École Doctorale de Grenoble
Master 2 Recherche “Systèmes et Logiciel”

Consensus - 1

Systèmes synchrones et asynchrones avec pannes franches

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/people/krakowia

Pourquoi s’intéresser au consensus ?

- **Un problème fondamental de l’algorithmique répartie**
  - Un modèle pour la prise de décision concertée
  - Des solutions simples en l’absence de défaillances
  - Des solutions très complexes en présence de défaillances
  - De nombreux cas d’impossibilité
  - Des problèmes encore ouverts
- **Une base pour la construction d’outils importants**
  - Diffusion atomique
  - Protocoles de groupe
  - Validation non bloquante

Plan

Spécifications du consensus

Soit un ensemble de processus p_1, …, p_n reliés par des canaux de communication

Initialement : chaque processus p_i propose une valeur v_i
À la fin (si l’algorithme se termine) : chaque processus p_i décide une valeur d_i

Conditions :

- **Accord** : la valeur décidée est la même pour tous les processus corrects
- **Intégrité** : tout processus décide au plus une fois (une décision prise est définitive)
- **Validité** : toute valeur décidée est l’une des valeurs proposées
- **Terminaison** : si au moins un processus correct lance le consensus, tout processus correct décide au bout d’un temps fini

Un processus est dit correct s’il n’est pas en panne (ou n’est jamais tombé en panne, si on admet la réparation et la réinsertion de processus défaillants)

Il existe diverses variantes des spécifications, cf plus loin

On peut faire diverses hypothèses de défaillances sur les processus et sur le système de communication

Remarques sur les spécifications

Démarrage
Le consensus doit être initialisé (tous les processus doivent “savoir” quand démarrer l’algorithme). Cela peut se faire, par exemple :

- à l’initiative d’un processus, qui diffuse aux autres un message *lancer_consensus*. Attention aux propriétés de cette diffusion
- à une heure fixée à l’avance, si les processus ont des horloges en temps réel. Attention à la dérive des horloges

Consensus simple ou consensus uniforme
Dans le consensus simple, on ne s’intéresse qu’aux décisions des processus corrects. Dans le consensus uniforme, on demande l’accord de tous les processus (un processus fautif a pu décider, et exécuter des actions irréversibles fondées sur cette décision, avant de tomber en panne)

Rôles des processus
Dans la version initiale, tous les processus ont le même rôle. On peut aussi séparer ceux qui proposent et ceux qui décident. Si un seul propose, on retrouve une forme de diffusion

Hypothèses de défaillances

Communication
On considère le plus souvent (cf paradoxe des généraux !) un système de communication fiable (tout message finit par arriver), mais il peut être

- synchrone (borne connue sur le temps de transmission)
- asynchrone (pas de borne connue sur le temps de transmission)

Sites (processus)
Nous considèrons les hypothèses suivantes

- Sites fiables (pas de défaillances)
- Pannes franches (*fail stop*) avec ou sans reprise (un processus défaillant peut ou non se réinsérer après réparation)
- Pannes byzantines (comportement arbitraire)

avec communication synchrone ou asynchrone

D’autres hypothèses (défaillances par omission) ont été examinées dans la littérature

Préambule : diffusion fiable

La diffusion est examinée en détail plus loin. Ici nous présentons un protocole particulier (le plus simple) dont nous aurons besoin dans la suite

Hypothèses : communication asynchrone fiable (tout message parvient intact à son destinataire) ; pannes franches des processus ; tout processus correct peut toujours communiquer avec tout autre processus correct

Spécifications de la diffusion fiable (*reliable broadcast*) :

Un processus diffuse un message à un ensemble spécifié de processus (y compris lui-même). Le protocole doit vérifier les trois propriétés suivantes.

- Accord : si un processus correct délivre un message m, tous les processus corrects délivrent m (au bout d’un temps fini)
- Validité : si un processus correct diffuse un message m, tous les processus corrects délivrent le message m (au bout d’un temps fini)
- Intégrité : Quel que soit le message m, il est délivré au plus une fois à tout processus correct, et seulement s'il a été diffusé par un processus

Rappel : distinguer “délivrer” (protocole de diffusion) de “recevoir” (système de communication sous-jacent)

Diffusion fiable : réalisation

Un protocole de diffusion fiable :

Tout processus p exécute le programme suivant :

pour exécuter **broadcast**(m) :

estampiller m avec
sender(m) (processus émetteur) et seq(m)
// seq est un numéro de séquence local à l’émetteur
// on a ainsi une identification unique du message
send(m) à tous les voisins de p, et à p

deliver(m) se produit comme suit :

sur exécution de **receive**(m) par processus p
if p n’a pas précédemment exécuté **deliver**(m) **then**
// vérifié grâce à l’identification unique de m
if sender(m) ≠ p **then send**(m) à tous les voisins de p
deliver(m)

send(m) et receive(m) sont les primitives du système de communication (fiable) disponible

voisin de p : processus q tel qu'il y ait un lien de communication direct de p vers q

Diffusion fiable : remarques

Sur les spécifications :

- Rien n'est dit sur l'ordre dans lequel les messages sont reçus
- Si un processus a une défaillance pendant l'exécution de **broadcast**(m), alors m sera délivré à tous les processus, ou à aucun (propriété "tout ou rien"). C'est la propriété minimale (connaissance partagée) qui permet les raisonnements sur la diffusion

Sur la réalisation :

- Ce protocole n'est pas optimisé ; l'objectif est de montrer que la diffusion fiable est possible sous les hypothèses indiquées

V. Hadzilacos , S. Toueg, Fault-Tolerant Broadcast and Related Problems, in S. Mullender (ed.), *Distributed Systems* (2nd edition), Addison-Wesley, 1993

Diffusion fiable : preuve de l'algorithme

Accord :

Si un processus correct a exécuté **deliver**(m1), il a au préalable envoyé m1 à tous ses voisins. Tous les processus corrects étant connectés, tout processus correct finit par exécuter **receive**(m1). Mais un processus correct ne peut pas exécuter **deliver**(m2) sans avoir auparavant exécuté **receive**(m2). Comme le système de communication ne modifie pas les messages, m1 = m2

Validité :

Si un processus correct a exécuté **broadcast**(m), il a envoyé m à tous ses voisins. Donc tout processus correct finit par exécuter **receive**(m), donc nécessairement **deliver**(m)

Intégrité :

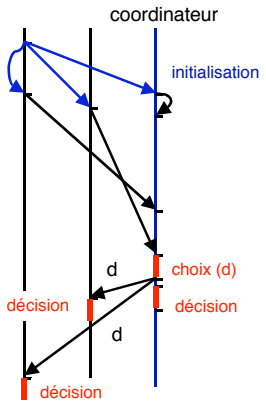
Résulte de la structure de l'algorithme (m n'est délivré que s'il ne l'a a pas déjà été, verifié par identification unique) et du fait que le système de communication ne crée pas de messages ex nihilo

Consensus en l'absence de pannes (1)

Hypothèses : communication fiable asynchrone ; sites fiables (pas de défaillances)

Une solution avec coordinateur

- Chaque processus p_i envoie sa valeur v_i à un coordinateur spécifié à l'avance
- Quand le coordinateur a reçu toutes les valeurs v_i, il choisit une de ces valeurs, soit d (sur un critère quelconque)
- Le coordinateur envoie d à tous les p_i
- Après réception, chaque p_i décide d



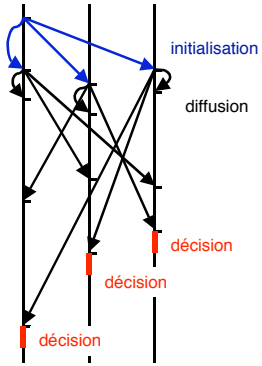
En communication asynchrone, tous auront décidé en temps fini mais non borné

Consensus en l'absence de pannes (2)

Une autre méthode (solution symétrique)

Méthode très redondante dans ce cas simple, mais utile pour préparer la suite

- Chaque processus p_i diffuse sa valeur v_i à tous
- Quand un processus p_i a reçu toutes les valeurs v_i, il applique un algorithme de choix d'une de ces valeurs (par exemple définir un ordre sur un critère quelconque et prendre la première)
- L'algorithme de choix est le même pour tous les p_i



Si le protocole de diffusion garantit que tous les destinataires reçoivent le message (sans difficulté en l'absence de pannes), alors tous les p_i décident la même valeur
En communication asynchrone, tous auront décidé en temps fini mais non borné

Consensus synchrone avec pannes franches : hypothèses

La communication est fiable (tout message parvient à destination) et synchrone (borne supérieure connue pour la durée de transmission d'un message)
Le maillage du réseau n'est pas nécessairement complet (la communication entre deux sites peut nécessiter le passage par un ou plusieurs autres). Toutefois on suppose que deux sites corrects peuvent toujours communiquer entre eux (les pannes de sites ne partitionnent pas le réseau)

Les processus peuvent avoir des pannes franches ; on suppose qu'il subsiste au moins un processus correct (n–1 défaillances au plus pour n processus)

On suppose disponible un mécanisme de diffusion fiable tel que spécifié plus haut :

- tous les processus corrects délivrent le même ensemble de messages
- tous les messages diffusés par les processus corrects sont délivrés
- tout message délivré a été diffusé par un processus

Consensus synchrone avec pannes franches : algorithme

Le principe de l'algorithme est le même que pour le système sans pannes (avec diffusion)

Après initialisation, chaque processus p_i diffuse (diffusion fiable) sa valeur v_i
Grâce à l'hypothèse de synchronisme et à la connaissance de la topologie du réseau, on peut déterminer un temps T au bout duquel tous les messages qui doivent être délivrés ont effectivement été délivrés (cf calcul de T plus tard)

D'après les propriétés de la diffusion fiable, tous les processus corrects ont reçu le même ensemble de messages. Si on applique alors le même algorithme de choix, alors chaque processus correct décidera (en temps borné) la même valeur.

Remarques

Si un processus tombe en panne pendant la diffusion d'un message, celui-ci sera délivré à tous les processus corrects ou à aucun.

La valeur décidée peut être celle diffusée par un processus fautif (panne pendant ou après la diffusion). Mais cela ne contredit pas la spécification du consensus

Consensus asynchrone avec pannes franches

- **Résultat fondamental (Fischer, Lynch, Paterson - 1983)**
  - Dans un système asynchrone avec pannes franches, il n'existe pas de protocole déterministe réalisant le consensus si on admet la défaillance d'un seul processus
- **Idée très sommaire de la preuve**
  - Supposons que 2 valeurs soient proposées (0, 1). Le système est dans une configuration bivalente si les deux valeurs sont des choix possibles pour la décision
  - La configuration initiale est bivalente
  - Il est toujours possible de construire une séquence d'événements qui maintienne indéfiniment le système dans une configuration bivalente

  La source du problème est que, dans un système asynchrone, il est impossible de distinguer un processus lent (à s'exécuter ou à répondre) d'un processus défaillant

M. J. Fischer, N. Lynch, M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process, *Journal of the Association for Computing Machinery*, 32(2), pp. 374-382, April 1985 (publication initiale : *Proc. 2nd ACM Principles of Database Systems Symposium*, March 1983)

Comment vivre avec FLP ?

- **Lever une des restrictions**
  - Système "partiellement synchrone" (divers modèles)
    - Dwork, Lynch, Stockmeyer (1988)
  - Algorithmes non déterministe
    - Rabin (1983)
- **Accepter une solution imparfaite (et faire "le mieux possible" (*best effort*) en un certain sens)**
  - Algorithme de Paxos
    - Lamport (1989)

    intérêt : hypothèses très faibles ; transposable au cas de pannes byzantines
  - Détecteurs de pannes imparfaits
    - Chandra, Toueg (1991) ; améliorations ultérieures (1998)

    intérêt : identification précise de l'origine de l'impossibilité et des limites des algorithmes réalisables

Algorithme de Paxos

Algorithme pour le consensus asynchrone
Une première idée très informelle de son principe :

L'algorithme utilise un processus coordinateur (ou *primaire*) qui gère un ensemble de processus (dits *agents*) et tente d'obtenir une valeur de décision par vote majoritaire.

L'algorithme doit résister à la défaillance du primaire. Plusieurs processus peuvent jouer le rôle du primaire, successivement ou même simultanément

L'algorithme se termine s'il existe un primaire unique pendant une période suffisamment longue pour que le primaire ait deux tours d'échanges avec une majorité d'agents.

On essaie d'assurer cette condition à l'aide de délais de garde. Cette méthode peut échouer (la communication étant asynchrone), et dans ce cas l'algorithme ne se termine pas (FLP est toujours valable !)

Article original : L. Lamport, The part-time parliament, Tech. Report 49, Systems Research Center, Digital Equipment Corp., Palo Alto, Sep. 1989

Présentation pédagogique dans B. Lampson, The ABCDs of Paxos, in *ACM Conference on Principles of Distributed Computing* (PODC), 2001, en ligne

Algorithme de Paxos : hypothèses

Communication
Asynchrone
Pas d'altération des messages
Possibilité de perte de messages

Processus
Nombre fixe de processus.

Pannes franches avec possibilité de reprise. Chaque processus a un état volatile (perdu en cas de panne) et un état persistant (préservé en cas de panne)

Une version récente (Castro, Liskov, 2000) s'applique aux pannes byzantines

Algorithme de Paxos

Point de départ : recherche d'une **majorité** par un ensemble de processus

- Si une majorité choisit une valeur v, alors v est la valeur décidée
- S'il n'y a pas de majorité, pas de décision (par convention valeur nil)

Si des membres de la majorité tombent en panne avant la décision, pas de décision si les processus survivants ne sont pas majoritaires

Pour remédier à cela, on définit une séquence de "vues" (1, 2, … i, …). Une valeur v_i est choisie dans chaque vue i (elle peut être nil en cas d'échec). Si une vue n choisit une valeur $v_n \neq$ nil, alors v_n est la valeur décidée

Atteindre une majorité dans une vue ne suffit pas pour garantir une décision dans cette vue (à cause des défaillances). Néanmoins, si une valeur a été majoritaire une fois, c'est cette valeur qui sera toujours proposée ensuite (donc si deux vues atteignent une majorité, c'est pour la même valeur). Pour arriver à une décision, chaque vue est organisée par un primaire. Celui-ci interroge les processus (ou agents) sur leur état (i.e. leur "vote") dans les vues précédentes et propose une valeur "acceptable". S'il recueille une majorité, il diffuse la valeur, qui devient la valeur décidée

Reste à traiter le cas de la panne du primaire…

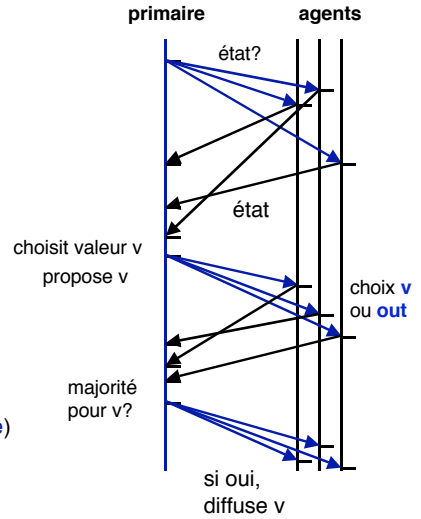
Paxos : schéma d'une vue

Lors de chaque vue, le primaire
- interroge les agents sur leur état passé
- choisit et propose une valeur considérée comme acceptable (cf plus loin)
- si cette valeur recueille la majorité, il la distribue (c'est la valeur décidée)

Un agent a 3 états possibles dans une vue :
- init (état initial)
- out (refus)
- v (une valeur)

Un agent ne change plus d'état (dans la même vue) une fois qu'il a choisi (**out** ou **v**), même s'il tombe en panne et est réparé

Le résultat d'une vue est "échec" (la vue est nulle) s'il y a une majorité de non, "succès" (la vue est réussie) s'il y a une majorité pour une valeur v



Paxos : invariants (1)

Valeur d'une vue : par définition, la vue n a la valeur v si un agent a la valeur v dans cette vue; sinon la valeur de la vue est nil. Pour que cela ait un sens, il faut que tous les agents qui ont une valeur dans une vue aient la même valeur.

Invariant 1 : une vue a au plus une valeur

La décision est la valeur d'une vue réussie (majorité pour une valeur). Donc on doit avoir :

Invariant 2 : toutes les vues réussies ont la même valeur

Bien noter que le fait qu'une vue soit réussie ne veut pas dire qu'une décision sera prise à l'issue de cette vue. On peut atteindre une majorité sans qu'une décision soit prise, par exemple à cause de la panne du primaire, ou de la panne de plusieurs agents avant transmission de leur valeur au primaire, les survivants n'étant plus majoritaires

Paxos : invariants (2)

L'invariant 2 dit que toutes les vues réussies ont la même valeur. Pour aider le raisonnement, on va imposer un ordre sur les vues. Entre deux vues réussies successives, on ne peut avoir que des vues nulles. Donc Invariant 2 est assuré par :

Invariant 3 : pour tout n, **et pour tout** m ≤ n **:**
si m **est réussie alors** v_n = nil **ou** $v_n = v_m$

ce qui peut aussi s'écrire (puisqu'une vue est nulle ou réussie) :

Invariant 3' : pour tout n, **et pour tout** m ≤ n **:**
si vue m **n'est pas nulle alors** v_n = nil **ou** $v_n = v_m$

Ceci peut se transformer ainsi :
pour tout n et pour tout m ≤ n, vue m est nulle **ou** v_n = nil **ou** $v_n = v_m$
pour tout n, v_n = nil ou (pour tout m ≤ n, vue m est nulle **ou** $v_n = v_m$)

Définition : une vue n est fixée si pour tout m ≤ n, vue m est nulle ou $v_n = v_m$

Invariant 3'' : pour tout n, v_n = nil **ou** vue n est fixée

C'est une propriété stable

Paxos : maintien des invariants

L'algorithme vise à maintenir les invariants (sûreté) en assurant la progression (vivacité).

Invariant 1 : une vue a au plus une valeur ; assuré s'il y a un primaire unique par vue

Invariant 2 : toutes les vues réussies ont la même valeur ; assuré si on garantit l'invariant 3, sous la forme 3'' :

Invariant 3'' **: pour tout** n, v_n = nil **ou** vue n est fixée. Assuré par le primaire, qui ne propose une valeur que si c'est celle d'une vue réussie antérieure (si aucune vue antérieure n'a été réussie, pas de contraintes). Si une vue est fixée, elle peut décider

Il faut maintenant assurer la progression malgré les défaillances

Les défaillances peuvent entraîner deux problèmes :
une vue est "trop lente" (ne parvient pas à déterminer une valeur)
un primaire tombe en panne avant qu'une décision ne soit prise
Les symptômes sont les mêmes (la vue ne décide pas)

Paxos : résistance aux défaillances

Si une vue tarde à prendre une valeur (v ou nil) : il faut le détecter (délai de garde) et lancer une nouvelle vue n (n >m). Mais il faut aussi "fermer" la vue m (pour la neutraliser). Pour cela, on impose la valeur out à tout processus qui a encore la valeur init dans la vue m. Noter que le primaire de m peut être toujours là mais la vue m ne peut plus décider (elle est donc nulle)

Attention : il y a là une source potentielle de délai infini, si de nouvelles vues sont créées indéfiniment et neutralisent les vues précédentes avant que celles-ci aient décidé. Mais ce n'est pas surprenant : les délais de garde ne sont pas infaillibles en asynchrone, FLP est toujours valable !

Rappel : une vue peut être réussie (une valeur v a recueilli une majorité) sans que cette vue ait décidé ; ceci à cause des défaillances. Mais à partir de ce moment, aucune vue ultérieure ne peut décider autre chose que v (c'est le sens de la propriété "vue fixée")

Paxos : exemple (d'après [Lampson 2001])

	Exécution 1				Exécution 2			
	valeur vue v	a	b	c	valeur vue v	a	b	c
	c_v	r_v^a	r_v^b	r_v^c	c_v	r_v^a	r_v^b	r_v^c
Vue 1	7	7	out	out	8	8	out	out
Vue 2	8	8	out	out	9	9	out	9
Vue 3	9	out	out	9	9	out	out	9
prop. possibles Vue 4	{7, 8, 9} si on voit a, b, c {8} si on voit a, b {9} si on voit a, c ou b, c				{9} dans tous les cas			

Remarques. Dans l'exécution 2, la vue 2 est réussie, mais n'a pas décidé (le primaire est mort, ou elle a été fermée par la vue 3). Dans l'exécution 1, toutes les vues sont nulles, mais on ne peut le savoir qu'en consultant les 3 processus

Paxos : quelques détails

Comment ordonner les vues ?

Une vue est définie par un numéro composé (numéro du primaire, numéro unique). On choisit un numéro unique strictement croissant (horloge physique par exemple). On garantit ainsi l'association primaire-vue (et l'existence d'au plus un primaire par vue)

Comment changer de vue ?

Le changement de vue consiste en l'élection d'un primaire pour la nouvelle vue. Le primaire doit commencer par interroger les vues antérieures, i.e. demander à chaque processus ses états antérieurs dans les vues précédentes, fermer les vues non décidées (valeur init mises à out) et choisir une valeur à proposer parmi celles possibles (pour fixer la vue courante).

La vue décide si elle est fixée et si le primaire peut faire deux aller-retour avec une majorité de processus (pas nécessairement la même pour les deux échanges)

Paxos : algorithme du primaire (résumé)

Un primaire est élu lorsqu'une vue n'aboutit pas dans un délai fixé (estimation du temps de 2 aller-retour + temps de traitement)

Le primaire signale à tous le changement de vue, ce qui provoque la fermeture des vues antérieures (pour tout processus, mettre à out les valeurs encore à init ; il suffit de le faire entre la vue courante et la dernière vue pour laquelle le processus a une valeur)

Le primaire collecte les valeurs des processus et en choisit une (parmi celles déjà choisies ; pas de restriction si pas de choix antérieur)

Le primaire envoie la valeur choisie r_v et collecte les réponses. Si majorité pour la valeur r_v, celle-ci devient la valeur décidée d, et le primaire l'envoie à tous les processus

Noter que si les actions ci-dessus n'aboutissent pas dans le délai fixé, un nouveau primaire fermera la vue (imposera out aux processus qui n'ont pas encore une valeur dans la vue) et la vue échouera, d'où nouveau cycle…

Paxos : conclusion

Paxos est un protocole pour le consensus en présence de fautes dans un système asynchrone. Son intérêt est à la fois théorique et pratique

Paxos fonctionne avec des hypothèses très faibles sur les défaillances : arrêt avec reprise, perte de messages

Le principe de Paxos est de construire une suite de vues, dont on espère (sans certitude) que l'une parviendra à décider

- chaque vue choisit une valeur et cherche à obtenir une majorité
- le choix se fait parmi des valeurs (stables) ayant obtenu une majorité (sans décision) dans des vues antérieures

L'analyse considère séparement les propriétés de sûreté et de vivacité

La mise en œuvre utilise un processus primaire, avec délai de garde et élection

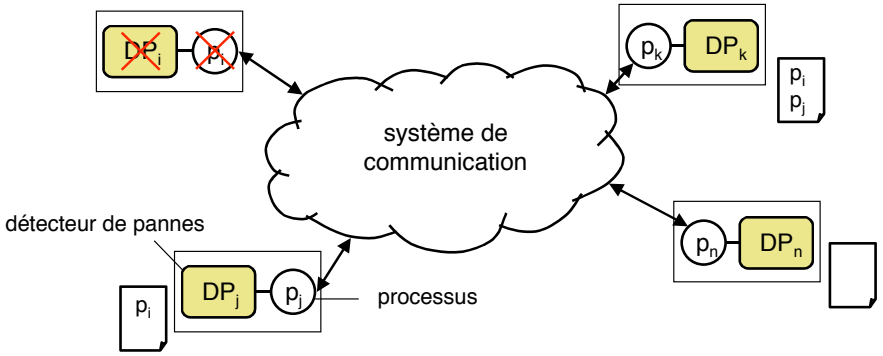
A voir plus tard : extension aux pannes byzantines

Détecteurs de pannes imparfaits

- **Motivation et démarche**
  - La source de l'impossibilité du consensus en asynchrone est qu'il est impossible de distinguer un processus en panne d'un processus lent (ou dont les messages sont retardés). En d'autres termes, on ne dispose pas d'un détecteur de pannes parfait
  - On va donc construire des détecteurs imparfaits, en essayant de quantifier l'écart entre ces détecteurs et le détecteur parfait (inaccessible)
- **Résultats**
  - Le consensus est-il possible en asynchrone avec un détecteur imparfait ? La réponse est oui !
    - On sait même caractériser le "plus faible" détecteur imparfait permettant le consensus
    - Un tel détecteur n'est pas réalisable en asynchrone (FLP), mais on peut en construire des approximations réalistes

T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, *Journal of the ACM*, vol. 43, 2, 1991
T. D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, *Journal of the ACM*, vol. 43, 4, July 1996

Qu'est-ce qu'un détecteur de pannes ?



Un détecteur de pannes est un service réparti composé de détecteurs locaux attachés à chaque processus (ou site) . Un détecteur fournit sur demande la liste des processus qu'il soupçonne d'être défaillants. Les divers détecteurs locaux coopèrent entre eux pour établir cette liste

Les qualités d'un détecteur de pannes

- **Complétude (*completeness*)**
  - Le détecteur doit détecter les processus fautifs. Plus précisément :
    - Complétude forte : tout processus fautif finit par être soupçonné par tout processus correct
    - Complétude faible : tout processus fautif finit par être soupçonné par un processus correct
- **Exactitude (*accuracy*)**
  - Le détecteur ne doit pas déclarer fautif un processus correct. Plus précisément :
    - Exactitude forte : aucun processus correct n'est jamais soupçonné par un autre processus correct
    - Exactitude faible : il existe un processus correct qui n'est jamais soupçonné par un autre processus correct
  - On peut encore affaiblir ces propriétés en considérant le temps
    - Exactitude finalement forte : au bout d'un certain temps, aucun processus correct n'est plus jamais soupçonné par un processus correct
    - Exactitude finalement faible : au bout d'un certain temps, il existe un processus correct qui n'est plus jamais soupçonné par un autre processus correct

  NB : "finalement" traduit *eventually* ; on peut aussi dire "inévitablement"

Les classes de détecteurs de pannes

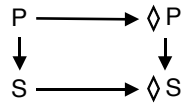
Hypothèse : pannes franches sans reprise, communication fiable asynchrone

exactitude

complétude	forte	faible	finalement forte	finalement faible
forte	P	S	◊P	◊S
faible	Q	W	◊Q	◊W

A priori, 8 classes de détecteurs. En réalité, complétude forte et faible sont équivalentes. On sait réaliser la complétude forte si on dispose de la complétude faible (cf plus loin). Donc 4 classes distinctes : P, ◊P, S, ◊S

La flèche signifie : "implique" (si on assure P, on assure S, etc.



Complétude faible vs complétude forte

Hypothèse : pannes franches, communication fiable asynchrone

Supposons que l'on dispose d'un détecteur assurant la complétude faible. Ce détecteur garantit que tout processus fautif finit par être soupçonné par un processus correct.

Donc au bout d'un temps fini T, un processus correct, soit p, possède la liste de tous les processus fautifs. Si tout processus diffuse périodiquement sa liste (par diffusion fiable), tous les processus fautifs seront soupçonnés au bout d'un temps fini par tout processus correct (d'après les propriétés de la diffusion fiable).

Donc complétude faible implique complétude forte. Comme on a aussi l'implication inverse, les deux formes de complétude sont équivalentes

Cette propriété repose sur la possibilité (vue plus haut) de réaliser la diffusion fiable en asynchrone avec pannes franches

Le détecteur de pannes parfait

- **Qualités**
  - Un détecteur parfait (de classe P) possède les qualités
    - de complétude forte
    - d'exactitude forte
- **Capacités**
  - un détecteur parfait permet de résoudre le consensus en asynchrone avec pannes franches (cf plus loin) …
  - … donc (FLP) un détecteur parfait n'est pas réalisable en asynchrone avec pannes franches
- **La suite…**
  - Nous allons voir qu'un détecteur n'a pas besoin d'être parfait pour résoudre le consensus
  - Nous allons caractériser la classe des détecteurs imparfaits (toujours non réalisables) qui y parviennent
  - En pratique, on essaie de réaliser une approximation d'un tel détecteur

Consensus avec détecteur parfait

Hypothèse :
Pannes franches sans restauration, communication fiable asynchrone
Soit f le nombre maximal de défaillances auxquelles on veut résister

Structures de données :
Chaque processus p_i entretient un vecteur $V_i[1..n]$ contenant l'ensemble (tel qu'il est connu de p_i) des valeurs proposées par les processus. Donc $V_i[j]$ est la valeur proposée par p_i , telle qu'elle est connue de p_i.

Initialement, tous les composants du vecteur contiennent nil, sauf pour $V_i[i] = v_i$ (la valeur proposée par p_i lui-même).

Le vecteur se remplit au fur et à mesure que p_i a connaissance des valeurs proposées par les autres processus

source : démonstrations, code et exemples pour détecteurs P et S
cours de R. Baldoni : www.dis.uniroma1.it/~baldoni/

Consensus avec détecteur parfait : principe

Principe : L'algorithme fonctionne avec des tours successifs asynchrones. Il comporte deux phases

Phase 1 : diffusion (f+1 tours)

Chaque processus p_i envoie à tous les valeurs de son vecteur V_i différentes de nil qu'il n'a pas déjà diffusées. Ainsi, au premier tour, il n'envoie que sa propre valeur. A chaque tour, il envoie les valeurs qu'il a reçues depuis le tour précédent. Il s'agit d'envois simples (pas d'algorithme de diffusion)

Chaque processus p_i reçoit les valeurs envoyées par les autres processus. Il attend de recevoir un message de tous les processus qui ne sont pas sur la liste des suspects fournie par son détecteur de pannes de classe P.

Phase 2 : choix

Après f + 1 tours (f = nombre max. de défaillances tolérées), chaque processus décide la première valeur non égale à nil de son vecteur V

Consensus avec détecteur parfait : algorithme

Programme du processus p_i

V_i := [nil, nil, … nil] ; $V_i[i] := v_i$; $Delta_i := V_i$

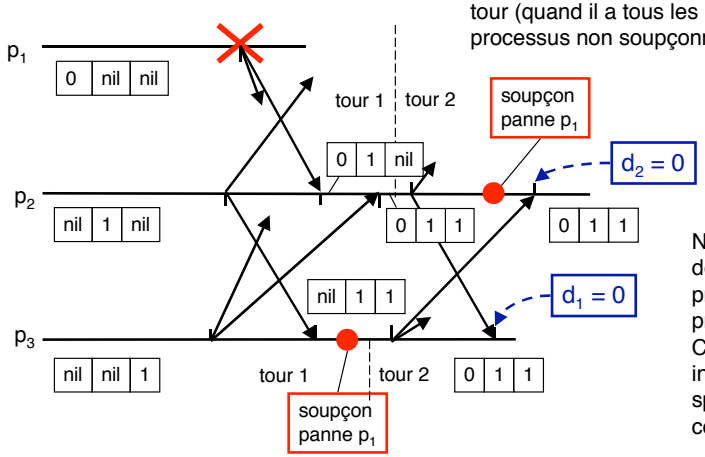
for r_i = 1 .. f + 1 **do** // f : nombre max de processus défaillants
  send([r_i, $Delta_i$, i]) to all ;
  wait until (**for all** j : recv([r_i, $Delta_j$, j]) **or** j ∈ DP_i); // consulte DP_i
  // DP_i : liste des suspects fournie par le détecteur (P) du processus i
  $Delta_i$ = [nil, nil, … nil] ; // $Delta_i$: différence avec tour précédent
  for k = 1 .. n **do**
    if received [r_j, D, j] such that (D[k] ≠ nil **and** $V_i[k]$ = nil)
      then $V_i[k]$: = $Delta_i[k]$: = D [k]

d_i := first entry $V_i[j]$ ≠ nil

Remarque : chaque composant d'un vecteur V_i est **stable** : il est mis à jour au plus une fois, et si sa valeur n'est pas nil, elle ne sera plus modifiée.

Consensus avec détecteur parfait : exemple

On suppose f = 1, donc 2 tours
chaque processus décide à l'issue du 2ème tour (quand il a tous les messages des processus non soupçonnés)



Noter que la valeur décidée est celle proposée par un processus défaillant. Ce n'est pas incompatible avec les spécifications du consensus

Consensus avec détecteur parfait : sûreté (1)

Accord : si un processus correct décide v, tous les processus corrects décident v

Supposons que deux processus corrects p_i et p_j décident deux valeurs différentes. Puisqu'ils ont décidé, ils ont terminé le tour f + 1 de l'algorithme avec deux vecteurs V_i et V_j différents. Supposons que V_i contienne une valeur x non présente dans V_j. Il y a alors deux possibilités :

a) x a été transmis à p_i par un processus correct ; celui-ci a été soupçonné par p_j, qui n'a donc pas attendu de message de lui

b) un processus p_k a envoyé la valeur x à p_i au dernier tour mais est tombé en panne avant de l'envoyer à p_j

Le cas a) est contradictoire avec la propriété d'exactitude du détecteur P : aucun processus correct n'est soupçonné. Reste à traiter le cas b)

Consensus avec détecteur parfait : sûreté (2)

Accord (suite)

V_i contient une valeur x non présente dans V_j

Cas b) Un processus p_k a envoyé la valeur x à p_i au dernier tour mais est tombé en panne avant de l'envoyer à p_j.

Puisque p_k connaît cette valeur, elle lui a été envoyée lors d'un tour t antérieur par un autre processus, et celui-ci est tombé en panne avant de pouvoir l'envoyer à p_j. Ce tour t est nécessairement le tour f, car sinon p_k (alors non défaillant) aurait transmis x à p_j dans un tour intermédiaire entre t et f+1. Par récurrence, puisque p_j n'a jamais eu connaissance de x, alors nécessairement lors de chacun des tours précédents le processus qui devait envoyer x à p_j est tombé en panne avant de pouvoir le faire.

Mais alors f + 1 processus sont tombés en panne. Contradiction avec l'hypothèse de f pannes au maximum.

Consensus avec détecteur parfait : sûreté (3)

Intégrité et validité : un processus décide au plus une fois, et s'il décide v, alors un processus a proposé v

La valeur décidée par un processus p_i est nécessairement un composant de son vecteur V_i. Mais tout $V_i[j]$, $j \neq i$ n'est mis à jour que lors de l'arrivée d'un message portant un Delta venant d'un autre processus, et et les éléments de Delta sont des copies d'éléments de V_k. Par récurrence, seules sont propagées les valeurs constituant les V_k initiaux, qui sont les valeurs proposées, et les messages ne sont pas altérés par le système de communication. Enfin $V_i[i]$ est la valeur proposée par p_i lui-même

D'autre part, l'algorithme comporte un seul point de décision, à la fin, hors de toute boucle. Un processus ne peut franchir ce point qu'une fois au plus.

Noter que les raisonnements utilisent implicitement la stabilité des valeurs des composants des V_i

Consensus avec détecteur parfait : vivacité

Tout processus correct décide

Supposons qu'un processus p_i correct ne décide pas. Alors nécessairement ce processus est en attente indéfinie sur l'instruction :

wait until (**for all** j : recv([rj, $Delta_j$, j]) **or** $j \in DP_i$);

qui est le seul point de blocage potentiel de l'algorithme.

Donc un message venant d'un processus p_j non actuellement soupçonné par p_i n'est pas arrivé. Deux cas sont possibles :

- p_j est correct. Alors son message finira par arriver (propriété du système de communication asynchrone, mais fiable)
- p_j est défaillant. Alors il finira par être sur la liste des processus soupçonnés par p_i (propriété de complétude du détecteur de défaillances parfait)

Il y a contradiction dans les deux cas. Donc l'attente indéfinie est impossible

Consensus avec détecteur parfait : remarques

Notes sur la preuve

Notons que la preuve utilise les deux propriétés du détecteur de pannes P : complétude forte, exactitude forte.

- L'exactitude a servi à éliminer les fausses suspicions
- La complétude a servi à lever l'incertitude sur les processus fautifs

Dans les deux cas, on élimine une source d'attente

La suite…

- Pour les algorithmes utilisant S et ◊ S, l'affaiblissement porte sur l'exactitude. Pour S, on sait qu'un processus correct n'est jamais soupçonné ; donc on doit faire n — 1 tours pour être certain de le voir.
- Pour ◊ S, l'exactitude est seulement inévitable (⟨◊⟩) ; pour l'assurer, on utilise un coordinateur dont le rôle "tourne", jusqu'à tomber sur un processus qui n'est (finalement) plus soupçonné.

Consensus avec détecteur S : principe

S assure la complétude forte (tout processus fautif sera soupçonné par tout processus correct) et l'exactitude **faible** (il y a un processus correct qui n'est jamais soupçonné)

On part de l'algorithme utilisant P. Une phase de plus est nécessaire

Phase 1 : identique à celle de l'algorithme avec P (mais n — 1 tours)
A la fin de cette phase, chaque processus a un vecteur, mais ces vecteurs ne sont pas identiques car des processus corrects ont pu être soupçonnés

Phase 2 (nouvelle) : chaque vecteur des valeurs proposées est envoyé à tous les processus. Un processus p_i attend les valeurs proposées par tout processus qu'il ne soupçonne pas, et il met à jour son vecteur V_i. A la fin de cette phase, tous les processus corrects ont le même vecteur. Le k-ième élément contient la valeur v_k proposée par p_k, ou bien nil. L'un des v_k au moins n'est pas nil

Phase 3 (décision) : choix de la valeur décidée, comme avec P

Consensus avec détecteur S : algorithme (1)

Programme du processus p_i : phase 1

V_i := [nil, nil, … nil] ; $V_i[i] := v_i$; $Delta_i := V_i$

for r_i = 1 .. n – 1 **do** // n : nombre total de processus
  send([r_i, $Delta_i$, i]) to all ;
  wait until (**for all** j : recv([r_i, $Delta_j$, j]) **or** j ∈ DP_i); // consulte DP_i
  // DP_i : liste des suspects fournie par le détecteur (P) du processus i
  $Delta_i$ = [nil, nil, … nil] ; // $Delta_i$: différence avec tour précédent
  for k = 1 .. n **do**
    if received [r_j, D, j] such that (D[k] ≠ nil **and** $V_i[k]$ = nil)
      then $V_i[k]$: = $Delta_i[k]$: = D [k]

A la fin de la phase 1, chaque processus a un vecteur V des valeurs proposées, mais ces vecteurs ne sont pas identiques car des processus corrects ont pu être soupçonnés (ce qui n'était pas le cas avec P)

Consensus avec détecteur S : algorithme (2)

Programme du processus p_i : phase 2

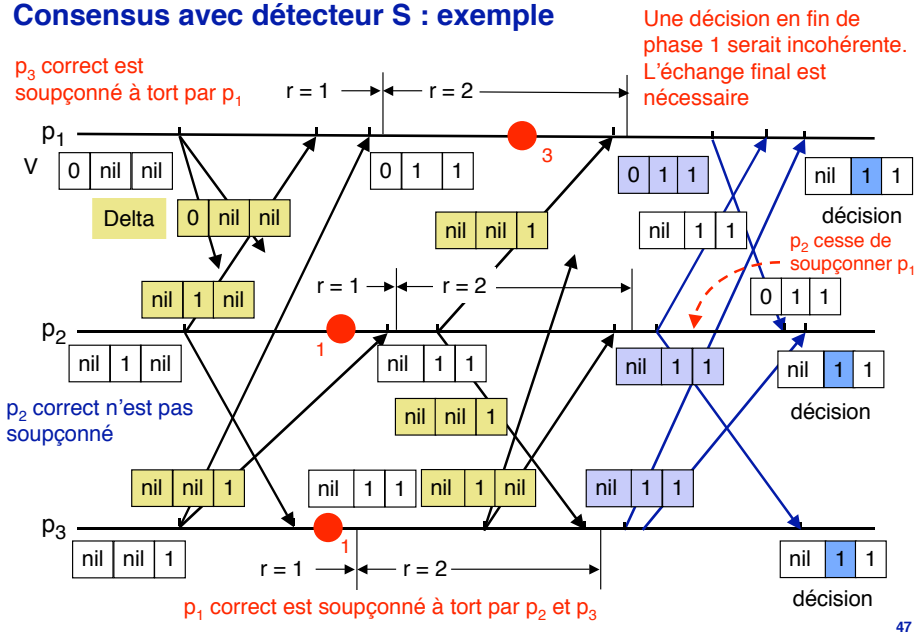
send([r_i, V_i, i]) to all ;
wait until (**for all** j : recv([r_i, V_j, j]) **or** j ∈ DP_i); // consulte DP_i
// DP_i : liste des suspects fournie par le détecteur (P) du processus i
for k = 1 .. n **do**
  if (received [n, V, j] such that V[k] = nil)
    then $V_i[k]$:= nil

A la fin de la phase 2, tous les vecteurs sont identiques ; les valeurs proposées par des processus corrects soupçonnés à tort peuvent ne pas y figurer (vue pessimiste), mais y figure nécessairement (au moins) la valeur proposée par un processus correct (celui qui n'est pas soupçonné)

Programme du processus p_i : phase 3

d_i := first entry $V_i[j]$ ≠ nil // un tel composant existe

Consensus avec détecteur S : exemple



Consensus avec détecteur S : éléments de preuve

Raisonnement informel (peut être rendu rigoureux, cf Chandra-Toueg 91)

A la fin de la phase 1, chaque processus correct p_i posède un vecteur V_i contenant au moins une valeur ≠ nil ($V_i[i] = v_i$), et des valeurs nil pour ceux des processus que p_i a soupçonnés (à raison ou à tort). Ces vecteurs ne sont pas identiques, mais pour tout k, ou bien $V_j[k] = v_k$, ou bien $V_j[k]$ = nil (si p_k a été soupçonné par p_j)

Dans la phase 2, les processus échangent leurs vecteurs. Chaque processus corrige son vecteur de manière pessimiste : si p_k a été soupçonné par un seul processus, alors $V_i[k]$ = nil pour tout i. Si p_m est un processus correct qui n'a jamais été soupçonné (il y en a au moins un), alors $V_i[m] = v_m$ ≠ nil pour tout i. Tous les vecteurs sont donc identiques en fin de phase 2 et contiennent au moins un élément ≠ nil

La preuve de vivacité est la même que pour P : un processus ne peut attendre indéfiniment sur un point de blocage. En conséquence, tout processus atteint la phase 3 en temps fini, et décide la même valeur ≠ nil

Consensus avec détecteur ◊ S : principe (1)

L'algorithme utilisant S ne peut pas se transposer à ◊ S. En effet, on ne peut plus garantir en fin de phase 2 qu'il existe un processus correct qui n'est pas soupçonné. La clause ◊ (finalement) provoque la perte de la vivacité.

L'algorithme avec ◊ S utilise un principe complètement différent : celui du **coordinateur tournan**t. Chaque processus sert de coordinateur à tour de rôle et cherche à construire une décision à l'aide d'une majorité (cf Paxos). La vivacité est assurée par la propriété d'exactitude faible : au bout d'un temps fini, un processus correct non soupçonné deviendra coordinateur et pourra alors décider (à condition qu'une majorité de processus ne soient pas en panne).

Différence avec Paxos (qui utilise aussi un coordinateur) : la propriété d'exactitude faible simplifie beaucoup le choix du coordinateur (rotation au lieu d'élection) et garantit son unicité à tout instant, et la terminaison. Paxos a des hypothèses de défaillance plus faibles, mais peut ne pas décider. Le nombre de défaillances tolérées est le même pour les deux algorithmes (< ⌈n/2⌉)

Consensus avec détecteur ◊ S : principe (2)

L'algorithme nécessite une majorité de processus corrects. Si on veut tolérer f défaillances, il faut au moins 2f + 1 processus

L'algorithme comporte un nombre non défini à l'avance de tours successifs. Un coordinateur est désigné lors de chaque tour ; son identité c est déterminée par le numéro r du tour : c = (r **mod** n) + 1 et donc connue de tous les processus. Un coordinateur décide

- quand il est correct et non soupçonné (ce qui finit par se produire en vertu de l'exactitude faible)
- s'il réunit une majorité de processus corrects (assuré par la condition n ≥ 2f + 1)

Il diffuse alors (de manière fiable) sa décision, qui est adoptée par tous les processus corrects

Consensus avec détecteur ◊ S : principe (3)

Chaque tour comporte 4 phases

Phase 1. Chaque processus envoie au coordinateur (connu) son estimation de la valeur décidée, estampillée par le numéro du dernier tour où elle a été mise à jour. L'estimation initiale, pour un processus, est la valeur qu'il propose.

Phase 2. Le coordinateur réunit une majorité de valeurs estimées (au moins ⌈n +1/2⌉ valeurs), choisit la plus récente (estampille max.) et l'envoie à tous les processus ; cette valeur devient la nouvelle estimation

Phase 3. Chaque processus correct réagit comme suit à cet envoi

- s'il ne soupçonne pas le coordinateur, il lui renvoie ack, et adopte la valeur reçue comme nouvelle estimation de la décision
- s'il soupçonne le coordinateur, il lui renvoie nack, et ne modifie pas son estimation

Phase 4. Le coordinateur recueille les réponses (ack ou nack). Si majorité de ack, la valeur v est fixée (et stable). Il la diffuse (diffusion fiable) à tous les processus, qui décident cette valeur, et l'algorithme se termine. Si pas de majorité, pas de décision, on passe au tour suivant avec un autre coordinateur.


```
upon propose(v)                          // programme processus p_i        } initialisation
  r:=0                                   // tour courant                   }
  t:=0                                   // dernier tour où v a été mis à jour }
  while not decided do                                                     } phase 1
    c := (r mod n) + 1                   // p_c est le coordinateur        }
    send (vote, r, v, t) to p_c          //                                }
    if i = c then                        // comportement du coordinateur   } phase 2
      wait until (receive (vote, r, r, v', t') from ⌈n+1⌉/2 processes)    }
      maxt := largest t' received                                          }
      v := some v' received with t' = maxt                                 }
      send (propose, r, v) to all        // v est la nouvelle estimation   }
    wait until (receive (propose, r, v') from p_c or c ∈ DP_i )            } phase 3
    if a (propose, r, v') message was received then                        }
      v := v' ; t := r                   // mise à jour estimation         }
      send (ack) to p_c                  // message accepté                }
    else send (nack) to p_c              // p_i soupçonne le coordinateur  }
    if i = c then                        // comportement du coordinateur   } phase 4
      wait until (receive ack or nack messages from ⌈n+1⌉/2 processes)     }
      if all are ack then send (decide, v) to all                          }
    r := r + 1                           // changement de tour             }
upon receiving (decide, v')              // algorithme de la diffusion fiable
  if not decided then
    send (decide, v') to all
    decide(v')
```

Consensus avec détecteur ◊ S : remarques

Consensus uniforme.

On peut montrer que le détecteur ◊ S permet de réaliser non seulement le consensus, mais le **consensus uniforme**, dans lequel la propriété d'accord devient :

Accord uniforme : deux processus (corrects ou non) ne peuvent décider des valeurs différentes

Le consensus uniforme est utilisé plus loin pour la validation atomique non bloquante et pour les groupes à vues synchrones

Pannes avec réinsertion.

Un détecteur amélioré ◊ S_u a été introduit pour résoudre le consensus lorsque des processus peuvent se réinsérer après réparation (M. Aguilera, W. Chen, S. Toueg. Failure detection and consensus in the crash-recovery model, *Proc 12th Int. Symp. on Distributed Computing*, 1998)

Détecteurs de panne : résumé

Le consensus peut être résolu en système asynchrone avec pannes franches si on dispose de détecteurs de pannes. Soit n le nombre de processus.

Détecteur parfait P (complétude forte, exactitude forte) : tolère jusqu'à n — 1 défaillances, nécessite f +1 tours (f : nombre de défaillances tolérées)

Détecteur S (complétude forte, exactitude faible) : tolère jusqu'à n — 1 défaillances, nécessite n tours

Détecteur ◊ S (complétude forte, exactitude finalement faible) : tolère jusqu'à ⌈n/2⌉ — 1 défaillances, le nombre de tours est fini, mais non borné

Peut-on aller au-delà ? La réponse est **non** (Chandra, Hadzilacos, Toueg - 1996) : le détecteur ◊ W (donc son équivalent ◊ S) est **le plus faible** capable de résoudre le consensus en asynchrone avec pannes franches. Cela signifie que si on affaiblit une des propriétés de ◊ W (complétude faible, exactitude finalement faible), le consensus n'est plus résoluble.

Détecteurs de pannes : aspects pratiques (1)

Les détecteurs de pannes P, ◊ P, S, ◊ S permettent de résoudre le consensus en système asynchrone. En conséquence (FLP), **ils ne sont pas réalisables** en asynchrone par un algorithme déterministe.

Pour une réalisation approchée, il faut lever l'hypothèse d'asynchronisme et estimer des délais de garde pour détecter la panne d'un processus. Notons δ une borne supérieure estimée pour le temps de transmission d'un message.

Soit, pour un processus p, à déterminer si un processus q est en panne. Dans la pratique, on utilise deux techniques.

- **Active** (*ping*). p envoie périodiquement un message à q, demandant un acquittement. Sans réponse au bout du délai 2 δ, p déclare q défaillant
- **Passive** (*heartbeat*). q diffuse périodiquement un message "q est vivant". On utilise en pratique des horloges physiques synchronisées. Soit T l'heure prévue d'émission d'un signal de q ; si p n'a pas reçu ce signal à T + δ (test périodique), alors p déclare q défaillant

Détecteurs de pannes : aspects pratiques (2)

La méthode passive (*heartbeat*) est la plus utilisée dans la pratique, en régime permanent (par exemple surveillance mutuelle dans un groupe de serveurs). La méthode active (*ping*) est utilisée lorsqu'on recherche l'état d'un processus particulier dont on est sans nouvelles.

Quelle que soit la méthode utilisée, l'estimation du délai δ n'est pas infaillible si le système est asynchrone. Deux risques :

- δ trop petit : fausses détections de pannes
- δ trop grand : temps trop long avant détection

Dans les deux cas, l'exécution de l'application peut être ralentie ou même compromise. Le choix dépend de l'impact d'une détection erronée dans un sens ou dans l'autre.

Détecteurs de pannes : aspects pratiques (3)

Etant donné l'incertitude sur la bonne estimation des temps de transmission, une méthode plus raffinée utilise l'adaptation dynamique du délai de garde. On utilise une méthode passive (*heartbeat*). Pour un processus p, un processus q dont le signal manque après un délai de garde est soupçonné d'être défaillant (ajouté sur la liste des suspects maintenue par p). En cas de fausses détections répétées, on augmente le délai de garde. En cas de détection tardive, on le diminue. Diverses variantes selon la rapidité de réaction, etc.

Noter que l'utilisation de délais de garde assure la complétude forte, mais non l'exactitude. Une technique pour assurer l'exactitude consiste à tuer (ou considérer comme définitivement défaillant) tout processus soupçonné (simulant ainsi un détecteur de classe P), mais elle n'est évidemment viable que si la probabilité de fausse détection est très faible.

Détecteurs de pannes : conclusion

La notion de détecteur de pannes imparfait a deux apports :

- **théorique** : elle a fait progresser la compréhension profonde de la difficulté du consensus ; elle a intégré et simplifié des résultats antérieurs (ainsi les classes de détecteurs englobent les différents cas de "synchronisme partiel") ; elle a permis de clarifier la relation entre le consensus et la validation (cf plus loin)
- **pratique** : elle sert de base à la réalisation et à l'évaluation de détecteurs utilisables en pratique, approximations de détecteurs imparfaits ; ces détecteurs peuvent aussi servir à résoudre d'autres problèmes liés au consensus (groupes, validation non bloquante)

Comparaison entre détecteurs et Paxos : question ouverte…