Rapport de Projet - Algorithmique Répartie

- DHT CHORD -

M1 - Systèmes et Applications Répartis

Ung Richard - 3680881

Gadouche Lotfi - 3673119

2020 / 2021

Exercice 1: Recherche d'une clé

Dans cet exercice nous avons implémenté un **DHT CHORD** qui est initialisé par un processus de manière centralisé, c'est-à-dire que ce dernier calcule l'ensemble des finger tables après avoir tiré aléatoirement les identifiants des pairs.

Comme demandé, notre code comporte tous les commentaires utiles à la compréhension de son exécution.

Exercice 2: Calcul des finger tables

Dans cet exercice, nous devons réaliser l'initialisation de la DHT CHORD avec une complexité en messages **sous-quadratique** de manière **distribuée.** Pour cela, nous supposons que les pairs sont initialement organisés en anneau bidirectionnel (pas nécessairement ordonnée en fonction des identifiants des pairs) et qu'un nombre non nul quelconque de pairs peuvent être initiateurs du calcul des finger tables.

L'exercice se décompose en trois partie :

- l'initialisation de la structure de l'anneau par le simulateur
- l'élection d'un leader et transmission des identifiants CHORD
- calcul les finger tables

Initialisation de la structure d'anneau

Dans cet algorithme le **simulateur** se contentera de tirer aléatoirement les identifiants CHORD des pairs, construire un anneau ordonné selon les rangs MPI et déterminer aléatoirement un ensemble non vide d'initiateurs parmi ces pairs. Enfin il enverra à tous les nœuds de l'anneau leurs identifiants CHORD, l'identifiant MPI de leurs voisins de gauche et de droite ainsi que leur état (s'ils sont initiateurs ou pas).

Avant de calculer les fingers tables, **chaque pair doit au moins connaître l'ensemble des ID CHORD de l'anneau** et en l'occurrence ce n'est pas notre cas parce que nous sommes dans un contexte distribué où nous n'avons pas d'information complète sur tous les pairs du réseau.

La difficulté de l'exercice réside dans la découverte de l'ensemble des identifiants chord présents en respectant la complexité demandée. Nous devons donc trouver un moyen de récolter l'ensemble des identifiants puis de les transmettre dans l'anneau avec le moins de message possible.

Election d’un leader

Pour cela, nous devons élire un nœud pour qu’il puisse initier la collecte des identifiants CHORD de l’anneau puis effectuer la **transmission** de ces identifiants à tous les nœuds.

Pour ce faire, nous utilisons l'algorithme d'élection d'un chef de **Hirschberg & Sinclair** [1980] qui possède des hypothèses similaires à notre problème:

- Les nœuds sont capables de s'organiser en **anneau bidirectionnel.**
- Chaque nœud possède un **identifiant unique** dans l'anneau (en l'occurrence id CHORD), une référence voisins[NEXT] vers son successeur, une référence voisins[PREV] vers son prédécesseur.
- Aucun nœud ne connaît la taille de l'anneau.
- Plusieurs candidats simultanés possibles **(en l'occurrence les nœuds initiateurs du calculs des fingers tables).**
- Communications fiables et **asynchrones.**
- Exécution sans faute.

A la fin de cet algorithme, **un leader sera élu** parmi les nœuds initiateurs avec pour critère **le plus grand identifiant CHORD**. Au cours de cet algorithme, il nous a été possible de déduire la taille de l’anneau et de le transmettre en même temps que l’identité du nœud elu (qui va nous être utile pour l’envoie de l’ensemble des identifiants chord).

Cette algorithme à une complexité en messages de **O(N log N)** avec **N** la taille de l'anneau et **log N =** nombre de rondes de l'algorithme.

Après l'élection du chef, ce dernier entamera l’étape de **transmission des identifiants CHORD** aux autres nœuds, c'est-à-dire qu’il enverra un message à son successeur contenant un tableau. À sa réception, chaque nœud de l’anneau y ajoute son identifiant CHORD et l’envoie à son successeur. Ce tableau effectue un tour complet de l’anneau. Il y aura donc **N envoies de messages.**

À la fin de cette étape, le leader aura en sa possession un tableau contenant **tous les identifiants CHORD** de l’anneau. Il réitère l’envoie le tableau possédant cette fois tous les identifiants CHORD de l’anneau qu’il va **trier au préalable**. À sa réception, chaque nœud enregistre les valeurs de l’ensemble des identifiants CHORD avant de le transmettre à son successeur. Lorsque le leader reçoit de nouveau le tableau, il le détruit. Il y aura donc encore **N envoies de messages**.

La transmission des identifiants CHORD aura donc une complexité en message en **O(2N)**, soit en **O(N)**.

Calcul des fingers tables

À la réception de tous les identifiants CHORD de l'anneau, un nœud peut commencer à calculer ses fingers tables avant de l'afficher.

Bilan

L'initialisation de la DHT CHORD aura donc eu une complexité en messages en **O(N log N + N)** , soit **O(N log N)**. C'est-à-dire une complexité sous-quadratique **(O(N log N) < O(N²) avec N = |Π|)**.

Exercice 3: Insertion d'un pair

Rappel : Comme la question 2 nous suggère de reprendre l'implémentation de l'exercice 1, nous allons donc supposer les hypothèses de l'exercice 1 pour cet exercice.

Lors de l'insertion d'un pair P dans l'anneau, il ne peut le faire qu'entre deux nœuds (nous supposons que le pair souhaitant s'insérer connaît son identifiant CHORD et n'est pas présent dans le système) que l'on notera **O** et **Q**.

On rappelle qu'initialement **P** ne peut envoyer des messages qu'à un unique pair (choisi arbitrairement) de la DHT, que l'on va noter **A**. Pour s'insérer entre **O** et **Q**, **P** doit connaître le pair **responsable** de sa clé id_chord_p, dans le contexte de notre algorithme, le **responsable** est **O** (le prédécesseur actuel de **Q**),et le prévenir de sa présence. Pour envoyer un message à d'autres pairs (en l'occurrence **O** et **Q**), **P** devra être informé de leur existence par un pair (**A**) déjà présent dans la DHT.

Lorsque **P** souhaite s'insérer, il a besoin de connaître l'id CHORD de son successeur et de son prédécesseur, il va donc demander a A, qui à partir de l'id chord de P pourra déduire les ids de **O** et **Q** et ainsi les prévenir de l'existence de **P**.

Cette opération est réalisé en **O(log |Π|)** messages (envoie de messages jusqu'à atteindre O et Q).

Maintenant, il faut mettre à jour les relations de précédence. **O** a donc désormais comme successeur **P**, **P** a pour prédécesseur **O** et successeur **Q** et **Q** a pour prédécesseur **P**.

Ensuite, il faut mettre à jour les tables de fingers des pairs. Pour cela, **P** a besoin de l'ensemble **Π (l'ensemble des id CHORD de l'anneau)**. Il envoie donc un message à **A** afin de le récupérer, et calcule sa table avec la fonction processus pair(). Il faut aussi mettre à jour les tables de fingers des pairs contenu dans la liste inverse(Q) , car ce ceux qui ont le plus de probabilité de contenir **P** dans leur table de finger .

Lorsqu'un pair **B** contenu dans la liste inverse(Q) doit recalculer sa finger table, il a juste à mettre à jour les identifiants qui sont égales à **Q**.

Deux cas de figures s'impose si finger_b[index] = id_chord_Q:

- ➔ (id_chord_B + 2^i) **[**M**]** ∈ **]** id_chord_O, id_chord_P **]**
- ➔ (id_chord_B + 2^i) **[**M**]** ∈ **]** id_chord_P, id_chord_Q **]**

Dans le premier cas **finger_b[index] = id chord P,** et le second **finger_b[index] = id chord Q.**

Comme la liste inverse(Q) a une taille inférieur à |Π|, la demande de mise à jour des tables nécessite un nombre de messages en **O(log|Π|)**.

L'ensemble de l'algorithme d'insertion est donc en **O(log|Π|).**

Algorithme

```
InsertionPair(P, A): //P: pair a insérer, A pair avec  qui P peut communiquer
{
        tabSucPred[] =  A.Responsable(P); // Tableau contenant les ids de O (PRED)
// Q (SUC), envoyé par A
        O= tabSucPred [Predecesseur]; //Récupère l'id du predecesseur O
        Q = tabSucPred[Successeur];       //RÉcupère l'id du successeur P

        Precedence(O,P,Q);          // Envoie des msgs à O et Q pour leur dire de faire
une MAJ des précédences
        PI = getPi(A);                  // Récupère l'ensemble PI (les idChord de la DHT)

        // Pour chaque pair dans l'inverse de Q
        for(int cpt = 0; cpt < len(inverse(Q); ++cpt){
                RecalculeFinger(inverse(Q)[cpt], Q, PI);
        }
```