HAL
archives-ouvertes.fr

Table de hachage distribuée autostabilisante

Olivier Peres, Thomas Herault

▶ **To cite this version:**

Olivier Peres, Thomas Herault. Table de hachage distribuée autostabilisante. 9ème Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications, May 2007, Ile d'Oléron, France. pp.63-66. inria-00176951

HAL Id: inria-00176951
https://hal.inria.fr/inria-00176951

Submitted on 5 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Table de hachage distribuée autostabilisante

Olivier Peres and Thomas Herault

Univ Paris Sud; LRI UMR8623; INRIA; Orsay F-91405

Nous présentons un algorithme autostabilisant qui construit et entretient une table de hachage distribuée dans un environnement pair à pair. L'utilisation d'un modèle dans lequel les processus ne connaissent pas leurs voisins a priori le rend utilisable sur des systèmes à grande échelle. Ses performances en nombre de messages sont de l'ordre des références du domaine (notamment Chord), la réplication des données est également assurée. La base sur lequel il est construit permet d'en donner une preuve formelle de correction.

Keywords: Tables de hachage distribuées, autostabilisation, systèmes pair à pair, systèmes à grande échelle

1 Introduction

La répartition de donnéees sur toutes les machines d'un système pair à pair est un problème classique pour lequel les solutions proposées impliquent la création d'une structure (CAN [RFH+01], Chord [SMK+01], Pastry [RD01], Tapestry [ZHR+04]) ou non (Freenet [CSWH01], Gnutella [GNU]). La première approche permet de garantir le succès des opérations de recherche mais impose d'entretenir certains liens entre les processus, organisés d'une façon spécifique à chaque algorithme. La deuxième élimine ce coût, mais augmente celui des recherches, qui fonctionnent alors par inondation et peuvent même échouer alors que l'objet cherché est bien dans le système.

Nous nous intéressons ici à la construction d'une telle structure. Notre approche est basée sur l'autostabilisation [Dij74], une propriété des programmes qui permet de retourner à un comportement correct après toute défaillance transitoire. Nous l'utilisons pour tolérer les insertions de processus dans le système, leurs départs inopinés, et leurs arrêts définitifs (crash). Afin de limiter autant que possible la quantité de mémoire utilisée par chaque processus, nous utilisons un modèle sans liste de voisins [HLP+07] et un algorithme précédemment écrit dans ce modèle [PH07].

2 Travaux connexes

CAN [RFH+01] figure parmi les premiers essais de construction d'une table de hachage distribuée. Elle se base sur une structure torique multidimensionnelle. Le principe de Pastry [RD01] est que la valeur v est enregistrée par le processus dont l'identifiant, dans un espace d'adresses à 128 bits, est le plus proche de v. Sa complexité en espace est $O(\log n)$ et elle fournit un routage en $O(\log n)$ messages. Tapestry [ZHR+04] fournit des performances similaires en construisant une structure d'hypercube basée sur les arbres de Plaxton. Ses avantages sont une meilleure décentralisation de l'algorithme et une meilleure tolérance aux défaillances. Chord [SMK+01] montre une autre piste basée sur un anneau à sauts où chaque processus connaît son prédécesseur et son successeur à distance 2^n pour tout $n < m$, où 2^m est une borne supérieure au nombre de processus du système. Il utilise la fonction de hachage SHA-1 pour calculer les identifiants de processus.

Ces travaux classiques ont pour points communs de reposer sur des principes probabilistes et de ne tolérer qu'un type de défaillance, l'arrêt définitif. Cet article présente un algorithme déterministe qui construit une table de hachage distribuée et tolère aussi les défaillances transitoires portant sur les variables des processus et le contenu des canaux de communication.Pour cela, l'algorithme décrit est autostabilisant. Ce principe décrit par Dijkstra [Dij74] signifie que l'on définit un ensemble $\mathcal{L}$ de configurations légitimes fermé par l'exécution de l'algorithme tel que toute configuration de $\mathcal{L}$ satisfait les propriétés de correction

de l'algorithme (ici, décrit une table de hachage distribuée) et que l'algorithme, initialisé dans une configuration arbitraire, converge vers une configuration de $\mathcal{L}$. Notre intérêt pour cette propriété est lié au fait que l'exécution d'un système pair à pair, avec de nombreux départs et arrivées de processus, peut le conduire dans un état complexe et difficile à décrire. L'autostabilisation permet d'abstraire toute suite de telles erreurs en une seule, l'initialisation arbitraire des variables et canaux du système.

3 Modèle

Nous utilisons un modèle [HLP$^+$07] asynchrone dans lequel tout processus porte un identifiant unique dans un ensemble totalement ordonné et peut envoyer, via un canal FIFO sans perte, un message à tout autre processus dont il connaît l'identifiant. Un processus ne connaît pas a priori d'autre processus, contrairement au modèle classique en autostabilisation, dans lequel chaque processus a une liste de voisins. Ceci permet d'abaisser la complexité en espace liée à la connaissance de ses voisins, ce qui est essentiel dans un système pair à pair où la taille d'une telle liste peut être de l'ordre du million de processus : en plus de l'occupation de mémoire, il se pose alors le problème de l'entretien de la liste. Pour enlever tout processus qui quitte le système des listes de voisins de tous les autres, ou y ajouter ceux qui arrivent dans le système, la quantité de données échangées serait rédhibitoire. Pour la remplacer, chaque processus dispose de deux dispositifs locaux : un oracle et un détecteur de défaillances.

Lorsqu'un processus interroge son oracle, celui-ci renvoie un identifiant de processus correctement formé, mais qui n'est pas nécessairement celui d'un processus en cours d'exécution (ce peut être, par exemple, celui d'un processus arrêté à cause d'une défaillance). Pour qu'il soit possible de construire une structure réunissant tous les processus, il faut imposer une condition globale sur les oracles : si dans un suffixe infini S d'une exécution, P est l'ensemble des processus qui interrogent leur oracle, alors tous les processus de P doivent obtenir le plus grand identifiant de P dans S.

Le détecteur de défaillances correspond à la définition de Chandra et Toueg [CT96], à ceci près que pour éviter de construire une liste contenant potentiellement tous les processus du système, nous le dotons d'un prédicat *suspect* qui prend en paramètre un identifiant de processus et indique, sans garantie de fiabilité, s'il correspond à un processus arrêté (ou à aucun processus). Il est dans la classe $\diamond\mathcal{P}$ des détecteurs inéluctablement parfaits, ce qui signifie qu'il existe un suffixe infini de toute exécution du système dans lequel son prédicat *suspect* renvoie toujours une valeur correcte.

4 Algorithme

Dans le modèle ci-dessus, nous avons proposé un algorithme autostabilisant qui construit une structure similaire à un arbre binaire équilibré avec une complexité spatiale moyenne en $O(1)$ [PH07]. Dans un système de n processus, où $n = \sum_{i \in I} 2^i$, cet algorithme construit un paquet de 2^i processus par i de I : il y a donc un paquet, dont le cardinal est une puissance de 2, par bit à 1 dans l'écriture de n en base 2. La figure 1 montre la structure d'un paquet. Dans une représentation simplifiée pour la clarté du propos, chaque processus possède un vecteur de voisins qui peut contenir à chaque niveau compris entre 0 et t, où $t > \log n$ (tous les logarithmes sont en base 2), soit un identifiant de processus, soit $\perp$. Pour chaque paire de processus (p,q) où p est le voisin de q au niveau l et vice versa, un leader est élu (en gras sur la figure) et lui seul peut avoir un voisin au niveau $l+1$. Chaque paquet a donc une structure interne arborescente binaire.

Nous utilisons ici une propriété des algorithmes autostabilisants : la possibilité de les composer. Étant donnés deux algorithmes autostabilisants A et B tels que A n'écrit pas dans la mémoire de B, leur composition $A \oplus B$, obtenue par la concaténation de leurs variables et de leurs codes, est autostabilisante [Dol00]. C'est ainsi qu'un algorithme de chaînage est composé avec l'algorithme de construction de paquets afin de les lier entre eux. Le leader de chaque paquet, qui connaît son niveau, a comme prédécesseur le leader du paquet de niveau immédiatement supérieur et comme successeur le leader du paquet de niveau immédiatement inférieur. Ceci conduit à l'élection d'un leader global, ou racine.

Une fois cette structure arborescente construite, il est possible de composer un nouvel algorithme avec les deux précédents pour fournir un service. Nous avons ainsi présenté [PH07] un algorithme sans mémoire qui numérote consécutivement les processus en fonctionnant uniquement par routage. Nous montrons ici

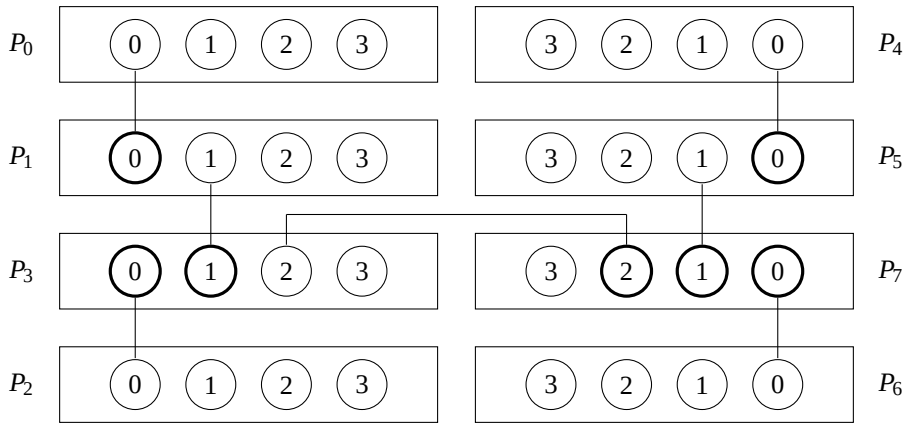

FIG. 1: Structure d'un paquet.

comment tirer parti de cette structure pour construire une table de hachage distribuée. L'algorithme que nous présentons fonctionne par-dessus la structure existante. Il fait correspondre à chaque valeur à insérer dans la table le numéro d'un processus chargé de l'enregistrer et définit le comportement à adopter en cas d'arrivée ou de départ d'un processus. De plus, chaque processus enregistre également les données d'autres processus pour garantir le niveau de réplication demandé par l'utilisateur.

4.1 Nombre de processus du système

Rappelons que dans un système de $n = \sum_{i \in I} 2^i$ processus, l'algorithme de formation de paquets en fait un, de taille 2^i, pour chaque $i \in I$. Nous commençons par composer un nouvel algorithme autostabilisant avec celui qui assure le chaînage. Il donne au leader de chaque paquet une variable n de taille t bits, avec $2^t > n$, pour estimer la taille du système. Sa mise à jour fonctionne comme suit : chaque leader de paquet envoie périodiquement à son successeur et son prédécesseur un vecteur de valeurs prises dans $\{0, 1, x\}$ où 0 indique que le processus sait que ce bit de n est à zéro (seuls deux processus en envoient : le leader global, qui sait que tous les bits supérieurs au sien sont à 0 et le dernier maillon de la chaîne, qui sait que tous les bits inférieurs au sien sont à 0), 1 indique que le processus sait que ce bit vaut 1 (c'est le sien) et x est utilisé pour tous les autres bits. Lors de la réception d'un tel message, chaque processus met à 0 dans sa variable n les bits valant 0 dans le vecteur, à 1 les bits valant 1 dans le vecteur et ne change pas les bits à x. Il transmet aussi le message à ses propres successeur et prédécesseur. D'autre part, chaque processus met périodiquement à 1 le bit de n correspondant à son niveau. Ainsi, toutes les valeurs de n valent inéluctablement le nombre de processus du système.

4.2 Répartition des clés

Pour chaque valeur qui doit être enregistrée dans la table, une clé entière est calculée par une fonction de hachage. Les processus sont numérotés séquentiellement de 0 à $m = 2^{\lceil \log n \rceil} - 1$ et chacun est responsable de toutes les valeurs dont la clé modulo m vaut son numéro.

Dans le cas où un paquet de processus manque, ce qui se produit dès lors que m n'est pas tel que $\exists p$ tq $m = 2^p - 1$, les clés sont réparties dans le paquet de niveau immédiatement supérieur dans la chaîne. Pour cela, ils sont à leur tour numérotés séquentiellement. Comme chaque paquet compte plus de processus que la somme de tous les paquets inférieurs, chaque processus est responsable d'au plus deux séries de clés.

4.3 Réplication des données

Afin de permettre au système de tolérer un départ inopiné de processus, chacun enregistre non seulement les valeurs dont il est responsable, mais aussi celles dont il serait responsable si les paquets inférieurs et supérieurs n'étaient pas présents. La tolérance de d départs inopinés simultanés s'obtient en étendant ce principe à d niveaux successifs.

La réplication autostabilisante ne pouvant être silencieuse, il est nécessaire que chaque processus envoie périodiquement ses valeurs aux autres processus qui dupliquent ses données. Ceci signifie que la tolérance aux arrivées et départs de processus, dans tous les cas considérés comme des défaillances, ne nécessite pas davantage de code : les processus voient simplement arriver les nouvelles valeurs qui leur sont destinées.

4.4 Performances des principales opérations

La nature même de la structure arborescente fournit immédiatement un routage des demandes d'insertion et de lecture de données en $O(\log n)$ messages. En effet, chaque processus transmet ses messages au leader de son paquet, qui les fait circuler dans la chaîne jusqu'au paquet de destination, où il est livré au destinataire.

5 Conclusion

Nous présentons une nouvelle façon d'exploiter l'autostabilisation à grande échelle pour enregistrer des données dans une table de hachage répartie. Ceci permet de donner des algorithmes totalement déterministes dont les bornes théoriques sur les performances ne sont donc pas probabilistes. Le modèle sans liste de voisins évite l'enregistrement de nombreuses données qui empêchent habituellement le passage à l'échelle de ce genre d'algorithmes.

Il reste cependant une inconnue importante : les performances réelles sur une plate-forme expérimentale, particulièrement dans des scénarios comportant des mouvement importants et rapprochés de processus. Leur évaluation constitue donc la principale étape suivante de ce travail.

Références

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet : A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009 :46– ? ?, 2001.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, March 1996.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11) :643–644, 1974.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[GNU] Gnutella : http ://www.gnutella.com/.

[HLP+07] Thomas Herault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. A model for large scale self-stabilization. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.

[PH07] Olivier Peres and Thomas Herault. Self-stabilising overlay network for efficient numbering in large scale systems. Technical Report 1466, LRI, http ://www.lri.fr/Rapports-internes/2006/RR1466.pdf, 2007.

[RD01] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[ZHR+04] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry : A resilient global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1) :41–53, January 2004.