Projet INFO833 – DHT

PACCOUD William – OUKZIZ Salma – TAKAHASHI Vincent

Objectif

L'objectif de ce projet est de simuler une Table de hachage distribuée à l'aide du simulateur d'événements discrets Simpy.

Nous avons travaillé sur les fonctionnalités suivantes :

- L'ajout et la suppression de nœuds
- Le routing de messages entre 2 nœuds
- L'ajout et l'obtention de données contenues dans la DHT
- La réplication des données en 3 exemplaires
- Le maintien de la réplication des données lors de l'ajout, la suppression de nœuds

Construction de la DHT

Pour cette partie, nous avons créé plusieurs classes :

- Une classe DHT qui permettra la création de nœuds avec attribution des ids et le stockage de données. Cette classe sert de base à la simulation et contient un dictionnaire de couple (ip, Node) pour chaque nœud. Par la suite on ne stocke que l'adresse d'un voisin dans un nœud et on utilise ce dictionnaire pour accéder à l'objet nœud correspondant.
- Une classe nœud qui contient l'ip de ses voisins de gauche et de droite, une ip, un id, un dictionnaire de données et des méthodes pour gérer le routing de messages, s'insérer ou se retirer de la DHT, de retrouver ou stocker de la donnée et de gérer sa réplication y compris lorsqu'il y a des modifications dans la DHT.
- Une classe Data qui a une valeur et un id
- Une énumération MessageType qui permettra de représenter les types de requêtes entre nœuds pour que celui-ci puisse réagir en accord.
- Une classe Message qui contiendra un type, l'ip et l'id de l'envoyeur, la trace du message (les ip et id des nœuds par lesquels le message est déjà passé) et un contenu qui peut différer selon le type du message.

Pour construire notre DHT, nous devons d'abord initialiser notre objet DHT puis appeler la méthode create_node de la classe DHT jusqu'à avoir suffisamment de nœuds.

Ajout/Suppression de nœuds

Lorsque notre DHT a été initialisée et est encore vide, nous ajoutons le premier nœud. Nous avons géré les cas particuliers concernant l'ajouts de nœuds pour une DHT vide, composée d'un, de deux ou de trois nœuds.

- Lorsque la DHT est vide, la méthode dht.create_node créer un premier nœud et lui attribue un id aléatoire. On ajoute le couple (id, ip) à une table contenue dans l'objet DHT. Cette table nous permettra d'afficher notre DHT, de choisir un nœud aléatoire pour lancer une requête mais aussi de créer des id et ip uniques pour les prochains nœuds qui vont rejoindre la DHT.

- Lorsque la DHT contient un nœud, on ajoute un deuxième nœud à la table et on met à jour ses voisins pour qu'ils correspondent au nœud déjà présent dans la DHT, on met aussi à jour les voisins du nœuds qui est déjà dans la DHT.

- Lorsque la DHT contient 2 voisins ou plus, on choisit un nœud aléatoire dans la table des nœuds. Le nœud qui rejoint va envoyer un message de type JOIN et se bloque jusqu'à la réception d'un message de type CAN_JOIN à ce nœud aléatoire. Le nœuds en question va alors regarder si le nœud qui rejoint peut-être son voisin, dans ce cas il l'insère et lui renvoit un message de type CAN_JOIN. Dans le cas contraire, il relai le message JOIN à son voisin dont l'id se rapproche le plus de l'id du nouveau nœud, qui va répéter l'opération. Les nœuds de la DHT vont s'envoyer des messages jusqu'à trouver la bonne place pour le nœud qui rejoint, puis mettre à jours les différents voisins en envoyant des messages de type UPDATE_NEIGHBORS. Ensuite, le nœud qui a rejoint va regarder il n'y a pas une donnée parmi celles de ses voisins qui est plus proche de son id à lui, et la stocker si c'est le cas.

Les requêtes de suppression de nœuds se font à partir de la méthode leave() du nœud lui-même.

Le nœud qui va se retirer de la DHT envoie un message à ses voisins de gauche et de droite et va leur indiquer de mettre à jour leur liste de voisins en envoyant ses voisins de droite à son voisin de gauche et ses voisins de gauche à son voisin de droite.

Notez que l'on a choisi de mettre les voisins de gauche et de droite sous forme de liste et avons géré la multiplicité de ceux-ci dès le début pour pouvoir faciliter le routage plus rapide de message, que nous n'avons malheureusement pas pu implémenter faute de temps.

Routing de messages

Un message qui va être envoyé à un nœud va contenir son id de destination, qui peut être l'id d'un nœud ou l'id d'une donnée, dans ce cas le message va aller vers le nœud qui la stocke : celui dont l'id est le plus proche de celle de la donnée.

Le rooting est géré de la manière suivante :

- Lorsque l'on reçoit un message, la méthode receive_message est appelée. A l'intérieur, on traite le message en fonction de son type.

- La méthode send_message du nœud sert à envoyer un message à un nœud dont on a connaissance. On considère qu'on ne peut utiliser cette méthode directement que pour contacter un voisin.

- Si le nœud de destination du message n'est pas un voisin, la méthode route_message va s'occuper du routing et relayer le message avec send_message a son voisin le plus proche de la destination, et ainsi de suite. Il regarde l'id de destination du message (qui sera l'id du nœud destinataire ou de la donnée à placer/récupérer). Il prend la liste de tous ces voisins et la trie par id décroissante ou croissante si l'id de destination est plus petite ou grande que la sienne. Il va alors transférer le message en ajoutant son id ou son ip à la trace, au premier voisin dont l'id est plus petite/grande que l'id de destination du message. Si aucun voisin ne correspond aux conditions, il envoie le message à son voisin le plus petit/grand.

- La méthode deliver_message émet un message dont on ne connait pas le nœud de destination. Exemple : on veut obtenir la donnée d'id x, mais on ne sait pas où elle se trouve, x sera la destination du message et on appelle route_message.

- A chaque fois qu'un message est reçu, sauf exception, le nœud doit envoyer un message de type OK a l'expéditeur du message reçu. Ainsi pour déterminer si un nœud a crash, il suffit de définir un temps limite de réponse de la part d'un nœud contacter. Nous n'avons pas pu implémenter cette fonctionnalité.

Ce routing de message marche en théorie peu importe le nombre de voisins que notre nœud connaît.

Nous avons déjà utilisé ce système d'envoi de message pour la partie précédente.

Nous avons eu des difficultés dans cette partie, notamment à gérer les « bouts » de la DHT, lorsque qu'un nœud a 2 voisins avec des id plus grandes ou 2 voisins avec des id plus petites que lui.

Ajout et obtention de données

Les données que contiendra notre DHT possèdent aussi un id unique, qui peut être le même que l'id d'un des nœuds. Cet id de la donnée nous permettra de la placer dans la DHT, dans le nœud dont l'id est le plus proche de la donnée.

On appelle la méthode dht.store_data, qui va simuler la demande de stockage d'une donnée sur la DHT en choisissant aléatoirement un nœud comme point de départ.

Le premier nœud appelle une unique fois sa méthode upload_data qui va alors regarder si l'id de la donnée est plus proche de son id que de celle de ses voisins. Si c'est le cas, il va ajouter la donnée à sa liste de données stockées. Sinon, il envoie un message de type STORE_DATA à son voisin dont l'id est le plus proche du message. Son voisin va alors répéter l'opération avec la méthode handle_store_data, et ainsi de suite jusqu'à que l'on trouve une place pour la donnée.

Réplication de la donnée

Lorsque nous stockons une donnée, nous voulons la répliquer sur ses 2 voisins les plus proches. Pour se faire, nous avons ajouté un type de message REPLICATE. Lorsque qu'un nœud ajoute une donnée à sa liste de données, il envoie un message de ce type à ses deux voisins les plus proches, qui vont eux aussi stocker la donnée. Il faut faire la différence car STORE_DATA va impliquer une réplication, alors que REPLICATE indique au nœud de simplement ajouter la donnée a son dictionnaire.

Lors de la récupération de la donnée, le nœud qui cherche à la récupérer doit appeler deliver_message avec un message de type GET_DATA qui va être routé. Chaque nœud qui reçoit le message va vérifier s'il contient la donnée, sinon relayer le message. On va s'arrêter au premier nœud qui contient la donnée, ce qui va aussi distribuer les requêtes entre les trois nœuds qui contiennent la donnée.

Gestion de la réplication lors de modifications de la DHT

La réplication des données doit rester la même lors de l'ajout et la suppression de nœuds dans la DHT. C'est pourquoi nous avons dû trouver un moyen de gérer les données originales contenues dans un nœud ainsi que les données répliquées.

On différencie les données originales des données répliquées selon si elles sont présentes dans la liste des données d'un voisin (= donnée répliquée) ou de deux (= originale). Par la suite, cette implémentation nous a assez compliqué la tâche pour distinguer donnée répliquée et originale, et nous n'avions pas pensé au fait que cela ne marchait pas pour une DHT < 4 nœuds.

On ne gère que la réplication de la donnée lors de suppression de nœuds.

Pour la suppression d'un nœud

Pour les données originales, présentes chez les deux voisins :

- On regarde de quel voisin l'id de la donnée est la plus proche.
- On met à jour les voisins des nœuds adjacents
- On rappelle la méthode qui va stocker le nœud sur ce voisin. Ce voisin contient déjà la donnée mais va gérer une nouvelle fois la réplication des données avec ses nouveaux voisins comme si c'était la première fois que la donnée arrive dans la DHT. On a ajouté une condition pour que chaque nœud ne stocke pas 2 fois la même donnée.
- On quitte la DHT

Pour les données répliquées, présente seulement chez un voisin :

- On envoie un message au voisin direct qui ne possède pas la donnée.
- On met à jour les voisins des nœuds adjacents et on quitte la DHT

Pour l'ajout d'un nœud

Lorsque des données ont un identifiant plus proche de celui du nouveau nœud que du nœud sur lequel elles sont stockées, on relocalise la donnée :

- On ajoute la donnée au nouveau nœud

Problème de messages

Dans la partie précédente, nous avons eu des problèmes lors de l'implémentation de la gestion des réplications des données lors de l'ajout d'un nœud.

Nous avons implémenté notre DHT sans ordre logique particulier dans les différents messages, ce qui nous a posé des problèmes.

Pour l'ajout de nœud en conservant la logique de réplication, certains nœuds envoyaient des messages pour confirmer que leur donnée était une donnée originale et non répliqué pendant que d'autres mettaient à jour leurs voisins, or comme l'identification d'une donnée répliquée ou originale se base sur une comparaison avec ses voisins, les résultats étaient imprévisibles

Aussi, l'ajout d'un délai aléatoire lors du transport des messages au lieu d'un délai constant mène aussi à des résultats faux.

Pour pallier à ce problème, nous aurions pu imposer un ordre causal entre les messages envoyés par chacun des nœuds.