Construction universelle d'objets partagés sans connaissance des participants

Pierre Sutra, Étienne Rivière, Pascal Felber

Construction universelle d'objets partagés sans connaissance des participants †

Pierre Sutra, Étienne Rivière et Pascal Felber

Université de Neuchâtel, Suisse

Une construction universelle est un algorithme permettant à un ensemble de processus concurrents d'accéder à un objet partagé en ayant l'illusion que celui-ci est disponible localement. Dans cet article, nous présentons un algorithme permettant la mise en œuvre d'une telle construction dans un système à mémoire partagée. Notre construction est sans verrou, et contrairement aux approches proposées précédemment, ne nécessite pas que les processus accédant à l'objet partagé soient connus. De plus, elle est adaptative : en notant n le nombre total de processus dans le système et $k \leq n$ le nombre de processus qui utilisent l'objet partagé, tout processus effectue $\Theta(k)$ pas de calcul en l'absence de contention.

Keywords: Mémoire partagée, Concurrence, Objet Partagé, Construction Universelle, Consensus.

1 Introduction

Les systèmes concurrents sont un élément constitutif important de l'informatique d'aujourd'hui. Ceci est dû à l'arrivée des processeurs multi-cœurs auprès du grand public, et aux nécessités en calcul intensif parallèle pour traiter les grandes masses de données. L'écriture d'une application concurrente est néanmoins difficile. Afin de faciliter cette tâche, il est nécessaire de fournir au programmeur une abstraction simple et bien définie pour synchroniser les processus. Une construction universelle est un algorithme permettant de transformer tout code séquentiel en un code concurrent garantissant l'atomicité des opérations, c'est-à-dire offrant l'illusion d'être accédé localement. Cette abstraction est par exemple mise en œuvre par les mémoires transactionnelles logicielles (6), ou le principe de moniteur (7)

Cet article propose une nouvelle construction universelle pour les systèmes à mémoire partagée. Cette construction est sans verrou, utilise uniquement des registres, et ne nécessite pas que l'ensemble des processus accédant à l'objet soit connu. De plus, elle est adaptative. Si n est le nombre total de processus dans le système, alors en l'absence de contention tout processus effectue $\Theta(k)$ pas de calcul, où $k \leq n$ est le nombre de processus qui utilisent réellement l'objet partagé.

Notre construction s'inscrit dans la ligne des travaux antérieurs de Herlihy (5) sur la construction universelle, ainsi que sur la décomposition de consensus proposée par Gafni (3) et Aspnes (1). Elle suit une approche hiérarchique et modulaire. Dans la suite de cet article, nous présentons successivement l'ensemble des modules utilisés.

Plan. En Section 2, nous introduisons le modèle de calcul utilisé. Dans la Section 3, nous présentons les briques de base nécessaires à notre construction. Celle-ci est décrite en détail dans la Section 4. La Section 5 conclut l'article après une brève discussion sur les résultats obtenus.

2 Modèle

Nous considérons un système réparti asynchrone où les processus communiquent par l'intermédiaire d'une mémoire partagée. Les processus prennent leur identifiants dans un ensemble Π de cardinalité n. L'ensemble Π n'est pas en soi accessible aux processus, toutefois ces derniers peuvent effectuer des calculs sur les identités, comme par exemple un test d'égalité.

†Ce travail est soutenu par la Commission Européenne sous l'accord FP7 numéro 318809 (LEADS).

Afin de modéliser les interruptions et les contraintes d'ordonnancement, il est supposé qu'au cours d'une exécution un processus peut subir une panne franche et s'arrêter prématurément. Du fait de telles fautes, nous nous intéressons à la mise en œuvre d'objets non bloquants. Plus précisément, la construction que nous présentons est sans obstruction : tout processus qui s'exécute seul finit à terme par retourner de son appel, et ceci indépendamment de l'état des autres processus.

Notre mesure de la complexité suppose que le cas le plus courant est celui où un processus exécute sans entrelacement une opération sur l'objet partagé. En conséquence, la complexité en temps d'un algorithme correspond au nombre maximal d'accès à la mémoire partagée lorsqu'un processus exécute seul une opération.

Notre construction s'appuie sur plusieurs contributions : deux nouveaux objets de synchronisation que sont le *grafarius* et la *course*, et un algorithme de consensus adaptatif en mémoire partagée. Dans ce qui suit, nous présentons successivement chacune de ces briques de base. Notons que plusieurs détails techniques ainsi que les preuves ont été volontairement omis de cet article. Le lecteur intéressé pourra les consulter dans la version étendue (13).

3 Briques de base

Splitter. Le premier objet utilisé par notre construction est le *splitter* (11). Le splitter permet de détecter lorsque deux processus effectuent un accès concurrent à un objet partagé. Cet objet se compose d'une seule opération $split()$ ne prenant pas de paramètre, et retournant une valeur dans $\{vrai, faux\}$. Lorsqu'un processus retourne *vrai*, nous dirons qu'il *gagne* le splitter ; dans le cas contraire, qu'il le *perd*. Au cours d'une exécution, si plusieurs processus exécutent $split()$, au plus un reçoit la valeur *vrai*. Par ailleurs, si un seul processus appelle $split()$, son appel doit nécessairement retourner *vrai*. Il est à noter qu'un splitter peut trivialement être mis en œuvre dans une mémoire partagée à l'aide de deux registres (9, 11).

Grafarius. Le second objet que nous utilisons est nommé *grafarius*. Le grafarius est une abstraction proche de l'objet adopt-commit proposé par Gafni (3) qui modélise une ronde du protocole Paxos (10). Étant donné un ensemble de valeurs $\mathcal{V}$, le grafarius offre une opération $adoptCommit(v \in \mathcal{V})$ qui retourne un couple (u, f), où u appartient à $\mathcal{V}$, et f est un drapeau à valeur dans $\{adopt, commit\}$. Un processus p *adopte* la valeur u lorsqu'il retourne $(u, adopt)$ lors d'un appel au grafarius. De façon similaire, p *valide* la valeur u lorsque la valeur retournée est $(u, commit)$. Les propriétés d'un grafarius sont les suivantes : (*Validité*) Si p adopte v, alors un processus a exécuté $adoptCommit(v)$ précédemment. (*Cohérence*) Si p valide v, tout processus adopte ou valide v. (*Convergence Solo*) Si p retourne de $adoptCommit(u)$ avant qu'un autre processus exécute *adoptCommit*, alors p valide u. (*Continuation*) Si un processus retourne avant que p invoque *adoptCommit*, p adopte ou valide une valeur proposée avant son appel à *adoptCommit*.

Nous pouvons construire un grafarius à l'aide d'un splitter s, d'un drapeau c et d'un registre d de la manière suivante : Lorsqu'un processus p exécute $adoptCommit(u)$, il accède au splitter s. Si p gagne le splitter, le processus p écrit la valeur u dans d puis vérifie la valeur du drapeau c. Dans le cas où c est à sa valeur initiale (*faux*), p valide la valeur u. Sinon, p adopte la valeur u. Maintenant, dans le cas où p perd le splitter s, p indique qu'une collision à eu lieu en écrivant la valeur *vrai* dans c. Puis p consulte la valeur du registre d. Sous l'hypothèse où d n'est pas à sa valeur initiale ($\perp$), p adopte cette dernière. Dans le cas contraire, p écrit la valeur u dans d avant de l'adopter.

À l'aide du grafarius, nous allons construire *consensus*. À ce propos, nous rappelons que consensus est un objet partagé offrant une opération $propose(u \in \mathcal{V})$ à valeur dans $\mathcal{V}$ qui garantit que (*Validité*) si v est retournée, alors un processus a invoqué $propose(v)$ précédemment, et (*Accord*) deux processus doivent toujours retourner la même valeur.

Dans l'esprit des travaux de Aspnes (1), nous observons que des appels cohérents à une succession de grafarius résolve consensus. Schématiquement, l'algorithme fonctionne comme suit. Chaque processus p parcourt de manière croissante un tableau (infini) de grafarius. Pour chaque case du tableau, p appelle le grafarius correspondant. Soit maintenant v la valeur retournée par cet appel. Si p valide la valeur v, alors v est sa décision pour consensus. Sinon v est la nouvelle valeur que propose p, et p continue sa progression dans le tableau.

Au cours de l'algorithme précédent, un processus peut avoir beaucoup de retard par rapport aux autres

Algorithm 1 Construction Universelle – algorithme pour le processus p

```
1:  Variable partagée :
2:      R                                                  // Une course sur des consensus
3:
4:  Variables locales :
5:      C                                                  // Initialement, R.entrer()
6:      s                                                  // Initialement, s_0
7:
8:  invoquer(op) :=
9:      tant que vrai faire
10:         d ← C.d
11:         si d ≠ ⊥ alors
12:             s ← d[1]
13:             C ← R.entrer()
14:         sinon
15:             (s', v) ← τ(s, op)
16:             si s = s' alors
17:                 renvoyer v
18:             d ← C.propose((p, s'))
19:             si d[0] = p alors
20:                 renvoyer v
```

processus dans le parcours du tableau. Par conséquent, la complexité de cet algorithme n'est pas bornée. Afin de pallier à ce problème, nous allons stocker l'index de la plus haute case du tableau atteinte par un des processus. Cette idée simple est abstraite par un nouvel objet que nous nommons course.

Course. Plusieurs algorithmes (par exemple, (4)) accèdent de manière itérée à des objets partagés pour progresser. Une *course* est un objet qui capture cette notion d'appels itérés. Son interface consiste en une seule opération $entrer(p,l)$, définie sur un ensemble dénombrable de *tours*. Au cours d'une exécution, un processus p *entre* dans le tour l lorsque l'opération $entrer(p,l)$ a lieu. Le processus p *quitte* le tour l lorsque l est le dernier tour dans lequel p est entré, et qu'il entre dans un nouveau tour. L'invariant suivant est maintenu au cours de toute course : *(Bon Ordre)* Il existe un ordre total strict $\ll$ sur l'ensemble des tours dans lesquels les processus sont entrés au cours de l'exécution tel que pour tout processus p qui entre dans un tour l, soit (i) un processus a quitté l avant p, ou (ii) le dernier tour quitté par p est le plus petit tour plus grand que l pour l'ordre $\ll$.

Une course est aisément mise en œuvre en numérotant les tours et en utilisant un tableau adaptatif (2), qui contient pour chacun des processus le numéro du tour dans lequel il se situe. Lorsqu'un processus p souhaite entrer dans un nouveau tour, il lit le tableau et choisit, soit le plus grand tour inscrit si p ne vient pas de le quitter, soit le tour suivant par ordre des numéros.

Consensus. Dans l'esprit de l'algorithme de Aspnes (1) décrit plus haut, la notion de course permet de construire de manière simple consensus. Pour ce faire, nous considérons une course sur un ensemble infini de grafarius, et un registre d. À chaque itération, un processus p entre dans un nouveau grafarius g déterminé par R. Si d a une valeur non nulle, alors p décide de la valeur stockée. Sinon, p propose sa valeur à g. Dans le cas où cette valeur est validée par g, p décide de celle-ci dans consensus en l'écrivant dans d. Sinon, p continue sa course sur les grafarius.

Nous disposons à présent de l'ensemble des briques de base permettant de partager de manière universelle un objet entre plusieurs processus concurrents, ce que nous détaillons ci-après.

4 Construction Universelle

Une construction universelle est un algorithme qui permet d'obtenir un objet partagé atomique à partir de son code séquentiel (5). Le code séquentiel d'un objet est formalisé par un machine ayant un ensemble d'états ($Etats$), un état initial ($s_0 \in Etats$), un ensemble d'opérations (Op), un ensemble de valeurs de retour ($Valeurs$), et une relation de transition ($\tau : Etats \times Op \to Etats \times Valeurs$). Il est à noter que sans perte de généralité, nous supposons que chaque opération op est *totale*, c'est à dire que $Etats \times \{op\}$ fait partie du domaine de τ.

Notre construction universelle est présentée par l'Algorithme 1. Elle fonctionne comme suit. Chaque processus maintient une copie locale s de l'état de l'objet partagé, et accède à une course R sur des consensus. Lorsqu'un processus p invoque une opération op sur l'objet partagé, p scrute le dernier consensus dans lequel il est entré (ligne 11). Si ce consensus est décidé, p met à jour sa copie s avec cette décision, puis entre dans le prochain consensus (ligne 13). Une fois que p atteint un consensus non décidé, la variable s contient un état de l'objet postérieur au moment où p a invoqué op. Le processus p exécute alors de manière optimiste l'opération op sur s, puis stocke localement le nouvel état ainsi que le résultat de cette opération dans la paire (s', v). Si s égale s', l'invocation ne change pas l'objet, et p peut alors retourner immédiatement la valeur v (ligne 17). Dans le cas contraire, p propose la paire (p, s') à consensus. Si cette paire est la valeur décidée, p retourne la réponse v (ligne 20). Sinon, le processus p continue sa course sur les consensus.

Complexité. Comme indiqué dans l'introduction, la situation que nous considérons être la plus fréquente est celle sans entrelacement des opérations sur l'objet partagé. En considérant cette mesure de complexité, dans le pire cas un processus qui exécute *invoquer*() retrouve d'abord le plus grand consensus décidé, entre dans ce consensus, puis dans le consensus suivant afin de décider de la valeur de retour de son opération. Si nous mettons en œuvre une course à l'aide du collecteur adaptatif de (2), chaque accès à *entrer*() requiert $O(k)$ opérations, où k est le cardinal de l'ensemble des processus ayant accédé à la course. En conséquence, la complexité en temps de notre construction universelle est $O(k)$. Le résultat de Jayanti et al. (8) nous dit que cette valeur est aussi une borne inférieure sur la complexité en temps de toute construction universelle en mémoire partagée, et est par conséquent optimale.

5 Conclusion

Cet article présente une méthode universelle de partage d'objets entre plusieurs processus. Contrairement aux travaux précédents (tels que par exemple (5)), notre méthode ne nécessite pas de connaître au préalable les participants. Par ailleurs, en notant k le cardinal de l'ensemble des processus accédant à l'objet partagé, elle requiert le temps optimal de $\Theta(k)$ pas de calcul sur les registres partagés. Pour ce faire, elle s'appuie sur deux nouveaux objets pour le contrôle de concurrence, le *grafarius* et la *course*, objets que nous décrivons en détail dans cet article.

Pour être utilisable en pratique, une construction universelle doit aussi être bornée en mémoire. Nous avons proposé un mécanisme de recyclage des objets qui atteint ce but. Faute de place, ce nouveau mécanisme n'est pas détaillé dans cet article. Le lecteur intéressé pourra néanmoins le consulter dans la version étendue (13). Cette version étendue contient aussi des résultats d'évaluation dans une mémoire partagée répartie où nous comparons notre approche à l'algorithme Paxos. Le code source de cette mise en œuvre est disponible publiquement (12).

Références

[1] J. Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *29th ACM Symposium on Principles of Distributed Computing*, PODC, 2010.

[2] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 2002.

[3] E. Gafni. Round-by-round fault detectors (extended abstract) : unifying synchrony and asynchrony. In *17th Annual ACM Symposium on Principles of Distributed Computing*, PODC, 1998.

[4] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 2007.

[5] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 1991.

[6] M. Herlihy and J. E. B. Moss. Transactional memory : Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture*, ISCA, 1993.

[7] C. A. R. Hoare. Monitors : An operating system structuring concept. *Commun. ACM*, 1974.

[8] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 2000.

[9] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 1987.

[10] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[11] M. Moir and J. Anderson. Fast, long-lived renaming. In *Distributed Algorithms*. 1994.

[12] P. Sutra. http ://github.com/otrack/pssolib, 2013.

[13] P. Sutra, E. Rivière, and P. Felber. A practical distributed universal construction with unknown participants. In *18th International Conference on Principles of Distributed Systems*, OPODIS, 2014.