Huitième partie

Transactions

Contenu de cette partie

- Nouvelle approche : programmation concurrente «déclarative»
- Mise en œuvre de cette approche déclarative : notion de transaction (issue du domaine des SGBD)
- Protocoles réalisant les propriétés de base d'un service transactionnel
  - Atomicité (possibilité d'annuler les effets d'un traitement)
  - Isolation (non interférence entre traitements)
- Adaptation de la notion de transaction au modèle de la programmation concurrente avec mémoire partagée (mémoire transactionnelle)

Plan

Comment garantir la cohérence d'activités concurrentes ?

Situation

≪ objet ≫ partagé, utilisé simultanément par plusieurs processus

Problème

garantir que cet objet est ≪ correctement utilisé ≫

Points de vues/approches possibles

- mise en œuvre directe : *synchronisation*, coordination des actions des différents processus, contrôlant explicitement l'attente/la progression des processus
- utilisation d'un service/abstraction général : *concurrence*. Dans ce cas, chaque processus interagit avec l'objet partagé comme s'il était seul à l'utiliser : le partage est transparent.

Cohérence et concurrence dans le contexte des SGBD

Contexte : traitements concurrents / données partagées

- données partagées, existant indépendamment des traitements
- système ouvert : les traitements ne sont pas connus a priori
  → chaque traitement doit pouvoir être conçu indépendamment

→ approche analogue à celle suivie pour la synchronisation :

- Caractérisation des utilisations concurrentes correctes (cohérentes) par un ensemble d'états possibles/permis pour les données partagées, que tout traitement doit respecter.
- Cet ensemble est défini en intention par un prédicat d'état : contrainte(s) d'intégrité, invariant

Abstraction de la concurrence : service transactionnel

Service de gestion des accès concurrents aux données partagées

- basé sur la notion d'état cohérent
- déclaratif : le programmeur doit simplement indiquer les traitements (transactions) pour lesquels la cohérence doit être garantie par le service transactionnel.

Définition de base : transaction

Suite d'opérations qui, exécutée seule
à partir d'un état initial cohérent, aboutit à un état final cohérent

Domaines d'utilisation

- Systèmes d'information : bases de données → intergiciels
- Mémoire transactionnelle (architectures mutiprocesseurs) (HTM/STM = hardware/software transactional memory)

Interface du service

- `tdébut()/tfin()` : parenthésage des opérations transactionnelles
- `tabandon()` : annulation des effets de la transaction ;
- `técrire(...), tlire(...)` : accès aux données.
  (Opérations éventuellement implicites, mais dont l'observation est nécessaire au service transactionnel pour garantir la cohérence)

Contrat du service transactionnel : propriétés ACID

Cohérence	toute transaction maintient les contraintes d'intégrité La validité sémantique est du ressort du programmeur
Isolation	pas d'interférences entre transactions : les états intermédiaires d'une transaction ne sont pas observables par les autres transactions. → modularité
Atomicité	ou « tout ou rien » : en cas d'abandon (volontaire ou subi) tous les effets d'une transaction sont annulés
Durabilité	permanence des effets d'une transaction validée

Exemple : base de données bancaires

- Données partagées : ensemble des comptes (X,Y ...)
- Contraintes :
  - la somme des comptes est constante (X.val + Y.val = Cte)
  - chaque compte a un solde positif (X.val ≥ 0 et Y.val ≥ 0)
- Transaction :
  virement d'une somme S du compte X au compte Y

```
tdébut
  si S > X.val alors
    "erreur" ;
  sinon
    X.val := X.val - S;
    Y.val := Y.val + S;
  finsi ;
tfin
```

Remarques :

- les états intermédiaires ne sont pas forcément cohérents
- expression déclarative : parenthésage par `tdébut/tfin`

Plan

Atomicité (tout ou rien)

Objectif

- Intégrer les résultats des transactions « bien » terminées
- Assurer qu'une transaction annulée n'a aucun effet sur les données partagées

Difficulté

Tenir compte de la possibilité de pannes en cours

- d'exécution,
- ou d'enregistrement des résultats définitifs,
- ou d'annulation.

Mise en œuvre de l'atomicité

Opérations de base

- *défaire* : revenir à l'état initial d'une transaction annulée
- *refaire* : restaurer l'état atteint par une transaction annulée temporairement ou une (in)validation interrompue

Réalisation des opérations *défaire* et *refaire*

Basée sur la gestion d'un *journal*, conservé en *mémoire stable*.

- Contenu d'un enregistrement du journal :
  [date, id. transaction, id. objet, valeur avant (et/ou valeur après)]
- Utilisation des journaux
  - *défaire* → utiliser les avant pour revenir à l'état initial
  - *refaire* → utiliser les valeurs après pour rétablir l'état atteint
- Remarque : en cas de panne durant une opération *défaire* ou *refaire*, celle-ci peut être reprise du début.

Approche optimiste : propagation en continu

Utilisation d'un journal des valeurs avant

- técrire → écriture directe en mémoire permanente
- valider (tfin) → effacer les images avant
- défaire (tabandon) → utiliser le journal avant
- refaire → sans objet (validation sans pb)

Problèmes liés aux abandons

- Rejets en cascade

(1) técrire(x,10)	(2) tlire(x)
	(3) técrire(y,8))
(4) tabandon()	→ abandonner aussi

- Perte de l'état initial

initialement : x=5

(1) técrire(x,10)	(2) técrire(x,8))
(3) tabandon()	(4) tabandon() → x=10 au lieu de x=5

Remède (?) : bloquer les accès en conflit avec técrire(x,-) → pas de parallélisme

Approche pessimiste : propagation différée

Utilisation d'un journal des valeurs après

Principe

- Ecriture dans un espace de travail, en mémoire volatile
  → adapté aux mécanismes classiques de gestion mémoire (caches...)
- Journalisation de la validation

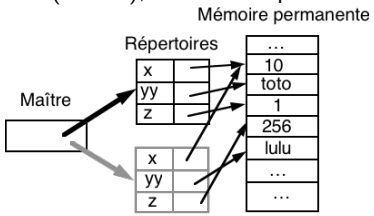
- écrire → préécriture, dans l'espace de travail
- valider → recopier l'espace de travail en mémoire stable (*liste d'intentions*), puis copier celle-ci en mémoire permanente
  → protection contre les pannes en cours de validation
- défaire → libérer l'espace de travail
- refaire → reprendre la recopie de la liste d'intentions

Technique sans journal (SGBD/SGF) : pages d'ombre

Principe

- Hypothèse : les blocs de données sont accessibles indirectement, via des blocs/pages d'index
- Chaque transaction dispose d'une copie des blocs d'index

- técrire → écriture en mémoire rémanente, via la copie des index
- défaire → purger la copie
- valider → remplacer l'original par la copie, de manière atomique
  Garantir l'atomicité → pages d'ombre : la copie valide est repérée par un enregistrement (maître), écrit atomiquement



- refaire → sans objet (validation atomique)

Plan

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 (si possible)

→ recherche d'un résultat final identique à celui qui aurait été obtenu en exécutant les transactions en exclusion mutuelle

Terminologie

- Exécution sérialisée : isolation par exclusion mutuelle.
- Exécution sérialisable : contrôler l'entrelacement des actions pour que *l'effet final* soit équivalent à une exécution sérialisée.

Remarque : Il peut exister plusieurs exécutions sérialisées équivalentes ; il suffit qu'il en existe au moins une.

Comment vérifier la sérialisabilité ?

Problème

On considére une exécution concurrente $(T_1||T_2||\ldots||T_n)$ d'un ensemble de transactions $\{T_1,\ T_2\ \ldots\ T_n\}$.
Cette exécution donne-t-elle le même résultat que l'une des exécutions en série $(T_1\,;\ T_2\,;\ \ldots\ ;\ T_n)$, ou $(T_2\,;\ T_1\,;\ \ldots\ ;\ T_n)$, ... ?

Idée

- le résultat de l'exécution de $(T_1||T_2||\ldots||T_n)$ sera celui d'un entrelacement des opérations de $\{T_1,\ T_2\ \ldots\ T_n\}$.
- si les différents entrelacements donnent le même résultat, alors toutes les exécutions série, et toutes les exécutions de $(T_1||T_2||\ldots||T_n)$ donneront le même résultat.
  $\Rightarrow$ pour vérifier la sérialisabilité, on peut se limiter aux opérations dont l'ordre d'exécution influence le résultat.

Notion de conflit

L'ordre d'exécution change-t-il le résultat ?

- `x := y/2  || u := w + v` → non
- `x := y/2  || y := y + 1` → oui
- `y := y + 1 || y := y + 1` → non

→ opérations en *conflit* :

opérations non commutatives exécutées sur un même objet

Exemple principal : opérations *lire(x)* et *écrire(x,v)*

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

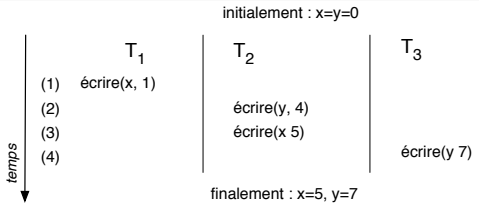
Remarque : la notion de conflit n'est pas spécifique à *lire/écrire*

→ généralisation : définir un tableau de commutativité entre actions
Exemple : lire, écrire, incrémenter, décrémenter

Graphe de dépendance

Idée : Les conflits déterminent l'ordre série équivalent, s'il existe.

Exemple



Dans toute exécution série donnant le même résultat,

- on doit trouver T_1 avant T_2 (sinon, x=1 au final)
- on doit trouver T_2 avant T_3 (sinon, y=4 au final)

Règle générale : s'il existe une exécution série S donnant le même résultat qu'une exécution concurrente $C = (T_1||T_2||\ldots||T_n)$, **alors** lorsqu'une opération op_i de T_i est en conflit avec une opération op_k de T_k, et que op_i a été exécutée avant op_k, T_i se trouve nécessairement avant T_k dans S (sinon le résultat final de S serait différent de celui de C)

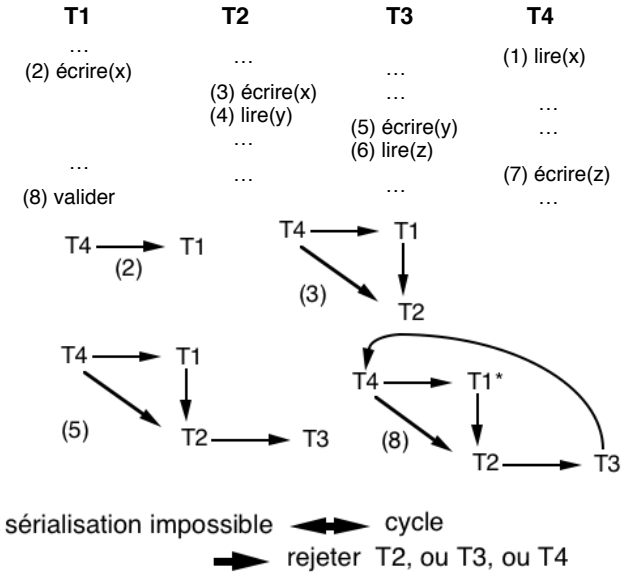
Définitions

- Relation de dépendance → : $T_1 \to T_2$ ssi une opération de T_1 précède et est en conflit avec une opération de T_2.
- Graphe de dépendance : relations de dépendance pour les transactions déjà validées.

Théorème (Critère de sérialisabilité [Papadimitriou])

Exécution sérialisable ⇔ son graphe de dépendance est acyclique.

Exemple



Méthodes de contrôle de concurrence

Quand vérifier la sérialisabilité ?

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

Comment garantir la sérialisabilité ?

- utilisation explicite du graphe de dépendance (coûteux)
- définir un ordre sur les transactions ($\rightarrow$ acyclicité) et bloquer/rejeter toute (trans)action introduisant une dépendance allant à l'encontre de cet ordre
  - ordre arbitraire $\rightarrow$ estampilles
  - ordre chronologique d'accès $\rightarrow$ verrous (méthodes continues)

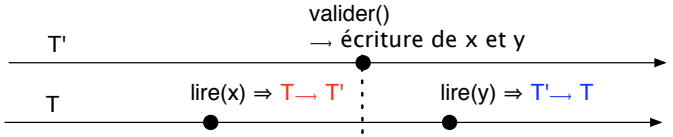
Contrôle de concurrence par certification : principe

- **Hypothèse** : écritures en mémoire privée avec recopie à la validation
- ordre de sérialisation = ordre de validation
- une transaction valide s'il est certain que ses conflits avec les transactions ayant déjà validé suivent l'ordre de validation

T demande à valider $\rightarrow$ écritures de T pas encore effectives

Conflits possibles entre T et les transactions validées :

- tous les conflits EE suivent l'ordre de validation : les écritures validées précèdent celles de T (qui n'ont pas eu lieu)
- conflits LE
  - les lectures validées précèdent toujours les écritures de T
  - mais il n'est pas certain que les écritures validées précèdent toujours les lectures de T



Contrôle de concurrence par certification : algorithme

Algorithme

```
C : nb de transactions certifiées  (ordonne les transactions)
T.déb, T.fin : valeurs de C au début et 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 (∀ T' : T.déb < T'.val < T.fin : T.lus ∩ T'.ecrits = ∅)
alors
    C ← C + 1
    T.val ← C
sinon
    tabandon(T)
finsi
```

Contrôle continu : estampilles

ordre de sérialisation = ordre des estampilles

→ pour toute transaction T, les accès de T doivent passer après ceux de toutes les transactions d'estampille inférieure à celle de T

Algorithme

```
T.E : estampille de T
O.lect : estampille du plus «récent» (grand) lecteur de O
O.réd : estampille du plus «récent» (grand) écrivain de O

procédure lire(T,O)                 | procédure écrire(T,O,v)
si T.E ≥ O.réd                      | si T.E ≥ O.lect ∧ T.E ≥ O.réd
alors /* lecture de O possible */   | alors /* écriture de O possible */
   lecture effective                |    écriture effective
   O.lect ← max(O.lect,T.E)         |    O.red ← T.E
sinon                               | sinon
   abandon de T                     |    abandon de T
finsi                               | finsi
```

Estampille fixée au départ de la transaction ou au premier conflit.

Estampilles : remarques (1/2)

Amélioration : réduire les cas d'abandons.

Algorithme (règle de Thomas)

```
procédure écrire(T,O,v)
si T.E ≥ O.lect alors
   /* action sérialisable : écriture possible */
   si T.E ≥ O.réd alors
       écriture effective
       O.red ← T.E
   sinon
       rien : écriture écrasée par transaction plus récente
   finsi
sinon
   abandon de T
finsi
```


Estampilles : remarques (2/2)

Les opérations `lire(...)` et `écrire(...)` peuvent devoir être complétées/adaptées, en fonction de la politique de propagation :

- propagation continue (optimiste)
  → gérer les abandons en cascade
- propagation différée (pessimiste)
  → les écritures effectives n'ont lieu qu'en fin de transaction.
  Par conséquent
  - les estampilles d'écriture (`O.red`) ne peuvent être fixées qu'au moment de la validation
  - les tests relatifs aux opérations d'écriture doivent être (ré)effectués à la terminaison de la transaction

Contrôle continu : verrouillage à deux phases

Verrous en lecture/écriture :
si $T_1 \rightarrow T_2$, T_2 est bloquée jusqu'à ce que T_1 valide.

Ordre de sérialisation = ordre chronologique d'accès aux objets

Si toute transaction est

- bien formée (prise du verrou avant tout accès)
- à deux phases (pas de prise de verrou après une libération)
  `phase 1 : acquisitions et accès`
  `< point de verrouillage maximal >`
  `phase 2 : libérations`

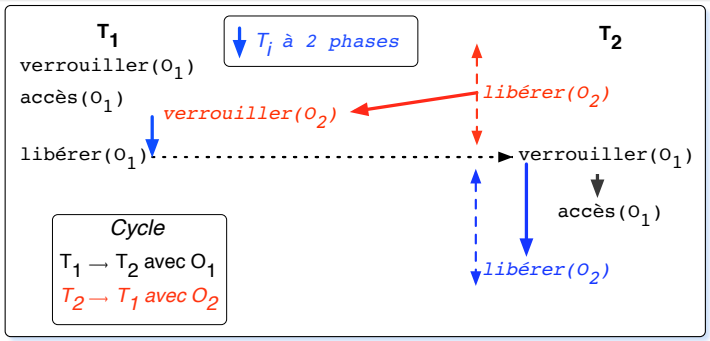
alors la sérialisation est assurée.

Ordre série = ordre d'apparition des points de verrouillage maximaux
(ordre des validations, dans le cas de 2PL strict, cf infra)

Verrouillage à deux phases : justification du protocole

Idée de base

Lorsque 2 transactions sont en conflit, tous les couples d'opérations en conflit et qui sont effectivement exécutées, sont toujours exécutés dans le même ordre
→ pas de dépendances d'orientation opposée → pas de cycle



Verrouillage à deux phases : schéma de preuve

Notation : $e_1 \prec e_2 \equiv$ l'événement e_1 s'est produit avant l'évt. 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 :
  $T_j.\text{libérer}(O_2) \prec T_i.\text{verrouiller}(O_2) \prec T_i.\text{libérer}(O_1) \prec T_j.\textit{verrouiller}(O_1)$

Verrouillage à deux phases : remarques

Mise en œuvre simple : 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 (verrous conservés jusqu'à la fin)

Emploi de verrous

→ restriction du parallélisme potentiel

→ restriction accrue par le report des libérations jusqu'à l'instant du point de verrouillage maximal.

→ risque d'interblocage

Verrouillage à deux phases : traitement de l'interblocage

Techniques classiques
- délai de garde (Tandem...)
- ordre sur la prise des verrous (classes ordonnées)
- prédéclaration (et prise atomique) de *tous* les verrous requis

Techniques particulières : utilisation des estampilles
pour prévenir la formation de cycles dans le graphe d'attente
- $T_i.E$ désigne l'estampille de T_i
- situation : T_i demande l'accès à un objet déjà alloué à T_j
- **wait-die** : si $T_i.E < T_j.E$, bloquer T_i, sinon abandonner T_i
  - attentes permises seulement dans l'ordre des estampilles
  - non préemptif
- **wound-wait** : si $T_i.E < T_j.E$, abandonner T_j, sinon bloquer T_i
  - attentes seulement dans l'ordre inverse des estampilles
  - préemptif ; équitable
  - amélioration : marquer T_j comme « blessée » et attendre qu'elle rencontre un second conflit pour l'abandonner

Conclusion (méthodes de CC) : comment garantir la cohérence *efficacement* ?

Objectif

Eviter d'évaluer la cohérence *globalement, et à chaque instant*

- *Evaluation épisodique*/périodique (après un ensemble de pas)
  → pouvoir annuler un ensemble de pas en cas d'incohérence
- *Evaluation approchée* : trouver une condition suffisante, plus simple à évaluer (locale dans l'espace ou dans le temps)
  → notions de sérialisabilité et de conflit
- *Relâcher* les exigences de cohérence, pour simplifier l'évaluation

  Exemple (BD) : SQL définit quatre niveaux d'isolation
  - *Serializable* : sérialisabilité proprement dite
  - *Repeatable_read* : possibilité de *lectures fantômes*
    (lorsqu'une transaction lit un *ensemble* de données la stabilité de cet ensemble n'est pas garantie : des éléments peuvent apparaître ou disparaître)
  - *Read_commited* : possibilité de lectures fantômes ou *non répétables*
    (la même donnée lue 2 fois de suite peut retourner 2 valeurs différentes)
  - *Read_uncommited* : possibilité de lectures fantômes, non répétables ou *sales*
    (lecture de données écrites par des transactions non validées)

Conclusion (2/2) : bilan sur les méthodes de contrôle de concurrence

- 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

Plan

Mémoire transactionnelle

But

Fournir un service de contrôle de l'accès concurrent à une mémoire partagée garantissant l'exécution atomique d'une série d'opérations

- niveau matériel : accès à une mémoire/un cache partagé sur un multiprocesseur/multicœur
- niveau logiciel : mécanisme (et service) de contrôle de concurrence des threads d'une application parallèle

Similitudes avec les bases de données

- situation : concurrence d'accès à des données partagées
  → système ouvert
- relation naturelle entre sérialisabilité et exclusion mutuelle : recherche/mise en œuvre d'une cohérence forte

Intérêt

Abstraction

gestion déclarative et automatique de la concurrence

→ élimine les risques d'erreur dans la programmation de la synchronisation : granularité des objets verrouillés, interblocage ; gestion des traitements en attente (ordonnancement, priorité, équité)

Compositionnalité

- l'exécution des transactions est indépendante : il est possible de lancer une nouvelle transaction à tout moment
  → adapté à un environnement ouvert, où l'ensemble des traitements exécutés évolue n'est pas connu à l'avance
- alors que la bonne utilisation des verrous dépend du comportement des autres traitements
  Exemple : prévention de l'interblocage

Intérêt (2)

Exécution spéculative

les protocoles optimistes de CC éliminent les blocages et accroissent donc le parallélisme *potentiel*.

- Réalisation simple de structures de données concurrentes non bloquantes. (Algorithmique très complexe sans transactions)
- Traitement efficace de volumes importants de données irrégulières/évolutives
  - parcours de graphes (sans transactions : algorithmique complexe ou verrou global)
  - simulation, jeux en réseau (évite un calcul préalable pour déterminer les objets voisins à verrouiller)

Remarque : il reste tout à fait possible de faire des erreurs de programmation : transactions trop longues (risque accru d'abandon), ou trop courtes (risque de mauvaise isolation)...

Intégration aux langages de programmation

Interface explicite 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 »

Bloc atomique (mot-clé atomically)

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

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

Application phare : « ellipse » des verrous

Idée

transformer les programmes existants
en remplaçant les sections critiques par des transactions

- motivation : les verrous sont pessimistes.
  → si les conflits sont peu nombreux (ce qui est courant), on réduit inutilement le degré de parallélisme
- expérimentation (Herlihy) sur une JVM implantant (de manière transparente) les blocs synchronized par des transactions.
  Résultats conformes aux prévisions : quelques applications nettement accélérées (facteur 5), une grande majorité modérément accélérées, quelques applications très ralenties (conflits nombreux)
- l'ellipse de verrous a des limites dans de nombreux cas : conflits fréquents, volumes mémoires importants (hors cache)

Poursuite de l'idée de l'ellipse de verrous : transactions et synchronisation

Traitement des conflits

- par la synchronisation → blocage (interactions explicites)
- par les transactions → annulation (interactions transparentes)

→ traduction de la synchronisation dans les transactions

1. Abandonner la transaction en cas de conflit
2. Attendre que des valeurs lues aient changé
3. Relancer (automatiquement) la transaction

→ opération retry

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

Remarque : schéma a priori inefficace et peu pertinent, en général...

Mise en œuvre : spécificités de la mémoire transactionnelle

Par rapport aux transactions « classiques » :
ordres de grandeur différents dans le nombre d'objets, de conflits et dans les temps d'accès

→ recherche d'efficacité :

- utilisation de protocoles simples
- utilisation de la propagation directe (éviter des recopies)
  - → réalisation du contrôle de concurrence plus complexe
    - nécessité de contrôler la propagation des valeurs et d'éviter les effets de bord des transactions annulées
    - → notion d'opacité : sérialisabilité + pas de dépendance par rapport aux transactions actives

MTL : Mémoire transactionnelle logicielle (STM)

Implantation purement logicielle de la mémoire transactionnelle.

Interface explicite

- Opérations sur les transactions : `Start()`, `Commit()`, `Abort()`
- Opérations sur les mots mémoire : `Read(Tx)`, `Write(Tx, val)`

Programmation explicite, ou insertion par le compilateur.

Exemple (1/3)

- Mémoire partagée = tableau Mem[0..Max] de mots mémoire
- Utilisation d'un service de verrous non bloquants, fournissant :
  - L.trylock_shared() demande L en mode partagé→ ok/échec
  - L.trylock() demande L en mode exclusif → ok/échec
  - L.unlock() libère L.
  - un verrou est associé à chaque mot mémoire
    → tableau global L[0..Max] de verrous

Exemple (2/3) : opérations sur la mémoire

Structures de données locales à chaque transaction T_k

- $SvMem[0..Max]$: valeur la mémoire avant T_k
- ensembles *lus*, *ecrits* : indices des mots accédés par T_k

Opérations

- $m.read(T_k)$

```
if m ∉ Tk.lus ∪ Tk.ecrits then
    if not L[m].trylock_shared() then abort(Tk);return "echec";endif;
    Tk.lus := Tk.lus ∪ {m} ;
endif
return Mem[m].read();
```

- $m.write(T_k, val)$

```
if m ∉ Tk.ecrits then
    if not L[m].trylock then abort(Tk); return "echec"; endif;
    Tk.ecrits := Tk.ecrits ∪ {m} ;
    Tk.SvMem[m] := Mem[m].read() ;
endif
Mem[m].write(val);
return "ok";
```

Exemple (3/3) : opérations sur les transactions

- *commit*(T_k)

  unlock_all(T_k);
  return "*ok*" ;

- *abort*(T_k)

  // *restaurer les valeurs ecrites*
  foreach $m \in T_k.ecrits$ do $Mem[m].write(T_k.SvMem[m])$;
  unlock_all(T_k);
  return "*ok*" ;

- *unlock_all*(T)

  // *liberer tous les verrous obtenus par* T
  foreach $m \in T.lus \cup T.ecrits$ do $L[m].unlock()$;
  $T.lus := \emptyset$;
  $T.ecrits := \emptyset$;
  return ;

MTM : Mémoire transactionnelle matérielle (HTM)

Instructions processeur

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

Accès implicite ⇒
code existant automatiquement pris en compte + isolation forte

Implantation

- *ensembles lus/écrits* : pratiquement le rôle du cache
- détection des conflits ≈ cohérence des caches
- *journal avant/après* : dupliquer le cache

MTM – limites

Basées sur l'utilisation des caches mémoire
→

- 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 ↔ ligne de cache
- code non portable (lié à un matériel donné)

Difficultés du modèle : quelques exemples (1/3)

Considérer les conflits avec les transactions validées (sérialisabilité) ou avec toutes (opacité) ?

Propagation directe ⇒ opacité

```
                    init x=y
atomic {                        | atomic {
    if (x != y)                 |     x++; /*(1)*/
        while (true) {}         |     y++; /*(4)*/
}                               | }
```

Difficultés (2/3) : interaction avec code non transactionnel

Lectures non répétables

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

Lectures sales : écritures abandonnées mais observées

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

→ garantir la cohérence ⇐ abandon si conflit hors transaction

Difficultés (3/3) : actions non annulables

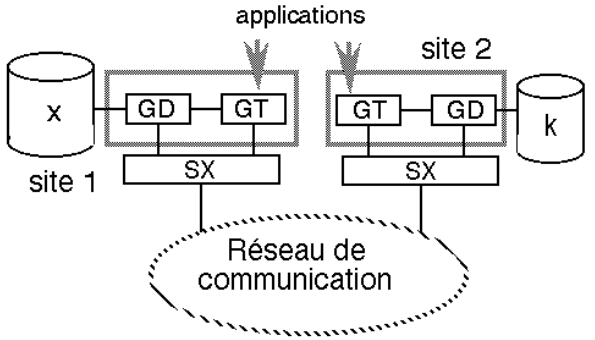
Une transaction annulée doit être sans effet : comment faire s'il y a des effets de bords (p.e. entrées/sorties), avec un contrôle de concurrence optimiste ?

1. Interdire : uniquement des lectures/écritures de variables.
2. 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.
3. virtualiser les actions irrévocables, pour les effectuer seulement après validation

Mémoire transactionnelle : conclusion

+ simple à appréhender
+ ellipse de verrous
+ réduction des erreurs de programmation
+ nombreuses implantations portables en logiciel
  - Java/C++/etc (externe au langage) : XSTM, Deuce, Multiverse
  - Clojure (langage fonctionnel compilé pour la JVM)
  - Haskell (langage fonctionnel)
– surcoût d'exécution, mais
  - la MT logicielle permet de tirer parti des multicœurs, → justifie un surcoût, même important
  - la MT logicielle peut être améliorée (p. ex. couplage avec les mécanismes de la MT matérielle)
– nombreuses sémantiques, souvent floues (mais ce n'est pas pire que les modèles de mémoire partagée)
– questions ouvertes : composition avec le code hors transaction, intégration de la synchronisation

Annexe : architecture de principe du service d'accès aux données



- le *noyau transactionnel* (GT) ordonnance et contrôle les accès aux données de manière à garantir l'atomicité et l'isolation. Les opérations d'accès aux données permises sont transmises
- au *gérant de données* (GD) (SGF ou SGBD) qui réalise les opérations d'accès aux données proprement dites (*traite les requêtes*, dans la terminologie BD)