École Doctorale de Grenoble
Master 2 Recherche "Systèmes et Logiciel"

Gestion répartie de données - 1

Duplication et cohérence

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

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

Gestion répartie de données : bref historique (1)

Pendant longtemps, la gestion de données est restée centralisée : les données sont conservées en un lieu unique

- Système de gestion de fichiers (SGF classique, composant du système d'exploitation)
- Système de gestion de bases de données (SGBD)

Puis des considération de disponibilité ont amené à dupliquer (localement) les fichiers ou bases de données, totalement ou partiellement

- d'où problèmes de cohérence entre les copies multiples

Vers les années 1975 ont été développés des prototypes de serveurs de fichiers accessibles à distance, puis de SGF répartis

Vers les années 1985 sont apparus les premiers SGF répartis commerciaux (NFS), utilisant un protocole client-serveur (RPC)

Les problèmes de cohérence se posent à propos des différents caches présents dans les SGF. Parallèlement sont développées les transactions réparties pour les SGBD répartis (cf problème de la validation atomique)

© 2003-2004, S. Krakowiak

Gestion répartie de données : bref historique (2)

Au début des années 1980 sont apparus les premiers systèmes à objets répartis (prototypes de recherche).

La notion d'intergiciel (*middleware*) est issue de ces travaux. La première norme concernant un intergiciel à objets (CORBA) est sortie en 1991. D'autres formes d'intergiciel (composants logiciels) ont suivi depuis.

Les problèmes de cohérence dus à la duplication de données (pour la disponibilité et l'efficacité d'accès) et à la réalisation de transactions sont toujours présents

Principaux problèmes nouveaux :

- Modèles de programmation (client-serveur, objects répartis, événements asynchrones, composants) et liaison avec les langages
- Structures de l'intergiciel pour l'accès à des informations distantes
- Sécurité

Ces aspects sont traités dans le cours "Construction d'applications réparties"

© 2003-2004, S. Krakowiak

Gestion répartie de données : bref historique (3)

Le Web apparaît au début des années 1990 et modifie la vision de l'accès à l'information. Le potentiel est celui d'un vaste SGBD, peu structuré, à l'échelle de tout l'Internet, dont le contenu et même l'organisation changent en permanence

Les modèles d'organisation des SGF et SGBD classiques sont trop rigides et ne passent pas à grande échelle. On explore de nouvelles formes, comme les systèmes pair à pair (*peer to peer, P2P*), et les outils correspondants

- modèles à événements
- diffusion à grande échelle
- nouvelles formes de stockage réparti

Autre aspect : les réseaux mobiles. Caractéristiques

- connectivité non permanente et de qualité variable
- localisation variable des données

La gestion de données dans ces réseaux est un autre thème de recherche actuel

© 2003-2004, S. Krakowiak

Gestion répartie de données : problèmes abordés

Duplication et cohérence des données (*)
- Modèles de cohérence
- Protocoles pour la cohérence
- Exemples

Diffusion à grande échelle
- Algorithmes épidémiques
- Exemples

Systèmes pair à pair
- Classification et choix de conception
- Exemples
- Tables de hachage réparties (DHT)

(*) Source et référence pour la duplication et la cohérence :
A. S. Tanenbaum & M. van Steen, *Distributed Systems - Principles and Paradigms*, Prentice Hall, 2002, ch. 6

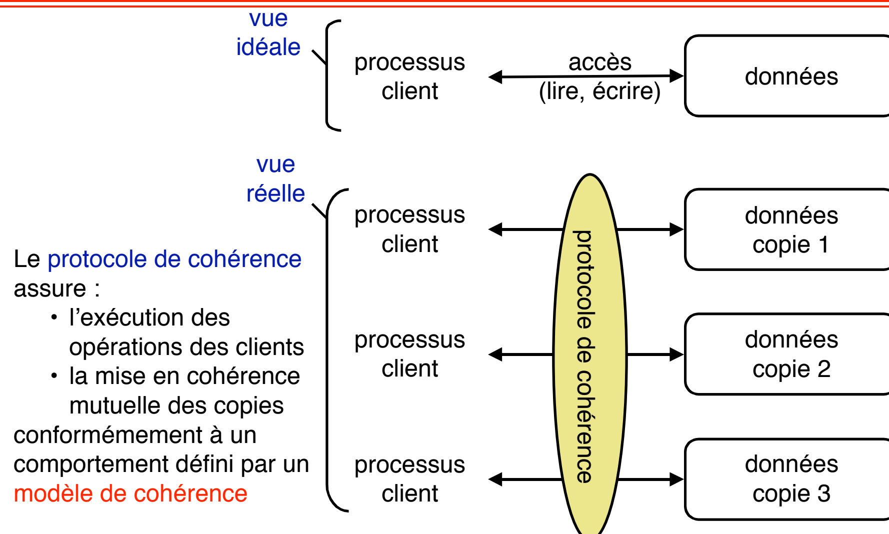
© 2003-2004, S. Krakowiak

Duplication et cohérence

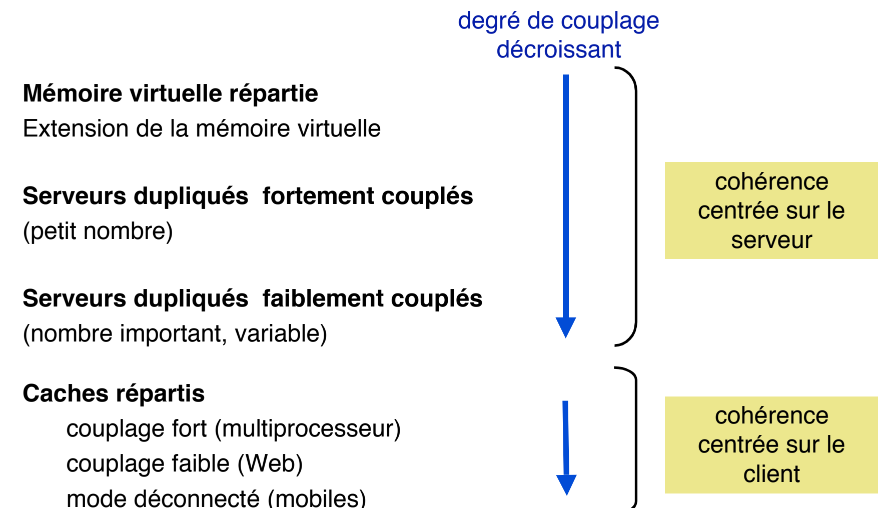
- **Pourquoi dupliquer les données ?**
  - **Disponibilité des données : permettre l'accès aux données même en cas de défaillance d'un support**
    - **technique déjà utilisée pour des données centralisées, avec copie(s) locale(s)**
  - **Rapidité d'accès : placer une copie des données "près" (en temps d'accès) de leur point d'utilisation**
    - **technique des caches, déjà utilisée en centralisé**
    - **extension aux "caches logiciels", données locales vs données distantes**
- **Cohérence : la contrepartie de la duplication**
  - **Les diverses copies d'une même donnée doivent être cohérentes, c'est-à-dire "apparaître comme une copie unique"**
  - **En fait, cette notion même de cohérence peut avoir de nombreuses interprétations**
  - **Le maintien de la cohérence a un coût. Il faut faire un compromis entre le coût et la "qualité" de la cohérence**

© 2003-2004, S. Krakowiak

Accès à des données dupliquées



Divers cas de duplication de données



Modèle de cohérence

Un modèle de cohérence pour des données réparties spécifie un contrat entre un client et le système de gestion de données (avec engagement mutuel)

Les données réparties sont un ensemble d'éléments (un élément peut avoir une granularité quelconque : octet, enregistrement, fichier, etc.). Les opérations sont lire et écrire (éventuellement des opérations plus complexes, mais toujours divisées entre consultation et modification)

Le modèle de cohérence le plus contraignant est celui qui correspond exactement à la "vue idéale" des données

Cohérence stricte
Toute lecture sur un élément x renvoie une valeur correspondant à l'écriture la plus récente sur x

Cohérence stricte

Toute lecture sur un élément x renvoie une valeur correspondant à l'écriture la plus récente sur x

La réalisation d'un protocole réalisant la cohérence stricte dans un système réparti pose deux problèmes.

- Définition de "l'événement le plus récent"
- Réalisation "instantanée" des opérations

La définition de l'événement le plus récent nécessite des horloges parfaitement synchronisées (écart nul entre deux sites quelconques)

Une opération qui nécessite un accès distant ne peut être instantanée (plus précisément, une opération locale lancée après une opération distante peut se terminer avant)

La cohérence stricte est un modèle idéal (non réalisable), que l'on essaie d'approcher au moyen de modèles moins contraignants

Linéarisabilité

La condition de linéarisabilité a déjà été définie pour les serveurs dupliqués, pour des requêtes transmises aux serveurs. L'accès à des données dupliquées en est un cas particulier, dans lequel les opérations se réduisent à lire et écrire

Le résultat de l'exécution d'un ensemble de processus clients est identique à celui d'une exécution dans laquelle :

- Toutes les opérations sur les données (vues comme centralisées) sont exécutées selon une certaine séquence S
- Si deux opérations op1 et op2 (lectures ou écritures) sont telles que op1 → op2, alors op1 et op2 figurent dans cet ordre dans S
- La cohérence interne des données est respectée dans S (après une écriture, et jusqu'à la suivante, une lecture délivre la valeur écrite)

Rappel : on dit que op1 → op2 si t(fin(op1)) < t(début(op2)), où t est une date vérifiant la validité forte (en pratique une heure donnée par des horloges physiques synchronisées)

Cohérence séquentielle

La linéarisabilité est coûteuse à réaliser en pratique. Aussi définit-on une condition plus faible, la cohérence séquentielle :

Le résultat de l'exécution d'un ensemble de processus clients est identique à celui d'une exécution dans laquelle :

- Toutes les opérations sur les données (supposées centralisées) sont exécutées selon une certaine séquence S
- Les opérations exécutées par tout processus p figurent dans S dans le même ordre que dans p
- La cohérence interne des données est respectée dans S

Différence par rapport à la linéarisabilité : on ne contraint pas l'ordre relatif des opérations dans des processus différents, tant que la cohérence interne des données est respectée

Linéarisabilité et cohérence séquentielle

contrainte de temps : W(x)a précède W(x)b

p_1 W(x)a
p_2 W(x)b
p_3 R(x)a R(x)b
p_4 R(x)a R(x)b

Notation :
$R_i(x)a$: par p_i, lecture de x, résultat a
$W_i(x)a$: écriture de x, valeur a

S = W(x)a R_3(x)a R_4(x)a W(x)b R_3(x)b R_4(x)b

linéarisable

p_1 W(x)a
p_2 W(x)b
p_3 R(x)a R(x)b
p_4 R(x)a R(x)b

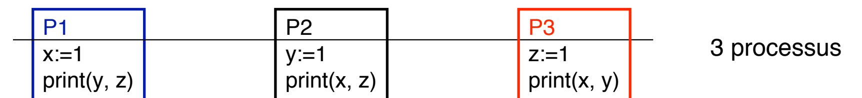
S = W(x)a R_3(x)a R_4(x)a W(x)b R_4(x)b R_3(x)b

séquentiel, non linéarisable

p_1 W(x)a
p_2 W(x)b
p_3 R(x)b R(x)a
p_4 R(x)a R(x)b

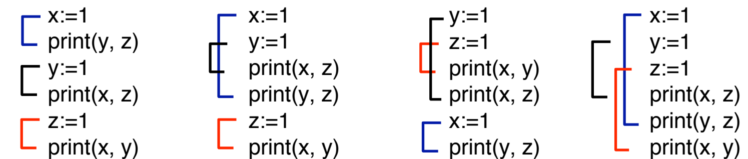
non séquentiel

Autre exemple de cohérence séquentielle



3 processus

4 exécutions possibles parmi les 90 qui respectent la cohérence séquentielle



Impression réalisée

001011 101011 010111 111111

Mais le résultat 000000 est impossible à obtenir en respectant la cohérence séquentielle

Conclusion sur la cohérence séquentielle

Modèle moins strict que la linéarisabilité, car il permet une classe plus large d'ordonnancements différents (grâce au découplage entre processus)

Néanmoins un protocole de cohérence séquentielle reste encore coûteux à réaliser. En effet, on peut montrer que :

si

- t est le temps minimal de transfert d'un message élémentaire entre 2 sites
- r est la durée d'une lecture
- w est la durée d'une écriture

alors

$r + w \geq t$

Autrement dit, tout gain sur le temps de lecture entraîne une perte sur le temps d'écriture, et vice-versa

Cohérence causale (1)

Modèle plus faible que la cohérence séquentielle, car on ne considère que des événements reliés par une relation de causalité
Soit deux événements E1 et E2 tels que E1 → E2 (précédence causale).
Alors tout processus doit "voir" E1 avant E2

Exemples :

- si E1: p écrit x, puis E2: q lit x, alors E1 → E2
- si E1: p lit x, puis E2: p écrit y, alors E1 → E2 (car la valeur écrite par p peut dépendre d'un calcul fait à partir de x)
- si E1: p écrit x, puis E2: q écrit y (de manière indépendante), alors E1 II E2

La condition sur écriture et lecture est déjà incluse dans la cohérence interne des données
Reste à spécifier la condition sur les écritures

Cohérence causale (2)

Définition :
Des écritures causalement liées (→) doivent être vues par tous les processus dans leur ordre causal. Des écritures causalement indépendantes (||) peuvent être vues dans un ordre différent

p_1	W(x)a		W(x)c		
p_2	R(x)a	W(x)b			
p_3	R(x)a			R(x)c	R(x)b
p_4	R(x)a			R(x)b	R(x)c

Les lectures de p_3 et p_4 renvoient b et c dans des ordres différents. C'est possible car les écritures sont causalement indépendantes

Ce scénario est impossible avec la cohérence séquentielle

La dépendance causale peut être détectée avec des horloges vectorielles. Celles-ci peuvent donc servir à réaliser un protocole de cohérence causale (cf exemple plus loin, protocole de Ladin et al.)

Cohérence FIFO

La cohérence causale peut encore être affaiblie, si on ne considère la causalité qu'à l'intérieur d'un seul processus, non entre processus différents. Dans un processus unique, la causalité se réduit à l'ordre FIFO

Définition :
Des écritures réalisées par un même processus doivent être vues par tous les processus dans leur ordre de réalisation. Des écritures réalisées par des processus différents peuvent être vues dans un ordre différent

p_1	W(x)a					
p_2	R(x)a	W(x)b	W(x)c			
p_3				R(x)b	R(x)a	R(x)c
p_4				R(x)a	R(x)b	R(x)c

R(b) précède R(c) partout ; R(a) est indépendant

Réalisation : il suffit d'un compteur (scalaire) par processus

Cohérence utilisant la synchronisation (1)

En fait, il n'est pas toujours nécessaire d'assurer la cohérence à tout instant. Par exemple, si un processus modifie les données à l'intérieur d'une section critique, les autres processus ne peuvent pas voir les états intermédiaires. Il n'est donc pas nécessaire d'imposer des contraintes sur les opérations internes à une section critique

D'où l'idée d'associer synchronisation et maintien de la cohérence, à l'aide de variables et d'opérations de synchronisation.

Une variable de synchronisation S est associée à un ensemble de données. L'appel de l'opération synchronize(S) par un processus p provoque la mise en cohérence des données locales (vues par p). En dehors de ces opérations, la cohérence peut ne pas être assurée.

Les modèles diffèrent selon le moment où la synchronisation est réalisée.

Cohérence utilisant la synchronisation (2)

La cohérence avec synchronisation (ou **cohérence faible**) est définie ainsi :

- L'accès aux variables de synchronisation associées à un ensemble de données respecte la cohérence séquentielle [les opérations de synchronisation sont vues par tous les processus dans le même ordre]
- Un processus ne peut pas exécuter une opération sur une variable de synchronisation tant que toutes les écritures antérieures n'ont pas été exécutées sur les copies locales [la synchronisation force la mise en cohérence de toutes les copies locales]
- Un processus ne peut pas exécuter une lecture ou une écriture tant qu'il n'a pas exécuté toutes les opérations antérieures sur les variables de synchronisation [après synchronisation, un processus peut accéder à la version la plus récente des données synchronisées]

Cohérence utilisant la synchronisation (3)

Exemples

p_1	W(x)a W(x)b S	
p_2		R(x)a R(x)b S
p_3		R(x)b R(x)a S

possible

p_1	W(x)a W(x)b S	
p_2		S R(x)a

impossible

Pour avoir un contrôle encore plus fin sur la synchronisation, on peut spécifier le moment auquel la synchronisation (mise en cohérence) doit avoir lieu, dans le cas d'une suite d'opérations exécutées en section critique

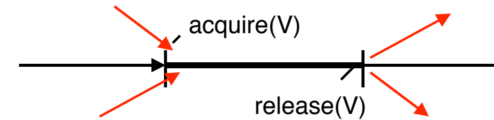
- Cohérence à la sortie (*release consistency*)
- Cohérence à l'entrée (*entry consistency*)

Cohérence à la sortie (*release consistency*) (1)

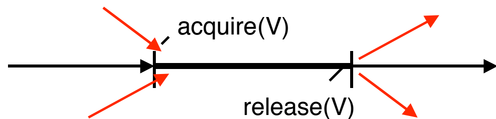
On peut protéger une séquence d'opérations sur des éléments (ou variables) dans un processus en l'incluant entre acquire(V) et release(V), où V est une variable de synchronisation (verrou)

Au moment de acquire : toutes les copies locales des variables protégées sont mises à jour pour tenir compte des écritures non encore répercutées

Au moment de release : toutes valeurs des variables locales qui ont été modifiées sont envoyées vers les copies distantes.

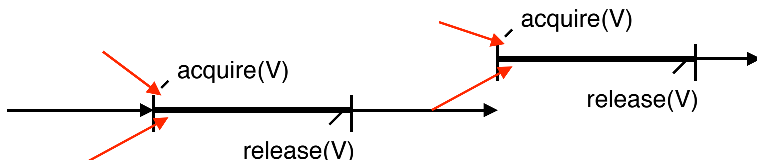


Cohérence à la sortie (*release consistency*) (2)



Une variante : cohérence paresseuse à la sortie (*lazy release consistency*)

Les modifications ne sont pas propagées à la sortie. Elles sont seulement faites (*pull*) à la prochaine entrée (par le processus considéré ou par un autre processus)



Cohérence à l'entrée (*entry consistency*)

La cohérence à l'entrée est voisine de la cohérence paresseuse à la sortie. La différence est que toute variable modifiée indépendamment doit être associée à un verrou spécifique. Au moment de acquire, seules les variables associées au verrou utilisé sont mises à jour

p_1	Acq(Vx) W(x)a Acq(Vy) W(y)b Rel(Vx) Rel(Vy)	
p_2		Acq(Vx) R(x)a R(y)nil
p_3		Acq(Vy) R(y)b

Conclusion sur la cohérence avec synchronisation

La cohérence avec synchronisation permet de spécifier de manière fine les variables à protéger, en utilisant des informations sur les accès prévus

La motivation est de réduire le nombre d'opérations de mise en cohérence

Cadre principal d'application : **mémoire virtuelle répartie**

Exemples

Cohérence paresseuse à la sortie : Treadmarks (Rice)
Cohérence à l'entrée : Midway (CMU)

Cohérence à terme (1)

Dans beaucoup de situations pratiques
- La plupart des accès sont des lectures
- Les conflits d'écriture sont très rares (les données sont partitionnées en sous ensembles et un processus unique écrit dans chaque sous-ensemble)
- Il est acceptable de lire une donnée non à jour

Exemples : le DNS (Domain Naming System) ; le World Wide Web ; certaines bases de données

Dans ce cas, on peut tolérer des données incohérentes pendant un certain temps, si on respecte la condition suivante :

En l'absence de mises à jour pendant un temps "assez long", toutes les copies finiront par devenir cohérentes.

C'est la **cohérence à terme** (*eventual consistency*)

Cohérence à terme (2)

La cohérence à terme est réalisée par des protocoles qui propagent les modifications à l'ensemble des copies, avec une propriété de convergence à terme. Voir plus loin (algorithmes dits épidémiques)

Les conflits en écriture sont en principe peu fréquents (par hypothèse) et réglés au cas par cas

Néanmoins ce type de cohérence peut poser un problème lorsqu'un client s'adresse successivement à plusieurs copies différentes d'une donnée.

Exemples
- clients mobiles
- utilisation de caches

D'où une classe de modèles de cohérence définis à partir de la vue du client

Modèles de cohérence centrés sur le client (1)

Lectures monotones

Si un processus a lu la valeur d'une donnée x, toute lecture ultérieure par ce même processus doit rendre la même valeur ou une valeur plus récente

Ce modèle garantit qu'un client ne "reviendra pas en arrière"

Écritures monotones

Si un processus exécute deux écritures successives sur une même donnée x, la deuxième écriture ne peut être réalisée que quand la première a été terminée

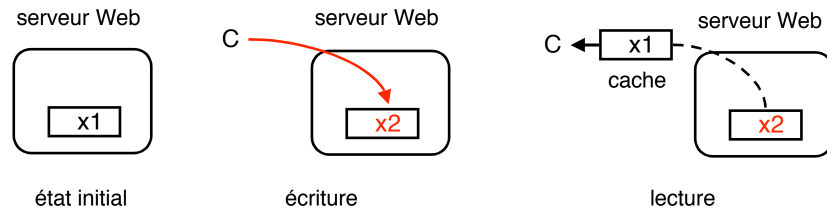
Ce modèle garantit qu'un client ne fera une modification sur une copie que quand cette copie sera à jour des modifications antérieures par ce même client.
Ce modèle est voisin du modèle de cohérence FIFO, mais ne concerne que la vue d'un processus client unique et non celle de tous les processus

Modèles de cohérence centrés sur le client (2)

Cohérence écriture-lecture (*Read Your Writes*)

Si un processus a modifié la valeur d'une donnée x, cette modification doit être visible par toute lecture ultérieure par ce même processus

Un exemple courant de violation de cette propriété



état initial   écriture   lecture

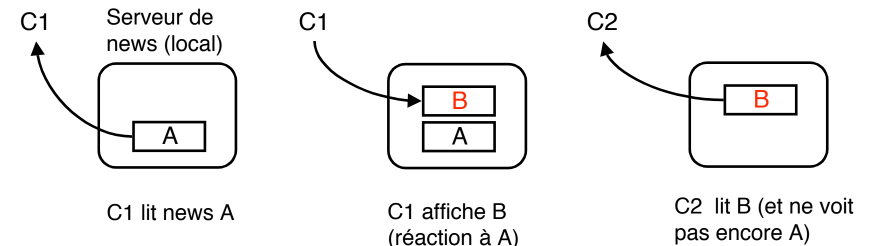
Le problème vient de ce que le cache du navigateur n'est utilisé que pour la lecture, pas pour l'écriture. Dans la pratique, on rafraîchit explicitement le cache

Modèles de cohérence centrés sur le client (3)

Cohérence lecture-écriture (*Writes Follows Reads*)

Si un processus exécute une écriture sur une donnée x après avoir lu cette donnée, l'écriture va modifier (partout) une valeur de la donnée au moins aussi récente que la valeur lue

Un exemple de violation de cette propriété (donnée = un *newsgroup*)



C1 lit news A   C1 affiche B (réaction à A)   C2 lit B (et ne voit pas encore A)

Mise en œuvre des modèles de cohérence (1)

Objectif commun : propagation des écritures aux différentes copies, en essayant d'éliminer ou de réduire
- le nombre et le volume des transferts
- la taille de l'information conservée

Problème 1 : **Définition et placement des copies**

Solution 1 : copies permanentes. Ensemble de copies définies a priori

Solution 2 : copies temporaires crées par le serveur. Exemple : copies crées par un serveur Web pour résister à un pic de demande. Le nombre de copies d'une information particulière peut être lié à la demande constatée pour cette information

Solution 3 : copies créées par les clients. Exemple : diverses formes de cache liées à un client ou à un groupe de clients

Mise en œuvre des modèles de cohérence (2)

Problème 2 : **Propagation des mises à jour**

Mode de propagation.

Soit X une donnée en exemplaires multiples. Après modification d'une copie locale X1 de X par une certaine opération op(X),quelles informations vont-elles être transmises pour propager cette modification ?

Invalidation. On signale à chaque serveur gérant une copie distante que sa copie est maintenant invalide. Avantage : information minimale

Recopie de données. On transfère une copie des données modifiées. Avantage : mise à jour rapide ; inconvénient : volume des données transmises

Opération à distance. On signale à chaque serveur distant qu'il doit réaliser la modification, en refaisant l'opération op(X). Avantage : peu d'informations transmises, mais peut occuper les serveurs si opération complexe

Mise en œuvre des modèles de cohérence (3)

Problème 2 : **Propagation des mises à jour (suite)**

Initiative de la propagation

Initiative du serveur (*push*)
En général réalisé pour la propagation entre serveurs permanents, et lorsqu'on recherche un haut niveau de cohérence
Plus compliqué à réaliser lorsqu'il y a un nombre variable de copies, et pour des copies temporaires telles que les caches (par exemple, pratiquement irréalisable pour un serveur Web)

Initiative du client (*pull*)
Souvent réalisée dans les systèmes de caches. Le gérant du cache vérifie que la copie en cache est toujours fraîche en la comparant à la copie de référence. En fait il suffit de comparer les dates de dernière modification (estampilles)
Souvent, on impose au cache une fréquence minimale de rafraîchissement (par exemple caches clients des fichiers NFS : 3 secondes pour les fichiers, 30 secondes pour les catalogues)

Protocoles de cohérence (1)

La réalisation d'un protocole de cohérence dépend du modèle de cohérence que l'on souhaite mettre en œuvre.
On a déjà vu (pour les serveurs) qu'un modèle de linéarisabilité pouvait être réalisé de deux manières : **serveur (ou copie) primaire** et **duplication active** (si les horloges sont bien synchronisées). Cela reste vrai pour la gestion de données réparties, avec les mêmes algorithmes (déjà vus)

Deux variantes pour la **copie primaire**

Serveur fixe. Le serveur primaire qui gère une copie de chaque donnée est fixé. Si la mise à jour est faite sur un autre site, on fera un appel à distance

Serveur local. La mise à jour est toujours faite localement. Pour cela, on transfère d'abord une copie du primaire sur le site local (le primaire initial se comporte comme un serveur de secours). Avantage quand on a une succesion de modifications par le même client, et que les clients distants peuvent éventuellement lire des versions périmées.

Protocoles de cohérence (2)

Pour la **duplication active**, on rappelle qu'il faut réaliser la diffusion totalement ordonnée (atomique) des modifications. En général, on réalise cette diffusion par un algorithme à base de séquenceur (vu précédemment).

Les algorithmes de diffusion atomique ne passent pas bien à l'échelle car ils nécessitent un nombre élevé de messages pour être tolérants aux fautes. Pour un grand nombre de copies, on utilise plutôt un modèle de cohérence à terme (*eventual consistency*), et des algorithmes de diffusion épidémique, à voir plus tard.

Pour un nombre modéré de copies, on peut rechercher une optimisation entre degré de duplication (pour résister aux défaillances) et efficacité. Pour cela, on utilise des techniques de vote.

Votes et quorums pour la duplication (1)

Idée de base : un client doit avoir l'autorisation de plusieurs serveurs (quorum) avant de réaliser une écriture sur une donnée

Par exemple, si on a N serveurs, on peut spécifier qu'il faut l'autorisation de plus de N/2 d'entre eux (un serveur donne l'autorisation s'il est vivant). Ainsi la modification sera toujours faite sur une majorité de serveurs.

Exemple : si N = 3, on ne peut écrire que si 2 serveurs se disent prêts à accepter l'écriture. Ensuite, on est certain que si on consulte 2 serveurs quelconques sur les 3, l'un au moins aura une copie à jour de la dernière modification (il suffit de regarder les estampilles pour connaître la version la plus récente)

On peut généraliser ce principe à toutes les opérations de lecture ou d'écriture.

Votes et quorums pour la duplication (2)

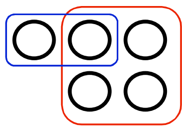
Quorums en lecture et écriture

N = nombre de copies
nr = quorum de lecture, nw = quorum d'écriture, **nr + nw > N**

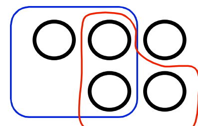
Cette condition garantit que l'on pourra toujours accéder en lecture à la version la plus récente
Les quorums en lecture et écriture sont différents selon les propriétés souhaitées

Exemple

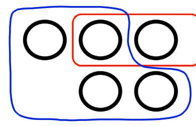
N=5, nr = 2, nw = 4



N=5, nr = 3, nw = 3



N=5, nr = 4, nw =2

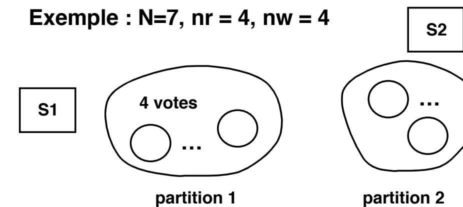


Votes et quorums pour la duplication (3)

- **Principe**
  - **Donner des poids différents aux copies selon leur importance présumée (raisons d'accessibilité, administration, etc.)**
  - **Utilité : partition de réseau (copies inaccessibles)**

N = nombre de votes disponibles (et non plus nombre de copies)
nr = quorum de lecture, nw = quorum d'écriture
nr + nw > N et 2w > N

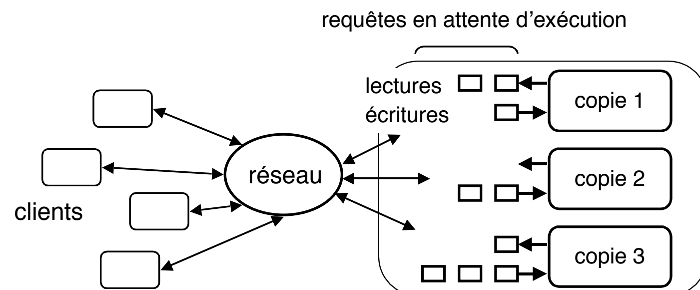
Exemple : N=7, nr = 4, nw = 4



Si S1 écrit dans la la partition 1, on est sûr que S2 ne peut pas écrire dans la partition 2 (car elle ne peut pas avoir plus de 3 votes
Il est également impossible à S2 de lire pendant que S1 écrit (lectures et écritures sont sérialisées)

Un exemple de protocole de cohérence géré par les serveurs (Ladin et al.)

Serveur de données dupliquées réalisant la cohérence à terme, mais préservant la dépendance causale entre opérations



R.Ladin, B. H. Liskov, L. Shrira, S. Ghemataw. Providing Availability Using Lazy Replication,*ACM Trans. on Computer Systems (TOCS)*, vol. 10, 4, Nov. 1992

Protocole de cohérence de (Ladin et al.) : principe

Serveur de données dupliquées réalisant la **cohérence à terme**, mais préservant la **dépendance causale** entre opérations.

Pour réaliser la dépendance causale : estampilles analogues aux horloges vectorielles. Une opération n'est réalisée que si toutes les opération dont elle dépend causalement (potentiellement) ont été réalisées. Donc on doit retarder certaines opérations (d'où les files d'opérations en attente) : analogie avec la diffusion causale avec horloges vectorielles

Pour réaliser la cohérence à terme : échange périodique, entre les serveurs, d'informations sur les mises à jour (protocole dit "anti-entropie"). À terme, toute copie est "au courant" de toutes les modifications et celles-ci peuvent être réalisées, en respectant la dépendance causale.

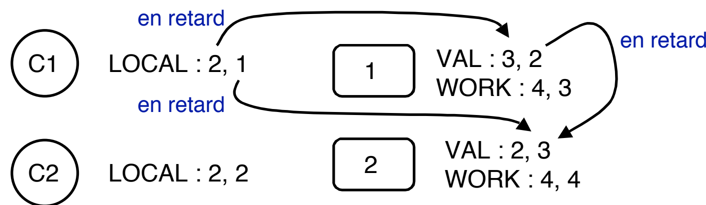
Protocole de (Ladin et al.) : données

Chaque copie locale L_i maintient deux vecteurs VAL(i) et WORK(i)

- VAL(i)[j] : nombre de mises à jour exécutées sur L_i depuis L_j
  - (VAL(i)[i] : écritures locales)
- WORK(i)[j] : nombre de mises à jour totales (faites et à faire) sur L_i depuis L_j (WORK(i)[i] : écritures locales)

Chaque client C maintient un vecteur LOCAL(C)

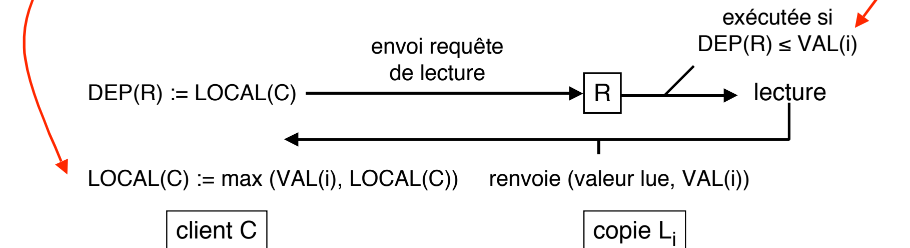
- LOCAL(C)[i] : état de L_i vu par C (nombre de mises à jour)



Protocole de (Ladin et al.) : lecture

Principe : on envoie la requête à une copie (par exemple la plus proche du client).

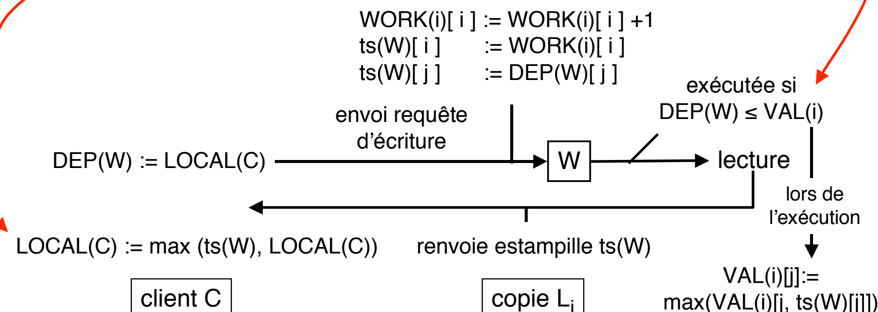
- La requête sera exécutée lorsque toutes les opérations causalement antérieures auront été exécutées
- Au retour, le vecteur LOCAL du client sera mis à jour pour refléter sa nouvelle connaissance de l'état des serveurs



Protocole de (Ladin et al.) : écriture

Principe : comme pour la lecture, on envoie la requête à une copie (par exemple la plus proche du client).

- La requête sera exécutée lorsque toutes les opérations causalement antérieures auront été exécutées
- Au retour, le vecteur LOCAL du client sera mis à jour pour refléter sa nouvelle connaissance de l'état des serveurs



Protocole de (Ladin et al.) : mise en cohérence mutuelle

Principe : les différents serveurs exécutent périodiquement un protocole d'anti-entropie (échange mutuel des dernières modifications connues)

Opération de base :

- L_i envoie à L_j les requêtes d'écriture en attente dans sa file et son vecteur WORK(i)
- L_j met à jour son propre vecteur WORK(j) (règle du max.) et fusionne les requêtes en écritures de L_i et L_j
- L_j examine sa nouvelle file de requêtes en écriture pour déterminer si une opération peut être exécutée
  - Pour cela, on utilise les estampilles DEP(W) attachées à chaque requête W
  - Une requête W1 peut être exécutée si DEP(W1)≤VAL(j) et si elle ne dépend d'aucune autre requête non exécutée :
    $\nexists$ W : ($\forall$ k : DEP(W)[k] ≤ DEP(W1)[k])
    Les requêtes d'écriture exécutables sont successivement exécutées et retirées de la file

Optimisation : lorsqu'une écriture a été exécutée partout (à vérifier par échanges), la requête correspondante est sortie des files W (pour limiter leur taille)

Implémentation de la cohérence centrée sur le client (1)

Les données sont gérées par un ensemble de serveurs. Chaque serveur contient une copie des données. Une opération locale a lieu sur un serveur particulier.
Toute écriture sur un serveur S_i reçoit un identificateur global d'écriture WID, et une estampille ts(WID).
Tout serveur S_i gère un vecteur $RCVD_i$, où $RCVD_i[k]$ est l'estampille de la dernière écriture faite sur le serveur S_k, reçue et traitée par S_i

Quand un client doit faire une opération sur un serveur particulier, le serveur lui renvoie une estampille avec le résultat

L'ensemble de lecture d'un client est l'ensemble des identificateurs d'écriture associés aux lectures (c'est-à-dire, pour toute lecture, l'identificateur global pour l'écriture qui a engendré la valeur lue).
L'ensemble d'écriture d'un client est l'ensemble des identificateurs des écritures faites par ce client

Implémentation de la cohérence centrée sur le client (2)

Pour chacun des ensembles A de lecture et d'écriture d'un client, on note VT(A) un vecteur tel que VT(A)[i] = l'estampille maximale de toutes les opération de A lancées à partir du serveur i.

On peut alors définir des opérations sur ces ensembles
union : VT(A+B) tel que VT(A+B)[i] = max(VT(A)[i], VT(B)[i])
inclusion : A inclus dans B si pour tout i, VT(A)[i] ≤ VT(B)[i]

Ces estampilles peuvent servir à réaliser les modèles ci-dessus.

Exemple : lecture monotone.

Soit $RCVD_i[k]$ l'estampille de la dernière écriture faite sur le serveur S_k. Soit VT(RSet) l'estampille de l'ensemble de lecture pour un client C. Alors on exécute, pour tout k : VT(RSet)[k] := max(VT(RSet)[k], $RCVD_i[k]$). donc l'ensemble de lecture du client C reflète toutes les opérations d'écriture vues par C. Avant de délivrer la valeur lue au client, le serveur local doit vérifier que toutes les écritures décrites dans VT(RSet) ont été faites.

Mode déconnecté

- **Motivations**
  - **Généralisation de l'usage des portables**
  - **Développement des communications sans fil**
  - **La plupart du temps, les portables sont utilisés**
    - **soit en mode autonome**
    - **soit connectés à un réseau**
  - **On cherche à assurer une transition facile entre les deux modes**
- **Principe**
  - **Idée de départ : mécanisme AFS (mise en cache de fichiers entiers sur le poste client)**
  - **Problèmes**
    - **choix des fichiers à conserver lors de la déconnexion**
    - **mise en cohérence à la reconnexion**

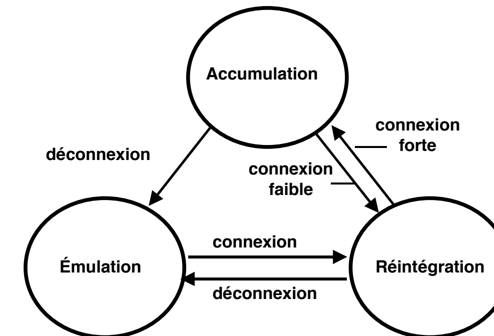
Exemple de système en mode déconnecté : Coda

- **Historique**
  - **Projet de recherche à Carnegie Mellon Univ. (suite d'Andrew, 1990-...) - Coda = *Constant Data Availability***
  - **Actuellement : intégré dans Linux**
- **Objectifs**
  - **Viser une disponibilité permanente de l'information chez le client, y compris dans les cas suivants**
    - **déconnexion volontaire**
    - **déconnexion accidentelle**
    - **faible connectivité**
      - **connexion intermittente (communications mobiles)**
      - **connexion à très faible débit**
- **Principes**
  - **Duplication des serveurs**
  - **Mode déconnecté**

Coda : principes de conception

- **Extension d'AFS**
  - séparation client-serveur
  - cache client sur disque
  - cache de fichiers entiers
- **Duplication des serveurs AFS**
  - Un volume peut avoir des copies sur plusieurs serveurs
  - Augmente la probabilité qu'un volume soit accessible
- **Modification du client AFS**
  - Pendant la connexion, cherche à stocker en cache les fichiers potentiellement utiles
  - En mode déconnecté, utilise les fichiers en cache
  - À la reconnexion, processus de réintégration
    - automatiser autant que possible
    - stratégie "optimiste"

Coda : états du client



- Accumulation : mode pleinement connecté ; le client collecte les fichiers à conserver
- Émulation : mode déconnecté ; le client utilise les fichiers en cache
- Réintégration : état intermédiaire ; le client réintègre les modifications effectuées durant la déconnexion

Coda : mode accumulation (1)

- **Ouverture d'un fichier**
  - Accès à un fichier *f* d'un volume *V* : déterminer l'ensemble des serveurs (*Volume Storage Group*, VSG) stockant une copie de *V*
  - Parmi ces serveurs, déterminer ceux qui sont accessibles (*Available VSG*, AVSG), et en choisir un parmi ceux-ci ("serveur favori")
  - Charger une copie du fichier (avec témoin de rappel sur le serveur)
- **Fermeture d'un fichier**
  - Si modifié, renvoyer la copie à tous les serveurs de l'AVSG
  - Si AVSG ≠ VSG (certains serveurs non accessibles), protocole de remise à jour entre serveurs
  - Si d'autres témoins de rappel présents, le serveur favori exécute les rappels

Coda : mode accumulation (2)

- **Détection d'incohérence**
  - Si tous les serveurs accessibles n'ont pas la même version du fichier (détection par estampilles)
    - le client charge la plus récente
    - le client déclenche un protocole de mise à niveau (exécuté par les serveurs) - intervention manuelle si nécessaire
- **Chargement anticipé des fichiers**
  - En mode accumulation, le client charge des fichiers en tâche de fond pour préparer période de déconnexion
    - selon une liste de préférences fournie par l'utilisateur
    - selon un algorithme d'apprentissage (observation des accès)

Coda : mode accumulation (3)

- Le client surveille la disponibilité des serveurs
  - Toutes les T unités de temps (de l'ordre de 10 minutes), le client examine le VSG des fichiers qu'il a en cache
  - Si variation de l'AVSG (panne ou réinsertion d'un serveur), mise à jour éventuelle des témoins de rappel chez le client (invalidation des copies en cache)
    - réinsertion d'un serveur : invalider les fichiers dont ce serveur a une copie (car sa version est peut-être plus récente)
    - disparition d'un serveur "favori" pour un volume : invalider les fichiers de ce volume
    - disparition d'un autre serveur : rien à faire

Coda : mode émulation

- En mode émulation (déconnecté) le client utilise les fichiers en cache local
  - Les modifications sont journalisées sur le disque pour être rejouées ultérieurement
  - Erreur si défaut de fichier dans le cache (en principe rare)

Coda : mode réintégration (2)

- Restauration de la cohérence
  - Les modifications effectuées pendant la déconnexion sont "rejouées" et propagées vers les serveurs
  - Si la connectivité est faible, propagation "goutte à goutte" en tâche de fond pour ne pas dégrader les performances
- Résolution des conflits
  - Si un conflit est détecté (modifications pendant la déconnexion)
    - le système tente une résolution automatique
      - version "dominante"
      - modifications permutables (ajouts dans catalogue)
    - si impossible, alors résolution manuelle

Coda : conclusion

- Expérience utile pour le mode déconnecté
  - Plusieurs versions successives
  - Mesures et observations
- Conclusions pratiques
  - Les conflits non résolubles sont en fait très rares
    - justifie a posteriori la stratégie optimiste
  - La disponibilité est satisfaisante
    - pas de pertes de données
  - Le coût de la réintégration est acceptable

Pour en savoir plus :

http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html