Multicore Programming — Projet Mémoire transactionnelle logicielle

Filaudeau Eloi - Louis Boursier
eloi.filaudeau@etu.univ-nantes.fr - louis.boursier@etu.univ-nantes.fr
Université de Nantes - Programmation multi-coeurs (MP:X2I3010)

Abstract

Ce document est un rapport du projet de mémoire transactionelle logicielle du module de programmation multi-coeurs. On discutera en particulier de la propriété de vivacité sur les transactions garantie par l'algorithme TL2.

1 L'algorithme TL2

Pour résoudre les problèmes de concurrence, c'est à dire les incohérences, qui peuvent apparaître lors d'une utilisation de ressources partagées par plusieurs threads, une approche classique est d'utiliser des verrous. Seulement, cette approche se révèle souvent soit inefficace avec un verrouillage gros grain, soit complexe lors d'un verrouillage grain fin. Une autre approche existe avec les solutions non bloquantes et les instructions spéciales, mais cette solution souffre des mêmes problèmes que la première. Une autre solution est la mise en place d'une transaction, où la section critique est délimitée par les appels aux fonctions begin et tryToCommit. Dans la section critique, on peut écrire et lire dans des registres à l'aide de fonctions read et write. Au moment de l'appel à tryToCommit, soit l'appel est un succès et les changements sont appliquées aux variables partagées, soit la transaction annule et c'est comme si il ne s'était rien passé. On peut alors tenter de renouveler la transaction jusqu'à ce qu'elle réussisse. Cette approche s'appelle la mémoire transactionnelle logicielle, et l'algorithme TL2 en fait partie.

Avec TL2, une écriture sur un registre par une transaction écrit en faite sur une copie locale ce registre dans cette transaction. Si cette copie n'existe pas, on la créer en faisant une copie du registre partagé.

Listing 1: write

```
1 public void write(Transaction transaction, Object value) {
2     if (transaction.getLocalCopies().get(this) == null) {
3         transaction.getLocalCopies().put(this, this.makeCopy());
4     }
5     transaction.getLocalCopies().get(this).setValue(value);
6     transaction.getLocallyWritten().add(this);
7     lastTransactionWriter = transaction; // used for the read method
8 }
```

Dans une transaction, on sauvegarde les copies locales des registres qu'on a lu ou écrit dans une map qui a pour clé les registres partagés et pour valeurs les copies locales de ces registres avec les valeurs qu'on veut soumettre.

Listing 2: localCopies

```
1 // maps a register to its local copy with new value
2 private Map<Register, Register> localCopies = new HashMap<>();
```

Ce n'est que lors du commit que la copie locale pourra être écrite sur le registre partagé, si cette transaction n'annule pas.

Lors d'une lecture, on retourne la valeur locale du registre si elle est à jour. Sinon, on la met à jour et on la retourne.

Listing 3: read

```
1 public Object read(Transaction transaction) throws CustomAbortException {
2     // if the register has been written by the same transaction, we return its value
3     if (transaction.getLocalCopies().get(this) != null && lastTransactionWriter == transaction) {
4         return transaction.getLocalCopies().get(this).value;
5     } else {
6         transaction.getLocalCopies().put(this, this.makeCopy());
7         transaction.getLocallyRead().add(this);
8         // transaction has possibly read other values that are no longer consistent
```

```
 9        // with the value of register just obtained, in that case we abort
10        if (transaction.getLocalCopies().get(this).date > transaction.getBirthdate()) {
11            throw new CustomAbortException("Date incoherence");
12        } else {
13            return transaction.getLocalCopies().get(this).value;
14        }
15    }
16 }
```

On a vu que la transaction sauvegarde tous les registres qu'elle a lu ou ceux dans lesquels elle a écrit. Dans cette sauvegarde du registre, on trouve la nouvelle valeur et la date de la dernière modification de ce registre. Cette date nous permet d'annuler le commit si la lecture des registres n'est pas cohérente les uns avec les autres. Cette vérification doit être linéarisable et thread safe. Nous adresserons ce problème dans une prochaine partie.

Listing 4: commit

```
1 /* checks if the current values of the objects register it has read are still mutually consistent,
2 and consistent with respect to the new values it has (locally) written
3 if one of these dates is greater than its birthdate,
4 there is a possible inconsistency and consequently transaction is aborted */
5 for (Register lrst : locallyRead) {
6     if (lrst.getDate() > birthdate) {
7         throw new CustomAbortException("Date incoherence");
8     }
9 }
10 Integer commitDate = EventuallyCommittedTest.globalClock.incrementAndGet();
11 for (Register register : locallyWritten) { // makes our local change visible to all the threads
12     register.setValue(localCopies.get(register).getValue());
13     register.setDate(commitDate);
14 }
15 isCommited = true;
```

2 Adapatation du Dictionnaire pour TL2

Le dictionnaire correspond à un set de String. Ces String sont ordonnées par ordre lexicographique , et on optimise la gestion de la mémoire en encodant les préfixes communs une seule fois.

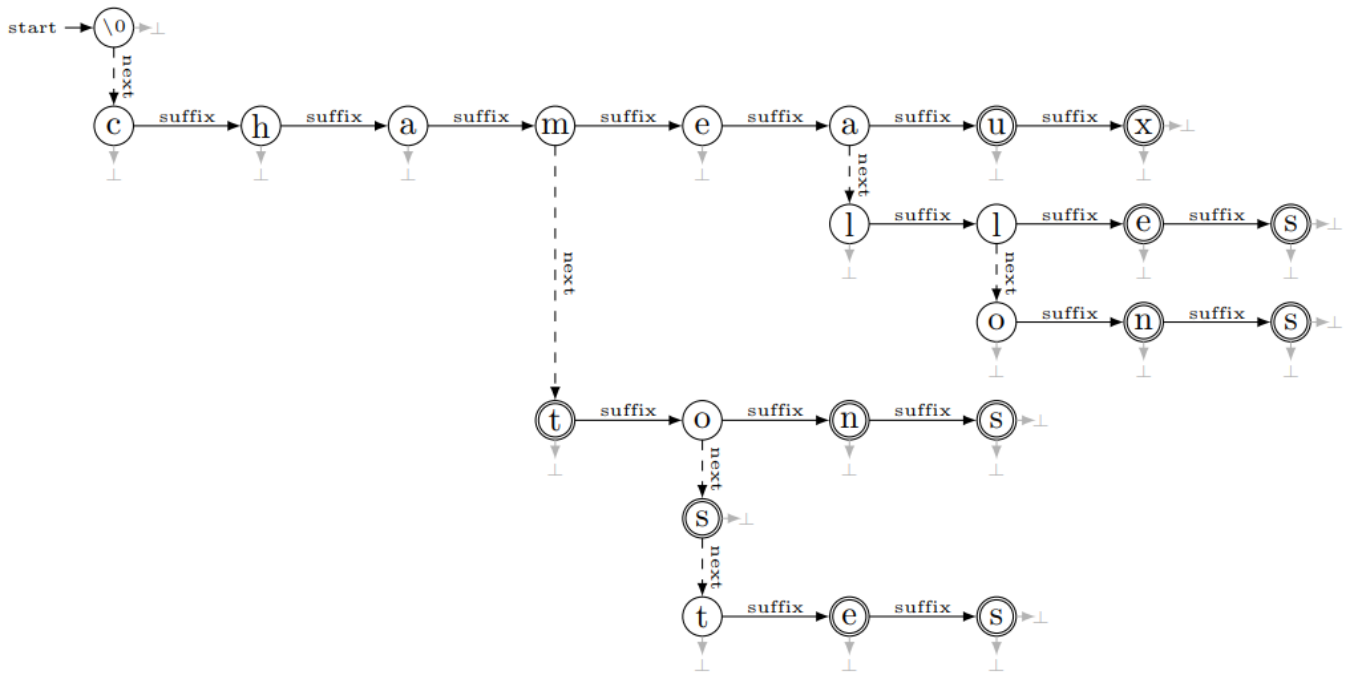

Figure 1: **Dictionnaire** par Matthieu Perrin

Cette structure de donnée a une seule fonction d'ajout, qui retourne faux si le dictionnaire contient le mot, et vrai sinon. Cette fonction n'est pas thread safe, car il peut y avoir plusieurs threads qui modifient le dictionnaire en même temps, générant des incohérences. Par exemple, un même mot peut être ajouté plusieurs fois au dictionnaire.

Une solution est de rendre cette structure de donnée immuable, comme si l'on voulait pouvoir faire de l'optimistic locking. C'est à dire lire le dictionnaire, y faire nos modifications, et soumettre nos modifications si le dictionnaire n'a

pas changé entre temps. Il faut que cette vérification soit atomique, pour cela, on utilise une AtomicReference<V> en Java, avec sa méthode compareAndSet(V expect, V update). Et on boucle tant que le compareAndSet ne réussie pas. Dans notre cas, c'est la même chose si ce n'est que les instructions spéciales sont remplacées par les transactions.

La fonction d'ajout est donc modifiée pour répondre à ce besoin. Elle retourne le même pointeur si l'ajout a échoué (la valeur était déjà contenue), ou un nouveau pointeur vers le dictionnaire modifié sinon. Ainsi, un thread ne peut pas modifier le dictionnaire d'un autre thread. La copie est faite en surchargeant la méthode clone. Le choix a été fait de faire une deep copy plutôt qu'une shallow copy, pour assurer la déclaration de la phrase précédente, au dépit de la performance.

```
1 public DictionaryImmutable add(String s) {
2    DictionaryImmutable newDictionary = null;
3    try {
4       newDictionary = (DictionaryImmutable) this.clone();
5    } catch (CloneNotSupportedException e) {
6       e.printStackTrace();
7    }
8    boolean addSuccessful;
9
10   if (s != "") {
11      addSuccessful = newDictionary.start.add(s, 0);
12   }else{
13      addSuccessful = newDictionary.emptyAbsent;
14      newDictionary.emptyAbsent = false;
15   }
16   return addSuccessful ? newDictionary : this;
17 }
```

3 Sur l'interblocage et la famine

Notre dictionnaire est prêt à être utilisé dans un contexte multi coeurs. Cependant, la fonction commit présentée au début n'est toujours pas thread safe. En effet, cette fonction doit être linéarisable pour éviter les problèmes de cohérence. Les threads doivent donc demander un verrou sur les registres qu'ils modifient. On pourrait encadrer la fonction dans un bloc synchronized, mais cela ne résoudrait pas les problèmes de famine. D'un autre côté, une mauvaise gestion des verrous peut entraîner des interblocages. Nous allons voir deux solutions possibles à ce problème.

3.1 Théorème de l'ordre sur les verrous

On peut établir l'ordre d'acquisition des verrous sur les registres sans interblocage grâce à une méthode utilisée pour le diner des philosophes. On associe un numéro à chaque verrou, et on oblige les transactions à demander les verrous dans l'ordre. Ainsi, il n'y aura pas d'interblocage. En elle même, cette solution ne résout pas la famine. Pour répondre à ce problème, on peut utiliser un ReentrantLock avec true en paramètre de constructeur. Cela a pour effet de garantir l'équité dans l'offre des verrous.

```
1 private void orderedLocking() throws CustomAbortException {
2
3    // fusions registers we need a lock on (locallyWritten & locallyRead)
4    List<Register> registers = makeUnionOfRegisters();
5
6    // orders registers by their number
7    Collections.sort(registers, (o1, o2) -> o1.getRegisterNumber() - o2.getRegisterNumber());
8
9    // asks for a lock
10   for (Register register : registers) { register.getLock().lock(); }
11
12   try {
13
14      for (Register lrst : locallyRead) {
15         if (lrst.getDate() > birthdate) { throw new CustomAbortException("Date incoherence"); }
16      }
17
18      Integer commitDate = EventuallyCommittedTest.globalClock.incrementAndGet();
19
20      for (Register register : locallyWritten) {
21         register.setValue(localCopies.get(register).getValue());
22         register.setDate(commitDate);
23      }
24
25      isCommited = true;
26
27   } finally {
28      for (Register register : registers) { register.getLock().unlock(); }
29   }
30 }
```

3.2 Intervention d'un arbitre

Une autre solution est l'intervention d'un arbitre, qui est un point de passage obligatoire pour l'acquisition des verrous. Ainsi, il peut facilement vérifier que la demande ne génère pas d'interblocage et peut résoudre la famine en utilisant une queue sur la demande de verrous par les threads. Cependant, cette méthode peut ralentir la parallélisation. En effet, l'arbitre ne donne normalement les verrous qu'à un seul thread à la fois pour éviter les problèmes. Dans notre implémentation, nous avons autorisé l'offre de verrous en parallèle si ceux ci sont disjoints. Dans notre cas, il n'y a qu'un seul registre pour le dictionnaire, alors cela ne change rien.

```
1 public class Arbitrator {
2
3    private static Set<Register> lockedRegisters = new CopyOnWriteArraySet<>();
4    private static ReentrantLock reentrantLock = new ReentrantLock(true);
5
6    public static boolean askForLocks(List<Register> registers) {
7       Arbitrator.reentrantLock.lock();
8       try {
9          for (Register r : registers) { if (lockedRegisters.contains(r)) return false; }
10         lockedRegisters.addAll(registers);
11         return true;
12      }finally {
13         Arbitrator.reentrantLock.unlock();
14      }
15   }
16
17   public static void releaseLocks(List<Register> registers) {
18      // if the set is thread safe, no lock needed for releasing the registers
19      lockedRegisters.removeAll(registers);
20   }
21 }
```

On utilise un set qui est thread-safe pour pouvoir libérer les registres sans avoir besoin du verrou de l'arbitre. Pour une raison encore à déterminer, le programme ne fonctionne pas avec une ConcurrentSkipListSet. On remarque qu'il est possible d'implémenter l'arbitre en lock free.

```
1 public class Arbitrator {
2
3    private static Set<Register> lockedRegisters = new CopyOnWriteArraySet<>();
4    private static ReentrantLock reentrantLock = new ReentrantLock(true);
5
6    public static boolean askForLocks(List<Register> registers) {
7       Arbitrator.reentrantLock.lock();
8       try {
9          for (Register r : registers) { if (lockedRegisters.contains(r)) return false; }
10         lockedRegisters.addAll(registers);
11      }finally {
12         Arbitrator.reentrantLock.unlock();
13      }
14      return true;
15   }
16
17   public static void releaseLocks(List<Register> registers) {
18      // if the set is thread safe, no lock needed for releasing the registers
19      lockedRegisters.removeAll(registers);
20   }
21 }
```