Cohérence et réplication pour les données réparties

Exercices pour le Module 9

INF8480 Systèmes répartis et infonuagique

Michel Dagenais

École Polytechnique de Montréal

Département de génie informatique et génie logiciel

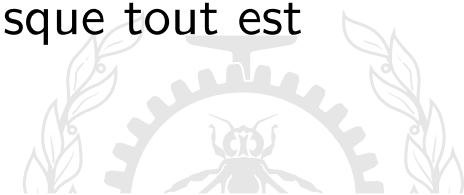
Cohérence vue des clients

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.

Cohérence vue des clients

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.

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.



Transactions concurrentes cohérentes

Un serveur gère plusieurs valeurs $a_1, \ldots a_n$. Ce serveur offre deux opérations à ses clients: `value = Read(i)`, `Write(i, value)`. Les transactions T et U sont définies comme suit. Donnez deux sérialisations équivalentes de ces transactions?

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);

Transactions concurrentes cohérentes

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

$Tx = a_j^0$; $Ty = a_i^0$; $Ux = a_k^0$; $Uy = 44$; $a_i = 55$; $a_j = 44$; $a_k = 66$;

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)

Si U procède avant T, le résultat final sera:

$Tx = a_j^0$; $Ty = 55$; $Ux = a_k^0$; $Uy = a_j^0$; $a_i = 33$; $a_j = 44$; $a_k = 66$;

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)	

Les verrous

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

Les 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	

Verrouillage en deux phases

Soit deux transactions T et U, lesquelles de ces sérialisations sont valides? Sont possibles avec un verrouillage en deux phases?

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

a. `T x=Read(i); U write(i,55); T write(j,44); U write(j,66);`
b. `U write(i,55); U write(j,66); T x=Read(i); T write(j,44);`
c. `T x=Read(i); T write(j,44); U write(i,55); U write(j,66);`
d. `U write(i,55); T x=Read(i); U write(j,66); T write(j,44);`

Verrouillage en deux phases

Soit deux transactions T et U, lesquelles de ces sérialisations sont valides? Sont possibles avec un verrouillage en deux phases?

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

a. `T x=Read(i); U write(i,55); T write(j,44); U write(j,66);`
b. `U write(i,55); U write(j,66); T x=Read(i); T write(j,44);`
c. `T x=Read(i); T write(j,44); U write(i,55); U write(j,66);`
d. `U write(i,55); T x=Read(i); U write(j,66); T write(j,44);`

Les 4 sérialisations proposées sont valides (*a* et *c* pour T avant U, *b* et *d* pour U avant T).

En *a* et *d*, le verrou sur *i* bloquerait la deuxième opération et ces sérialisations ne pourraient se produire.

Relâcher les verrous

Expliquez pourquoi il ne faut pas relâcher les verrous avant de se commettre, même après la fin des opérations.

Relâcher les verrous

Expliquez pourquoi il ne faut pas relâcher les verrous avant de se commettre, même après la fin des opérations.

Si une autre transaction utilise les valeurs modifiées et la première transaction échoue, l'autre transaction aura utilisé des valeurs incorrectes. Il serait possible cependant de relâcher les verrous de lecture.

Validation avec contrôle optimiste

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?

Validation avec contrôle optimiste

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.

Transactions imbriquées

Expliquez comment une transaction imbriquée s'assure que toutes les sous-transactions sont correctement commises ou annulées.

Transactions imbriquées

Expliquez comment une transaction imbriquée s'assure que toutes les sous-transactions sont correctement commises ou annulées.

Chaque sous-transaction maintient une liste des transactions prêtes à se commettre.

La transaction de premier niveau peut donc faire une recherche en profondeur des transactions prêtes pour la validation finale.

Journal de transactions

Les transactions T, U, et V sont définies comme suit. Décrivez le journal produit par cette application si un verrouillage en deux phases est utilisé et U acquiert i et j avant T? Décrivez comment ce journal peut être utilisé pour récupérer en cas de panne?

```
T: x = Read(i); Write(j,44);
U: Write(i,55); Write(j,66);
V: Write(k,77); Write(k,88);
```

Journal de transactions

Journal

```
P0:   ...; P1:   Write(i,55);
P2:   Write(j,66);
P3:   Prépare U(i:P1, j:P2), P0;
P4:   Compléter U, P3;
P5:   Write(j,44);
P6:   Préparer T(j:P5), P4;
P7:   Compléter T, P6;
P8:   Write(k,88);
P9:   Prépare V(k:P8), P7;
P10:  Compléter V, P9;
```

Journal de transactions

Journal

```
P0:   ...; P1:   Write(i,55);
P2:   Write(j,66);
P3:   Prépare U(i:P1, j:P2), P0;
P4:   Compléter U, P3;
P5:   Write(j,44);
P6:   Préparer T(j:P5), P4;
P7:   Compléter T, P6;
P8:   Write(k,88);
P9:   Prépare V(k:P8), P7;
P10:  Compléter V, P9;
```

- Pour récupérer, les variables $a_1, \ldots a_n$ sont replacées à leur valeur par défaut.
- Ensuite le journal est lu en reculant.
- Le récupérateur voit que V est complété et à P9 que V contient P8; il met donc a_k à 88.
- Il continue à reculer à P7, voit que T est complété et peut ensuite lire a_j à 44 en voyant P6 qui pointe à P5.
- La chaîne remonte ensuite à P4, où U est marqué complété; ceci fait reculer à P3 qui pointe vers une valeur de j qui est ignorée car la valeur la plus récente est déja lue, et vers une valeur pour i ce qui permet de trouver que a_i devient 55.

Journal et validation en reculant

Les transactions T et U utilisent un contrôle optimiste de la concurrence basé sur la validation en reculant. Décrivez l'information écrite dans le journal si T débute en premier.

Transac. T	**Transac. U**
x = Read(i)	
	Write(i, 55)
	Write(j, 66)
Write(j, 44)	
	Commit
Commit	

Journal et validation en reculant

Les transactions T et U utilisent un contrôle optimiste de la concurrence basé sur la validation en reculant. Décrivez l'information écrite dans le journal si T débute en premier.

Transac. T	Transac. U
x = Read(i)	
	Write(*i*, 55)
	Write(*j*, 66)
Write(j, 44)	
	Commit
Commit	

Journal

```
P0:  ...;
P1:  i = 55;
P2:  j = 66;
P3:  Prépare U(i:P1, j:P2), P0;
P4:  Commettre U, P3;
P5:  j = 44;
P6:  Prépare T(j:P5), P4;
P7:  Commettre T, P6;
```

- Lorsqu'une transaction est validée, le numéro de transaction apparaît dans le journal.
- U est validée car elle ne fait aucune lecture.
- Par contre, la transaction T est annulée car elle lit *i* qui est écrit par U.
- T est redémarrée et peut finalement compléter.

Verrous pour les transactions réparties

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.
2. Expliquez pourquoi les verrous ne suffisent pas à assurer la sérialisation.

Verrous pour les transactions réparties

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

- Supposons que B_m tombe en panne avant d'être verrouillé par U.
- U pourra procéder. La validation locale permet de vérifier si une copie qui n'a pas fonctionné n'a pas été recouvrée.
- Dans le cas de T, lorsque B_m est revenu et a déjà été lu, il découvre que le verrou détenu ne tient plus et la transaction doit être annulée.

Ordre local et global

Donnez un exemple d'ordonnancement entre deux transactions, T et U qui lisent et écrivent a et b, qui est cohérent sur chaque serveur mais incohérent globalement?

Ordre local et global

Serveur X, T passe avant U, **correct localement**:

Transaction T	Transaction U
Read(a)	
Write(a)	
	Read(a)
	Write(a)

Serveur Y, U passe avant T, **correct localement**:

Transaction T	Transaction U
	Read(b)
	Write(b)
Read(b)	
Write(b)	

Globalement il y a un problème:

- la transaction U pourrait lire la valeur initiale de *b*, puis la valeur modifiée par T de *a* pour les combiner en une nouvelle valeur de *b* et de *a*.
- Autrement dit, T précède U qui précède T pour respecter l'ordre sur chaque ordinateur, ce qui est impossible.