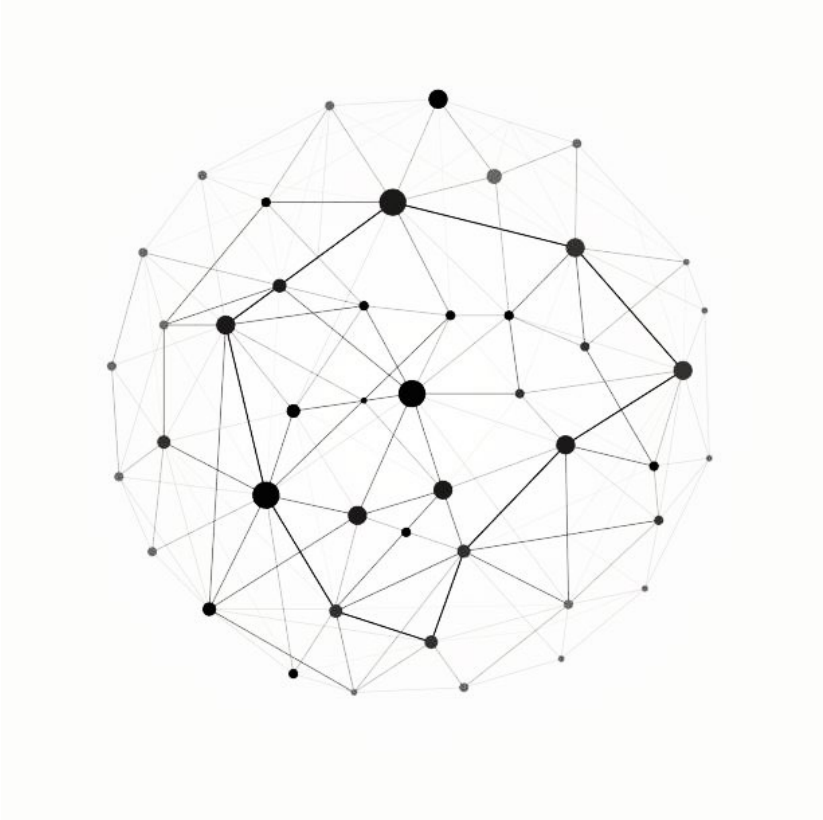
Kademlia

Protocole de routage pour les réseaux peer-to-peer décentralisés



André PINTO NUNES

EI-SE 4

Intro

L'objectif de ce travail est d'expliquer le protocole de routage Kademlia. Ce protocole se base sur une table de hachage distribuée pour des réseaux pair à pair décentralisés. Il a été créé en 2002 par Petar Maymounkov et David Mazières pour apporter des solutions aux problèmes des réseaux pair à pair existants (problèmes surtout d'ordre légale associés à la centralisation mais aussi d'extensibilité).

L'avantage des réseaux peer-to-peer (réseau maillé) par rapport aux réseaux clients-serveur (réseau en étoile) est la décentralisation. En effet, dans un réseau en étoile, tous les fichiers sont stockés dans un serveur. Ceci veut dire que le réseau dépend de ce serveur : s'il se déconnecte, le réseau n'existe plus. Dans les réseaux peer-to-peer les fichiers se trouvent dispersés dans les machines du réseau de façon redondante. Même si une machine se déconnecte, on peut trouver ses fichiers dans d'autres machines.

> *Note : Dans Kademlia, lorsqu'une machine publie un fichier dans le réseau, une clé associée au fichier est générée. Alors, la machine envoie dans le réseau la paire [clé du fichier – adresse de la machine] avec la commande STORE. Si on veut avoir le fichier, alors en cherchant sa clé dans le réseau (commande FIND_VALUE) on obtient l'adresse de la machine qui le contient. C'est donc la paire [clé – adresse] qui circule dans le réseau, pas le fichier lui-même. Kademlia* ***impose une redondance pour les paires*** *(elles sont répétées dans k nœuds),* ***mais pas pour les fichiers associés*** *(cette redondance est possible et probable mais pas imposée/garantie par le protocole).*

Le peer-to-peer pose alors un problème : si tous les fichiers sont dispersés, comment fait-on pour les retrouver ?

La solution trouvée par les premiers réseaux peer-to-peer était une table de hachage à laquelle toutes les machines avaient accès. Cette table permettait de trouver rapidement l'emplacement d'un fichier. Pourtant, ce système reste centralisé et dépendant de cette table de hachage.

Pour résoudre ce problème, il faut décentraliser les tables de hachage. On utilisera alors des tables de hachage distribuées, comme Kademlia. De plus, la méthode de calcul de distances de Kademlia, permet de retrouver un nœud ou un fichier assez rapidement, par rapport à d'autres algorithmes (comme le *Query Flooding* qui n'est pas du tout adapté pour des grands réseaux). Sa complexité logarithmique fait que 20 itérations suffisent pour retrouver n'importe quelle machine dans un réseau qui en contient 2^{20} = 1 045 576.

NodeID

Les réseaux qui utilisent Kademlia sont des réseaux de recouvrement. Des réseaux virtuels qui se superposent au réseau physique. Quand un ordinateur se connecte à un tel réseau, il est identifié par une adresse NodeID. Le *NodeID* correspond à la position du nœud dans cet espace virtuel. Il s'agit d'un numéro de 160 bits obtenu par un algorithme de hachage (SHA-1). Ce numéro sera utilisé pour calculer des distances entre nœuds mais aussi entre fichiers. En effet, quand un fichier est publié dans le réseau, il reçoit aussi une adresse qui est aussi obtenue par l'algorithme de hachage. Il est important de mentionner qu'un nœud et un fichier peuvent avoir la même adresse. On peut considérer que les fichiers sont sur encore un autre réseau qui se superpose à ceux dont on a parlé.

Distance XOR

Kademlia calcule des distances de façon non euclidienne. La distance entre deux nœuds n'est pas la différence (entre leurs *NodeID*) mais le XOR (le 'ou exclusif'). On dit que deux nœuds sont proches s'ils ont un préfixe en commun. Plus le préfixe commun est grand, plus les nœuds sont proches. Cela vient du fait que les bits en commun s'annulent avec l'opérateur XOR. Cet opérateur a des propriétés très intéressantes pour le calcul de distances. Soit d(X,Y) la distance entre les nœuds X et Y, alors : d(A,A) = 0 ; d(A,B) = d(B,A) ; d(A,B) + d(B,C)= d(A,C) pour A<B<C. De plus, toutes les distances sont uniques, il n'y a pas 2 points A et B à la même distance d'un troisième point C, ce qui facilite la recherche du nœud le plus proche. Ce mode de mesure de distances est cohérent avec la structure en arbre binaire de ce protocole, que l'on expliquera par la suite.

Les Nœuds dans Kademlia

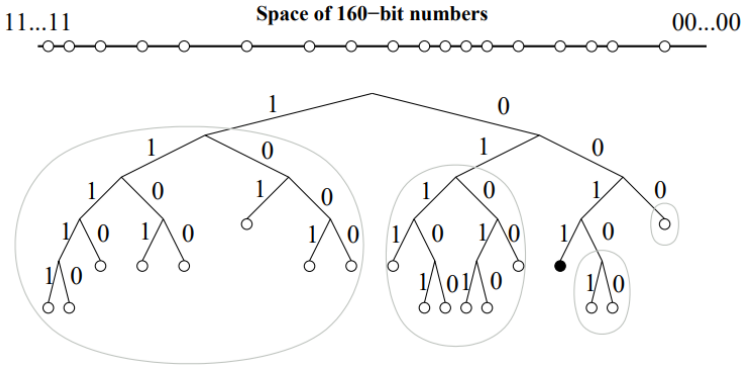


Figure 1 : Arbre Binaire : nœud de préfixe 0011
Source : pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

Un réseau Kademlia est un arbre binaire. Les feuilles de l'arbre binaire sont des nœuds. Chaque nœud a un numéro d'identification, le *NodeID*. À chaque nœud, on associe 160 listes qu'on appelle *k-bucket*. Chaque entrée de cette liste contient l'information nécessaire pour trouver un autre nœud (adresse IP, Port UDP et *NodeID*).

Comme nous pouvons voir dans la figure 1, le NodeID correspond au chemin à parcourir pour trouver un nœud dans un arbre. Dans l'image on voit un nœud noir qu'on appellera *N*. À chaque pas de 1 bit, l'arbre se divise en deux sous-arbres : le sous-arbre qui contient *N* et celui qui ne le contient pas. Sur la figure, les sous-arbres qui ne contiennent pas *N* sont entourés. Le $i^{ème}$ sous-arbre qui ne contient pas *N*, contient les nœuds à une distance entre $2^{160 - i}$ et $2^{161 - i}$ -1 de *N*. L'algorithme de Kademlia garantie que chaque nœud connait au moins un voisin de chaque sous-arbre (s'ils existent). Alors un nœud connait surtout les voisins les plus proches (car il y a beaucoup de sous-arbres proches de lui et peu de noeuds) et il connait au moins un voisin lointain. Ceci permet à tous les nœuds de retrouver n'importe quel nœud dans le réseau.

Les k-buckets

Les k-buckets sont des listes qui contiennent l'information de plusieurs nœuds (adresse IP, Port UDP et *NodeID*). Leur taille varie entre 0 et *k*. Un nœud a 160 k-buckets. Un k-bucket contient des nœuds d'un des 160 sous-arbres. Les k-buckets associés aux premiers sous-arbres, seront probablement remplis car il y a beaucoup de nœuds lointains. Les k-buckets associés aux derniers sous-arbres, ne seront probablement pas remplis (voire vides) car il y a peu de nœuds proches et beaucoup de sous-arbres. Ceci montre que le nœud connaitra tous ses voisins proches et un petit nombre de ses voisins lointains. L'ensemble des k-buckets forment la table de routage.

Messages du protocole

Le protocole Kademlia contient 4 messages RPC .

Le message PING permet de savoir si un nœud est toujours actif.

Le message STORE permet placer une paire [clé – valeur] dans un nœud.

Le message FIND_NODE permet de demander à un nœud l'information [adresse IP – Port UDP – NodeID] des *k* nœuds, parmi ceux qu'il connait, les plus proches d'une adresse passée en paramètre (numéro de 160 bits). Si le nœud en question connait moins de *k* nœuds, il envoie l'information de tous les nœuds qu'il connait.

Le message FIND_VALUE a deux comportements différents : si le nœud qui reçoit le message avait auparavant reçu un message STORE avec la clé passée en paramètre, alors il renvoie la valeur correspondante ; sinon le message a la même fonction que FIND_NODE.

Node Lookup

Le Node Lookup est le processus qui permet de trouver les *k* nœuds les plus proches d'une certaine adresse (on l'appellera la *clé*). Ce processus est récursif.

Dans l'initialisation de la récursion, le nœud qui entame le processus sélectionne un certain nombre, α, de nœuds dans le *k-bucket* le plus proche de la clé (on rappelle que « l'adresse » d'un *k-bucket* est le préfixe commun aux nœuds qu'il contient).

De façon récursive, il envoie une commande FIND_NODE à ces nœuds. La commande FIND_NODE demande à ces derniers les k nœuds les plus proches de la clé, qu'ils connaissent. Parmi les nœuds récupérés, seulement un certain nombre α est gardé pour le tour suivant de la récursion.

Si aucun des α nœuds ne renvoie un nœud plus proche de la clé que les nœuds déjà connus, alors on envoie une commande FIND-NODE à tous les autres nœuds (dans la liste des k plus proches) qui n'avaient pas été interrogés.

Une fois qu'on reçoit une réponse de tous les k nœuds, l'algorithme se termine.

Toutes les commandes de cet algorithme sont envoyées en parallèle et elles sont indépendantes pour éviter des problèmes de déconnexions. En effet, si on devait attendre l'arrivée d'une réponse pour envoyer une autre commande, l'algorithme serait beaucoup ralenti en cas de déconnexion.

L'algorithme qui permet de stocker une paire dans un nœud est identique à celui-ci, en envoyant des messages STORE aux nœuds les plus proches de l'adresse de la paire à stocker. À un moment la paire « convergera » vers le nœud dont l'adresse est la plus proche et l'ordre de stockage est envoyé à ce nœud et à ses « *k-1 voisins* ».

L'algorithme qui permet de récupérer une valeur dans un nœud est le même, mais on utilisera la commande FIND_VALUE. Cette commande, comme vu précédemment, se comporte comme FIND_NODE jusqu'à trouver le nœud qui contient la valeur souhaitée.

Republication de clés

Les clés doivent être republiées régulièrement à cause des nœuds qui intègrent ou quittent le réseau. D'une part si un nœud qui contient une clé se déconnecte, on perd cette information. Alors il faut qu'il la republie périodiquement. D'autre part, si un nouveau nœud se connecte avec une adresse encore plus proche de la clé, alors il doit impérativement l'avoir car c'est vers lui que l'algorithme de lookup va « converger » lorsqu'on cherchera à récupérer la paire.

Appliquons l'algorithme que l'on vient de voir (lookup), pour comprendre comment cela fonctionne .

Si le réseau ne bouge pas :

Le nœud qui a la clé est celui qui est le plus proche de son adresse. En la publiant, il va l'envoyer à quelques-uns de ses voisins les plus proches (entre 1 et α selon la taille du k-bucket). À leur tour, ses voisins vont l'envoyer vers leurs voisins, donc, vers le nœud de départ. Ainsi la clé se trouve en permanence dans les plusieurs nœuds autour de son adresse.

Si un nœud plus proche de la clé rejoint le réseau :

Dans ce cas il recevra rapidement la clé, grâce aux republications.

Si le nœud le plus proche de la clé quitte le réseau :

Dans ce cas il n'y a pas de problème car la clé existe ailleurs dans le réseau. Un de ses voisins les plus proches prendra sa place et il est probable qu'il contienne la clé. Si ce n'est pas le cas, il la recevra grâce aux republications (car désormais c'est vers ce nœud que l'algorithme converge).

Vulnérabilités et solutions

Ce protocole est vulnérable à certaines attaques.

L'attaque Sybil est une attaque efficace contre les réseaux décentralisés. Lors d'une attaque Sybil, un individu rejoint le réseau plusieurs fois de façon à contrôler un très grand nombre de nœuds. Cela lui permettrait d'empêcher le bon fonctionnement du réseau, en cachant des clés ou des nœuds par exemple. Il a été prouvé, par J. R. Douceur, qu'il est impossible d'empêcher ce type d'attaques.

L'attaque Eclipse ressemble un peu à l'attaque Sybil. Lors d'une telle attaque, un individu rejoint le réseau en choisissant un *NodeID* spécifique. Cela lui permet de se placer stratégiquement dans le réseau de façon à cacher certains nœuds.

Une autre faille de Kademlia est le fait qu'un seul nœud malveillant suffise pour saboter un lookup en envoyant les mauvaises réponses aux demandes.

Certains protocoles basés sur Kademlia proposent des solutions à ces problèmes, notamment S/Kademlia.

La première mesure de sécurité de ce protocole est de générer les *NodeID* à partir d'une clé publique. Cette clé publique servira alors à signer les messages. Cela empêche qu'un message soit altérée par un nœud intermédiaire. Cette signature, si elle est supervisée (c'est-à-dire, si la clé a été envoyée à une autorité de certification), est nécessaire pour freiner une attaque Sybil dans de petits réseaux.

Une deuxième mesure consiste à utiliser un puzzle pour la création du *NodeID*. D'une part, ce puzzle empêchera le choix d'un NodeID spécifique (ce qui empêche les attaques Eclipse). D'autre part, il sera assez compliqué à résoudre pour freiner la génération de plusieurs NodeID (ce qui rend plus difficile une attaque Sybil).

Pour que le protocole résiste aux nœuds malveillants isolés, S/Kademlia propose que plusieurs lookup indépendants soient faits en parallèle et de façon disjointe (leurs chemins ne se croisent jamais). Ceci permet d'augmenter la probabilité de succès d'un lookup.

BIBLIOGRAPHIE

Tables de hachage:

https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Hash_tables_and_hashing

S/Kademlia:
https://www.researchgate.net/publication/4319659_SKademlia_A_practicable_approach_towards_secure_key-based_routing

Kademlia:

https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

https://www.youtube.com/watch?v=3Y6vLTzM7zA

https://www.youtube.com/watch?v=w9UObz8o8lY

Sybil attack:

https://cacm.acm.org/magazines/2010/10/99498-peer-to-peer-systems/fulltext