Systèmes et algorithmes répartis

Données réparties

Philippe Queinnec, Philippe Mauran

ENSEEIHT
Département Sciences du Numérique

24 octobre 2024

Plan

Contenu de cette partie

Toujours plus d'abstraction...

construction d'un environnement virtuellement centralisé

- Mémoire globale
  - principe
  - cohérence de copies multiples
- Système de fichier répartis
- Temps global : synchroniseurs et temps virtuel global
- Gestion des pannes partielles

Plan

Mémoire partagée virtuelle répartie

Motivation

Replacer le développeur d'applications (concurrentes)
dans les conditions d'un environnement centralisé :
$\rightarrow$ communication et synchronisation par variables/objets partagés

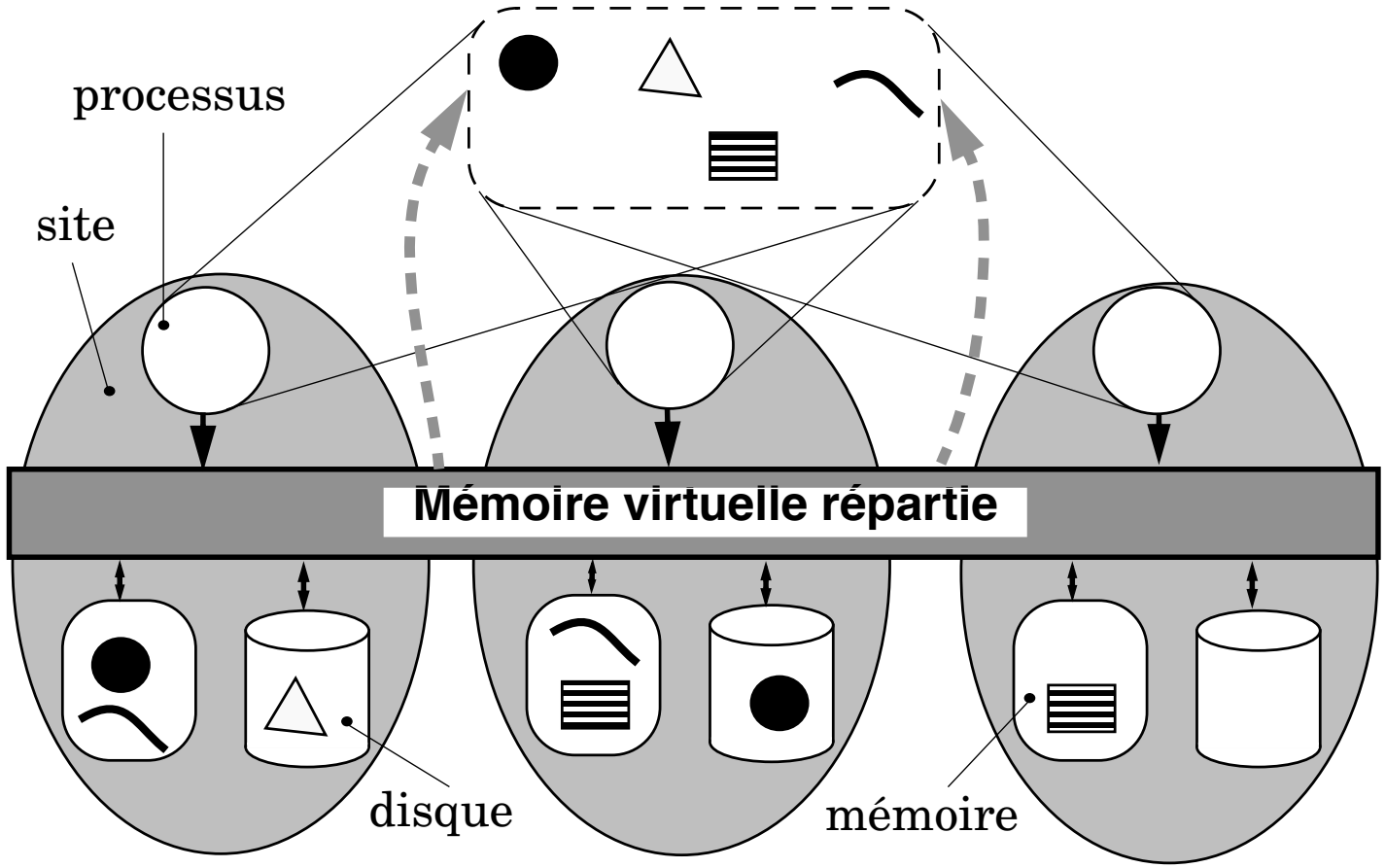
Avantages attendus pour le programmeur

- 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



Introduction : modèles de cohérence

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 répliquées 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 réplication 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.

Protocoles de cohérence : typologie

Relâcher la contrainte sur l'égalité des copies :

- 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 ou 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

1. Pas de conflit d'accès aux copies : chaque client a sa copie.
2. Opérations : lecture et écriture.
3. Les opérations peuvent être concurrentes entre elles (écritures 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 :

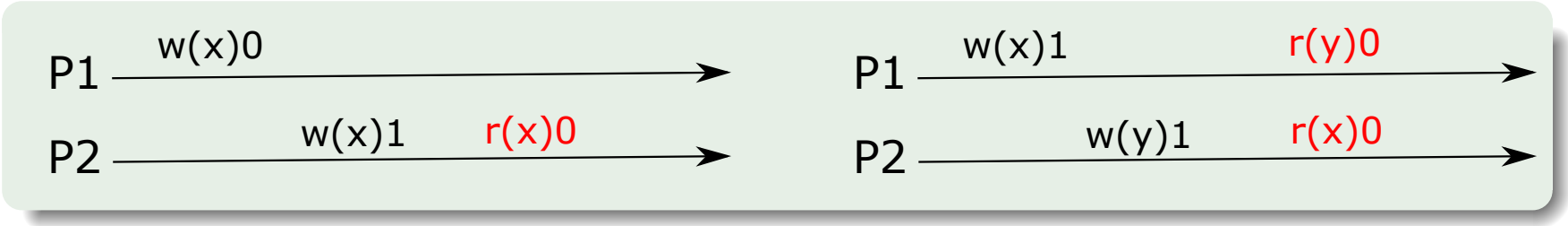
- l'appel de l'opération sur le site du client,
- le retour de l'opération, dont l'instant et le résultat sont déterminés par le protocole de cohérence.

Réplication et répartition des données

Deux sujets :

- Une même donnée, répliquée sur plusieurs sites ⇒ mêmes valeurs ?
- Plusieurs données, sur des sites différents ⇒ l'ensemble est-il cohérent ?

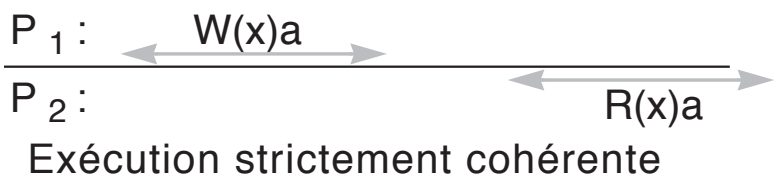
Mais en fait, c'est le même problème !



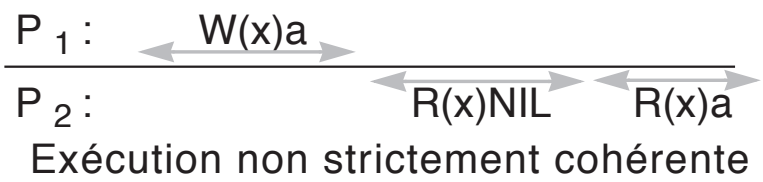
$r_i(x)a$ = lecture, sur le site i, de la variable x, renvoyant la valeur a
$w_i(x)b$ = écriture, sur le site i, de la variable x avec la valeur b

Cohérence stricte

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



Exécution strictement cohérente



Exécution non strictement cohérente

La réalisation de la cohérence stricte nécessite de :

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

Cohérence stricte : mise en œuvre

→ Diffusions atomiques pour construire l'ordre total global.

Protocole suivi par le processus P_i

- Appel local à $R_i(x)$ ou $W_i(x)a$
  1. diffusion atomique de l'opération à l'ensemble des sites
  2. recevoir (→ attendre) la requête émise
  3. traiter la requête (retourner la valeur de la copie locale ou réaliser l'écriture sur la copie locale)
  4. terminer l'appel local
- Réception d'un message $W_j(x)a$
  1. réaliser l'écriture sur la copie locale
- Réception d'un message $R_j(x)$
  1. retourner la valeur si $i = j$, ne rien faire si $i \neq j$

Trop coûteux en pratique → utilisation de modèles approchés, plus sobres

Cohérence séquentielle

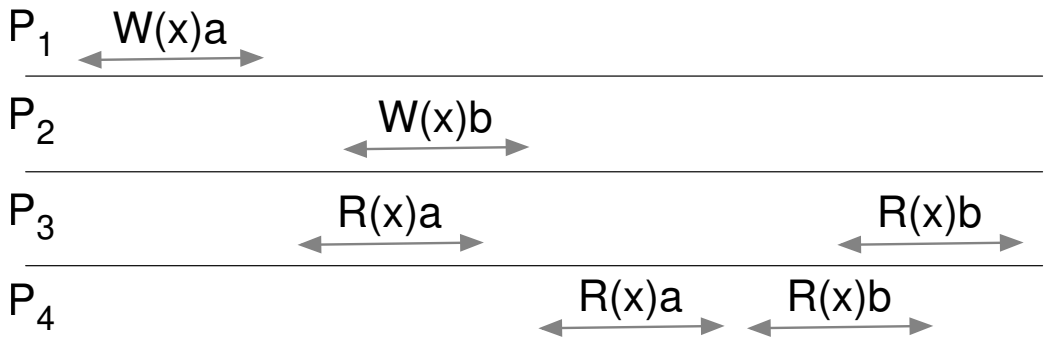
Condition de cohérence séquentielle

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

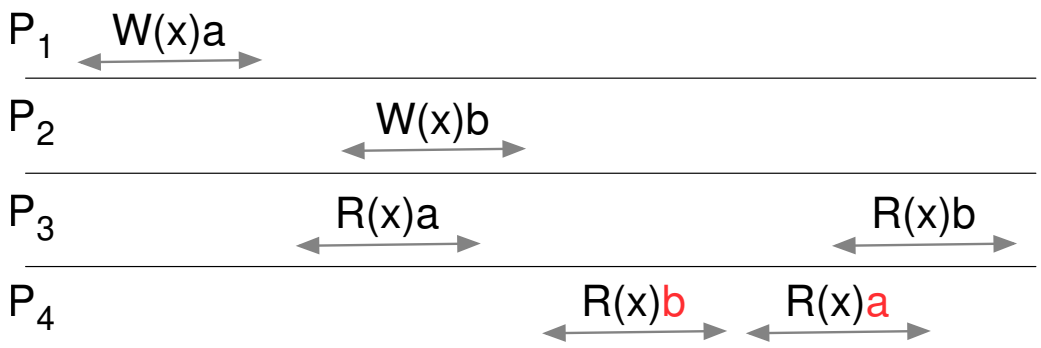
- Toutes les opérations sur les données (vues comme centralisées) sont exécutées selon une certaine séquence S
- Les opérations exécutées par tout processus P figurent dans le même ordre dans S et dans P
- La cohérence interne des données est respectée dans S : chaque lecture doit renvoyer la valeur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

- il suffit que la séquence S puisse être construite
- Définit une condition globale à vérifier : compatibilité des histoires des lectures des différents sites $\rightarrow$ coûteux

Cohérence séquentielle : (contre-)exemples

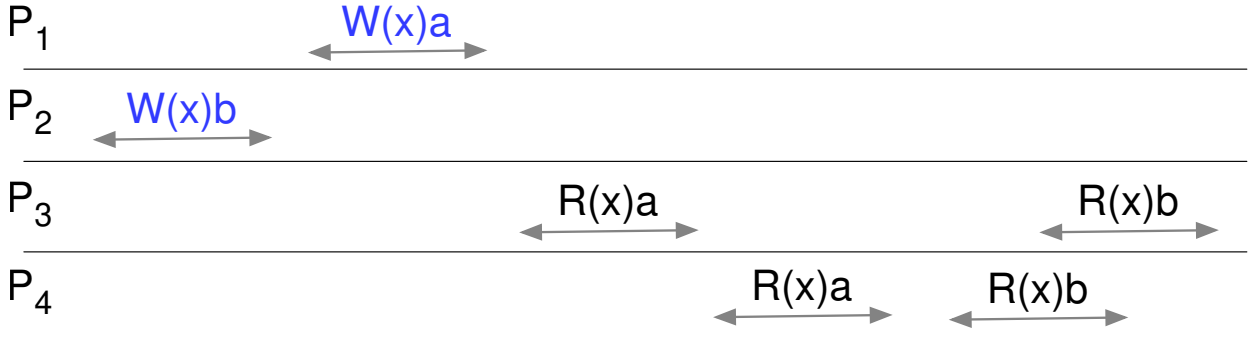


Oui : $S = W(x)a\ ; R_3(x)a\ ; R_4(x)a\ ; W(x)b\ ; R_3(x)b\ ; R_4(x)b$



Non : P_3 : W(x)a précède W(x)b dans S ; P_4 : W(x)b précède W(x)a dans S

? :



Mise en œuvre : lectures immédiates, écritures retardées

Protocole suivi par P_i

- Appel local à $R_i(x)$: retourner la valeur de la copie locale
- Appel local à $W_i(x)a$:
  1. diffusion atomique de $W_i(x)a$ à l'ensemble des sites
  2. attendre la réception de la requête émise
- Réception de $W_j(x)a$ par le protocole de cohérence :
  1. réaliser l'écriture sur la copie locale
  2. **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 deux écritures commutent $\rightarrow$ les lectures locales suffisent à assurer la compatibilité.

Mise en œuvre : lectures inhibées, écritures immédiates

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

Protocole suivi par P_i

- 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
- Réception de $W_j(x)a$ par le protocole de cohérence :
  1. réaliser l'écriture sur la copie locale
  2. **si** j $=$ i **alors**
     cpt−− ;
     si cpt $= 0$ **alors** retourner 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)

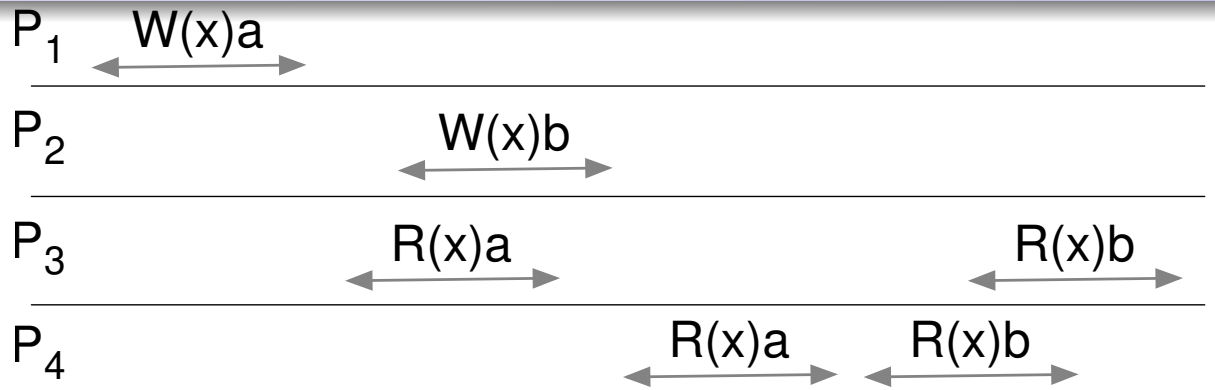
Condition de linéarisabilité

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

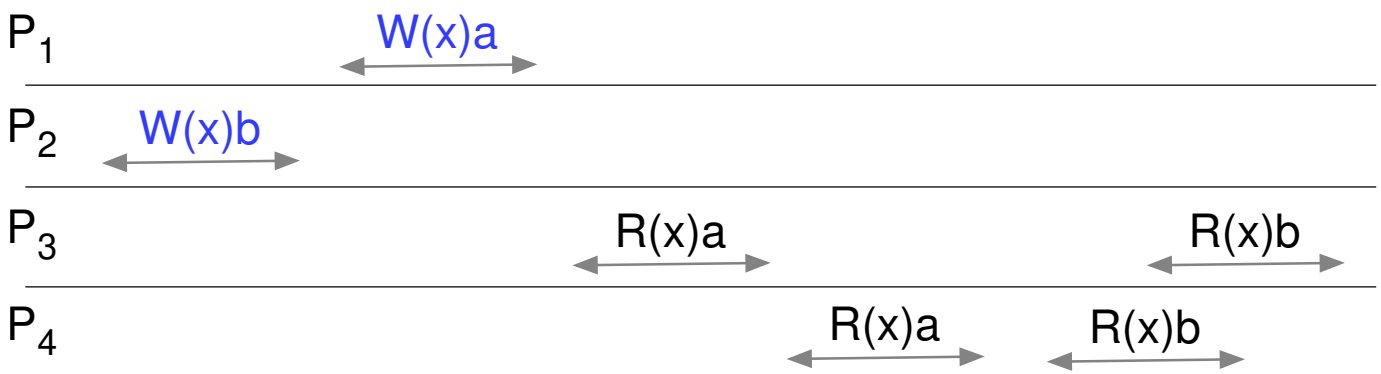
- Toutes les opérations sur les données sont exécutées selon une certaine séquence globale S
- La cohérence interne des données est respectée dans S
- Si deux opérations op_1 et op_2 (lectures ou écritures) sont telles que $op_1 \rightarrow op_2$, alors op_1 et op_2 figurent dans cet ordre dans S

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

Linéarisabilité et cohérence séquentielle



Linéarisable : $S = W(x)a\ ; R_3(x)a\ ; R_4(x)a\ ; W(x)b\ ; R_3(x)b\ ; R_4(x)b$



Cohérence séquentielle :
$S = W(x)a\ ; R_3(x)a\ ; R_4(x)a\ ; W(x)b\ ; R_3(x)b\ ; R_4(x)b$
Mais non linéarisable : P_3 et P_4 commencent par $R(x)a \Rightarrow$ pas de séquence $W(x)b\ ;\ \ldots$

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)
- e_1 est l'écriture d'une variable x et e_2 une lecture ultérieure de x (causalité entre écriture et lecture)
- 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.

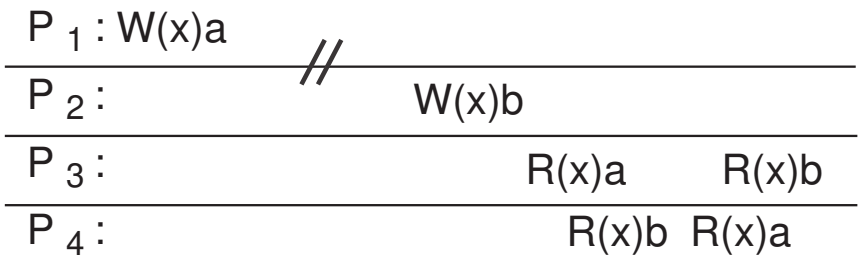
Cohérence causale

Remarque : comme toute lecture renvoie le résultat d'une écriture précédente, la condition de cohérence causale entre lecture et écriture est systématiquement assurée.
La condition de cohérence causale peut donc être reformulée en portant seulement sur les écritures :

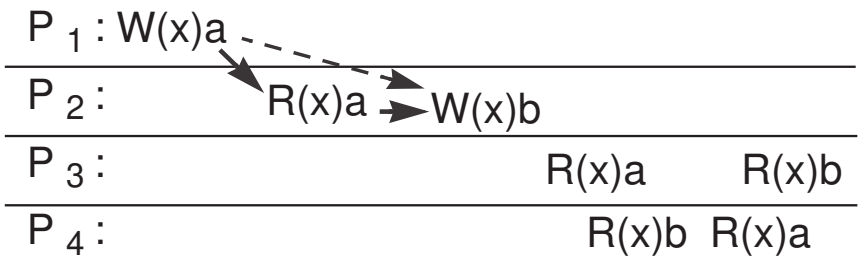
> Tout processus qui voit les résultats d'écritures causalement liées doit les voir dans l'ordre causal de ces écritures.

Note : des écritures causalement indépendantes ($\|$) peuvent être vues dans un ordre différent sur des processus différents.

Cohérence causale : exemples



non séquentiel, mais
causalement cohérent



P4 non causalement cohérent

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

Mise en œuvre : capturer les dépendances causales par des horloges vectorielles / matricielles.

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 sur des processus différents.

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)

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

Cohérence ponctuelle

Les protocoles de cohérence continue spécifient une relation invariante liant l'ensemble des valeurs successives de 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 $\rightarrow$ inutile de gérer la cohérence sur les opérations internes à SC.

Fournir des objets de synchronisation, 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
$\rightarrow$ mise en cohérence des copies locales du site appelant :

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

Remarques

- entre deux appels à `S.synchronize()`, la cohérence n'a pas à être assurée
- propager $\neq$ mettre immédiatement en cohérence
- différents modèles sont définis selon le moment où la synchronisation est réalisée

Cohérence faible

- 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*

Cohérence à la libération (*release consistency*)

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

- Appel de *acquire()* : les copies locales de l'ensemble de données protégé par le verrou sont mises à jour.
- Appel de *release()* : les valeurs des variables qui ont été modifiées sont envoyées vers les copies distantes.
- Cohérence FIFO sur les appels à *V.acquire()* et *V.release*() $\rightarrow$ exécution séquentiellement cohérente.
- Variante : cohérence paresseuse à la libération : 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.

Cohérence à l'entrée (*entry consistency*)

- Association explicite entre verrou et variables partagées
- Verrouillage obligatoire avant de lire ou écrire une variable
- Mise en cohérence des variables associées (et elles seules) lors de la prise du verrou

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

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

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 : DNS, 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

Principe

- une session par utilisateur
- chaque écriture a un identifiant de version global et une date
- 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

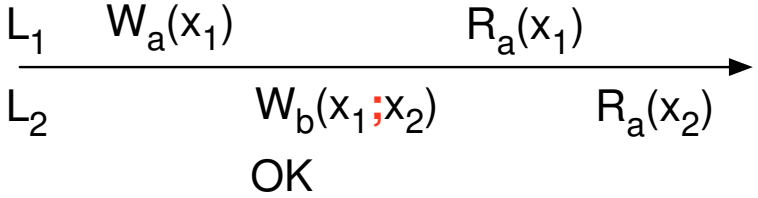
Notations

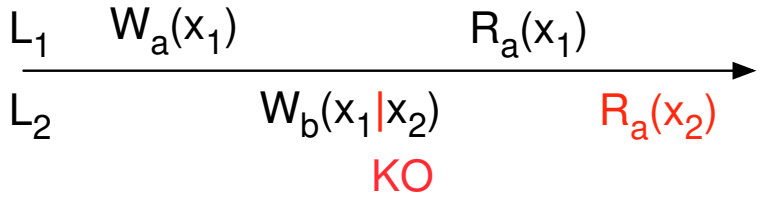
- L_s site (lieu/copie)
- x_i valeur/version de x sur une copie locale L_i
- $R_a(x_i)$ lecture de la version x_i par le client a
- $W_a(x_i)$ écriture indépendante de la version x_i par le client a
- $W_a(x_i; x_j)$ écriture de la version x_j par le client a, tenant compte de (dépendante de) la version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

Lectures monotones

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.



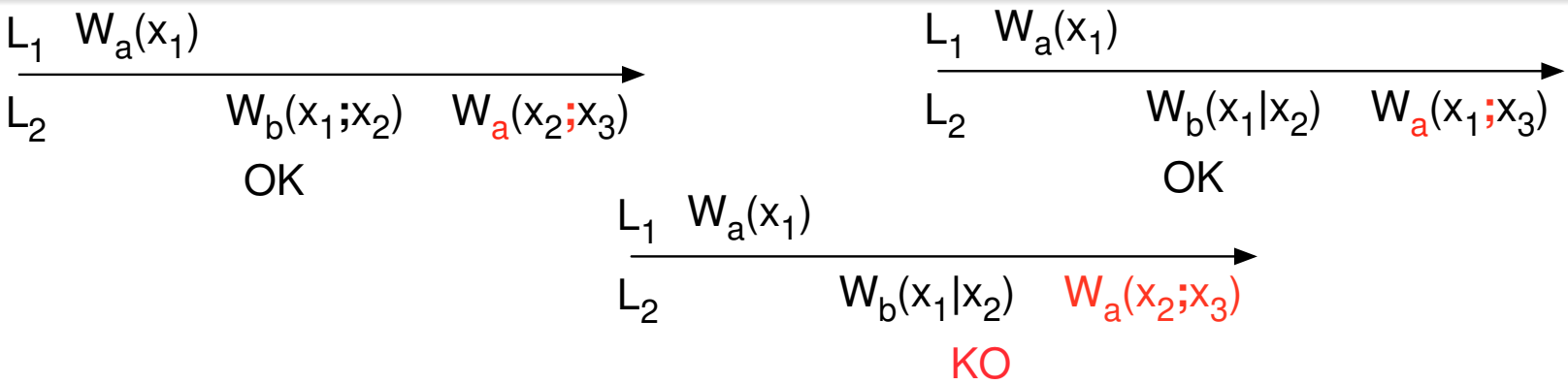


Mise en œuvre

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

Écritures monotones

Si un client exécute deux 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

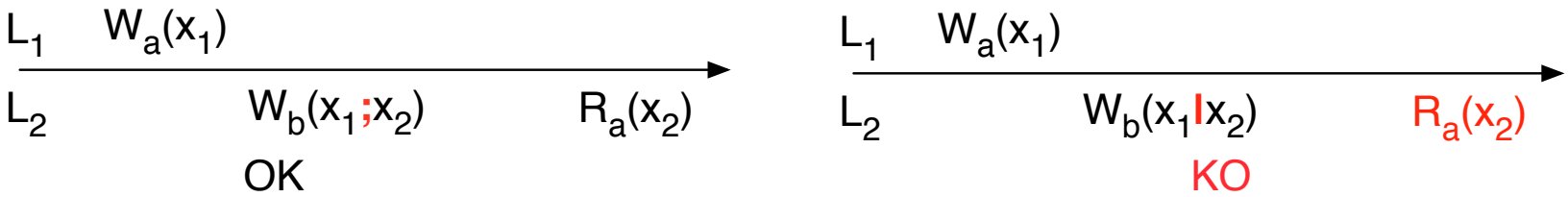
Mise en œuvre

- l'ensemble d'écritures du journal du client permet de déterminer et récupérer les écritures à faire 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

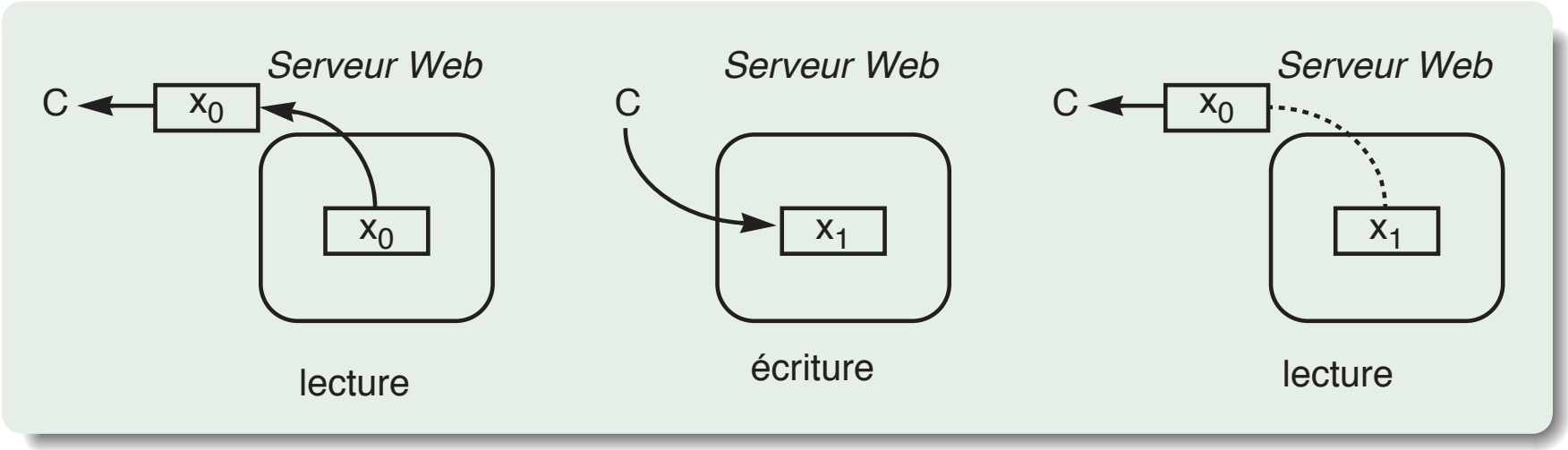


Mise en œuvre

- 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*)

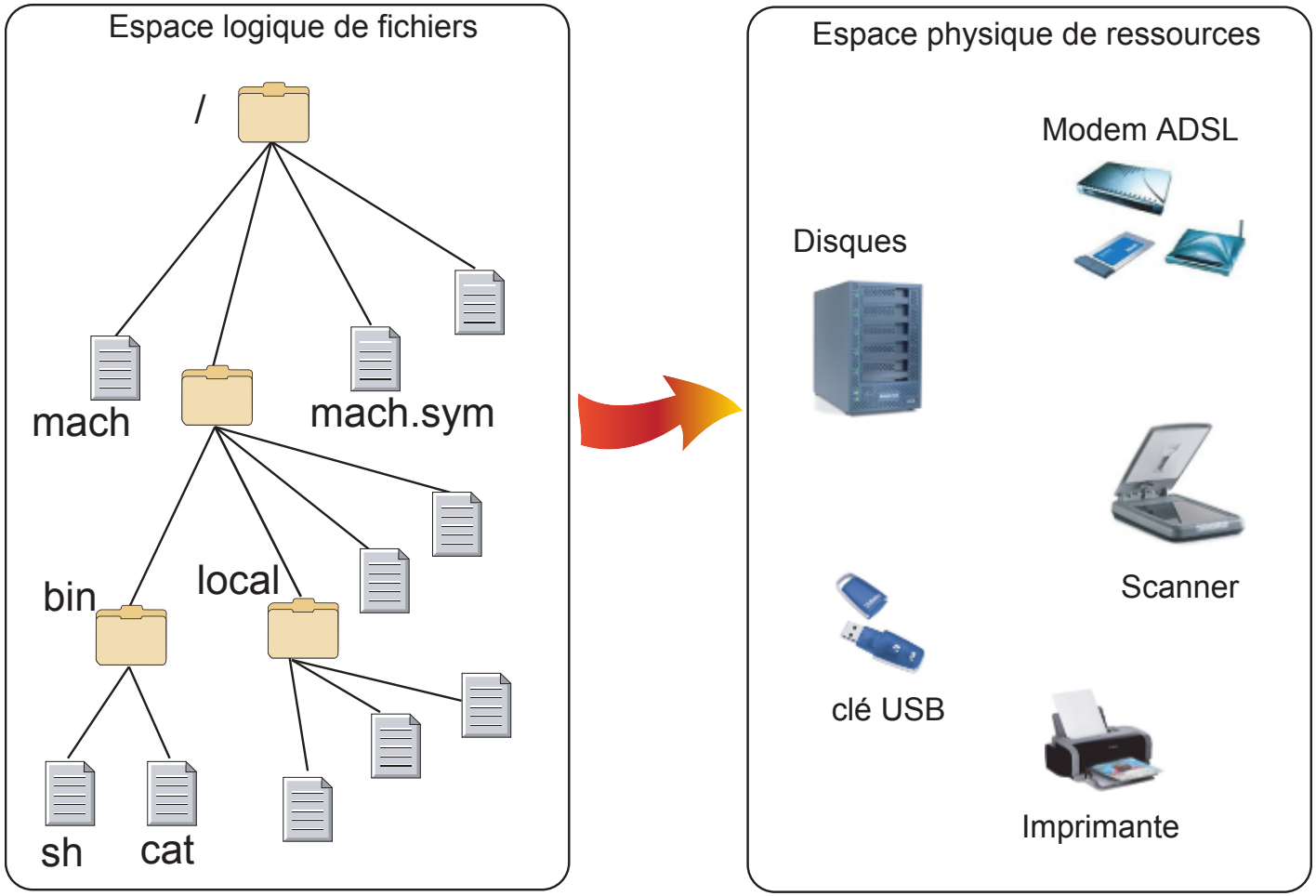
Exemple de violation



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

Plan

Données rémanentes et abstraction des ressources



Projection de l'espace logique sur l'espace physique

Objectif

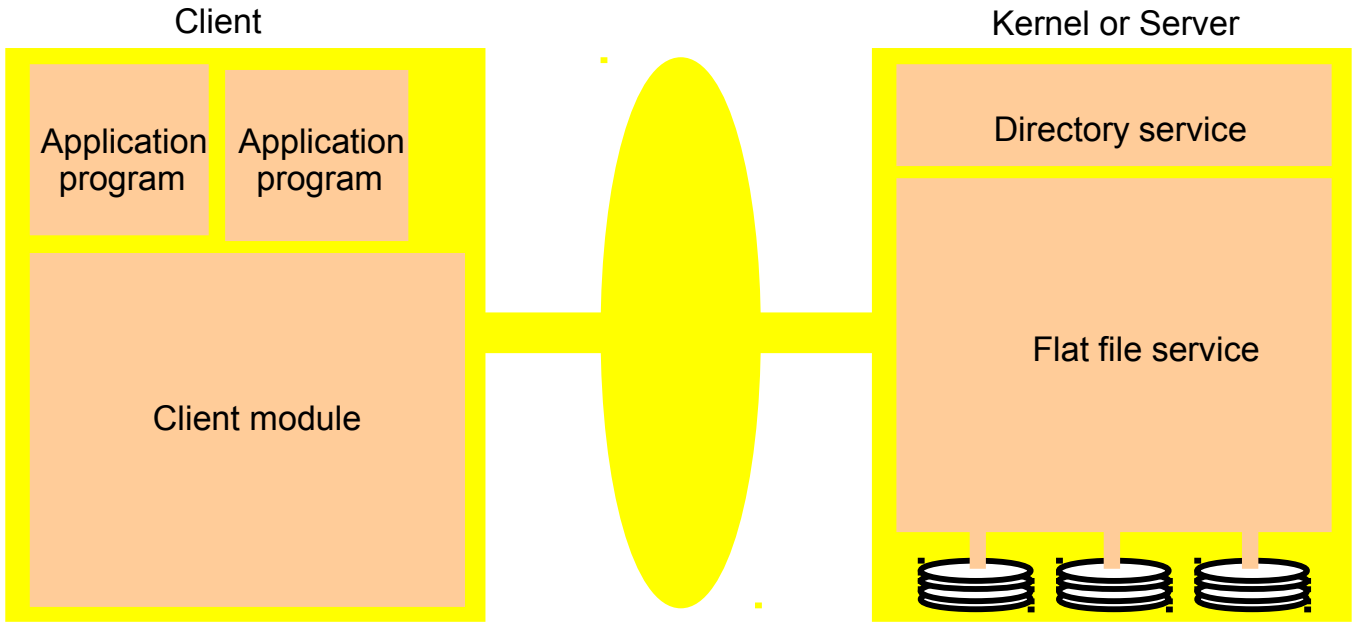
Un **bon** système de fichiers **répartis**
est un système de fichiers...
qui peut passer pour un système **centralisé**

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
- transparence de l'hétérogénéité
- transparence d'échelle
- ...

Architecture d'un système de fichiers

- Service de fichiers à plat (flat file system)
- Service de nommage (répertoires)
- Module client : API simple et unique, indépendante de l'implantation



(source : Coulouris – Dollimore)

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
stat(nom,attributs)	Lecture des attributs du fichier
unlink(nom)	suppression du nom dans le rép.
link(nom_orig,synonyme)	Nouvelle référence

SGF réparti : Niveau répertoire

Fonction

- Noms symboliques $\leftrightarrow$ noms internes (Unique File IDentifiers 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)	résolution du nom
AddName(UFID rép, String nom, UFID uid)	insérer le nom
UnName(UFID rép, String nom)	supprimer le nom
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

- Opérations idempotentes (sauf create) : RPC at-least-once
- Serveur sans état : redémarrage sans reconstruction de l'état précédent

Problème sémantique du partage

Idéalement

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

Difficultés

- ☹ Ordre total
- ☹ Propagation immédiate des modifications
- ☹ Pas de copie $\Rightarrow$ contention

$\Rightarrow$ 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 :
  $\Rightarrow$ 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

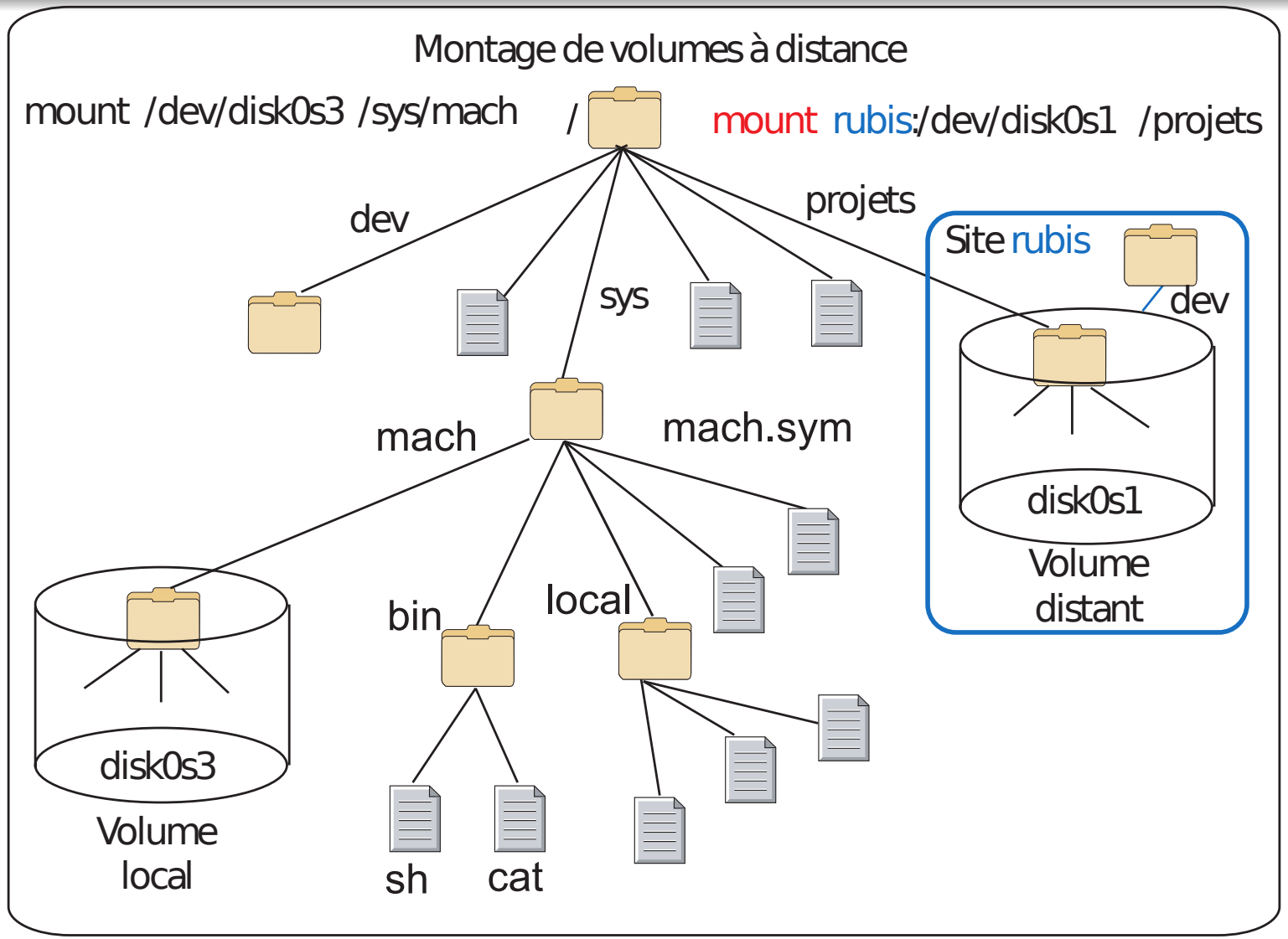
Autres solutions

- 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 (immuable)

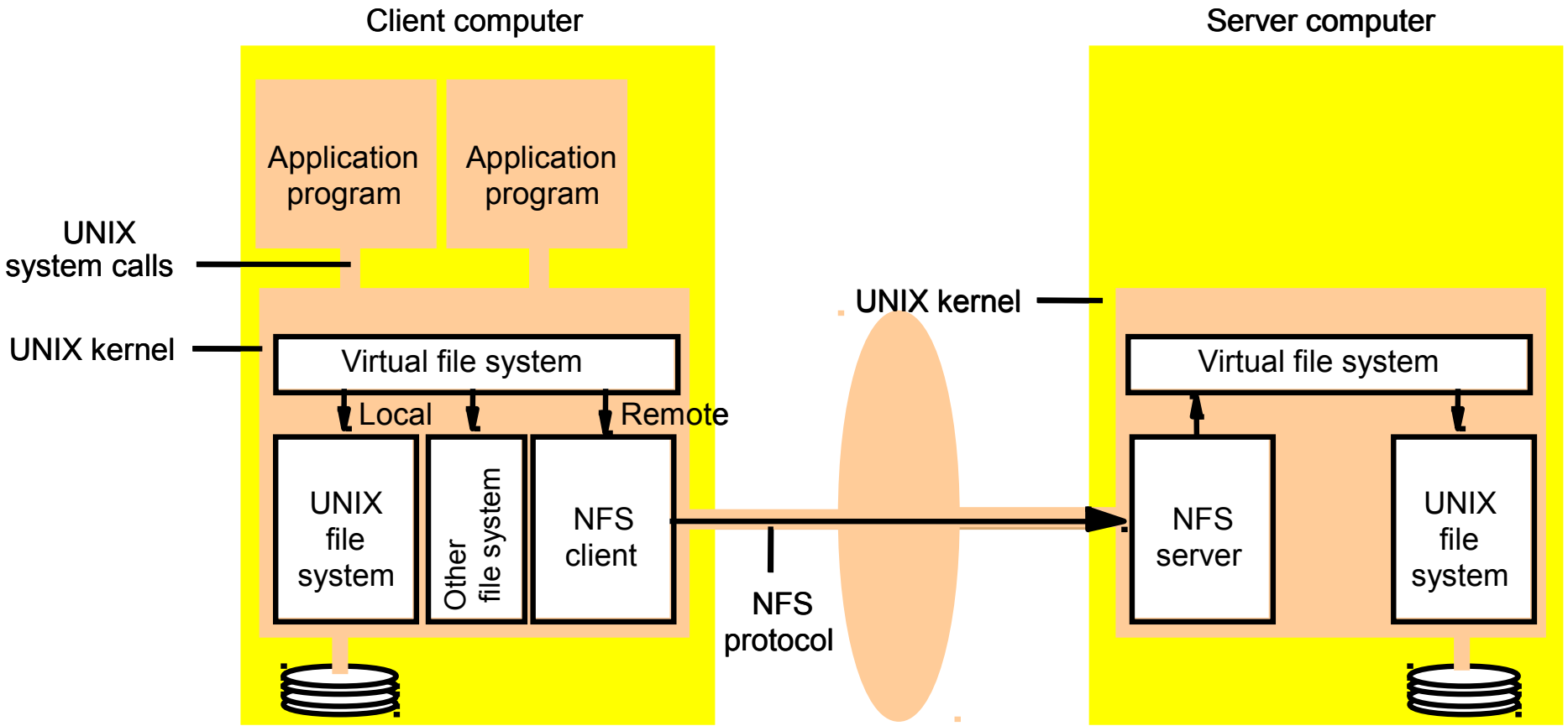
NFS (Network File System) : Principes

- Origine : Sun Microsystems (1984, 1995, 2003, 2010)
- Transparence d'accès et de localisation ($\rightarrow$ *virtual file system*)
- Traitement de l'hétérogénéité ($\rightarrow$ *virtual node* vnode)
- Désignation non uniforme (a priori)
- Approche intégrée (modification des primitives du noyau)
- Protocole RPC entre **noyaux** ($\rightarrow$ système fermé) : sémantique « au moins une fois »
- Serveur : site qui exporte des volumes
- Client : accès à un système de fichiers distants par montage à distance (extension du mount)
- Serveur sans état (mais v4 avec état)

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



Architecture de NFS



(source : Coulouris – Dollimore)

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 $\rightarrow$ 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^o inode	n^o 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)

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

AFS (Andrew File System)

Principes

- Origine : Carnegie Mellon University (1986)
- SGF uniforme pour servir plusieurs milliers de machines
- Deux composants :
  - les stations serveurs (VICE)
  - les stations clientes (VERTUE)
- Réplication en lecture seule des fichiers systèmes
- Migration des fichiers utilisateurs (serveur $\leftrightarrow$ client)
- Gestion de caches 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, contacter un serveur pour obtenir une copie de tout le fichier

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 (callback)

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

Évolutions

Limitations

- Introduction de la mobilité : partitionnement réseau, mode déconnecté, pannes, etc
- Réplication (résolution des conflits d'écriture ?)
- Unité de travail = document = ensemble de fichiers
  $\Rightarrow$ Constant Data Availability

Exemples d'évolutions

- Coda (CMU) : réplication optimiste en mode déconnecté, résolution de conflit manuelle
- Peer-to-peer (voir Locus, extension de NFS) : Ficus (UCLA)
- Travail collaboratif : Bayou (Xerox PARC)
- Systèmes décentralisés de gestion de versions (git)
- Sites d'hébergement / partage de fichiers (Dropbox, Google Drive...)

Conclusion

Des problèmes bien maîtrisés

- Implantation performante des fichiers dans un contexte réparti en assurant transparence d'accès et de localisation
  $\Rightarrow$ conduit à une re-centralisation : serveurs dédiés
- Problème : concurrence d'accès aux fichiers partagés modifiés
  $\Rightarrow$ Approches « session » ou spécifiques (transactions)

Des tendances fortes et/ou solutions matures

- Tolérance aux fautes : par réplication (transparence assurée)
- Évolution pour prendre en compte la mobilité des clients
- Sécurité : par partitionnement

GFS – Google File System

Objectifs

- Système de fichiers *scalable*
- Applications manipulant de gros volumes de données
- Haute performance pour un grand nombre de clients
- Tolérant aux fautes sur du matériel basique

Exemple d'applications : stockage de vidéo (youtube), moteur de recherche (google), transactions commerciales passées (amazon), catalogues de vente

1. *The Google File System*, Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. 19th ACM Symposium on Operating Systems Principles. October 2003.

GFS – hypothèses

- Hypothèses traditionnelles invalides (« fichiers majoritairement petits », « courte durée de vie », « peu de modifications concurrentes »)
- Les pannes sont normales, pas exceptionnelles : milliers de serveurs de stockage
- Les fichiers sont énormes : plusieurs TB est normal
- Accès aux fichiers :
  - Lecture majoritairement séquentielle, rarement arbitraire
  - Écriture majoritairement en mode ajout, rarement arbitraire
  - Deux sous-classes :
    - Fichiers créés puis quasiment que lus (ex : vidéos)
    - Fichiers constamment en ajout par des centaines d'applications, peu lus (ex : log)
  - Nombre de fichiers faible par rapport au volume (quelques millions)
  - Applications dédiées ⇒ cohérence relâchée

Interface

Nommage

Nommage non hiérarchique

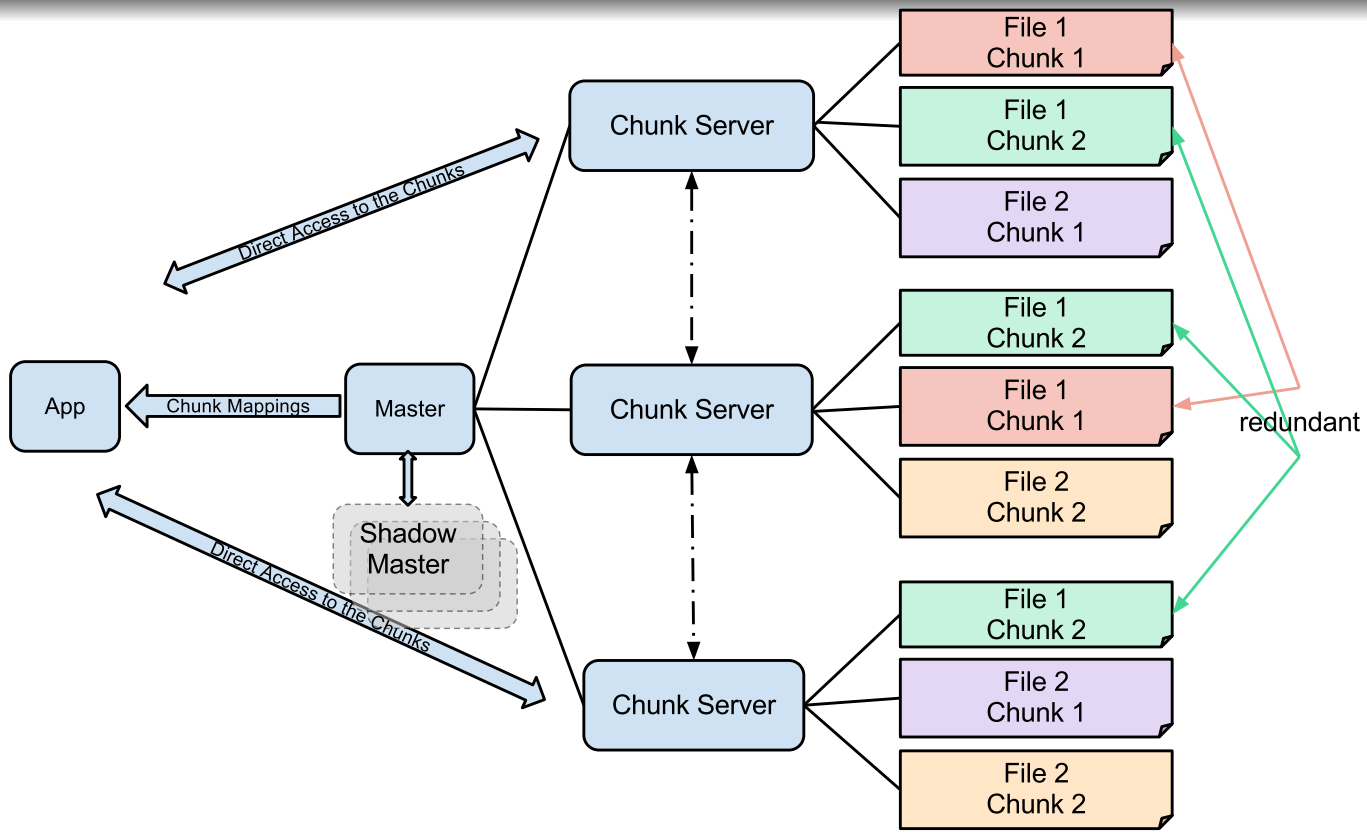
Opérations

- Classiques : *create*, *delete*, *open*, *close*, *read*, *write*
- Additionnelles :
  - *append* : partage en écriture, sans verrouillage
  - *snapshot* : copie efficace d'un fichier ou d'une arborescence, sans verrouillage

Implantation

- Ordinateurs de bureau, équipés d'un système de fichiers standard (linux : extfs)
- Serveurs GFS implantés en espace utilisateur

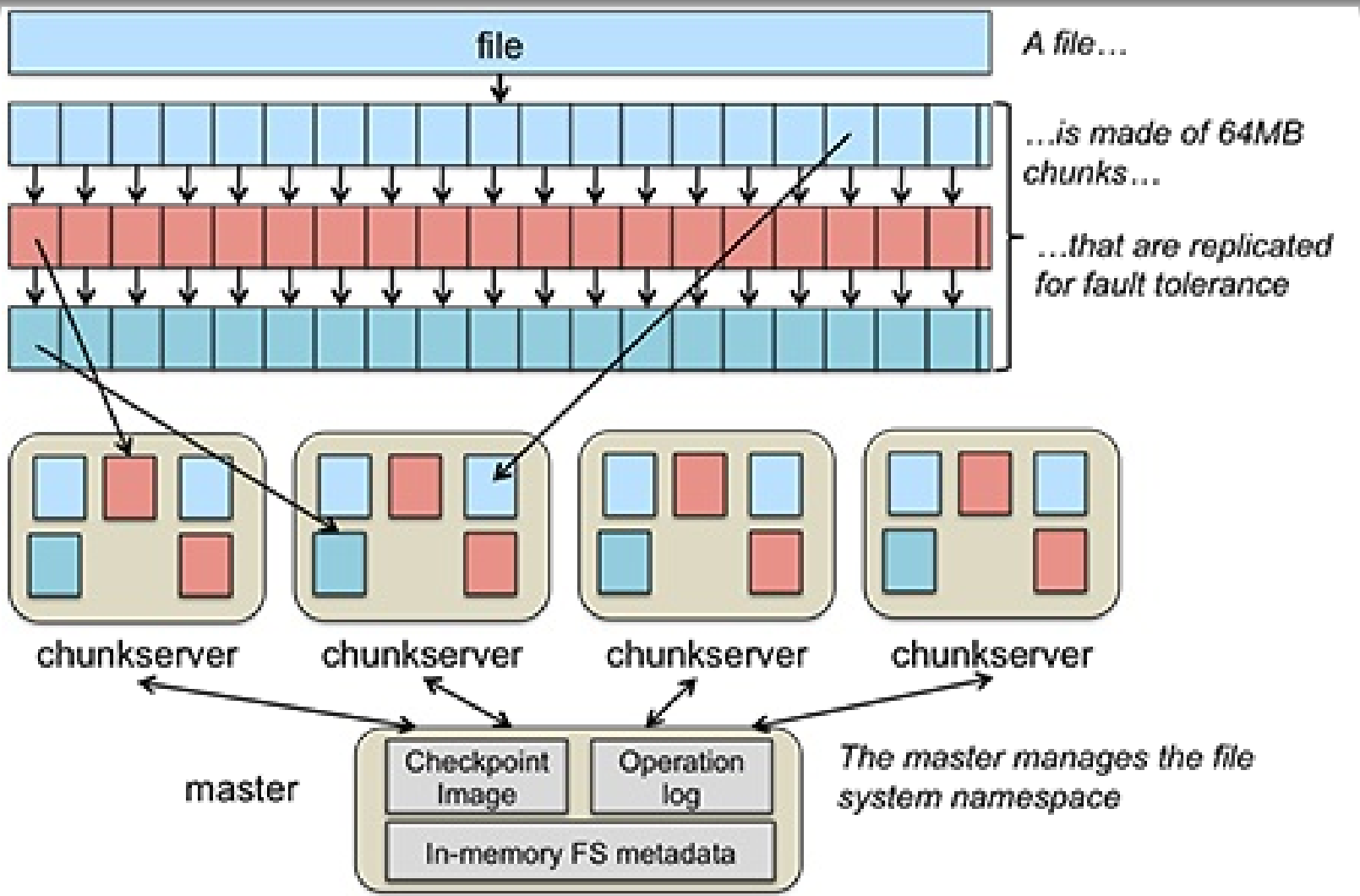
GFS cluster



- Un maître : metadonnées + table de chunks par fichier
- Des chunk servers : fichiers découpés en chunks (64 MB) répliqués, plusieurs milliers de serveurs par maître

(source : Wikipedia)

GFS fichiers



(source : Paul Krzyzanowski)

GFS – Master

- Métadonnées des fichiers
  - Nommage : bête table de hachage (pas de répertoire géré par le système, mais fichier applicatif contenant une liste de noms de fichiers)
  - Contrôle d'accès
  - Table fichier $\rightarrow$ chunks
  - Emplacement des chunks
  - Nommage et tables en mémoire (relativement petites) pour performance $\Rightarrow$ pas de cache côté serveur
- Maintenance
  - Verrouillage de chunk
  - Allocation/déallocation de chunk, ramasse-miettes
  - Migration de chunk (équilibrage, arrêt de machines)
- Répliqué (schéma maître-esclave) pour tolérance aux fautes

Lecture d'un fichier

1. Contacter le maître
2. Obtenir les métadonnées : identifiants de chunks formant le fichier
3. Pour chaque chunk handle, obtenir la liste des chunkservers
4. Contacter l'un quelconque des chunkservers pour obtenir les données (sans passer par le maître)

Écriture d'un fichier

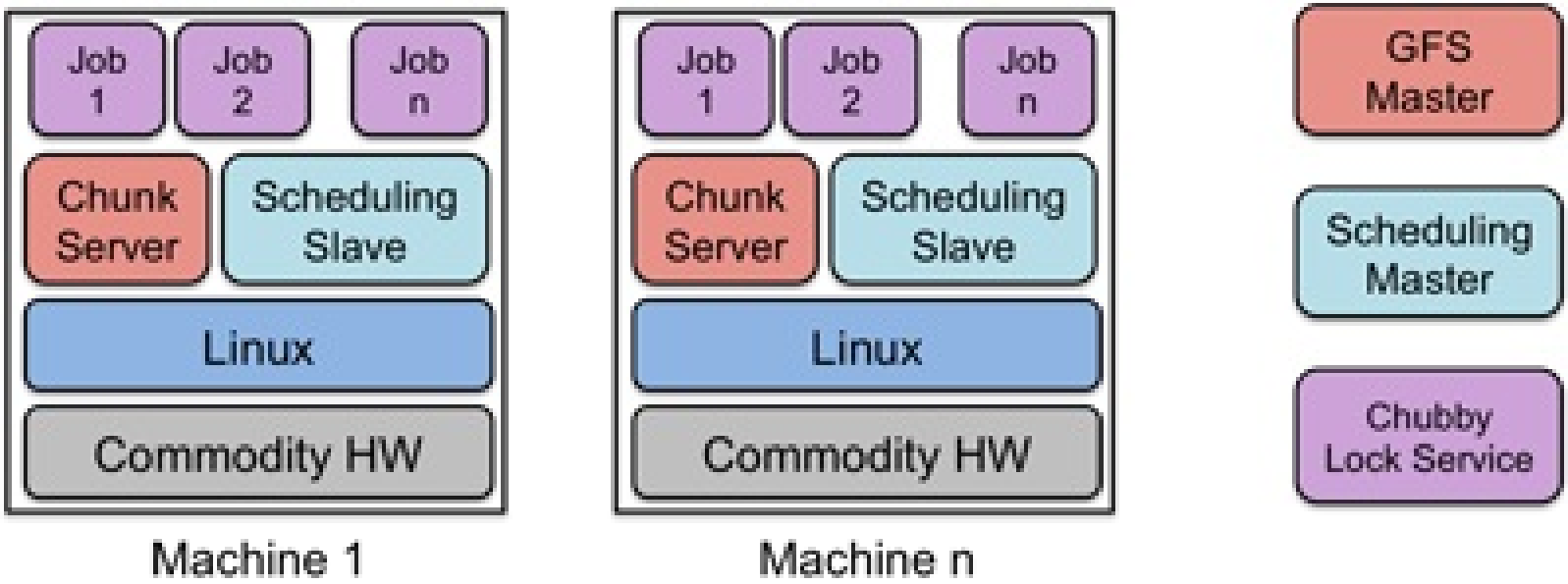
1. Phase 0 : demande d'écriture auprès du maître $\Rightarrow$ un chunk responsable + un nouveau numéro de version
2. Phase 1 : envoi des données :
   - Le client envoie ses données à écrire au serveur de chunk responsable
   - Ce serveur propage à l'un des réplicas, qui propage au suivant, etc
   - Les données sont conservées en mémoire, non écrites
3. Phase 2 : écriture des données
   - Le client attend l'acquittement de tous les réplicas (du dernier)
   - Il envoie un message de validation au primaire
   - Le primaire ordonne les écritures (numéro de version), effectue son écriture et informe les réplicas
   - Les réplicas effectuent l'écriture et l'acquittent au primaire. L'ordre des opérations est identique pour tous (numéro de version)
4. Le primaire informe le client

Google Cluster Environment

Amener les calculs aux données

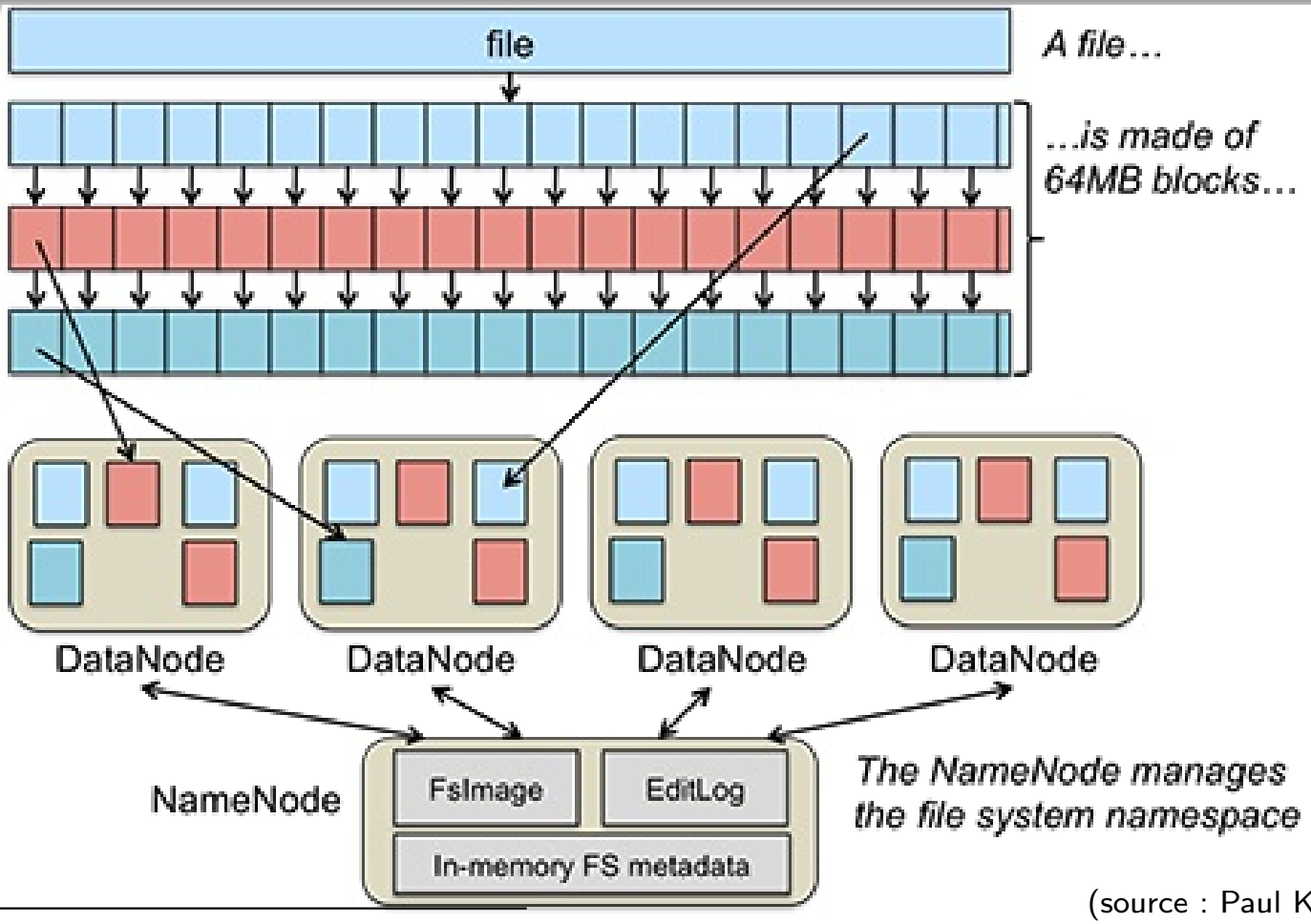
⇒ patron *mapreduce* :

1. Map : filtrage + tri (naturellement ||)
2. Shuffle : redistribution des résultats intermédiaires
3. Reduce : fusion (|| si associatif)



(source : Paul Krzyzanowski)

HDFS – Hadoop Distributed File System



(source : Paul Krzyzanowski)

1. *The Hadoop distributed filesystem : Balancing portability and performance*, Jeffrey Shafer, Scott Rixner and Alan L. Cox. Int'l Symposium on Performance Analysis of Systems and Software. March 2010.

Dropbox – partage de fichiers

Réplication grand public

- Répertoires sur différents ordinateurs/mobiles, synchronisés pour avoir le même contenu
- Schéma copie primaire (Dropbox) / copies secondaires (utilisateur)

Hypothèses

- Énormément d'utilisateurs ($>$ 200 millions)
- Par utilisateur : peu de volume (quelques Go) et peu de modifications (quelques centaines / jour)
- Ressources utilisateur faibles
- Bande passante utilisateur faible
- Rapport lecture/écriture proche de 1

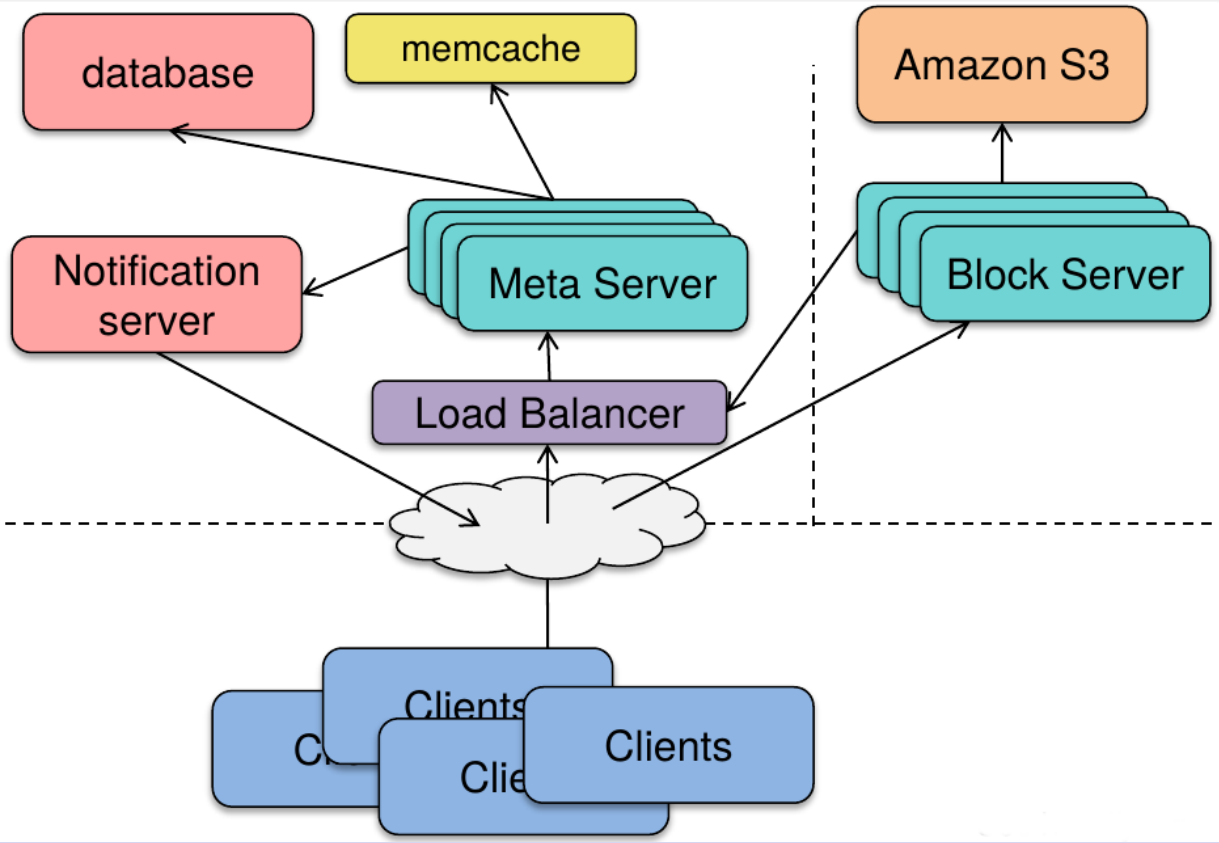
Dropbox – principes

Approche

- Copie locale complète du répertoire partagé (moins vrai avec l'application mobile : cache)
- Accès local et natif aux fichiers partagés (lecture et écriture)
- Trafic uniquement en cas de modification
- Client local léger assurant la cohérence
- Cohérence « suffisante » : écritures en arrière-plan, notification asynchrone des modifications (client connecté)

Dropbox – architecture

Séparation entre serveurs de metadonnées (nommage, droits d'accès, liste de blocs) et serveurs de blocs de données. Données hébergées chez Amazon ($\leq$ 2016), metadonnées chez Dropbox.



Conclusion

- Abstraction centralisée d'un système réparti :
  - mémoire virtuelle répartie
  - système de fichiers réparti
- Réplication pour la disponibilité et la performance
- Coût de la cohérence stricte $\rightarrow$ grande variété de cohérences, pas toujours intuitives, différentes selon les besoins