Septième partie

Mémoire partagée répartie

Contenu de cette partie

Toujours plus d'abstraction. . .
construction d'un environnement virtuellement centralisé

- Mémoire globale
  - principe
  - cohérence de copies multiples
  - abstraction de la communication et intégration des défaillances : registres
- (Temps global : synchroniseurs et temps virtuel global)
- (Gestion des pannes partielles)

Sources, références, compléments

- *A. S. Tanenbaum & M. van Steen,*
  Distributed Systems - Principles and Paradigms,
  ch. 7 : Consistency and Replication,
  accès libre : https ://www.distributed-systems.net
- *Sacha Krakowiak / A. Kshemkalyani & M. Singhal / M Raynal :*
  références en début de cours

Plan

Mémoire partagée répartie

Motivation

(re) placer le développeur d'applications dans les conditions d'un environnement centralisé :
→ communication et synchronisation par variables/objets partagés

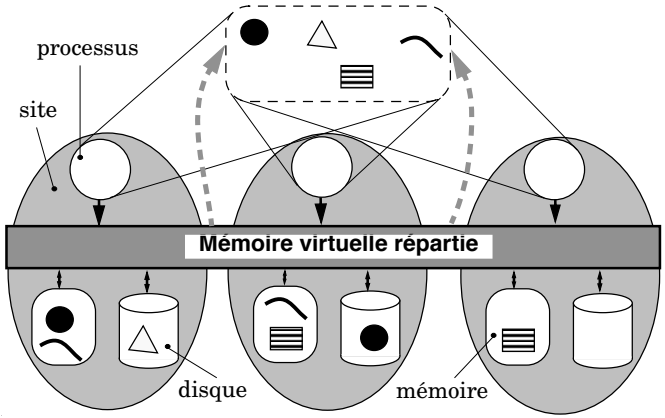
Avantages attendus pour le programmeur

- simplicité (communication implicite, espace d'adressage unique),
- efficacité potentielle (gestion automatique, heuristiques efficaces)
- capacité mémoire, bande passante
- portabilité (interface normalisée)

Réalisation directe

≡ mémoire virtuelle :

- indirection/adresses « interprétées »
- va et vient étendu à l'espace mémoire des sites distants



Mise en œuvre de la mémoire virtuelle partagée répartie

Points critiques

- gestion de l'espace libre (glanage de cellules)
- localisation/liaison des données
- stratégies de répartition/(dé)placement des données :
  - recours au principe de localité
  - statique ou dynamique
  - partition ou duplication → gestion des conflits/de la cohérence

Limites à la transparence

- performances (imprévisibles, inférieures à une (théorique) solution ad-hoc optimale)
- la gestion efficace de la cohérence des données dupliquées impose des compromis sur la sémantique des accès aux données.

Modèles

- Espace de tuples (base de données (ensemble de tuples) partagée)
  - Opérations : déposer, retirer, rechercher un motif
  - Exemples : Linda, JavaSpaces, JSDT (Oracle)
- Objets (persistants) répartis partagés
  Guide, SOR, WebObjects, Globe, Legion...

Plan

Introduction : modèles de cohérence

Situation

Mise en œuvre d'un espace virtuellement partagé pour un ensemble de processus répartis.
→ interaction par accès (lecture/écriture) à des données vues comme centralisées. (fichiers, mémoire virtuelle répartie...)

Les données partagées, vues par le programmeur comme uniques, sont en fait souvent *dupliquées* pour des raisons de

- *disponibilité* : permettre l'accès aux données même en cas de défaillance/perte d'une copie.
- *efficacité* : placer les données sur leur site d'utilisation permet des accès plus rapides et réduit les échanges entre sites
  Exemple : caches

Cohérence : la contrepartie de la duplication

La *duplication* devrait rester *transparente* pour le programmeur : idéalement, les copies d'une même donnée doivent se comporter comme une copie unique, être *cohérentes*

→ la mise à jour d'*une* copie doit affecter *l'ensemble* des copies

Problème : coût d'un maintien « strict » de l'identité entre copies

- *en temps* : coordination des mises à jour des différentes copies d'une même donnée
- *en volume* : propagation/diffusion des mises à jour vers les différentes copies d'une même donnée

→ *arbitrage* nécessaire entre le *coût* et la *qualité* de la cohérence.

Idée

Relâcher la contrainte sur l'égalité des copies, en fonction

- des besoins de l'application
- des schémas d'accès/d'utilisation des données dupliquées

→

Modèles de cohérence

Contrat entre le client et le service de données répliquées :

- Si le client utilise le service d'accès aux données dupliquées selon des règles/un protocole donné
- Alors le service garantit des propriétés sur les copies ramenant directement ou indirectement à une égalité des copies

Mise en œuvre

Restrictions sur les instants/valeurs de retour des *lectures* : moins les restrictions sont fortes, moins le protocole est coûteux.

Protocoles de cohérence : typologie

- Cohérence *continue* : les critères de cohérence sont assurés/vérifiés à chaque instant, pour toutes les données.
  - cohérence stricte,
  - cohérence séquentielle
  - linéarisabilité
  - cohérence causale
  - cohérence FIFO
- Cohérence *ponctuelle* (*faible/weak consistency*) : les critères de cohérence sont assurés/vérifiés localement dans l'espace et dans le temps.
  - cohérence à la sortie (release consistency)
  - cohérence à l'entrée (entry consistency)
- Cohérence *à terme* (*eventual consistency*) : les copies finissent par converger en l'absence de nouvelles mises à jour.
  - réplication optimiste
  - cohérence centrée sur les clients

Hypothèses communes aux différents modèles de cohérence

1. *pas de conflit* d'accès aux copies : chaque client a sa copie
2. opérations : *lectures* et *écritures*
3. les *écritures* peuvent être *concurrentes* entre elles, et/ou avec les lectures.
4. Chaque écriture est *propagée* vers les autres copies.

→ les critères de cohérence sont des critères *globaux*, liant l'ensemble des opérations sur la donnée dupliquée (ou les valeurs de l'ensemble des copies)

→ les opérations ont une *durée non nulle*. On distingue au moins :

- *l'appel* de l'opération sur le site du client.
- *le retour* de l'opération, dont l'instant et le résultat sont déterminés par le protocole de cohérence

Plan

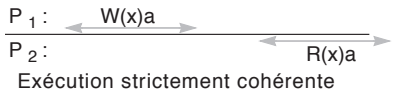
Cohérence stricte

Toute lecture sur une (copie d'une) donnée x renvoie une valeur correspondant à l'écriture la plus récente sur x.

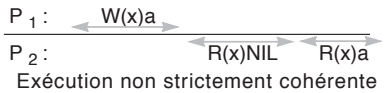
Modèle le plus fort, correspondant à une vue « idéale » des données.

Notations

- $R_i(x)a$: lecture de x par p_i, renvoyant a
- $W_i(x)a$: écriture par p_i de la valeur a dans x



Exécution strictement cohérente



Exécution non strictement cohérente

Mise en œuvre

Principe

La réalisation de la cohérence stricte nécessite de :

- construire un ordre global total sur les écritures (qui soit compatible avec le temps observé)
- rendre visible instantanément chaque écriture à chaque processus (réalisable en ordonnant aussi les lectures par rapport aux écritures)

Mise en œuvre

Protocole suivi par le processus P_i

Possible : *diffusions atomiques* pour construire l'ordre total global

- appel local à $R_i(x)$ ou $W_i(x)a$
  - diffusion_atomique de l'opération à l'ensemble des sites ;
  - recevoir ($\rightarrow$ attendre) la requête émise ;
  - traiter la requête (retourner la valeur de la copie locale ou réaliser l'écriture sur la copie locale)
  - terminer l'appel local
- réception d'un message diffusion_atomique($W_j(x)a$)
  réaliser l'écriture sur la copie locale
- réception d'un message diffusion_atomique($R_j(x)$)
  (ne rien faire)

. . . mais trop coûteux en pratique.
$\rightarrow$ utilisation de modèles approchés, plus sobres

Cohérence séquentielle

Condition de cohérence séquentielle

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

- Toutes les opérations sur les données (vues comme centralisées) sont exécutées selon une certaine *séquence* S
- Les opérations exécutées par tout processus P figurent dans le *même ordre* dans S et dans P
- La *cohérence interne* des données est respectée dans S : chaque lecture doit renvoyer la valeur de *l'écriture immédiatement précédente dans* S.

- Analogue à la sérialisabilité des transactions (mais grain plus fin : sérialisation des lectures/écritures et non des transactions) : il *suffit* que la séquence S *puisse* être construite
- Définit une condition globale à vérifier : compatibilité des histoires des lectures des différents sites
  → coûteux : pas de décision locale sans interaction

Mise en œuvre : lectures immédiates, écritures synchronisées

Protocole suivi par P_i

- appel local à $R_i(x)$: retourner la valeur de la copie locale
- appel local à $W_i(x)a$
  diffusion_atomique de $W_i(x)a$ à l'ensemble des sites ;
- réception d'un message diffusion_atomique($W_j(x)a$)
  /* par le protocole de cohérence */
  - réaliser l'écriture sur la copie locale ;
  - **si** j = i **alors** terminer l'appel à $W_i(x)a$;
    /* garantit l'ordre d'exécution global de $W_i(x)a$ */

- la diffusion atomique des écritures permet de garantir la compatibilité des histoires des sites avec S
- les lectures entre 2 écritures commutent
  → les lectures locales suffisent à assurer la compatibilité.
- lectures immédiates, écritures retardées
- la version suivante réalise la politique duale (écritures favorisées)

Mise en œuvre : lectures inhibées, écritures immédiates

Protocole suivi par P_i

Principe

- pour lire, attendre que la dernière écriture locale devienne la dernière écriture globale
- compter (*cpt*) les écritures locales qui auraient été anticipées.

- appel local à $R_i(x)$
  si cpt = 0 **alors** retourner la valeur de la copie locale ;
  sinon attendre
- appel local à $W_i(x)a$
  - cpt++ ;
  - diffusion_atomique de $W_i(x)a$ à l'ensemble des sites ;
  - terminer l'appel à $W_i(x)a$;
- réception d'un message diffusion_atomique($W_j(x)a$)
  /* par le protocole de cohérence */
  - réaliser l'écriture sur la copie locale ;
  - **si** j = i **alors**
    cpt- - ;
    si cpt = 0 **alors** retourner la valeur a à $R_i(x)$ s'il est en attente

Introduction d'une référence temporelle : linéarisabilité

Notation : $op_1 \rightarrow op_2 \triangleq t(\text{fin}(op_1)) < t(\text{début}(op_2))$,
t date vérifiant la validité forte (p. ex. horloges vectorielles)

Condition de linéarisabilité

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

- Toutes les opérations sur les données sont exécutées selon une certaine *séquence* globale S
- La *cohérence interne* des données est respectée dans S
- Si deux opérations op_1 et op_2 (lectures ou écritures) sont telles que $op_1 \rightarrow op_2$, *alors* op_1 et op_2 figurent *dans cet ordre dans* S

Différence par rapport à la cohérence séquentielle :
ajout d'une contrainte sur l'ordre des opérations non concurrentes
→ plus fort/réalisation plus coûteuse que la cohérence séquentielle

Linéarisabilité et cohérence séquentielle

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

linéarisable :

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

séquentiel, mais non linéarisable (pas d'ordre série commençant par W(x)b) :

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

P_1 :	W(x)a			
P_2 :		W(x)b		
P_3 :			R(x)a	R(x)b
P_4 :			R(x)b	R(x)a

non séquentiel

Cohérence causale

Causalité dans un contexte de variables partagées

Soient e_1 et e_2 deux événements,

$e_1 \rightarrow e_2$ (précédence causale) *lorsque* :

- e_1 et e_2 sont des opérations d'un même processus, et e_1 survient avant e_2 (*causalité interne* aux processus)
- ou e_1 est l'écriture d'une variable x et e_2 une lecture ultérieure de x (*causalité entre écritures et lectures*)
- il existe un évt. e_3 tel que $e_1 \rightarrow e_3$ et $e_3 \rightarrow e_2$ (*transitivité*)

Condition de cohérence causale

Si $e_1 \rightarrow e_2$, alors
tout processus qui observe e_1 *et* e_2 doit observer e_1 avant e_2

Remarque

Comme toute lecture renvoie le résultat d'une écriture précédente, la condition de cohérence causale entre lectures et écritures, est systématiquement assurée.

La condition de cohérence causale peut donc être reformulée en portant seulement sur les écritures :

Tout processus qui voit les (résultats d') écritures causalement liées doit les voir dans l'ordre causal de ces écritures.

Note : des écritures causalement indépendantes (||) peuvent être vues dans un ordre différent

Exemples

P_1 : W(x)a	//			
P_2 :		W(x)b		
P_3 :			R(x)a	R(x)b
P_4 :			R(x)b	R(x)a

non séquentiel, mais ***causalement cohérent***

P_1 : W(x)a				
P_2 :		R(x)a	W(x)b	
P_3 :			R(x)a	R(x)b
P_4 :			R(x)b	R(x)a

non causalement cohérent

Remarque

Modèle plus faible que la cohérence séquentielle, car on ne considère que des événements reliés par une relation de causalité

Mise en œuvre

La dépendance causale peut être évaluée avec des horloges vectorielles. Celles-ci peuvent donc servir à réaliser un protocole de cohérence causale (cf Annexe A)

Cohérence FIFO (ou PRAM (Pipelined RAM))

Condition de cohérence FIFO

Des écritures réalisées par un même processus doivent être vues par tous les processus dans leur ordre de réalisation.

Note : les 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)a	R(x)b	R(x)c	
P_4 :				R(x)b	R(x)a	R(x)c	

non causalement cohérent, mais
cohérence FIFO (R(b) précède R(c) partout ; R(a) est indépendant)

Remarques

- Modèle plus faible que la cohérence causale, car on ne considère que la causalité locale à un processus, non entre processus différents.
- Mise en œuvre
  un compteur (entier) par processus, pour estampiller les écritures

Plan

Cohérence ponctuelle

Situation

Les protocoles de cohérence continue spécifient une relation *invariante* liant *l'ensemble* des valeurs successives de *l'ensemble* des copies

Même la cohérence FIFO impose que toutes les mises à jour soient toujours propagées (dans l'ordre) à l'ensemble des copies

Idée

Il n'est *pas toujours* nécessaire d'assurer *la cohérence en permanence* : tous les sites n'ont pas nécessairement besoin de voir toutes les mises à jour de toutes les données, dans un ordre précis.

Exemple

Si un processus modifie les données à l'intérieur d'une section critique *SC*, les états intermédiaires ne seront pas vus par les autres processus.

→ inutile de gérer la cohérence sur les opérations internes à *SC*

→ fournir des « objets de synchronisation » de bas niveau, afin que le *programmeur définisse lui-même les moments où garantir la cohérence*

Cohérence faible

Principe

Définir des *points de synchronisation*, où la cohérence des données est assurée

Définition d'un point de synchronisation

appel d'une opération *S.synchronize()* sur une *variable de synchronisation S*

→ mise en cohérence des copies locales des données du site appelant

- mise à jour des copies locales avec la dernière valeur écrite
- propagation des écritures faites par le site

Remarques

- entre 2 appels à *S.synchronize()* la cohérence n'a pas à être assurée
- propager $\neq$ mettre immédiatement en cohérence

Cohérence faible
Définition (propriétés assurées par le protocole)

- L'accès aux variables de synchronisation est séquentiellement cohérent : *tous les sites voient les opérations de synchronisation dans le même ordre*
- Une opération de synchronisation ne peut se terminer que quand les écritures antérieurement en cours sur l'ensemble des sites sont achevées : *la synchronisation force/attend la fin des mises à jour en cours de toutes les copies locales du site qui effectue l'opération de synchronisation.*
- Un site ne peut lire ou écrire tant que ses opérations de synchronisation antérieures ne sont pas terminées : *après synchronisation, les accès d'un site portent nécessairement sur la version la plus récente des données, au moment de la synchronisation*

Exemple (***S*** : opération de synchronisation)

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

Cohérence faible
Modèles fins

L'appel *S.synchronize()* sur une variable de synchronisation *S* permet de demander
- la mise à jour des copies locales
- la propagation des modifications effectuées sur les copies locales

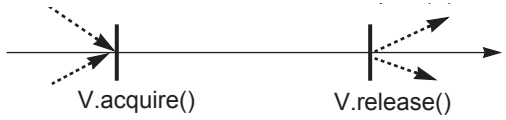
Ces opérations ne sont pas forcément liées :
- la première est utile à l'entrée d'une section critique,
- la seconde en sortie de section critique.

Des modèles plus fins distinguent ces deux opérations, afin d'alléger le protocole :
- *cohérence à la libération* (*release consistency*) : Treadmarks (Rice)
- et *cohérence à l'entrée* (*entry consistency*) : Midway (CMU)

Cohérence à la libération

Cohérence gérée lors des opérations *acquire()*/*release()* sur un verrou

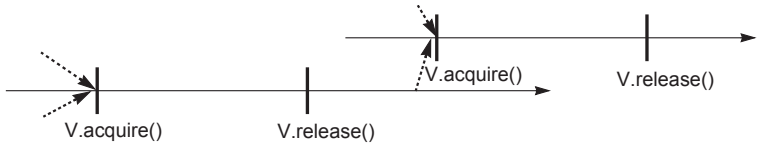
- À l'appel de *acquire()* : toutes les copies locales de l'ensemble de données protégé par le (associé au) verrou sont mises à jour pour tenir compte des écritures non encore répercutées
- À l'appel de *release()* : toutes les valeurs des variables locales qui ont été modifiées sont envoyées vers les copies distantes.
- cohérence FIFO sur les appels à *V.acquire()* et *V.release*()
  → exécution séquentiellement cohérente
  [V verrou ⇒ordre total sur les *V.acquire()*]



Variante : cohérence paresseuse à la libération

(lazy release consistency)
Lors du *release()*, les modifications ne sont pas propagées.
Elles sont conservées, pour être transmises à la demande, aux sites qui appelleront *acquire()* par la suite.

Intéressant dans le cas d'accès répétés à une même donnée.



Cohérence à l'entrée (entry consistency)

Cohérence paresseuse à la sortie,
où chaque variable doit être associée à un verrou spécifique.

À l'appel de *acquire()*, seules les variables associées au verrou utilisé sont mises à jour

P 1 : V(x).acq() W(x)a V(y).acq() W(y)b V(x).rel() V(y).rel()	
P 2 :	V(x).acq() R(x)a R(y)NIL
P 3 :	V(y).acq() R(y)b

Plan

Protocoles optimistes de cohérence

Limites des protocoles pessimistes (cohérence continue ou ponctuelle)

- capacité de croissance et d'évolution réduite (diffusion fiable)
- mise en œuvre de la cohérence basée sur l'attente
  - handicap pour le traitement des pannes (cf impossibilités)
  - inadapté à la connectivité intermittente, à la mobilité
  - inadapté au travail coopératif (cf git, édition partagée. . .)

→ approches optimistes

- évolution libre des différentes copies, sans blocage
- garantie de cohérence *à terme*
  → détecter les conflits et gérer la convergence des copies
- gestion de la propagation adaptée aux systèmes ouverts à large échelle (p. ex. propagation épidémique)
- transparence pour l'utilisateur

Cohérence à terme

Condition de cohérence à terme

En l'absence de nouvelle opérations sur les données, les différentes copies finissent par converger vers la même valeur finale.

Stratégies

- détecter les conflits (mises-à-jour causalement indépendantes), puis résoudre les divergences
  point dur : résolution difficile à automatiser
- limiter/contrôler la divergence entre copies
  - limiter le nombre/l'âge des mises-à-jour indépendantes
  - rafraîchir les copies à partir de modèles de comportement probabilistes

Approches

- *syntaxique* : datation globale des écritures.
- *sémantique* : utiliser en outre les possibilités de commutation entre opérations sur les données.

Sémantiques de session : protocoles de cohérence centrés sur le client

Contrainte : respecter l'ordre causal apparent pour l'utilisateur

(Contre) exemple

L'utilisateur d'une application mobile peut utiliser et modifier successivement plusieurs copies différentes d'une donnée :

- modification d'une copie,
- déconnexion, mobilité, reconnexion...
- ...et travail sur une copie antérieure

→ protocoles de cohérence centrés sur l'utilisateur,
contrôlant les opérations pour garantir des propriétés de « cohérence » moins gourmandes que la causalité, et choisies par l'utilisateur.

Modèles de cohérence centrés sur le client

Contexte d'utilisation

- La plupart des accès sont des lectures
- Les conflits d'écriture sont très rares
  (par exemple : chaque donnée possède une copie « maître », gérée par un site unique, et la copie maitre est la seule modifiée)
- Il est acceptable de lire une donnée périmée

Exemples le DNS, le WWW, systèmes de fichiers en nuage...

Modèle d'utilisation : notion de session

session $\triangleq$ séquence des opérations (lectures/écritures) effectuées par un *même* utilisateur (client), sur des copies ou lieux (mobilité) distincts

Modèles de cohérence centrés sur le client
Mise en œuvre

Principe

- une session par utilisateur, virtuellement attachée à l'utilisateur
- chaque *écriture* a un *identifiant de version global* et une date (*estampille*)
- chaque session conserve le journal des identifiants de versions lues et écrites par l'utilisateur
- chaque site journalise les écritures locales, avec leur id+date

Notations

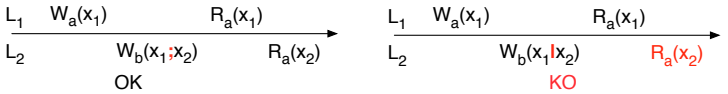
- L_s site (lieu/copie)
- x_i valeur/version de x sur une copie locale
- $R_a(x_i)$ lecture de la version x_i par le client a
- $W_a(x_i)$ écriture *indépendante* de la version x_i par le client a
- $W_a(x_i ; x_j)$ écriture de la version x_j par le client a, tenant compte de (*dépendante* de) la version x_i
- $W_a(x_i | x_j)$ écriture de la version x_j par le client a *concurremment* à l'écriture de x_i par un autre client

Lectures monotones

Condition pour les lectures monotones

Si un client a lu une version d'une donnée x, toute lecture ultérieure par ce même client doit rendre une version postérieure ou égale.

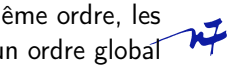
Ce modèle garantit qu'un client ne « reviendra pas en arrière »



$R_a(x_2)$ survenant après $R_a(x_1)$, x_2 doit être postérieure à x_1

Mise en œuvre

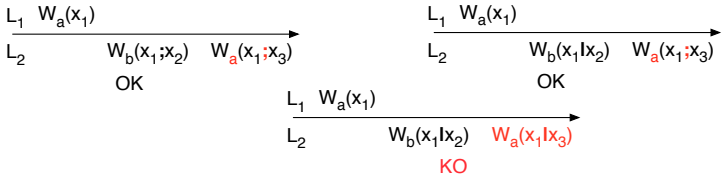
- les identifiants de versions lues par le client (journal) permettent de déterminer les écritures nécessaires sur la copie locale
- récupérer les versions, en utilisant les identifiants de version
- mettre à jour le journal de lectures du client et la version locale.
- afin que les écritures se fassent partout dans le même ordre, les id. de version (ou les estampilles) doivent définir un ordre global



Écritures monotones

Condition pour les écritures monotones

Si un client exécute 2 écritures successives sur une même donnée, la deuxième écriture ne peut être réalisée que quand la copie qu'elle modifie est à jour par rapport à la première écriture.



≈ cohérence FIFO réduite au client lui-même

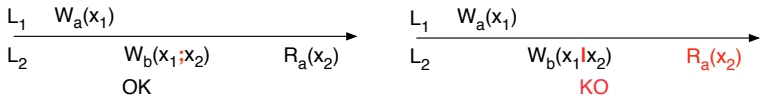
Mise en œuvre

- l'ensemble d'écritures du journal du client permet de déterminer et récupérer les écritures à réaliser sur la copie locale
- mettre à jour l'ensemble d'écritures du client

Cohérence écriture-lecture (Read Your Writes)

Condition de cohérence écriture-lecture

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

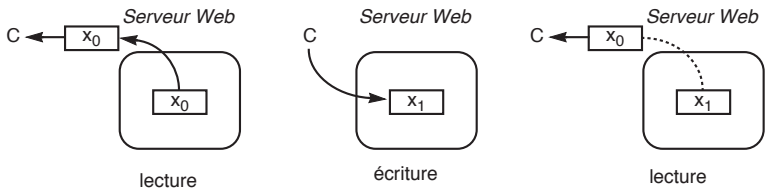


Mise en œuvre

- l'ensemble d'écritures du journal du client permet de déterminer et récupérer les écritures à réaliser sur la copie locale (avant une lecture)
- mettre à jour l'ensemble de lectures du client.

Cohérence écriture-lecture (Read Your Writes)

Exemple de violation



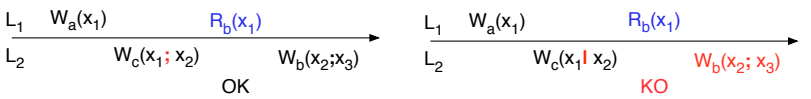
Cause de l'incohérence

Le cache du navigateur ne sert que pour la lecture, pas pour l'écriture.
En pratique, on rafraîchit explicitement le cache

Cohérence lecture-écriture (Writes Follow Reads)

Condition de cohérence lecture-écriture

Si un client C écrit une donnée après l'avoir lue, aucun client ne peut lire la version écrite par C avant la version lue par C.
Autrement dit : la version écrite par C doit dépendre de (être postérieure ou égale à) la version lue par C.

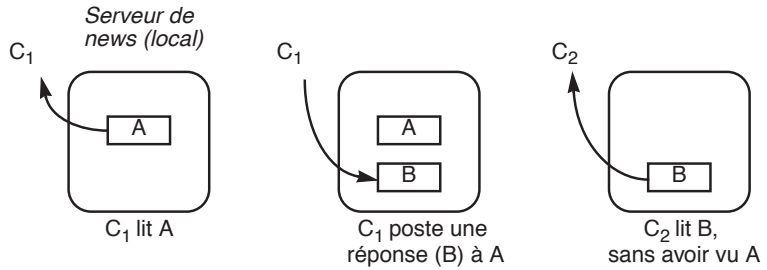


Mise en œuvre

- l'ensemble de lectures du journal du client permet de déterminer et récupérer les écritures à réaliser sur la copie locale
- lors de l'écriture, ajouter au journal d'écritures du client les écritures effectuées à partir de son journal de lectures

Cohérence lecture-écriture (Writes Follow Reads)
Exemple de violation

Donnée partagée = groupe de news



A doit avoir été écrit sur toutes les copies lues avant que B ne le soit.

Plan

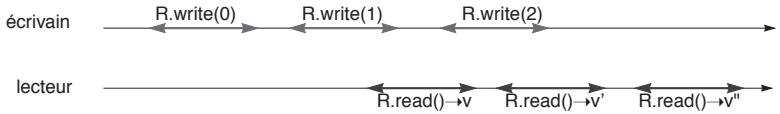
Cohérence dans les systèmes asynchrones avec défaillances

Registres

- objet concurrent élémentaire R, partagé par des processus séquentiels
- opérations :
  - écriture : *R.write(valeur)*
  - lecture : *R.read()→valeur*
- les opérations ont une *durée*
  l'exécution d'une opération *op* par un processus se traduit par l'occurence de deux événements associés et successifs
  - l'appel de l'opération, noté *inv[R.op(arg)]*
  - le retour de l'opération, noté *resp[R.op(res)]*
- le *type* du registre définit le modèle de cohérence

Registres réguliers

- un écrivain, plusieurs lecteurs (1WMR)
  - pas de conflit en écriture
  - écritures séquentielles
- résultat d'une lecture
  - non concurrente avec une écriture
    → valeur courante du registre (dernière valeur écrite)
  - concurrente avec une/des écritures
    → valeur courante ou valeur d'une des écritures concurrentes



v ∈ {0,1,2} / v' ∈ {1,2} / v" = 2

Remarque

inversion possible de valeurs (v=2, v'=1)
→ un *registre régulier* n'a *pas de spécification séquentielle* :
il existe des exécutions possibles non séquentielles

Registres atomiques

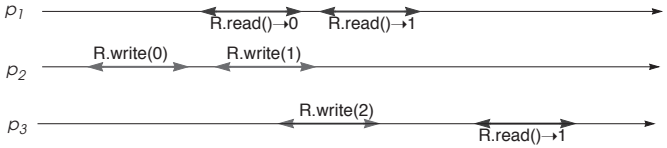
Plusieurs écrivains, plusieurs lecteurs (MWMR)

Propriété : accès *linéarisables*

Pour tout ensemble d'accès concurrents au registre,
il existe une exécution séquentielle S

- donnant les mêmes résultats
- respectant la chronologie des opérations non concurrentes
- *légale* :
  toute lecture fournit la valeur de l'écriture immédiatement précédente

Exemple



S = R.write(0) ;R.read()→0 ; R.write(2) ; R.write(1) ;R.read$_1$()→1 ; R.read$_3$()→1

Importance de l'atomicité

Abstraction

les opérations peuvent être vues comme ponctuelles/instantanées, sans chevauchement
→ possibilité de raisonner sur les *entrelacements* d'opérations, et non sur les instants d'exécution dans le temps réel

Modularité

- l'atomicité est une propriété locale à chaque registre
  → l'atomicité peut être implantée indépendamment pour chaque registre
  → la composition de registres atomiques est atomique

Mise en œuvre de registres dans un système asynchrone avec défaillances

Modèle et notations

- $\Pi = \{p_1, p_2, \ldots, p_n\}$ ensemble de n sites (processus séquentiels)
- les processus répètent des *pas atomiques*.
  pas $\triangleq$ calcul local suivi d'un envoi ou d'une réception de message
- processus non synchronisés, mais vitesse non nulle
- pannes franches (panne = arrêt, sans reprise)
- réseau maillé, fiable, asynchrone
- communication point à point, envoi non bloquant, réception bloquante
  diffusion = ensemble d'envois indépendants, vers chaque site de Π

Notation

$AS_{n,t}[P]$: système asynchrone

- comportant n processus,
- dont au plus t sont défaillants,
- utilisant des registres,
- et dont les exécutions vérifient le prédicat P.

Réalisation de registres atomiques

Spécification

- *Sûreté* : les n processus exécutent un protocole réalisant le registre atomique (tel que toute exécution est linéarisable)
- *Vivacité* : tout appel à *R.write(–)*/*R.read(–)* effectué par un processus non défaillant se termine

Démarche

- réalisation d'un registre régulier
- réalisation d'un registre atomique 1WMR à partir du registre régulier
- réalisation d'un registre atomique MWMR à partir du registre 1WMR

Réalisation d'un registre régulier dans $AS_{n,t}$[t<n/2]

Principe

- les versions écrites par le rédacteur sont identifiées, et ordonnées
- les requêtes de lecture sont également identifiées et ordonnées
- chaque site p_i conserve
  - la dernière version lue : cur_val_i
  - l'identifiant de la dernière version lue : w_sn_i
  - l'identifiant de la dernière requête de lecture émise : req_sn_i

Lecture
- diffusion de la requête
- attente d'une majorité de réponses
- retour de la version maximum

Écriture
- diffusion de la requête
- attente dune majorité d'acquittements

Réception d'une requête d'écriture
- mise à jour de la version
- envoi acquittement

Réception d'une requête de lecture req_sn_k par le site p_i
- envoi de $< cur\_val_i, w\_sn_i, req\_sn_k >$

Algorithme (Source M. Raynal, op. cit)

Ecriture du registre R [exécutée par le seul (et unique) rédacteur p_w]

```
R.write(v) {
  w_sn_w++;
  diffuser("ecriture(v, w_sn_w)");
  attendre la réception de "acq_ecriture(w_sn_w)" d'une majorité de processus;
}
```

Lecture du registre R par un processus p_i ($1 \le i \le N$)

```
R.read() {
 req_sn_i++;
 diffuser("req_lect(req_sn_i)";
 attendre la réception de "acq_req_lect(req_sn_i,–,–)" d'une majorité de processus;
 max_sn := max({s/"acq_req_lect(req_sn_i,s,–)" a été reçu};
 retourner val tel que "acq_req_lect(req_sn_i,max_sn,val)" a été reçu
}
```

Traitement sur réception par un processus p_i ($1 \le i \le N$) de "ecriture(val, w_sn)" du processus p_w

```
{ si w_sn ≥ w_sn_i alors cur_val_i := val; w_sn_i :=  w_sn finsi;
  envoyer "acq_ecriture(w_sn)" à p_w;
}
```

Traitement sur réception par un processus p_i ($1 \le i \le N$) de "req_lect(req_sn)" du processus p_j ($1 \le j \le N$)

```
{ envoyer "acq_req_lect(req_sn, w_sn_i, cur_val_i)" à p_j;}
```

Preuve (arguments)

Vivacité

Si une majorité reste correcte,
toute attente par un processus correct se finira (réseau fiable)

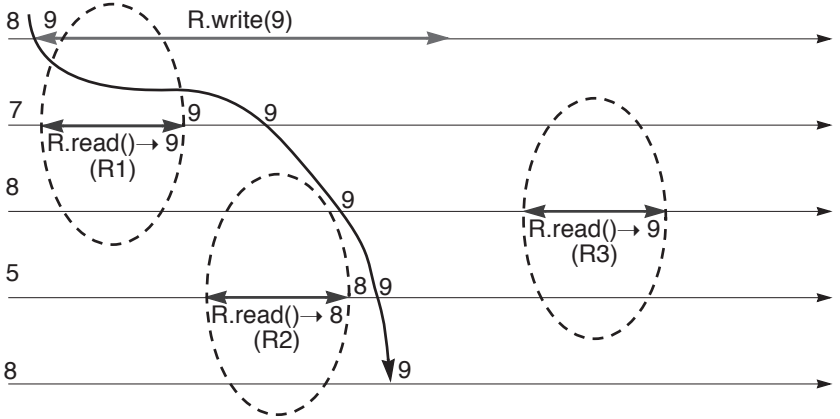
Sûreté

La plus grande version reçue correspond à une écriture concurrente, ou à la version en début de lecture

- Au départ de la lecture,
  la version courante est connue d'une majorité de processus,
- Les versions évoluent en croissant.
  - → dans les réponses reçues, il y aura donc au moins un site renvoyant une version supérieure (s'il y a eu des écritures démarrées depuis) ou égale.
  - → le max des réponses sera donc une version supérieure ou égale (S'il est supérieur, il s'agit d'une écriture concurrente avec la lecture)

Remarque

L'atomicité n'est pas garantie par ce protocole



R2 suit R1 chronologiquement,
mais fournit une valeur antérieure à celle de R1

Réalisation d'un registre atomique 1WMR dans $AS_{n,t}$[t<n/2]

Principe

aider à propager la plus grande version trouvée suite à une lecture

→ le lecteur (re)diffuse l'écriture de la plus grande version trouvée et attend une majorité d'acquittements

(→ ajouter l'identification des sites aux requêtes d'écriture)

Algorithme (Source M. Raynal, op. cit)

Lecture du registre R par un processus p_i (1≤ i≤ N)

```
R.read() {
 req_sn_i++;
 diffuser("req_lect(req_sn_i)";
 attendre la réception de "acq_req_lect(req_sn_i,–,–)" d'une majorité de processus;
 max_sn := max({s/"acq_req_lect(req_sn_i,s,–)" a été reçu};
 v := val tel que "acq_req_lect(req_sn_i,max_sn,val)" a été reçu
 diffuser("ecriture(v,max_sn)");
 attendre la réception de "acq_ecriture(max_sn)" d'une majorité de processus;
 retourner v;
}
```

Traitement sur réception par un proc. p_i (1≤ i≤ N) de "ecriture(val, w_sn)" du proc. p_j (1≤ j≤ N)

```
{ si w_sn ≥ w_sn_i alors cur_val_i := val; w_sn_i :=  w_sn finsi;
  envoyer "acq_ecriture(w_sn)" à p_j;
}
```

Preuve (arguments) (1/2)

Vivacité

- *Lectures* : le réseau étant fiable, si une majorité reste correcte, toute attente par un processus correct se finira
  (tant la collecte des versions que la réception des acquittements)
- *Écritures* : l'écriture ne progresse que si une majorité a répondu
  (« compléter » cette majorité n'affecte pas la progression des écritures)

Preuve (arguments) (2/2)

Sûreté

Trouver une séquence équivalente respectant l'ordre chronologique, et la légalité des lectures

- la version lue est postérieure ou égale au début de la lecture
  → une écriture qui (commence et) se termine entre la fin de l'écriture de la version lue et la fin de la lecture, n'induit aucune contrainte
  - de précédence chronologique, car elle serait nécessairement concurrente à la lecture en cours (postérieure à son début et antérieure à sa fin)
  - ou de légalité, puisque la possibilité que la lecture suive immédiatement l'écriture de la version lue est préservée, du fait de cette même concurrence
- lorsque la lecture se termine, la version lue est connue d'une majorité de sites
  → impossible qu'une lecture commençant ensuite retourne une version inférieure.

Réalisation d'un registre atomique général dans $AS_{n,t}$[t<n/2]
Principe

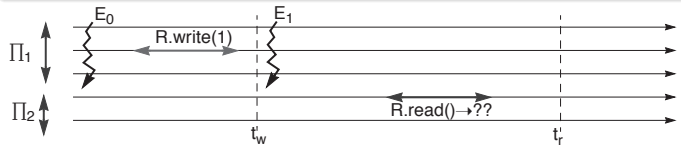
Garantir l'existence d'un ordre total sur les opérations d'écriture

→

- horloges logiques pour construire un ordre global sur les versions
- gestion du numéro de version selon le même schéma :
  - une écriture commence par une enquête auprès des sites de Π, pour obtenir le numéro de version courant de chaque site
  - le numéro de version est calculé (maximum) dès qu'une majorité de processus a répondu
  - ordre total, car l'identifiant de site est utilisé pour départager les numéros de version égaux
- fin des écritures similaire :
  Une écriture qui aurait calculé un identifiant de version trop petit sera acquittée, mais ignorée par les sites disposant d'un identifiant de version supérieur : tout se passe alors comme si cette écriture avait été déjà effectuée, puis son résultat écrasé par la version courante.

Résultat d'impossibilité

Il n'existe pas d'algorithme réalisant un registre atomique dans $AS_{n,t}[t \geq n/2]$



- on suppose qu'un tel algorithme A existe
- partition de Π en Π_1 et Π_2, t.q. $|\Pi_1| = \lceil n/2 \rceil$ et $|\Pi_2| = \lfloor n/2 \rfloor$
- Π_1 et Π_2 restent mutuellement isolés
  (échanges de messages retardés jusqu'à t_r)
- on considère deux exécutions E_0 et E_1
  - les sites de Π_2 ne font rien jusqu'à t_w
  - les sites de Π_1
    - pour E_0 : sont défaillants dès le départ → *R.read()* renvoie 0
    - pour E_1 : défaillent après *R.write(1)*, à t_w → *R.read()* renvoie 1
- pour l'ensemble des sites corrects (Π_2), E_0 et E_1 sont indistinguables. *R.read()* devrait donc retourner la même valeur dans les 2 exécutions, ce qui n'est pas le cas.

Plan

Annexe A : protocole de cohérence causale (Ladin et al. 92)

<rédaction réservée>

Systèmes de fichier répartis
Principes de conception

- Structure logique arborescente → Désignation par chemins
- Définition d'une abstraction "fichier" : classe d'objet
- Projection (mapping) de cette arborescence sur l'architecture réelle
- Contrôle de l'intégrité de cette projection (fsck)
- Protection, sécurité (listes d'accès, capacités)
- Contrôle des accès concurrents : problème sémantique
- Répartition → Transparence d'accès, de localisation...

Objectif

Un **bon** système de fichiers **répartis**
est un système de fichiers...
qui peut passer pour un système **centralisé**

Obtenir le meilleur niveau de transparence de la répartition

- transparence d'accès
- transparence de la localisation
- transparence du partage, de la concurrence
- transparence de la réplication
- transparence des fautes
- ...

Un premier pas : exemple ftp

- Pas de transparence (ni d'accès, ni de localité)
- Seule fonctionnalité : répliquer des fichiers sur différents sites
- Protocole client-serveur

Exemple

Machine cliente		Machine serveur mozart
ftp mozart	demande	acceptation des connexions
ftp> cd projet	requête	exécute : cd projet
ftp> get ff	requête	transfert du fichier ff vers client
ftp> quit	déconnexion	exit

Transparence de la localité

Une étape : le réseau est la "racine"

- Domain OS (Apollo) : //mozart/usr/...
- Cedar : /serveur/<chemin local>

Disparition des noms de sites : NFS

- Tous les fichiers sont intégrés dans une arborescence unique
- Implantation : par extension du montage de volume à distance
- Implantation : par exportation sélective des volumes

Transparence de la localité $\neq$ Désignation uniforme

Structure en couches d'un SGF

Modèle de Madnick-Alsop

Abstraction	Niveau	fonction
Répertoires	6	Désignation (chemins via répertoires)
	5	Localisation (des descripteurs)
Fichiers	4	Contrôle d'accès (listes, capacités)
	3	Accès au contenu (lire, écrire)
Fragments,blocs	2	Allocation et caches
	1	Gestion des entrées/sorties

Cas centralisé : descripteur de fichier (i-node)

Compteur de référence
Longueur du fichier (en octets)
Propriétaire créateur (groupe,id)
Date de créaction
Date de dernière modification
Date de dernière lecture
Type de fichier
Protection : indicateurs par exemple
Carte d'implantation

SGF centralisé

API Unix

canal open(nom,mode)	connexion du fichier à un canal
canal creat(nom, mode)	connexion avec création
close(canal)	déconnexion
int read(canal,tampon,n)	lecture de *n* octets au plus
int write(canal,tampon,n)	écriture de *n* octets au plus
pos = lseek(canal, depl, orig)	positionnement du curseur
unlink(nom)	suppression du nom dans le rép.
link(nom_orig,synonyme)	Nouvelle référence
state(nom, attributs)	Lecture des attributs du fichier

SGF réparti : Niveau répertoire

Fonction

- Passage de noms symboliques à des noms internes (UFID)
- Protection : contrôle d'accès

Génération de noms internes (UFID) :

- Engendrés par le serveur pour les clients
- Unicité, non réutilisation, protection

API RPC Service Répertoire

UFID Lookup(UFID rép, String nom)	retrouver le nom
AddName(UFID rép, String nom, UFID uid)	insérer le nom
UnName(UFID rép, String nom)	supprimer le nom
Delete(UFID uid)	supprimer le fichier
String[] GetNames(UFID rép, String motif)	recherche par motif

SGF réparti : Niveau fichier

Fonction

- Passage d'un nom interne global (UFID à un descripteur)
- Accès au fichier (attributs)

API RPC Service Fichier

byte[] Read(UFID uid, int pos,int n)	lire *n* octets au + en *pos*
Write(UFID uid,int pos,byte[] z,int n)	écrire *n* octets en *pos*
UFID Create()	créer un fichier
Delete(UFID uid)	supprimer l'UID du fichier
GetAttributes(UFID uid, Attributs a)	lire les attributs du fichier
Attributs SetAttributes(UFID uid)	mettre à jour les attributs

SGF réparti : Niveau bloc

Fonction

- Allocation d'espace
- Accès aux blocs physiques → gestion de caches

API Service Bloc

Bloc Allocate()	allouer dynamiquement un bloc
Desallocate(Bloc)	libérer le bloc
byte[] Read(Bloc)	lire le bloc
byte[] Write(Bloc, tampon)	écrire le tampon dans le bloc

→ Gestion des caches et sémantique d'accès

Problème sémantique du partage

Rappel

- Sémantique « centralisée » (Unix-like) :
  ⇒ lecture de la dernière version écrite
- Sous une autre forme : une lecture voit TOUTES les modifications faîtes par les écritures précédentes

Difficultés

- ⇒ un ordre total
- ⇒ Propagation immédiate des modifications
- Pas de copie ⇒ contention

⇒ Idée : copies « caches » et sémantique de session

L'approche « session »

Stratégie orientée « session »

- Un serveur unique maintient la version courante
- Lors de l'ouverture à distance par un client (début de session), une copie locale au client est créée
- Les lectures/écritures se font sur la copie locale
- Seul le client rédacteur perçoit ses écritures de façon immédiate (sémantique centralisée)
- Lors de la fermeture (fin de session), la « nouvelle version » du fichier est recopiée sur le serveur :
  ⇒ La « session » (d'écritures du client) est rendue visible aux autres clients

L'approche « session » (suite)

Problème

Plusieurs sessions de rédacteurs en parallèle...

- L'ordre des versions est l'ordre de traitement des fermetures
- La version « gagnante » est indéfinie

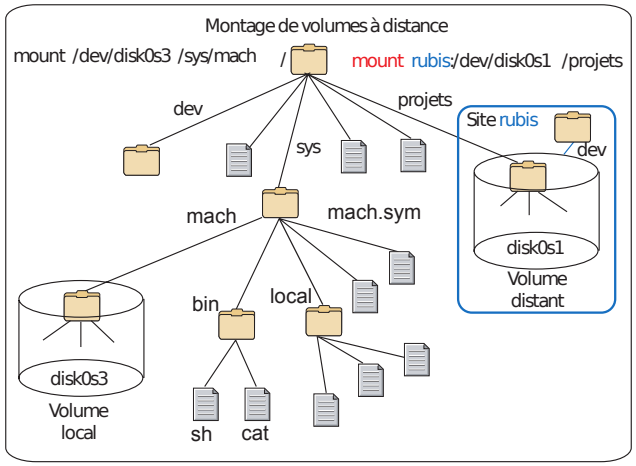
Autres solutions

- Invalidation les copies clientes par le serveur lors de chaque création de version
- Le contrôle du parallélisme : garantir un seul rédacteur
- Une autre idée : Fichiers à version unique (immutable)

Exemple NFS : Principes

- Transparence d'accès et de localisation (→ *virtual file system*)
- Traitement de l'hétérogénéité (→ *virtual node* vnode)
- Désignation non uniforme (a priori)
- Approche intégrée (modification des primitives du noyau)
- Protocole RPC entre **noyaux** (→ système fermé) : sémantique « au moins une fois »
- Serveur : site qui exporte des volumes (+ démons *mountd*, *nfsd*)
- Client : accès à un système de fichiers distants par montage à distance (extension du mount)
- Serveur sans état (mais dernières versions avec état)

Idée de base de NFS : le montage à distance de volumes



Exemple NFS
Transparence de localisation et hétérogénéité

La notion de *virtual file system* (VFS)

- Étend la notion de système de fichiers (file system) aux volumes à distance et à des systèmes de fichiers hétérogènes
- VFS → un volume support d'un système de fichiers particulier

La notion de *virtual node* (vnode)

- Extension de la notion de *inode*
- Pointe un descripteur local (*inode*) ou distant (*rnode*)

Notion de file handle ou UFID

Un nom global, engendré par le serveur et donné au client

Identification du FS	n° inode	n° de génération

Gestion de caches clients et serveur

Objectif

Garantir la sémantique centralisée :
la version lue est la dernière version écrite

Approximation. . .

- Sur ouverture, le client mémorise :
  (date de dernière modification, date présente)
- Interrogation du serveur si lecture après plus de p secondes ($p = 3$ sec. pour fichier ordinaire, $p = 10$ sec. si répertoire)

Exemple NFS : Interface serveur niveau répertoire

API RPC - Niveau répertoire

rename(dirfh, name, todirfh, toname)	renommer "name" par "toname" dans le répertoire "todirfh"
link(newdirfh,newname, dirfh,name)	nouvelle référence au fichier dans un nouveau répertoire
symlink(newdirfh,newname, string)	créer un lien symbolique vers le fichier nommé par la chaîne
string readlink(fh)	renvoyer le lien symbolique
dirfh mkdir(dirfh,name,attr)	créer un répertoire
rmdir(dirfh,name)	détruire un répertoire **vide**
entry[] readdir(dirfh, cookie,count)	lire des entrées du répertoire à partir de "cookie"
fsstats statfs(fh)	renvoyer des infos. sur le SGF

Exemple NFS : Interface serveur niveau fichier

API RPC - niveau fichier

fh lookup(dirfh,name)	renvoyer la référence au fichier "name"
fh create(dirfh,name,attr)	idem avec création
remove(dirfh,name)	supprimer le fichier du répertoire
attr getattr(fh)	lire les attributs du fichier
setattr(fh,attr)	mettre à jour les attributs du fichier
byte[] read(fh,pos,count)	lire "count" octets à partir de "pos"
write(fh,pos,count,byte[])	écrire "count" octets à partir de "pos"

Une nouvelle version de NFS avec serveur à état : v4

Principales différences...

- Introduction des notions de serveur à état et de session
  `OPEN` et `CLOSE` réapparaissent !
  procédures de reprise sur incident
- Protocole sécurisé beaucoup plus poussé : usage de kerberos, chiffrement
- Groupement des requêtes pour de meilleures performances
- Notion de délégation : le client a une copie locale utilisable jusqu'à une alerte du serveur (callback)
- Traitement de la transparence de migration et/ou réplication
- Verrouillage temporisé : notion de bail (lease)
- Usage de TCP/IP

Les processus serveurs démons acceptant les RPC

- `nfs` : lancement global du service
- `nfslock` : service de gestion de verrous
- `rpc.nfsd` : processus serveur nfs proprement-dit : lance un thread par requête client
- `rpc.idmapd` : service de nommage global → local :
  ⇒ utilisateur@domaine → (UID,GID)
- `prc.svcgssd` et `rpc.gssd` : processus gérant l'authentification par le protocole kerberos

Exemple de AFS (Andrew File System)

Principes

- SGF uniforme pour servir plusieurs milliers de machines
- Deux composants :
  - les stations serveurs (VICE)
  - les stations clientes (VERTUE)
- Réplication des fichiers systèmes
- Migration des fichiers utilisateurs (serveur $\leftrightarrow$ client)
- Gestion de caches disques sur les stations clientes
- Arborescence partagée commune /vice

Gestion de caches de fichiers

Traitement d'un accès à un fichier /vice/...

$\Rightarrow$ Approche orientée « session »

Lors de l'ouverture du fichier

- si fichier déjà présent en cache local, ouverture locale
- si fichier absent $\rightarrow$ contacter un serveur

Lors de la fermeture du fichier

- recopie de la version locale sur le serveur
- le serveur avertit les autres stations clientes qui contiennent ce fichier dans leur cache d'invalider leur version (call-back)

Gestion de caches de fichiers

Avantages et inconvénients

Avantages

- Minimise la charge d'un serveur : ceux-ci ne gèrent que les fichiers partagés, or 80% sont privés et temporaires
- L'invalidation par rappel est efficace car les cas de partage parallèle sont rares
- Les transferts de fichier dans leur totalité sont efficaces

Inconvénients

- Cohérence des copies d'un même fichier non garantie
- Sémantique « copie lue $\equiv$ dernière version écrite » non garantie