

# Tables De Hachage Distribuées

## *Distributed Hash Tables (DHT)*

Denis Berthod

8 mars 2006

### Table des matières

<b>1</b>	<b>Panorama Des Tables de Hachage Distribuées</b>	<b>2</b>
1.1	Présentation des tables de hachage distribuées . . . . .	2
1.2	Topologies . . . . .	2
1.2.1	Les différentes topologies utilisées . . . . .	2
1.2.2	L'hypercube . . . . .	3
1.2.3	Le graphe de De Bruijn . . . . .	3
1.3	Types de liens . . . . .	4
1.3.1	Les liens stricts ( <i>strict links</i> ) . . . . .	5
1.3.2	Les liens lâches ( <i>loose links</i> ) . . . . .	5
1.3.3	Quel type de liaison utiliser? . . . . .	7
<b>2</b>	<b>Broose</b>	<b>7</b>
2.1	Présentation de Broose . . . . .	7
2.2	Recherche à droite ( <i>Right lookup</i> ) . . . . .	7
2.3	Recherche de voisinage ( <i>Brother lookup</i> ) . . . . .	9
2.4	Recherche à gauche ( <i>Left lookup</i> ) . . . . .	9
2.5	Rafraîchissement des buckets . . . . .	10
<b>3</b>	<b>Pour en savoir plus...</b>	<b>10</b>

### Table des figures

1	Hypercube de dimension 3 . . . . .	3
2	Hypercube de dimension 4 et quelques sous-hypercubes . . . . .	4
3	Graphe de De Bruijn pour $k = 3$ avec décalage droit . . . . .	4
4	Routage depuis le nœud 001 vers le nœud 110 . . . . .	5
5	(a) le nœud considéré (en rouge) et ses voisins (en vert) dans la topologie sous-jacente. (b) liens stricts. (c) liens lâches. . . . .	5
6	Réplication de l'association ( $c, d$ ) sur les nœuds proches de $c$ . . . . .	6
7	Mécanisme de la recherche dans un réseau lâche . . . . .	6
8	Exemples de recherche à droite sur le réseau Broose illustré en (a). . . . .	8
9	Exemple de recherche à gauche sur le graphe de la figure 8 (a). . . . .	10

# 1 Panorama Des Tables de Hachage Distribuées

## 1.1 Présentation des tables de hachage distribuées

Les tables de hachage distribuées sont avant tout des tables de hachage, c'est à dire qu'elle permettent la recherche d'une donnée grâce à une clé.

On se donne donc un espace des clés  $K$  et une *fonction de hachage*  $f : D \rightarrow K$  où  $D$  est l'espace dans lequel les données résident. La fonction  $f$  permet alors d'affecter de manière déterministe une clé à une donnée.

Dans une table de hachage distribuée, les associations sont réparties sur différents nœuds. Chaque nœud à un identifiant à valeur dans  $K$ . Par la suite, par souci de clarté, on identifiera le nœud à son identifiant.

La donnée  $d$  est stockée sur le nœud  $n$  tel que  $distance(f(d), n)$  soit minimale.

Afin d'éviter de maintenir une table avec tous les nœuds du réseau de la table de hachage, on utilise un routage de proche en proche à la manière du réseau internet.

Le réseau ainsi constitué étant virtuel, il est possibles d'utiliser des topologies de réseaux aux caractéristiques connues. Nous verrons quelques une de ces topologies dans la section 1.2.

Les tables de hachage distribuées sont apparues avec la nécessité d'indexer le contenu des réseaux peer-to-peer de manière décentralisée. Le but étant alors d'avoir un mécanisme d'indexation qui respecte les contraintes suivantes :

- La recherche d'une association doit se faire en un nombre de saut raisonnable, typiquement en  $O(\log n)$  si  $n$  est le cardinal de l'espace des clés.
- Disponibilité d'une association après le départ du nœud la contenant.
- Disponibilité d'une association après le crash du nœud la contenant.
- Répartition de la charge de manière équitable entre les différents nœuds du réseau
- Limitation de l'occupation de la bande passante par les messages de contrôles du réseau

Différentes stratégies s'appliquent pour répondre à ces contraintes, en particulier, on peut utiliser soit des liens stricts soit des liens lâches comme nous le verrons dans la section 1.3.

## 1.2 Topologies

### 1.2.1 Les différentes topologies utilisées

Les différentes tables de hachage qui existent créer un réseau virtuel entre les nœuds qui la composent. Ce réseau virtuel peut être construit sur différentes topologies de bases qui possèdent de *bonnes* propriétés comme un un diamètre réduit, par exemple.

Voici quelques topologies communément rencontrées :

- Arbre (DNS<sup>1</sup>)
- Hypercube et sous-hypercube (Chord, Kademia, Pastry)
- Hypertore (CAN)
- Papillon (Viceroy)
- Graphe de De Bruijn (D2b, Broose)

On remarquera que la plupart de ces topologies, possèdent un nombre de nœuds qui est fixé par un paramètre. Donc le fait que, par essence, le nombre de nœuds de la table de hachage distribuée soit variable impose des distorsions par rapport à la topologie sous-jacente. Cependant, les DHT qui sont basée sur une même topologie vont partager les caractéristiques héritées de celle-ci :

	Nombre de voisins	Diamètre
Hypercube	$O(\log n)$	$O(\log n)$
Hypertore	$O(d)$	$O(d^{1/n})$
Papillon	$O(1)$	$O(\log n)$
Graphe de De Bruijn	$O(1)$	$O(\log n)$

Dans le tableau précédent  $n$  est le cardinal de l'espace des clés et  $d$  est la dimension de l'hypertore.

---

<sup>1</sup>Le DNS n'est généralement pas considéré comme une DHT, il en a cependant les caractéristiques. La seule différence étant qu'il ne s'autogère pas et qu'il faut le gérer à la main.

Le nombre de voisins influe sur la taille des tables de routage et sur le nombre de messages envoyé lors de l'insertion et de la suppression d'un nœud du réseau. D'autre part, plus le nombre de voisin est élevé, plus le réseau est résistant aux pannes.

Le diamètre, quant à lui, influe sur le nombre de saut maximum que les messages doivent effectuer pour atteindre leur cible.

Nous allons maintenant aborder plus en détail deux de ces topologies : l'hypercube et le graphe de De Bruijn.

### 1.2.2 L'hypercube

La topologie en hypercube est souvent utilisée dans les réseaux car elle possède un bon rapport *diamètre/Nombre de voisins*.

Un hypercube de dimension  $d$  possède  $2^d$  nœuds et est construit comme suit :

- Chaque nœud est étiqueté par un mot appartenant à  $\{0, 1\}^d$ .
- Deux nœuds  $u$  et  $v$  sont reliés si et seulement si ils ont un et un seul bit de différence.

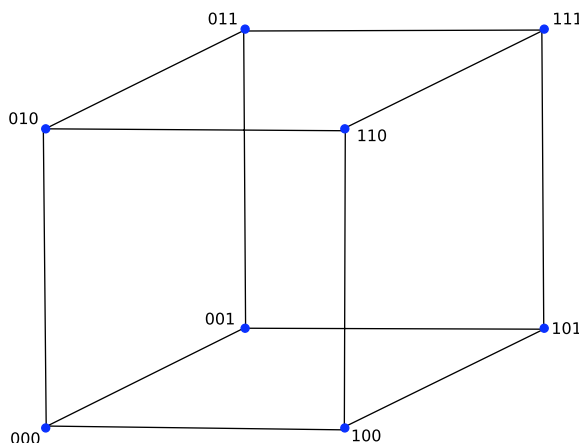


FIG. 1 – Hypercube de dimension 3

La figure 1 montre un hypercube de dimension 3 avec les nœuds répartis de manière à représenter une perspective cavalière du cube.

Il est possible de définir une notion de *sous-hypercube* : un sous-hypercube, est un hypercube de dimension inférieure qui est inclus dans l'hypercube.

Il est possible de déterminer les sous-hypercubes qui ne s'intersectent pas (pour une dimension  $d'$  donné) en considérant les hypercubes formés par les nœuds ayant en commun un préfixe de longueur  $n - d'$ . La figure 2 montre un hypercube de dimension 4 avec quelques-uns de ses sous-hypercubes.

Les sous-hypercubes sont, par exemple, utilisés par Kademia pour déterminer les ensembles parmi lesquels sont choisis les éléments des buckets.

### 1.2.3 Le graphe de De Bruijn

Afin de limiter le nombre de voisins de chaque nœud, de manière à faciliter la mise à jour des réseaux, les recherches se sont portées sur des topologies où les nœuds ont un nombre constant de voisins quelque soit la taille du réseau. Se fut d'abord Viceroy sur une topologie en Papillon qui y parvint, il est cependant jugé trop complexe pour être implémenté efficacement. Les dernières années ont vu l'émergence des DHT basées sur les graphes de De Bruijn qui parviennent au même résultat avec des protocoles plus efficaces.

**Topologie.** Le graphe de De Bruijn sur l'alphabet  $\{0, 1\}$  a les propriétés suivantes :

- C'est un graphe orienté.
- Il contient  $2^k$  sommets étiquetés par des mots appartenant à  $\{0, 1\}^k$ .
- Si on effectue un décalage droit : l'arc  $uv$  existe si  $v = 0u[1, k - 1]$  ou  $v = 1u[1, k - 1]$ .
- Si on effectue un décalage gauche : l'arc  $uv$  existe si  $v = u[2, k]0$  ou  $v = u[2, k]1$ .

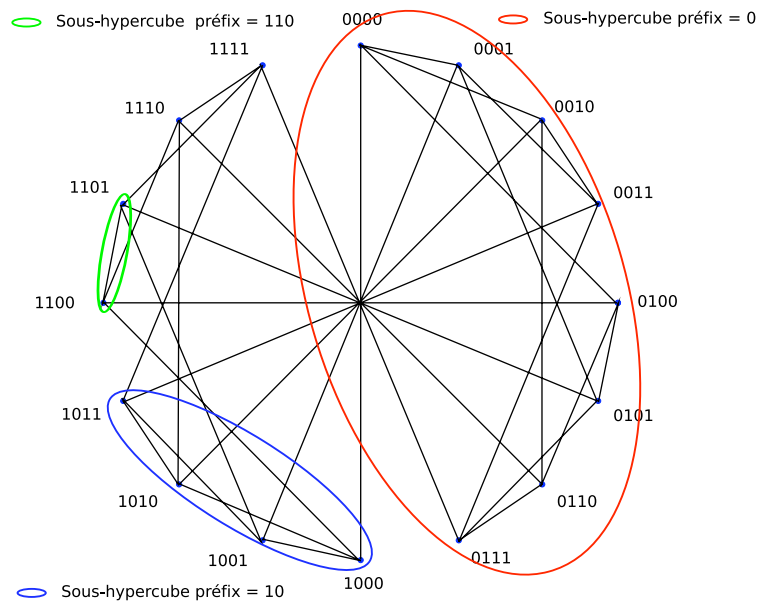


FIG. 2 – Hypercube de dimension 4 et quelques sous-hypercubes

La figure 3 montre un graphe de De Bruijn pour  $k = 3$  en effectuant un décalage droit.

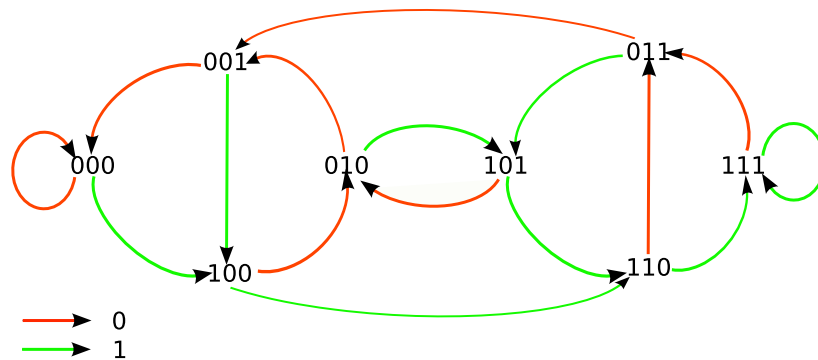


FIG. 3 – Graphe de De Bruijn pour  $k = 3$  avec décalage droit

On remarquera que pour passer du graphe obtenu par un décalage droit au graphe obtenu par un décalage gauche, il suffit d'inverser le sens des arcs.

**ROUTAGE.** Le routage sur un graphe de De Bruijn s'effectue en insérant au fur et à mesure les bits du nœud de destination dans l'identifiant du nœud de départ. La figure 4 nous montre un exemple de routage sur le graphe de la figure 3 : Le nœud de départ  $u$  a pour identifiant 001 et le nœud d'arriver  $v$ , l'identifiant 110. La requête va circuler par les nœud obtenu par décalage droit de  $u$  en insérant les bits droits de  $v$  dans les emplacements ainsi libéré :  $110001 \rightarrow 110001 \rightarrow 110001 \rightarrow 110001$ .

On remarquera que le routage n'est pas optimal car on aurait pu suivre le chemin :  $001 \rightarrow 100 \rightarrow 110$ . Cependant cette technique de routage a le mérite d'être simple et de s'effectuer toujours en  $\log n$  sauts.

### 1.3 Types de liens

Une fois une topologie déterminée pour un réseau, il faut choisir la manière de se relier à ses voisins, il y a alors deux manières de faire :

- liens stricts (*strict links*) : les liens du réseau suivent ceux de la topologie sous-jacente.
- liens lâches (*loose links*) : les liens du réseau pointent sur des ensembles de nœuds, nommés *buckets*, "proches" de la cible du liens de la topologie sous-jacente.

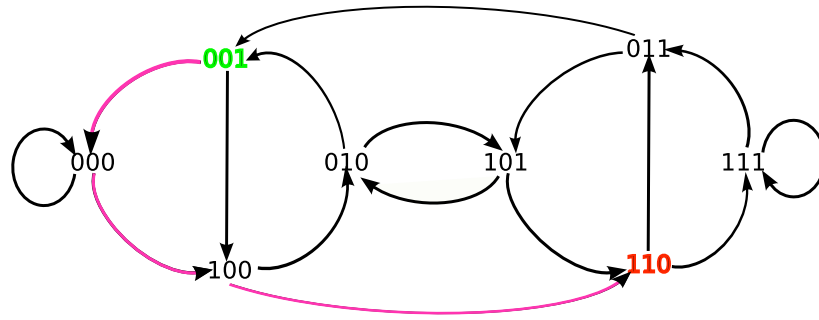


FIG. 4 – Routage depuis le nœud 001 vers le nœud 110

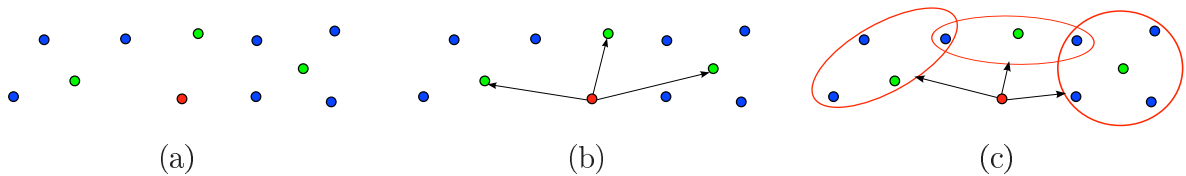


FIG. 5 – (a) le nœud considéré (en rouge) et ses voisins (en vert) dans la topologie sous-jacente. (b) liens stricts. (c) liens lâches.

La figure 5 illustre la différence entre les liens stricts et les liens lâches.

Les liens stricts sont utilisés dans des tables de hachages comme Chord, CAN, Pastry ou D2b. Les liens lâches sont utilisés dans le DNS, Kademia et Broose.

### 1.3.1 Les liens stricts (*strict links*)

Les liens stricts sont caractérisés par :

- Chaque nœud suppose a priori que ses voisins ne sont pas en panne.
- Une clé est stockée sur un seul nœud du réseau.
- Chaque nœud gère une partie de l'espace des clés, l'ensemble de ces parties forme une partition de l'espace des clés.
- Il faut prévoir des procédures de *recovery* en cas de crash de l'un des nœuds.

La recherche d'une association s'effectue en routant la requête en utilisant la clé. Le routage s'apparente à celui d'un paquet IP dans le routage d'internet.

En général, les opérations d'une table de hachage distribuée avec liaisons stricts s'effectuent de la manière suivante :

Soit  $n$  le nœud gérant la clé  $c$ .

- *join(c)* : Récupérer du nœud  $n$  une partie de l'espace des clés et les liens vers les voisins.
- *leave()* : Transférer la partie de la gestion de l'espace des clés à un autre nœud et prévenir les voisins du départ.
- *insert(c,d)* : Stocker la correspondance  $(c,d)$  dans le nœud  $n$ .
- *lookup(c)* : Réponse renvoyé au demandeur par le nœud  $n$ .

### 1.3.2 Les liens lâches (*loose links*)

Les liens lâches sont caractérisés par :

- Il y a peu d'hypothèses de faites sur l'état des nœuds voisins, en particulier, on suppose a priori qu'ils peuvent avoir disparu.
- La correspondance  $(c,d)$  est répliqué sur les  $k$  nœuds les plus "proches" de  $c$ . Voir la figure 6

- L'ensemble des parties de l'espace des clés gérées par les nœuds ne forme pas une partition mais un recouvrement de l'espace des clés.
- Il est inutile de prévoir une procédure de *recovery* en cas de crash d'un nœud, l'information a de fortes chances d'être accessible sur un autre nœud.
- Les DHT ainsi conçus sont adaptés à une forte volatilité du réseau.

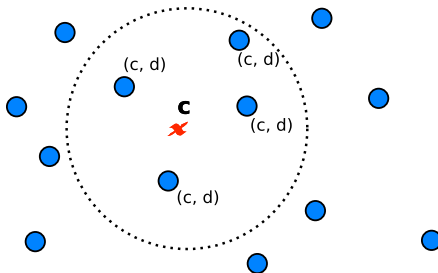


FIG. 6 – Réplication de l'association  $(c, d)$  sur les nœuds proches de  $c$

La recherche dans une table de hachage distribuée utilisant des liaisons lâches, se déroule comme suit :

- Le nœud initiant la requête demande à un nœud qu'il estime être le plus proche de  $c$  parmi ceux qu'il connaît, son meilleur *bucket* pour  $c$ .
- Le nœud initiant la requête recommence en choisissant un nœud parmi ceux qui lui ont été envoyés.

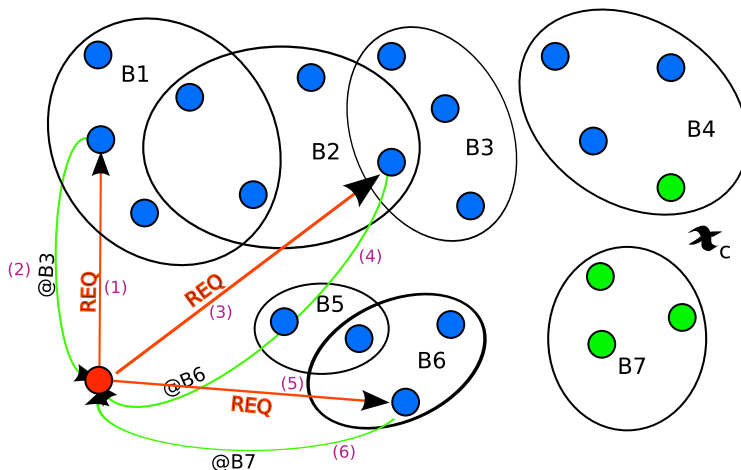


FIG. 7 – Mécanisme de la recherche dans un réseau lâche

La figure 7 montre un exemple de recherche dans un réseau lâche :

1. Le nœud initial émet une première requête vers un nœud du bucket  $B_1$ .
2. Le nœud lui répond avec les adresses des nœuds du bucket  $B_3$  qu'il estime être le plus proche de  $c$ .
3. Le nœud initial émet une seconde requête vers un nœud du bucket  $B_3$ .
4. Le nœud lui répond avec les adresses des nœuds du bucket  $B_6$  qu'il estime être le plus proche de  $c$ .
5. Le nœud initial émet une troisième requête vers un nœud du bucket  $B_6$ .
6. Le nœud lui répond avec les adresses des nœuds du bucket  $B_7$  qu'il estime être le plus proche de  $c$ .

En général, les opérations d'une table de hachage distribuée avec liaisons lâches s'effectuent de la manière suivante :

- *join(c)* : le nœud construit ses buckets.
- *leave()* : éventuellement le nœud prévient ses voisins.
- *insert(c, d)* : trouver les  $k$  nœuds les plus proches de  $c$  et écrire la correspondance sur ces nœuds.
- *lookup(c)* : trouver un bucket contenant une partie des nœuds stockant  $(c, d)$  et demander la correspondance à l'un de ces nœuds.

### 1.3.3 Quel type de liaison utiliser ?

D'après ce qui est dit précédemment, il est clair que les liens stricts vont favoriser une faible surcharge du réseau due aux messages de requête qui reste peu nombreux et de taille réduite. Cependant cela se fait au détriment de la fiabilité du réseau qui peut être sensible aux crashes de nœuds. De plus l'arrivée et le départ d'un nœud nécessite des transferts de données qui peuvent être importants. Il faut noter qu'il existe des techniques qui permettent d'augmenter la fiabilité de ces réseaux, principalement en utilisant plusieurs instances du réseau en parallèle.

Au contraire, les réseaux lâches sont très peu touchés par le départ d'un nœud que se soit de manière régulière ou dû à un crash. Cependant, une requête demande l'envoi d'un plus grand nombre de messages de taille plus importante.

Il faut donc choisir le type de table à utiliser selon l'utilisation que l'on va en faire :

- Pour un réseau que l'on sait relativement stable, il est préférable d'utiliser une liaison strict.
- Pour un réseau très volatile, il est préférable d'utiliser une liaison lâche.

## 2 Broose

### 2.1 Présentation de Broose

Broose est une table de hachage distribuée, basée sur un graphe de De Bruijn, elle utilise des liaisons lâches. Elle a été décrite par MM. Gai et Viennot en 2004.

La métrique utilisée par Broose est la distance XOR : la distance entre le nœud  $u$  et le nœud  $v$  est donnée par la valeur numérique de  $u \oplus v$ . On remarquera qu'avec cette métrique, plus le préfixe commun de  $u$  et de  $v$  est long, plus ils sont proches.

De plus, Broose possède des caractéristiques qui lui sont propres comme le rafraîchissement automatique des buckets ou la gestion intelligente des hotspots.

La recherche peut se faire de trois manières différentes :

- La recherche à droite (*Right shifting lookup*) : les buckets sont contactés successivement en suivant les liens du graphe de De Bruijn avec décalage par la droite.
- La recherche à gauche (*Left shifting lookup*) : idem que la précédente mais en utilisant le graphe des décalages gauche.
- La recherche de voisinage (*Brother lookup*) : on cherche la clé dans le voisinage d'un bucket.

Les recherches gauche et droite ont la même fonction : se rapprocher le plus possible d'un bucket contenant au moins un nœud avec l'association recherchée. La recherche de voisinage permet d'obtenir tous les nœuds qui contiennent l'association recherchée. Une recherche droite (ou gauche) suivie d'une recherche de voisinage est appelée une recherche complète (*complete lookup*).

Plusieurs paramètres régissent Broose :

- $n$  : le nombre de bits des identifiants.
- $k$  : le nombre de nœuds sur lesquels une association est reproduite. Typiquement  $k = 20$ .
- $k'$  : paramètre gérant la taille des buckets. Typiquement  $\frac{k}{2} < k' \leq k$ .
- $\alpha$  : nombre de demandes simultanées. Il dépend de la volatilité du réseau.
- $k''$  : nombre de nœuds à contacter en premier lieu dans une recherche à gauche. Typiquement  $k'' \approx \frac{k}{2}$ .

### 2.2 Recherche à droite (*Right lookup*)

Pour effectuer une recherche à droite, il faut que chaque nœud  $u$  définisse deux buckets de taille  $k'$  :

- $R_0$  : les  $k'$  nœuds les plus proches de  $0u[1, n-1]$ .
- $R_1$  : les  $k'$  nœuds les plus proches de  $1u[1, n-1]$ .

L'idée du routage de  $u$  vers  $w$  est alors de suivre la logique du routage sur le graphe de De Bruijn en appliquant un algorithme itératif qui fait en sorte qu'à la  $i^{\text{ème}}$  le bucket retourné soit celui qui est le plus proche de  $v[n-i+1, n]u[1, n-i]$ .

**Algorithme.** L'algorithme consiste, dans un premier temps à évaluer le nombre de saut nécessaire  $d$  pour atteindre la cible : on se sert de l'heuristique suivante :

- Soit  $l$  le plus long préfixe commun des nœuds dans le bucket  $R_0$  (ou  $R_1$ ) de  $u$ .

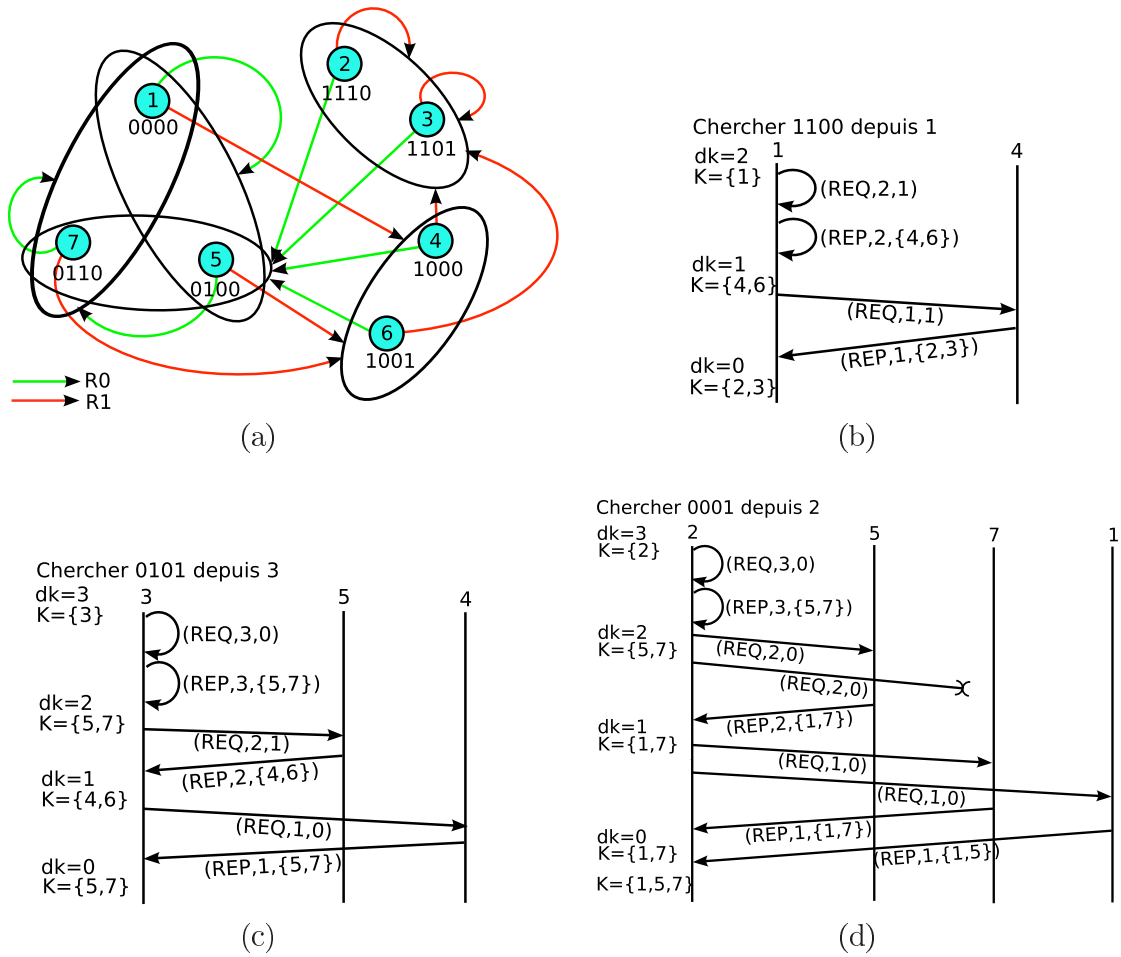


FIG. 8 – Exemples de recherche à droite sur le réseau Broose illustré en (a).

–  $d = l + 1$

L'algorithme possède deux variables :

- un entier  $d_k$  : le nombre de saut estimé restant. Valeur initiale :  $d_k = d$ .
- un bucket  $K$  : les nœuds actuellement les plus proches de la cible. Valeur initiale :  $K = \{u\}$ .

Il faut ensuite répéter les étapes suivantes jusqu'à ce que  $d_k$  vaille 0 :

- $u$  contact de 1 à  $\alpha$  nœuds dans  $K$  pour leur demander leur bucket  $R_{w[d_k]}$ .
- Si  $u$  reçoit un bucket  $R$  pour  $d_k$  sauts,  $K$  est remplacé par  $R$  et  $d_k$  est décrémenté.
- Si  $u$  reçoit un bucket  $R$  pour  $d_k + 1$  sauts,  $R$  est ajouté à  $K$ .
- Si  $u$  reçoit une réponse pour plus de  $d_k + 1$  sauts, la réponse est ignorée.

**Exemples.** La figure 8 montre quelques exemples de recherche à droite :

- Le graphe représente le réseau de Broose avec les buckets  $R_0$  et  $R_1$  de chacun des nœuds, les buckets étant délimités par les ellipses noires et les liaisons étant représenté par les flèches vertes pour  $R_0$  et les flèches rouges pour  $R_1$ . Ici  $k = k' = 2$ .
- Recherche de 1100 depuis le nœud 1 :
  - On considère  $\alpha = 1$ .
  - On commence par calculer  $d_k$  qui vaut 2 car le plus long préfixe commun de  $R_0$  est 0.
  - Requête demandant à 1 de retourner son bucket  $R_1$  car 1100. Format de la requête :  $(REQ, d_k, w[d_k])$ .
  - 1 répond en retournant les adresse de 4 et 6. Format de la réponse :  $(REP, d_k, \{v_1, \dots, v_{k'}\})$ .
  - À la réception de la réponse pour 2 sauts égale à  $d_k$ ,  $K = \{4, 6\}$  et  $d_k = 1$ .
  - Requête demandant à 4 de retourner son bucket  $R_1$  car 1100.
  - 4 répond en retournant les adresses de 2 et 3.
  - À la réception de la réponse pour 1 saut égale à  $d_k$ ,  $K = \{2, 3\}$  et  $d_k = 0$ .

- La recherche à droite s'arrête ici. On a bien le fait que l'un au moins des nœuds 2 et 3 soit parmi les 2 nœuds les plus proches de 1100.
- (c) Recherche de 0101 depuis le nœud 3 :
  - On considère  $\alpha = 1$ .
  - On commence par calculer  $d_k$  qui vaut 3 car le plus long préfixe commun de  $R_0$  est 01.
  - A partir d'ici, le déroulement est similaire à l'exemple (b) mais avec un saut de plus.
- (d) Recherche de 0001 depuis le nœud 2 :
  - On considère  $\alpha = 2$ .
  - L'algorithme est similaire mis à part le fait que l'on fait 2 requêtes en parallèle.
  - La requête pour  $d_k = 2$  à destination de 7 est perdue, mais l'algorithme peut se poursuivre car l'information arrive par ailleurs.
  - La dernière réponse arrive avec un  $d_k = 1$  alors que le  $d_k$  du nœud est passé à zéro, donc on ajoute les sites trouvés à  $K$ .

### 2.3 Recherche de voisinage (*Brother lookup*)

Comme la recherche à droite ne permet pas toujours d'obtenir le résultat escompté et en particulier ne donne pas forcément les  $k$  nœuds les plus proches d'un point donné, on utilise dans ces cas une recherche de voisinage. Cette recherche part d'un ensemble de nœuds connus pour être relativement proches de la cible, typiquement le bucket  $K$  à la fin d'une recherche à droite.

Pour effectuer une recherche de voisinage, il faut que chaque nœud  $u$  définisse un bucket  $B$ . Le bucket  $B$  stocke tous les  $\delta$  nœuds les plus proches de  $u$ . Typiquement  $\delta = 7k$ .

L'algorithme de recherche des  $k$  nœuds les plus proches de  $w$  est le suivant :

- $u$  connaît un ensemble  $K$  de nœuds proches de  $w$ .
- Envoi d'une requête au  $k$  nœuds les plus proches de  $w$  dans  $K$ .
- Chaque nœud contacté retourne les  $k$  nœuds les plus proches de  $w$  dans son bucket  $B$ .
- On recommence tant que  $k$  réponses n'ont pas été reçues.

### 2.4 Recherche à gauche (*Left lookup*)

Pour effectuer une recherche à gauche, il faut que chaque nœud  $u$  définisse un bucket  $L$ . Le bucket  $L$  stocke tout nœud  $v$  tel que  $u$  est parmi les  $k'$  nœuds les plus proches de  $u[1]v[1, n-1]$ . C'est à dire que le bucket  $L$  est constitué des nœuds  $v$  tels que  $u$  devrait appartenir soit au bucket  $R_0$  soit au bucket  $R_1$  de  $v$ .

Le bucket  $L$  peut être construit à partir du bucket  $B \cup \{u\}$ .

**Algorithme.** L'algorithme de recherche de  $w$  à partir de  $u$  est très similaire à celui utilisé pour effectuer une recherche droite. On commence par calculer le nombre de saut estimé  $d$  de la même manière que pour une recherche à droite. On initialise les mêmes variables  $d_k = d$  et  $K = u$ .

Répéter les étapes suivantes jusqu'à ce que  $d_k$  vaille 0 :

- $u$  commence par contacter les  $k''$  nœuds les plus proches de  $w \ll d_k$  puis, en cas d'échec,  $\alpha$  autres nœuds afin de leur demander les  $k'$  nœuds  $v$  de leur bucket  $L$  tel que  $v \ll d_k$  soit le plus proche de  $w$ .
- Si  $u$  reçoit un bucket  $R$  pour  $d_k$  sauts,  $K$  est remplacé par  $R$  et  $d_k$  est décrémenté.
- Si  $u$  reçoit un bucket  $R$  pour  $d_k + 1$  sauts,  $R$  est ajouté à  $K$ .
- Si  $u$  reçoit une réponse pour plus de  $d_k + 1$  sauts, la réponse est ignorée.

**Exemple.** La figure 9 montre un exemple de recherche à gauche :

- (a) Buckets  $L$  correspondant au graphe de la figure 8 (a).
- (b) Recherche de 0001 depuis le nœud 2 :
  - On considère  $\alpha = 1$  et  $k'' = 1$ .
  - $d_k = 3$
  - Requête demandant à 2 de retourner sa réponse pour 3 sauts et 0001.
  - 2 répond par les nœuds 2 et 4 car  $1110 \ll 3 = 1000 \ll 3 = 0000$  qui est le proche de 0001.

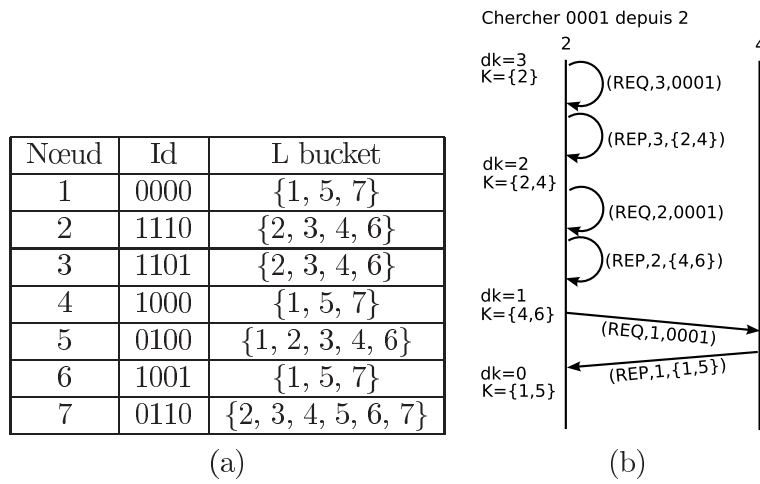


FIG. 9 – Exemple de recherche à gauche sur le graphe de la figure 8 (a).

- $d_k = 2$  et  $K = \{2, 4\}$ .
- On choisit comme prochain nœud 2 car  $0001 \ll 2 = 0100$  est plus proche de 1110 que de 1000.
- Requête demandant à 2 de retourner sa réponse pour 2 sauts et 0001.
- 2 répond par les nœuds 4 et 6 car  $1000 \ll 2 = 0000$  et  $1001 \ll 2 = 0100$  sont les plus proches de 0001. (Remarque : on aurait pu prendre 3 à la place de 6)
- $d_k = 1$  et  $K = \{4, 6\}$ .
- On choisit comme prochain nœud 4 car  $0001 \ll 1 = 0010$  est plus proche de 1000 que de 1001.
- Requête demandant à 4 de retourner sa réponse pour 1 saut et 0001.
- 4 répond par les nœuds 1 et 5 car  $0000 \ll 1 = 0000$  et  $0100 \ll 1 = 1000$  sont les plus proches de 0001.
- $d_k = 0$  et  $K = \{1, 5\}$ .

La recherche à gauche est généralement moins efficace que la recherche à droite : elle entraînera plus souvent la nécessité d'une recherche de voisinage.

## 2.5 Rafraîchissement des buckets

L'utilité de disposer des recherches gauche et droite réside dans le fait que leur symétrie permet la mise à jour automatique des buckets  $R_x$  et  $L$  :

- Pendant une recherche à droite, si  $u$  reçoit de  $v$  une réponse contenant  $u$ ,  $v$  est ajouté au bucket  $L$  de  $u$ .
- Pendant une recherche à gauche, si  $u$  reçoit de  $v$  une réponse contenant  $u$ ,  $v$  est ajouté à l'un des buckets  $R_x$  de  $u$
- D'autre part si un message contient un nœud inconnu, on regarde s'il n'a pas sa place dans l'un des buckets.

## 3 Pour en savoir plus...

Les papiers suivants décrivent en détails les mécanismes des tables de hachage distribuées citées dans ce rapport :

- Broose :
  - A. Gai et L. Viennot. A practical distributed hashtable based on the De Bruijn Topology, 2004.
- CAN :
  - S. P. Ratnasamy. A scalable content addressable network, 2002.
- Chord :
  - I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications, 2001.

- D2b :  
P. Fraignaud et P. Gauron. The content addressable network d2b, 2003.
- Kademia :  
P. Maymounkov et D. Mazieres. Kademia : a peer-to-peer information système based on the xor metric, 2002.