Cohérence et réplication pour les données réparties

Module 9

INF8480 Systèmes répartis et infonuagique

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

Cohérence et réplication pour les données réparties

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.

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: séquence d'opérations que le serveur exécute d'une manière atomique, même en présence d'accès simultanés par plusieurs clients et de pannes.
- 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.

Exemple: Banque

- Chaque compte est représenté par un objet.
- 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);
```

- Transaction atomique: deux propriétés à satisfaire
  - Tout ou rien: 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 interférence avec d'autres transactions (effets intermédiaires invisibles aux autres transactions)

Cohérence et réplication pour les données réparties

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 pourrait produire si elle s'exécutait seule dans le système

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

Pannes inévitables mais objets récupérables

- Objets pouvant être récupérés suite à une panne de serveur. Le serveur doit stocker suffisamment d'information en mémoire stable (disque), possiblement redondante.
- Un journal sur disque permet de lister les éléments d'une transaction avant de commettre ou d'annuler, et permet en cas de panne de savoir ce qui était en train d'être fait.
- La synchronisation des transactions pour les sérialiser est une méthode permettant l'isolation.

Coordonnateur de transaction

- Creation: 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 et annule une transaction annulée.
- Une transaction est complétée suite à la coopération entre le programme du client, les objets récupérables et le coordonnateur.

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*

En cas de 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 cetain 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).

En cas de 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.

Le problème de la cohérence

- Une banque contient deux comptes avec les soldes A = \$200 et B = \$200.
- Un client 1 fait une transaction V: le transfert de \$100 de A vers B.
  - Retirer \$100 de A
  - Ajouter \$100 à B
- Un client 2 fait simultanément une transaction W: calculer la somme des comptes de la banque.
- Sans synchronization, la somme peut donner \$300 (incorrect) ou \$400.

Sérialisation des transactions

- Une transaction exécutée seule ne présente pas de problème.
- L'exécution sérialisée des transactions, une par une, permet aussi d'assurer les propriétés requises, T1, T2, . . . , Tn .
- Entrelacement des transactions équivalent à une exécution en série, pour un ordre des transactions quelconque, est aussi acceptable.

Contraintes pour sérialiser les transactions

- Deux opérations sont en conflit si leur effet combiné dépend de l'ordre dans lequel elles sont exécutées.
- Lecture-Lecture, pas de conflit.
- Lecture-Ecriture, conflit, le résultat de la lecture est affecté par l'écriture de la variable à lire.
- Ecriture-Ecriture, conflit, le résultat des lectures subséquentes dépend de l'ordre des deux écritures sur une même variable.

Sérialisation des transactions: exemple

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

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

- Cet ordonnancement n'est pas équivalent à une exécution sérielle, il faudrait 1) ou 2):
  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

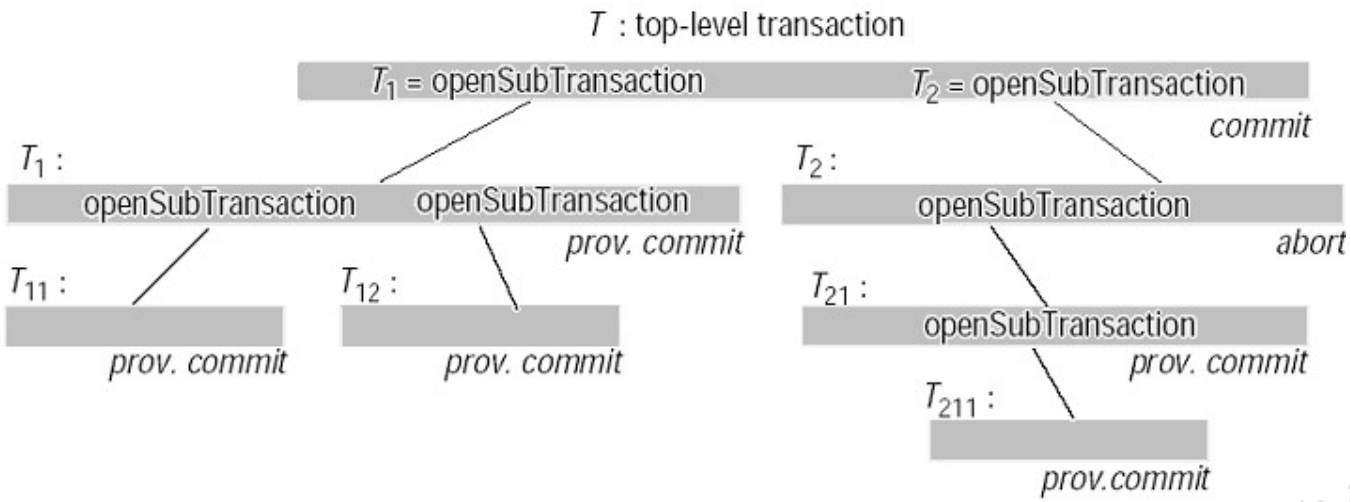
Cohérence et réplication pour les données réparties

Transactions Imbriquées

- Transaction contenue dans une autre et qui peut elle-même contenir des transactions imbriquées
- Une transaction imbriquée à un niveau de la hiérarchie n'est complétée que si son parent complète son exécution
- Accroit 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

- Sous-transactions du même niveau, e.g. T1 et T2, peuvent s'exécuter simultanément, mais leurs accès aux objets partagés sont sérialisés (verrouillage)
- Si une sous-transaction est abandonnée: le parent peut choisir une autre sous-transaction pour compléter sa tâche



Cohérence et réplication pour les données réparties

Approches de contrôle de la concurrence

- Le protocole doit produire un ordonnancement équivalent à une exécution en série.
- Le verrouillage:
  - 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:
  - On suppose l'absence de conflit.
  - Chaque transaction est exécutée jusqu'à la fin sans bloquer.
  - 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.

Recouvrement après l'abandon d'une Transaction

- Le serveur doit:
  - Sauvegarder les effets des transactions complétées
  - Assurer qu'aucun effet des transactions abandonnées ne soit stocké
- Problèmes associés à l'abandon de transactions, si mal géré:
  - Problème de lectures contaminées (*dirty reads*)
  - Problème d'écritures prématurées (*premature writes*)

Problème de lectures contaminées

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

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

Se prémunir contre les écritures prématurées

- Chaque transaction possède ses propres versions provisoires(tentative) des objets qu'elle met à jour.
- Les écritures sont faites sur les versions provisoires.
- Les lectures d'autres transactions sont faites à partir des versions provisoires (correct si finit après), ou à partir des versions permanentes (correct si finit avant ou est annulé).
- Les versions provisoires sont transférées dans les versions permanentes des objets seulement lorsque la transaction est complétée, en une seule étape.
- Pendant le transfert, les autres transactions n'ont pas accès aux objets modifiés.
- Si les objets modifiés ne sont pas verrouillés (contrôle optimiste), il faut vérifier la cohérence des transactions concurrentes.

Verrouillage

- 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, il doit attendre son déverrouillage.
- Verrou de lecture: avant une opération de lecture d'un objet, le serveur pose un verrou (non exclusif) de lecture sur l'objet.
- Verrou d'écriture: avant une opération d'écriture d'un objet, le serveur pose un verrou (exclusif) d'écriture sur l'objet.

Exemple de verrouillage (suite)

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

Granularité de verrouillage

- Verrou complet ou séparé lecture et écriture.
- Verrou sur un seul objet.
- Verrou sur un groupe d'objets (e.g. page sur disque).
- Verrou global (sérialiser complètement les transactions).

Protocoles de verrouillage

- Verrouillage à 2 phases:
  - 1ère phase (*growing phase*): la transaction acquiert de nouveaux verrous.
  - 2ème phase (*shrinking phase*): la transaction libère ses verrous.
  - Aucun verrou ne peut être posé si un verrou a été levé.
- Verrouillage strict à 2 phases:
  - Verrouillage à 2 phases dans lequel les verrous sont libérés seulement lorsque la transaction est complétée ou abandonnée.
  - Prévient les lectures contaminées et les écritures prématurées
  - Il faut attendre le *commit* ou *abort* mais pas nécessairement la fin de la transaction pour libérer les verrous de lecture.

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.

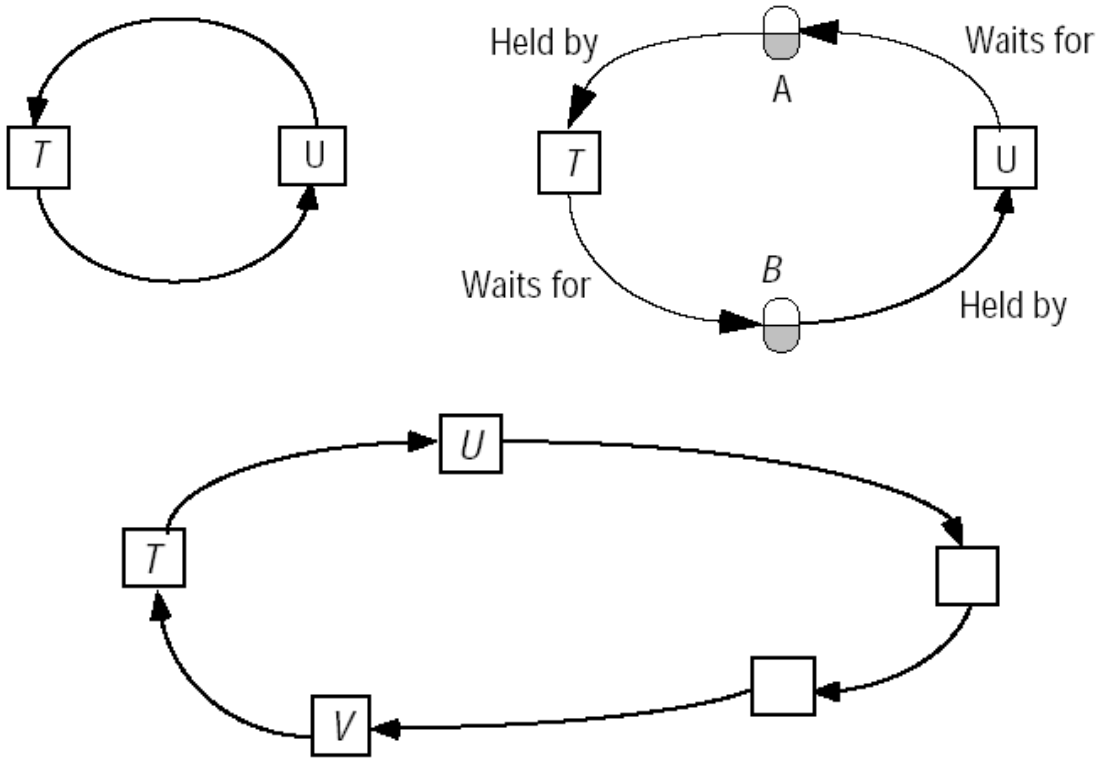
Interblocages

- Implique au moins deux transactions.
- 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
⋮	⋮	⋮	⋮

Graphes de dépendance

- Dépendance directe: T attend que U libère le verrou et vice-versa.
- Dépendance indirecte: V attend après T qui attend après U...



Prévention des interblocages

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.

- Pour ordonner, il faut aussi prédire les objets utilisés.

Détection d'interblocage

- Détecter les cycles dans le graphe de dépendance.
- Abandonner une transaction appartenant à chacun des cycles.
- Le 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.
  - Vérification périodique de l'existence de cycle.
  - Lorsqu'un cycle est détecté, une des transactions est abandonnée (choisie selon son âge, le nombre de cycles dans lesquels elle apparaît).

Résolution d'interblocage

- Chaque verrou pris a une période durant laquelle il est non vulnérable.
- A 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: Principe de validation des transactions

- 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.

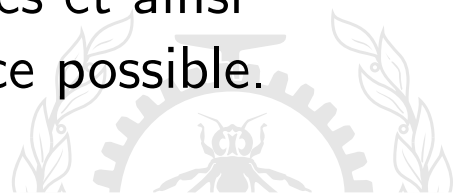
Transaction T	Transaction U	Règle
écriture	lecture	U ne doit pas lire un objet écrit par T
lecture	écriture	T ne doit pas lire un objet écrit pat U
écriture	écriture	T ne doit pas écrire un object écrit par U U ne doit pas écrire un objet écrit par T

COC: Approches de validation des transactions

- **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.

Validation des transactions par estampille de temps

- Un numéro d'ordre (temps) est affecté à chaque transaction au moment du début de la transaction. T_i précède T_j si $i < j$.
- A chaque lecture, ou écriture d'une variable, on note le temps de la transaction.
- Pour une écriture d'une variable, si la transaction est avant la dernière lecture ou avant la dernière écriture commise, la transaction est annulée, autrement l'écriture est acceptée.
- Pour la lecture d'une variable, si la transaction est avant la dernière écriture commise, la transaction est annulée, autrement la lecture est acceptée.
- Des versions plus élaborées de l'algorithme peuvent lire des anciennes valeurs ou des nouvelles valeurs tentatives et ainsi éviter plusieurs conflits et augmenter la concurrence possible.



Cohérence et réplication pour les données réparties

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.
- Disque + ruban.
- Rubans ou 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 donnée.
- 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.

Méthode des doubles versions

- Un espace de donnée suffisant pour les valeurs acceptées et les valeurs tentatives de toutes les transactions en cours est utilisé.
- Une carte dit pour chaque variable où se trouve la valeur acceptée.
- Pour préparer une transaction, les nouvelles valeurs sont stockées et une carte qui y réfère est créée.
- Pour accepter une transaction, la nouvelle carte remplace l'ancienne.

Cohérence et réplication pour les données réparties

Les transactions réparties

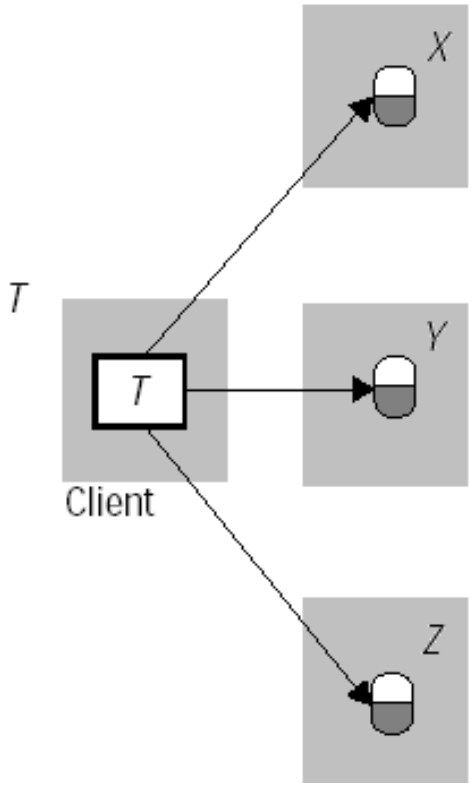
- Une transaction répartie peut faire des requêtes à plusieurs serveurs.
- Un serveur peut faire appel à d'autres serveurs pour traiter une requête
- A la fin de la transaction, tous les serveurs doivent soit compléter, soit abandonner (prennent conjointement la même décision).

Défis des transactions réparties

- Contrôle de la concurrence :
  - Les transactions doivent être *sérialisées globalement* sachant que chaque serveur sérialise localement les transactions.
  - Des *interblocages répartis* 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.

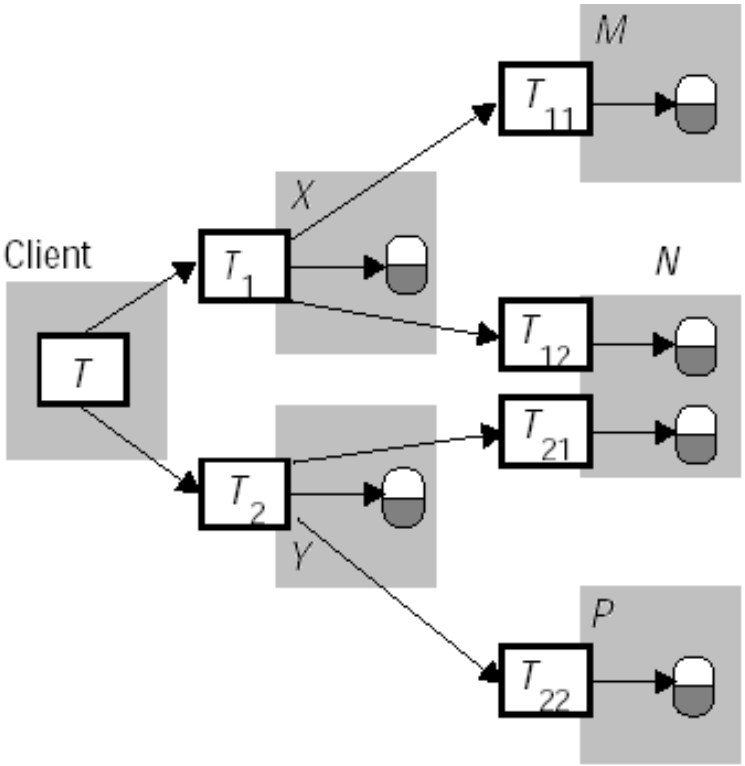
Transactions Réparties Simples

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



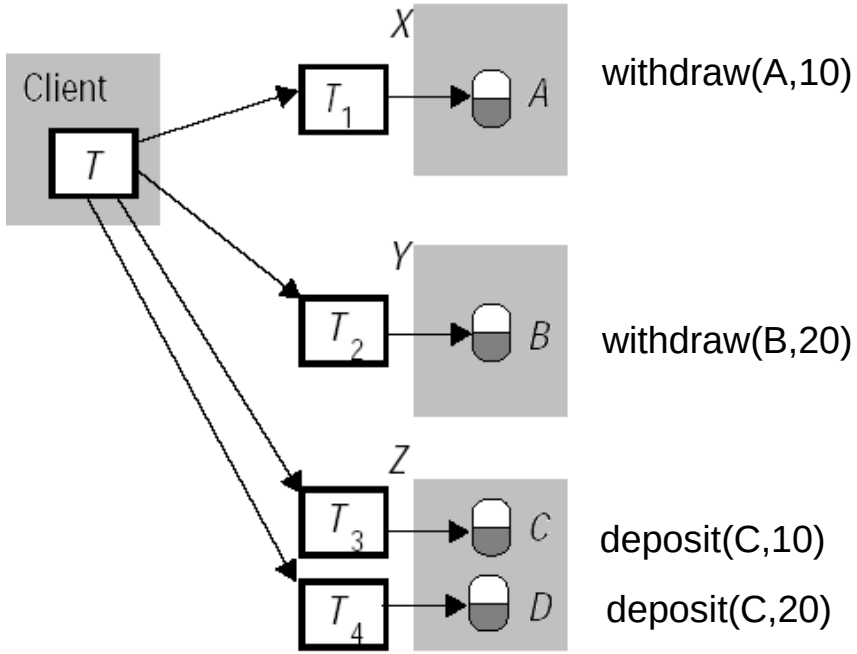
Transactions Réparties Imbriquées

- 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

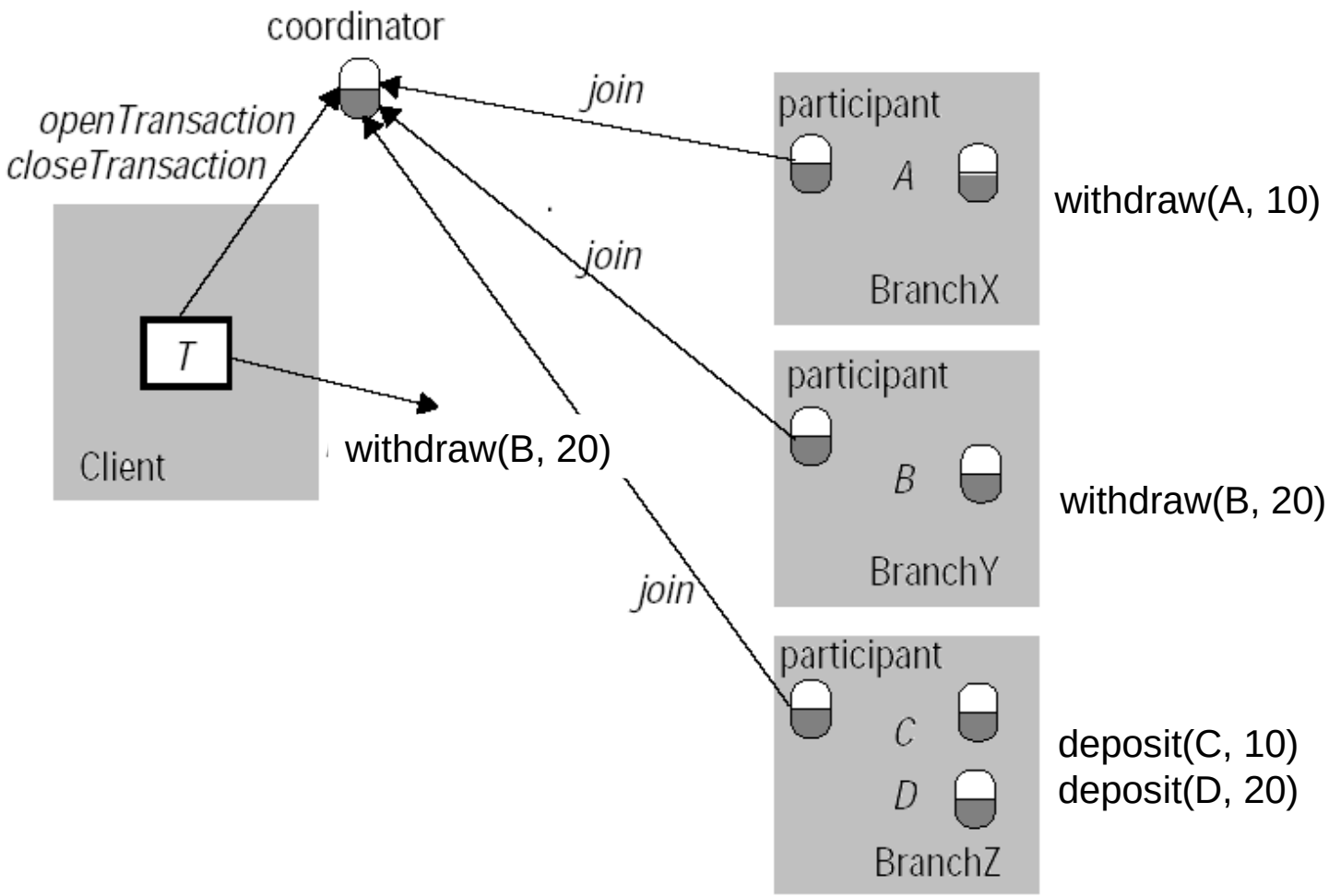


Transactions Réparties

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



Traitement d'une transaction



Protocoles de fin de transaction

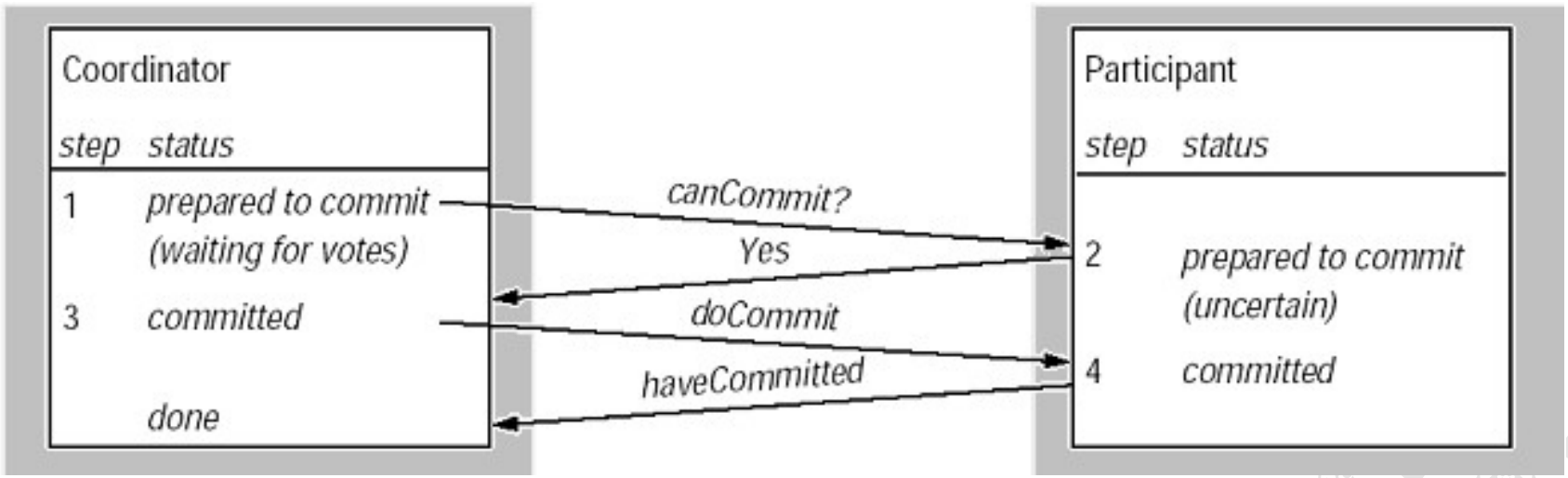
- Assurer l'atomicité d'une transaction répartie: toutes les opérations sont exécutées correctement, ou aucune opération n'est exécutée (elle est abandonnée).
- Protocole de fin de transaction atomique *à une phase*: le coordonnateur envoie d'une façon répétitive un message de fin de transaction aux participants jusqu'à ce que tous les participants répondent par un accusé de réception. Ne fonctionne que si tous acceptent (ou tous refusent).
- Protocole de fin de transaction atomique *à deux phases*: premier tour pour vérifier si tous les participants peuvent accepter la transaction, deuxième tour pour donner le verdict.

Protocole de fin de transaction atomique à deux phases

- Phase de vote:
  - Chaque serveur (coordonnateur et participants) vote pour compléter ou abandonner la transaction.
- Phase de décision:
  - Si au moins un des serveurs a voté pour abandonner, la décision sera d'abandonner la transaction.
  - Les serveurs exécutent la décision prise par le coordonnateur.

Interface du protocole à deux phases

- Interface du participant:
  - canCommit(trans): Yes / No
  - doCommit(trans)
  - doAbort(trans)
- Interface du coordonnateur
  - 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 $4(N-1)$.

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

Cohérence et réplication pour les données réparties

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).