Algorithmique distribuée

Accord & coordination

Eric Cariou

Master Technologies de l'Internet 1ère année

Université de Pau et des Pays de l'Adour
Département Informatique

Eric.Cariou@univ-pau.fr

Accord & coordination

- Une famille de problèmes en algorithmique distribuée
  - Coordination et accord (*agreement*) entre processus
- Catégories de problèmes de cette famille
  - Accord sur l'accès à une ressource partagée
    - Exclusion mutuelle
  - Accord sur un processus jouant un rôle particulier
    - Élection d'un maître
  - Accord sur une valeur commune
    - Consensus
  - Accord sur une action à effectuer par tous ou personne
    - Transaction
  - Accord sur l'ordre d'envoi de messages à tous
    - Diffusion atomique

Accord & coordination

- Architectures générales de solution
  - Avec un élément centralisateur par lequel toutes les communications et tous les messages transitent
    - Simplifie les algorithmes et les solutions
    - Mais point faible potentiel
  - Mise en oeuvre totalement distribuée
    - Processus identiques
      - Dont certains peuvent jouer un rôle particulier dans une interaction
    - Permet plus de facilité pour tolérance aux fautes
    - Mais algorithmes plus complexes
- A prendre en compte dans la définition d'un algorithme
  - Modèle de système distribué : synchrone ou asynchrone
  - Modèle de faute
    - Algorithmes souvent simples dans un contexte fiable mais complexes voire impossibles à définir dans un contexte non fiable

Diffusion

Diffusion

- Mode d'émission de messages avec multiples récepteurs
  - Un processus appartient à un groupe global de processus
  - Un message émis par un émetteur est reçu par plusieurs récepteurs
    - Tous les processus du groupe dans le cas de la diffusion (broadcast)
      - L'émetteur s'envoie aussi le message à lui même
    - Les processus d'un sous-groupe dans le cas du multicast
      - Avec possibilité que l'émetteur fasse ou pas partie du groupe des récepteurs et autres variantes ...
- Ordre et fiabilité de la diffusion des messages
  - Par défaut et selon les temps de propagation, les messages arrivent ... quand ils arrivent
    - Pas dans le même ordre ni dans un ordre donné pour tous les processus
  - Peuvent aussi ne pas arriver chez tous les processus si canaux de communication non fiables ou si l'émetteur plante en cours de diffusion

Diffusion : support de communication

- Coté récepteur, on différencie deux actions
  - La réception du message par le système de communication/un canal associé au processus
  - Le délivrement du message : le processus est informé du message
    - Un message reçu peut ne pas être délivré selon les cas
    - Le délivrement d'un message peut être retardé par rapport à sa réception tant que certaines conditions d'ordre ne sont pas respectées
- Mise en oeuvre peut être faite sur plusieurs types de supports de communication
  - Couche réseau réalisant nativement de la diffusion
    - Typiquement : multicast UDP/IP (mais non fiable)
  - Communication point à point « classique »

Protocole de diffusion basique

- Modèle de communication et de faute
  - Canaux de communication fiable 1 vers 1 entre tous les processus
  - Processus peuvent planter (panne franche)
  - Système asynchrone
- Protocole de diffusion basique, 2 opérations
  - B_broadcast(m) executé par processus *p*
    - Le processus *p* diffuse un message *m* à tous les autres processus
  - B_deliver(m) exécuté sur processus *p*
    - Délivrement du message *m* à *p*
- Opérations associées aux canaux de communication
  - receive(m, q) sur processus *p*
    - Réception du message *m* venant de *q*
  - send(m, q) sur processus *p*
    - Envoi par *p* du message *m* au processus *q*

Diffusion basique

- Mise en oeuvre diffusion basique
  - B_broadcast(m) sur *p*
    - On envoie le message à tout le monde (y compris à soi même)
      - Pour chaque processus *q* du groupe : exécuter send (m, q)
  - Lors d'un receive(m, q) sur *p*
    - On vient de recevoir un message *m* venant de *q*, on le délivre directement
      - Exécuter B_deliver(m)

Diffusion fiable

- Diffusion fiable (*reliable broadcast*)
  - Tous les processus corrects délivrent un message diffusé
- Caractéristiques à assurer
  - Validité
    - Si un processus correct diffuse *m*, alors il finira par être délivré par tous les processus corrects (en un temps fini)
      - Un message diffusé est délivré par tout le monde
  - Accord
    - Si un processus correct délivre *m*, alors tous les autres processus délivreront *m* également (en un temps fini)
      - Tout le monde délivre les mêmes message
  - Intégrité
    - Un processus correct délivre un message *m* au plus une fois, avec *m* un message qui a été diffusé par un processus
      - Pas de duplication ni de création de message

Diffusion fiable

- Le protocole de diffusion basique n'est pas fiable
  - Pourtant la communication est fiable
    - Pas de perte de messages
  - Le problème est pour l'exécution de B_broadcast(m)
    - Pendant que l'on parcourt la liste des processus pour leur envoyer un par un le message, le processus émetteur peut planter
      - Certains processus recevront le message mais pas tous
      - Les propriétés d'accord et de validité ne sont pas respectées
- Définition d'opérations de diffusion fiable
  - R_broadcast(m) et R_deliver(m)
    - Variantes fiables des opérations de la diffusion basique (R : reliable)
    - Utiliseront de la diffusion basique

Diffusion fiable

- Mise en oeuvre diffusion fiable
  - R_broadcast(m) sur *p*
    - Envoi le message à tout le monde via de la diffusion basique
      - Exécuter B_broadcast(m)
  - Lors d'un receive(m, q) sur *p*
    - Si on avait pas déjà reçu le message *m*, exécuter :
      1. Si on est pas l'émetteur du message ($p \neq q$) alors diffuser le message à tout le monde : exécuter B_broadcast(m)
      2. Se délivrer le message : exécuter R_deliver(m)
- Principes de l'algorithme
  - A la première réception d'un message, un processus le rediffuse à tous les autres, ce qui permet d'assurer que tout le monde le recevra même si certains plantent pendant un B_broadcast(m)
  - Inconvénients
    - Pour N processus, N diffusions basiques réalisées, très coûteux en nombre de messages : peu optimal mais fonctionne
    - Nécessité de vérifier qu'on a pas reçu déjà le message (avec un historique des messages ou des estampilles)

Diffusion : version uniforme

- Diffusion fiable
  - Propriétés (accord, validité, intégrité) à assurer pour les processus corrects (non plantés pendant ou avant la diffusion)
- Variante de la diffusion fiable
  - Diffusion fiable uniforme : prend en compte tous les processus y compris les plantés
    - Parce que certains processus peuvent avoir réalisé des actions suite à une diffusion puis planter ensuite
  - Propriétés d'accord, validité et intégrité sont les mêmes sauf qu'elles ne sont plus restreintes aux processus corrects

Diffusion fiable

- Preuve de la fiabilité, se base sur
  - Communication fiable : pas de perte de message ni de duplication
  - Un processus qui reçoit un message le rediffuse à tout le monde **puis** se le délivre
  - Au final, un processus délivre forcément un message qui a été diffusé par un processus et délivré sur tous les autres
- Diffusion basique ou fiable : ordonnancement
  - Aucune contrainte ou propriété par défaut sur l'ordre de délivrement des messages sur les processus
  - Mais des variantes peuvent assurer un certain ordre : FIFO, causal ou total (avec combinaisons)

Diffusion : ordonnancement

- Ordre FIFO
  - Si un processus diffuse un message *m* puis un message *m'*, alors un processus correct qui délivre *m'*, délivrera *m* avant *m'*
    - Si un processus plante après avoir diffusé *m*, la contrainte reste correcte bien qu'il ne délivrera pas *m'* : pas d'obligation de délivrement de *m'* (ni de *m*) pour tous les processus, contexte non fiable
- Ordre causal
  - Si la diffusion d'un message *m'* dépendait causalement de la diffusion d'un message *m*, alors un processus correct qui délivre *m'* délivrera *m* avant *m'*
- Ordre total
  - Si un processus correct délivre *m* avant de délivrer *m'*, alors tous les autres processus corrects délivreront *m* avant *m'*
    - Même ordre de délivrement des messages pour tous les processus corrects

Diffusion : ordonnancement

- Relations entre les ordres
  - L'ordre causal implique l'ordre FIFO
    - L'ordre FIFO est un ordre causal local : les diffusions lancées localement par le même processus sont en dépendances causales par principe
  - L'ordre total est indépendant des ordres FIFO ou causal
    - FIFO et causal : ordres partiels uniquement
  - Combinaisons possibles
    - Ordre FIFO-total
    - Ordre causal-total
- Cas particulier
  - Diffusion fiable avec un ordre total
  - Appelé diffusion atomique
    - Tous les processus reçoivent tous les messages diffusés et tous dans le même ordre
    - On ne « mélange » pas les diffusions, une diffusion doit être « terminée » sur tous les processus avant de pouvoir en exécuter une autre

Diffusion fiable : combinaisons

- Au final, 6 combinaisons possibles

	Pas d'ordre total	**Ordre total**
Ni FIFO ni causal	Diffusion fiable	Diffusion atomique
FIFO	Diffusion fiable FIFO	Diffusion atomique FIFO
Causal	Diffusion fiable causale	Diffusion atomique causale

  - Sans oublier les versions uniformes ou les versions temporisées (respect d'un certain délai temporel pour la réalisation d'une diffusion) ...
- Autres protocoles de diffusion
  - Pour fonctionner dans d'autres critères de fautes (perte de messages, fautes byzantines ...)

Diffusion : conclusion

- Conclusion sur la diffusion
  - La diffusion fiable/atomique est à la base de la plupart des algorithmes de coordination et de prise de décision en commun (accord)
    - Et la plupart de ces algorithmes de prise de décision en commun peuvent se ramener à des adaptations de la diffusion
- Indispensable pour gérer un algorithme de groupe
  - Groupe = ensemble de processus
  - Peut entrer et quitter le groupe
    - Nécessité que tout le monde ait la même connaissance ou perception du groupe (savoir qui est présent sans se tromper)
    - Peut se faire en échangeant régulièrement sa liste de présents via une diffusion fiable
  - Variante de la diffusion : en multicast, dans un sous-groupe du groupe global

Transaction distribuée

Transaction

- Transaction
  - Concept venant des bases de données
- Principe
  - Exécution d'une action ou d'une séquence d'actions
  - Soit par un seul élément / processus
  - Soit par plusieurs
    - Cas des transactions distribuées
- Exemple de séquence d'action
  - Transfert d'argent d'un compte vers un autre
    - Requière un débit *puis* un crédit
    - Il faut faire les 2 actions ou aucune sinon on se retrouve dans un état incohérent
    - Begin Transaction
      Debiter (#1244, 1000€)
      Crediter (#8812, 1000€)
      End Transaction

Transaction

- Propriétés d'une transaction
  - Propriétés ACID [Härder & Reuter, 83]
  - Atomicité
    - Tout ou rien : l'action de la transaction est entièrement réalisée ou pas du tout, pas d'intermédiaire à moitié fait
  - Cohérence
    - L'exécution d'une transaction fait passer le système d'un état cohérent à un autre
  - Isolation
    - Les transactions n'interfèrent pas entre elles
  - Durabilité
    - Les effets de la transaction sont enregistrés de manière permanente

Transaction distribuée

- Transaction distribuée
  - Des processus distribués effectuent tous l'action qui leur est demandée ou aucun ne la fait
    - L'action peut être la même pour tous ou différer selon les processus
- Exemple
  - Retrait sur un compte bancaire
  - Avec redondance de la base gérant les comptes
    - Le serveur gérant les comptes est dupliqué (une ou plusieurs fois) pour augmenter la fiabilité en cas de panne
  - Quand un processus client demande à faire un retrait sur son compte, deux résultats possibles
    - Le retrait est fait chez tous les serveurs
    - Le retrait n'est fait chez aucun serveur
    - Il faut en effet assurer la cohérence des données stockées entre les serveurs
  - Dans les deux cas, le client est averti de ce qu'il s'est passé
    - Sauf si limite de l'algorithme utilisé dans un certain contexte de faute

Validation atomique

- Validation atomique
  - Au coeur de la problématique de transaction distribuée
  - Protocole de coordination entre processus
    - Se mettent d'accord sur la faisabilité de la transaction
    - Détermination d'une décision globale entre
      - Valider : l'action de la transaction sera exécutée par tous les processus
      - Annuler : aucune action ne sera exécutée par aucun processus
  - Existe de nombreux protocoles dont
    - Validation à 2 phases : 2 phases commit (2PC)
      - Le plus utilisé en pratique mais n'assure pas la terminaison
    - Validation à 3 phases : 3 phases commit (3PC)
      - Extension du 2PC qui assure la terminaison

Validation atomique

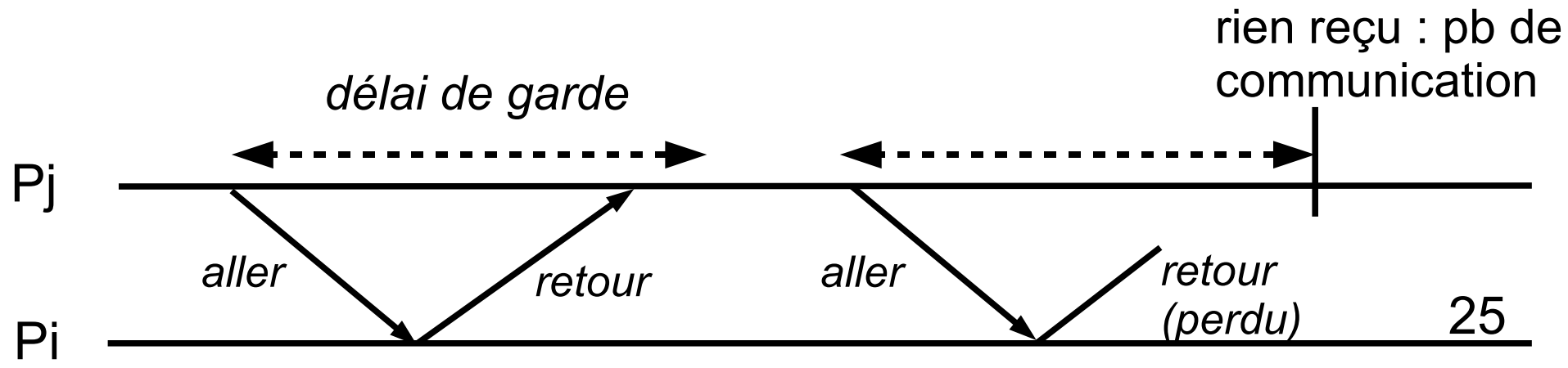
- Propriété d'un protocole de validation atomique
  - Validité
    - La décision est soit valider, soit annuler, et si un processus décide de valider, alors tous les processus avaient voté « oui »
  - Intégrité
    - Un processus décide au plus une fois
  - Accord (uniforme)
    - Tous les processus corrects (ou non) prendront la même décision
  - Terminaison
    - Tout processus correct décide en un temps fini
  - Non-trivialité
    - Si tous les processus votent « oui » et qu'il n'y a pas de panne, alors la décision est de valider
      - Propriété servant à rejetter la solution triviale de décision systématique d'annuler

Validation atomique

- Validation atomique est simple a priori
  - Demander l'avis de tous les processus sur la faisabilité de l'action
  - Si tous les processus répondent « je peux le faire »
    - La décision est de valider, chacun exécute l'action
  - Sinon, si au moins un processus avait répondu qu'il ne pouvait pas effectuer l'action
    - La décision est d'annuler, personne n'exécute l'action
- Mais devient bien moins simple dans un contexte de fautes ...
- Dans ce qui suit, description des algorithmes 2PC et 3PC avec le contexte de fautes suivant
  - Pannes franches de processus mais avec reprise possible
  - Canaux de communication fiables (pas de perte de message)
  - Système synchrone

Principe d'aller / retour

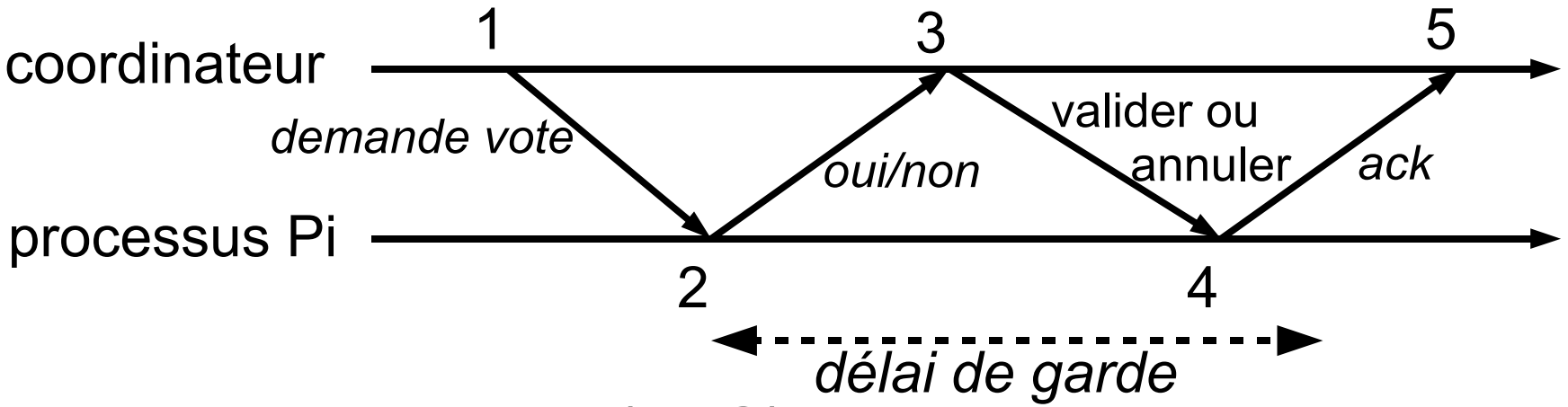
- La détection d'un problème avec un élément passe par un aller – retour de message
  - On envoie un message à un élément
  - On attend sa réponse dans un intervalle de temps (délai de garde)
    - Estimé à 2 fois le temps de transit d'un message (+ delta éventuel)
  - Si pas reçu de réponse avant l'expiration du délai de garde
    - Message perdu ou l'élément avec qui on communique est planté
    - Pb dans un système asynchrone : estimation du délai de garde



Validation atomique : 2PC

- Deux types d'éléments
  - Un processus coordinateur qui lance et coordonne la réalisation de la validation atomique
  - L'ensemble des processus participants à la transaction
    - On demandera à chacun d'entre eux s'il peut ou pas réaliser l'action
- Validation à 2 phases (2PC)
  - Première phase : demande de vote et réception des résultats
  - Deuxième phase : diffusion de la décision à tous et exécution de l'action chez les processus le cas échéant

Validation atomique 2PC



- Validation à 2 phases (2PC), cas nomimal, sans erreur
  1. Le coordinateur envoie une demande de vote à tous les processus
  2. Chaque processus étudie la demande selon son contexte local et répond *oui* ou *non* (selon qu'il peut ou pas exécuter la requête demandée)
  3. Quand le coordinateur a reçu tous les votes, il envoie la décision finale : valider si tout le monde a répondu *oui* ou annuler si au moins un processus a répondu *non*
  4. Chaque processus exécute la requête s'il reçoit valider ou ne fait rien s'il reçoit annuler, et envoie un acquittement au coordinateur pour lui préciser qu'il a reçu la décision globale

2PC : gestion des problèmes

- Gestion des problèmes, cadre général
  - Si un délai de garde expire, c'est que l'élément dont on attendait un message ne l'a pas envoyé
    - On suppose alors qu'il est en panne
    - Dans un contexte où la panne peut être temporaire et le processus peut être relancé
      - Peut renvoyer un message au processus
      - Mais pas certain que le procesus sera remis sur pied et ne sait pas, si ça arrive, combien de temps il faudra attendre
      - Un processus peut enregistrer son état ou ses décisions, il saura alors ce qu'il avait décidé s'il repart
  - Un processus qui ne reçoit pas de réponse du coordinateur peut interroger les autres processus
    - Selon les cas, certains processus connaissent des choses qu'on devrait également savoir, on peut leur demander
      - Mais pas sûr que quelqu'un ait l'information cherchée

2PC : gestion des problèmes

- Gestion des problèmes, coté coordinateur
  - Phase de vote, expiration du délai de garde en 3 : un processus Pi n'a pas répondu
    - Considère qu'il est planté ... donc pas capable d'effectuer l'action
      - La décision globale est alors d'annuler la transaction
      - Variante si possibilité de reprise d'un processsus : relancer la requête à Pi
  - Phase de décision, expiration du délai de garde en 5 : pas reçu tous les acquittements des processus
    - Le coordinateur ne sait pas si tout le monde a reçu la décision
    - Si c'était annuler : pas si grave, il ne fallait rien faire
      - Pb tout de même en cas de reprise d'un processus planté qui avait choisi oui : ne connait pas la décision globale
      - Peut la demander à d'autres processus ou au coordinateur
    - Si c'était valider : problème important car n'est pas assuré que toutes les actions ont bien été réalisées
    - Dans les 2 cas, on peut renvoyer la décision globale aux processus n'ayant pas répondu en espérant recevoir les acquittements

2PC : gestion des problèmes

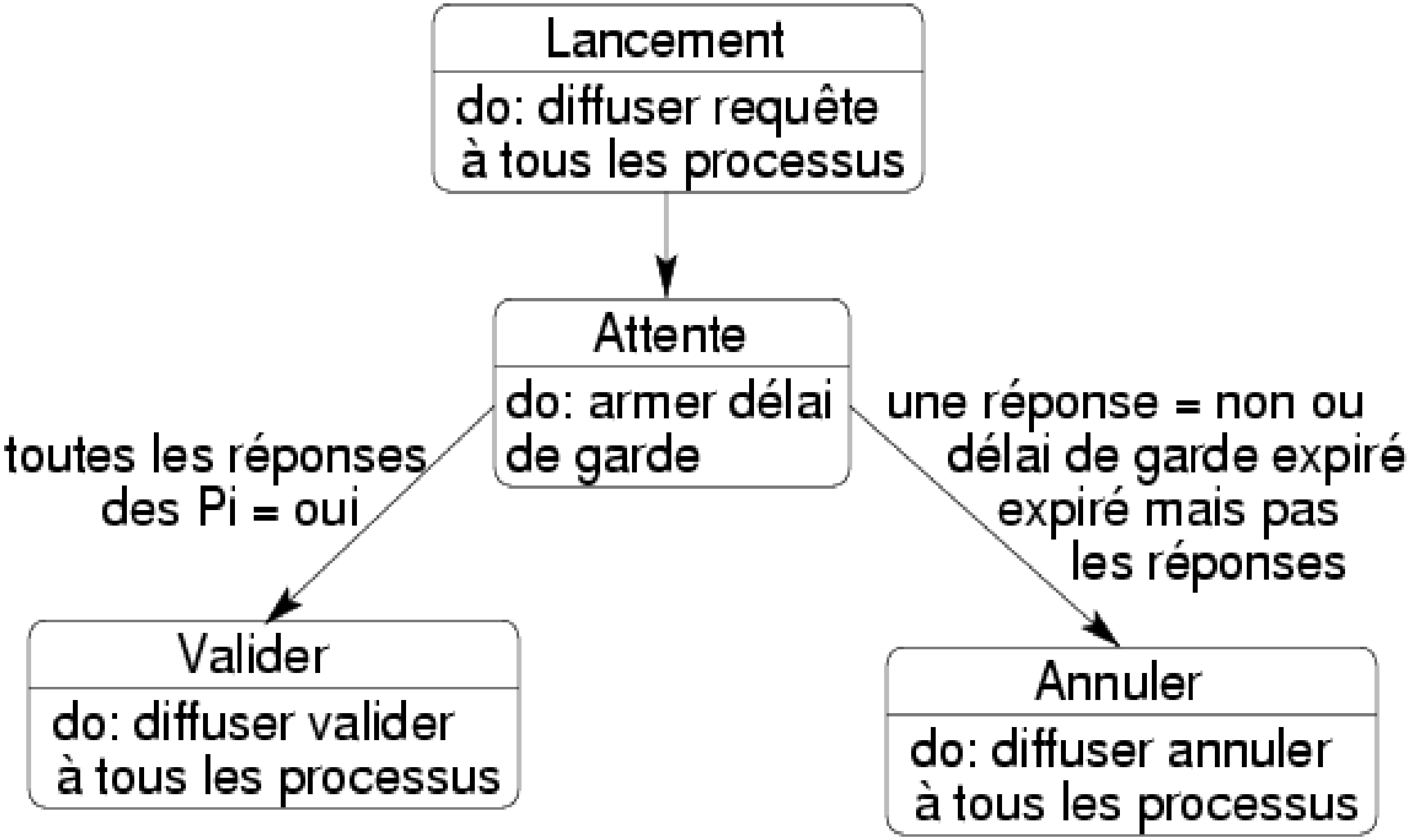
- Gestion des problèmes, coté processus
  - Expiration du délai de garde en 4 sur Pi : pas reçu la décision globale de la part du coordinateur
    - Si le Pi avait voté non
      - Pas un problème, car si au moins un processus vote non, la décision globale est d'annuler et comme le processus sait qu'il a voté non, il connait la décision globale qui est forcément d'annuler
      - N'a pas besoin de recevoir l'information de la part du coordinateur
    - Si Pi avait voté oui
      - Ne peut pas savoir quoi faire : ne sait pas si tout le monde avait voté oui ou si au moins un processus avait voté non
      - Le problème est que le coordinateur est planté
        - A-t'il planté avant de prendre la décision globale ?
          - Personne alors ne sait ce qui devait être décidé
        - A-t'il planté pendant qu'il envoyait la décision globale ?
          - Certains processus ont alors reçu la décision globale
          - (mais peuvent être plantés depuis)

2PC : gestion des problèmes

- Gestion des problèmes, coté processus
  - Non réception de la décision globale en 4 sur Pi (fin)
    - Pour connaître la décision globale, Pi peut interroger les autres processus
      - Mais pas certain de recevoir une réponse, peut-être que personne ne la connaît
      - Et de manière plus générale, les processus ne se connaissent pas forcément entre eux
    - Si le coordinateur reprend son exécution après sa panne, il peut relancer le protocole là où il s'était arrêté
      - Arrivera alors à informer les processus et donc Pi
    - Reste toujours des cas où un processus ne connaîtra pas la décision finale en ayant voté oui
      - Mais Pi ne peut pas choisir seul ce qu'il faut faire
      - Car dans le cas où certains processus ont reçu la décision du coordinateur
        - Si Pi décide d'exécuter l'action alors que la décision était annuler ou s'il décide de ne pas exécuter l'action alors que la décision était valider, on se retrouver dans un état incohérent, tous les procesus ne feront pas la même chose !

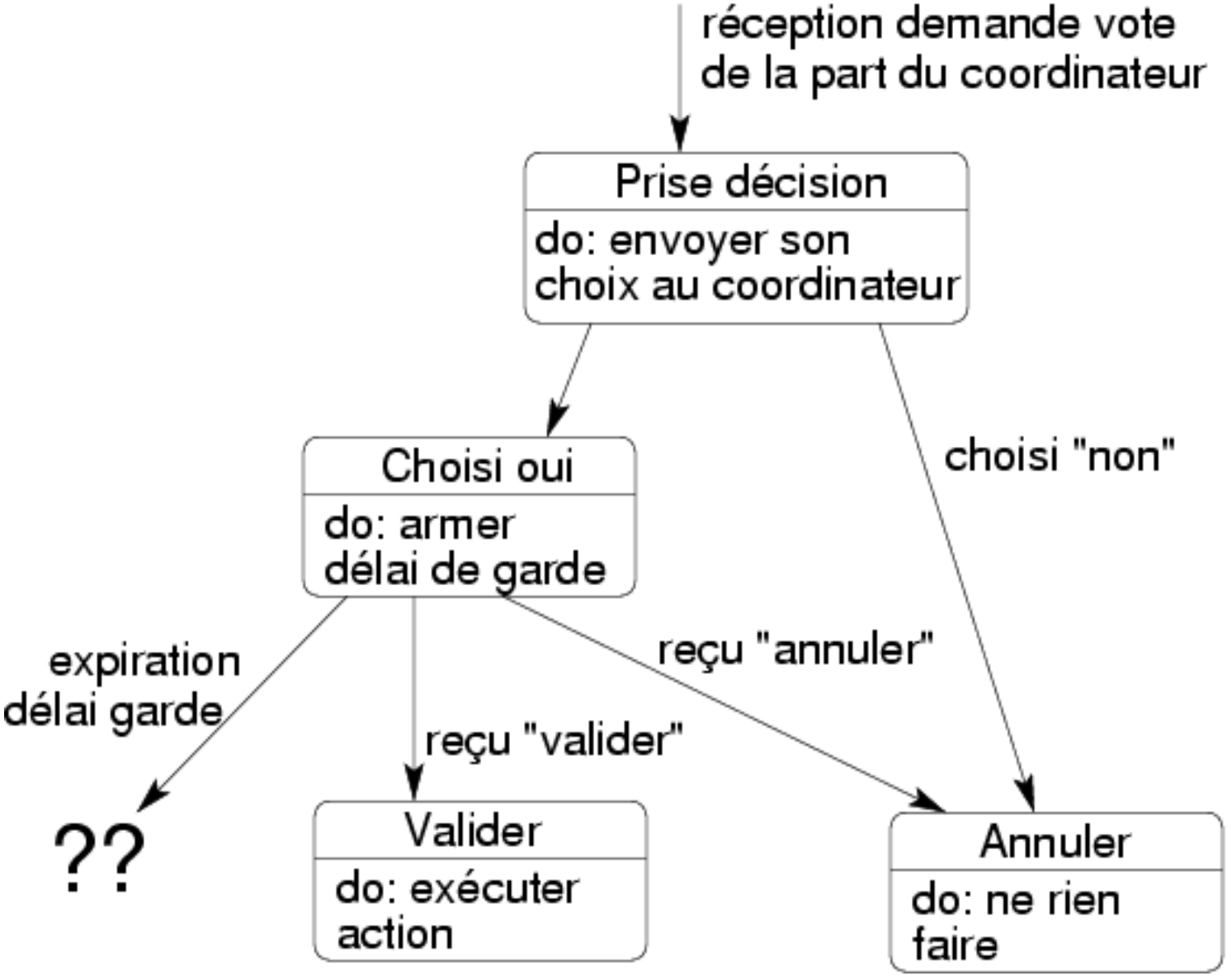
2PC : coordinateur

- Diagramme d'état du coordinateur



2PC : processus

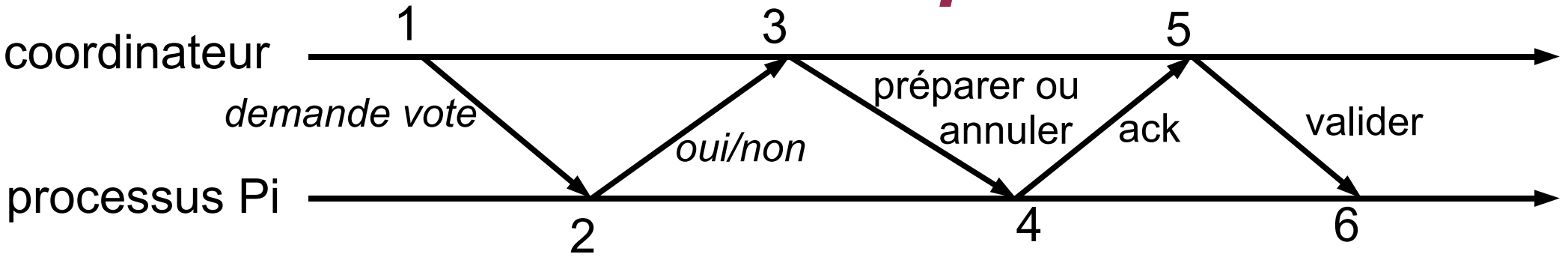
- Diagramme d'état d'un processus Pi



Validation atomique : vers la 3PC

- Conclusion sur la 2PC
  - La terminaison n'est pas assurée, le protocole est potentiellement bloquant
    - Un processus peut se retrouver dans un état où il ne sait pas quoi faire
  - Le problème général est qu'entre les moments 2 et 4, les processus sont dans une zone d'incertitude
    - Ne savent pas ce qui a été choisi globalement
  - La validation atomique à 3 phases rajoute une étape où on informe les processus de la décision globale *avant* de leur demander de la réaliser
- Validation atomique à trois phases (3PC)
  - Première phase : demande de vote et attente des votes (idem que pour 2PC)
  - Deuxième phase : envoie à tous les processus de la décision globale et attente des acquittements de la réception
  - Troisième phase : diffusion de la demande d'exécution de la requête (sauf si décision annuler)

Validation atomique : 3PC



- Validation à 3 phases (3PC), cas nomimal, sans erreur
  1. Le coordinateur envoie une demande de vote à tous les processus
  2. Chaque processus étudie la demande selon son contexte local et répond oui ou non
  3. Quand le coordinateur a reçu tous les votes, il envoie
     - Soit annuler si un processus avait voté non
       - La validation est alors terminée, pas d'autres échanges de messages
     - Soit une demande aux processus de se préparer à exécuter la requête
  4. Chaque processus envoie un acquittement au coordinateur de la réception de cette demande
  5. Une fois tous les acquittements reçus, le coordinateur diffuse la validation finale
  6. Chaque processus exécute alors la requête

3PC : gestion des problèmes

- Gestion des problèmes, coté coordinateur
  - Phase de vote, expiration du délai de garde en 3 : un processus Pi n'a pas répondu
    - Idem que pour 2PC : la décision globale est d'annuler
  - Phase de décision, expiration du délai de garde en 5 : un processus Pi n'a pas envoyé son acquittement de pré-validation
    - Pi a planté et ne pourra donc pas valider l'action
      - Si possibilité de reprise, peut essayer de lui renvoyer le message de préparation
      - Ou il aura de toute façon enregistré le fait qu'il avait voté oui
    - Coordinateur sait que tous les autres processus avait forcément voté oui (sinon le coordinateur ne serait pas dans cet état de pré-validation en 5)
      - Peut donc tout de même lancer l'ordre de validation à tous les processus
  - Note
    - Peut rajouter des acquittements identiques pour l'annulation et validation globale envoyés des processus au coordinateur comme en 2PC
      - Gérera les problèmes comme en 2PC

3PC : gestion des problèmes

- Gestion des problèmes, coté processus
  - Expiration du délai de garde en 4 sur Pi : pas reçu la décision globale de la part du coordinateur
    - Comme en 2PC, Pi ne connaît pas la décision globale et ne peut la déterminer
    - Lance alors un protocole de terminaison, dont la version la plus courante est l'élection d'un nouveau coordinateur ou la relance du coordinateur en panne
      - Ce dernier reprendra le protocole de validation
  - Expiration du délai de garde en 6 sur Pi : pas reçu la demande de validation de la part du coordinateur
    - A la différence du 2PC, Pi connaît la décision globale qui est forcément de valider, il peut donc quoiqu'il arrive exécuter l'action
    - Sauf qu'il ne sait pas si tout le monde l'exécutera (vu que seulement certains processus ont pu recevoir la validation finale de la part du coordinateur avant qu'il ne plante)
    - On lance alors là aussi le protocole de terminaison (nouveau coordinateur)

3PC : gestion des problèmes

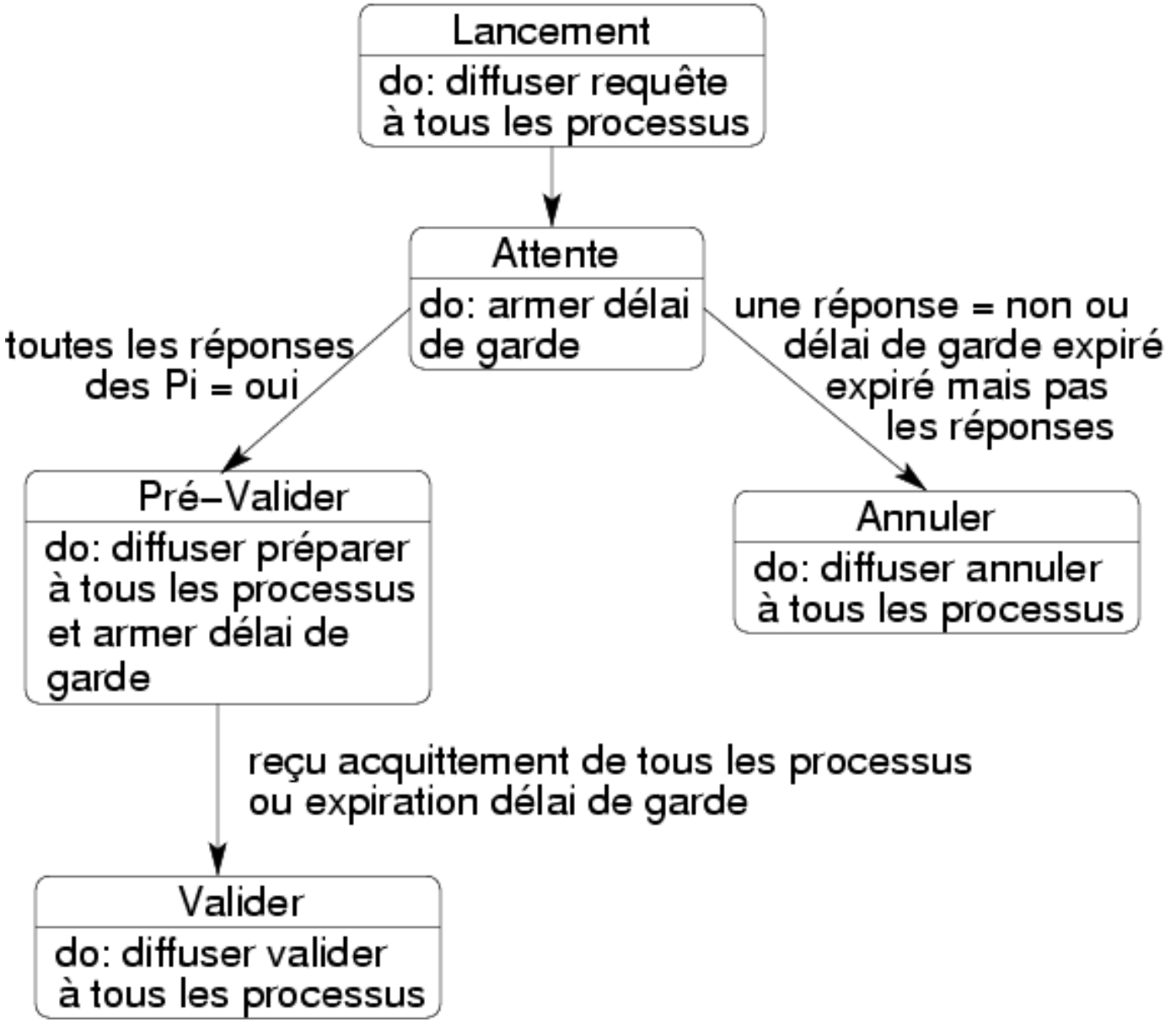
- Gestion des problèmes : nouveau coordinateur
  - Principe général est de retrouver l'état dans lequel le coordinateur était avant de planter
  - Si lancement d'un nouveau coordinateur : doit interroger des processus pour le déterminer
    - Si au moins un processus dans l'état annuler : passe dans l'état annuler et envoie une décision d'annulation globale à tous les processus
    - Si certains processus dans état valider : la décision globale est forcément de valider et le message valider a déjà été envoyé à certains mais pas reçu par tous
      - Envoyer message de validation à tous
    - Si certains processus dans l'état de pré-validation et d'autres dans l'état d'attente (juste après le vote) : la décision globale est forcément de valider mais tout le monde n'a pas reçu le message de préparation
      - Reprendre le protocole au niveau de l'envoi du message préparer
    - Tous les processus dans l'état d'attente : avaient tous voté oui
      - Reprendre le protocole au niveau de l'envoi du message préparer

3PC : gestion des problèmes

- Gestion des problèmes : nouveau coordinateur (fin)
  - Si relance du coordinateur : simple, il avait enregistré son dernier état avant de planter
    - Reprend alors où il s'était arrêté
    - Si dans l'état d'attente des votes
      - Annule la transaction : envoyer annuler
    - Si dans l'état de pré-validation ou de validation
      - Sait que la décision de valider avait été prise
      - Renvoie les messages de préparation et/ou de validation selon l'état

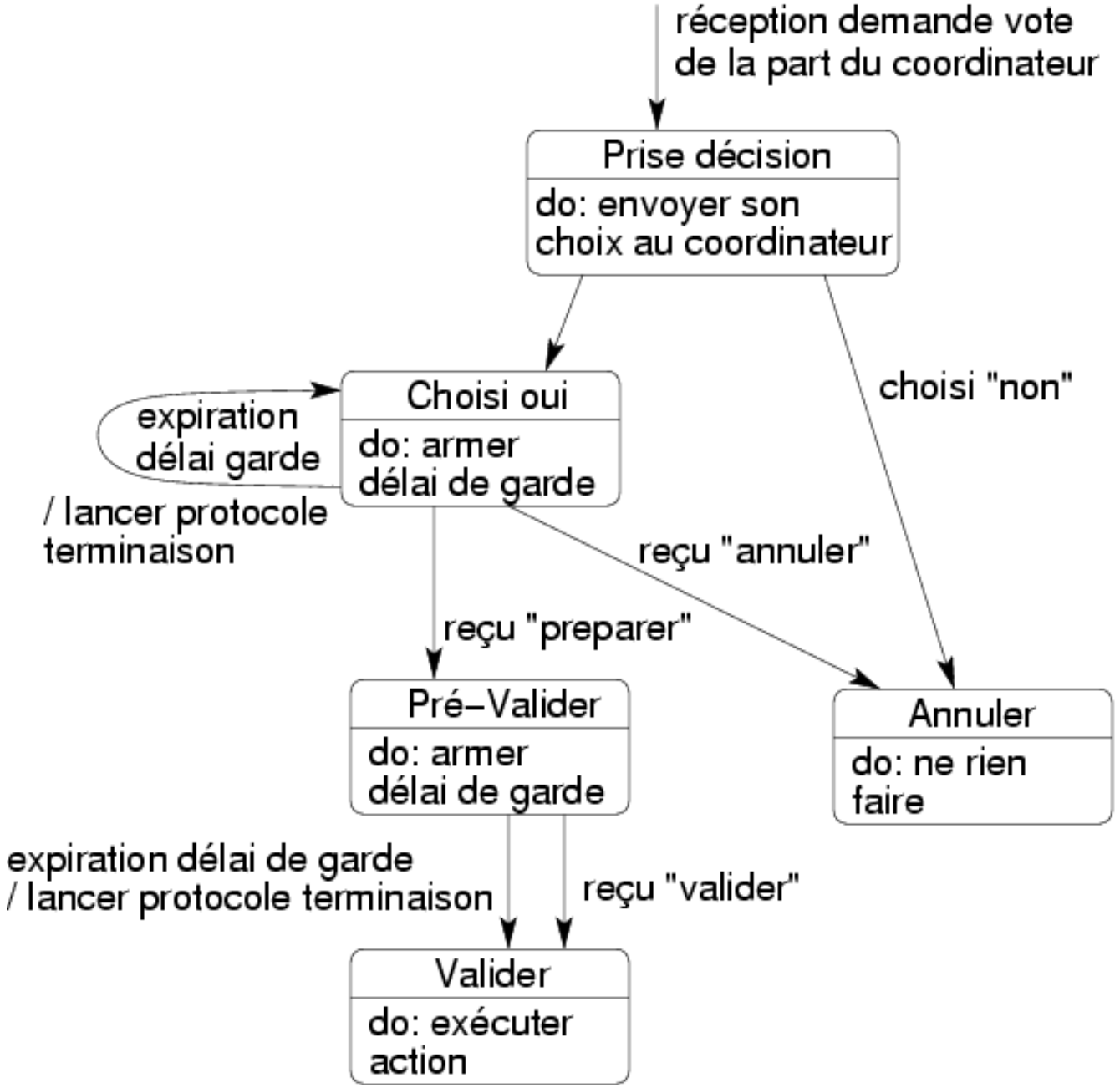
3PC : coordinateur

- Diagramme d'état du coordinateur



3PC : processus

- Diagramme d'état d'un processus Pi



Validation atomique : 3PC

- Conclusion 3PC
  - Si un processus reçoit un « préparer » de la part du coordinateur, il sait alors que quoiqu'il arrive tous les processus ont décidé de valider la requête
  - Principe est qu'un coordinateur ou un processus n'a pas de doute sur le fait de transiter vers annuler ou valider
    - Pas de possibilité d'aller vers les 2 états finaux directement
      - Comme on peut l'avoir en 2PC : ex pour processus, état attente de la décision globale a une transition vers valider et une vers annuler
    - Passe notamment par un état de pré-validation avant de valider définitivement

Consensus

Consensus

- Problème « classique » en algorithmique distribuée
  - Intérêt : simple voire trivial dans un contexte sans faute mais difficile voire impossible à réaliser dans un contexte avec fautes
- Principe général
  - Des processus doivent se mettre d'accord sur une valeur commune
  - Deux étapes
    - Chaque processus fait une mesure ou un calcul : valeur locale qui est proposée à tous les autres processus
    - Les processus, à partir de toutes les valeurs proposées, doivent se décider sur une valeur unique commune
      - Soit un processus initie la phase d'accord
      - Soit la phase d'accord est lancée à des instants prédéterminés

Consensus

- Conditions à valider
  - Accord
    - La valeur décidée est la même pour tous les processus corrects
  - Intégrité
    - Un processus décide au plus une fois : pas de changement de choix de valeur
  - Validité
    - La valeur choisie par un processus est une des valeurs proposées par l'ensemble des processus
  - Terminaison
    - La phase de décision se déroule en un temps fini : tout processus correct décide en un temps fini

Consensus : sans faute

- Contexte sans faute
  - Solution simple
    - Après la mesure de la valeur, un processus envoie sa valeur à tous les autres via une diffusion fiable
    - A la réception de toutes les valeurs (après un temps fini par principe), un processus applique une fonction donnée sur l'ensemble des valeurs
    - La fonction est la même pour tous les processus
      - Chaque processus est donc certain d'avoir récupéré la valeur commune qui sera la même pour tous
  - Autre solution simple
    - Passer par un processus coordinateur qui centralise la réception des valeurs proposées et fait la décision, puis la diffuse
  - Ces solutions fonctionnent pour les systèmes distribués synchrones ou asynchrones
    - La différence est le temps de terminaison qui est borné ou non

Consensus : synchrone, panne franche

- Avec pannes franches et système synchrone
  - Et communications fiables : pas de perte ni de duplication de messages
  - Bases de l'algorithme
    - On connaît la borne max d'attente de réception de messages
    - Peut donc déterminer si un processus n'a pas répondu
      - C'est-à-dire s'il est planté
    - Utilise de la diffusion fiable
      - Tous les processus recevront exactement les mêmes valeurs et pourront appliquer la même fonction de choix qui aboutira forcément au même choix

Consensus : asynchrone, panne franche

- Avec pannes franches et système asynchrone
  - En 1983, Fischer, Lynch & Paterson (FLP) ont montré que
    - Dans un système asynchrone, même avec un seul processus fautif, il est impossible d'assurer que l'on atteindra le consensus
  - Malgré ce résultat théorique, il est possible en pratique d'atteindre des résultats satisfaisants
    - FLP précise en effet qu'on atteindra pas toujours le consensus, mais pas qu'on ne l'atteindra jamais
  - Deux approches en pratique
    - Se baser sur des systèmes « partiellement synchrones »
      - Limiter l'asynchronisme pour avoir des hypothèses mieux adaptées
    - Ou faire du mieux qu'on peut (*best-effort*) pour essayer d'atteindre le consensus
      - Définir des algorithmes non parfaits
      - Utiliser des détecteurs de pannes non parfaits

Consensus : asynchrone, panne franche

- Algorithme de Paxos [Lamport, 89]
  - Algorithme non parfait de consensus en asynchrone, contexte de pannes franches
  - Principe très général
    - Un processus coordinateur tente d'obtenir les valeurs des processus et choisi la valeur majoritaire
    - Si le processus coordinateur se plante, d'autres processus peuvent reprendre son rôle
      - Possibilité de plusieurs coordinateurs
  - Terminaison de l'algorithme
    - Quand un coordinateur a existé suffisamment longtemps pour que le consensus soit obtenu
      - Importance des délais de garde pour les communications et déterminer quand l'algorithme est terminé
    - Ne se termine pas toujours
      - FLP reste valable quoiqu'il arrive ...

Consensus : asynchrone, panne franche

- Utilisation de détecteurs de fautes
  - Problème en asynchrone : impossibilité de différencier un processus ou un message lent d'un processus planté
  - Un détecteur de fautes permet de savoir ou de soupçonner si un processus est planté ou pas
    - Les algorithmes développés se basent alors sur ces connaissances
- Quels types de détecteurs ?
  - A priori il faudrait de préférence un détecteur parfait
    - Mais il a été montré qu'un détecteur parfait (P) ou imparfait (de classe S, ◊P ou ◊S) permet d'assurer d'atteindre le consensus en asynchrone
  - Mais ces détecteurs ne sont pas réalisables en asynchrone !
    - FLP toujours : en théorie, le consensus n'est pas atteignable systématiquement en asynchrone, or un détecteur de fautes le permet
      - Un tel détecteur de fautes n'existe donc pas en pratique
  - En pratique, on peut tout de même réaliser des détecteurs suffisamment pertinents et réalistes

Consensus : async. fautes byzantines

- Contexte de fautes byzantines
  - Bien plus complexe à gérer que les pannes franches
- En asynchrone
  - Théorie
    - Consensus pas toujours atteignable
    - Ne l'était déjà pas avec des fautes franches, ne peut donc pas a fortiori l'être avec des fautes byzantines
  - Pratique
    - Essayer là aussi d'obtenir des résultats pertinents et suffisamment fiables
      - Peut adapter notamment l'algorithme de Paxos
      - Des algorithmes spécifiques existent également

Consensus : sync. fautes byzantines

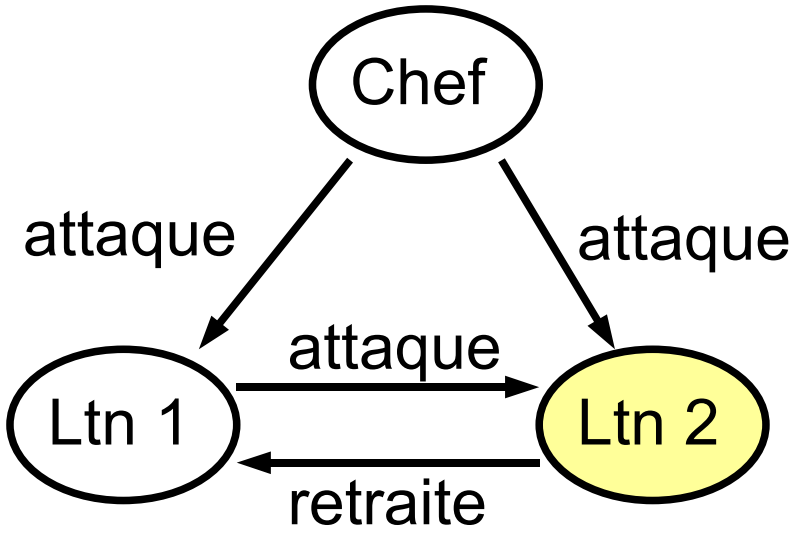
- Fautes byzantines en synchrone
  - Contrairement aux pannes franches, il n'est pas possible d'assurer que le consensus sera toujours atteint dans un contexte de fautes byzantines
  - Lamport, Shostak et Pease ont montré en 1982, via le problème des généraux byzantins, que
    - Si *f* est le nombre de processus fautifs, il faut au minimum *3f* + 1 processus (au total) pour que le consensus soit assuré
    - En dessous, il n'est pas assuré
  - Les algorithmes à mettre en oeuvre sont relativement lourds en nombre de messages

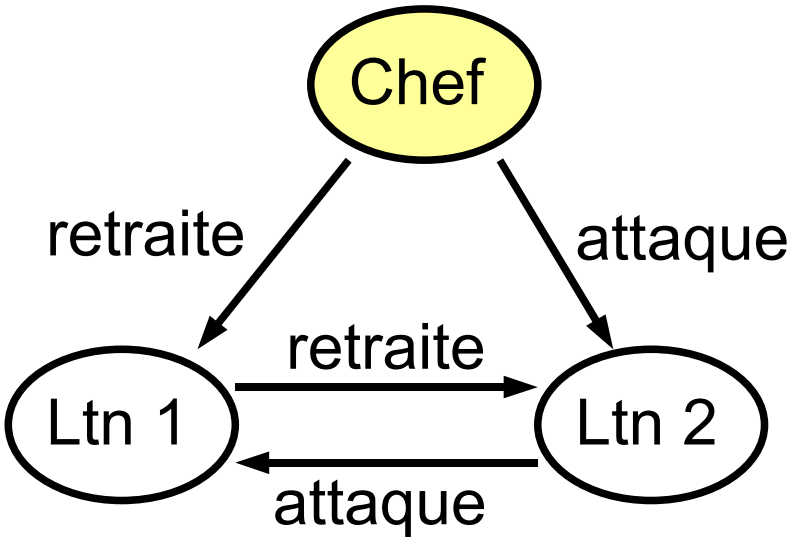
Fautes byzantines

- Problème des généraux byzantins
  - Un ensemble de généraux doivent se coordonner pour mener une attaque
    - Doivent tous faire la même chose (attaquer ou battre en retraite)
  - Le général en chef prend l'ordre et l'envoie aux autres généraux (ses lieutenants)
    - Ces derniers se renvoient l'ordre entre eux pour s'assurer qu'il est bien reçu et est bien le bon
  - Problème
    - Un certain nombre de généraux peuvent être des traîtres
    - Ils vont envoyer des ordres différents aux autres généraux
    - Les généraux loyaux doivent détecter les traîtres et ignorer leurs ordres
- Problème caractéristique pour les fautes byzantines
  - Montre l'obligation d'avoir au moins *3f + 1* processus si on a *f* processus fautifs si on veut gérer correctement les fautes byzantines

Fautes byzantines

- Contexte avec 3 généraux
  - Généralisable à des multiples de 3 d'où le « 3f + 1 »





- Deux cas
  - Gauche : le lieutenant 2 est le traitre, il renvoie le mauvais ordre
  - Droite : le chef est le traître, il envoie deux ordres différents
  - Dans les 2 cas, le lieutenant 1 reçoit deux ordres contradictoires en étant incapable de savoir lequel est le bon
  - Avec un troisième lieutenant (donc 4 généraux), il recevrait un message de plus qui permettrait de savoir quel est le bon ordre

Consensus : fautes byzantines

- Algorithme des généraux byzantins
  - Correspond à une réalisation de consensus dans un contexte de fautes byzantines
- Spécification du problème
  - L'armée byzantine est formée de plusieurs généraux (processus)
    - Dont un chef
    - Les autres sont les lieutenants
  - Certains généraux sont des traîtres, les autres sont loyaux
    - Certains processus effectuent des fautes byzantines : envoi de valeur erronée voire aucun envoi de valeur
  - Les généraux loyaux doivent aboutir à prendre la bonne décision (attaquer ou battre en retraite) : consensus
    - Bonne décision : suivre l'ordre du chef (sauf dans le cas où c'est un traitre)
  - Les traîtres ne doivent pas aboutir au fait que la prise de décision des généraux loyaux soit mauvaise ou pas identique chez tous

Consensus : fautes byzantines

- Caractéristiques du système
  - Chaque général peut envoyer un message à n'importe quel autre général et savoir de qui vient un message
    - Les processus sont identifiés et se connaissent
  - Les messages ne sont pas perdus ou modifiés pendant leur transfert
    - Communication fiable
  - Système synchrone
    - Ou asynchrone si possibilité de détecter qu'on aurait du recevoir un message
    - Car il faut gérer le cas où un traitre décide de ne pas envoyer un message alors qu'il était censé le faire
  - Pas de « disparition » de général
    - Pas de pannes (franches) de processus

Consensus : fautes byzantines

- Contraintes à assurer par l'algorithme
  - Respect des conditions de « consistance interactive » (Interactive Consistency)
    - IC1 : tous les lieutenants loyaux obéissent au même ordre
    - IC2 : si le chef est loyal, les lieutenants loyaux doivent obéir à son ordre
- L'algorithme suivant assure le consensus et le respect des conditions IC1 et IC2
  - Si le nombre de traitres est limité
  - n = nombre de généraux, m = nombre de traîtres
    - Il faut : n > 3m

Consensus : fautes byzantines

- Algorithme des généraux
  - Chaque lieutenant *Li* possède une valeur locale v_i qui est l'ordre qu'il a décidé de suivre
    - Déterminé en fonction de ce qu'il a reçu des autres généraux
  - Fonctionne pour un certain nombre de traitres acceptables
  - P(m) : algorithme pour *m* traitres au plus
    - S'appliquera de manière récursive : P(m – 1), P(m – 2) jusqu'à atteindre P(0)
    - Pour un P(x), un processus joue le rôle de chef et diffuse sa valeur à tous les autres lieutenants
    - Plus il y a de traîtres possibles, plus on s'échange de messages
      - Nombre de messages : $O(n^m)$
      - Avec toujours $n > 3m$

Consensus : fautes byzantines

- P(0) : pas de traitre
  1. Le chef diffuse sa valeur à tous les lieutenants *Li*
  2. Pour tout *i*, à la réception de la valeur du chef, chaque lieutenant positionne v_i à la valeur reçue
     - Si ne reçoit rien, v_i = DEF (absence de valeur)
- P(m) : *m* traitres
  1. Le chef diffuse sa valeur à tous les lieutenants *Li*
  2. Pour chaque *Li*, à la réception de la valeur du chef, chaque lieutenant positionne v_i à la valeur reçue
     - Si ne reçoit rien, v_i = DEF (absence de valeur)
  3. Chaque lieutenant exécute P(m – 1) en jouant le rôle du chef : diffusion de sa valeur aux n – 2 autres lieutenants
  4. Une fois reçu la valeur v_j de chacun des lieutenants (v_j = DEF si rien reçu de *Lj*), détermine la valeur locale v_i
     - v_i = valeur majoritaire parmi les v_j ou DEF si pas de majorité

Consensus : résumé

- Selon le contexte de faute et le modèle temporel (avec communications fiables dans tous les cas)
  - Asynchrone
    - Impossibilité de définir des algorithmes assurant d'aboutir à un consensus dans tous les cas
      - Problème principal : impossibilité de différencier un processus planté (ou lent) d'un message lent
    - Doit se contenter de faire au mieux (généralement suffisant)
  - Synchrone
    - Panne franche de processus
      - Consensus atteignable systématiquement
    - Panne byzantine
      - Consensus assuré si on ne dépasse pas un seuil de processus au comportement byzantin en proportion du nombre global de processus

Bibliographie

Algorithmique distribuée

- *Algorithmique et techniques de base des systèmes répartis*, Sacha Krakowiak, Cours de M2 recherche "Systèmes et Logiciel", Université Joseph Fournier (Grenoble) http://proton.inrialpes.fr/~krakowia/Enseignement/M2R-SL/SR/
- *Systèmes d'Exploitation et Applications Répartis*, Françoise Baude & Michel Riveill, Maitrise MIAGE, Université de Nice, http://corbieres.unice.fr/~baude/Systemes-Distribues/
- *Conception des Systèmes répartis,* Gérard Padiou, enseignements à l'ENSEEIHT (Toulouse) http://padiou.perso.enseeiht.fr/
- *Synchronisation et état global dans les systèmes répartis*, Michel Raynal, Eyrolles, 1992
- *Distributed Systems: Concept and Design*, George Couriolis, Jean Dollimore & Tim Kindberg, 3ème édition, Addison Wesley