

Consensus et détection de fautes

Pierre Sens

Projet REGAL (LIP6/INRIA)

Pierre.Sens@lip6.fr

<http://regal.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

Spécification du consensus (1)

- Permettre l'accord entre processus

Initialement

1 valeur initiale par processus

Finalement

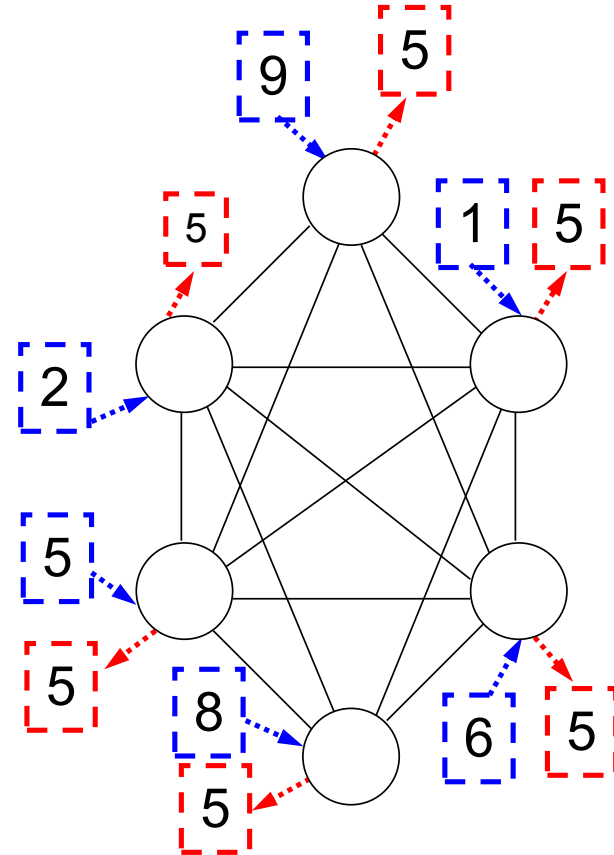
Tous les processus corrects
décident une même valeur

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]



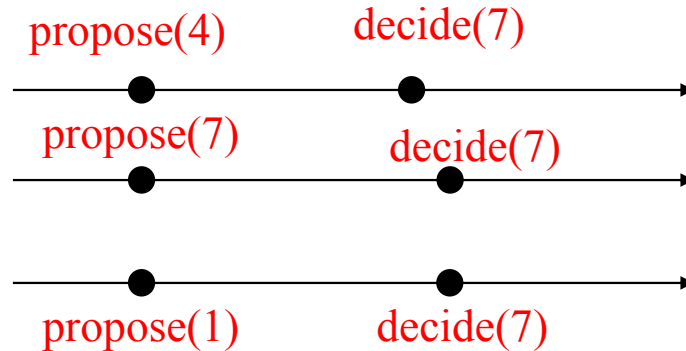
Spécification du consensus (2)

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

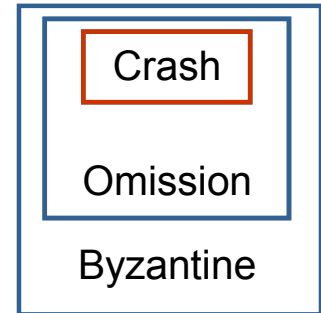
2 types de processus:

- **correct** : ne défaille pas pendant toute la durée de l'exécution
- **fautif** : pas correct
- Interconnexion :
 - $\Pi = \{p_1, p_2, \dots, p_N\}$ – 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 :

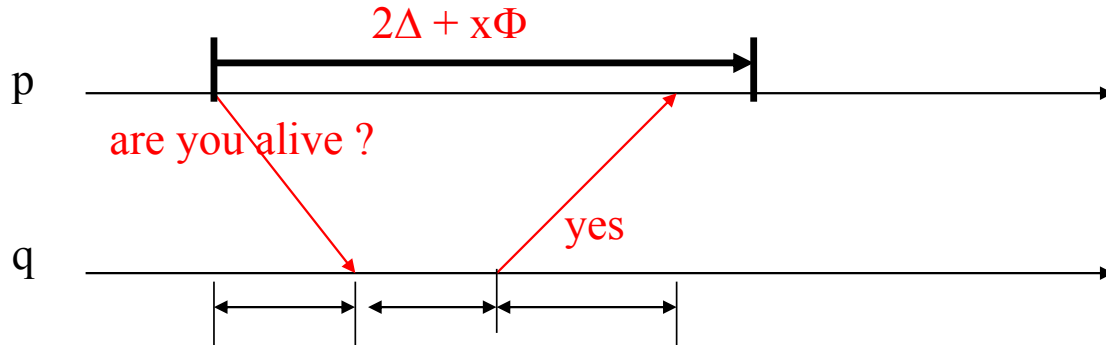
- **Fiable** (*reliable*): si p exécute `send (m)` vers q et *q est correct*, alors q recevra m
- **Quasi-fiable** (*quasi-reliable*): si p exécute `send (m)` vers q et *p et q sont corrects*, alors q recevra m
- **Equitable** (*fair-lossy*) : si un processus correct envoie un message m à q une infinité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

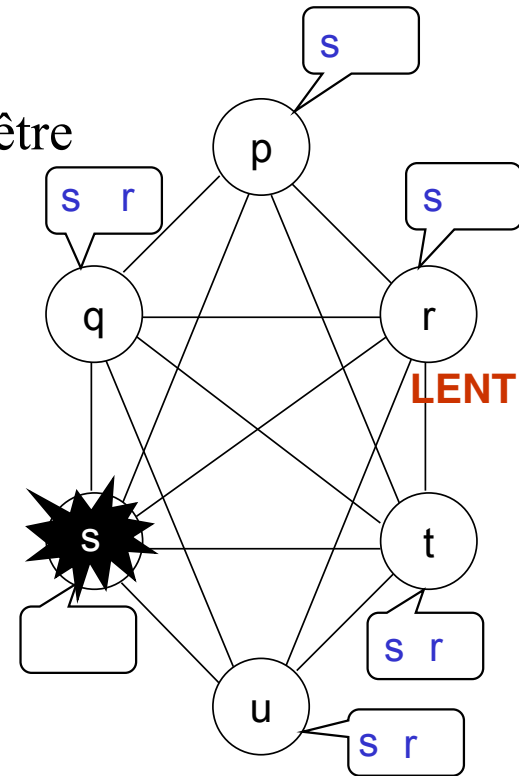
- **Changer le problème**
 - k -agreement [Cha90]
 - Plusieurs valeurs peuvent être décidées
- **Systèmes partiellement synchrones** [DDS87]
 - Les bornes sont non connues, valables uniquement à partir d'un moment
 - Variantes : alternance de bonne et mauvaise périodes
 - Borne restreinte à certains 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

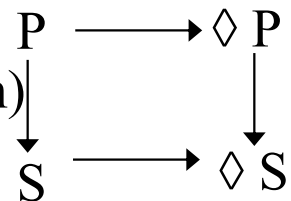
Classes de détecteurs

Hypothèses : pannes franches, communication fiable, réseau asynchrone

	Justesse			
	Forte	Faible	Finalement forte	Finalement faible
Complétude forte	P	S	$\diamond P$	$\diamond S$
Complétude faible	Q	W	$\diamond Q$	$\diamond W$

- Complétudes forte et faible sont équivalentes
(on peut construire une complétude forte à partir d'une faible)
 \Rightarrow 4 classes : P, S, $\diamond P$, $\diamond S$

- Force des détecteurs (\longrightarrow = implication)



Détecteur et consensus

- Consensus résoluble avec $\diamond S$
- $\diamond S$ le plus faible détecteur pour résoudre le consensus (minimalité)
- FLP \Rightarrow Impossible à implémenter en asynchrone

Autre détecteur

- 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 :

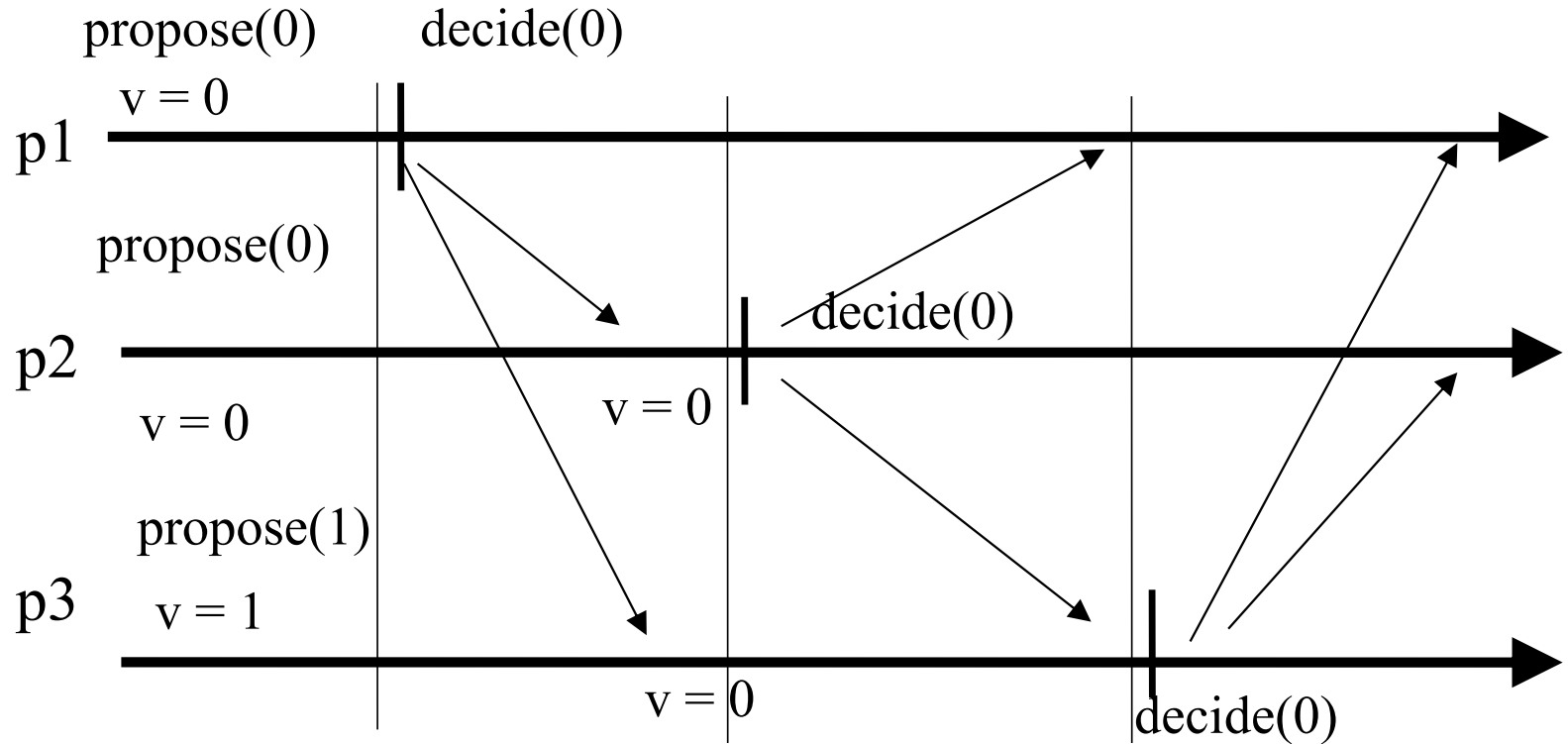
un jour tous les processus corrects feront confiance au *même* processus *correct*.

Ω et $\diamond S$ équivalent

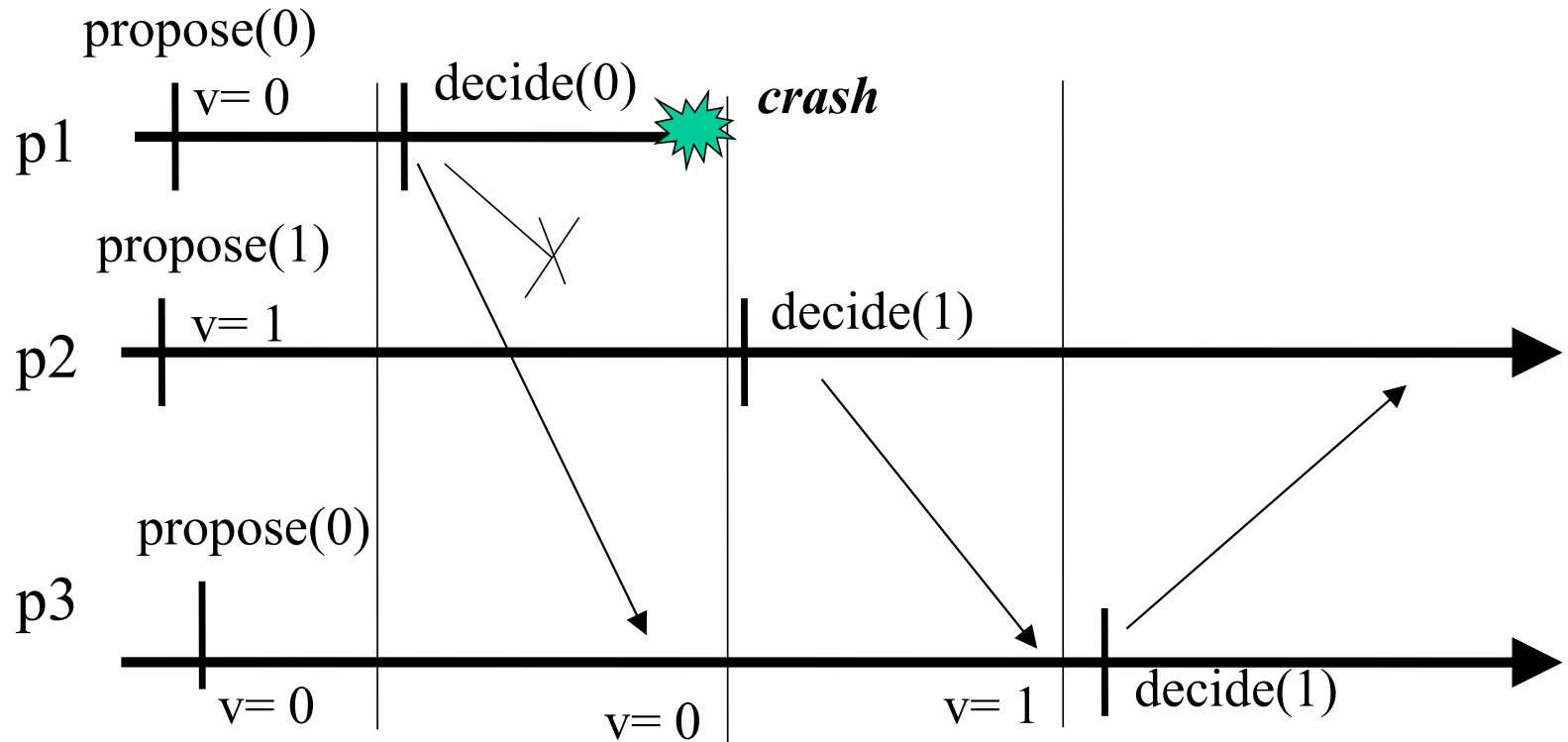
Consensus avec P – Algorithme 1

- 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 ($\text{id leader} = \text{id ronde} \% N$)
- Le leader choisit sa valeur courante, la décide et la diffuse à tous.
- Les “non” leader ($\text{id node} \neq \text{id ronde} \% N$) 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

$$f = 1$$


Consensus avec $\diamond S$

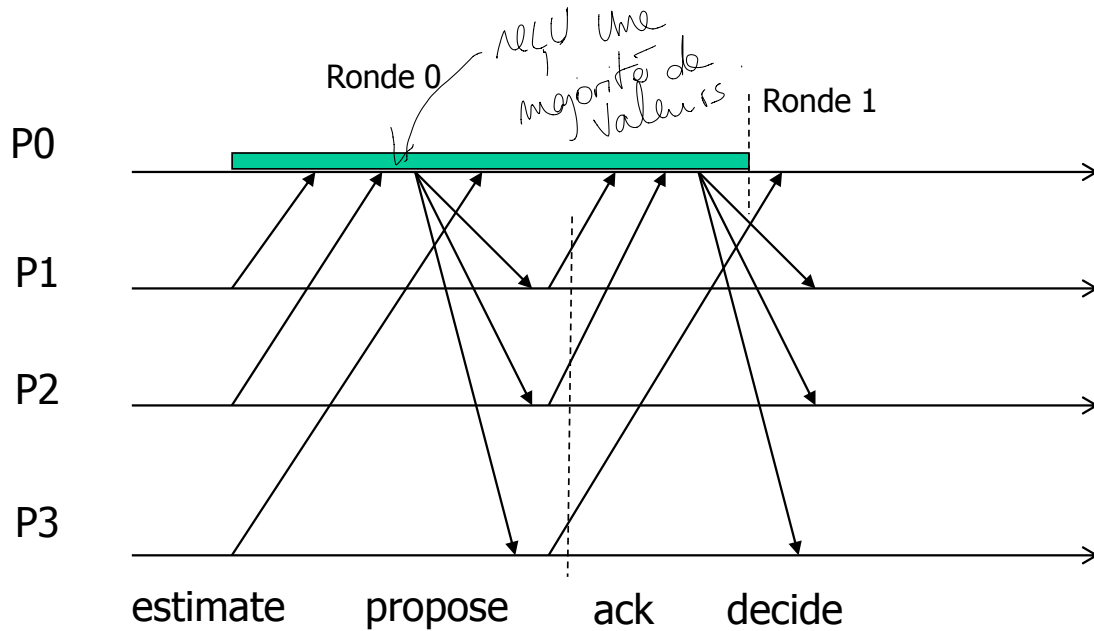
à terme on détecte bien les crash
on détecte des nœuds qui ne sont pas forcément
Crashés mais à terme
on en détecte au moins
Ajuste si il y a.

- Algorithme du coordinateur tournant [CHT 96]
- $f < n/2$ crashes
- Processus numérotés $1, 2, \dots, n$
- 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

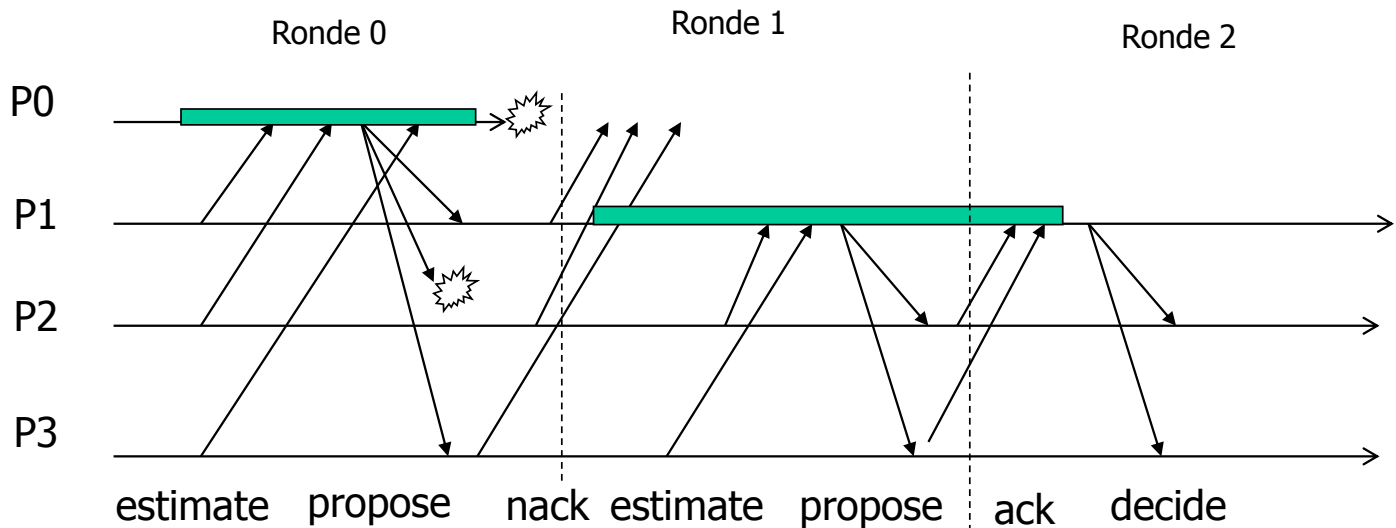
Consensus avec $\langle \rangle S$ (2)

- 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 les réponses : 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

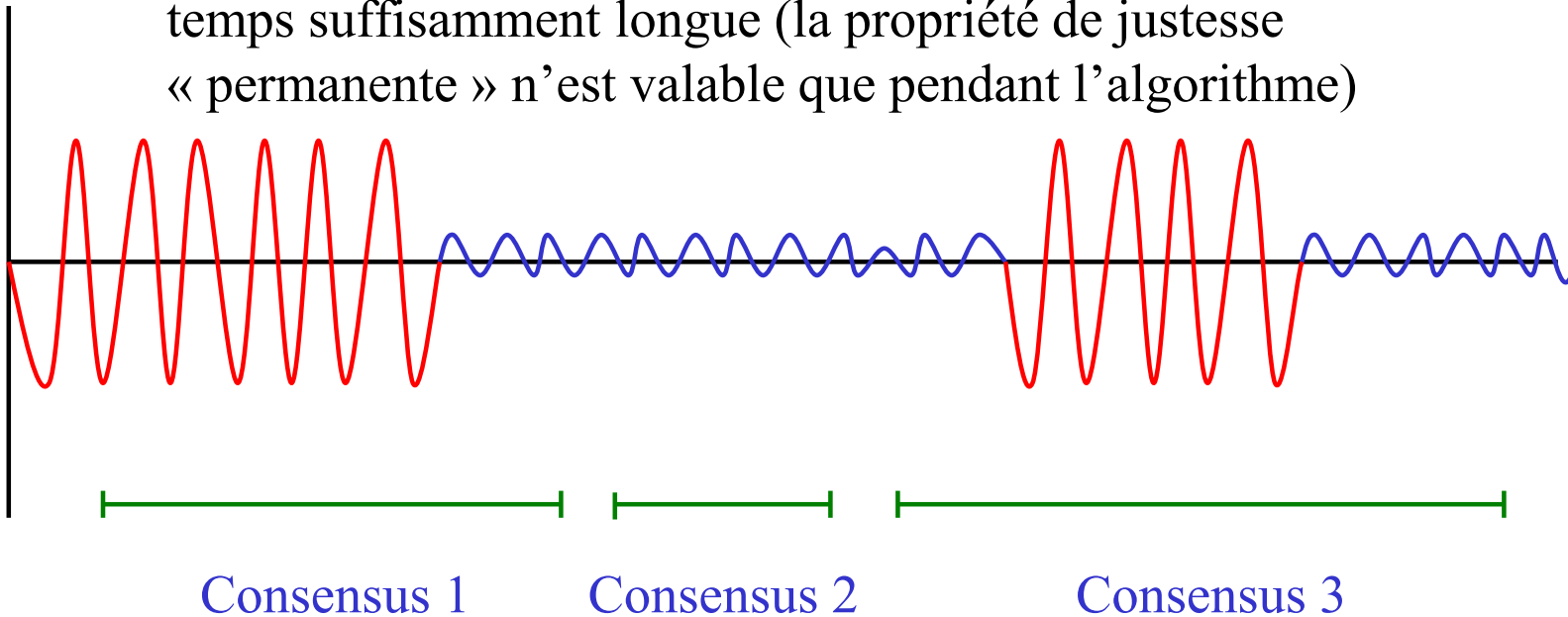


Coordinateur tournant : Exemple (2)



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

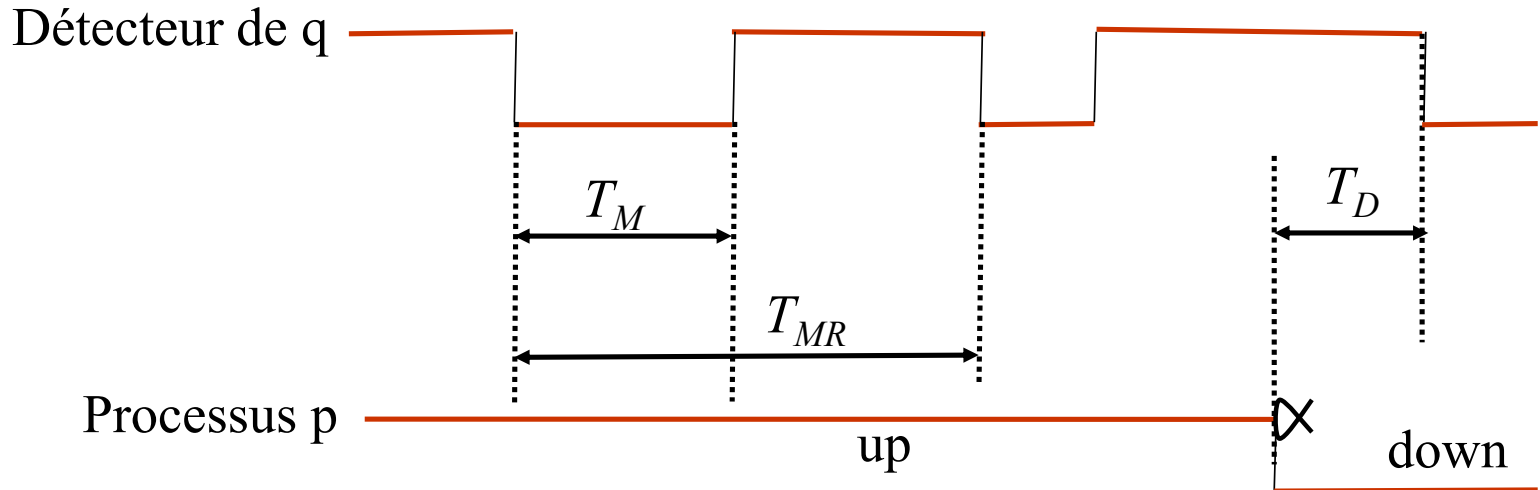
Métriques : Qualité de détection (QoD)

- Complétude

- Temps de détection (T_D)

- Justesse

- Temps entre deux erreurs (T_{MR})
- Durée des erreurs (T_M)



Hypothèses temporelles (1)

- Implémentations reposant sur des temporisateurs :
 - Systèmes partiellement synchrones

Pour $\diamond P$ (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 $\diamond P$

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 \diamondsuit S et Ω (à terme plus d'erreur sur 1 processus) :

hypothèse réduite à un ensemble de canaux ultimement synchrones
(lien ultimement ponctuel \diamondsuit -timely)

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

Attention: la borne n'est pas connue

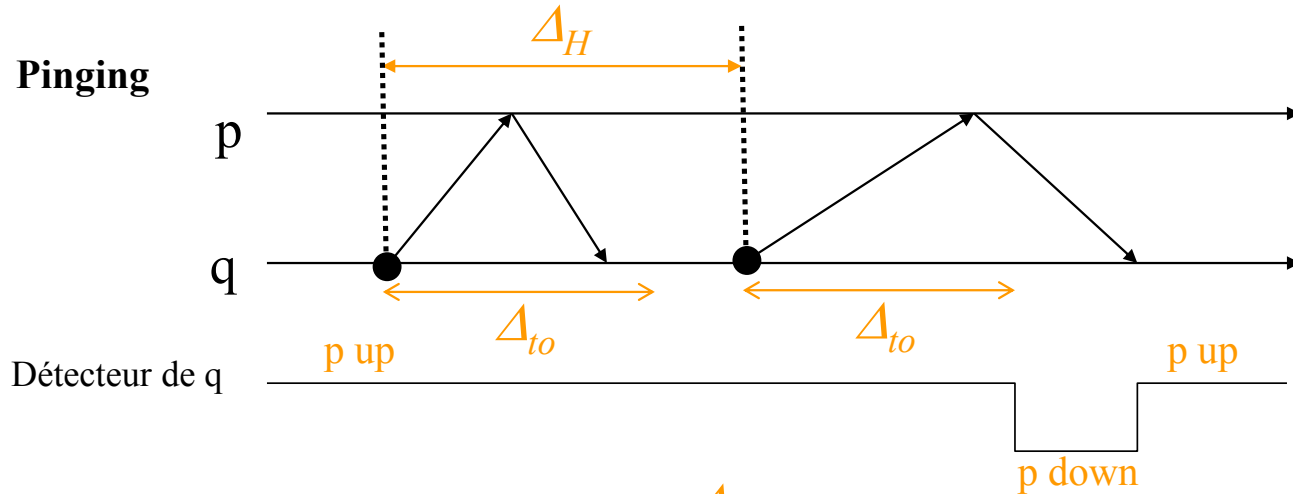
Ω peut être implémenté si au il y a au moins une \diamondsuit_f -source correcte
(f = nombre de défaillants)

Hypothèses temporelles (3)

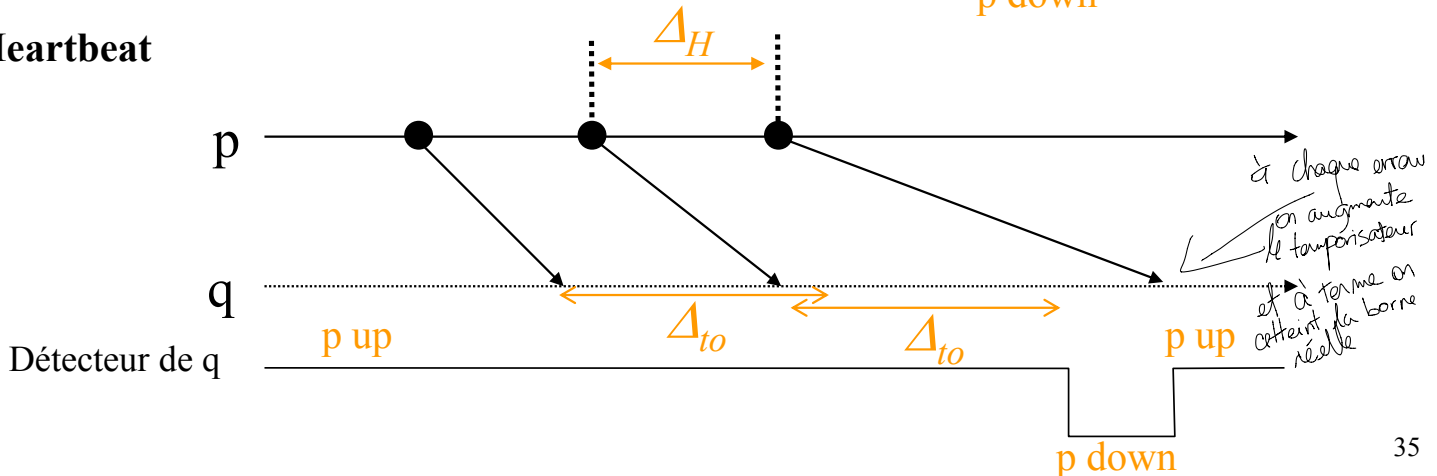
- Implémentations asynchrones (sans temporisateur):
 - Basées sur query-response (attendre un nombre fini de réponses – $n - f$)
 - Connaissance a priori du nombre de processus défaillants (f)
 - Hypothèse relative sur des canaux de communication (canaux plus rapides que d'autres)

Techniques d'implémentation

- **Pinging**

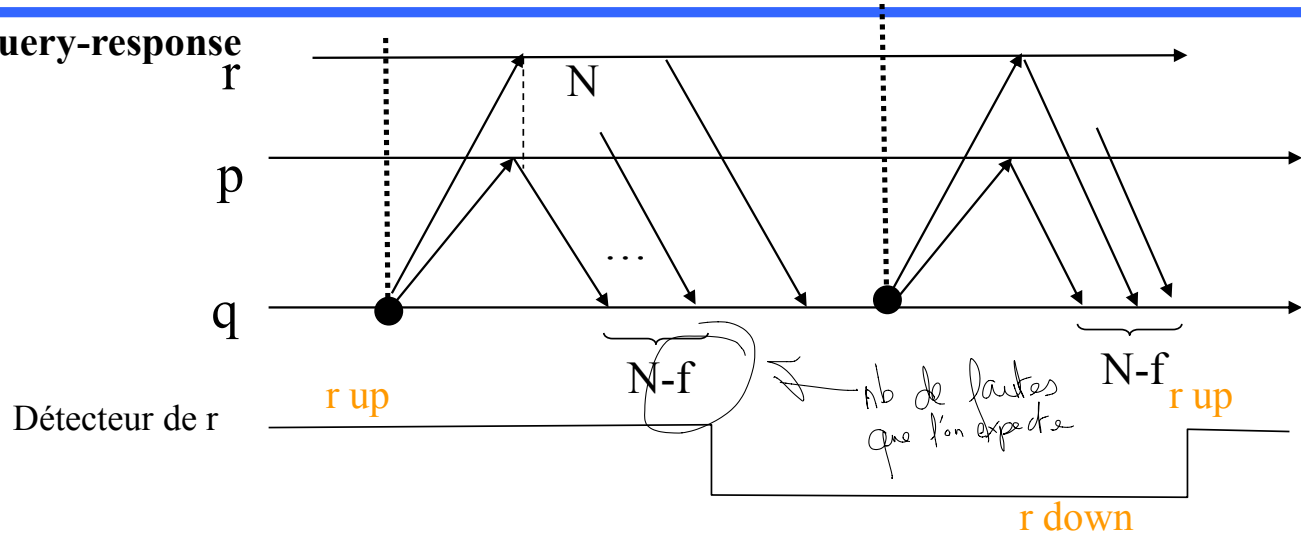


- **Heartbeat**

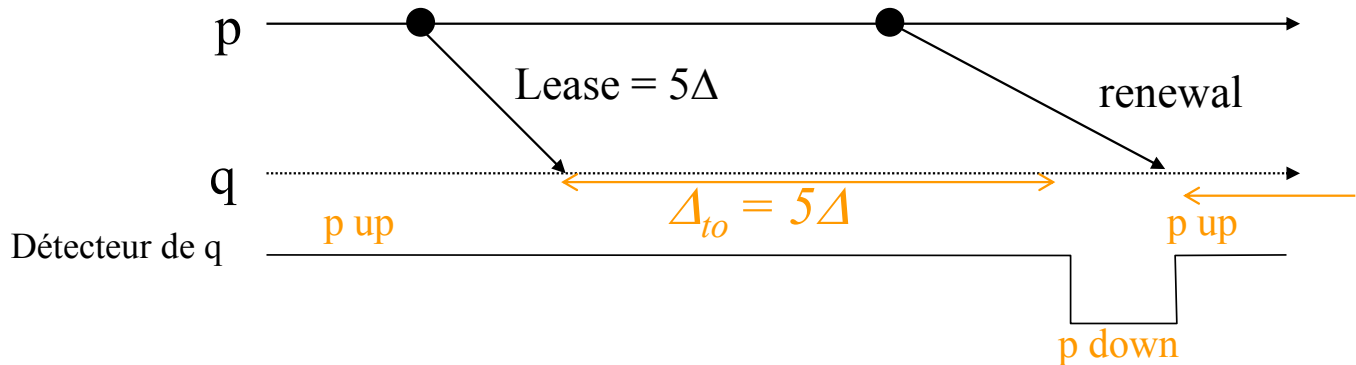


Implementations (2)

- Query-response

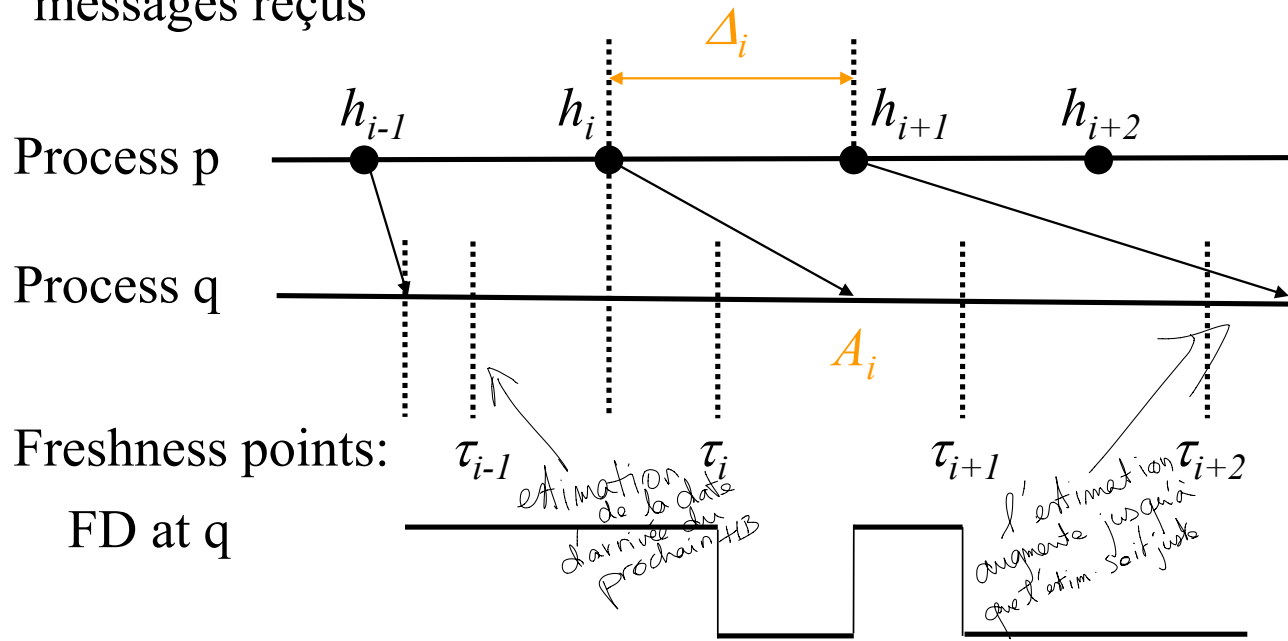


- Lease



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)

$$error_{(k)} = A_k - EA_{(k)} - delay_{(k)}$$

$$delay_{(k+1)} = delay_{(k)} + \gamma.error_{(k)}$$

$$var_{(k+1)} = var_{(k)} + \gamma.(|error_{(k)}| - var_{(k)})$$

$$\alpha_{(k+1)} = \beta.delay_{(k+1)} + \phi.var_{(k+1)}$$

Algorithme $\triangleleft \triangleright$ P (Bertier et al.)

Every process $p \in \Pi$ executes :

Initialization :

$suspect_p \leftarrow \emptyset$

for all $q \in \Pi - \{p\}$

$\Delta_{m_p}(q) = 0$

← Marge d'erreur

Task 1 :

upon receive message m_k at time t from q :

if $q \in suspect_p$ **then**

$suspect_p \leftarrow suspect_p - \{q\}$

{ trust q since $m_k(q)$ is fresh anymore at time t }

$\Delta_{m_p}(q) \leftarrow \Delta_{m_p}(q) + 1$

{ increase the timeout period }

endif

Task 2 :

upon $\tau_{k+1}(q) =$ the current time :

{ if the current time reaches τ_{k+1} ,

then none of the messages received is fresh anymore }

wait during $\Delta_{m_p}(q)$ and if no message received from q

{ detection moderation }

$suspect_p \leftarrow suspect_p \cup \{q\}$

{ suspect q since no received message is fresh anymore at this time }

Algorithme Ω

Hypothèse une $\diamond f$ -
source correct

on initialization :

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

Idee : un vecteur
(counter) qui compte
le nombre de fausses
suspensions incrémenté
si un nœud est
suspecté par $n-f$
processus

maintenir un compteur des fausses suspicions faites pour chaque processus, afin de ne pas choisir en leader un proc^{us} potentielle-
ment dangereux

task 0 :

repeat forever

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

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** $\text{counter}[r] \leftarrow \max \{ \text{counter}[r], c[r] \}$
reset timer(q) to Timeout[q]

upon expiration of timer(q) do

$\text{Timeout}[q] \leftarrow \text{Timeout}[q] + 1$

send (SUSPECT, q) to all

reset timer(q) to Timeout[q]

upon receive (SUSPECT, q) from r do

$\text{suspect}[q] \leftarrow \text{suspect}[q] \cup \{r\}$

if $|\text{suspect}[q]| \geq n - f$ **then**

$\text{suspect}[q] \leftarrow \emptyset$

$\text{counter}[q] \leftarrow \text{counter}[q] + 1$

↑
l'union des
cpt de tout le
monde.

↑
convergence

↑ suspecté par une majorité de proc^{us}
→ peut plus être leader.

Bonus : Attendre qu'un certain nb de proc^{us} ait suspecté un certain proc^{us}

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

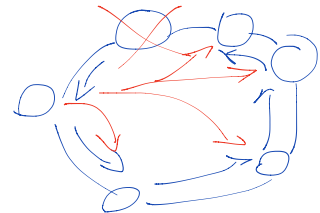
Utilisation normale du laboratoire

	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

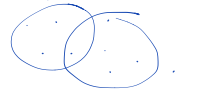
Passage à l'échelle

- Détecteur de défaillances $\diamond P$
 - n^2 messages / intervalle de temps
 - Non utilisable à large échelle
- Solutions existantes :
 - Organisation en anneau [LFA00, LAF99]
 - Peu coûteux vs temps de propagation long
 - Approche probabiliste [GCG01]
 - Difficulté pour le dimensionnement des temporisateurs
- Solution proposée :
 - Organisation hiérarchique [BMS 03]

Quadratique



fonctions de groupes

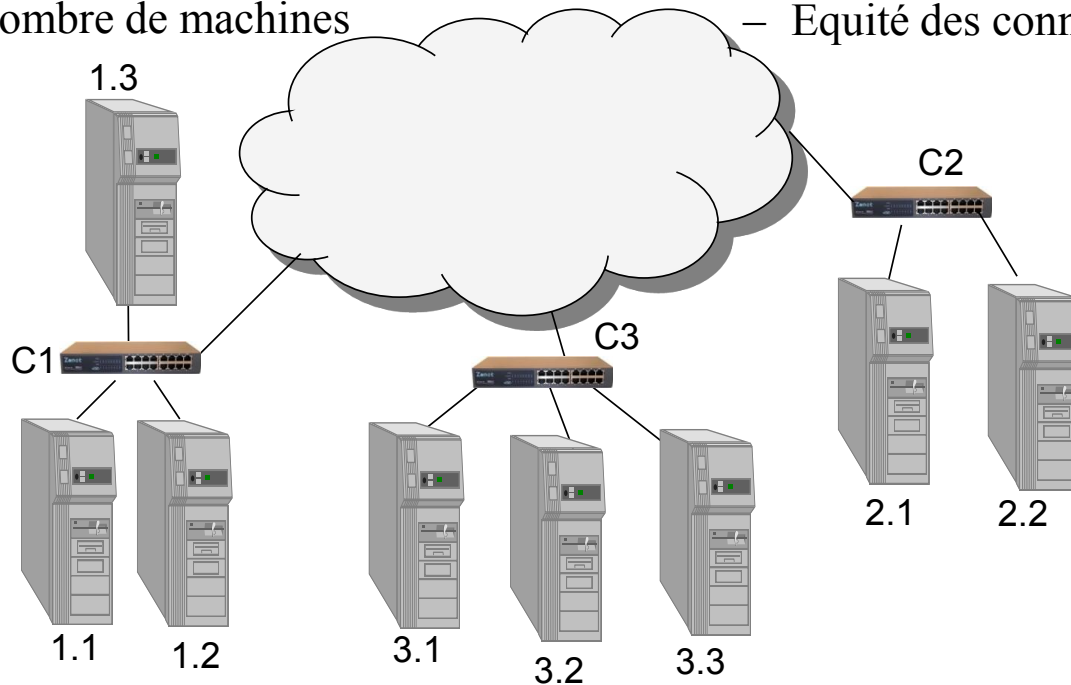


Grille de Calcul

- Interconnexion de clusters
- Grand nombre de machines

•Hypothèses

- Égalité des noeuds (rôles)
- Équité des connexions



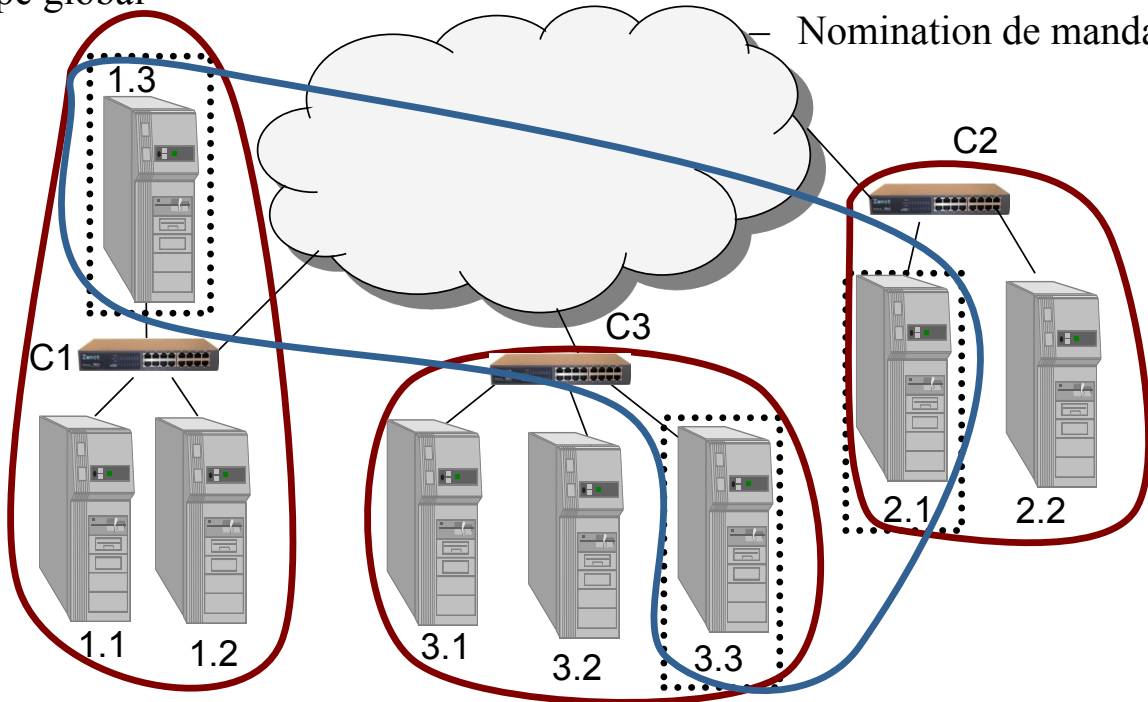
Organisation hiérarchique

- En accord avec la topologie du réseau

- 1 groupe local / cluster
- 1 groupe global

- Composition du groupe global

- Représentant dans chaque groupe local = mandataire
- Nomination de mandataire



Rôle du mandataire

- Permet la connexion du groupe local avec le reste du système
- Détecteur local : surveillance de nœud
- Détecteur global : surveillance de groupe

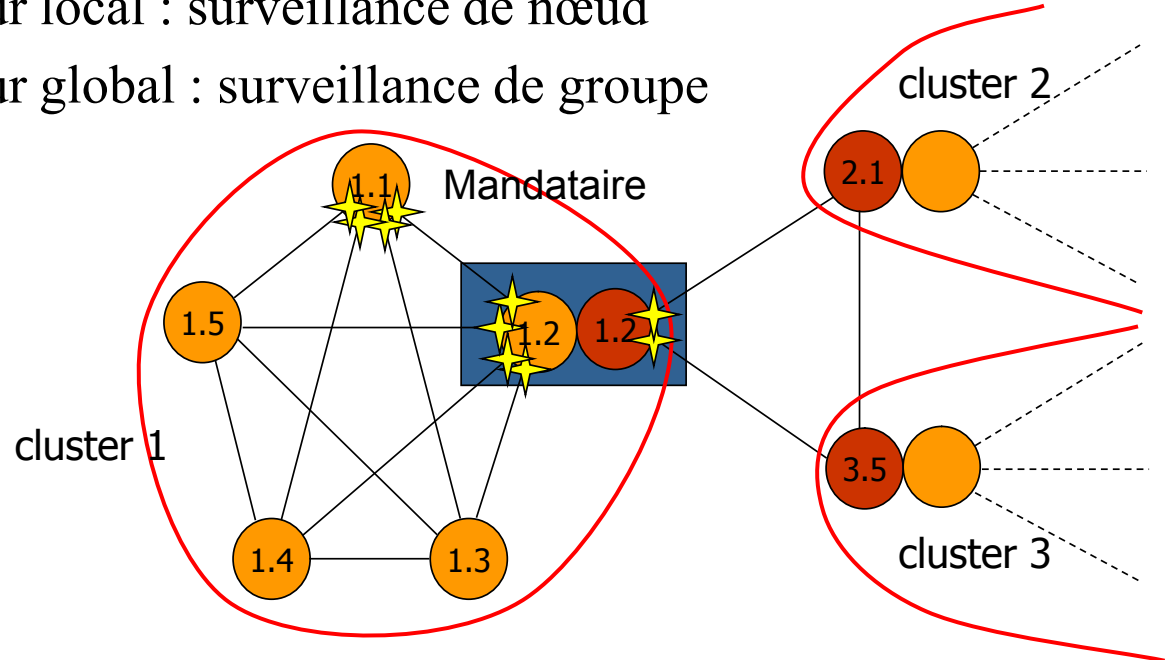
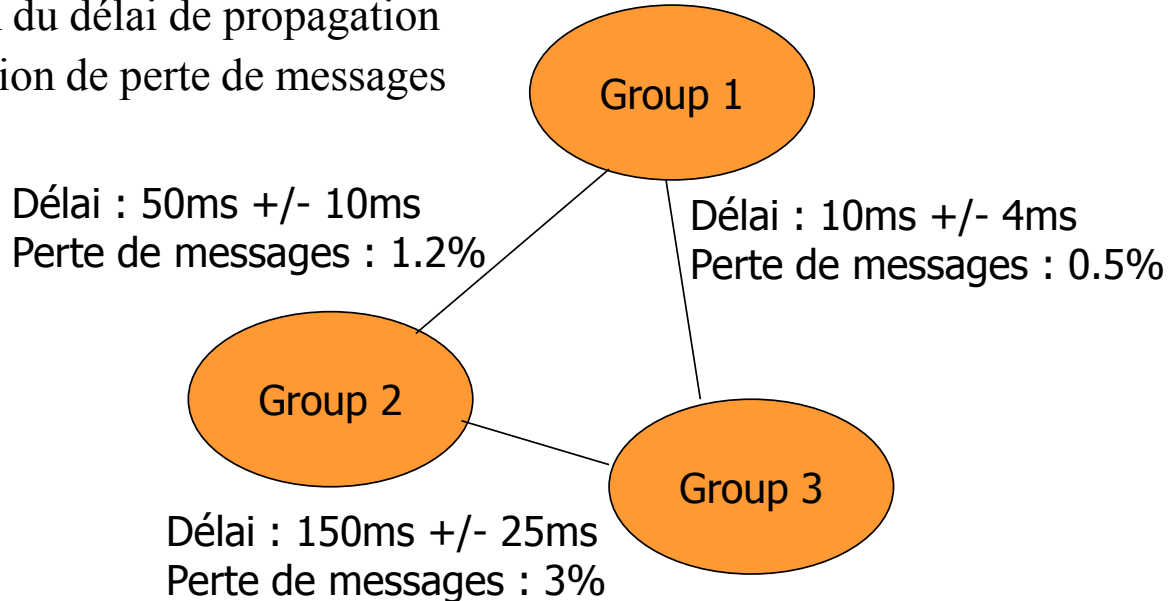


Plate-forme de test

- Utilisation de routeurs logiciels (dummynet)

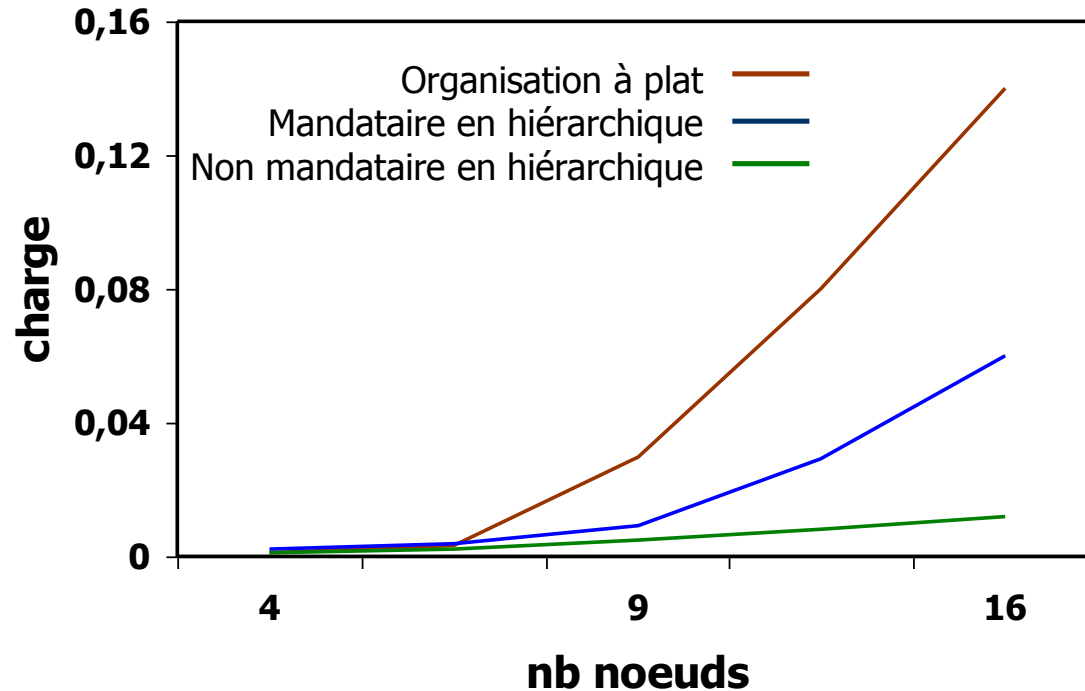
- Introduction de délai de propagation
- Variation du délai de propagation
- Introduction de perte de messages



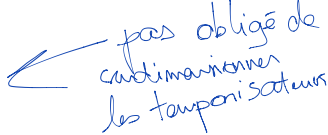
Performances

– $\Delta_H = 700\text{ms}$

Nb de nœuds	4	6	9	12	16
Nb de nœuds par groupe local	2	3	3	4	4
Nb de groupes locaux	2	2	3	3	4



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 hierarchical failure detector. In Proceedings of the International Conference on Dependable Systems and Networks, june 2003.
- [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 failure detectors. In Proc. of First Conference on Dependable Systems and Networks, june 2000.
- [DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. Journal of the ACM, 34(1) :7797, 1987.
- [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.
- [GCG01] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient 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.



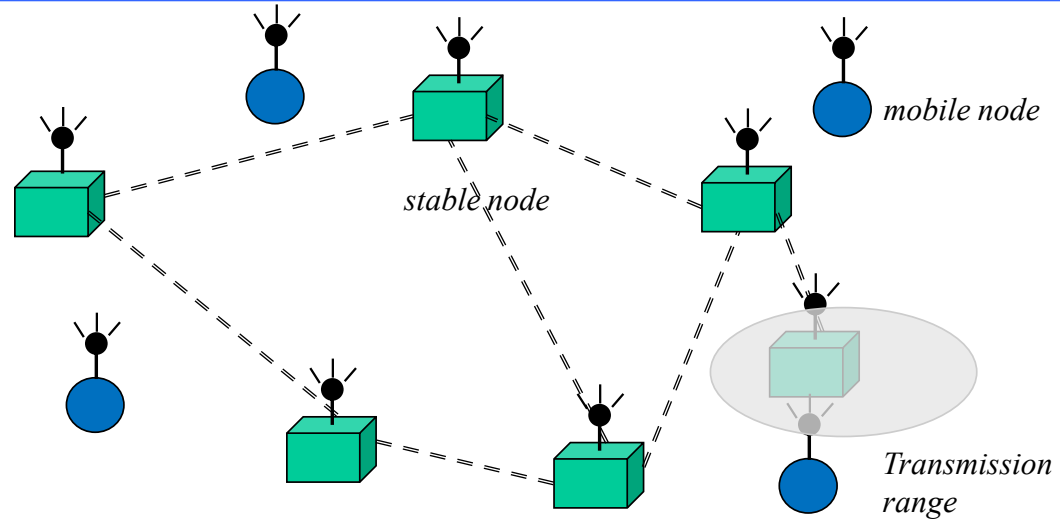
A Failure detector for Wireless Networks with Unknown Membership

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

- $\Pi = \{p_1, p_2, \dots, p_n\}$
- 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 Π : known_i
(initially, $\text{known}_i = \{i\}$)

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

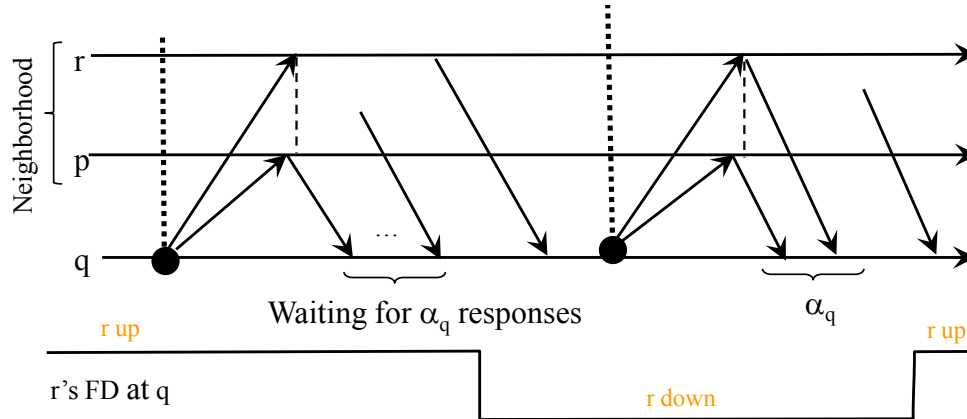
$\Diamond S^M$: Eventually strong FD with unknown membership

- 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 $\geq \alpha_i$ processes

$rec_from_i \leftarrow$ all p_j , a RESPONSE is received

 For **all** $p_j \in known_i \setminus rec_from_i \mid \langle p_j, - \rangle \notin susp_i$ do

 If $\langle p_j, ct \rangle \in mist_i$

 Add($susp_i, \langle p_j, ct + 1 \rangle$)

$mist_i = mist_i \setminus \{ \langle p_j, - \rangle \}$

 Else

 Add($susp_i, \langle p_j, 0 \rangle$)

End repeat

Algorithm (2) : Reception of responses

Task T2:

Upon reception of QUERY ($susp_j, mist_j$) from p_j do

$known_i \leftarrow known_i \cup \{p_j\}$

For all $\langle p_x, ct_x \rangle \in susp_j$ do

If $\langle p_x, - \rangle \notin susp_i \cup mist_i$ or $(\langle p_x, ct \rangle \in susp_i \cup mist_i$ and $ct < ct_x)$

If $p_x = p_i$

 Add($mist_i, \langle p_i, ct_x + 1 \rangle$)

Else

 Add($susp_i, \langle p_x, ct_x \rangle$)

$mist_i = mist_i \setminus \{\langle p_x, - \rangle\}$

The receiver was suspected :
generation of a mistake

Update susp. set with more
recent information

For all $\langle p_x, ct_x \rangle \in mist_j$ do

If $\langle p_x, - \rangle \notin susp_i \cup mist_i$ or $(\langle p_x, ct \rangle \in susp_i \cup mist_i$ and $ct < ct_x)$

 Add($mist_i, \langle p_x, ct_x \rangle$)

$susp_i = susp_i \setminus \{\langle p_x, - \rangle\}$

If $(p_x \neq p_j)$

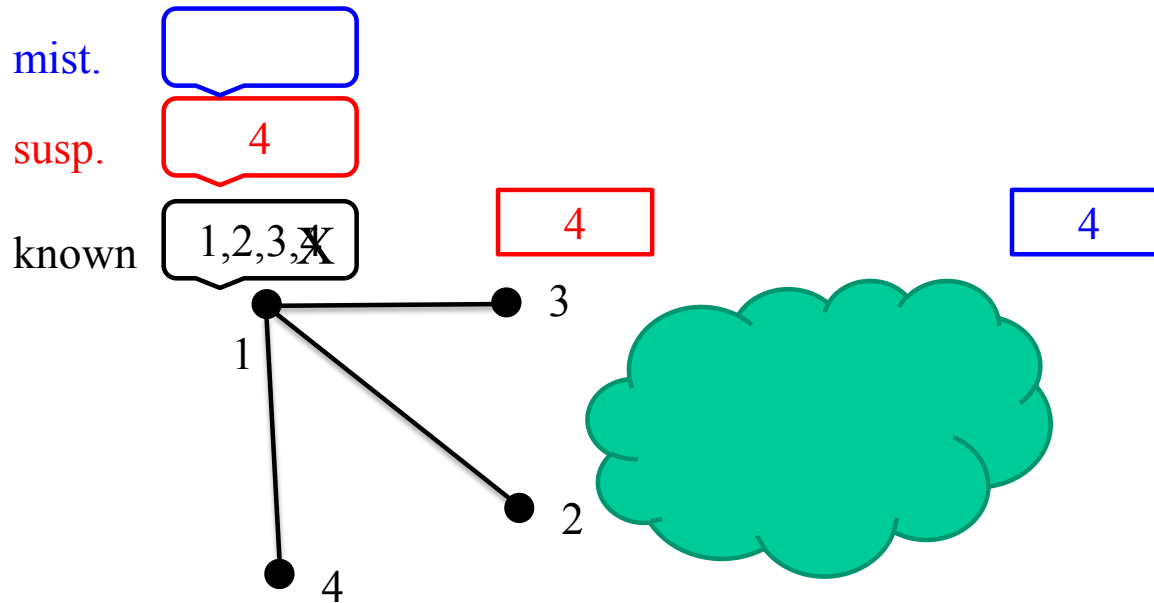
$known_i = known_i \setminus \{p_x\}$

The sender is not the mistake node :
suspicion of move

Update mist. set with more
recent information

Send RESPONSE to p_j

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

 \Rightarrow *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.

 \Rightarrow *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

 \Rightarrow *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 $\Diamond S$
 - Membership
 - Minimum stability of moving nodes
 - Responsiveness
- Perspectives:
 - Model: A relaxed model with weaker connectivity assumptions