Programmation Multi-cœurs
Impossibilité du consensus

L'impossibilité du consensus est un théorème central du calcul réparti. Il a été démontré par Fischer, Lynch et Paterson en 1985 [1] pour les processus communiquant par messages, et ce même en supposant qu'au plus une panne peut se produire pendant une exécution. La preuve que nous exposons ici est une simplification dans le cas de la programmation multi-cœurs par Maurice Herlihy [2].

La figure 1 montre une implémentation du consensus qui utilise l'instruction spéciale `getAndSet`. La figure 2 présente, sous la forme d'un graphe, l'ensemble des exécutions possible.

Remarquez que, dans l'état 10, que T_0 ou T_1 fasse le prochain pas, il décidera forcément la valeur bleue. Dans cet état, même si aucun thread ne le sait, la valeur a déjà été décidée. On dit que l'état 10 est •-monovalent.

Définition 1 (État monovalent)**.** *Un état s est v-monovalent si seule la valeur v peut être décidée par une exécution passant par s.*

Un état s est monovalent s'il est v-monovalent pour un certain v.

Inverement, dans l'état 2, si T_0 effectue le prochain pas, la valeur rouge peut être décidée ; inversement, si T_1 effectue les deux prochains pas, la valeur bleue peut être décidée. L'état n'est donc pas monovalent. On dit qu'il est bivalent.

Définition 2 (État bivalent)**.** *Un état s est bivalent s'il n'est pas monovalent.*

Remarquez que, si les threads proposent une valeur différente, l'état initial est toujours bivalent.

Enfin, l'état 5 est particulier puisqu'il est bivalent, mais que le prochain état sera monovalent, quel que soit le thread qui fait le prochain pas. On dit qu'il est critique.

Définition 3 (État critique)**.** *Un état s est critique s'il est bivalent et que tous ses fils sont monovalents.*

Remarquez un point important : dans cet algorithme, c'est l'opération `getAndSet` qui permet de décider dans l'état critique. L'idée de la preuve d'impossibilité est de montrer que ni l'opération read, ni l'opération write, n'a la puissance de forcer l'état à devenir monovalent.

Dans la suite, on va démontrer qu'il n'existe pas de solution au consensus qui n'utilise que des lectures et des écritures. Il s'agit d'une démonstration par l'absurde : on suppose qu'il existe un algorithme A qui résout le consensus, et on aboutit à une contradiction.

Lemme 1. *Tout algorithme qui résout le consensus contient au moins un état critique.*

Démonstration. La démonstration de ce lemme est elle-même par l'absurde. Soit un algorithme A qui résout le consensus. On suppose que A ne contient pas d'état critique. Supposons que T_0 propose • et T_1 propose •. On va construire par récurrence une exécution infinie, ce qui entre en contradiction avec la propriété de terminaison du consensus.

Si T_0 s'exécute complètement avant que T_1 fasse son premier pas, T_0 ne peut pas savoir que T_1 n'a pas proposé la même valeur que lui. Dans le doute, et pour respecter la validité du consensus, T_0 est obligé de retourner •. Par un raisonnement similaire, si T_1 s'exécute seul, il devra retourner •. Cela signifie que l'état initial est bivalent.

```
   class Consensus<T> {
     // One atomic read/write register per thread
     final AtomicReferenceArray<T> proposed = new AtomicReferenceArray<T>(2);
     // One getAndSet object used to decide who wins the consensus
     final AtomicBoolean lost = new AtomicBoolean(false);

     T propose(T v) {
       // First, announce your value
(1)    proposed.set(id, v);
       // The first who calls getAndSet will get false, the other true
(2)    if (lost.getAndSet(true)) {
         // If you get true, you lose, return the other's value
(3)      return proposed.get(1 - id);
       } else {
         // If you get false, you win, return your own value
(4)      return proposed.get(id);
       }
     }
   }
```

FIGURE 1: Implémentation du consensus avec `getAndSet`

Supposons que l'on a réussi à construire une exécution de longueur t qui ne passe que par des états bivalents. Le dernier état est bivalent, et par hypothèse, il n'est pas critique. L'un de ses fils est donc bivalent et on peut construire une exécution de longueur $t+$ qui ne passe que par des états bivalents.

En continuant de la sorte, on construit une exécution infinie. Cela entre en contradiction avec le fait que A est wait-free. On en déduit que notre hypothèse est fausse : A contient des états critiques. □

Lemme 2. *Un état dans lequel la prochaine opération de chaque thread est une lecture ou une écriture ne peut pas être critique.*

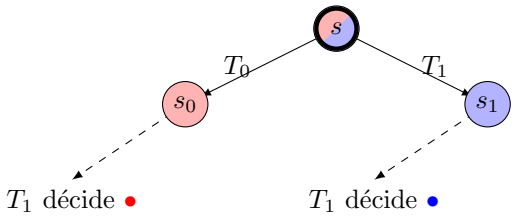
Démonstration. Soit algorithme A et un état s accessible par A. Cette démonstration liste toutes les possibilités pour les prochaines opérations des deux threads, et démontre qu'aucune n'est compatible avec un état critique.

Supposons que s est un état critique.

1. Supposons que T_0 écrive dans un registre r et T_1 lise le même registre. Si T_0 écrit à l'état s, on arrive dans l'état s_0, si T_1 lit à l'état s, on arrive dans l'état s_1. Comme s est bivalent, s_0 et s_1 sont v- et w-monovalents respectivement, avec $v \neq w$.

   Plaçons-nous du point de vue de T_0. S'il s'exécute seul dans s jusqu'à sa décision, il passera par l'état s_0 v-monovalent, donc T_0 décidera v. Si T_1 fait sa lecture puis T_0 s'exécute seul dans s_1 jusqu'à sa décision, il passera par l'état s_1 w-monovalent, donc T_0 décidera w. Or, aucun élément ne permet à T_0 de distinguer entre les deux exécutions, puisque la seule différence est l'état interne de T_1. Il est donc impossible qu'il retourne une valeur différente dans les deux cas, et ce premier cas est impossible.

   Le cas où T_0 lit et T_1 écrit est symétrique.



2. Supposons maintenant que T_0 et T_1 écrivent tous les deux dans le même registre r. Ce cas est exactement similaire au précédent : si T_0 s'exécute seul, il devra décider une valeur v. Si T_1 écrit avant de laisser

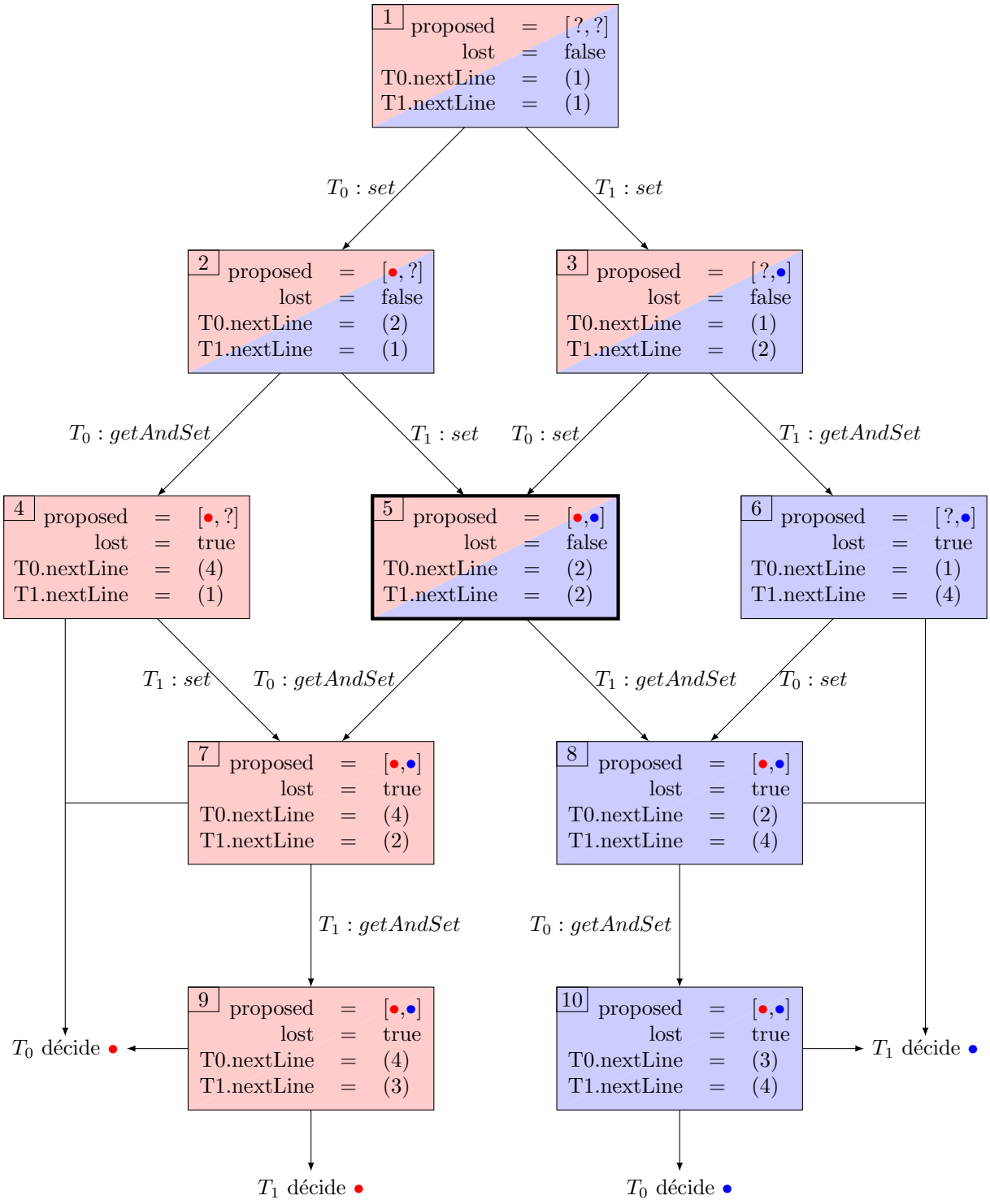
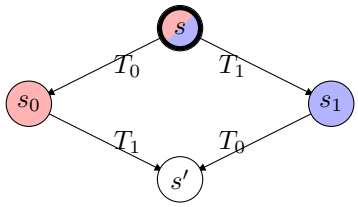


FIGURE 2: Graphe des exécutions de l'algorithme 1

T_0 s'exécuter seul, la première écriture de T_0 écrasera celle de T_1, donc les deux exécutions seront indistingables pour T_0. Pourtant, comme s est un état critique, T_0 doit décider w dans la deuxième exécution. Cela est absurde.

3. Supposons que les prochaines opérations de T_0 et T_1 commutent, c'est-à-dire que l'exécution de l'opération de T_0 avant celle de T_1 mène dans le même état que l'exécution de l'opération de T_1 avant celle de T_0. Cette hypothèse regroupe tous les cas restants :
   — T_0 et T_1 accèdent à des registres différents,
   — T_0 et T_1 lisent le même registre,
   — T_0 ou T_1 effectue une action locale qui n'accède pas à un registre partagé.
   Si T_0 exécute son opération en s, on arrive dans l'état v-monovalent s_0. Si T_1 exécute son opération en s, on arrive dans l'état w-monovalent s_1. Par définition de la commutativité, si T_0 exécute son opération en s_1 ou T_1 exécute son opération en s_0, on arrive dans le même état s'. Quelle est la valence de s' ? Comme s' est un fils de s_0, s' est v-monovalent ; comme s' est un fils de s_0, s' est w-monovalent. Cela est contradictoire, donc ce cas est impossible.



On a étudié tous les cas, et aucun n'est possible. On en déduit que s n'est pas un état critique. □

Théorème 1. *Il n'existe pas d'implémentation du consensus qui n'utilise que des lectures et écritures.*

Démonstration. Un tel algorithme devrait avoir un état critique d'après le lemme 1, mais cela ne se peut pas d'après le lemme 2. □

Références

[1] Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson *Impossibility of Distributed Consensus with One Faulty Process.* JACM (1985)

[2] Maurice P. Herlihy. *Wait-Free Synchronization.* PODC (1988)