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

Consensus - 2

Pannes byzantines
Consensus et validation

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

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

Plan

Pannes byzantines
- synchrones
- asynchrones

Compléments sur le consensus
- consensus uniforme
- consensus et validation non bloquante
- consensus et tolérance aux fautes

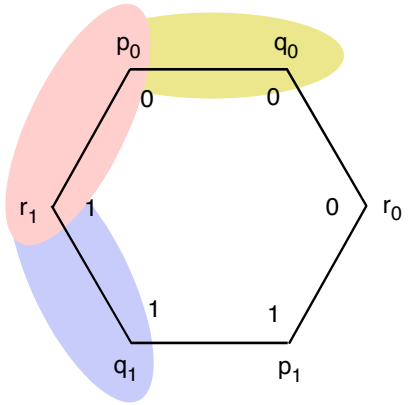
Pannes byzantines

- **Rappel**
  - Panne byzantine (ou arbitraire) : comportement quelconque, y compris “malveillant”
- **Pourquoi considérer les pannes byzantines ?**
  - Intérêt théorique : mode de défaillance le plus général ; toute solution à un problème de pannes byzantines est universellement applicable
  - Intérêt pratique : application aux conditions extrêmes (environnement hostile, besoin élévé de disponibilité)
- **Plan**
  - Consensus avec communication synchrone
  - Consensus avec communication asynchrone

Un résultat d’impossibilité

Il est impossible de résoudre le consensus entre 3 processus si un seul d’entre eux a un comportement byzantin (plus généralement, 3f et f)

Supposons le consensus possible entre p, q, r. Soit p_0, q_0, r_0, p_1, q_1, r_1 copies de p, q, r



Scénario a) p_0, q_0, r_0 proposent 0
p_1, q_1, r_1 proposent 1

Scénario a') p et q corrects, r fautif
p et q doivent décider 0 dans a' donc dans a

Scénario a'') q et r corrects, p fautif
q et r doivent décider 1 dans a'' donc dans a

Scénario a''') p et r corrects, q fautif
p et r doivent décider la même valeur dans a'''
donc dans a. Contradiction !

Consensus synchrone avec pannes byzantines (1)

Hypothèses

Système de communication assurant la délivrance synchrone des messages (soit δ une borne supérieure du temps de transmission)
Processus avec pannes byzantines (comportement arbitraire)

Le problème

On ramène le problème du consensus à celui de la diffusion fiable temporisée : diffusion fiable (tout ou rien, accord) + temps borné. Si chacun diffuse ainsi sa valeur aux autres, le choix peut se faire après un temps fixé, sur le même ensemble de valeurs

Résultat (Lamport, Shostak, Pease, 1982)

On sait résoudre le consensus synchrone pour **f** pannes byzantines avec **3f + 1** processus. Cette limite est stricte (impossible avec 3f)

Consensus synchrone avec pannes byzantines (2)

Algorithme pour n processus. Soit f le nombre de pannes tolérées. On exécute P(f), l'algorithme P(i) étant défini récursivement comme suit.

P(0)

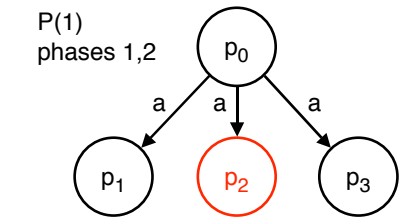
1. L'émetteur envoie val à tous les n — 1 destinataires p_i
2. Si p_i reçoit val, alors v_i = val, sinon v_i = def (valeur par défaut) (rappel : borne δ sur temps transmission)

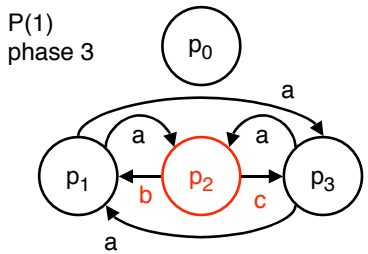
P(m), m > 0

1. L'émetteur p_0 envoie val à tous les destinataires p_i
2. Si p_i reçoit val, alors v_i = val, sinon v_i = def
3. p_i agit comme émetteur en exécutant P(m — 1) vers les n — 2 destinataires avec val = v_i
4. Pour tout i, et tout j ≠ i : soit v_j la valeur que p_i reçoit de p_j lors de P(m — 1) phase 2 ; v_j = def si pas de valeur reçue (avant δ).
5. val = valeur majoritaire dans $\{v_0, v_1, v_{i-1}, v_{i+1}, \ldots v_{n-1}\}$; si pas de majorité, val = def. La valeur décidée dans ce tour est val

Consensus synchrone avec pannes byzantines : exemple (1)

Algorithme pour 4 processus, 1 panne tolérée
p_0 émetteur, p_1, p_2, p_3 récepteurs. On suppose p_2 byzantin

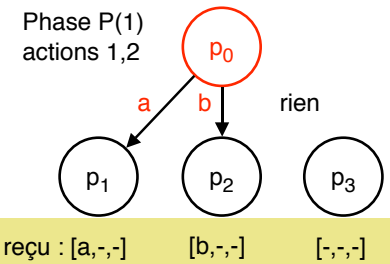


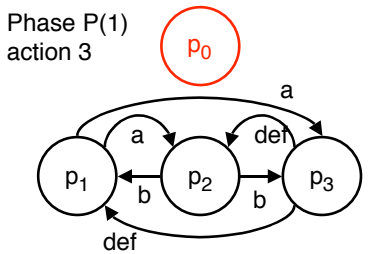


	reçu : [a,-,-]	[a,-,-]
phase 4	reçu : [a, b, a]	[a, a, c]
phase 5	décide : a	a

Consensus synchrone avec pannes byzantines : exemple (2)

Algorithme pour 4 processus, 1 panne tolérée
p_0 émetteur, p_1, p_2, p_3 récepteurs. On suppose p_0 byzantin





	reçu : [a,-,-]	[b,-,-]	[-,-,-]
action 4	reçu : [a, b, def]	[b, a, def]	[def, a, b]
action 5	décide : def	def	def

Consensus synchrone avec pannes byzantines

Conclusion

Le consensus est résoluble en synchrone avec une redondance de 3f + 1

L'algorithme est néanmoins peu efficace

au moins f + 1 tours
complexité exponentielle en nombre de bits transmis

L'algorithme peut être amélioré (redondance f + 1) si les messages sont authentifiés, et nécessite f + 1 tours au pire

La complexité est inhérente au comportement arbitraire

Consensus asynchrone avec pannes byzantines

Le consensus étant impossible en asynchrone avec pannes franches, il est a fortiori impossible avec pannes byzantines

Néanmoins des solutions imparfaites de type *best effort* peuvent être trouvées. Le point intéressant est que la redondance requise reste 3f + 1, comme en synchrone.

Algorithme ancien (Bracha, Toueg, 1985) pour la diffusion fiable
Si l'émetteur est correct, tous les récepteurs corrects délivrent la valeur correcte en temps fini (non borné)
Si l'émetteur est fautif (byzantin)
- ou bien tous les récepteurs corrects délivrent la même valeur
- ou bien aucun ne décide et aucune valeur n'est délivrée

Algorithme récent (Castro, Liskov, 2000)
Extension de Paxos - intérêt pratique
Améliorations diverses (travaux en cours) : rapidité vs redondance

Diffusion asynchrone avec pannes byzantines (1)

Diffusion en système asynchrone (communication fiable), pannes byzantines.

Soit **n** le nombre de processus, **f** le nombre maximal de pannes tolérées

On suppose **n ≥ 3f + 1**

Description informelle

Trois phases successives, avec 3 types de messages : initial, echo, ready

L'émetteur commence par envoyer à tous un message initial, pour lancer l'algorithme, puis (echo,v) [v = la valeur diffusée]

Chaque destinataires transmet à tous la valeur reçue avec un message (echo, v). Si un processus a reçu plus de (n+f)/2 messages (echo,v), ou plus de f messages (ready, v) il transmet à tous [y compris lui-même] un message (ready, v)

Si un processus a reçu 2f + 1 message (ready, v) avec la même valeur, il décide cette valeur

G. Bracha, S. Toueg, Asynchronous Consensus and Broadcast Protocols, *Journal of the ACM*, 32, 4, October 1985, pp. 824-840

Diffusion asynchrone avec pannes byzantines (2)

Preuve. On suppose que les valeurs possibles sont 0 et 1

Deux processus corrects p et q ne peuvent pas envoyer des messages ready avec des valeurs différentes. Supposons que ce soit possible. Alors p a reçu plus de (n + f)/2 messages (echo, 1) ou au moins f +1 messages (ready, 1). Parmi ces derniers, un au moins vient d'un processus correct car il y a au plus f fautifs. De même, q a reçu plus de (n + f)/2 messages (echo, 0) ou au moins un message (ready, 0) venant d'un processus correct. Donc (en raison des conditions pour l'envoi de ready), deux processus corrects différents ont reçu plus de (n + f)/2 messages (echo, 0) et plus de (n + f)/2 messages (echo, 1).

Mais l'intersection des deux ensembles de plus de (n+f)/2 processus ayant envoyé (echo, 0) et (echo, 1) contient plus de f processus (car la somme de leur tailles est supérieure à n+f, et il y a n processus au total). Donc cette intersection contient au moins un processus correct. Contradiction (un processus correct ne peut pas envoyer à la fois 0 et 1)

[à suivre]

Diffusion asynchrone avec pannes byzantines (3)

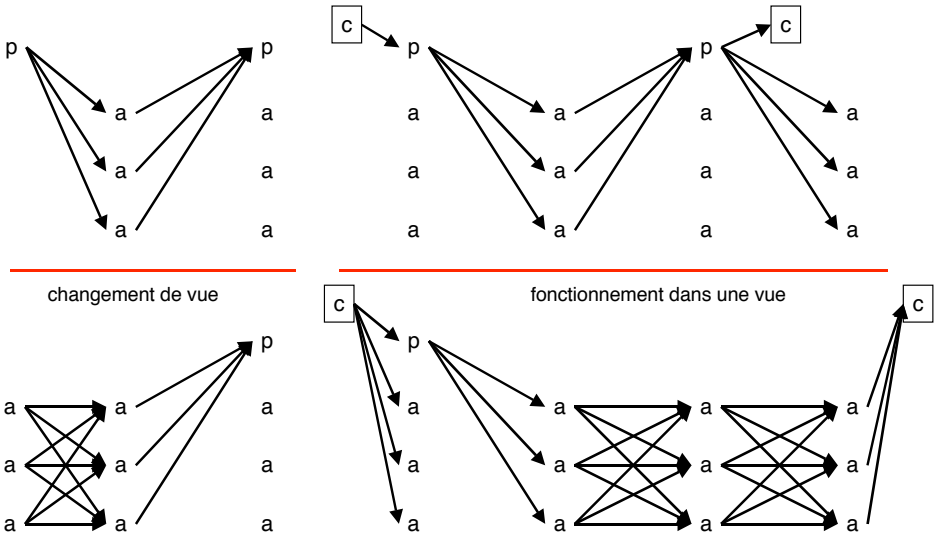
Deux processus corrects ne peuvent pas envoyer des messages (ready, v) avec des valeurs différentes. Un processus doit avoir 2f+1 messages (ready, v) pour décider v. Puisque n ≥ 3f+1, on a n – f ≥ 2f + 1. Mais comme il y a au moins n – f processus corrects, un de ces messages ready au moins a été envoyé par un processus correct. Donc deux processus corrects ne peuvent pas décider des valeurs différentes.

Si p décide v, il a reçu 2f + 1 messages (ready, v), dont au moins f + 1 venaient d'un processus correct. Donc tout processus correct va lui aussi recevoir au moins f + 1 messages et va envoyer un message (ready, v). Ainsi au moins n – f processus vont envoyer (ready, v), et tout processus correct recevra au moins 2f + 1 messages (ready, v), puisque n – f ≥ 2f + 1, et décidera v

Donc si un processus correct décide v, tous les processus corrects décident v.

Si l'émetteur est correct et transmet v, tous les processus corrects décident v.
Si l'émetteur est fautif, on peut seulement dire que tous les processus corrects, s'ils décident, décident la même valeur.

Comparaison entre Paxos classique et Paxos byzantin



d'après Lampson, The ABCDs of Paxos, *Proc. PODC*, 2001 , à lire pour analyse détaillée

Consensus uniforme

Dans le consensus uniforme (en asynchrone avec pannes franches), la condition d'accord devient :

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

On a les deux résultats suivants (Guerraoui, 1995)

Résultat 1: tout algorithme qui résout le consensus avec un détecteur de pannes de classe ◊P ou S ou ◊S résout également le consensus uniforme

Résultat 2 : il existe des algorithmes qui résolvent le consensus avec un détecteur de pannes de classe P, mais ne résolvent pas le consensus uniforme

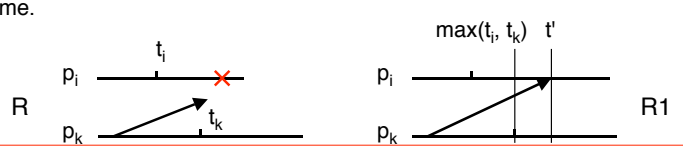
Intuitivement, dans un algorithme de consensus utilisant P, les processus corrects peuvent détecter sans erreur la panne d'un processus fautif, et peuvent décider indépendamment de la décision éventuelle de ce dernier

Résolution du consensus uniforme (1)

Tout algorithme qui résout le consensus avec un détecteur de pannes de classe ◊P, S, ◊S résout également le consensus uniforme

Preuve par l'absurde : supposons qu'il existe un algorithme A avec un détecteur D (◊P, S, ◊S) qui résout le consensus, mais non le consensus uniforme. Soit R une exécution particulière de A qui assure le consensus, mais non le consensus uniforme. On va montrer qu'il existe alors une exécution R1 de A qui ne résout pas le consensus.

Soit une exécution R de A qui assure le consensus mais non le consensus uniforme. Dans R, 2 processus p_i et p_k décident différemment et l'un d'eux au moins (soit p_i) est fautif. p_i décide v_i au temps t_i et p_k décide v_k au temps t_k ($v_i \neq v_k$). Soit l'exécution R1, avec les mêmes défaillances que R, sauf que p_i et p_k sont corrects dans R1. Dans R, on considère tous les messages envoyés par p_i et p_k et non reçus avant $\max(t_i, t_k)$. Dans R1, on retarde la réception de ces messages jusqu'à un temps $t' > \max(t_i, t_k)$; c'est possible à cause de l'asynchronisme.



Résolution du consensus uniforme (2)

Soit HD et HD1 l'histoire des détections de pannes avec D (liste des processus soupçonnés, en fonction du temps) dans R et R1 respectivement. On construit HD1 ainsi :
Jusqu'au temps t', HD1 est identique à HD
Après t', dans R1, aucun processus correct n'est plus soupçonné et tous les processus fautifs sont soupçonnés en permanence
HD1 vérifie donc la complétude forte, l'exactitude finalement forte et l'exactitude finalement faible. Comme HD1 ne contient pas plus de processus soupçonnés que HD, HD1 vérifie l'exactitude faible si D est de classe S. Donc HD1 peut être engendré par le détecteur utilisé

Pour p_i l'exécution de R1 est indistinguable de R jusqu'à t_i (p_i exécute les mêmes événements dans R et R1). De même pour p_k l'exécution de R1 est indistinguable de R jusqu'à t_k (p_k exécute les mêmes événements dans R et R1)
Donc, dans R1, p_i décide v_i et p_k décide $v_k \neq v_i$. Mais p_i et p_k étant corrects dans R1, R1 viole la condition d'accord du consensus, et l'algorithme A considéré ne résout pas le consensus, contrairement à l'hypothèse. Donc un tel algorithme A n'existe pas.

Résultat : tout algorithme qui résout le consensus avec un détecteur de classe ◊P, S, ◊S résout aussi le consensus uniforme (c'est le cas des algorithmes présentés précédemment pour S et ◊S).

Consensus et validation atomique

Résultat 1. La validation atomique non bloquante n'est pas réductible au consensus

Résultat 2. On peut définir une forme "faible" (mais acceptable en pratique) de validation non bloquante qui est réductible au consensus

validation non bloquante

Validité : La décision est soit valider, soit annuler

Intégrité : Tout processus décide au plus une fois (une décision est définitive)

Accord uniforme : Tous les processus (corrects ou non) qui décident prennent la même décision

Justification : Si la décision est valider, alors tous les processus ont voté OUI

Obligation : Si tous les processus votent OUI, et si tous les processus sont corrects, alors la valeur décidée est valider

Terminaison : tout processus correct décide au bout d'un temps fini

La validation faible remplace la condition d'obligation (ou non-trivialité) par la suivante :

Obligation faible : Si tous les processus votent OUI, et si aucun processus n'est soupçonné, alors la valeur décidée est valider

La VANB n'est pas réductible au consensus (1)

Dans un système asynchrone avec pannes franches, la validation atomique non bloquante ne peut pas être résolue avec un détecteur de pannes ◊P ou S (Guerraoui, 1995)

Donc la validation atomique non bloquante (VANB) n'est pas réductible au consensus

Preuve (par l'absurde)
Supposons qu'un algorithme A résolve la VANB avec un détecteur de classe ◊P ou S. Soit D de classe ◊P ou S et une exécution R de l'algorithme, où tous les participants votent oui. Un participant p_1 défaille immédiatement (sans avoir envoyé aucun message, et tous les autres sont corrects. Considérons un participant p_2 (correct). Il doit décider dans R (terminaison). Soit t le moment de sa décision. Deux cas possibles.

1) p_2 décide valider au temps t. Soit une exécution R1 identique à R, sauf que p_1 vote non. Alors p_2 voit exactement les mêmes événements dans R1 et R, et décide valider dans R1. Mais alors R1 viole la condition de justification.

[à suivre]

La VANB n'est pas réductible au consensus (2)

Soit t le moment de la décision de p_2 (correct) dans R. Deux cas possibles.

2) p_2 décide annuler au temps t. Soit une exécution R2, dans laquelle tous les participants sont corrects (y compris p_1) et la délivrance de tous les messages de p_1 est retardée jusqu'à un temps $t' > t$. Soit HD l'histoire des détections dans R et supposons que HD2 est identique à HD, sauf que p_1 n'est plus soupçonné dans R2 après t'. HD2 vérifie donc la complétude forte puisqu'il n'y a pas de pannes dans R2. Considérons l'exactitude :

- si D est de classe ◊P, HD vérifie l'exactitude finalement forte. D'après la définition de HD2, HD2 vérifie aussi cette propriété.
- si D est de classe S, HD vérifie l'exactitude faible : un participant correct p_k n'est jamais soupçonné dans HD. Comme p_1 est fautif dans R, $p_k \neq p_1$. Donc HD2 vérifie l'exactitude faible.

Jusqu'au temps t, p_2 exécute les mêmes événements dans R2 que dans R, donc décide annuler au temps t. Comme tous les participants sont corrects et ont voté oui dans R2, R2 viole la condition d'obligation.
Donc l'algorithme A n'existe pas.

Validation atomique non bloquante faible (1)

La validation faible remplace la condition d'obligation, ou non-trivialité, par la suivante : aucun processus soupçonné (au lieu de tous corrects)

Obligation faible : Si tous les participants votent OUI, et si aucun participant n'est soupçonné, alors la valeur décidée est valider

Résultat : la validation atomique non bloquante faible est réductible au consensus uniforme (donc au consensus). Algorithme (pour un participant p_i)

```
atomicCommitment (vote_i)
    send(p_i, vote_i) to all
    for j = 1 to n
        wait until [received(p_j, vote_j) or p_j ∈ D_i]
            if  p_j ∈ D_i or vote_i = no then
                decision = uniformConsensus(abort) ;
                return decision ;
    decision = uniformConsensus(commit) ;
    return decision ;
```


Validation atomique non bloquante faible (2)

```
atomicCommitment (vote_i)
    send(p_i, vote_i) to all
    for j = 1 to n
        wait until [received(p_j, vote_j) or p_j ∈ D_i]
            if  p_j ∈ D_i or vote_i = no then
                decision = uniformConsensus(abort)
                return decision
    decision = uniformConsensus(commit)
    return decision
```

Accord uniforme. Toute décision est obtenue par application du consensus uniforme. Donc il y a accord uniforme

Justification. Si la décision est valider, un processus a dû proposer valider au consensus. Il ne peut le faire que s'il a reçu un vote oui de tous (et qu'aucun n'est soupçonné)

Terminaison. Si un participant p reçoit un vote oui de tous, il lance le consensus uniforme. Même chose s'il reçoit au moins un vote non. Sinon, puisque tout participant correct a voté, il y a au moins un processus fautif, que p va finir par détecter (grâce à la complétude forte) et p va lancer le consensus. Comme celui-ci se termine, la validation se termine également.

Obligation. Si aucun participant n'est soupçonné et si tous les votes sont oui, alors tout participant qui lance le consensus propose valider. D'après la condtion de validité du consensus, tout participant correct décide valider.

Consensus et Validation

Deux problèmes fondamentaux d'accord dans les systèmes répartis

Pourquoi la validation non bloquante est-elle plus "difficile" ?

Réponse : la condition d'obligation (non-trivialité) de la VANB impose une connaissance forte des pannes (la condition dit "s'il n'y a pas de défaillances", ce qui implique un détecteur parfait). En revanche le consensus se contente d'une connaissance plus faible (détecteurs imparfaits)

Une spécification plus faible de la condition d'obligation ("si aucun participant n'est soupçonné" rend les deux problèmes équivalents tout en restant acceptable en pratique. La validation peut alors être résolue si on dispose d'un service de consensus

Autre exemple d'application : Validation avec Paxos
J.Gray, L. Lamport, Consensus on Transaction Commit, *Microsoft Research Tech. Report* MSR-TR-2003-96, April 2004

Consensus et tolérance aux fautes (1)

Le consensus est un **outil générique** pour la tolérance aux fautes

Un système informatique (déterministe) peut être considéré comme une machine à états finis.

(new state, output) = f(state, input)

Comment assurer la disponibilité d'un tel système ?

On réalise n copies identiques de la machine à états. Si ces n copies partent du même état initial et reçoivent la même séquence d'entrées, elles passeront par la même séquence d'états et fourniront la même séquence de sorties.

Il faut donc assurer la même séquence d'entrées. C'est possible si on sait réaliser le consensus (voir plus loin équivalence entre consensus et diffusion totalement ordonnée). Dans ce cas, on peut tolérer n—1 défaillances si on a n exemplaires du système initial. Néanmoins la réalisation du consensus lui-même peut imposer une redondance plus forte, selon les hypothèses de défaillance

Consensus et tolérance aux fautes (2)

La duplication totale de la machine à états est coûteuse. On peut envisager des solutions plus efficaces.

Une solution consiste à avoir une seule exécution active (par exemple, un seul processus pour une ressource particulière, représentée par la machine à états), mais alors le système est vulnérable à une défaillance de ce processus Donc on n'alloue la ressource que pour un temps déterminé ("bail"). A la fin de ce temps, ou bien le processus est correct, et on renouvelle le bail (implicitement ou explicitement) ou bien il est défaillant, et il faut reconstituer un état correct de la ressource. Il faut en pratique des horloges synchronisées (donc une hypothèse de synchronisme)

On retrouve indirectement l'utilisation du consensus pour l'allocation de la ressource (il faut déterminer quel processus la gère et quel processus traitera le cas de défaillance)

Exemple d'application : les deux solutions pour serveurs disponibles (redondance active et copie primaire)