ÉCOLE POLYTECHNIQUE MONTRÉAL

Module 7:

Transactions réparties

Introduction

Motivations:

- Transaction: peut faire des requêtes à plusieurs serveurs
- Serveur: peut faire appel à d'autres serveurs pour traiter une requête

Propriétés à satisfaire:

- Commit ou abort: à la fin de la transaction, tous les serveurs doivent soit compléter, soit abandonner (prennent conjointement la même décision)

Introduction

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

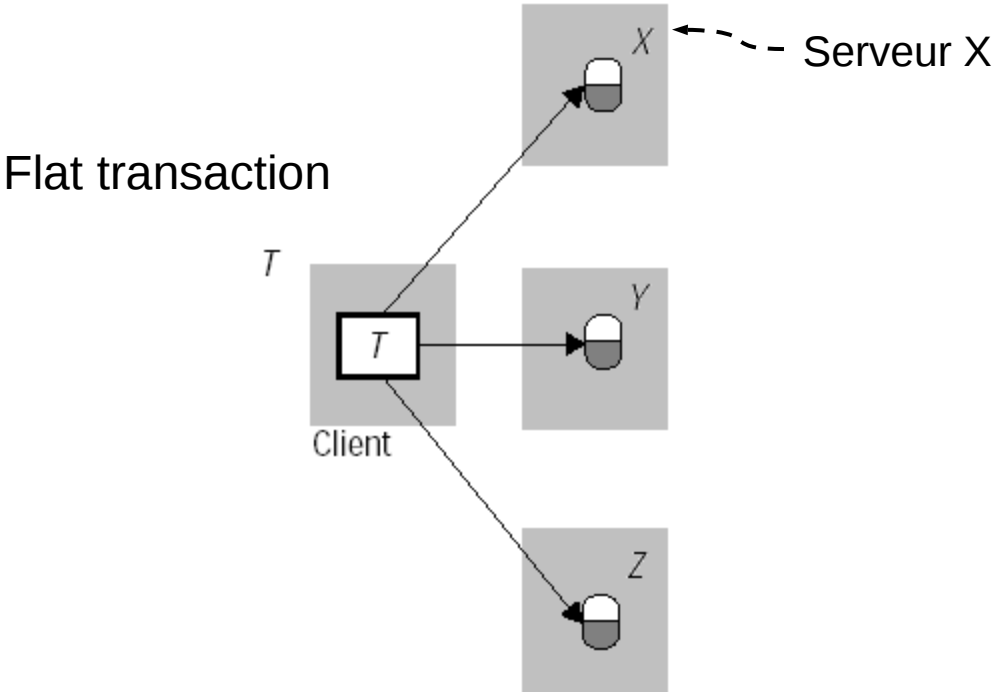
Recouvrement de panne:

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

Transaction répartie simple :

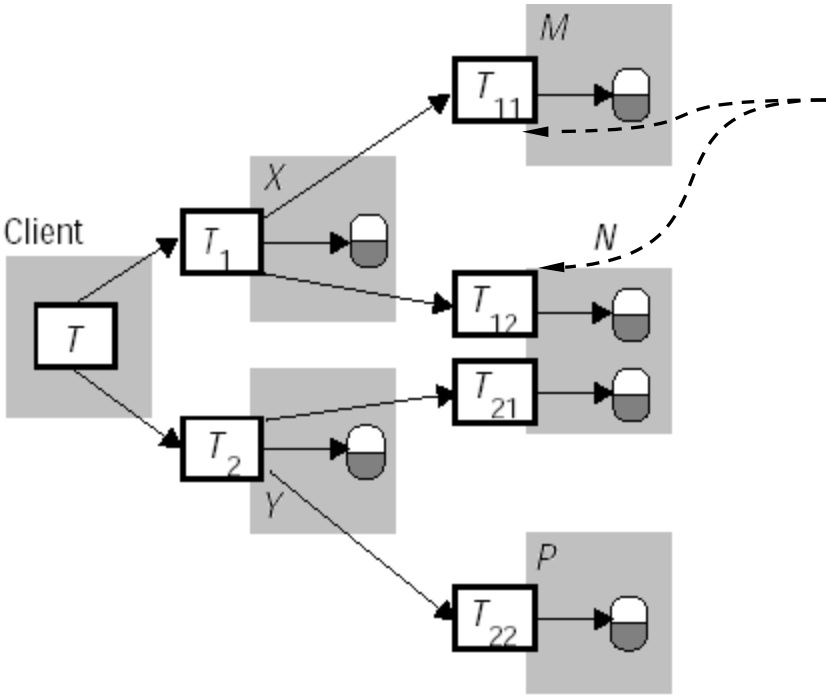
- Contient des opérations qui s'adressent à plusieurs serveurs, mais un seul à la fois
- Les requêtes sont envoyées et traitées séquentiellement



• Transactions Réparties Imbriquées

Transaction répartie imbriquée :

- 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



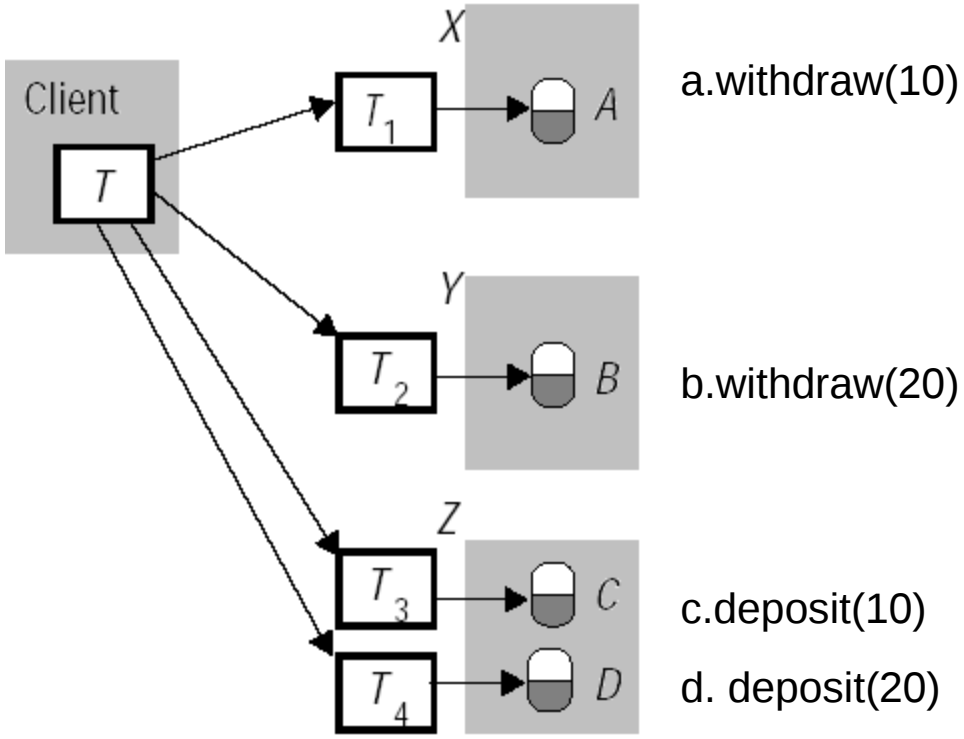
Peuvent s'exécuter simultanément (en parallèle) puisque différents serveurs

• Transactions Réparties

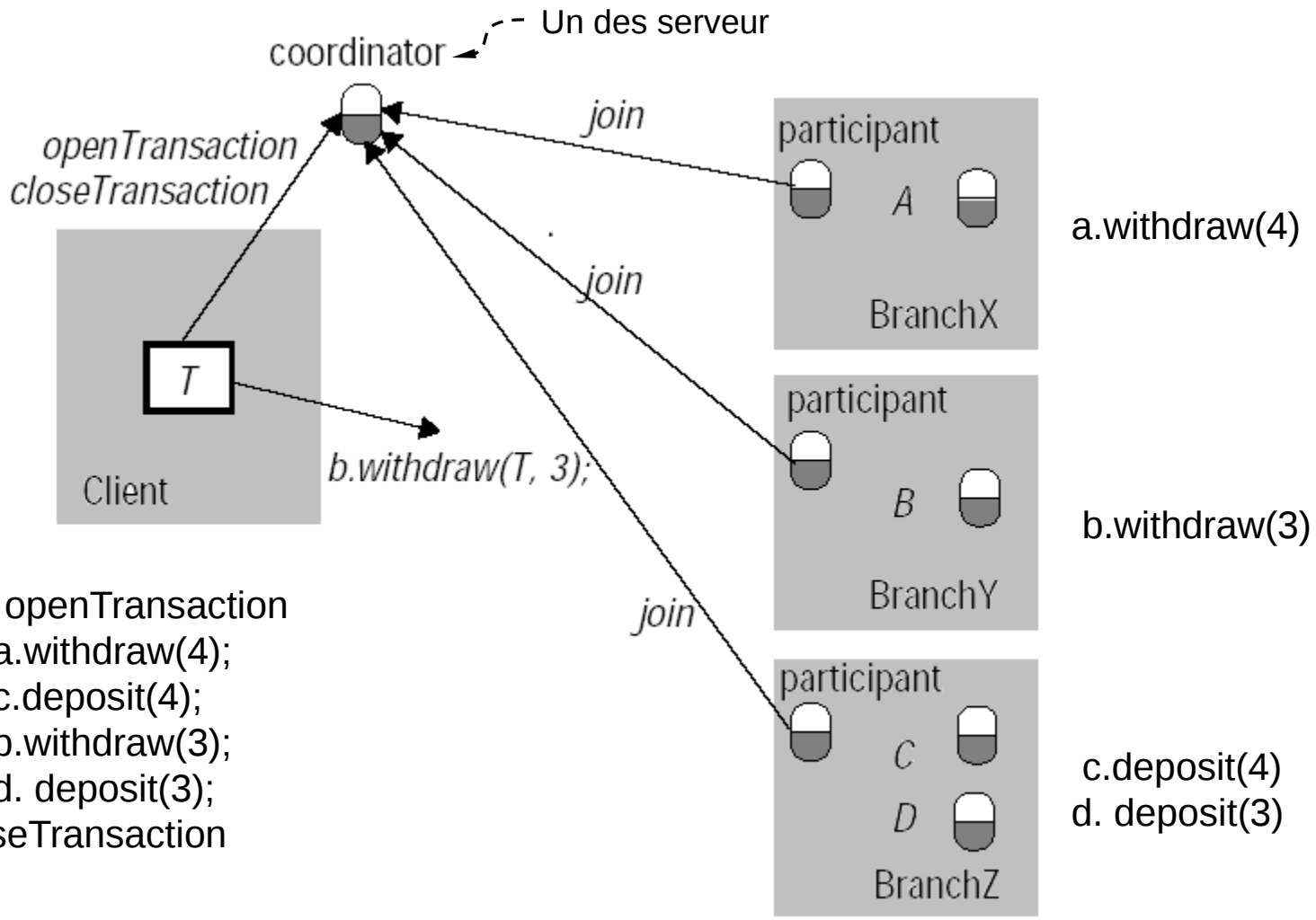
Exemple : transaction répartie

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

```
T = openTransaction
    a.withdraw(10);
    b.withdraw(20);
    c.deposit(10);
    d. deposit(20);
closeTransaction
```



• Traitement d'une transaction



T = openTransaction
    a.withdraw(4);
    c.deposit(4);
    b.withdraw(3);
    d. deposit(3);
closeTransaction

• 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> :

Coordonnateur :

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

<u>Inconvénients:</u> un serveur ne peut abandonner localement et unilatéralement la transaction (suite à des 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 coordonnateur informe tous les participants qu'ils doivent abandonner la transaction.

<u>Phase de vote:</u>

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

<u>Phase de décision:</u>

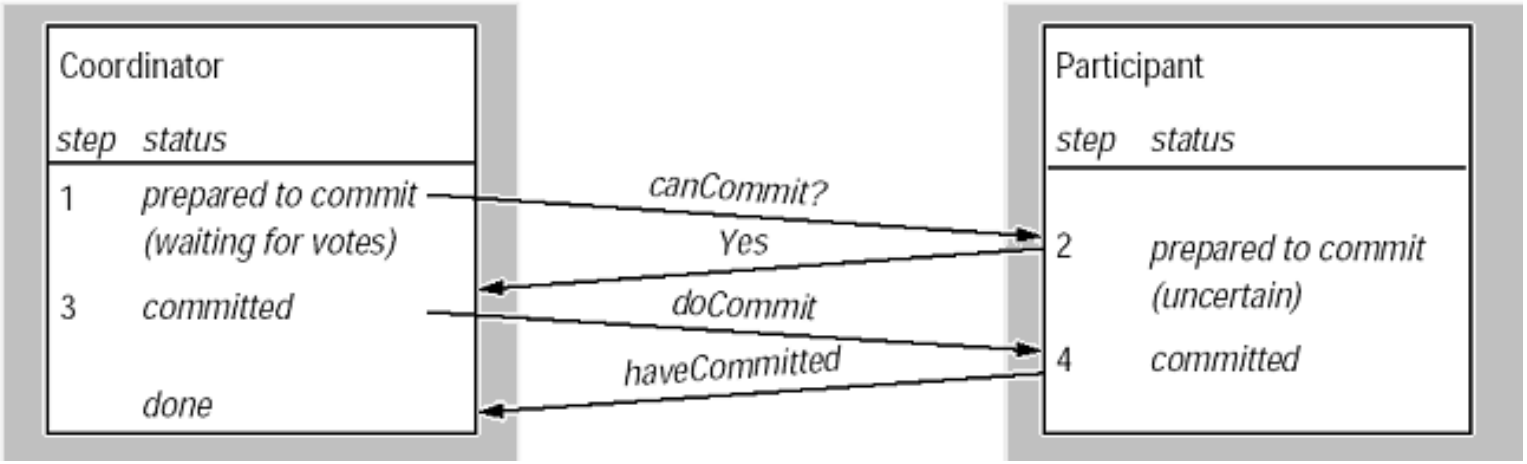
- Les serveurs exécutent la décision prise par le coordonnateur.
- Si au moins un des serveurs a voté pour abandonner, la décision sera d'abandonner la transaction.

• Protocoles de fin de transaction

Primitives du protocole:

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

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



• Protocoles de fin de transactions

Délais d'attente trop longs:

- lorsque le coordonnateur attend les votes des participants (il peut abandonner la transaction).
- lorsqu'un participant a traité toutes les requêtes qui lui étaient destinées mais qu'il n'a pas encore reçu *canCommit?(T)* du coordonnateur (il peut abandonner la transaction).
- un participant a voté *YES* mais il n'a pas encore reçu le message *doCommit(T)* ou *abortCommit(T)* du coordonnateur (il peut envoyer *getDecision(T)* au coordonnateur).

Complexité de l'algorithme : nécessite au minimum 3(N-1) messages pour compléter normalement une transaction (ou 4(N-1) en comptant *HaveCommitted(T)*)

• Protocoles de fin de transactions imbriquées

Client : démarre la transaction imbriquée en utilisant *OpenTransaction()*

Serveur : traite la transaction de haut niveau

- Démarre des sous-transactions avec *OpenSubTransaction()*
- Maintient à jour une liste des transactions provisoirement complétées et une des transactions abandonnées
- Ces listes sont transmises au parent, et ainsi de suite jusqu'au serveur qui traite la transaction de haut niveau

Identificateur d'une sous-transaction : doit être unique

Concaténation de :

- Identificateur de la transaction parent
- Identificateur du serveur
- Numéro unique (généré localement par le serveur de la sous-transaction)

• Protocoles de fin de transactions imbriquées

Identificateur d'un parent ou de la transaction de haut-niveau: peut être déduit de l'identificateur d'une sous-transaction.

Sous-transaction:

- Se termine: le serveur enregistre l'information comme si la sous-transaction était provisoirement complétée ou abandonnée.
- N'est pas réellement complétée à moins que la transaction imbriquée soit complétée.

Transaction:

- N'est pas nécessairement abandonnée si une sous-transaction est abandonnée.
- Programmée selon qu'une sous-transaction abandonne ou pas.
- Si abandonnée, elle ne passe aucune information à propos de ses sous-transactions.

• Protocoles de fin de transactions imbriquées

Phase 1 (phase de vote):

1) Coordonnateur: envoie *canCommit(trans, abortList)* à chaque participant impliqué dans la transaction.

2) Participant reçoit le message *canCommit(trans, abortList)*:

Si le participant gère des transactions provisoirement complétées (qui sont des descendants de la transaction de haut niveau)
Alors

Si chaque transaction provisoirement complétée n'as pas un ancêtre dans *abortList*

Alors Il se prépare à compléter la transaction (enregistre la transaction et les objets sur le support permanent).

Sinon Abandonne les transactions provisoirement complétées ayant un ancêtre dans *abortList.*

• Protocoles de fin de transactions imbriquées

Phase 1 (suite):

- Envoie un vote *Yes* au coordonnateur
- **Si** le participant n'a pas de transactions provisoirement complétées qui sont des descendants de la transaction de haut niveau

  Alors il envoie *No* au coordonnateur (il peut être tombé en panne depuis qu'il a traité les sous-transactions)

Phase 2 *(confirmer selon le résultat du vote):*

1) Coordonnateur collecte les votes (incluant son propre vote)

   Si Tous les serveurs ont voté *Yes*

   Alors le coordonnateur complète la transaction et envoie *doCommit(trans)* à chaque participant

   Sinon le coordinateur abandonne la transaction et envoie *abortTransaction(trans)* à chaque participant qui a voté *Yes*

• Protocoles de fin de transactions imbriquées

Phase 2 (suite) :

1) Participants qui ont voté *Yes*:

- Attendent *doCommit(trans)* ou *abortTransaction(trans)* du coordinateur.
- Agissent en conséquence, et dans le cas d'un *doCommit(trans)*, ils lui retournent *haveCommitted(trans).*

Délais d'attente longs :

- Cas d'un serveur qui a des transactions provisoirement complétées provenant d'une transaction abandonnée (parent) et qui n'est pas un participant.

N'est pas informé par le coordonnateur parce qu'il n'est pas dans l'arbre :

- Utilise *getStatus(trans)* pour demander à son parent l'état de la transaction parent.
- Si le parent ne répond pas, alors il utilise *getDecision(trans)* pour demander au coordinateur l'état de la transaction entière.

- **Contrôle de la concurrence pour les transactions réparties**

Serveur:

- Gère un ensemble d'objets.
- Assure leur cohérence quand ils sont accédés par des transactions concurrentes.

Chaque serveur contrôle la concurrence de ses propres objets.

Serveurs qui traitent des transactions réparties: doivent s'assurer que leur exécution est équivalente à une exécution en série.

Algorithme:

Si 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.

• Verrouillage des transactions réparties

Verrous:

- Posés localement.
- Levés lorsque tous les serveurs impliqués dans la transaction ont pris une décision commune et l'ont exécutée.

Serveurs posent les verrous localement :

- Ils imposent des ordres différents sur les transactions et
- Il se produit des interblocages répartis.

Exemple d'interblocage réparti:

Transactions T et U s'exécutent sur deux serveurs X et Y

T			U		
Write(A)	at X	locks A			
			Write(B)	at Y	locks B
Read(B)	at Y	waits for U			
			Read(A)	at X	waits for T

• Contrôle par estampilles des transactions réparties

Chaque serveur doit générer des estampilles uniques globalement (par rapport à l'ensemble des serveurs)

Estampille =
< *localTimestamp*, *serverId* >

Identificateur du serveur est concaténé à chaque estampille locale (de façon à imposer un ordre global entre les estampilles de tous les serveurs)

Premier serveur auquel s'adresse la transaction:

- Génère une estampille unique globale
- Passe l'estampille générée à chaque serveur qui exécute une opération de la transaction.

Cas de transaction qui doit être abandonnée suite à un conflit :

- Le coordonnateur est averti et abandonne la transaction

Tous les serveurs: utilisent les estampilles pour ordonner toutes les opérations déclenchées par les transactions réparties

Résolution des conflits: se fait au moment où chaque opération est exécutée.

• Interblocages répartis

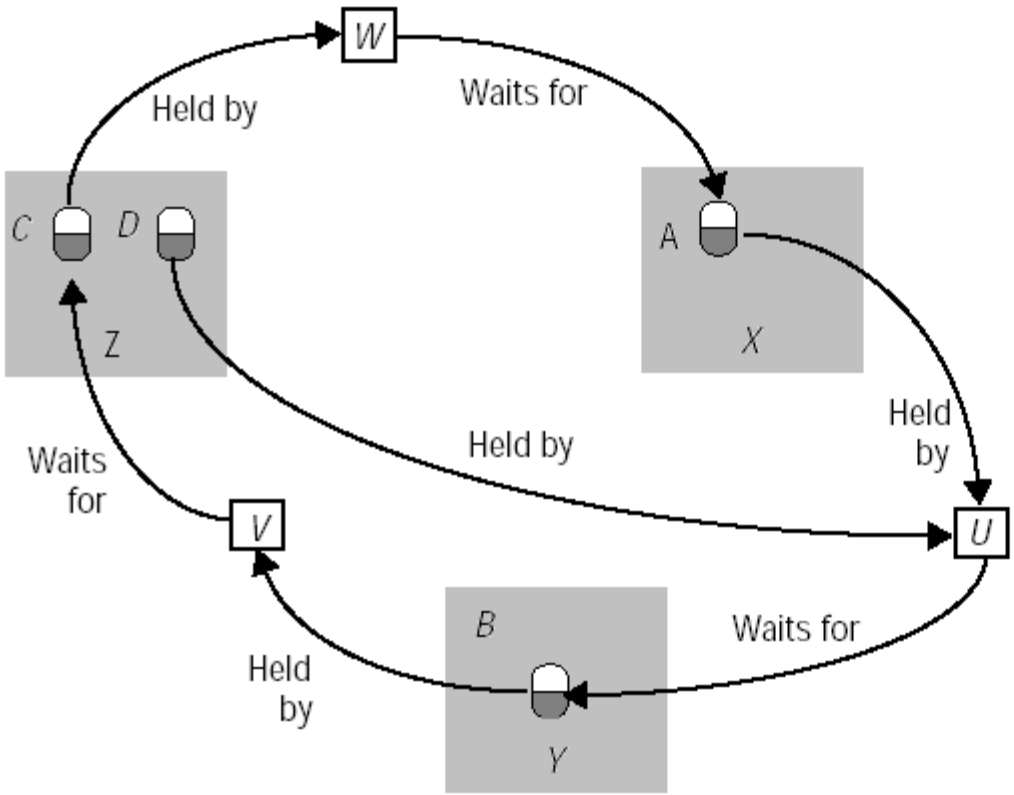
Graphe de dépendance global : peut être construit à partir des graphes de dépendance locaux

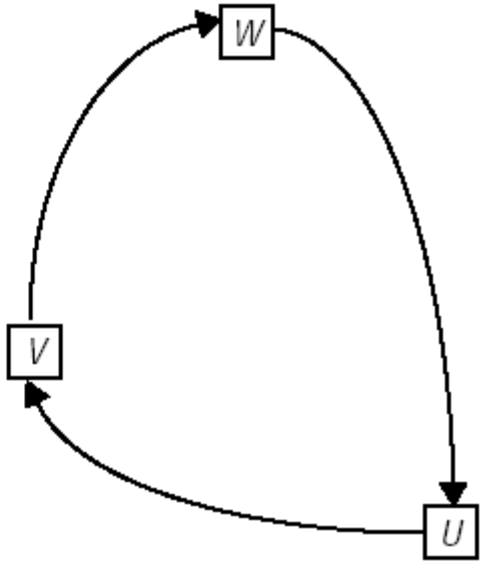
Graphes de dépendance locaux sans cycles peuvent engendrer un graphe de dépendance global avec des cycles

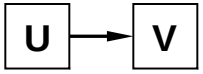
Exemple : Transactions imbriquées U, V et W
Objets : A (serveur X), B (serveur Y), C et D (serveur Z)

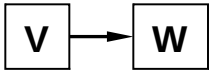
U	V	W
d.deposit(10) lock D at Z		
	b.deposit(10) lock B at Y	
a.deposit(20) lock A at X		
		c.deposit(30) lock C at Z
b.withdraw(30) wait at Y		
	c.withdraw(20) wait at Z	
		a.withdraw(20) wait at X

- **Interblocages répartis**





Serveur Y 

Serveur Z 

• Interblocages répartis

Détection centralisé d'interblocages:

- Un serveur assure le rôle d'un détecteur d'interblocage global.
- Collecte périodiquement les graphes de dépendance locaux.
- Construit le graphe de dépendance global.
- Vérifie l'existence de cycles.
- Si un cycle existe, alors il décide de la stratégie de résolution et informe les serveurs dont la transaction sera abandonnée.

Cette approche possède tous les désavantages d'une approche centralisée. Dans beaucoup de cas, un délai maximal est simplement utilisé pour décider d'annuler une transaction.