**Compte rendu**

* **INFO833 -**

**Construction d’une DHT**

# **Introduction**

Il nous a été demandé lors de ce TP de mettre en place une DHT fonctionnelle. Pour cela, il nous a fallu partir du code repris du TD nous apprenant à manier *Peersim* afin de mieux appréhender le sujet.

Vous retrouverez l’entièreté du code sur mon GitHub.

# **Table des matières**

[Introduction 2](#_Toc70612793)

[Table des matières 2](#_Toc70612794)

[Étape 1 3](#_Toc70612795)

[*Le Join* 3](#_Toc70612796)

[*Le Leave* 4](#_Toc70612797)

[Étape 2 5](#_Toc70612798)

[Étape 3 5](#_Toc70612799)

[Étape 4 6](#_Toc70612800)

# **Étape 1**

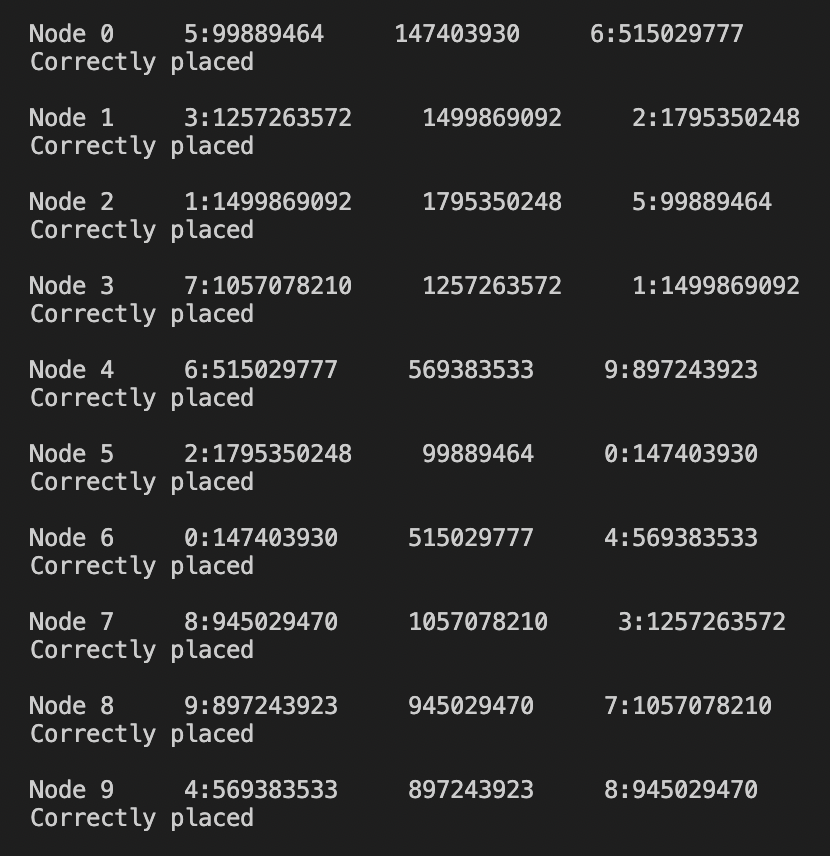
Afin de construire la DHT, je me suis fortement inspiré du fonctionnement d’une liste doublement chaînée. C’est pourquoi j’ai attribué à chaque nœud, un identifiant *random* ainsi que deux voisins (gauche et droite).

L’objectif est de construire cette DHT en partant d’un seul nœud aléatoire et de venir connecter les nouveaux nœuds qui viendront s’insérer au bon endroit i.e. de façon à conserver un rangement croissant des identifiants des nœuds.

## ***Le Join***

L’idée derrière ce *join* est de venir s’assurer que l’ordre de la DHT déjà établie est conservé afin de conserver la possibilité du parcours de proche en proche. Il faut aussi s’assurer que le voisinage est mis à jour afin de ne pas se retrouver avec un anneau ouvert.

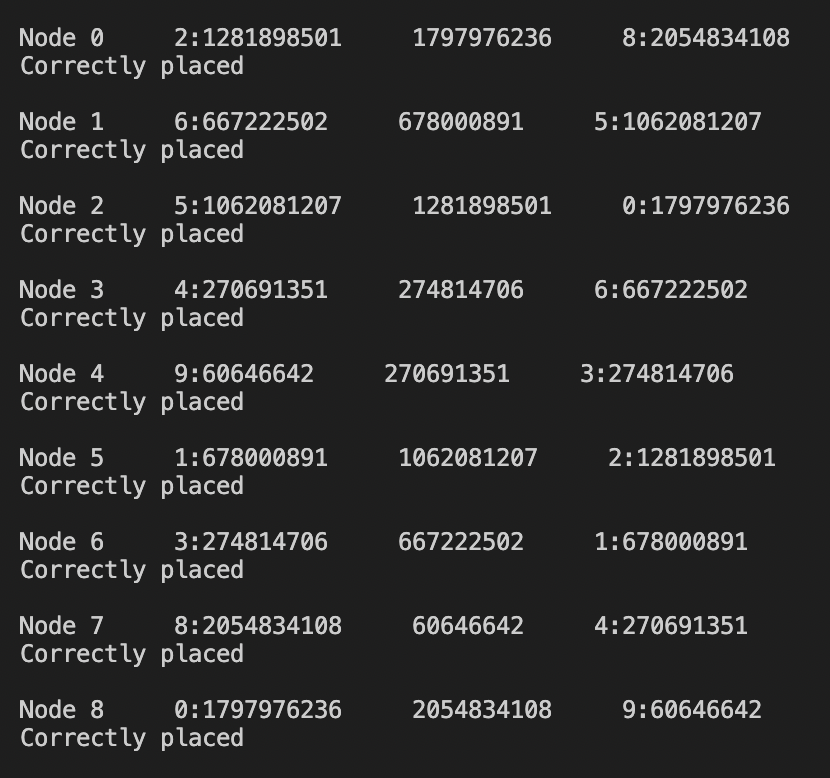
Ci-après, un *screenshot* d’un de mes tests permettant de voir la DHT dans son état final, après la création et l’insertion de 10 nœuds :

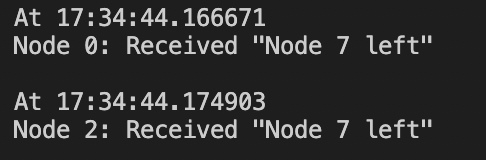


Sur chaque ligne on peut donc lire, le nœud concerné, les informations de son voisin de gauche ( [index]:[identifiant] ), son identifiant et les informations relatives à son voisin de droite. On retrouve aussi un indicateur permettant de savoir si le nœud est correctement placé dans la DHT.

## ***Le Leave***

Dans la même idée que pour le *join*, le but est de conserver l’intégrité de la DHT tout en enlevant un « maillon de la chaîne ».





Après avoir demandé le retrait du nœud 7 de la DHT, nous nous retrouvons donc avec une DHT de 9 nœuds (réindexé afin de garder une cohérence). Juste avant d’être retiré du réseau, le nœud 7 prévient ses voisins directs de son départ. On observe alors la réception dudit message par les nœuds 0 et 2 (les anciens voisins du nœud 7).

# **Étape 2**

L’envoi d’un message à un nœud de la DHT (pour informer un changement par exemple (voir la partie [« Le Leave »](#_Le_Leave))) est relativement trivial grâce à l’agencement réfléchi de la DHT. Il suffit de parcourir la DHT jusqu’au nœud de destination fourni et de commander un envoie de message.

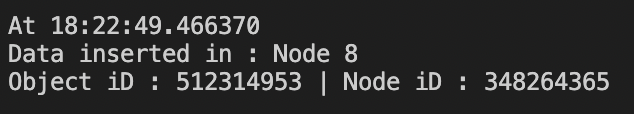
Afin de pouvoir suivre le déroulement des actions entreprises, il est intéressant d’implémenter un système permettant d’afficher des métadonnées relatives aux communications effectuées (date, émetteur, destinataire…).

# **Étape 3**

L’avantage de pouvoir communiquer avec les nœuds de la DHT permet la transmission facile de donnée parmi les nœuds. Ainsi, il est possible de faire circuler ou de stocker n’importe quel type d’objet (à condition qu’il ait un identifiant) et d’y accéder rapidement et efficacement.

Pour des soucis de simplification afin de ne pas à avoir à créer une classe abstraite permettant la création d’un objet quelconque, avec un identifiant et une fonction permettant d’y accéder, j’ai pris la liberté d’utiliser l’objet Message en guise d’objet à stocker (en utilisant son type comme identifiant aléatoire).

D’un point de vue de la conception, j’ai attribué à chaque nœud une liste de *HashMap* contenant comme clé l’identifiant de l’objet contenu et en valeur l’objet. Le nœud stockant l’objet étant celui ayant un identifiant le plus proche de celui de l’objet (simple recherche de minimum d’écart entre les différents identifiants).

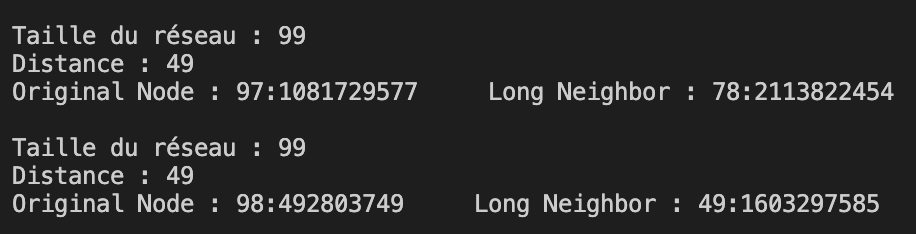


Par soucis de conservation de l’intégrité des données de la DHT, j’ai aussi veillé à la mise à jour du stockage par réplica lors des *join* et des *leave*. Ainsi, ma DHT ne se retrouve pas avec des « trous » dans les données partagées.

# **Étape 4**

Afin de mieux appréhender les DHT les plus conséquentes, j’ai décidé de donner un accès à un voisin lointain à tous les nœuds du réseau. Ce voisin lointain se situe simplement à l’opposé de la DHT afin que le raccourci soit le plus efficace.

Le problème de cette implémentation réside dans la mise à jour de ces voisins lointains lors de *join* ou de *leave*. En effet, sans mise à jour et en simulant une expansion rapide du nombre de nœud dans le réseau, on se retrouve avec des voisins pas plus lointains que les voisins directs. Afin de ne pas rendre cette opération plus gourmande en ressource qu’elle ne l’est déjà, j’ai établi des principes permettant de guider la démarche :

* Les mises à jour se font de moins en moins avec l’augmentation de la taille du réseau :
  + Là où l’ajout d’un nœud dans un petit réseaux créer un décalage conséquent dans la DHT, ce même ajout dans un réseaux de 10,000 nœuds est de suite bien plus insignifiant. Ainsi, la mise à jour se fait dès que le réseau a grandi de 10% depuis la dernière mise à jour.
* Dès qu’un nœud quitte la DHT, je force une mise à jour afin qu’aucun nœud ne se retrouve sans voisin lointain.
  + Je pars de l’hypothèse qu’il y a relativement peu de nœuds qui quittent le réseau.
* En dehors des mises à jour, il y a un calcul grossier du voisin lointain.
  + Ce calcul est fait à partir du voisin lointain du voisin de gauche afin de ne pas parcourir l’ensemble de la DHT.