

Systemes concurrents

Philippe Quéinnec

14 septembre 2012



Neuvième partie

Transactions



Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
 - Type d'incohérences
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



Interférences et isolation

Objets partagés + actions concurrentes \Rightarrow résultats cohérents ?

Approche :

- directe : synchronisation des actions (exclusion mutuelle)
- indirecte : **contrôle de concurrence**
assurer un contrôle d'équivalence à un résultat cohérent



Contraintes d'intégrité

États **cohérents** décrits en intention par des **contraintes d'intégrité** (prédicats portant sur les valeurs des données).

Exemple

Base de données bancaire

- données = ensemble des comptes
- contraintes :
 - la somme des comptes est constante
 - chaque compte est positif

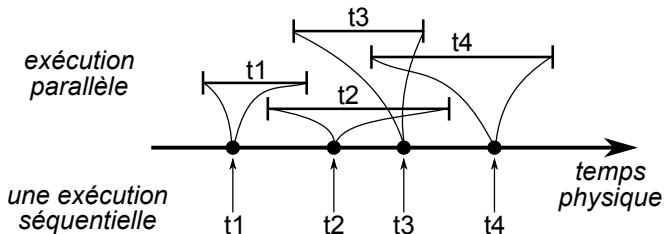


Transaction

Définition

Suite d'opérations menant à un état cohérent, à partir de tout état cohérent.

- masquer les états intermédiaires
- « parenthésage » des états non observables
- possibilité d'abandon **sans effet visible**
⇒ transaction **validée** (committed).



Domaines d'utilisation

- Systèmes d'information : bases de données
- Systèmes de fichiers (en particulier systèmes journalisés)
- Mémoire transactionnelle
(HTM/STM = hardware/software transactional memory)



Incohérences

Pertes de mises à jour

Écritures écrasées par d'autres écritures.

(1) $a := \text{lire}(x)$;		(a) $b := \text{lire}(x)$;
(2) $\text{écrire}(x, a+10)$;		(b) $\text{écrire}(x, b+20)$;

Lectures sales

Écritures abandonnées mais observées

(1) $\text{écrire}(x, 100)$;		(a) $b := \text{lire}(x)$;
(2) abandon ;		

Incohérences

Lectures non répétables

Donnée qui change de valeur

(1) $a := \text{lire}(x)$;
(2) $b := \text{lire}(x)$;

||| (a) $\text{écrire}(x, 100)$;

Lectures fantômes

Donnée agrégée qui change de contenu

(0) $\text{sum} := 0$;
(1) $\text{nb} := \text{cardinal}(S)$

||| (a) $\text{ajouter}(S, 15)$;

(2) $\forall x \in S : \text{sum} := \text{sum} + x$
(3) $\text{moyenne} := \text{sum} / \text{nb}$

Propriétés ACID

Atomicité ou « tout ou rien » : en cas d'abandon (volontaire ou subi), aucun effet visible

Cohérence : respect des contraintes d'intégrité

Isolation : pas d'interférences entre transactions = pas d'états intermédiaires observables

Durabilité : permanence des effets d'une transaction validée



Annulation/abandon

- Pour garantir la cohérence et/ou l'isolation \Rightarrow possibilité d'**abandon** (abort) d'un traitement en cours, décidé par le système de gestion.
- Du coup, autant le fournir aussi au programmeur.



Sémantique

- **Single-lock atomicity** : exécution équivalente à l'utilisation d'un unique verrou global.
- **Sérialisabilité** : résultat final équivalent à une exécution sérialisée des transactions qui valident.
- **Sérialisabilité stricte** : sérialisabilité + respect de l'ordre temps réel (si T_A termine avant que T_B ne démarre et que les deux valident, T_A doit apparaître avant T_B dans l'exécution sérialisée équivalente)
- **Linéarisabilité** : transaction considérée comme une opération atomique instantanée, à un point entre son début et sa validation.
(différence avec sérialisabilité : accès non transactionnels pris en compte)
- **Opacité** : sérialisabilité stricte, y compris des transactions annulées (indépendance par rapport aux transactions actives).



Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
 - Type d'incohérences
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



Abandon sans effet

Comment abandonner une transaction sans effet ?

- 1 Pessimiste / propagation différée : mémoire temporaire transférée en mémoire définitive à la validation (*redo-log*)
- 2 Optimiste / propagation immédiate (en continu) : écriture directe avec sauvegarde de l'ancienne valeur ou de l'action inverse (journaux / *undo-log*)

Effet domino : T' observe une écriture de T puis T abandonne $\Rightarrow T'$ doit être abandonnée



Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
 - Type d'incohérences
- 2 Atomicité/Tout ou rien
- 3 **Contrôle de concurrence**
 - Principe
 - Modélisation
 - Méthodes
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



Contrôle de concurrence

Objectif

Assurer une protection contre les interférences entre transactions identique à celle obtenue avec l'exclusion mutuelle, tout en autorisant une exécution concurrente (autant que possible)

- 1 Exécution **sérialisée** : isolation par exclusion mutuelle.
- 2 Exécution **sérialisable** : contrôler l'entrelacement des actions pour que *l'effet final* soit équivalent à une exécution sérialisée.
 - Il peut exister plusieurs exécutions sérialisées équivalentes
 - Contrôle automatique



Conflits

Conflit : opérations non commutatives exécutées sur un même objet

Exemple

Avec opérations Lire(x) et Écrire(x,v) :

- conflit LL : non
- conflit LE : $T_1.\text{lire}(x)$; ... ; $T_2.\text{écrire}(x,n)$;
- conflit EL : $T_1.\text{écrire}(x,n)$; ... ; $T_2.\text{lire}(x)$;
- conflit EE : $T_1.\text{écrire}(x,n)$; ... ; $T_2.\text{écrire}(x,n')$;

Autres conflits

La notion de conflit n'est pas spécifique aux opérations Lire/Écrire.

Exemple

	lire	écrire	incrémenter	décrémenter
lire	OK	–	–	–
écrire	–	–	–	–
incrémenter	–	–	OK	OK
décrémenter	–	–	OK	OK



Graphe de dépendance, sérialisabilité

Définition

Relation de dépendance \rightarrow : $T_1 \rightarrow T_2$ ssi une opération de T_1 précède et est en conflit avec une opération de T_2 .

Définition

Graphe de dépendance : relations de dépendance pour les transactions déjà validées.

Théorème

Exécution sérialisable : une exécution est sérialisable si son graphe de dépendance est acyclique.



Moins que sérialisable ?

La sérialisabilité est parfois un critère trop fort : **cohérence faible**.
SQL définit quatre niveaux d'isolation :

- Serializable : cohérence complète
- Repeatable_read : lectures fantômes acceptées
- Read_committed : lectures non répétables ou fantômes acceptées
- Read_uncommitted : lectures sales, non répétables ou fantômes acceptées



Contrôle de concurrence

Quand vérifier la sérialisabilité :

- 1 à chaque terminaison d'une transaction
(contrôle par certification ou optimiste)
- 2 à chaque nouvelle dépendance
(contrôle continu ou pessimiste)

Comment garantir la sérialisabilité :

- 1 Utilisation explicite du graphe de dépendance
- 2 Fixer/observer un ordre sur les transactions qui garantit l'absence de cycle : estampilles, verrous



Certification (concurrence explicite)

Algorithme

```
-- écritures en mémoire privée avec recopie à la validation  
T.lus, T.écrits : objets lus/écrits par T  
T.concur : transactions ayant validé pendant l'exéc. de T  
actives : ensemble des transactions en cours d'exécution
```

```
procédure Certifier(T) :  
si ( $\forall T' \in T.concur : T.lus \cap T'.écrits = \emptyset$ )  
alors  
    -- T peut valider  
     $\forall T' \in actives : T'.concur \leftarrow T'.concur \cup \{T\}$   
sinon  
    -- abandon de T  
fin
```

Protocole coûteux et coût du rejet \Rightarrow faible taux de conflit



Certification (estampille)

Algorithme

```
-- écritures en mémoire privée avec recopie à la validation
C : nbre de transactions certifiées
T.déb : valeur de C au début de T
T.fin : valeur de C à la fin de T
T.val : valeur de C si T certifiée
T.lus, T.écrits : objets lus/écrits par T
```

```
procédure Certifier(T) :
si ( $\forall T' : T.déb < T'.val < T.fin : T.lus \cap T'.écrits = \emptyset$ )
alors
    C  $\leftarrow$  C + 1
    T.val  $\leftarrow$  C
sinon
    abandon de T
fin
```

Protocole coûteux et coût du rejet \Rightarrow faible taux de conflit

Contrôle continu par estampilles

Ordre de sérialisation = ordre des estampilles

Algorithme

T.E : estampille de T

O.lect : estampille du plus récent lecteur de O

O.réd : estampille du plus récent écrivain de O

procédure lire(T,O)

si $T.E \geq O.réd$

alors

lecture de O possible

$O.lect \leftarrow \max(O.lect, T.E)$

sinon

abandon de T

finsi

procédure écrire(T,O,v)

si $T.E \geq O.lect \wedge T.E \geq O.réd$

alors

écriture de O possible

$O.red \leftarrow T.E$

sinon

abandon de T

finsi

Estampille fixée au démarrage de la transaction ou au 1^{er} conflit.

Estampilles : amélioration

Réduire les cas d'abandons. Exemple : règle de Thomas

Algorithme

```
procédure écrire(T,0,v)
si T.E  $\geq$  0.lect
alors
    action sérialisable : écriture possible
    si T.E  $\geq$  0.réd
        écriture effective
        0.red  $\leftarrow$  T.E
    sinon
        rien : écriture écrasée par transaction plus récente
    finsi
sinon
    abandon de T
finsi
```

Contrôle continu par verrouillage à deux phases

Ordre de sérialisation = ordre chronologique d'accès aux objets

$T_1 \rightarrow T_2$: bloquer T_2 jusqu'à ce que T_1 valide.

Verrous en lecture/écriture/...

Si toute transaction est

- bien formée (prise du verrou avant une opération)
- à deux phases (pas de prise de verrou après une libération)

phase 1 : acquisitions et opérations

point de validation

phase 2 : libérations

alors la sérialisation est assurée.

Note : et l'interblocage ? Cf gestion de la contention.



Verrouillage à deux phases : justification du protocole

Idée de base

Lorsque deux transactions sont en conflit, toutes les paires d'opérations en conflit sont exécutées dans le même ordre
→ pas de dépendances d'orientation opposée → pas de cycle

Schéma de preuve

- Notation :
 $e_1 \prec e_2 \equiv$ l'événement e_1 s'est produit avant l'événement e_2
- $T_i \rightarrow T_j \Rightarrow \exists O_1 : T_i.\text{libérer}(O_1) \prec T_j.\text{verrouiller}(O_1)$
- $T_j \rightarrow T_i \Rightarrow \exists O_2 : T_j.\text{libérer}(O_2) \prec T_i.\text{verrouiller}(O_2)$
- T_i à deux phases $\Rightarrow T_i.\text{verrouiller}(O_2) \prec T_i.\text{libérer}(O_1)$
- donc, T_j n'est pas à deux phases (contradiction), car :
 $T_j.\text{libérer}(O_2) \prec T_i.\text{verrouiller}(O_2) \prec T_i.\text{libérer}(O_1) \prec T_j.\text{verrouiller}(O_1)$

Verrouillage à deux phases

- Condition suffisante \Rightarrow restrictions « inutiles » du parallélisme
- Que faire en cas de conflit de verrouillage :
 - Abandon systématique \Rightarrow famine
 - Blocage systématique \Rightarrow interblocage
 - ordre sur la prise des verrous (classes ordonnées)
 - prédéclaration de *tous* les verrous nécessaires (pour les prendre tous ensemble atomiquement)
 - Soit un conflit $T_1 \rightarrow T_2$
 - wait-die : si T_2 a démarré avant T_1 , alors **bloquer** T_2
sinon **abandonner** T_2 .
 - wound-wait : si T_2 a démarré avant T_1 alors **abandonner** T_1
sinon **bloquer** T_2 .



Verrouillage à deux phases strict

- Prise implicite du verrou au premier accès à une variable
 - Libération automatique à la validation/abandon
-
- Garantit simplement les deux phases
 - Tout se fait à la validation : simple
 - Restriction du parallélisme
(conservation des verrous jusqu'à la fin)



Conclusion

- Chaque méthode a son contexte d'application privilégié
- Paramètres déterminants
 - taux de conflit
 - durée des transactions
- Résultats
 - peu de conflits → méthodes optimistes
 - nombreux conflits/transactions longues
→ verrouillage à deux phases
 - situation intermédiaire pour l'estampillage
- Simplicité de mise en œuvre du verrouillage à deux phases
→ choix le plus courant en base de données



Lectures invisibles

Lectures invisibles

Une opération de lecture est invisible si elle ne modifie que des données locales à la transaction.

Propriété importante pour la performance (pas de données partagées autre que celles effectivement partagées par les transactions), mais les écritures ne voient donc plus les lectures.

Contrôle mixte

- Contrôle continu pour les conflits EE (verrous),
- Certification pour les conflits LE (estampilles).



Gestionnaire de contention

Contention

En cas de conflit :

- 1 quelle transaction bloquer ?
- 2 quelle transaction annuler ?
- 3 éventuellement quelle transaction redémarrer et quand ?

Garantir la progression optimale **et** l'absence d'interblocage \Rightarrow d'innombrables stratégies.



Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
 - Type d'incohérences
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



Mémoire transactionnelle

- Introduire la notion de transactions au niveau du langage de programmation
- Objectif : se passer des verrous habituellement utilisés pour protéger les variables partagées
 - plus nécessaire d'identifier la bonne granularité des verrous
 - interblocage, etc : c'est le problème de la STM
 - gestion des priorité, etc : idem



Intégration explicite dans un langage

Exposer l'interface de manipulation des transactions et des accès.

Interface exposée

```
do {  
    tx = StartTx();  
    int v = tx.ReadTx(&x);  
    tx.WriteTx(&y, v+1);  
} while (! tx.CommitTx());
```



Intégration dans un langage

Introduire un bloc atomique :

atomically

```
atomically {  
    x = y + 2;  
    y = x + 3;  
}
```

(analogue aux régions critiques, sans déclaration explicite des variables partagées)



Transaction et synchronisation

Synchronisation \Rightarrow blocage \Rightarrow absence de progression de la transaction \Rightarrow absence de progression d'autres transactions \Rightarrow interblocage.

Intégrer la synchronisation dans les transactions

Abandonner la transaction pour la **redémarrer automatiquement** quand certaines des valeurs lues auront changé.

retry

```
procédure consommer
  atomically {
    if (nbÉlémentsDisponibles > 0) {
      // choisir un élément et l'extraire
      nbÉlémentsDisponibles--
    } else {
      retry;
    }
  }
}
```

Transaction et synchronisation 2

Exécuter une autre transaction si la première échoue :

```
orElse
  atomically {
    // consommer dans le tampon 1
  }
  orElse
  atomically {
    // consommer dans le tampon 2
  }
```



Cohérence interne

Sémantique définie sur les transactions validées (sérialisabilité) ou toutes (opacité)?

```
                                init x=y
atomic {                          atomic {
    if (x != y)                   x++;
        while (true) {}           y++;
}                                  }
```

```
                                int *x; bool nonnul;
atomic {                          atomic {
    if (nonnul)                   x ← NULL;
        *x ← 3;                   nonnul ← false;
}                                  }
```

Transaction zombie (ou condamnée) mais visible.

Interaction avec code non transactionnel

Lectures non répétables

Donnée qui change de valeur

```
atomic {  
  a := lire(x);  
  b := lire(x);  
} || écrire(x,100);
```

Lectures sales

Écritures abandonnées mais observées

```
atomic {  
  écrire(x,100);  
  abandon;  
} || b := lire(x);
```


Actions non annulables

Une transaction annulée doit être sans effet : comment faire s'il y a des effets de bords (p.e. entrées/sorties) ?

- 1 Interdire : uniquement des lectures/écritures de variables.
- 2 Ignorer le problème en considérant ces opérations comme des nop, et tant pis si la transaction est annulée.
- 3 **Irrévocabilité** : quand une transaction invoque une action non défaisable/non retardable, la transaction devient irrévocable : ne peut plus être annulée une fois l'action effectuée.
- 4 Intégrer dans le système transactionnel : monade d'IO d'Haskell



Transaction et exception

Exception dans une transaction ?

- Une transaction est une séquence tout ou rien de code ;
- La levée d'une exception dans un bloc doit sortir immédiatement du bloc.

- 1 **Valider** la transaction : considérer l'exception comme une branche conditionnelle.
Simple à mettre en œuvre, ne change pas la sémantique d'un code séquentiel.
- 2 **Annuler** la transaction : considérer l'exception comme abort.
Simplifie la composition et l'utilisation de librairie : dans `atomic { s.foo(); s.bar(); }`, si `bar` échoue à cause d'une exception, rien n'a eu lieu.
Mais si l'exception est due au code de la transaction, la cause de l'exception disparaît à l'annulation!


Imbrication

Transaction imbriquée

Transaction s'exécutant dans le contexte d'une transaction parente.

```
init x = 1  
atomic{ x ← 2; atomic{ x ← x+1; abort/commit;} ...}
```

- 1 Une seule transaction fille active ou plusieurs (parallélisme) ?
- 2 Dans la fille, visibilité des effets de la transaction parente ?
- 3 L'annulation de la fille entraîne l'annulation de la parente ?
- 4 Les effets de la fille sont-ils visibles dès sa validation (ou seulement lors de la validation de la parente) ?

À plat : 3 oui, 4 non ; fermée : 3 non, 4 non ; ouverte : 3 non, 4 oui. 

STM – Software Transactional Memory

Implantation purement logicielle de la mémoire transactionnelle.

Interface explicite

- StartTx, CommitTx, AbortTx
- ReadTx(T *addr), WriteTx(T *addr, T v),

Programmation explicite, ou insertion par le compilateur.

Points critiques

- Connaissances des accès *read-set*, *write-set*
- Journal \Rightarrow copie supplémentaire (*undo-log*, *redo-log*) ou double indirection (*shadow copy*)
- Meta-data associées à chaque objet élémentaire \Rightarrow granularité
- Efficacité

Nombreuses implantations, beaucoup de variété.



HTM – Hardware Transactional Memory

Instructions processeur

- `begin_transaction`, `end_transaction`
- Accès explicite (`load/store_transactional`) ou implicite (tous)

Accès implicite \Rightarrow code de bibliothèque automatiquement pris en compte + isolation forte

Implantation

- *read/write-set* : pratiquement le rôle du cache
- détection des conflits \approx cohérence des caches
- *undo/redo-log* : dupliquer le cache



HTM – limitations

- Pas de changement de contexte pendant une transaction
- Petites transactions (2 ou 4 mots mémoire)
- Granularité fixée = unité d'accès (1 mot)
- Faux conflits dus à la granularité mot \leftrightarrow ligne de cache
- Grande variété des propositions, sémantique incertaine
- Portabilité d'un code prévu pour une implantation ?



Implantation hybride

Coopération STM/HTM

- Petites transactions en HTM, grosses en STM
- Problème : détection d'inadéquation de l'HTM, basculement ?
- Problème : sémantiques différentes

Implantation STM sur HTM

- Une HTM pour petites transactions
- Implantation de la STM avec les transactions matérielles
- HTM non visible à l'extérieur

Implantation STM avec assistance matérielle

- Identifier les *bons* composants élémentaires nécessaires ⇒ implantation matérielle
- Cf Multithread / contexte CPU ou Mémoire virtuelle / MMU
- Encore à creuser

Conclusion

- + simple à appréhender
- + réduction des bugs de programmation
- + nombreuses implantations portables en logiciel
- + compatible avec la programmation événementielle
- nombreuses sémantiques, souvent floues (mais ce n'est pas pire que les modèles de mémoires partagées)
- surcoût d'exécution
- effet « polluant » des transactions
- questions pas résolues (pas solvables?) : code hors transaction, composition

