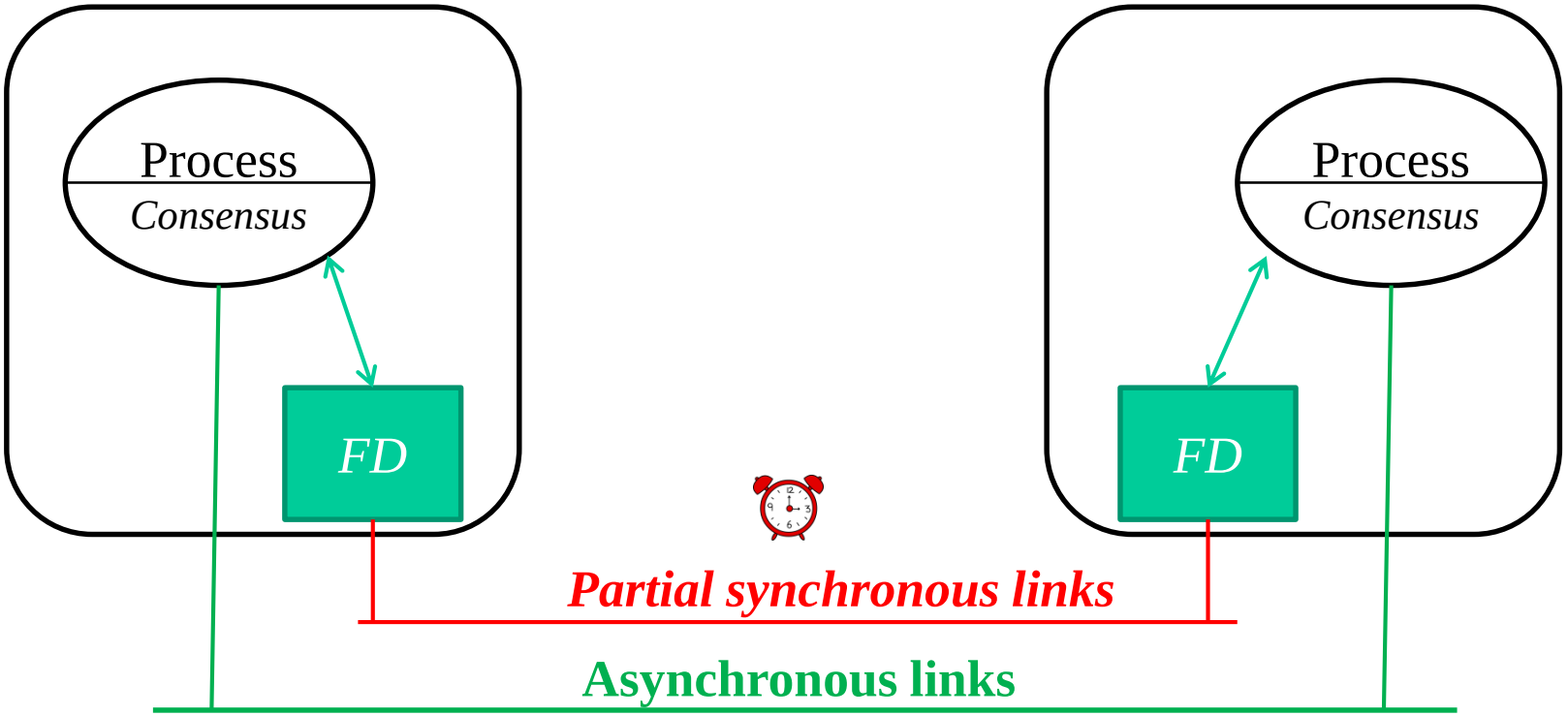
- Il suffit que le détecteur se stabilise pendant une période de temps suffisamment longue (la propriété de justesse « permanente » n'est valable que pendant l'algorithme)



Mise en œuvre des détecteurs de fautes

- Métriques
- Modèles temporels
- Implémentation
- Exemple : Passage à l’échelle

Implementation des FDs



Hypothèses temporelles (1)

- Implémentations reposant sur des temporisateurs :
  - Systèmes partiellement synchrones

Pour <> P (à terme, plus d'erreur)

  - Il existe un temps (GST : Global stabilization time) où il y a une borne inconnue sur les délais de transmission et de traitement des messages (Modèle M3 [CHT 96])

$\Rightarrow$ Permet d'implémenter <>P

Idée : à chaque erreur on augmente son temporisateur

$\Rightarrow$ Il existe un moment (après GST) où on ne fera plus d'erreur (le temporisateur a atteint la borne inconnue)

Hypothèses temporelles (2)

Pour <> S et Ω (à terme plus d'erreur sur 1 processus) :

hypothèse réduite à un ensemble de canaux ultimement synchrones (lien ultimement ponctuel <>-timely)

Définition: p est une ◊j-source: au moins j liens sortant de p sont ultimement ponctuels

Attention: la borne n'est pas connue

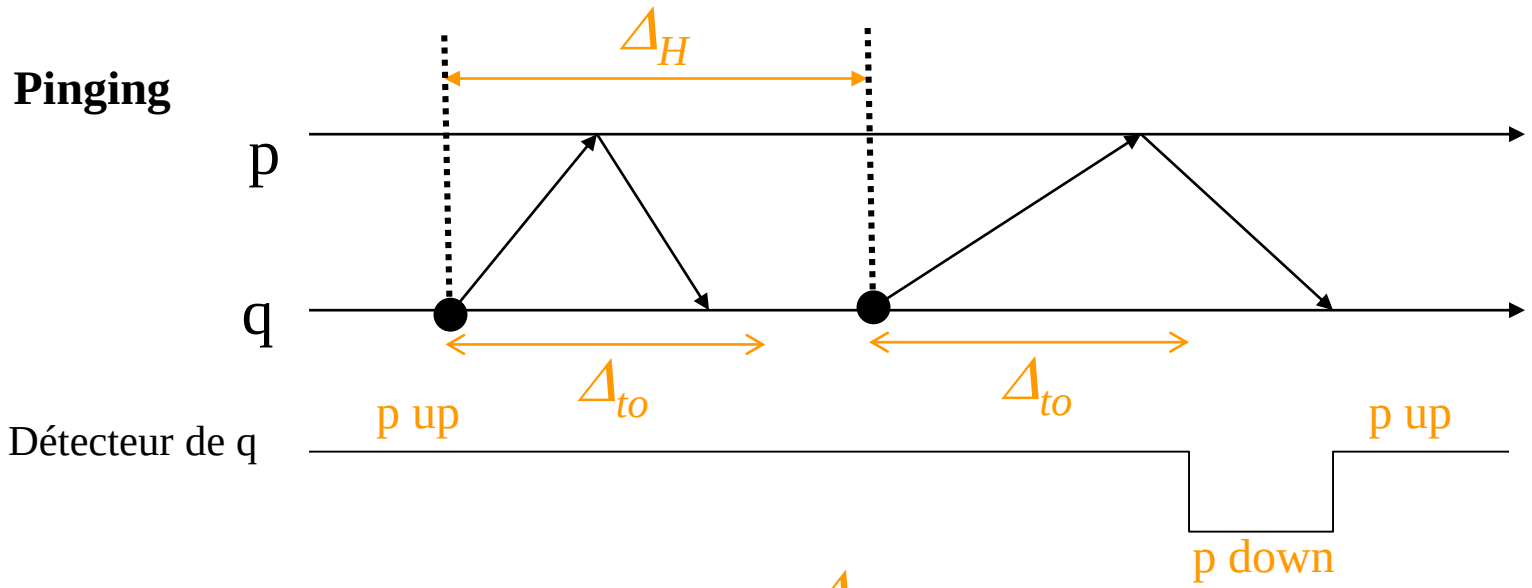
Ω peut être implémenté si au il y a au moins une ◊-f-source correcte (f = nombre de défaillants)

Hypothèses temporelles (3)

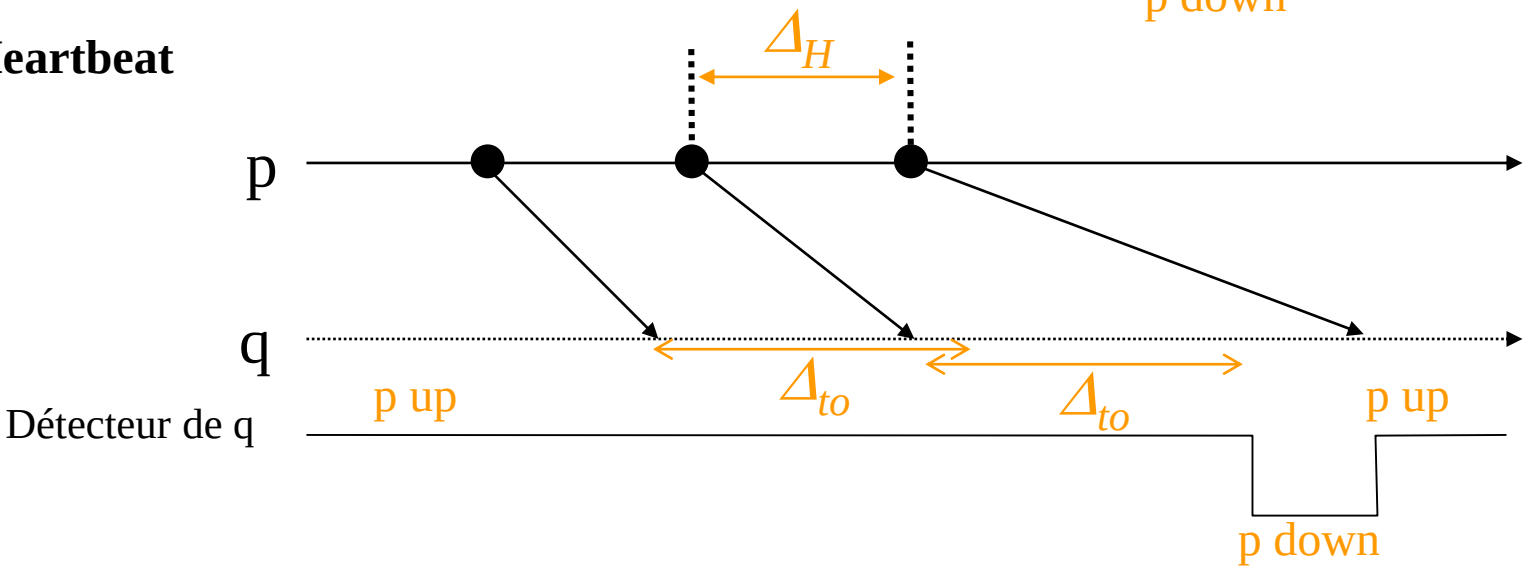
- Implémentations asynchrones (sans temporisateur):
  - Basées sur query-response (attendre un nombre fini de réponses – n - f)
  - Connaissance a priori du nombre de processus défaillants (f)
  - Hypothèse relative sur des canaux de communication (canaux plus rapides que d'autres)

Techniques d'implémentation

- **Pinging**

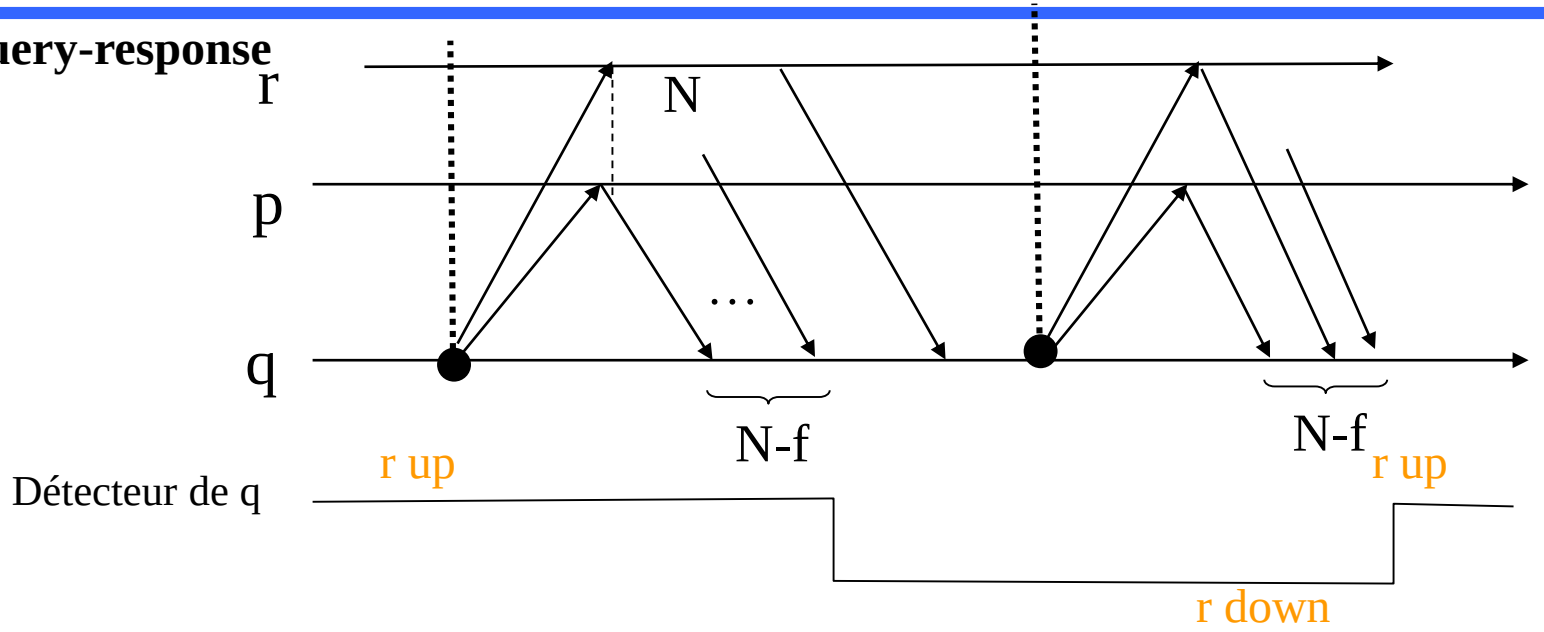


- **Heartbeat**

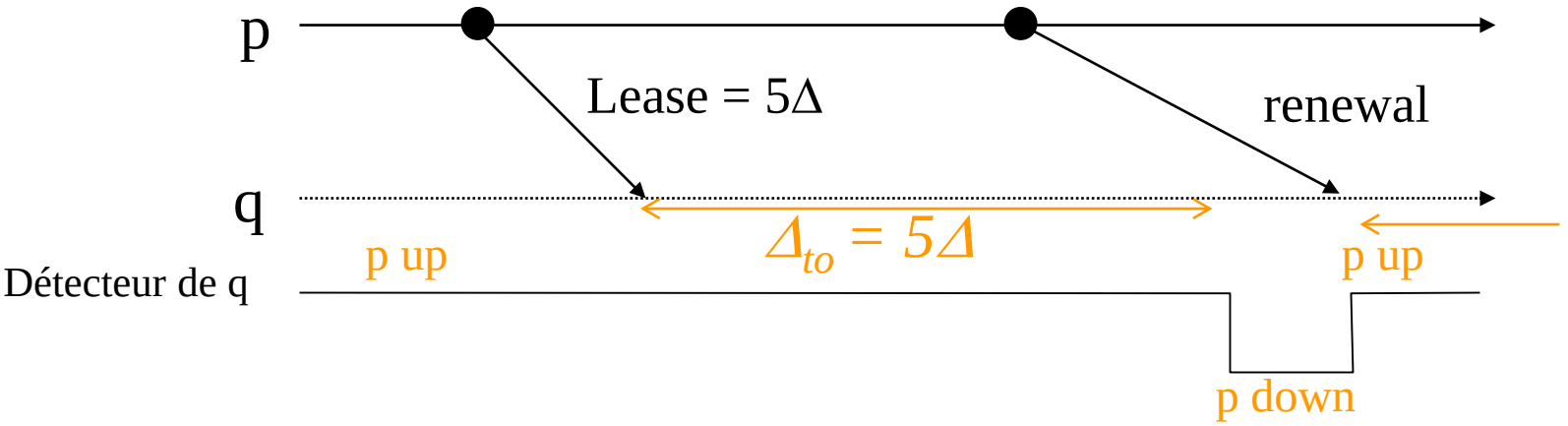


Implementations (2)

- **Query-response**

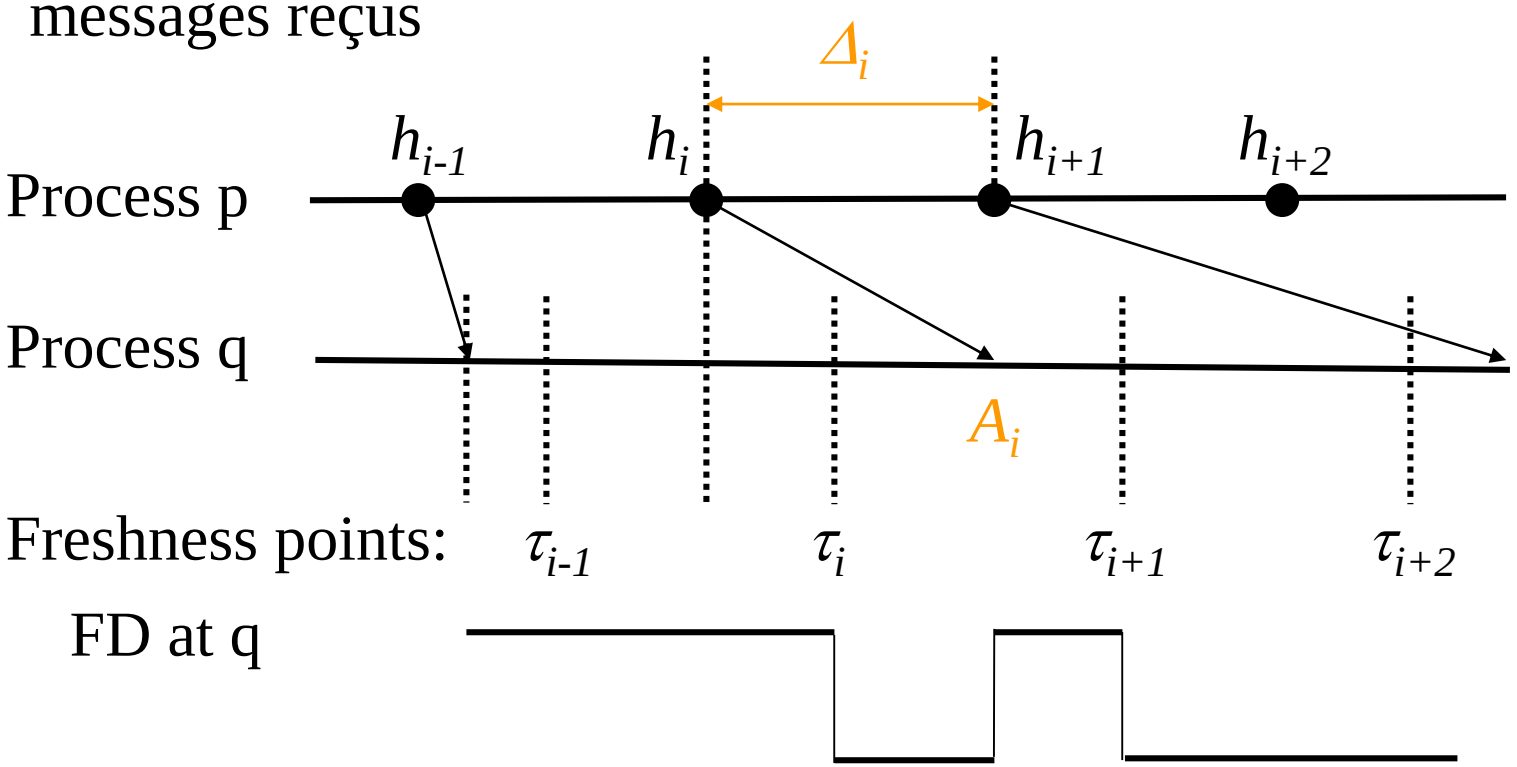


- **Lease**



Estimation de Chen et al.

Idée : Définition de la date de réception du prochain heartbeat (freshness point) en fonction des N derniers messages reçus



Estimation du temporisateur

- Prochain temporisateur :
  - *Timeout* (τ_{k+1})= date (EA_{k+1}) + marge de sécurité (α_{k+1})
- Date : Chen's estimation

$$EA_{(k+1)} \approx \frac{1}{n}\left(\sum_{i=k-n}^{k} A_i - \Delta_i.i \right) + (k+1).\Delta_i$$

- Marge : Fixe (Chen et al.)

  Dynamique (Bertier et al. – Basé sur RTT)

$$\begin{aligned} error_{(k)} &= A_k - EA_{(k)} - delay_{(k)} \\ delay_{(k+1)} &= delay_{(k)} + \gamma.error_{(k)} \\ var_{(k+1)} &= var_{(k)} + \gamma.(|error_{(k)}| - var_{(k)}) \\ \alpha_{(k+1)} &= \beta.delay_{(k+1)} + \phi.var_{(k+1)} \end{aligned}$$

Eventual Perfect FD (◊P)

Hypothèse : Le système devient synchrone après un temps GTS (inconnu)

Initialisation: $suspected_i = \{\}$; $\forall\ j \neq i \in \{1, .., n\}\ \Delta_{i,j} = \Delta_0$

Task 1: repeat every Δ

send HEARTBEAT to all – $\{p_i\}$

Task 2 : when did not receive HEARTBEAT during last $\Delta_{i,j}$ from p_j

$suspected_i = suspected_i \cup \{ p_j\}$

Task 3: when received HEARTBEAT from p_j and $p_j \in suspected_i$

$suspected_i = suspected_i - \{ p_j\}$

$\Delta_{i,j} = \Delta_{i,j} + 1$

Algorithme Ω

Hypothèse une <>f-source correct

on initialization :
$\forall q \neq p : Timeout[q] \leftarrow \Delta_H + 1$
$\forall q : counter[q] \leftarrow 0, suspect[q] \leftarrow \emptyset$
$\forall q \neq p$: reset $timer(q)$ to $Timeout[q]$
start tasks 0, 1 and 2

Idée : un vecteur (counter) qui compte le nombre de suspicions

incrémenté si un nœud est suspecté par n-f processus

task 0 :
repeat forever
$leader \leftarrow \ell$ such that $(counter[\ell], \ell) = \min \{(counter[q], q) : q \in \Pi\}$

task 1 :
repeat forever
send (ALIVE, $counter$) to all processes except p every Δ_H time

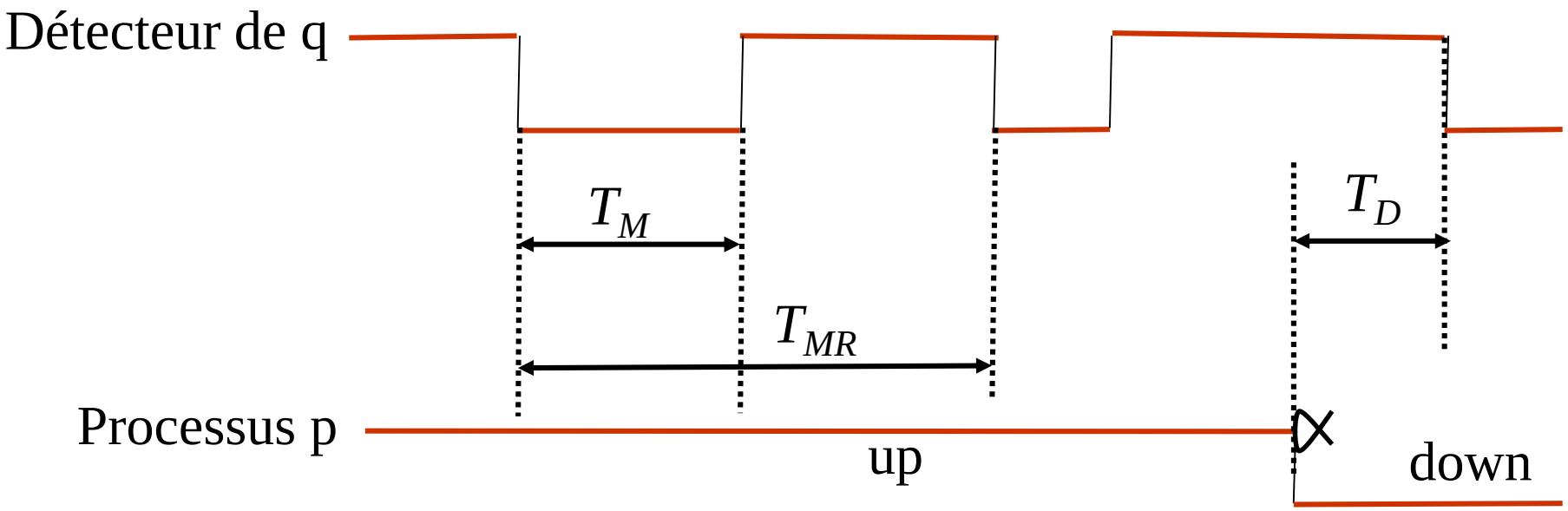
task 2 :
upon receive (ALIVE, c) from q **do**
for each $r \in \Pi$ **do** $counter[r] \leftarrow \max \{counter[r], c[r]\}$
reset $timer(q)$ to $Timeout[q]$

upon expiration of $timer(q)$ **do**
$Timeout[q] \leftarrow Timeout[q] + 1$
send (SUSPECT, q) to all
reset $timer(q)$ to $Timeout[q]$

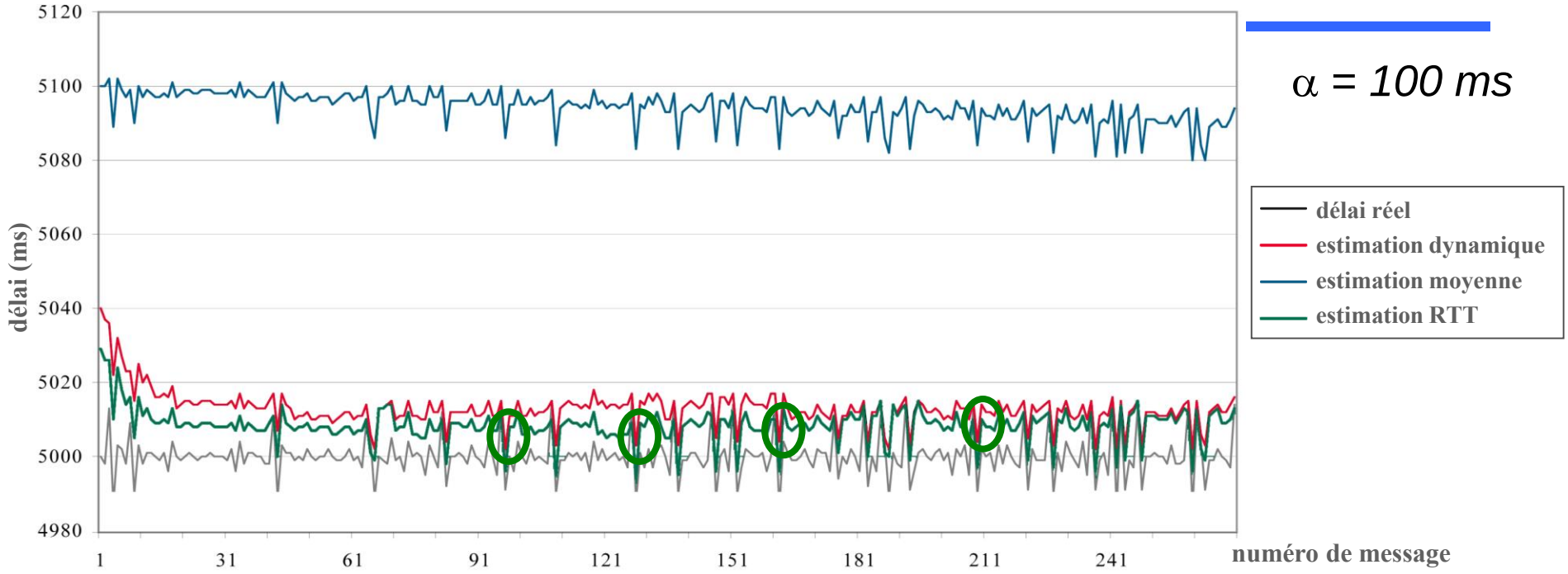
upon receive (SUSPECT, q) from r **do**
$suspect[q] \leftarrow suspect[q] \cup \{r\}$
if $|suspect[q]| \geq n - f$ **then**
$suspect[q] \leftarrow \emptyset$
$counter[q] \leftarrow counter[q] + 1$

Métriques : Qualité de détection (QoD)

- Complétude
  - Temps de détection (T_D)
- Justesse
  - Temps entre deux erreurs (T_{MR})
  - Durée des erreurs (T_M)



Performances



	Estimation RTT	Estimation de Chen	Estimation dynamique
Nombre de fausses détections	4	0	0
Temps de détection moyen (ms)	5011,9	5089,9	5016,6

Comparaison

Durée : 38 heures
Utilisation normale du laboratoire

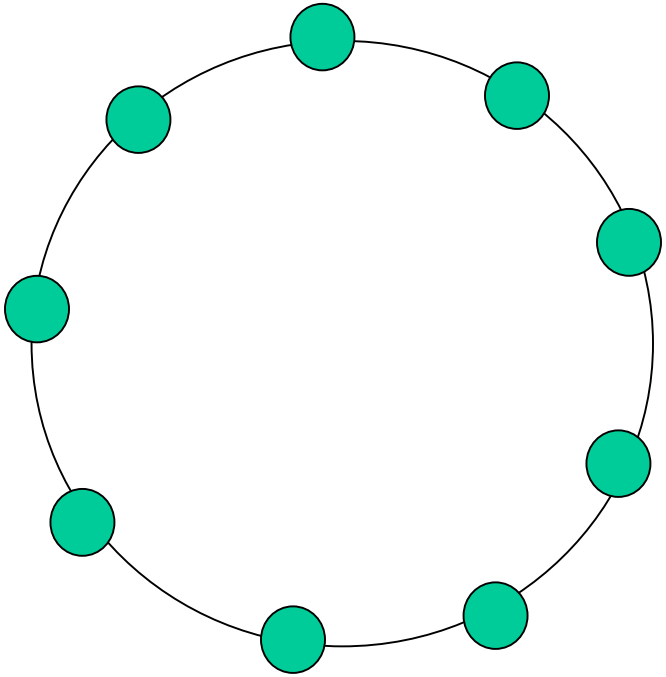
	Historic + RTT	RTT	Historic
False detection	24	54	29
Mistake duration (ms)	31,6	25,23	36,61
Detection time (ms)	5131,7	5081,79	5672,53

Passage à l’échelle des FD

Peu d'implementation à grande échelle

- Structure en anneau [WLL07] – PDC
- Approche probabiliste [GCG01] – PODC
- Organisation hierarchique [BMS03] – DSN
- **Hypercube pour HPC** [BBGHRS16] – Supercomputing

Structure en anneau



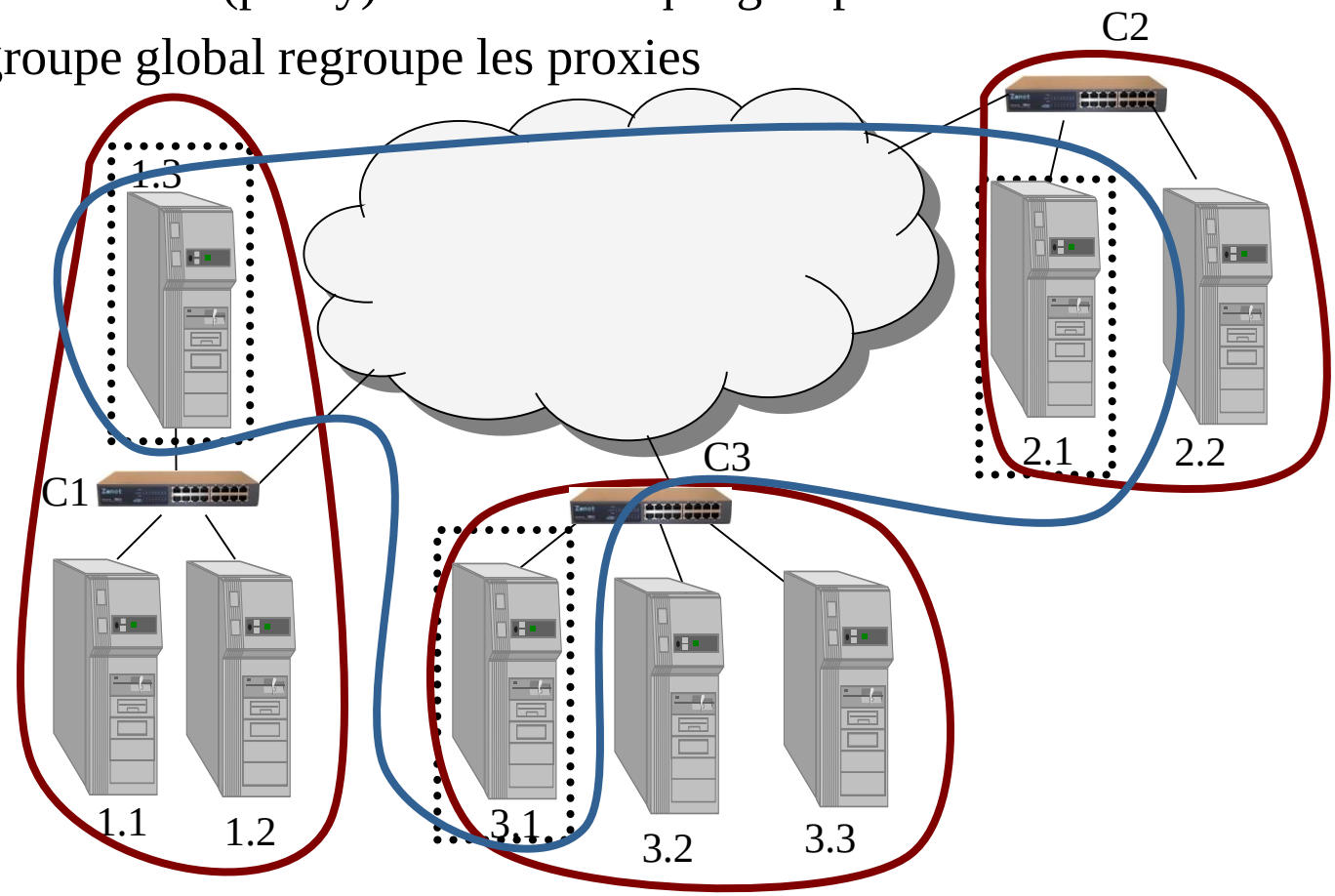
- Chaque noeud envoie un heartbeat à son successeur
- Faible coût en messages
- Temps de détection longs (propagation des information dans l'anneau)

Approche probabiliste

- A chaque ronde, chaque nœud choisit aléatoirement un nœud distant à surveiller
- Assure une complétude et justesse probabiliste
- Passe à l'échelle
- Difficulté pour dimmensionner les temporisateurs

Organisation hiérachique

- Cluster => groupe de détection local
- Un mandataire (proxy) élu dans chaque groupe local
- Un groupe global regroupe les proxies



Failure detection and propagation in HPC

George Bosilca[1], Aurélien Bouteiller[1], Amina Guermouche[1], Thomas Hérault[1], Yves Robert[1,2], Pierre Sens[3], and Jack Dongarra[1,4]

University Tennessee Knoxville[1]

ENS Lyon, France[2]

LIP6, Inria Paris, France[3]

University of Manchester, UK[4]

SC'16 – November, 2016

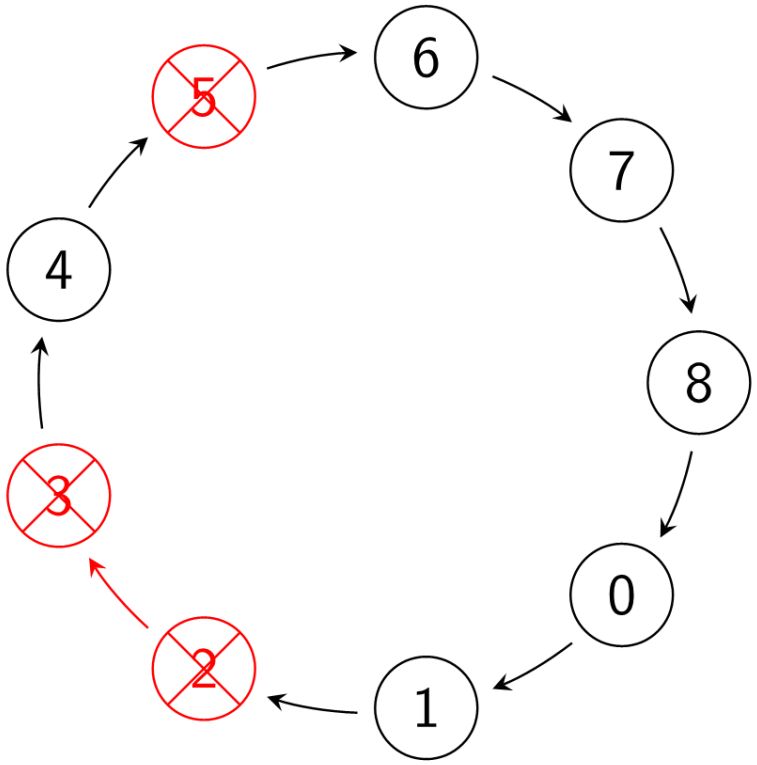
Resilient applications in HPC context

- Applications continue execution after crash of several nodes (nodes eventually occurs)
- Need rapid and global knowledge of group members
- 3 key features:

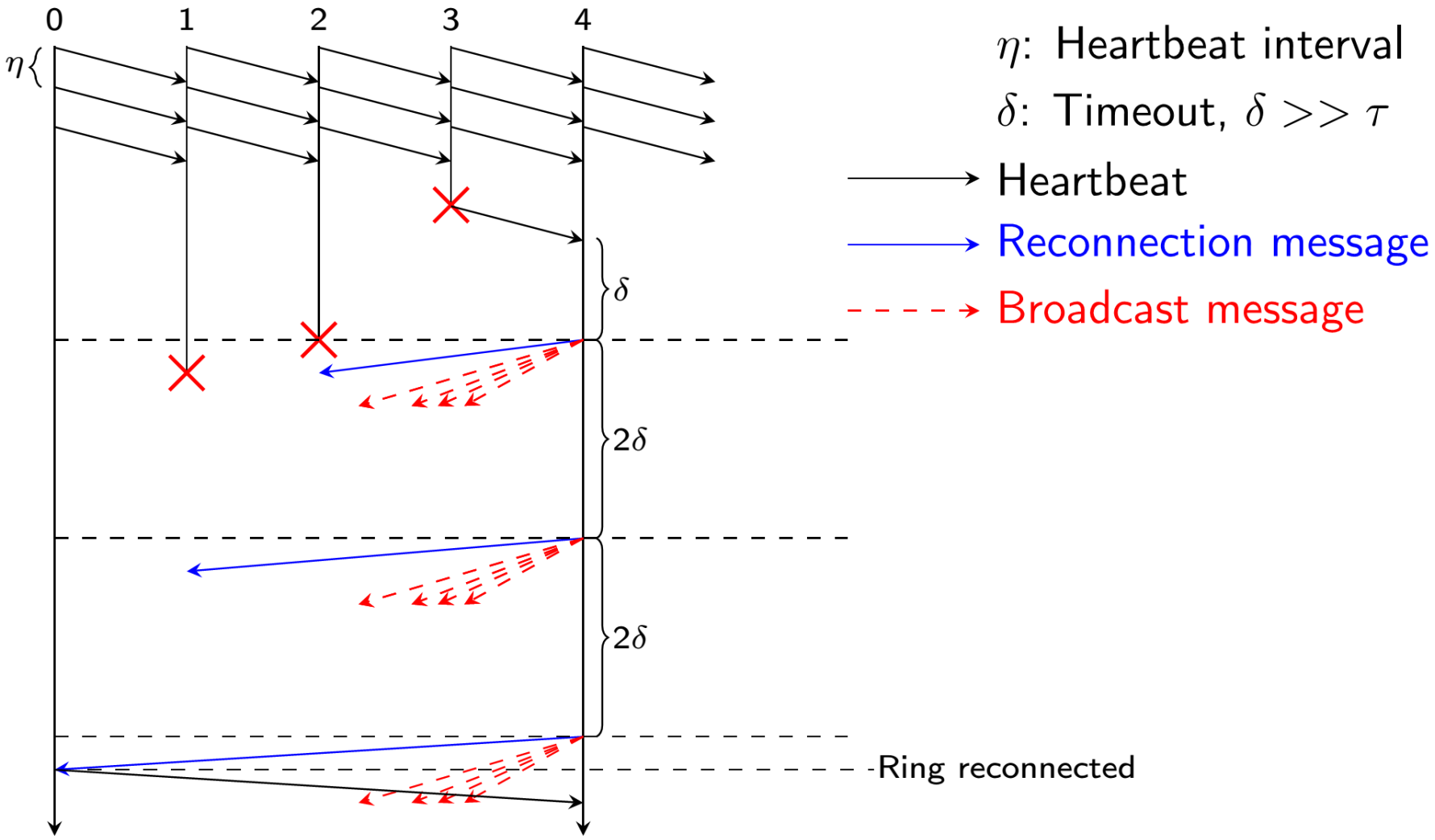
1. Rapid: failure detection
2. Global: failure knowledge propagation
3. Resilience mechanism should have minimal impact

Failure detection

- Processes arranged as a ring
- Periodic heartbeats from a node to its successor
- Maintain ring of live nodes
  → Reconnect ring after a failu
  → Inform all processes



Reconnecting the ring



η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

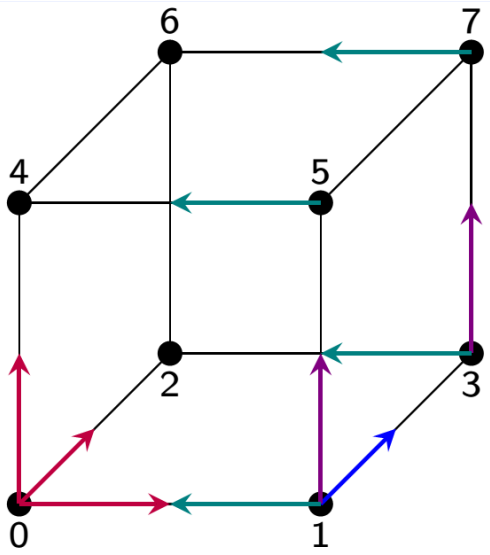
Reconnection message

Broadcast message

Broadcast algorithm

- Hypercube Broadcast Algorithm [1]
- Disjoint paths to deliver multiple broadcast message copies
- Completes if $f \leq \log(n) - 1$

(f : number of failures, n: number of live processes)



Receiver Node	Path via Node1	Node2	Node4
1	0	0-2-3	0-4-5
2	0-1-3	0	0-4-6
3	0-1	0-2	0-4-5-7
4	0-1-5	0-2-6	0
5	0-1	0-2-6-7	0-4
6	0-1-3-7	0-2	0-4
7	0-1-3	0-2-6	0-4-5

[1] P. Ramanathan and Kang G. Shin, 'Reliable Broadc Algorithm', IEEE Trans. Computers, 1988

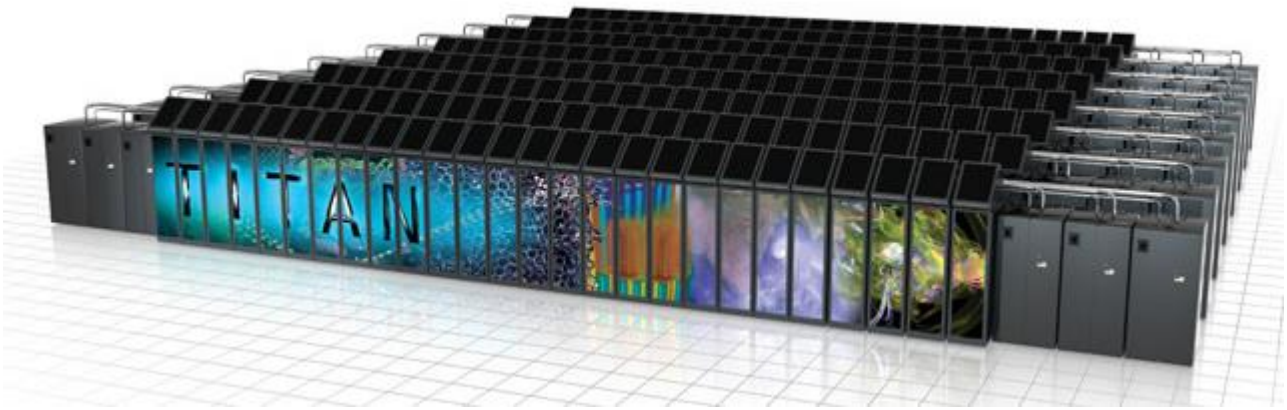
Implemention in MPI

- Implementation in ULFM / Open MPI
- Observation ring and propagation topology implemented in Byte Transport Layer (BTL)
- MPI internal thread independently from application communications
- RDMA put channel to directly raise a flag at receiver memory

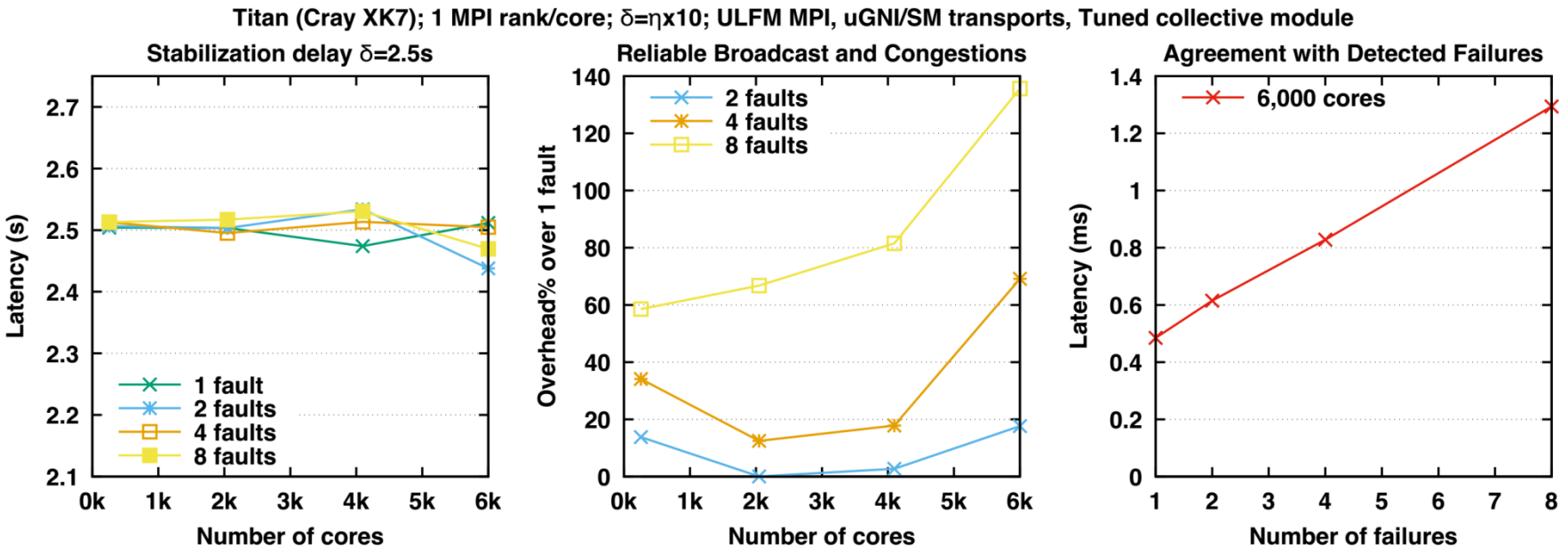
→ No allocated memory, no message wait queue

Experimental setup

- Titan ORNL Supercomputer
- 16-core AMD Opteron processors
- Cray Gemini interconnect
- ULFM / OpenMPI 2.x
- Up to 6, 000 cores
- Average of 30 times

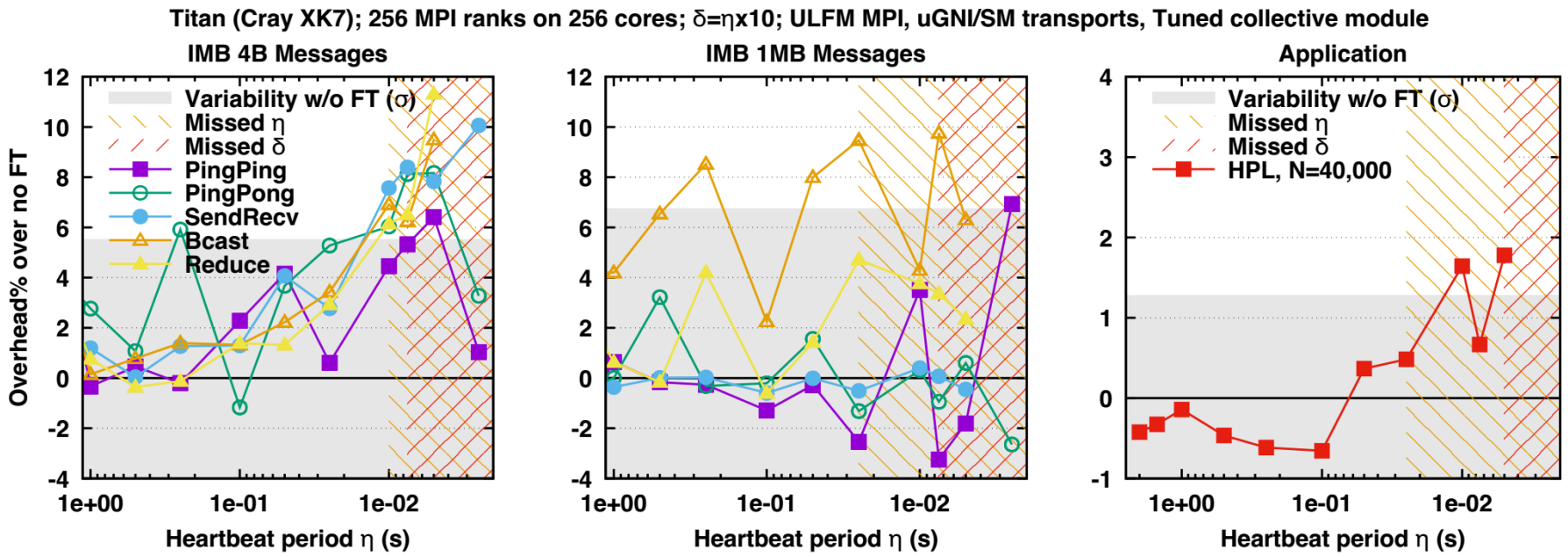


Detection and propagation delay



Noise

- $\delta = 10 \times \eta$



Conclusions

- Points forts des détecteurs non fiables :
  - Détection de défaillances comme abstraction : permet de s'abstraire de synchronisme
  - Détection rapide des fautes (aspect non fiable)
  - Permet de contourner FLP
- Points faibles des détecteurs :
  - Hypothèse de fiabilité des canaux => il existe des détecteurs supposant des canaux « équitable devant les fautes » (fair lossy channel :)
  - Propriété perpétuelle de justesse peu réaliste => perpétuité restreinte à la durée de l'algorithme
  - Modèle de fautes simple (crash) : extension vers fautes omission

Références

[BSM 03] M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchicalfailure detector. In Proceedings of the International Conference on Dependable Systems and Networks, june 2003.

[BBGHRS16] G. Bosilca, A. Bouteiller, A. Guermouche, T. Hérault, Y. Robert, P. Sens, J. Dongara. Failure Detection and Propagation in HPC systems. SC 2016 - The International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, United States, Nov 2016.

[Cha90] S. Chaudhuri. Agreement is harder than consensus : set consensus problems in totally asynchronous systems. In Proceedings of the ninth annual ACM symposium on Principles of distributed computing, pages 311--324. ACM Press, 1990.

[CHT 96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. Journal of the ACM, 43(4) :685722, 1996.

[CTA 00] W. Chen, S. toueg, M. K . Aguilera. On the quality of service of faulire detectors. In Proc. of First Conference on Dependable Systems and Networks, june 2000.

[DFG10] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui: Tight failure detection bounds on atomic object implementations. J. ACM 57(4): 22:1-22:32 (2010)

[DKGPS15] Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, Pierre Sens: The Weakest Failure Detector for Eventual Consistency. PODC 2015: 375-384

[DLS88] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. Journal of the ACM 35, 2 (Apr.), 288–323, 1988

DFGT08] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A. Tielmann: The Weakest Failure Detector for Message Passing Set-Agreement. DISC 2008: 109-120

[FLP 85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2) :374--382, apr 1985.

Références

[GCG01] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and ecient distributed failure detectors. In Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, pages 170179. ACM Press, 2001.

[LAF99] M. Larrea, S. Arévalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In Proceedings of the 13th International Symposium on Distributed Computing, pages 3448. Springer-Verlag, 1999.

[LFA00] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC 2000), pages 334334, NY, July 1619 2000. ACM Press.

[GK09]E. Gafni, P. Kuznetsov: The weakest failure detector for solving k-set agreement. PODC 2009: 83-91

[LH94] W-K. Lo, Vassos Hadzilacos: Using Failure Detectors to Solve Consensus in Asynchronous Sharde-Memory Systems (Extended Abstract). WDAG 1994: 280-295

[MMR03] A. Mostefaoui, E. Mourgaya, M. Raynal, "Asynchronous Implementation of Failure Detectors", Proc. Int'l IEEE Conf. Dependable Systems and Networks (DSN '03), pp. 351-360, 2003.

[WLL07] J. Wieland, M. Larrea, A. Lafuente, An evaluation of ring-based algorithms for the Eventually Perfect failure detector class. PDP 2007: 163-170

[Z10] P. Zielinski: Anti-Omega: the weakest failure detector for set agreement. Distributed Computing 22(5-6): 335-348 (2010)

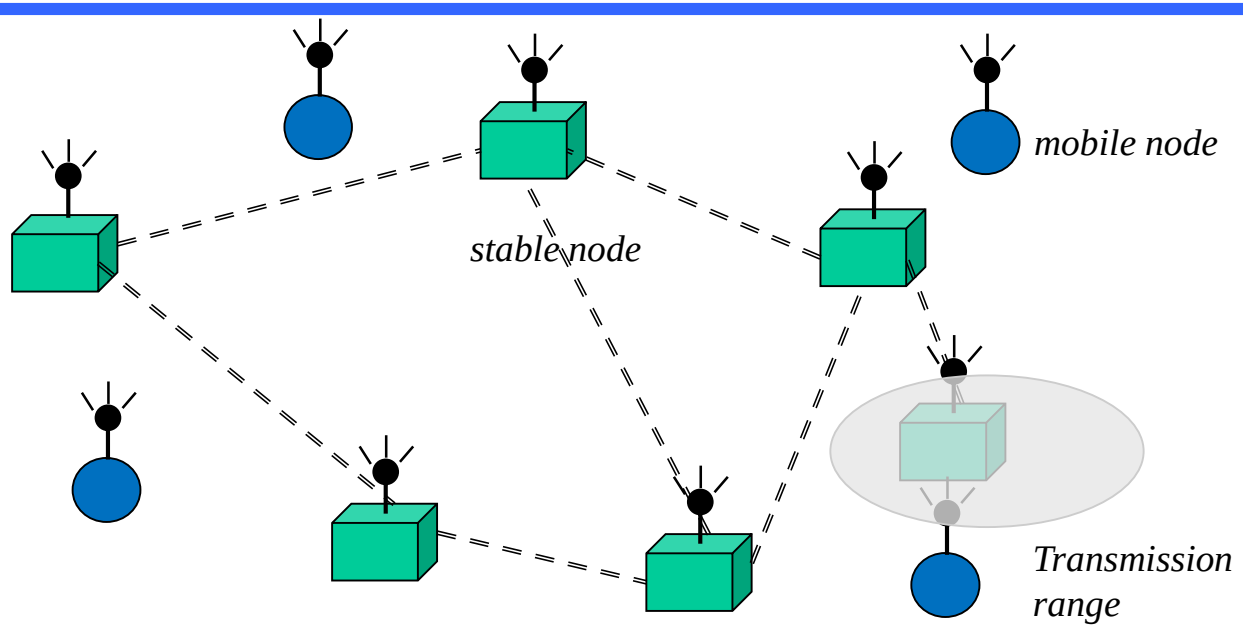
A Failure detector for Wireless Networks with Unknown Membership

Europar 2011

F. Greve, P. Sens, L. Arantes,V. Simon

UFBA / LIP6
INRIA / UPMC / CNRS

Features of dynamic networks



- Unknown set of nodes
- Dynamic graph due to mobility
- Communication via transmission range (broadcast to neighborhood)

Model : Processes

- $\pi = \{p_1, p_2, .., p_n\}$
- π and ***n*** are unknown (Unknown membership)
- Processes can crash or leave the system
- f = maximum number of failures in the neighborhood

Each process p_i maintains a partial knowledge of □ : $known_i$
(initiallly, $known_i = \{i\}$)

3 sets of processes :

STABLE (correct) : processes that eventually neither leave the system nor crash

FAULTY : processes that permanently crash

KNOWN : Processes known by at least one stable process.

Model : Communication

- Links :
  - Asynchronous links
  - Fair-lossy links
  - Broadcast in transmission range
- Connectivity :
  - Network : dynamic communication graph $G = (V,E)$ with $V = \pi$
  - For each node p_i, there is at least α_i correct neighbors ($\alpha_i = |Neighbors_i| - f_i$)
  - Eventually there is a path between every pair of stable (correct) processes

$\Diamond S^M$: Eventually strong FD with unknown membership

- Same properties of $\Diamond S$ FD restricted to known processes
- *Strong completeness* : every **known** and faulty process is eventually suspected
- *Eventual weak accuracy* : Eventually, at least one **known and stable** processes is never suspected

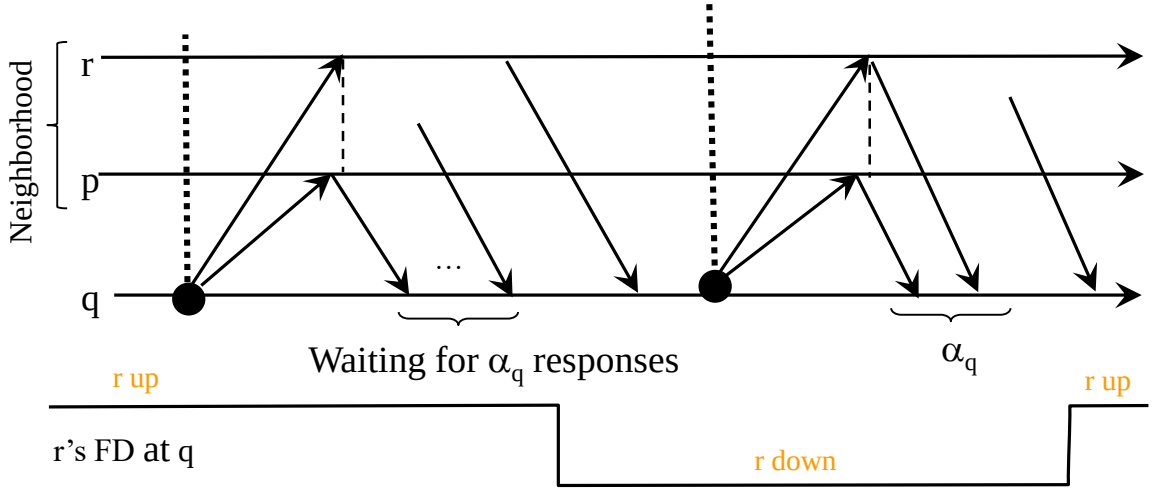
Algorithms (1)

- Principles :
  - Local detection of neighbor's failure based on **query-response** exchange
  - Flooding of failure information (suspected nodes and mistakes)

- Notations :
  - *susp*: set of processes suspected of being faulty
  - *mist.*: set of nodes which were previously suspected of being faulty but such suspicions are currently considered to be a mistake.
  - *known*: denotes the current knowledge of node about its neighborhood.

- suspected and mistake information is tagged by a local counter.

Algorithm : Sending of QUERY




```
Task T1:
Repeat forever
   broadcast QUERY(susp_i, mist_i)
   wait until RESPONSE received from ≥ α_i processes
   rec_from_i ← all p_j, a RESPONSE is received
   For all p_j ∈ known_i \ rec_from_i | ⟨p_j, −⟩ ∉ susp_i do
      If ⟨p_j, ct⟩ ∈ mist_i
         Add(susp_i, ⟨p_j, ct + 1⟩)
         mist_i = mist_i \ {⟨p_j, −⟩}
      Else
         Add(susp_i, ⟨p_j, 0⟩)
End repeat
```

Algorithm (2) : Reception of responses

```
Task T2:
Upon reception of QUERY (susp_j, mist_j) from p_j do
known_i ← known_i ∪ {p_j}
For all ⟨p_x, ct_x⟩ ∈ susp_j do
If ⟨p_x, −⟩ ∉ susp_i ∪ mist_i or (⟨p_x, ct⟩ ∈ susp_i ∪ mist_i and ct < ct_x)
    If p_x = p_i
        Add(mist_i, ⟨p_i, ct_x + 1⟩)
    Else
        Add(susp_i, ⟨p_x, ct_x⟩)
        mist_i = mist_i \ {⟨p_x, −⟩}
For all ⟨p_x, ct_x⟩ ∈ mist_j do
If ⟨p_x, −⟩ ∉ susp_i ∪ mist_i or (⟨p_x, ct⟩ ∈ susp_i ∪ mist_i and ct < ct_x)
    Add(mist_i, ⟨p_x, ct_x⟩)
    susp_i = susp_i \ {⟨p_x, −⟩}
    If (p_x ≠ p_j)
        known_i = known_i \ {p_x}
Send RESPONSE to p_j
```

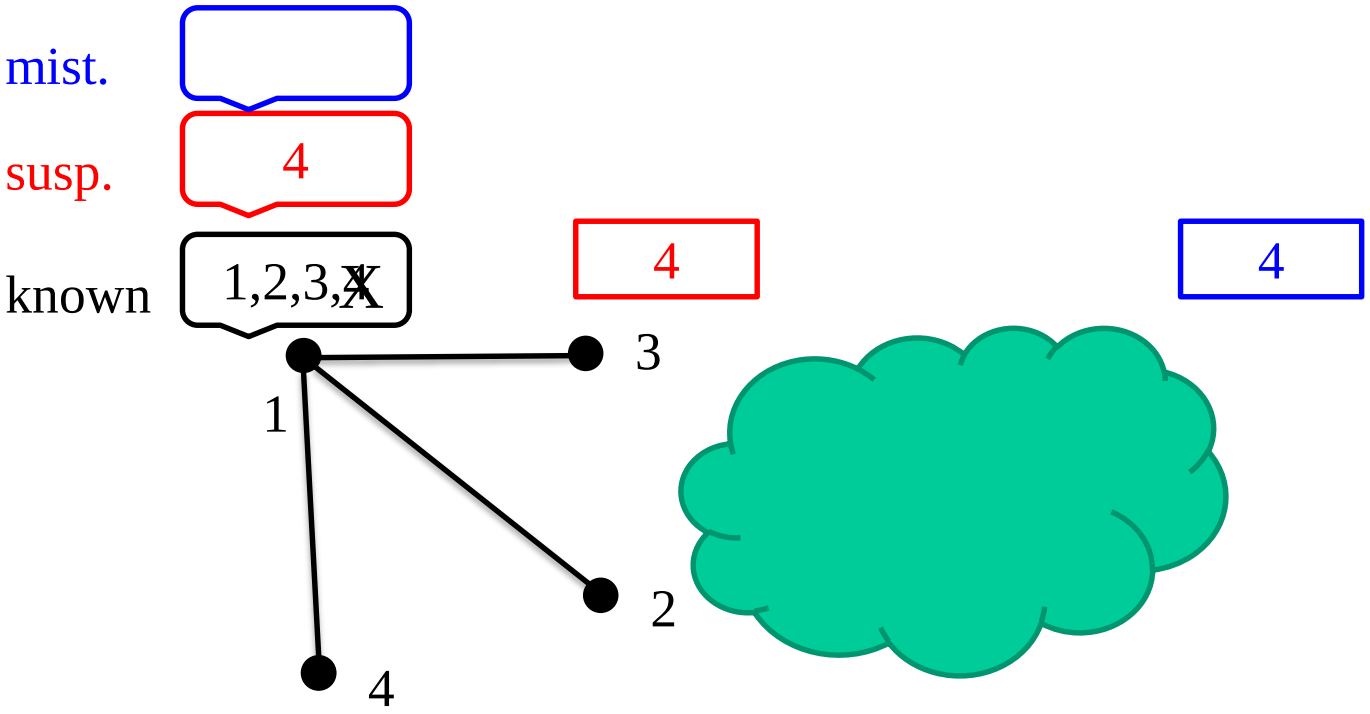
The receiver was suspected : generation of a mistakte

Update susp. set with more recent information

The sender is not the mistake node : suspicion of move

Update mist. set with more recent information

Exemple: Mobility of nodes



Properties to implement a $\Diamond S^M$ FD

1) Stable Termination Property (SatP): Each QUERY must be received by at least one stable and known node

⇒ *Necessary for the diffusion of the information*

2) Mobility Property (MobiP): In its new neighborhood, a moving node should have received a QUERY for at least one stable neighbor.

⇒ *Necessary to update of information*

3) Stabilized Responsiveness Property (SRP): eventually, one stable process p_i (a) always replies to a QUERY from any process p_j and (b) correct processes never leave neighborhood of p_i

⇒ *SRP should be hold for at least one stable known node*

Necessary for weak accuracy (eventually the "SRP node" will not be suspected)

Conclusion and Perspectives

- Model for dynamic networks :
  - Crash, Moving nodes, Churn
- Implementation of FD for dynamic networks:
  - Time-free approach
  - Based on local failure detection
- Definition of properties for ◊S
  - Membership
  - Minimum stability of moving nodes
  - Responsiveness
- Perspectives:
  - Model: A relaxed model with weaker connectivity assumptions