Cohérence et réplication pour les données réparties

Module 9

Michel Dagenais Polytechnique Montréal

Sommaire

Sommaire

Cohérence et réplication de données réparties

- Cohérence (consistency): refléter sur la copie d'une donnée les modifications intervenues sur d'autre copies de cette donnée.
- En anglais:
  - Coherence, en cas de plusieurs modifications concurrentes sur une donnée, la même valeur finale rejoindra éventuellement toutes les copies.
  - Consistency, l'ordre entre plusieurs modifications concurrentes sur différentes données sera vu de manière cohérente (plus ou moins stricte selon le modèle) par les clients de toutes les copies.
- Les modèles de cohérence sont utilisés pour les données réparties ou répliquées: bases de données, systèmes de fichiers, cache...

Modèles de cohérence

- Cohérence stricte: la mise à jour est vue en même temps sur toutes les copies (e.g. invalider la valeur actuelle sur toutes les copies, propager la nouvelle valeur sur toutes les copies).

- Cohérence séquentielle: l'ordre des modifications est le même sur toutes les copies.

- Cohérence causale: l'ordre des modifications reliées causalement (e.g. même origine) est le même sur toutes les copies.

Exercice cohérence

Un système A utilise des serveurs répliqués, et les clients peuvent lire ou écrire sur chaque serveur.
Sur le système B, les clients doivent écrire sur un serveur central mais peuvent lire de serveurs secondaires.
Un serveur qui reçoit une écriture la propage aux autres serveurs.
Les messages des clients sont numérotés et propagés dans l'ordre.

Dans chaque cas, dites s'il s'agit de cohérence stricte, cohérence séquentielle ou cohérence causale.

Exercice cohérence - SOLUTION

Le système A offre une cohérence causale. Les écritures d'un même client restent dans l'ordre mais les écritures de différents clients sur différents serveurs peuvent être propagées aux autres serveurs dans un ordre différent.

Le système B offre une cohérence séquentielle, puisque tout est linéarisé à travers un serveur central.

Les transactions

- Dans la plupart des cas, on doit s'assurer de la cohérence d'un groupe de modifications (transaction) plutôt qu'une modification seule (e.g. transactions dans les bases de données, opérations sur les données et métadonnées pour écrire dans un fichier).

- La cohérence sera donc étudiée principalement sous l'angle des transactions.

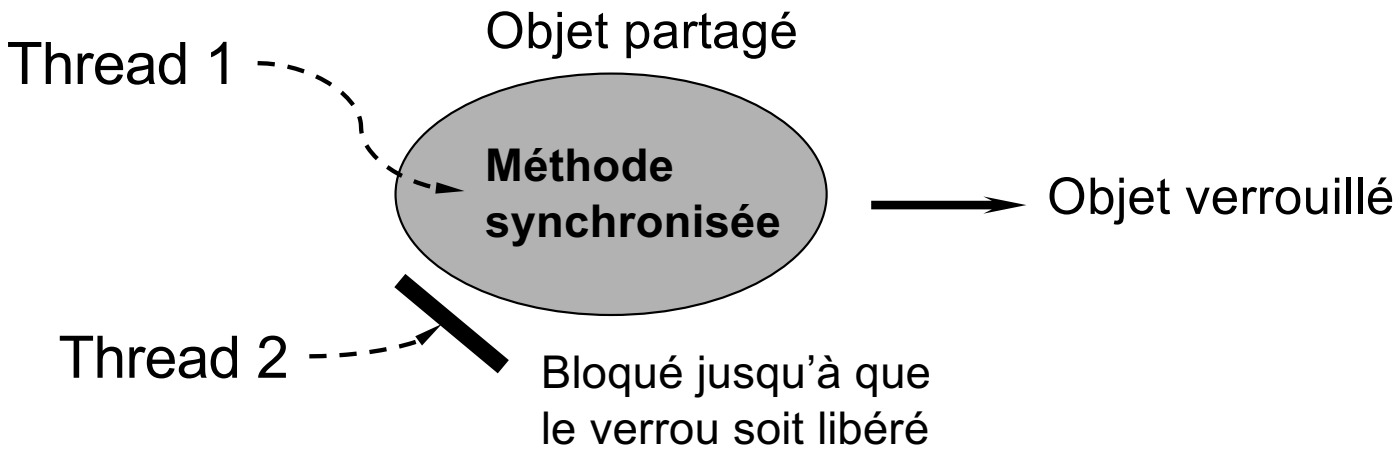
- Transaction = une séquence d'opérations que le serveur exécute d'une façon atomique même en présence d'accès simultanés par plusieurs clients et des pannes des serveurs

- Assurer le partage sécuritaire et cohérent des données des serveurs par plusieurs clients

Synchronisation simple (sans transaction)

- Opérations atomiques au niveau du serveur.
- Utilisation de plusieurs fils d'exécution au niveau du serveur.
- Les opérations des clients peuvent s'exécuter simultanément.
- Un verrou assure qu'un seul thread accède à un objet à un instant donné.

- **Synchronisation sans transactions (suite)**



Utilisation de synchronisation ⟶ Opérations atomiques

Coopération entre les objets : les clients utilisent le serveur pour partager certains objets

- **Synchronisation sans transactions (suite)**

Méthodes wait() et notify() permettent aux threads de communiquer

- **wait :** thread faisant appel à wait sur un objet, sera suspendu, permettant ainsi à un autre thread d'exécuter une méthode sur cet objet
- **notify :** un thread informe tout autre thread attendant l'accès à l'objet qu'il a mis à jour cet objet

Exemple : Objet file partagé

Méthodes : *first* (supprime et retourne le premier élément),

Si file vide **alors** wait();

append (ajoute un objet à la fin de la file)

ajouter un objet;
notify();

- **Exemple : Banque**

Chaque compte est représenté par un objet distant

Interface Account : fournit les opérations pouvant être exécutées par les clients

deposit(amount)	: deposit amount in the account
withdraw(amount)	: withdraw amount from the account
getBalance() → amount	: return the balance of the account
setBalance(amount)	: set the balance of the account to amount

Interface Branch : fournit les opérations pour les succursales de cette banque

create(name) → account	: create a new account with a given name
lookUp(name) → account	: return a reference to the account with the given name
branchTotal() → amount	: return the total of all the balances at the branch

Exemple : transaction bancaire

Transaction T:

a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Doit être exécutée de façon atomique :

- Pas d'interférences avec d'autres opérations simultanées déclenchées par d'autres clients

- Soit que toutes les opérations sont exécutées ou elles n'ont aucun effet si les serveurs tombent en panne

- **Transaction atomique :** deux propriétés à satisfaire

  - **Tout ou rien (All-or-nothing) :** soit la transaction est exécutée (ses effets sont alors permanents), ou bien elle n'a aucun effet (même en cas de panne du serveur)

  - **Isolation :** chaque transaction est exécutée sans interférences avec d'autres transactions (effets intermédiaires invisibles aux autres transactions)

Sommaire

- **Propriétés des transactions (ACID) :**

Atomicité : Les effets d'une transaction ne sont rendus visibles que si toutes ses opérations peuvent être réalisées. Sinon aucun effet n'est produit par la transaction

Cohérence : les effets d'une transaction doivent satisfaire les contraintes de cohérences de l'application (la transaction transforme l'état cohérent du système en un autre état cohérent)

Isolation : les effets d'une transaction sont identiques à ceux qu'elle produirait si elle s'exécutait seule dans le système

Durabilité : les effets d'une transaction terminée ne peuvent être détruits ultérieurement par une quelconque défaillance

- **Objets récupérables (recoverable objects)**

  - objets pouvant être récupérés à la suite d'une panne de serveur

  - Objets sauvegardés dans :

    • Mémoire volatile (RAM) : contenus perdus en cas de pannes
    • Mémoire stable (disque) : contenus préservés

  → Serveur : doit stocker suffisamment d'informations pour assurer la récupération des objets

  - Mécanismes de support afin de supporter les deux propriétés :

  Tout ou rien : Les objets doivent être récupérables à la suite d'une panne du serveur (journal de transaction)

  Isolation : Les transactions doivent être suffisamment synchronisées (exécutées en série ou simultanément, mais équivalentes à une exécution en série)

Coordonnateur de transaction

- Création: le coordonnateur crée la transaction et lui affecte un identificateur unique qui sera associé à toutes les opérations de cette transaction

- Opérations : le coordonnateur maintient la liste des objets ou processeurs impliqués dans les différentes opérations de la transaction.

- Fin: la transaction est confirmée (commit) ou annulée (abort) par le client, le coordonnateur accepte ou annule une transaction confirmée

- Une transaction est complétée à la suite de la coopération entre le programme du client, les objets récupérables et le coordinateur

Les divers scénarios d'une transaction

Confirmée	Annulée (client)	Annulée (serveur)
Begin Transaction	Begin Transaction	Begin Transaction
opération	opération	opération
opération	opération	opération
opération	opération	opération
⋮	⋮	⋮
Commit Transaction	Abort Transaction	Commit Transaction
Success: accept	*Success: abort*	*Server error: abort*

Actions possibles à la suite à d'une panne du client

- Le client redémarre et a oublié toute transaction qu'il avait initiée et qui était en cours.
- Si le serveur n'a plus de nouvelles d'un client après un certain délai (timeout) il annule la transaction.
- Si le client n'était pas en panne mais son réseau était trop lent et le délai expire, le serveur lui refusera la transaction puisqu'elle est annulée (ou répondra que la transaction n'existe plus).

Actions possibles à la suite à d'une panne du serveur

- Le serveur redémarre et a oublié toutes les transactions en cours (non complétées) qui sont conséquemment annulées.
- Une procédure de recouvrement (e.g. lecture du journal des transactions) permet au serveur de revenir à un état cohérent des objets (valeurs produites par les plus récentes transactions complétées).
- Un client ne recevant plus de réponse du serveur après un certain délai abandonne sa requête.
- Si le serveur a redémarré le client se fera répondre que la transaction n'existe plus.
- Si le serveur plante avant d'avoir répondu au client si la transaction est acceptée, le client reste dans l'incertitude. Il demandera le statut de la transaction après le redémarrage et se fera confirmer qu'elle est acceptée ou répondre que la transaction n'existe plus.

- **Problèmes associés au contrôle de la concurrence**

  - Deux grands problèmes des transactions concurrentes sont :
    - Perte d'une mise à jour (lost update problem)
    - Interrogations incohérentes (inconsistent retrievals problem)

  - **Hypothèse :** les opérations *deposit*, *withdraw*, *getBalance*, et *setBalance* sont des opérations synchronisées (leurs effets sont atomiques)

- **Problème de la perte d'une mise à jour**

Exemple :

Solde initial des comptes : A=100, B=200, C=300

Transactions : T (transfère un montant de A à B),
U (transfère un montant de C à B)

Montant transféré = 10% du solde de B

Effet net des deux transactions : B = $200^{T}\rightarrow 220^{U}\rightarrow 242$

- **Problème de la perte d'une mise à jour (suite) :**

Transaction T :		**Transaction U :**	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
balance = b.getBalance();	**$200**		
		balance = b.getBalance();	**$200**
		*b.setBalance(balance*1.1);*	**$220**
*b.setBalance(balance*1.1);*	**$220**		

- **Problème d'interrogations incohérentes :**

Exemple :

Solde initial des comptes : A=200, B=200

Transactions : V (transfère 100$ de A vers B),
W (calcule la somme de tous les comptes de la banque)

Transaction V :		**Transaction W :**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	**$100**		
		total = a.getBalance()	**$100**
		total = total+b.getBalance()	**$300**
		total = total+c.getBalance()	
		⋮	
b.deposit(100)	**$300**		

Sérialisation des transactions

- Une transaction exécutée seule ne présente pas de problème.
- L'exécution de toutes les transactions une par une, T_1, T_2, …, T_n, permet aussi d'assurer les propriétés requises
- Entrelacement des transactions équivalent à une exécution en série, pour un ordre des transactions quelconque, est aussi acceptable.

Sérialisation des transactions

Problème de perte de mise à jour :

- **survient lorsque deux transactions lisent l'ancienne valeur d'un objet, et ensuite l'utilisent pour calculer la nouvelle valeur**
- **Solution :** l'une des transactions est exécutée avant l'autre

Transaction T :		**Transaction U :**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
balance = b.getBalance()	**$200**		
*b.setBalance(balance*1.1)*	**$220**		
		balance = b.getBalance()	**$220**
		*b.setBalance(balance*1.1)*	**$242**

Sérialisation des transactions

Problème d'interrogations incohérentes :

- survient lorsque deux transactions, dont l'une fait une mise à jour et l'autre une interrogation, s'exécutent simultanément
- **Solution :** transaction d'interrogation est exécutée avant ou après l'autre

Transaction V :		**Transaction W :**	
a.withdraw(100);		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	**$100**		
b.deposit(100)	**$300**		
		total = a.getBalance()	**$100**
		total = total+b.getBalance()	**$400**
		total = total+c.getBalance()	
		...	

Contraintes pour sérialiser les transactions

Conflits d'opérations

- Deux opérations sont en conflits si leurs effets combinés dépendent de l'ordre dans lequel elles sont exécutées

- Lecture-Lecture, pas de conflit.
-
- Lecture-Écriture, conflit, le résultat de la lecture est affecté par l'écriture de la variable à lire.

- Écriture-Écriture, conflit, le résultat des lectures subséquentes dépend de l'ordre des deux écritures sur une même variable.

Sérialisation des transactions

Deux transactions sont *sérialisables si* toutes les paires d'opérations en conflits des deux transactions sont exécutées dans le même ordre sur tous les objets qu'elles accèdent

Exemple : T : x = read(i); write(i, 10); write(j, 20);
U : y = read(j); write(j, 30); z = read(i);

Transaction T :	**Transaction U :**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

L'ordonnancement n'est pas équivalent à une exécution en série : paires d'opérations en conflits ne sont pas exécutées dans le même ordre sur i et j

Ordonnancement équivalent à une exécution en série : nécessite l'une des conditions suivantes :

1) T accède à i avant U et T accède à j avant U
2) U accède à i avant T et U accède à j avant T

Exercice

Un serveur gère plusieurs valeurs a1, . . . an. Ce serveur offre deux opérations à ses clients: value = Read(i), Write(i, value). Les transactions T et U sont définies comme suit.

T: x=Read(j); y=Read(i); Write(j,44); Write(i,33);

U: x=Read(k); Write(i,55); y=Read(j); Write(k,66);

Soit a_j la valeur initiale de j, a_i la valeur initiale de j, a_k la valeur initiale de j.

Donnez une sérialisation équivalente pour le cas T procède entièrement avant U, et une une sérialisation équivalente pour le cas U procède entièrement avant T.

Exercice

Si T procède entièrement avant U, le résultat final sera:

- T: x=Read(j); y=Read(i); Write(j,44); Write(i,33);
  Résultat partiel : Tx = a_j, Ty = a_i, j = 44, i = 33.
- U: x=Read(k); Write(i,55); y=Read(j); Write(k,66);
  Résultat final : Ux = a_k, i = 55, Uy = 44, k = 66.

Valeurs sur le disque : j = 44, i = 55, k = 66.

L'entrelacement suivant des opérations produit le même résultat:

Transaction T	Transaction U
x = Read(j)	
y = Read(i)	
	x = Read(k)
Write(j, 44)	
Write(i, 33)	
	Write(i, 55)
	y = Read(j)
	Write(k, 66)

Exercice

Si U procède entièrement avant T, le résultat final sera:

- U: x=Read(k); Write(i,55); y=Read(j); Write(k,66);
  Résultat partiel : Ux = a_k, Uy = a_j, k = 66, i = 55.
- T: x=Read(j); y=Read(i); Write(j,44); Write(i,33);
  Résultat final : Tx = a_j, Ty = 55, j = 44, i = 33.

Valeurs sur le disque : j = 44, i = 33, k = 66.

L'entrelacement suivant des opérations produit le même résultat:

Transaction T	Transaction U
x = Read(j)	
	x = Read(k)
	Write(i, 55)
y = Read(i)	
	y = Read(j)
	Write(k, 66)
Write(j, 44)	
Write(i, 33)	

Approches de contrôle de la concurrence

Ordonnancement équivalent à une exécution en série est retenu comme critère pour la dérivation de protocoles de contrôle de la concurrence

But des approches de contrôle de la concurrence :
- Tentent de sérialiser les transactions durant leurs accès aux objets

Quelques approches :

Verrouillage (Locking) : Exécution immédiate ou différée

- Utilisé par la plupart des systèmes
- Chaque objet est verrouillé par la première transaction qui y accède
- L'objet est déverrouillé au *commit* ou *abort*

Exercice Verrous

Montrez pourquoi lorsqu'une transaction a relâché un verrou elle ne doit plus en obtenir afin de permettre l'équivalence avec la sérialisation?

Soit deux transactions T et U:

- **T:** x = Read(i); Write(j, 44);
- **U:** Write(i, 55); Write(j, 66);

Avec T avant U, le résultat est:

- $x = a_i^0$
- $a_i = 55$
- $a_j = 66$

Si T commence avant U mais les verrous sont relâchés trop tôt, nous pourrions avoir:

Transaction T	Transaction U
Lock *i*	
x = Read(*i*)	
Unlock *i*	Lock *i*
	Write(*i*, 55)
	Lock *j*
	Write(*j*, 66)
	Commit
Lock *j*	Unlock *i*, *j*
Write(j, 44)	
Unlock *j*	
Commit	

Approches de Contrôle de la Concurrence (suite)

Ordonnancement par estampilles : Exécution immédiate, différée ou abandonnée

- Les transactions et les objets ont des estampilles
- Serveur : enregistre le temps d'accès le plus récent de la lecture et de l'écriture de chaque objet et celui de chaque opération
- L'estampille de la transaction est comparée à celle de l'objet pour déterminer si elle peut être exécutée immédiatement, différée ou abandonnée

Recouvrement après l'abandon d'une Transaction

Le serveur doit :

- Sauvegarder les effets des transactions correctement complétées
- Assurer qu'aucun effet des transactions abandonnées ne soit stocké

Problèmes associés à l'abandon de transactions :

- Problème de lectures contaminées (dirty reads)
- Problème d'écritures prématurées (premature writes)

Recouvrement après l'abandon d'une Transaction (suite)

Problème de lectures contaminées : causé par l'interaction entre une opération de lecture dans une transaction et une ancienne opération d'écriture dans une autre transaction

Transaction T	Transaction U	Compte A
Begin Transaction		\$100
b = getBalance(A)		\$100
setBalance(A, $b + 10$)		\$110
	Begin Transaction	\$110
	b = getBalance(A)	\$110
	setBalance(A, $b + 20$)	\$130
Abort Transaction		\$100
	Commit Transaction	\$130

Recouvrement après l'abandon d'une Transaction (suite)

Problème de lectures contaminées (suite) :

- Solution : retarder l'opération *commit* de la transaction U, qui sera plutôt abandonnée

- Nouveau Problème : Abandons en cascades

  → **Solution :**

  - Éviter les abandons en cascade en autorisant aux transactions de lire seulement les objets écrits par des transactions complétées.
  - Dans l'exemple, il faut retarder la lecture contaminée.

Problème d'écritures prématurées :

- Survient si le système de gestion de bases de données restitue, après un abandon, l'ancienne valeur de l'objet par rapport à l'ensemble des opérations d'écriture

Recouvrement après l'abandon d'une Transaction

Problème d'écritures prématurées :

Transaction T	Transaction U	Compte A
Begin Transaction		$100
setBalance(A,$105)		$105
	Begin Transaction	$105
	setBalance(A, $110)	$110
Abort Transaction		
(restore A)		$100
	Abort Transaction	
	(restore A)	$105

Effet net : A = 105$ et non pas 100$

Solution aux problèmes de lectures contaminées et d'écritures prématurées :

Retarder les lectures et les écritures jusqu'à ce que toutes les transactions antérieures, qui modifient le même objet, soient complétées ou abandonnées

Recouvrement après l'abandon d'une Transaction

Solution aux problèmes de lectures contaminées et d'écritures prématurées (suite) :

- Transaction : possède ses propres versions provisoires (tentative) des objets qu'elle met à jour
- Écritures : faites sur les versions provisoires
- Lectures : faites soit à partir des versions provisoires, soit à partir des versions permanentes
- Versions provisoires : transférées dans les versions permanentes des objets seulement lorsque la transaction est complétée en une seule étape
- Pendant le transfert toutes les autres transactions n'ont pas accès aux objets modifiés

Sommaire

Transactions Imbriquées

Transaction imbriquée = transaction qui contient des transactions qui peuvent elles-mêmes être imbriquées

Transaction imbriquée à un niveau de la hiérarchie : n'est complétée que si son parent complète son exécution

Avantages :

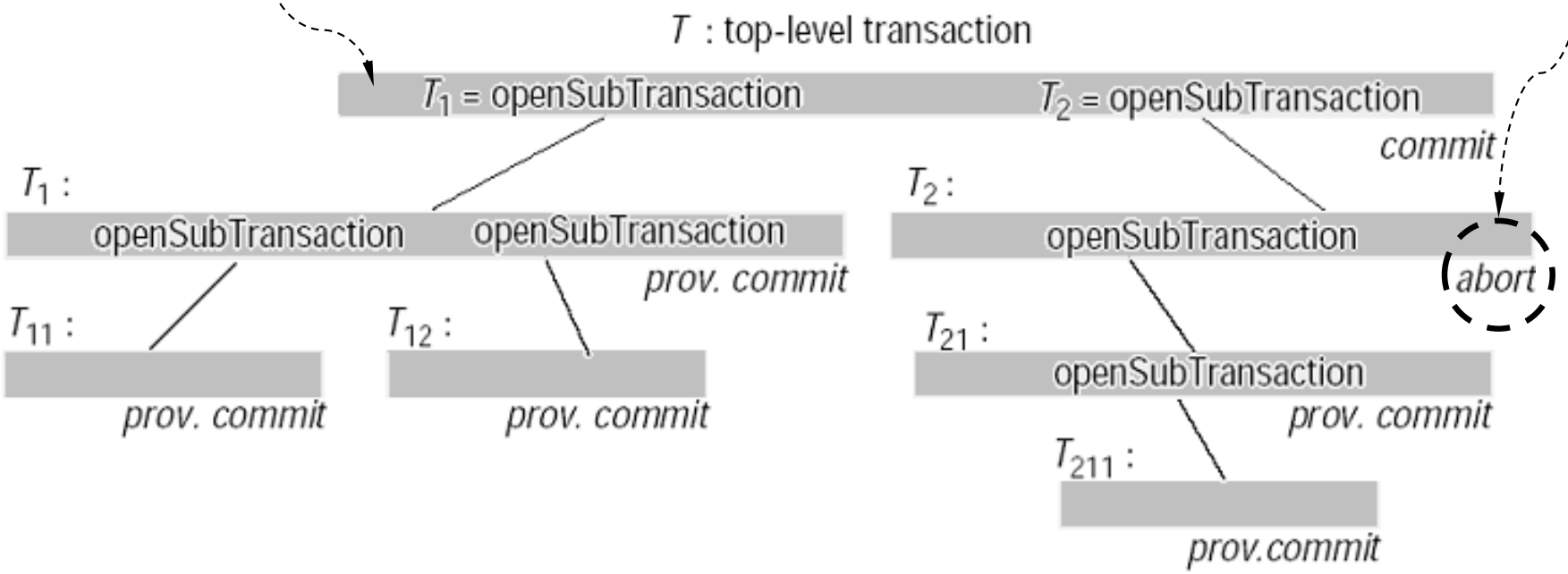
- Accroître le niveau de concurrence en permettant aux transactions imbriquées d'un même niveau dans la hiérarchie d'être exécutées simultanément

- Permet aux transactions imbriquées d'être complétées ou abandonnées indépendamment les unes des autres

- **Transactions Imbriquées (suite)**

Sous-transactions du même niveau, e.g. T_1 et T_2, peuvent s'exécuter simultanément, mais leurs accès aux objets partagés sont sérialisés (verrouillage)

Sous-transaction est abandonnée : le parent peut choisir une autre sous-transaction pour compléter sa tâche



Transactions Imbriquées (suite)

Exemple :

- Transaction T : livre un message à une liste d'abonnées

- Sous-transaction T_i : livre le message à un abonné

- Si une ou plusieurs sous-transactions sont abandonnées alors :

1) T sauvegarde quelles sous-transactions ont été abandonnées
2) T termine correctement (commit)
3) Démarre une autre transaction qui essayera de livrer les messages qui n'ont pas été envoyer la première fois

Transactions Imbriquées (suite)

Règles pour compléter les transactions imbriquées :

1) Une transaction ne se termine correctement ou n'est abandonnée, que si toutes ses sous-transactions sont complétées

2) Quand une sous-transaction se termine, elle prend une décision indépendante : commit provisoire ou abort. Si la décision est abort, elle est finale

3) Si la transaction parente est abandonnée, alors toutes ses sous-transactions sont abandonnées

Exemple : T_2 abort $\rightarrow$ T_{21} et T_{211} doivent être abandonnées

Transactions Imbriquées (suite)

Règles pour compléter les transactions imbriquées (suite) :

4) Quand une sous-transaction est abandonnée, le parent peut décider d'abandonner ou non

   Exemple : T décide commit même si T_2 a été abandonnée

5) Si la transaction de haut-niveau se termine correctement, alors toutes ses sous-transactions qui ont provisoirement complétées peuvent se terminer aussi, à condition qu'aucun de leurs ancêtres n'est abandonné

   Exemple : T commit $\rightarrow$ T_1, T_{11} et T_{12} commit

   T_{21} et T_{21} ne seront pas complétées puisque leur parent T_2 a été abandonné

Contrôle de la concurrence

Objectifs :

- Concevoir des techniques de contrôle de qui permettent de gérer les conflits de différentes transactions sur les mêmes objets
- Éviter les incohérences au niveau des transactions et des objets (ressources)
- Ordonner les transactions de façon que leur exécution soit équivalente à une transaction en série

Contrôle de la concurrence

- **Verrous (locks) :** utilisé pour ordonner les transactions, qui accèdent aux mêmes objets, selon l'ordre d'arrivée de leurs opérations aux objets. Utilisé par la plupart des systèmes.
  - Chaque objet est verrouillé par la première opération qui y ,accède.
  - L'objet est déverrouillé au commit ou abort.

- **Contrôle optimiste de la concurrence :** permet aux transactions de s'exécuter jusqu'à qu'elles soient prêtes à être complétées

  - Avant d'être complétée, le serveur vérifie si elle n'a pas exécuté des opérations sur des objets qui sont en conflit avec les autres opérations des autres transactions simultanées.
  - Si de tels conflits existent, le serveur abandonne la transaction et le client peut la redémarrer.

- **Ordonnancement par estampilles :** utilise les estampilles pour ordonner les transactions, qui accèdent aux mêmes objets, suivant leurs temps d'invocation

Verrouillage

Utilisation de verrous exclusifs :

- Un verrou permet de réserver l'accès unique à un objet (ressource)
- Si un client demande l'accès à un objet déjà verrouillé par une autre transaction, alors sa requête doit être suspendue et le client doit attendre jusqu'à que l'objet soit déverrouillé

Types de verrou :

- Verrous de lecture : avant une opération de lecture d'un objet, e serveur pose un verrou de lecture sur l'objet
- Verrous d'écriture : avant une opération d'écriture d'un objet, le serveur pose un verrou d'écriture sur l'objet

Exemple de verrouillage

Transaction T	Verrous T	Transaction U	Verrous U
Begin Transaction			
bal = getBalance(B)	Lock B		
setBalance(B, bal*1.1)		Begin Transaction	
withdraw(A, bal/10)	Lock A	bal = getBalance(B)	wait for B
Commit		⋮	⋮
Success	Unlock A, B	⋮	⋮
		bal = getBalance(B)	Lock B
		setBalance(B, bal*1.1)	
		withdraw(C, bal/10)	Lock C
		Commit	
		Success	Unlock B, C

Verrouillage

Granularité :

- Verrou complet ou séparé lecture et écriture.
- Verrou posé sur un objet
- Verrou posé sur un groupe d'objets
- Verrou posé sur tous les objets (non acceptable comme contrainte, puisqu'une seule transaction peut s'exécuter à la fois)

Inconvénient : verrou exclusif réduit fortement la concurrence (deux opérations en lecture ne sont jamais en conflit)

Solution (*many reader/single writer*) : utiliser un schéma de verrouillage qui contrôle l'accès aux objets de telle manière que :

- plusieurs transactions simultanées peuvent lire un objet, ou
- une seule transaction met à jour un objet, mais pas les deux en même temps

Exercice verrouillage

Un groupe de 21 processus, $p1$ à $p21$, utilisent un serveur central d'exclusion mutuelle. Les 21 processus demandent en même temps (e.g., à 16h00m00.000s) un même verrou $v1$. i) Que peut-on dire de l'ordre dans lequel ces 21 processus obtiendront le verrou? ii) Combien de messages seront échangés au total pour que tous ces processus obtiennent éventuellement le verrou demandé ?
On suppose qu'il n'y a pas de message perdu.

Exercice verrouillage (SOLUTION)

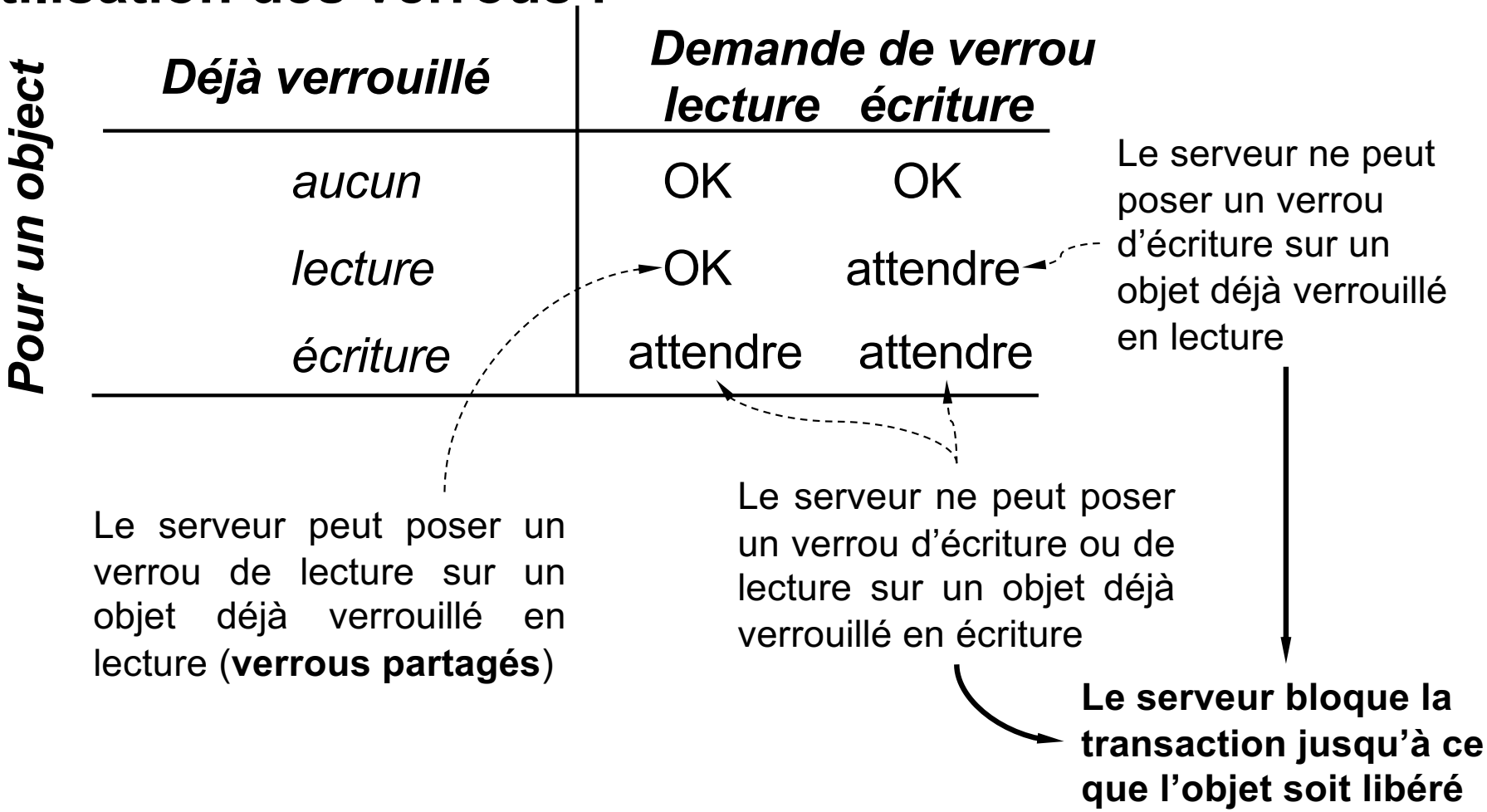
i. Si les horloges ne sont pas bien synchronisées, les demandes ne seront pas émises exactement en même temps, et l'ordre dans lequel elles sont envoyées dépendra de leur décalage de temps. Même si tous les processus demandent le verrou en même temps, ces requêtes seront sérialisées sur le réseau et arriveront dans un certain ordre, un peu aléatoire, au serveur qui pourra les traiter séquentiellement.

ii. Un message est requis pour qu'un processus demande le verrou, il reçoit un message de réponse du serveur qui le lui accorde, et le client enverra finalement un message lorsqu'il en a terminé avec le verrou. Le processus suivant en ligne, qui avait déjà envoyé une demande, recevra alors un message du serveur qui lui accorde le verrou. On voit donc qu'il faut 3 messages pour chaque client (demande, obtention, cession). Le nombre total de messages échangés est donc de 21 x 3 = 63 messages. Si des accusés de réception sont utilisés, car le délai n'est pas facilement prévisible, soit pour obtenir la réponse suite à une demande, ou soit pour que le client relâche le verrou après son obtention, alors le nombre des messages pourrait être doublé à 126.

Le gestionnaire de verrous

- Maintient la table de verrous pour les objets d'un serveur.
- Entrée pour chaque verrou:
  - objet associé au verrou;
  - type de verrou (écriture ou lecture);
  - identificateurs des transactions détenant le verrou (un seul pour écriture);
  - liste des transactions en attente du verrou.

Utilisation des verrous :



Peut-on garantir qu'une transaction qui demande un verrou en écriture va l'obtenir dans un moment donné (dans le futur) ?

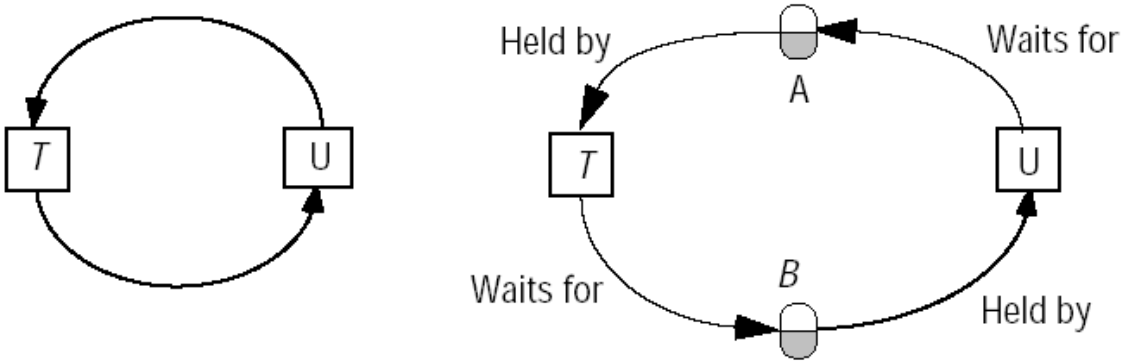
Interblocage (deadlock)

- État anormal d'un ensemble d'au moins deux transactions
- Caractérisé par le fait que chaque transaction est bloquée et ne peut être débloquée que par une autre transaction de l'ensemble

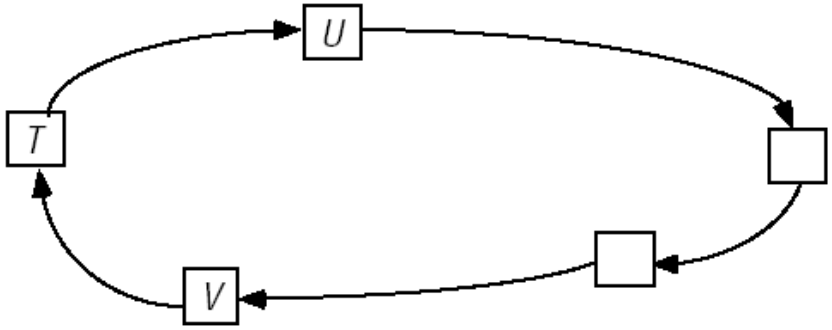
Transaction T	Verrous T	Transaction U	Verrous U
deposit(A, 100)	Lock A		
		deposit(B, 200)	Lock B
withdraw(B, 100)	Wait for B		
⋮	⋮	withdraw(A, 200)	wait for A
⋮	⋮	⋮	⋮

Interblocage - Graphe de dépendance

Dépendance directe : T attend que U libère le verrou et vice-versa.



Dépendance indirecte : T attend que U libère le verrou



Interblocage - exemple

Transaction T

```
read(C);
write(C);   (WAIT)
```

Transaction U

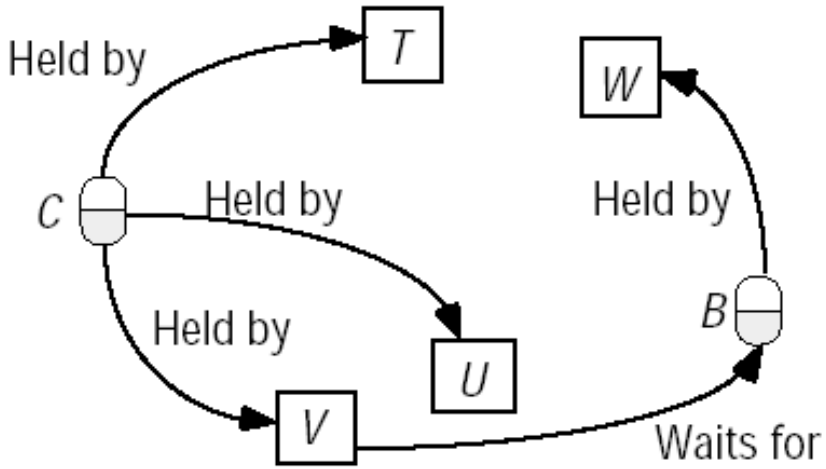
```
read(C);
```

Transaction V

```
read(C);
write(B);   (WAIT)
```

Transaction W

```
write(B);
write(C);   (WAIT)
```



Prévention d'interblocage

- **Solution 1 :** verrouiller tous les objets utilisés par la transaction dès le début
  - Génère un verrouillage prématuré, réduit la concurrence.
  - Parfois, il est impossible de prédire dès le début quels objets seront utilisés (cas d'applications interactives).

- **Solution 2 :** verrouiller les objets dans un ordre prédéfini (peu de concurrence)
  - Pour ordonner, il faut aussi prédire les objets utilisés.

Détection d'interblocage

Détecter les cycles dans le graphe de dépendance et abandonner une transaction appartenant à chacun des cycles

Module responsable de la détection (peut faire partie du gestionnaire de verrous) :

- Maintient une représentation du graphe de dépendance
- Arcs ajoutés ou supprimés dans le graphe lors des opérations sur les verrous
- Vérifie l'existence de cycle périodiquement
- Lorsqu'un cycle est détecté, une des transactions est abandonnée (critères considérés dans ce choix : âge des transactions, nombre de cycles dans lesquels une transaction elle apparaît)

Résolution d'interblocage

- Chaque verrou pris a une période durant laquelle il est non vulnérable
- À la fin de cette période, si une autre transaction attend après le verrou, la transaction qui le possède est annulée et le verrou libéré.
- La transaction qui attendait peut maintenant acquérir le verrou et poursuivre son travail.

Contrôle optimiste de la concurrence (COC)

Approche développée par Kung et Robinson (1981) pour pallier les inconvénients du verrouillage

- Chaque transaction a une version provisoire des objets qu'elle met à jour (Il est aisé d'abandonner une transaction)

- Fonctionnement:
  - Les lectures sont exécutées sur la version réelle (pas de lectures contaminées). Il serait possible mais plus difficile de lire dans les versions provisoires.
  - Les écritures sont exécutées sur la version provisoire.
  - Deux ensembles sont associés avec chaque transaction, les objets lus et les objets mis à jour.
  - Puis validation et mise à jour

COC : Validation des Transactions (suite)

- La validation est une phase atomique, on vérifie si les opérations ont produit des conflits ou pas.
  - Si la transaction est validée, tous les changements seront rendus permanents.
  - Sinon, les versions provisoires sont simplement abandonnées.

Transaction T	Transaction U	Règle
écriture	lecture	U ne doit pas lire un objet écrit par T
lecture	écriture	T ne doit pas lire un objet écrit pat U
écriture	écriture	T ne doit pas écrire un object écrit par U U ne doit pas écrire un objet écrit par T

COC : Validation des Transactions (suite)

Formes de validation

Méthode vers l'arrière, vérifier qu'aucune transaction concurrente terminée n'ait écrit une variable lue par la transaction à valider.

Méthode vers l'avant, vérifier que la transaction courante n'ait pas écrit une variable déjà lue par une transaction concurrente non terminée.

La phase de validation et d'écriture doit être atomique par rapport aux opérations des autres transactions. Tout le reste est bloqué lorsqu'on valide et termine une transaction.

Exercice COC

Une synchronisation optimiste est appliquée aux transactions T et U qui sont actives en même temps. Discutez ce qui arrive dans chacun des cas suivants:

```
T: x = read(i); write(j,44);
U: write(i,55); write(j,66);
```

a. T est prête en premier et la validation en reculant est utilisée?
b. En avançant?
c. U est prête en premier et la validation en reculant est utilisée?
d. En avançant?

Exercice COC - SOLUTION

a Reculant:

- Correct, Read(i) est vérifié avec les écritures de transactions concurrentes terminées (aucune).
- Pour U, aucune lecture n'est effectuée et il n'y a donc pas de conflit.

b Avançant:

- Write(j,44) est vérifié avec les lectures de transactions concurrentes actives (pas de lecture dans U).
- Lorsque U termine, les écritures ne posent pas problème car T est terminée.

c Reculant:

- Correct, U ne fait pas de lecture
- Read(i) est vérifié avec les écritures de transactions concurrentes terminées, un problème est détecté car la valeur a été modifiée par U; T doit être repris.

d Avançant:

- Write(i,55) est vérifié avec les lectures de transactions concurrentes actives, un problème est détecté car T veut lire i; la transaction U est annulée.
- Pour T, Write(j,44) est vérifié avec les lectures de transactions concurrentes actives, ce qui est correct.

Exercice COC

Les transactions T1, T2, T3 et T4 s'exécutent en même temps et leurs opérations de lecture et d'écriture sur des variables (x1, x2, x3, x4, x5 et x6) sont entrelacées. Les lectures d'une transaction sont effectuées sur les versions courantes des variables, et les écritures d'une transaction sont effectuées sur une version provisoire des variables pour la transaction. Lorsque la transaction se termine et est acceptée, la version provisoire des variables écrites par la transaction devient la version courante. Une validation de la cohérence par contrôle optimiste de la concurrence est effectuée pour accepter ou non chaque transaction. Il faut tenir compte des transactions précédentes qui ont été validées (et ignorer celles qui ne l'ont pas été) pour savoir si chacune des transactions est acceptée ou non. Lesquelles des transactions T1, T2, T3 et T4 pourraient être validées, si une validation en reculant était utilisée pour vérifier la cohérence des transactions? Pour chaque transaction non validée, donnez-la ou les variables en conflit.

```
 1 T1: Begin
 2 T1: Read(x5)
 3 T1: Write(x4)
 4 T1: Read(x1)
 5 T2: Begin
 6 T1: Read(x2)
 7 T2: Write(x6)
 8 T3: Begin
 9 T3: Write(x4)
10 T3: Write(x3)
11 T4: Begin
12 T3: Read(x4)
13 T3: Write(x4)
14 T1: End
15 T3: Write(x6)
16 T4: Read(x4)
17 T2: Read(x5)
18 T2: End
19 T4: Read(x5)
20 T4: Write(x6)
21 T3: End
22 T4: Read(x3)
23 T4: Write(x5)
24 T4: End
```

Exercice COC - SOLUTION

```
T1: Begin
T1: Read(x5)
T1: Write(x4)
T1: Read(x1)
T2: Begin
T1: Read(x2)
T2: Write(x6)
T3: Begin
T3: Write(x4)
T3: Write(x3)
T4: Begin
T3: Read(x4)
T3: Write(x4)
T1: End read (x1 x2 x5), write (x4), reculant: validé
T3: Write(x6)
T4: Read(x4)
T2: Read(x5)
T2: End read (x5), write (x6), reculant: validé
T4: Read(x5)
T4: Write(x6)
T3: End read (x4), write (x3 x4 x6), reculant: non validé (x4)
T4: Read(x3)
T4: Write(x5)
T4: End read (x3 x4 x5), write (x5 x6), reculant: non validé (x4)
```

```
 1 T1: Begin
 2 T1: Read(x5)
 3 T1: Write(x4)
 4 T1: Read(x1)
 5 T2: Begin
 6 T1: Read(x2)
 7 T2: Write(x6)
 8 T3: Begin
 9 T3: Write(x4)
10 T3: Write(x3)
11 T4: Begin
12 T3: Read(x4)
13 T3: Write(x4)
14 T1: End
15 T3: Write(x6)
16 T4: Read(x4)
17 T2: Read(x5)
18 T2: End
19 T4: Read(x5)
20 T4: Write(x6)
21 T3: End
22 T4: Read(x3)
23 T4: Write(x5)
24 T4: End
```

Sommaire

Gestionnaire de récupération en cas de panne

- Sauver les items de données sur support de stockage permanent.
- Récupérer les données après une panne du processeur.
- Optimiser la performance de la récupération.
- Libérer l'espace de stockage des données intermédiaires.

Stockage permanent

- Mémoire non volatile avec écriture atomique.
- Disque.
- Disque et cache + pile.
- Disques redondants.
- Disques redondants à distance.

Principe de la récupération en cas de panne

- Toutes les nouvelles valeurs à commettre doivent être mémorisées pour les appliquer si la transaction est acceptée.
- Toutes les anciennes valeurs doivent être conservées au cas où la transaction sera annulée.
- Il faut éventuellement se débarrasser des nouvelles valeurs de transactions annulées, ou des anciennes valeurs de transactions acceptées.

Méthode du journal

- Ajouter à la fin du journal les nouvelles valeurs pour une transaction, et les marques pour le début, la fin ou l'annulation d'une transaction.
- Lorsqu'une transaction est acceptée, une marque est écrite dans le journal et les nouvelles valeurs sont propagées éventuellement à la base de données.
- Pour récupérer: initialiser les données, lire à l'envers le journal, et ajouter toutes les valeurs provenant de transactions acceptées pour les éléments de données qui n'ont pas encore été lus du journal.
- Si la récupération est arrêtée et recommencée, le résultat n'est pas modifié.
- Lorsque le journal devient trop long, une copie de toutes les valeurs acceptées mais non propagées est écrite dans un nouveau journal, puis les entrées de transactions en cours sont ajoutées, et le nouveau journal remplace l'ancien.

Sommaire

Les transactions réparties

- Une transaction répartie peut faire des requêtes à plusieurs serveurs
- Un serveur peut faire appel à d'autres serveurs pour traiter une requête
- Commit ou abort : à la fin de la transaction, tous les serveurs doivent soit compléter, soit abandonner (prennent conjointement la même décision)

Défis des transactions réparties

Contrôle de la concurrence :

- Transactions doivent être sérialisées globalement sachant que chaque serveur sérialise localement les transactions

- Éliminer les interblocages répartis qui peuvent survenir suite à des cycles de dépendances entre différents serveurs

Recouvrement d'abandons de transactions:

- Chaque serveur doit assurer que ses objets soient récupérables

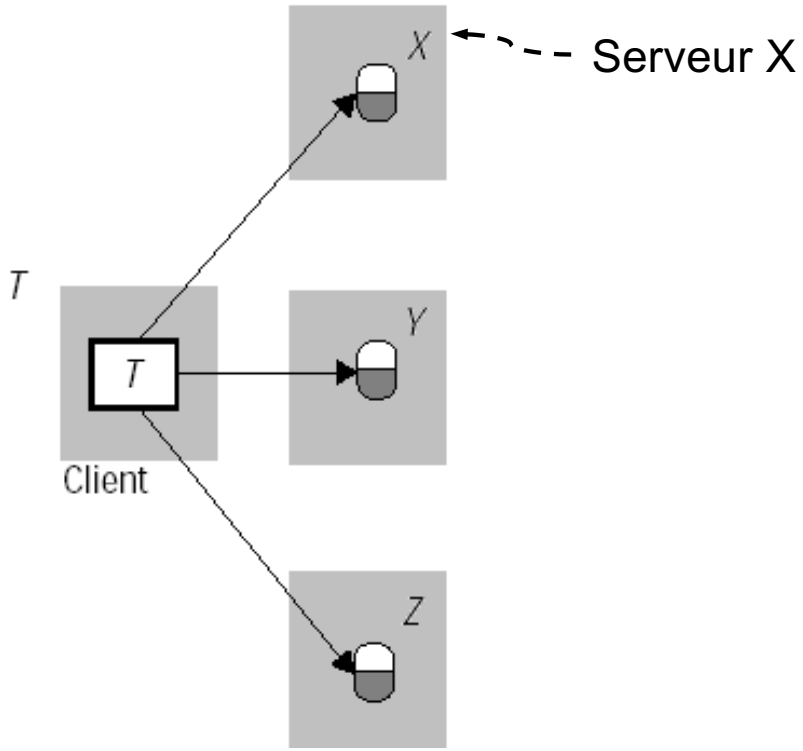
- Garantir que le contenu des objets reflète bien tous les effets des transactions complétées et aucun de celles abandonnées

Recouvrement de panne:

- Un serveur est désigné comme coordonnateur, il détermine si la transaction est approuvée ou non.
- Chaque serveur doit finaliser les transactions approuvées, même en cas de panne.

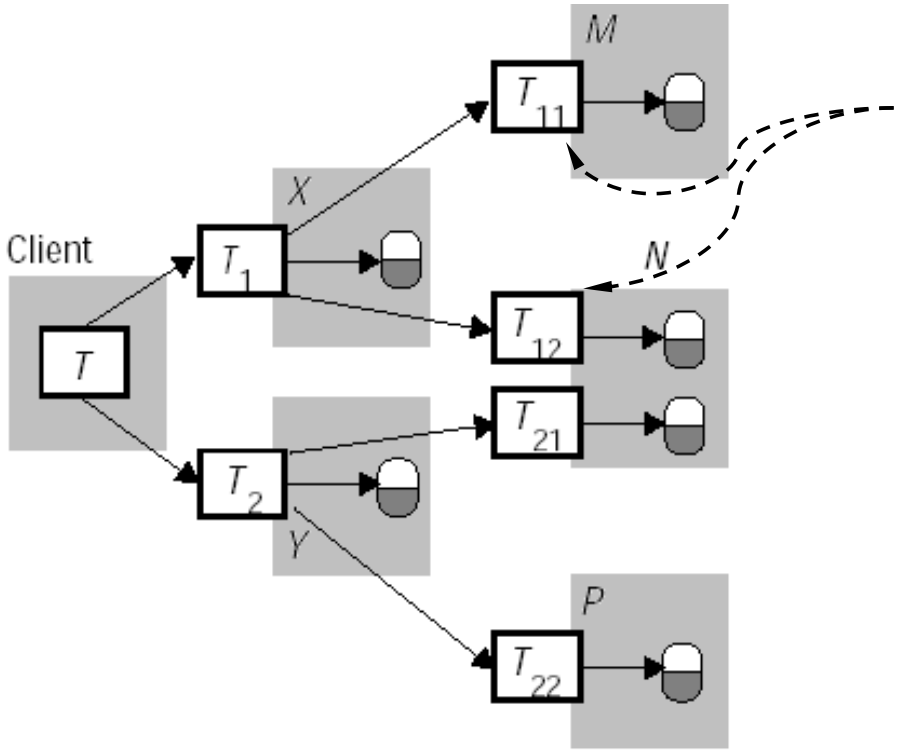
Transactions Réparties Simples

- Contient des opérations qui s'adressent à plusieurs serveurs mais un seul à la fois.
- Les requêtes sont envoyées et traitées séquentiellement.



Transactions Réparties Imbriquées

- Consiste en une hiérarchie de sous-transactions
- Les sous-transactions du même niveau s'exécutent simultanément, s'adressent à plusieurs serveurs et sont elles-mêmes des transactions imbriquées



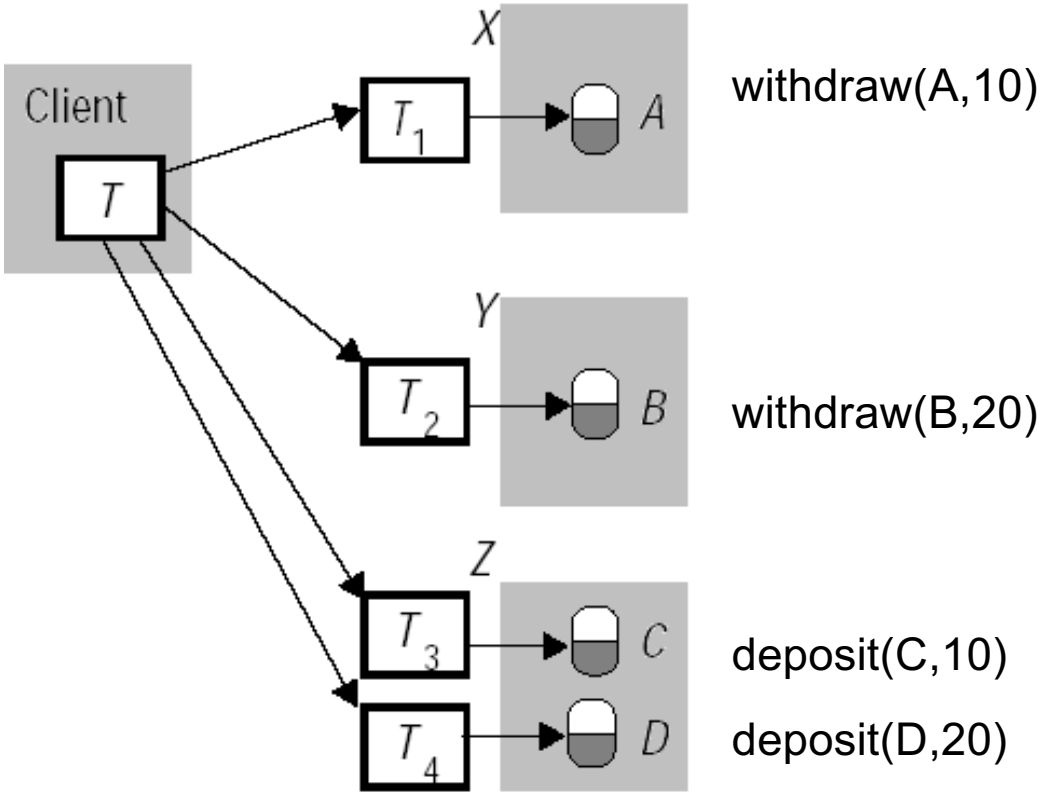
Peuvent s'exécuter simultanément (en parallèle) puisque différents serveurs

Exemple Transactions Réparties

Exemple : transaction répartie

Client transfère 10$ de A (serveur X) vers C (serveur Z), puis transfère 20$ de B (serveur Y) vers D (serveur Z),

- Begin Transaction T
  - withdraw(A,10)
  - withdraw(B,20)
  - deposit(C,10)
  - deposit(D,20)
- Commit T



Traitement d'une transaction

Serveurs (exécutant des requêtes appartenant à une transaction imbriquée) :

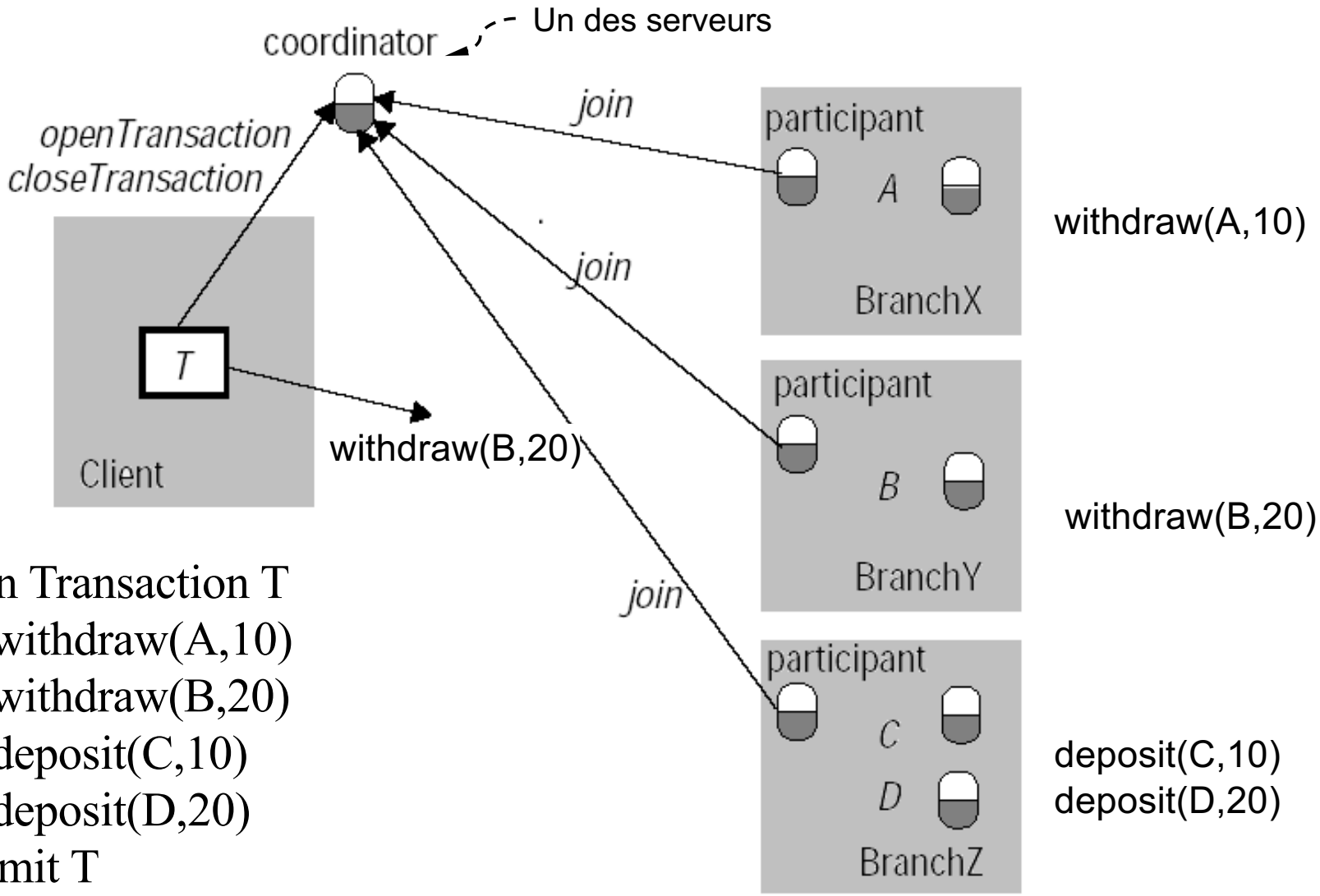
- Doivent se communiquer entre eux afin de coordonner leurs actions quand la transaction se termine correctement

- Réception d'une opération *openTransaction* par un serveur :

  - retourne un identificateur unique pour la nouvelle transaction (identificateur du serveur, e.g., adresse IP, concaténé avec un numéro unique généré localement)

  - devient le **coordinateur**, responsable de compléter (*commit*) ou d'abandonner (*abort*) la transaction

Traitement d'une transaction



- Begin Transaction T
  - withdraw(A,10)
  - withdraw(B,20)
  - deposit(C,10)
  - deposit(D,20)
- Commit T

Protocoles de fin de transaction

But = assurer l'atomicité d'une transaction répartie

- Toutes les opérations sont exécutées correctement, ou aucune opération n'est exécutée (elle est abandonnée)

Protocole de fin de transaction atomique <u>à une phase</u> :

Coordinateur :

- Envoie d'une façon répétitive un message de fin de transaction aux participants impliqués dans la transaction
- Jusqu'à que tous les participants répondent par un accusé de réception

<u>Inconvénients :</u> un serveur ne peut abandonner localement et unilatéralement la transaction (à la suite de conflits locaux engendrés par les mécanismes de contrôle de la concurrence)

- **Protocoles de fin de transaction**

Protocole de fin de transaction atomique <u>à deux phases</u> :

- Permet à un participant d'abandonner sa part de la transaction
- Utilisé lorsque le client envoie la requête *closeTransaction()*
- Client envoie *abortTransaction()* : le coordinateur informe tous les participants qu'ils doivent abandonner la transaction

<u>Phase de vote :</u>

- Chaque serveur (coordinateur et participants) vote pour compléter ou abandonner la transaction
- Serveur vote pour compléter la transaction

<u>Phase de décision :</u>

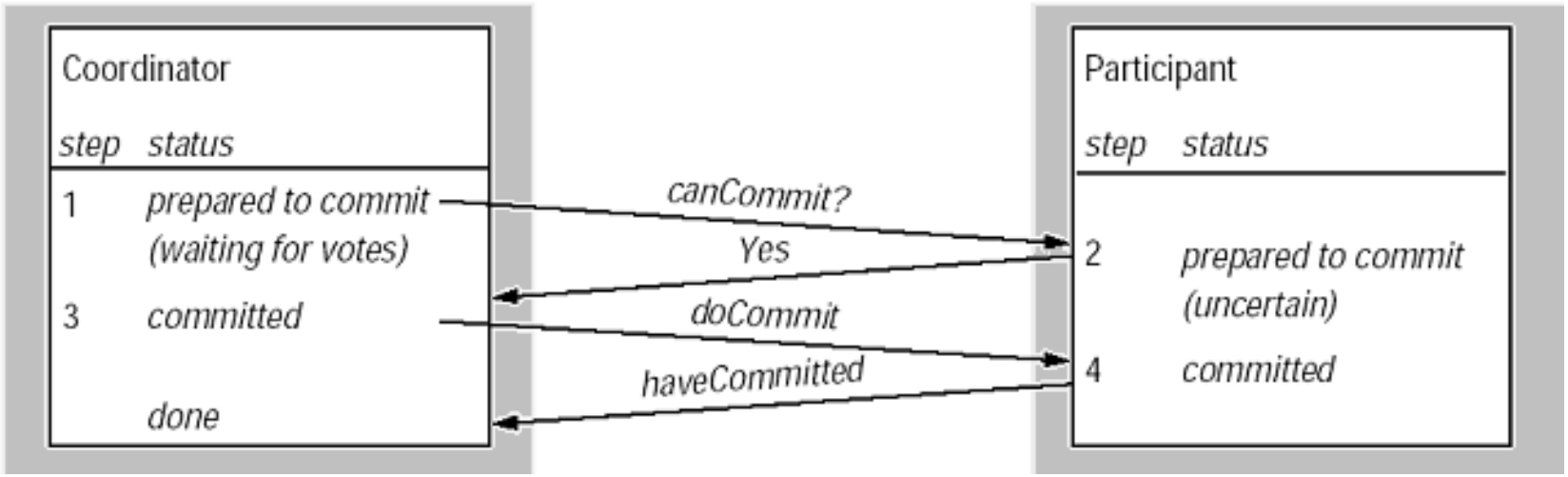
- Les serveurs exécutent la décision prise par le coordinateur
- Si au moins un des serveurs a voté pour abandonner, la décision sera d'abandonner la transaction

Protocoles de fin de transaction

Primitives du protocole :

Interface du participant :
- *canCommit?(trans)* → *Yes / No*
- *doCommit(trans)*
- *doAbort(trans)*

Interface du coordinateur :
- *haveCommitted(trans, participant)*
- *getDecision(trans)* → *Yes / No*



Problèmes possibles et solutions

- Si la réponse d'un participant (canCommit) en première phase tarde trop, le coordonnateur peut simplement annuler la transaction.
- Si un participant est sans nouvelle depuis trop longtemps d'une transaction amorcée mais pour laquelle il ne s'est pas commis, il peut abandonner la transaction.
- Si un participant s'est commis (répondu positivement à canCommit) mais ne reçoit pas de confirmation du coordonnateur (doCommit ou doAbort), il doit obtenir une réponse en relançant le coordonnateur avec getDecision.
- Le protocole requiert au minimum 4N messages soit 4 par participant (canCommit, réponse oui ou non, doCommit, haveCommitted). Le message haveCommitted est optionnel mais permet de libérer l'information sur la transaction. Si le coordonnateur est un participant, le nombre de messages réseau devient $4(N-1)$.

Exercice Coordonnateur

Une transaction répartie T, gérée par le coordonnateur s0, veut écrire les variables a=25 sur le serveur s1, b=33 sur le serveur s2, c=14 sur le serveur s3 et d=9 sur le serveur s4. Cette transaction répartie est commise en utilisant le protocole de fin de transaction atomique à deux phases. i) Quelles seront les entrées ajoutées au journal du coordonnateur et des serveurs s1 et s2, en lien avec cette transaction T? ii) Qu'est-ce qui arrive si s1 plante et redémarre avant la première phase de fin de transaction? Après la première phase?

Exercice Coordonnateur

SOLUTION : Le coordonnateur s0 va écrire les participants à la transaction puis va écrire que la transaction est complétée après avoir reçu un vote positif de chaque participant pendant la première phase. Il pourra ensuite annoncer que la transaction est complétée à chacun des participants et au client. Chacun des participants écrit les variables modifiées, ensuite écrit "prépare", lorsqu'on lui demande s'il est prêt à accepter la transaction, et finalement écrit "complété" lorsqu'il reçoit la confirmation du coordonnateur.

Si le serveur s1 plante avant la première phase, il aura oublié la transaction en cours et ne pourra donner un vote positif pendant la première phase. La transaction en cours sera donc abandonnée. Si le serveur s1 plante après la première phase, il aura fourni déjà un vote positif après s'être assuré d'avoir écrit dans son journal les informations nécessaires pour poursuivre la transaction, même après un redémarrage inopiné.

Contrôle de la concurrence pour les transactions réparties

- Les serveurs qui traitent des transactions réparties doivent s'assurer que leur exécution est équivalente à une exécution en série: Si une transaction T est avant U selon un des serveurs, alors elle doit être dans cet ordre pour tous les serveurs dont les objets sont accédés à la fois par T et U.

- Le contrôle optimiste de la concurrence avec sa phase de vérification atomique ne se prête pas bien aux transactions réparties.

- Le verrouillage fonctionne bien en réparti, chaque serveur contrôle la concurrence de ses propres objets avec des verrous posés localement et levés seulement à la fin de la transaction.

Interblocages dans les transactions réparties

- Les serveurs posent les verrous localement et imposent des ordres possiblement différents sur les transactions.
- Il se produit des interblocages répartis.
- Détecteur d'interblocage global (complexe) ou délai maximal (simple) pour se sortir d'un interblocage.

Transaction T	Verrous T	Transaction U	Verrous U
Write(A) on X	Lock A		
		Write(B) on Y	locks B
Read(B) on Y	Waits for U		
⋮		Read(A) on X	Wait for T

Exercice verrous deux phases et pannes

L'item a est répliqué sur A_x et A_y et l'item b sur B_m, et B_n.
Les transactions T et U sont définies ainsi:

```
T: Read(a); Write(b, 44);
U: Read(b); Write(a, 55);
```

1. Montrez un ordonnancement de T et U, dans le contexte de verrous en deux phases pour les réplicats.

Exercice verrous deux phases et pannes

```
T: Read(a); Write(b, 44);
U: Read(b); Write(a, 55);
```

Transac. T	Lock T	Transac. U	Lock U
x = Read(A_x)	lock A_x		
Write(B_m, 44)	lock B_m		
		x = Read(B_m)	wait B_m
Write(B_n, 44)	lock B_n	⋮	⋮
Commit	unlock A_x, B_m, B_n		
		x = Read(B_m)	lock B_m
		Write(A_x, 55)	lock A_x
		Write(A_y, 55)	lock A_y
		Commit	unlock B_m, A_x, A_y

Sommaire

Conclusion

- Dans beaucoup de cas, plusieurs étapes d'une opération logique doivent s'exécuter de manière atomique:
  - Transaction;
  - Modification de fichiers répliqués;
  - Installation de tous les fichiers d'une nouvelle version d'une application...

- Il faut assurer la cohérence par le contrôle de la concurrence (e.g. verrous).

- Le principe du protocole à deux phases est utilisé un peu partout pour assurer la validation
  - Transactions réparties possiblement imbriquées;
  - Envoi atomique de message de groupe;
  - Réserver un moment pour une réunion à plusieurs personnes (vérifier la disponibilité, ensuite confirmer ou annuler).