Systèmes et algorithmes répartis

Données réparties

Philippe Quéinnec, Gérard Padiou

Département Informatique et Mathématiques Appliquées
ENSEEIHT

4 octobre 2016

Première partie

Réplication de données

plan

Plan

Réplication de données

Réplication

Placement de plusieurs exemplaires d'une même donnée sur différents sites.

Intérêt de la réplication

- Favorise les accès locaux : performance
- Tolérance aux pannes (copies multiples)
- Répartition de charge
- Mode déconnecté envisageable en cohérence à terme

Principe général

- Principe : fournir des objets partagés par couplage dans les espaces d'adressage de structures d'exécution (couplage virtuel) réparties
- Partage par copie locale (efficacité)
- Programmation simple (accès local)
- Le système charge éventuellement les données à la demande
- Le système assure la cohérence des données partagées

Réplication optimiste / pessimiste

Réplication optimiste

- Autoriser l'accès à une copie sans synchronisation a priori avec les autres copies
- Modifications propagées en arrière plan
- Conflits supposés peu nombreux, et résolus quand ils sont détectés

Réplication pessimiste

- Garantit l'absence de conflits
- Mécanisme bloquant de prévention

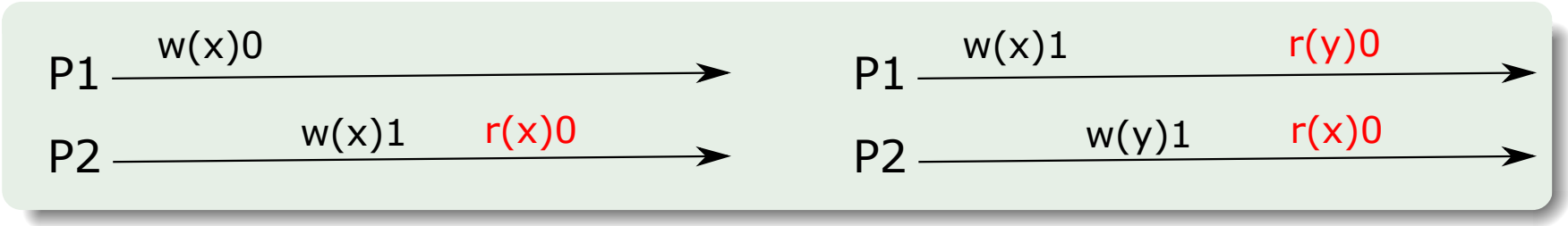
Where a pessimistic algorithm waits, an optimistic one speculates.

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 !



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

Plan

Cohérence

Définition

Cohérence : relation que gardent entre elles les différentes copies des données

- Cohérence stricte, linéarisabilité, cohérence séquentielle
- Cohérences faibles (weak consistency)
  - cohérence à la sortie (release consistency)
  - cohérence à l'entrée (entry consistency)
- Cohérences non séquentielles
  - cohérence causale
  - cohérence à terme (eventual consistency)

Cohérence stricte

Cohérence stricte

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

- Modèle idéal
- Difficile à mettre en œuvre
  - Synchronisation des horloges (difficile d'avoir un référentiel de temps absolu)
  - Temps de communication (temps de propagation des mises à jour)
  - Opérations instantanées
- $\Rightarrow$ approximation de ce modèle par des modèles moins contraignants

Linéarisabilité

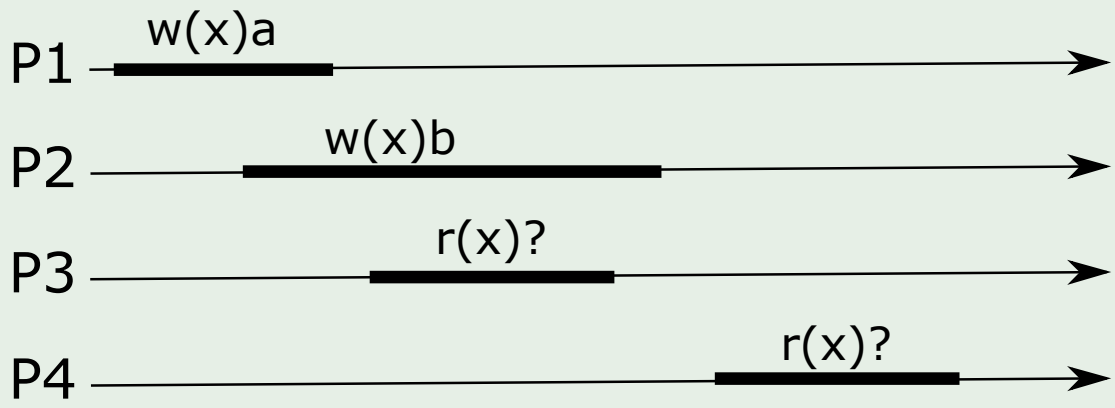
Définition

Le résultat d'une exécution parallèle est le même que celui d'une exécution séquentielle qui respecte l'enchaînement des opérations.

- Toutes les opérations sont exécutées selon une certaine séquence S (comme en centralisé)
- Si deux opérations (lecture ou écriture) sont telles que la première finit avant que la deuxième ne commence, alors elles figurent dans cet ordre dans S.
- La cohérence interne des données est respectée dans S (après une écriture et jusqu'à la prochaine, la lecture délivre la valeur écrite)

Note : le point 2 nécessite un ordre total sur les dates de début/fin, par exemple le temps absolu.

Cohérence stricte vs linéarisabilité



Séquences valides pour la linéarisabilité :

$w_1(x)a \cdot w_2(x)b \cdot r_3(x)b \cdot r_4(x)b$ (= stricte sur les débuts)

$w_1(x)a \cdot r_3(x)a \cdot w_2(x)b \cdot r_4(x)b$ (= stricte sur les fins)

$w_2(x)b \cdot w_1(x)a \cdot r_3(x)a \cdot r_4(x)a$

Séquences invalides pour la linéarisabilité :

$w_2(x)b \cdot r_3(x)b \cdot w_1(x)a \cdot r_4(x)a$ (début r_3 postérieur à fin w_1)

$w_2(x)b \cdot r_4(x)b \cdot w_1(x)a \cdot r_3(x)a$ (début r_4 postérieur à fin w_1 / r_3)

Cohérence séquentielle

Définition

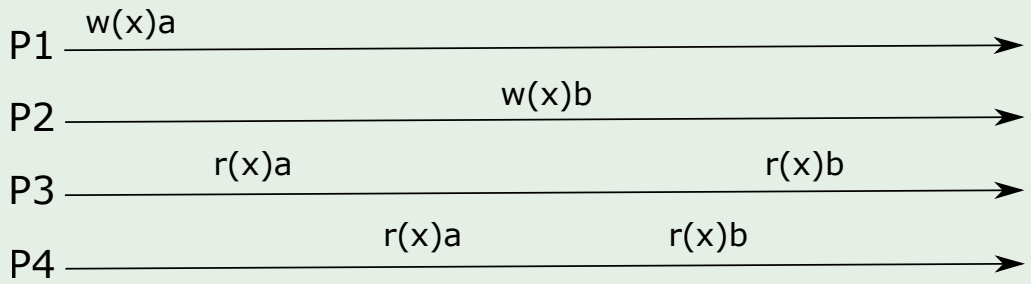
Le résultat d'une exécution parallèle est le même que celui d'une exécution séquentielle qui respecte l'ordre partiel de chacun des processeurs.

- Toutes les opérations sont exécutées selon une certaine séquence S (comme en centralisé)
- 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 (après une écriture et jusqu'à la prochaine, la lecture délivre la valeur écrite)

Remarque : on ne contraint pas l'ordre des opérations dans des processus différents.

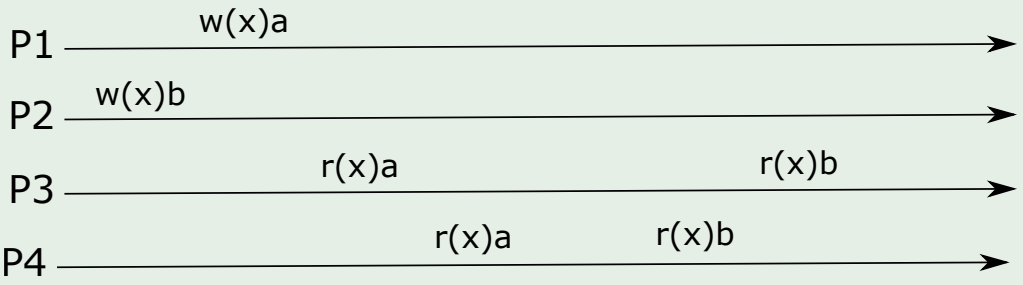
Linéarisabilité/cohérence séquentielle/non séquentielle

Linéarisable



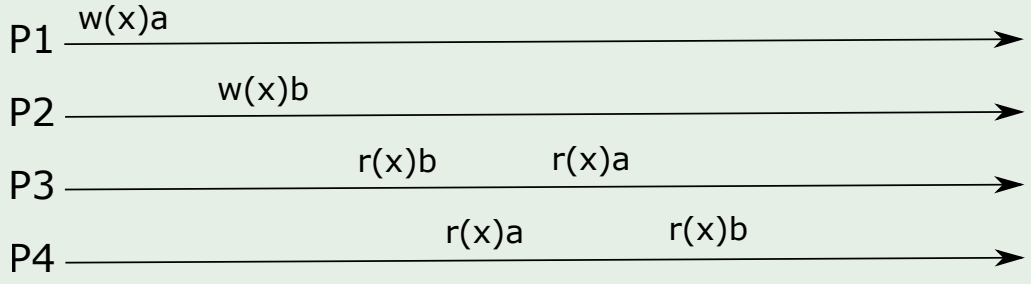
$$S = w_1(x)a \cdot r_3(x)a \cdot r_4(x)a \cdot w_2(x)b \cdot r_4(x)b \cdot r_3(x)b$$

Cohérence séquentielle



$$S = w_1(x)a \cdot r_3(x)a \cdot r_4(x)a \cdot w_2(x)b \cdot r_3(x)b \cdot r_4(x)b$$

Cohérence non séquentielle



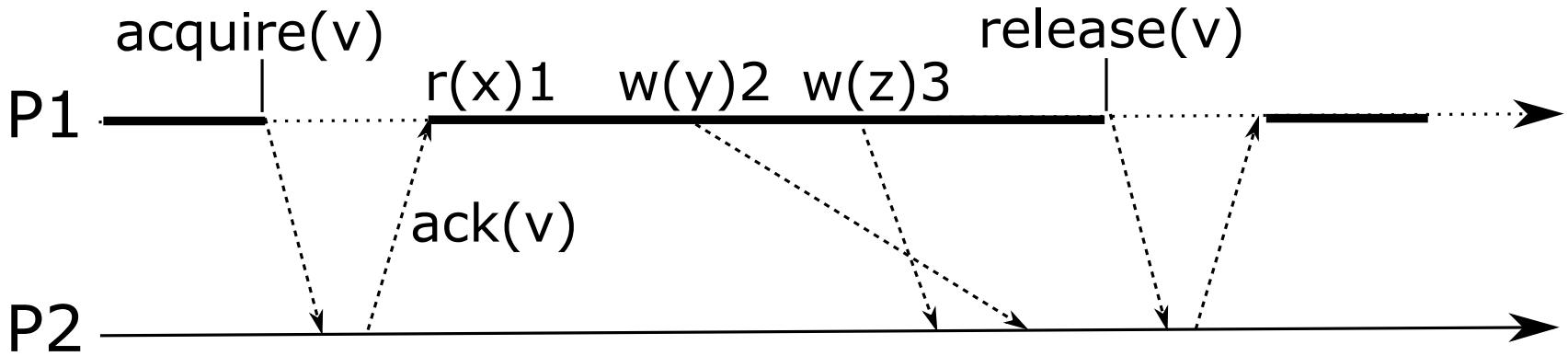
Cohérence séquentielle

- Les processeurs doivent s'accorder sur l'ordre des accès
  - un accès en écriture est terminé si une lecture à la même adresse rend la valeur écrite (quelque soit le processeur)
  - chaque écriture doit être terminée pour continuer l'exécution (sinon deux écritures peuvent s'inverser)
- Dans la pratique, algorithme d'ordonnancement global
  - diffusion des modifications à tous les sites (ordre total)
  - ou un serveur n'autorisant qu'une seule copie en écriture (par unité)
  - lourd et inefficace : $r + w \geq t$, où r est la durée d'une lecture, w d'une écriture, et t est le temps minimal de transfert d'un message
  - mais assez intuitif

Cohérence et synchronisation

La présence de code de synchronisation (p.e. exclusion mutuelle ou verrous) simplifie la cohérence nécessaire :

- Cohérence séquentielle pour la prise des verrous
- Cohérence faible pour les objets
- Règle : tous les accès mémoire doivent être terminés avant chaque point de synchronisation



Cohérences faibles

Différencier les points d'entrée et de sortie de synchronisation

Cohérence à la sortie (release consistency)

- en sortie de synchronisation, les accès doivent être terminés
- mise à jour précoce (eager) : envoi des modifications à tous les noeuds (ayant une copie)
- mise à jour paresseuse (lazy) : acquisition par le processeur qui reprend le verrou (sorte de cohérence à l'entrée)

Cohérence à l'entrée (entry consistency)

- association explicite entre variable de synchronisation et variables partagées
- verrouillage avant d'utiliser un objet
- mise en cohérence de l'objet lors de la prise du verrou

Cohérence causale

Relation de causalité

e précède causalement e' : $e \rightarrow e'$ si :

- Si e et e' sont sur le même site, et e précède e'
- Si $e = w(x)a$, et $e' = r(x)a$ (e' = lecture de la valeur écrite en e)
- Transitivité : si $e \rightarrow e''$ et $e'' \rightarrow e'$

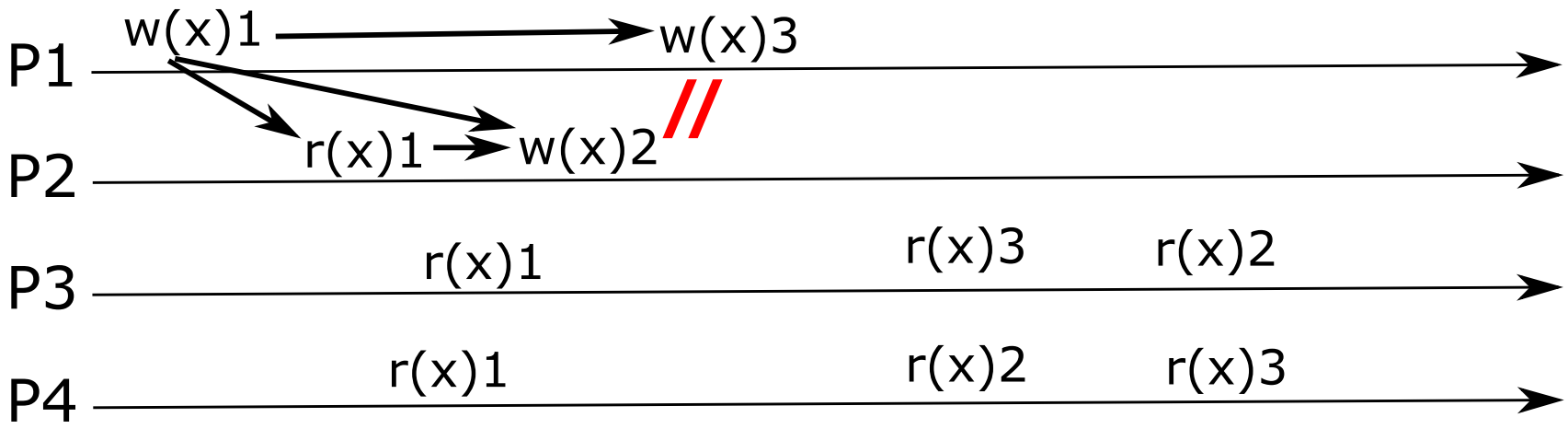
Événements concurrents = pas de précédence causale entre eux.

Cohérence causale

Les valeurs sont observées dans un ordre compatible avec la causalité.

Note : deux écritures concurrentes peuvent être vues dans des ordres différents sur des processeurs différents.

Cohérence causale



- Assez lourd à mettre en œuvre (horloges matricielles)
- Mais optimisations (horloges vectorielles)
- Gains considérables en général

Cohérence à terme (eventual consistency)

Cohérence à terme

Respect de l'ordre local + datation globale ordonnant les écritures sur un même objet.
Aucune contrainte sur les lectures (valeurs périmées possibles)

- Majorité de lectures, peu d'écritures
- Acceptable de lire une donnée non à jour
  Exemples : DNS, WWW, consultation des places disponibles (mais pas réservation)
- En l'absence prolongée de modifications, les copies finiront par devenir cohérentes
- Algorithmes de propagation dits "épidémiques"

Plan

Problème 1 : placement des copies

Copies permanentes

Ensemble de copies définies a priori. Configuration statique

⇒ Architecture statique de tolérance aux fautes

Copies à la demande

Création dynamique de copies selon les besoins

⇒ Adaptation à la charge

Problème 2 : propagation des mises à jour

Write-update

- Lecture : copie locale (sans communication)
- Écriture : locale et nouvelle valeur envoyée en multicast
- Cohérence : propriétés sur l'ordre des messages
- Multicast peut être cher (à chaque écriture)

Write-invalidate

- Invalidation diffusée en cas d'écriture
- Lecture $\Rightarrow$ chargement si nécessaire
- Cache $\Rightarrow$ plusieurs lectures/écritures locales sans communication
- Nouvelles valeurs transmises que si nécessaire mais lecture pas nécessairement instantanée

Problème 3 : protocoles de cohérence

Copie primaire fixe

Serveur fixe. Toute demande de modification passe par ce serveur qui les ordonne ($\Rightarrow$ accès distant).

Copie primaire locale

Le contrôle est donné au site qui fait une mise à jour $\Rightarrow$ accès local.
Intéressant avec les cohérences liées à la synchronisation.

Duplication active

Accord de l'ensemble des serveurs :

- Diffusion fiable totalement ordonnée (atomique)
- Votes et quorums
- Algorithmes probabilistes

Duplication active – Diffusion

Principe

- Écriture : diffusion atomique totalement ordonnée à toutes les copies
  $\Rightarrow$ toutes les copies mettent à jour dans le même ordre
- Lecture : utilisation d'une copie quelconque :
  - immédiatement (valeur passée possible)
  - causalement (cohérence causale)
  - totalement ordonnée avec les écritures (cohérence séquentielle)

Duplication active – Votes et quorums

Principe

- Pour effectuer une lecture, avoir l'accord de nl copies
- Pour effectuer une écriture, avoir l'accord de ne copies
- Condition : $nl + ne > N$ (nb de copies)
- Garantit la lecture de la plus récente écriture

Votes pondérés

Donner des poids différents aux copies, selon leur importance (p.e. accessibilité)

- N nombre total de votes ($\geq$ nb de copies)
- nl = quorum de lecture
- ne = quorum d'écriture
- Conditions : $nl + ne > N$ et $2 * ne > N$

Algorithmes probabilistes

Cf chapitre « systèmes à grande échelle »

Conclusion

Intérêt de la réplication

- Favorise les accès locaux : performance
- Tolérance aux pannes (copies multiples)
- Répartition de charge
- Mode déconnecté envisageable en cohérence à terme

Limites

- Diverses formes de cohérence, aucune simple ET performante
- Cohérences faibles $\Rightarrow$ comportements surprenants
- Diverses implantations, toutes complexes
- Difficiles de faire cohabiter plusieurs formes de cohérence selon la nature des données

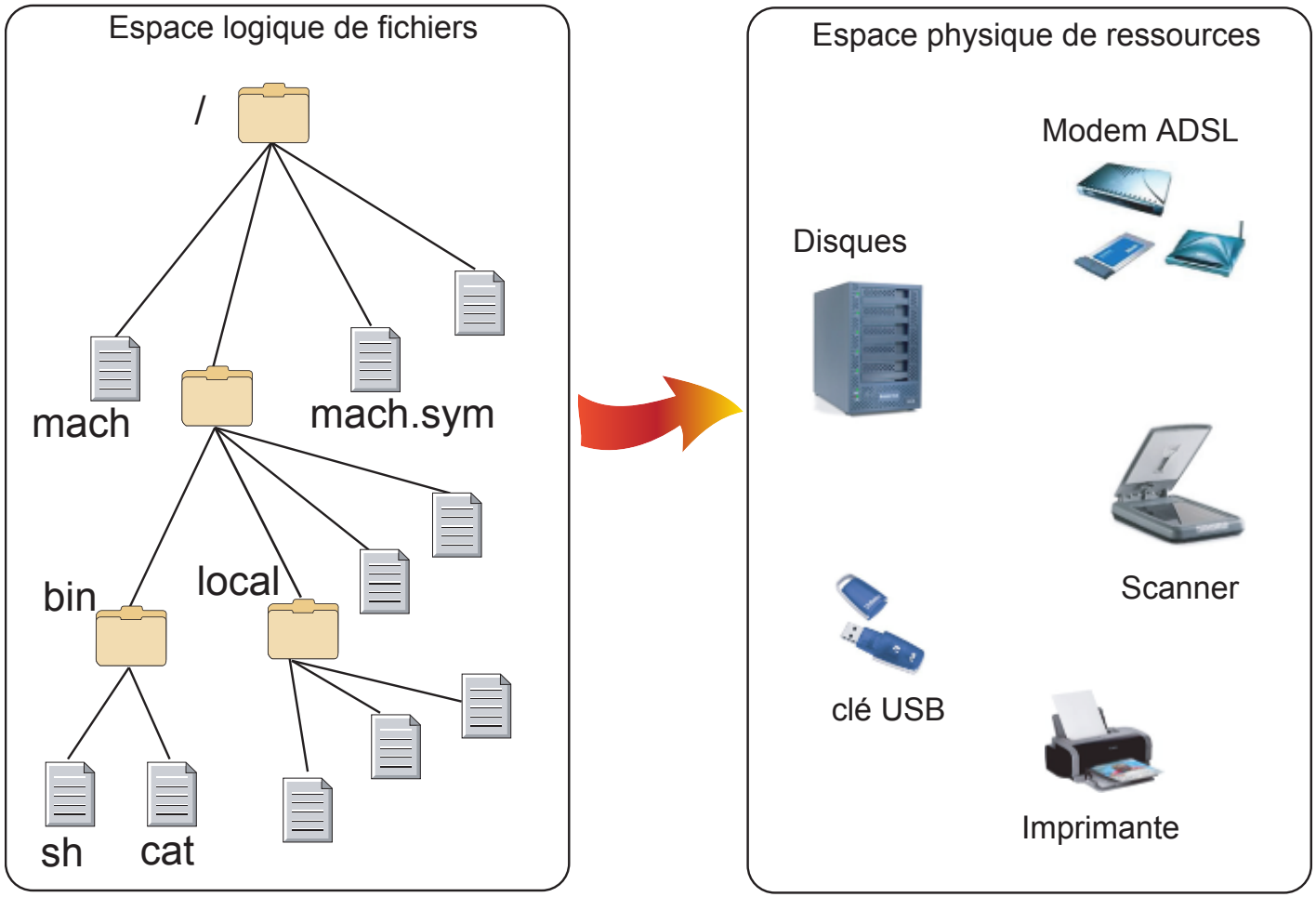
Deuxième partie

Systèmes de fichiers répartis

plan

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
- ...

Transparence de la localité

Une étape : le réseau est la "racine"

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

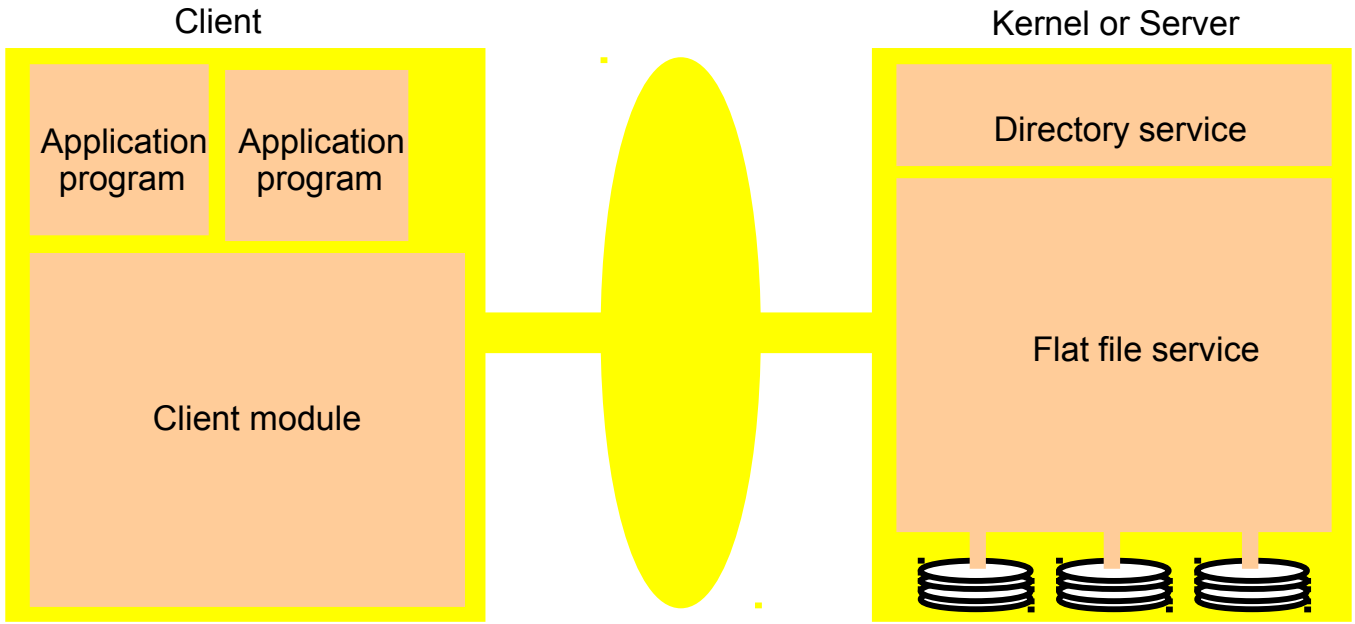
Disparition des noms de sites : NFS

- 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

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

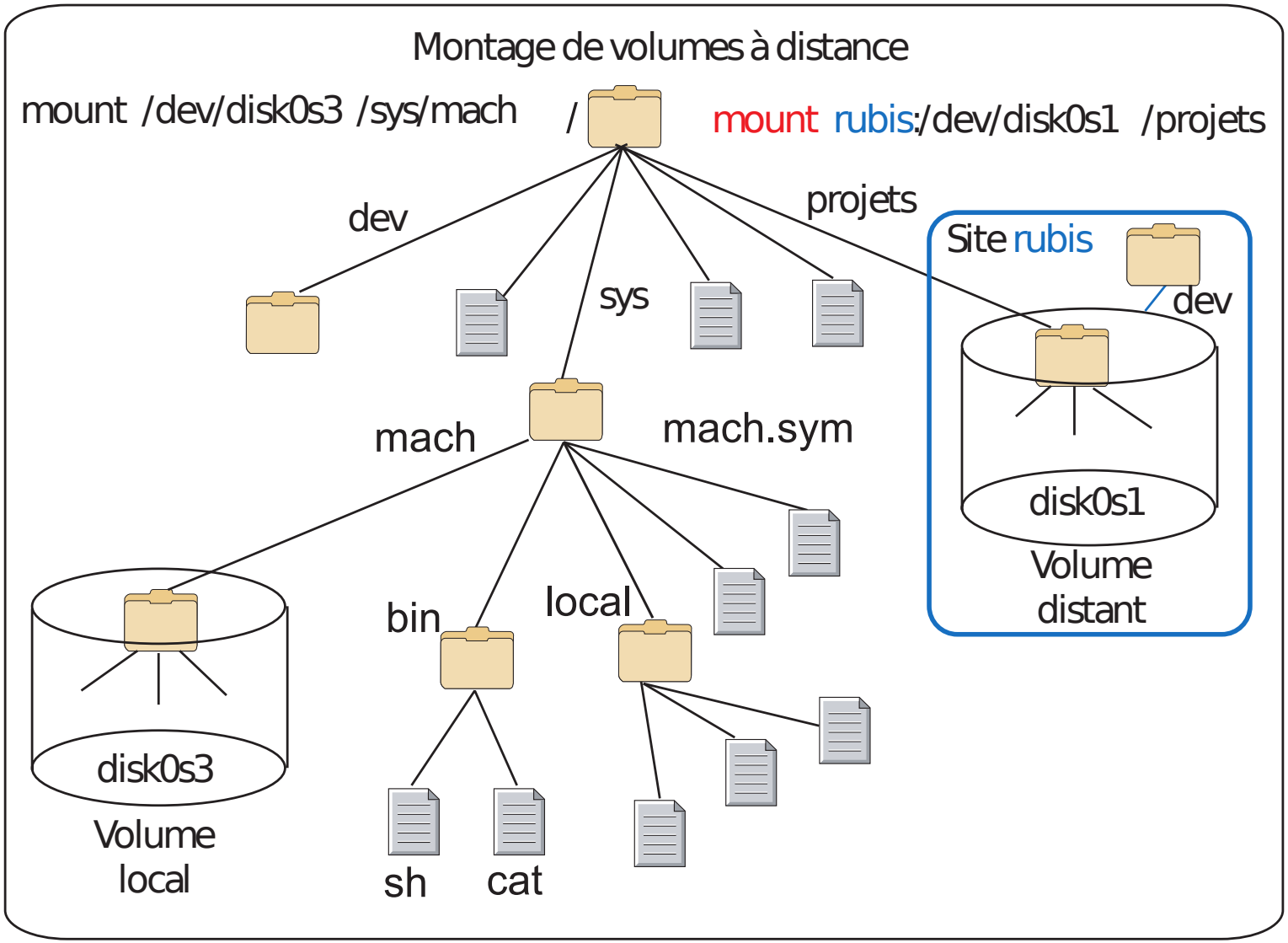
- Invalidation les copies clientes par le serveur lors de chaque création de version
- Le contrôle du parallélisme : garantir un seul rédacteur
- Une autre idée : Fichiers à version unique (immuable)

Plan

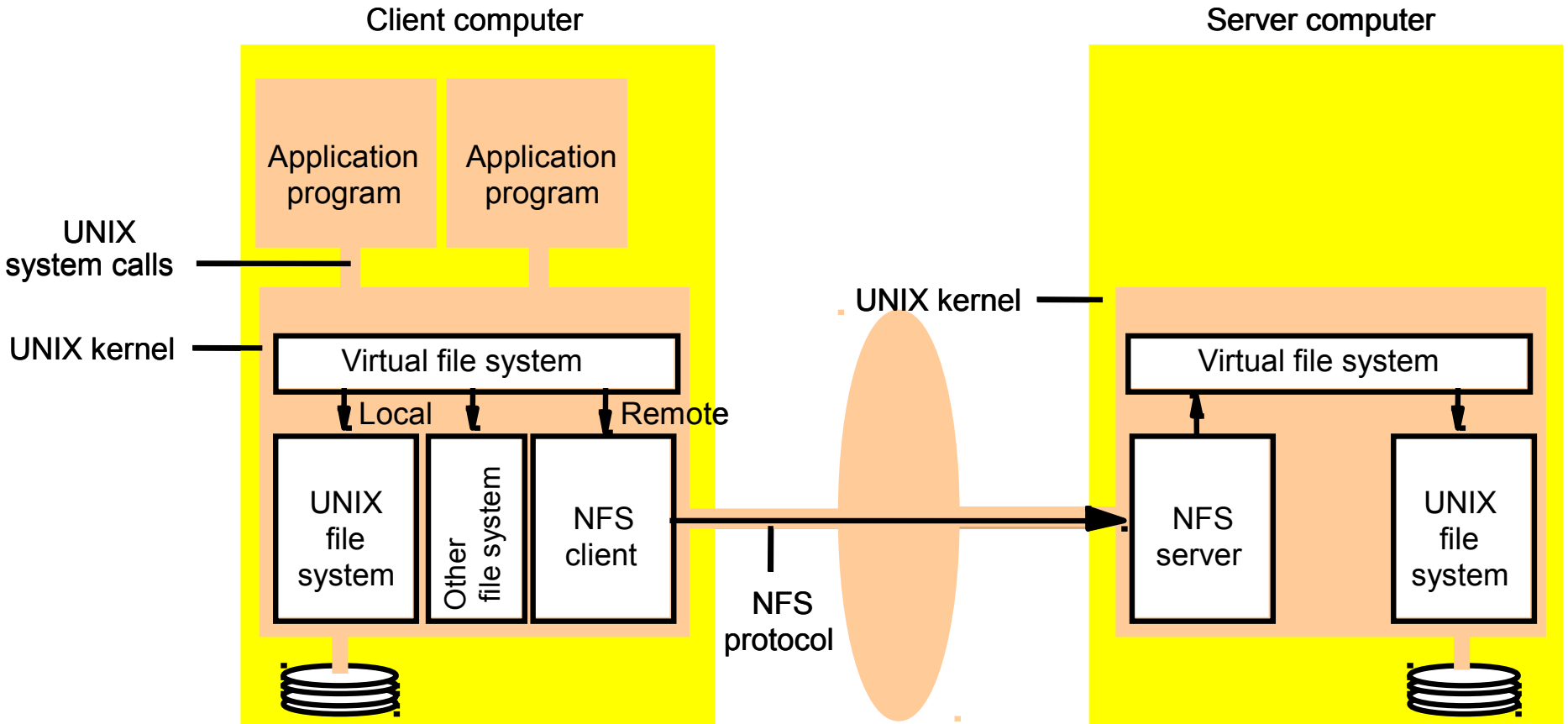
NFS (Network File System) : Principes

- 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 $\ll$ au moins une fois $\gg$
- 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 $\rightarrow$ 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 ≡ 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 de gestion de versions décentralisé (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

Plan

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

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

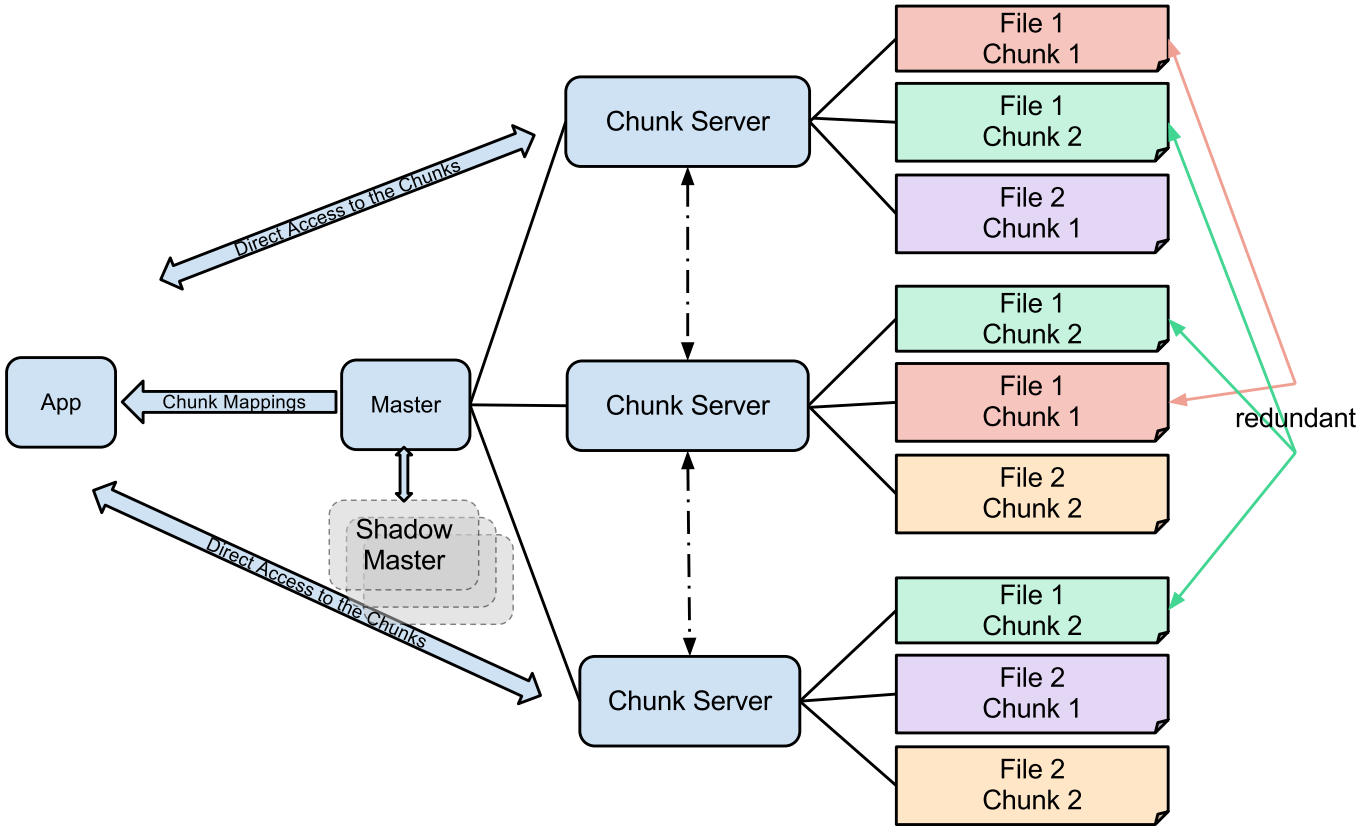
Opérations

- 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

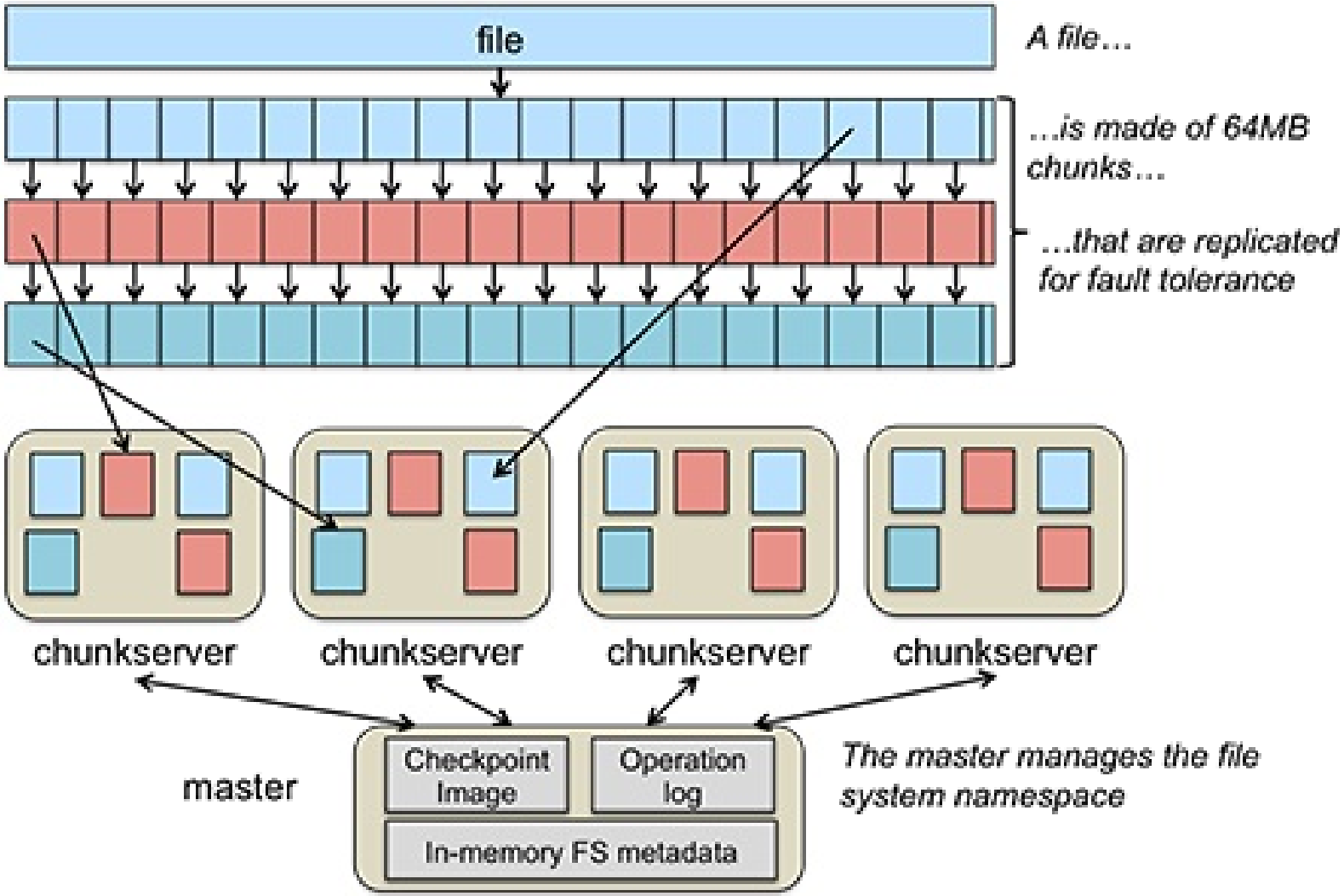
GFS cluster



- 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

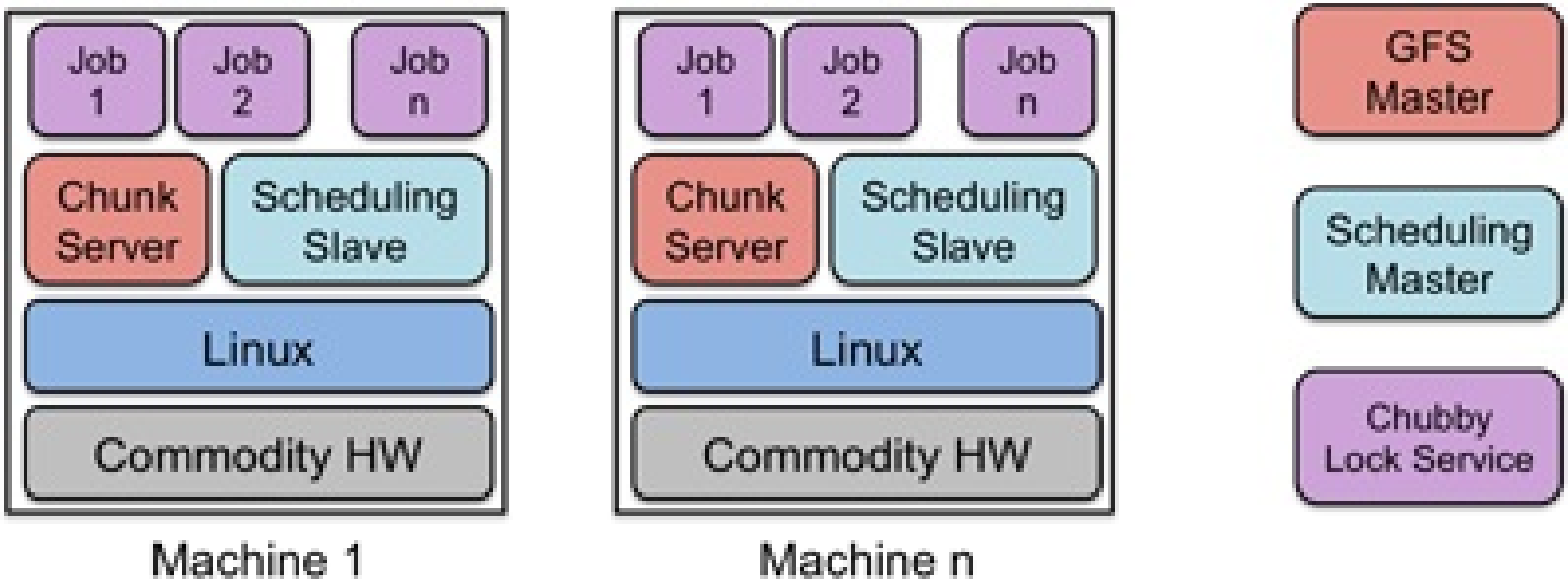
1. Phase 0 : demande d'écriture auprès du maître $\Rightarrow$ un chunk responsable + un nouveau numéro de version
2. Phase 1 : envoi des données :
   - Le client envoie ses données à écrire au serveur de chunk responsable
   - Ce serveur propage à l'un des réplicas, qui propage au suivant, etc
   - 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

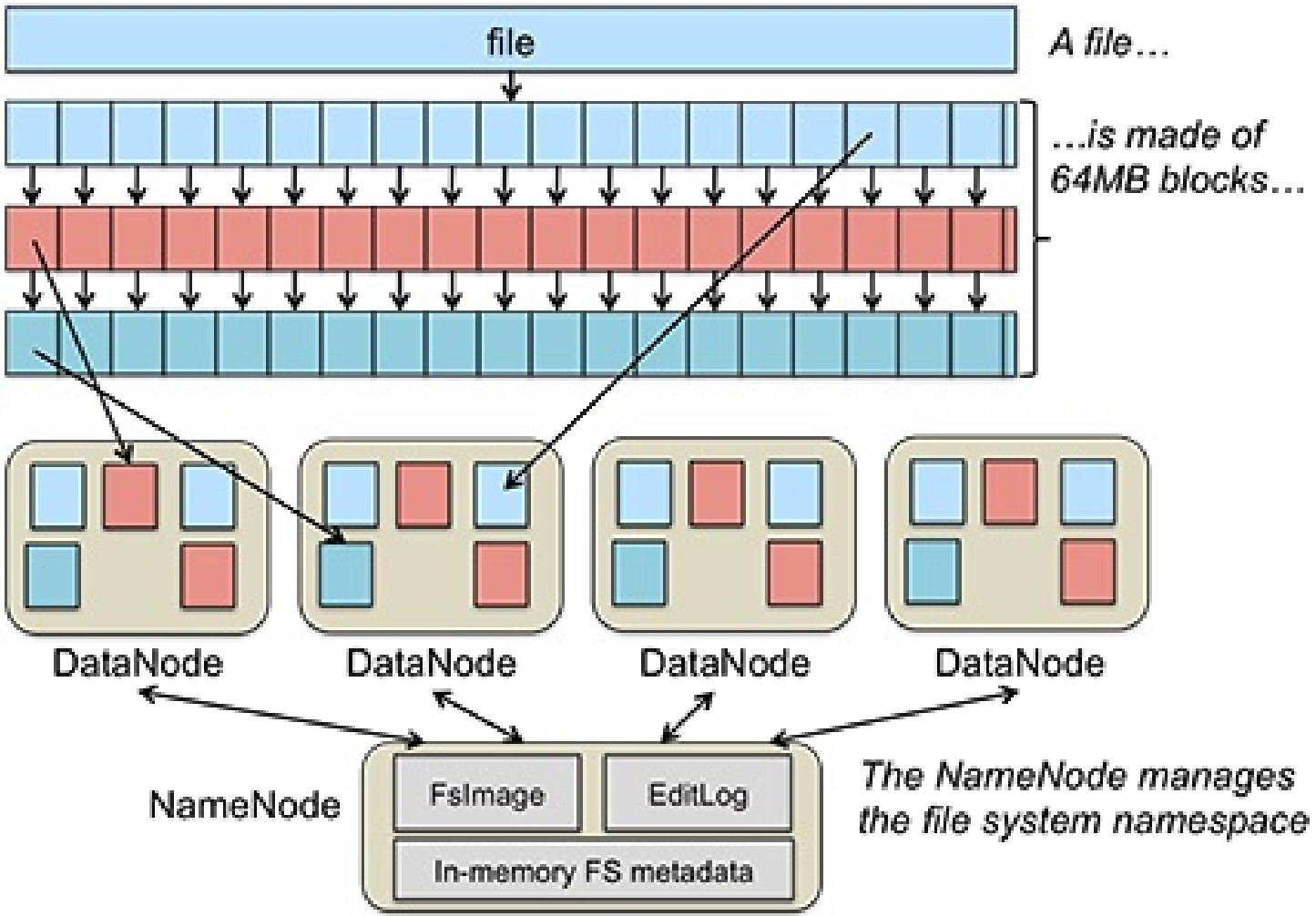
⇒ patron *mapreduce* :

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)

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.