Algorithmique distribuée

Exclusion mutuelle

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

Exclusion mutuelle distribuée

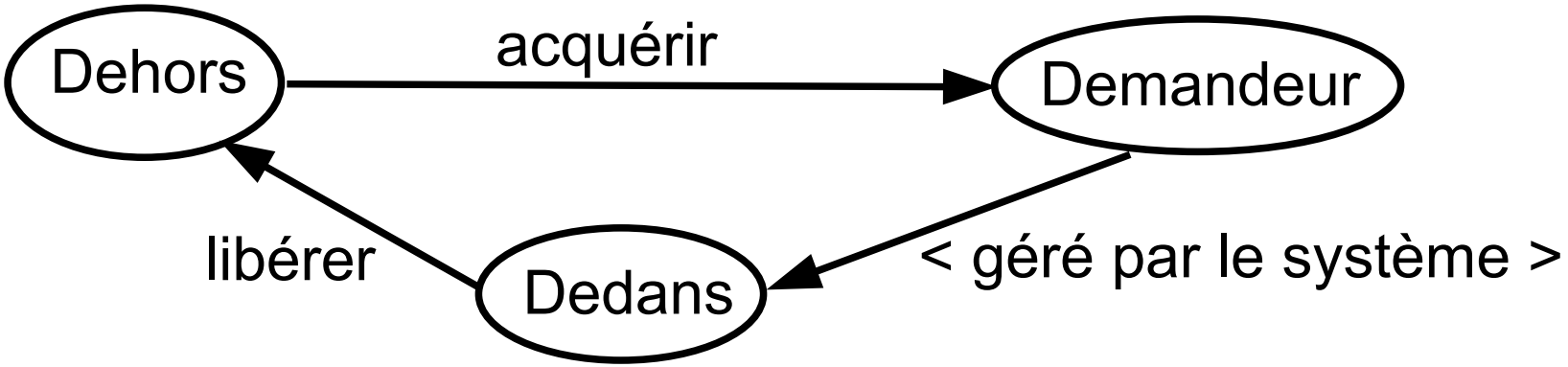
- Exclusion mutuelle
  - Contexte de plusieurs processus s'exécutant en parallèle
  - Accès à une ressource partagée par un seul processus à la fois
- Exclusion mutuelle en distribué
  - Accès à une ressource partagée distante par un seul processus à la fois
  - Processus distribués
    - Requêtes et gestion d'accès via des messages échangés entre les processus
    - Nécessité de mettre en oeuvre des algorithmes gérant ces échanges de messages pour assurer l'exclusion mutuelle
- *Algorithmes d'exclusion mutuelle décrits dans ce cours : plus de détails dans « Synchronisation et état global dans les systèmes répartis », Michel Raynal, Eyrolles, 1992*

Rappel exclusion mutuelle

- Exclusion mutuelle
  - Une ressource partagée ou une section critique n'est accédée que par un processus à la fois
  - Un processus est dans 3 états possibles, par rapport à l'accès à la ressource
    - Demandeur : demande à utiliser la ressource, à entrer dans la section
    - Dedans : dans la section critique, utilise la ressource partagée
    - Dehors : en dehors de la section et non demandeur d'y entrer
  - Changement d'état par un processus
    - De *dehors* à *demandeur* pour demander à accéder à la ressource
    - De *dedans* à *dehors* pour préciser qu'il libère la ressource
  - Le passage de l'état *demandeur* à l'état *dedans* est géré par le système et/ou l'algorithme de gestion d'accès à la ressource

Rappel exclusion mutuelle

- Diagramme d'états de l'accès en exclusion mutuelle



- L'accès en exclusion mutuelle doit respecter deux propriétés
  - Sûreté (safety) : au plus un processus est à la fois dans la section critique (dans l'état *dedans*)
  - Vivacité (liveness) : tout processus demandant à entrer dans la section critique (à passer dans l'état *dedans*) y entre en un temps fini

Exclusion mutuelle distribuée

- Plusieurs grandes familles de méthodes
  - Contrôle par un serveur qui centralise les demandes d'accès à la ressource partagée
  - Contrôle par jeton
    - Un jeton circule entre les processus et donne l'accès à la ressource
    - La gestion et l'affectation du jeton – et donc l'accès à la ressource – est faite par les processus entre eux
    - Deux approches : jeton circulant en permanence ou affecté à la demande des processus
  - Contrôle par permission
    - Les processus s'autorisent mutuellement à accéder à la ressource

Contrôle par serveur

- Principe général
  - Un serveur centralise et gère l'accès à la ressource
- Algorithme
  - Un processus voulant accéder à la ressource (quand il passe dans l'état *demandeur*) envoie une requête au serveur
  - Quand le serveur lui envoie l'autorisation, il accède à la ressource (passe dans l'état *dedans*)
    - Il informe le serveur quand il relâche la ressource (passe dans l'état *dehors*)
  - Le serveur reçoit les demandes d'accès et envoie les autorisations d'accès aux processus demandeurs
    - Avec par exemple une gestion FIFO : premier processus demandeur, premier autorisé à accéder à la ressource

Contrôle par serveur

- Méthode par serveur centralisateur, critiques
  - Avantages
    - Très simple à mettre en oeuvre
    - Simple pour gérer la concurrence d'accès à la ressource
  - Inconvénients
    - Nécessite un élément particulier pour gérer l'accès
    - Potentiel point faible, goulot d'étranglement
- Suppression du serveur centralisateur
  - Via par exemple une méthode à jeton : le processus qui a le jeton peut accéder à la ressource
  - La gestion et l'affectation du jeton est faite par les processus entre eux
    - Pas de besoin de serveur centralisateur

Méthode par jeton

- Principe général
  - Un jeton unique circule entre tous les processus
  - Le processus qui a le jeton est le seul qui peut accéder à la section critique
- Respect des propriétés
  - Sûreté : grâce au jeton unique
  - Vivacité : l'algorithme doit assurer que le jeton circule bien entre tous les processus voulant accéder à la ressource
- Plusieurs versions
  - Anneau sur lequel circule le jeton en permanence
  - Jeton affecté à la demande des processus

Méthode par jeton

- Algorithme de [Le Lann, 77]
  - Un jeton unique circule en permanence entre les processus via une topologie en anneau
  - Quand un processus reçoit le jeton
    - S'il est dans l'état *demandeur* : il passe dans l'état *dedans* et accède à la ressource
    - S'il est dans l'état *dehors*, il passe le jeton à son voisin
  - Quand le processus quitte l'état *dedans*, il passe le jeton à son voisin
- Respect des propriétés
  - Sûreté : via le jeton unique qui autorise l'accès à la ressource
  - Vivacité : si un processus lâche le jeton (la ressource) en un temps fini et que tous les processus appartiennent à l'anneau

Méthode par jeton

- Algorithme de [Le Lann, 77], critiques
  - Inconvénients
    - Nécessite des échanges de messages (pour faire circuler le jeton) même si aucun site ne veut accéder à la ressource
    - Temps d'accès à la ressource peut être potentiellement relativement long
      - Si le processus *i+1* a le jeton et que le processus *i* veut accéder à la ressource et est le seul à vouloir y accéder, il faut quand même attendre que le jeton fasse tout le tour de l'anneau
  - Avantages
    - Très simple à mettre en oeuvre
    - Intéressant si nombreux processus demandeurs de la ressource
      - Jeton arrivera rapidement à un processus demandeur
      - Équitable en terme de nombre d'accès et de temps d'attente
        - Aucun processus n'est privilégié

Méthode par jeton

- Variante de la méthode du jeton
  - Au lieu d'attendre le jeton, un processus diffuse à tous le fait qu'il veut obtenir le jeton
    - Le processus qui a le jeton sait alors à qui il peut l'envoyer
    - Évite les attentes et les circulations inutiles du jeton
- Algorithme de [Ricart & Agrawala, 83]
  - Soit *N* processus avec un canal bi-directionnel entre chaque processus
    - Canaux fiables mais pas forcément FIFO
  - Localement, un processus *Pi* possède un tableau *nbreq*, de taille *N*
    - Pour *Pi*, *nbreq [j]* est le nombre de requêtes d'accès que le processus *Pj* a fait et que *Pi* connaît (par principe il les connaît toutes)

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Le jeton est un tableau de taille *N*
    - *jeton [i]* est le nombre de fois où le processus *Pi* a accédé à la ressource
    - La case *i* de *jeton* n'est modifiée que par *Pi* quand celui-ci accède à la ressource
  - Initialisation
    - Pour tous les sites *Pi* : $\forall j \in [1 .. N] : nbreq[j] = 0$
    - Jeton : $\forall j \in [1 .. N] : jeton[j] = 0$
    - Un site donné possède le jeton au départ
  - Quand un site veut accéder à la ressource et n'a pas le jeton
    - Envoie un message de requête à tous les processus

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Quand processus *Pj* reçoit un message de requête venant de *Pi*
    - *Pj* modifie son *nbreq* localement : nbreq [i] = nbreq [i] + 1
      - *Pj* mémorise que *Pi* a demandé à avoir la ressource
    - Si *Pj* possède le jeton et est dans l'état *dehors*
      - *Pj* envoie le jeton à *Pi*
  - Quand processus récupère le jeton
    - Il accède à la ressource (passe dans l'état *dedans*)
  - Quand Pi libère la ressource (passe dans l'état *dehors*)
    - Met à jour le jeton : jeton [i] = jeton [i] + 1
    - Parcourt *nbreq* pour trouver un *j* tel que : nbreq [j] > jeton [j]
      - Une demande d'accès à la ressource de *Pj* n'a pas encore été satisfaite : *Pi* envoie le jeton à *Pj*
      - Si aucun processus n'attend le jeton : *Pi* le garde

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83], respect des propriétés
  - Sûreté : seul le processus ayant le jeton accède à la ressource
  - Vivacité : assurée si les processus distribuent équitablement le jeton aux autres processus
    - Méthode de choix du processus qui va récupérer le jeton lorsque l'on sort de l'état dedans
      - *Pi* parcourt *nbreq* à partir de l'indice i+1 jusqu'à *N* puis continue de 1 à i-1
      - Chaque processus teste les demandes d'accès des autres processus en commençant à un processus spécifique et différent de la liste
      - Évite que par exemple tous les processus avec un petit identificateur soient servis systématiquement en premier

Méthodes par permission

- Méthodes par permission
  - Un processus doit avoir l'autorisation des autres processus pour accéder à la ressource
- Principe général
  - Un processus demande l'autorisation à un sous-ensemble donné de tous les processus
  - Deux modes
    - Permission individuelle : un processus peut donner sa permission à plusieurs autres à la fois
    - Permission par arbitre : un processus ne donne sa permission qu'à un seul processus à la fois
      - Les sous-ensembles sont conçus alors tel qu'au moins un processus soit commun à 2 sous-ensembles : il joue le rôle d'arbitre

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81]
  - Permission individuelle
  - Chaque processus demande l'autorisation à tous les autres (sauf lui par principe)
    - Liste des processus à interroger par le processus *Pi* pour accéder à la ressource : Ri = { 1, ... , N } – { i }
  - Se base sur une horloge logique (Lamport) pour garantir le bon fonctionnement de l'algorithme
    - Ordonnancement des demandes d'accès à la ressource
    - Si un processus ayant fait une demande d'accès reçoit une demande d'un autre processus avec une date antérieure à la sienne, il donnera son autorisation à l'autre processus
      - Et passera donc après lui puisque l'autre processus fera le contraire

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81], fonctionnement
  - Chaque processus gère les variables locales suivantes
    - Une horloge *Hi*
    - Une variable *dernier* qui contient la date de la dernière demande d'accès la ressource
    - L'ensemble *Ri*
    - Un ensemble d'identificateurs de processus dont on attend une réponse : *attendu*
    - Un ensemble d'identificateurs de processus dont on diffère le renvoi de permission si on est plus prioritaire qu'eux : *différé*
  - Initialisation
    - Hi = dernier = 0
    - différé = ∅, attendu = Ri

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81], fonctionnement (suite)
  - Si un processus veut accéder à la ressource, il exécute
    - Hi = Hi + 1
    - dernier = Hi
    - attendu = Ri
    - Envoie une demande de permission à tous les processus de *Ri* avec estampille *(Hi, i)*
    - Se met alors en attente de réception de permission de la part de tous les processus dont l'identificateur est contenu dans *attendu*
    - Quand l'ensemble *attendu* est vide, le processus a reçu la permission de tous les autres processus
      - Accède alors à la ressource partagée
      - Quand accès terminé
        - Envoie une permission à tous les processus dont l'id est dans *différé*
        - *différé* est ensuite réinitialisé (différé = ∅)

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81], fonctionnement (suite)
  - Quand un processus *Pi* reçoit une demande de permission de la part du processus *Pj* contenant l'estampille *(H, j)*
    - Met à jour *Hi* : Hi = max (Hi, H)
    - Si *Pi* pas en attente d'accès à la ressource : envoie permission à *Pj*
    - Sinon, si *Pi* est en attente d'accès à la ressource
      - Si *Pi* est prioritaire : place *j* dans l'ensemble *différé*
        - On lui enverra la permission quand on aura accédé à la ressource
      - Si *Pj* est prioritaire : envoi permission à *Pj*
        - *Pj* doit passer avant moi, je lui envoie ma permission
      - La priorité est définie selon la datation des demandes d'accès à la ressource de chaque processus
        - Le processus prioritaire est celui qui a fait sa demande en premier
        - Ordre des dates : l'ordre << de l'horloge de Lamport :
          (dernier, i) << (H, j) si ((dernier < H) ou (dernier = H et i < j))

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81], fonctionnement (fin)
  - Quand processus *Pi* reçoit une permission de la part du processus *Pj*
    - Supprime l'identificateur de *Pj* de l'ensemble attendu : attendu = attendu – { j }
- Respect des propriétés
  - Sûreté : vérifiée (et prouvable...)
  - Vivacité : assurée grâce aux datations et aux priorités associées
- Inconvénient principal de cet algorithme
  - Nombre relativement important de messages échangés

Permission individuelle

- Permission individuelle, amélioration de l'algorithme de [Ricart & Agrawala, 81]
  - [Carvalho & Roucairol, 83]
    - Si *Pi* veut accéder plusieurs fois de rang à la ressource partagée et si *Pj* entre 2 accès (ou demandes d'accès) de *Pi* n'a pas demandé à accéder à la ressource
      - Pas la peine de demander l'autorisation à *Pj* car on sait alors qu'il donnera par principe son autorisation à *Pi*
      - Limite alors le nombre de messages échangés
  - [Chandy & Misra, 84], améliorations tel que
    - Les processus ne voulant pas accéder à la ressource et qui ont déjà donné leur permission ne reçoivent pas de demande de permission
    - Horloges avec des datations bornées (modulo *m*)
    - Pas d'identification des processus

Permission par arbitre

- Permission par arbitre
  - Un processus ne donne qu'une permission à la fois
    - Il redonnera sa permission à un autre processus quand le processus a qui il avait donné précédemment la permission lui a indiqué qu'il a fini d'accéder à la ressource
  - La sûreté est assurée car
    - Les sous-ensemble de processus à qui un processus demande la permission sont construits tel qu'ils y ait toujours au moins un processus commun à 2 sous-ensemble
    - Un processus commun à 2 sous-ensembles est alors arbitre
      - Comme il ne peut donner sa permission qu'à un seul processus, les processus de 2 sous-ensembles ne peuvent pas tous donner simultanément la permission à 2 processus différents
      - C'est donc ce processus commun qui détermine à qui donner la ressource

Permission par arbitre

- Algorithme de [Maekawa, 85]
  - Chaque processus *Pi* possède un sous-ensemble *Ri* d'identificateurs de processus à qui *Pi* demandera l'autorisation d'accéder à la ressource
    - $\forall i,j \in [\ 1..N\ ] : Ri \cap Rj \neq \emptyset$
      - Deux sous-ensembles de 2 processus différents ont obligatoirement au moins un élément en commun (le ou les arbitres)
      - Cela rend donc inutile le besoin de demander la permission à tous les processus, d'où les sous-ensembles Ri ne contenant pas tous les processus
    - $\forall i : | Ri | = K$
      - Pour une raison d'équité, les sous-ensembles ont la même taille pour tous les processus
    - $\forall i$: i est contenu dans D sous-ensembles
      - Chaque processus joue autant de fois le rôle d'arbitre qu'un autre processus

Permission par arbitre

- Algorithme de [Maekawa, 85] (suite)
  - Solution optimale en nombre de permissions à demander et de messages échangés
    - $K \sim \sqrt{N}$ et $D = K$
- Fonctionnement de l'algorithme
  - Chaque processus possède localement
    - Une variable *vote* permettant de savoir si le processus a déjà voté (a déjà donné sa permission à un processus)
    - Une file *file* d'identificateurs de processus qui ont demandé la permission mais à qui on ne peut la donner de suit
    - Un compteur *réponses* du nombre de permissions reçues
  - Initialisation
    - État non demandeur, vote = faux et file = ∅, réponses = 0

Permission par arbitre

- Algorithme de [Maekawa, 85], fonctionnement (suite)
  - Quand processus *Pi* veut accéder à la ressource
    - réponses = 0
    - Envoie une demande de permission à tous les processus de *Ri*
    - Quand *réponses* = | *Ri* |, *Pi* a reçu une permission de tous, il accède alors à la ressource
    - Après l'accès à la ressource, envoie un message à tous les processus de *Ri* pour les informer que la ressource est libre
  - Quand processus *Pi* reçoit une demande de permission de la part du processus *Pj*
    - Si *Pi* a déjà voté (vote = vrai) ou accède actuellement à la ressource : place l'identificateur de *Pj* en queue de *file*
    - Sinon : envoie sa permission à *Pj* et mémorise qu'il a voté
      - vote = vrai

Permission par arbitre

- Algorithme de [Maekawa, 85], fonctionnement (suite)
  - Quand *Pi* reçoit de la part du processus *Pj* un message lui indiquant que *Pj* a libéré la ressource
    - Si *file* est vide, alors vote = faux
      - *Pi* a déjà autorisé tous les processus en attente d'une permission de sa part
    - Si *file* est non vide
      - Retire le premier identificateur (disons *k*) de la file et envoie à *Pk* une permission d'accès à la ressource
      - *vote* reste à vrai

Permission par arbitre

- Algorithme de [Maekawa, 85], problème
  - La vivacité n'est pas assurée par cet algorithme car des cas d'interblocage sont possibles
  - Pour éviter ces interblocages, améliorations de l'algorithme en définissant des priorités entre les processus
    - En datant les demandes d'accès avec une horloge logique
    - En définissant un graphe de priorités des processus

Tolérance aux fautes

- Tolérance aux fautes
  - Les algorithmes décrits dans ce cours ne supportent pas des pertes de messages et/ou des crash de processus
    - Mais adaptation possibles de certains algorithmes pour résister à certains problèmes
      - Ex. pour la méthode par serveur : élection d'un nouveau serveur en cas de crash
    - Peut aussi améliorer la tolérance aux fautes en utilisant des détecteurs de fautes associés aux processus pour détecter les processus morts