Léo CHECK
Quentin MICHELETTI

Algorithmique répartie : Rendu de projet

M1 Informatique - Spécialité Systèmes et Applications Réparties

Sorbonne Université

Année universitaire 2019-2020

Exercice 2 - Calcul des fingers tables

Nous nous plaçons dans un système où les pairs sont initialement organisés en anneau bidirectionnel et qu'un nombre non nul quelconque de pairs peuvent être initiateurs du calcul des fingers tables. Après l'exécution du processus simulateur, nous faisons l'hypothèse que les processus du système connaissent l'ensemble des identifiants CHORD, notamment celui qui leur correspond, respectivement stockés dans les variables ***Π*** et ***id***. De plus, ils savent quels sont leurs processus voisins gauche et droite dans l'anneau, stockés dans les variables ***prev*** et ***next***. Enfin, leur état (s'ils sont initiateurs) est stocké dans la variable ***state***. Le calcul des fingers tables se réalisera en deux étapes.

Voici l'algorithme imaginé :

	p initiateur	p non initiateur
$SQueryid_p$	{*Spontanément, une fois}* envoyer un message <QUERYID, responsibleOf[]> à next;	
$RQueryid_p$	{un message <QUERYID, responsibleOf[]> arrive de prev} recevoir <QUERYID, responsibleOf[]>; fingerTables[NB_PROC][M]; pour tout i ∈ [1, NB_PROC], tq responsibleOf[i] == 1 faire fingerTables[i] = calculerFinger(id(i), Π); myFingers = calculerFinger(id, Π); envoyer un message <FINGER, fingerTables[][]> à prev;	{un message <QUERYID, responsibleOf[]> arrive de prev} recevoir <QUERYID, responsibleOf[]>; responsibleOf[rank] = 1; transmettre le message <QUERYID, responsibleOf[]> avec le tableau modifié à next;
$RFinger_p$	{un message <FINGER, fingerTables[][]> arrive de next} Destruction du message reçu	{un message <FINGER, fingerTables[][]> arrive de next} recevoir <FINGER, fingerTables[][]>; myFingers = fingerTables[rank]; transmettre le message<FINGER, fingerTables[][]> avec la matrice modifiée à prev;

Nous supposons l'existence des fonctions calculerFinger(int id, int *Π) et id(int rank) qui calcule respectivement les fingers table d'un pair d'identifiant CHORD id avec le tableau Π, et l'identifiant CHORD d'un rang MPI.

Validité et complexité :

En premier temps, chaque processus initiateur p envoie un message de type *QUERYID*, contenant un tableau de flag ***responsibleOf*** de taille <NB_PROC> initialisé à 0, à leur voisin de droite (**next**).

Lorsque ce message est reçu par un processus non initiateur, ce dernier mettra le flag ***responsibleOf[rank]*** à 1, où rank est le rang MPI du processus. Puis, il transmet ce message modifié à son voisin de droite.

À un moment donné, le message finira par être reçu par un processus initiateur p' (qui peut être le processus initiateur du message *QUERYID*). À sa réception, le processus prendra connaissance de l'ensemble des processus auxquels il est en charge de calculer les fingers tables, c'est à dire tous les processus i tel que ***responsibleOf[i] == 1*** (voir Fig 1). Autrement dit, ce sont tous les processus i tel que i ∈]rangMPI(p), rangMPI(p')[(au sens de la relation d'ordre cyclique vu au TD7).

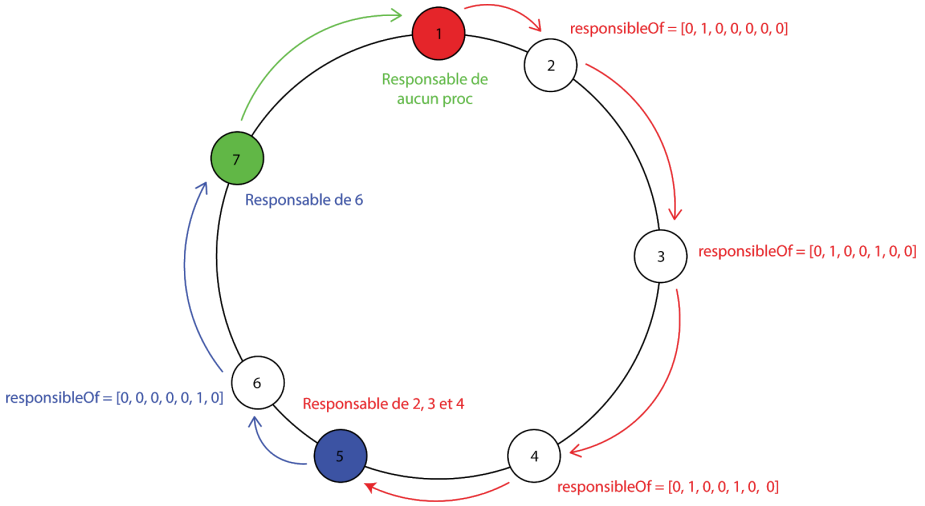

Fig 1 - Première phase de l'algorithme du calcul réparti des fingers tables

En deuxième temps, lorsqu'ils auront pris connaissance de leur responsabilité, chaque processus initiateur p calcule localement son fingers table et les fingers tables des processus dont il est responsable, et les conserve dans une matrice ***fingerTables*** de taille <NB_PROC>*<M>. Après ces calculs, ils enverront un message de type *FINGER* contenant la matrice à leurs voisin de gauche (**prev**).

Ainsi, un processus non initiateur recevant ce message prendra connaissance de son fingers table en accédant au tableau ***fingerTables[rank]***, où rank est son rang MPI. Il la conservera dans un tableau **myFingers**. Puis, il propage le message en l'envoyant à son voisin de gauche également (voir Fig 2). Finalement, ce message sera reçu par un processus initiateur p', et plus particulièrement, par le processus initiateur ayant communiqué le tableau responsibleOf à p.

La fin de l'algorithme est marquée lorsque tous les messages FINGER sont reçus par un processus initiateur. On a ainsi envoyé 2*<NB_PROC> messages, d'où une complexité en message en O(NB_PROC), soit O(|Π|).

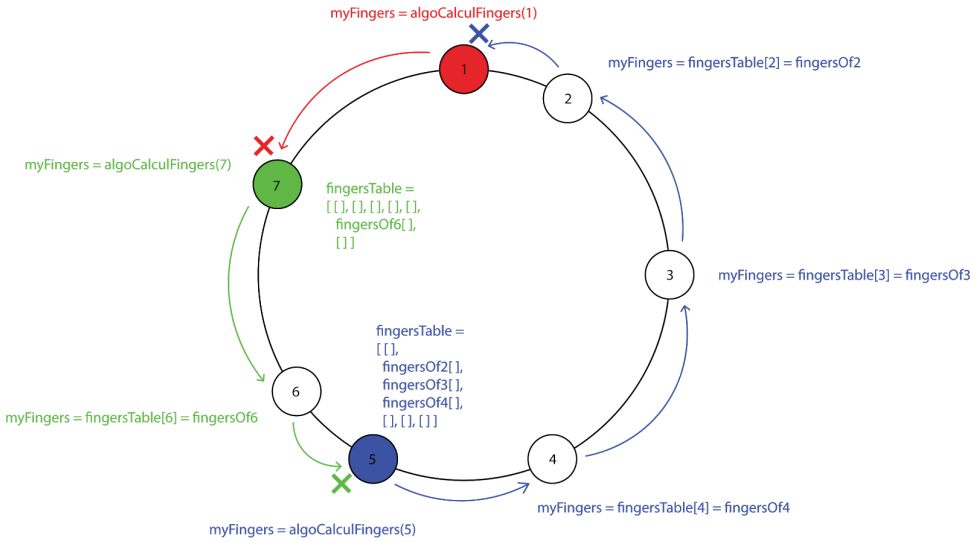

Fig 2 - Deuxième phase de l'algorithme du calcul réparti des fingers tables

Exercice 3 - Insertion d'un pair

Nous nous plaçons dans le même système DHT CHORD précédent, correctement initialisé, et les pairs de rang MPI p possède en plus une liste $inverse_p$ contenant l'identifiant (et le rang MPI) de tout pair q ayant un finger sur p. Nous supposons que le pair souhaitant s'insérer connaît son identifiant CHORD et n'est pas présent dans le système. Il ne sera capable initialement de n'envoyer des messages qu'à un unique pair (choisi arbitrairement) de la DHT. Pour envoyer un message à d'autres pairs, il devra être informé de leur existence par un pair déjà présent dans la DHT.

Soit p le nouveau pair communiquant avec un pair p' choisi aléatoirement dans le système. Voici l'algorithme imaginé:

```
join(p, p'):
        n = p'.find_responsible(p)      // Cherche qui est responsable de la clé id_p
        n' = p'.find_predecesor(n)
        n'.succ = p                     // Mise à jour du succ de p.pred
        n.pred = p                      // Mise à jour du pred de p.succ
        p.succ = n                      // Mise à jour du succ de p
        p.pred = n'                     // Mise à jour du pred de p
        get_Π(p');
        init_finger(p, Π);
        for i in inverse_n do
                fix_finger(i, p);
```

Validité et complexité :

Lorsqu'un nouveau pair est inséré dans notre système, nous devons satisfaire 2 invariants (il en existe un troisième, mais étant donné que nous nous concentrons pas sur le stockage de donnée, il ne sera pas pris en compte) :

1. Chaque successeur (resp. prédécesseur) d'un noeud correspond au successeur (resp. prédécesseur) direct dans le graph en anneau.
2. Chaque noeud doit avoir tableau de finger correct.

Supposons que le pair p souhaitant s'insérer est compris entre les pairs n et n' du système. Lorsque p souhaite s'insérer, il demandera à p' quel est le pair responsable de sa clé id_p, qui est n. Cette opération se réalise en $O(\log |\Pi|)$ messages. Une fois récupéré, nous mettons à jour les relations de prédécesseur et de successeurs : p a pour successeur n, et n met à jour son prédécesseur en y affectant p. Enfin, n' remplace son successeur n par p, et p aura pour prédécesseur n'. Nous vérifions ainsi le premier invariant. (voir Fig 3)

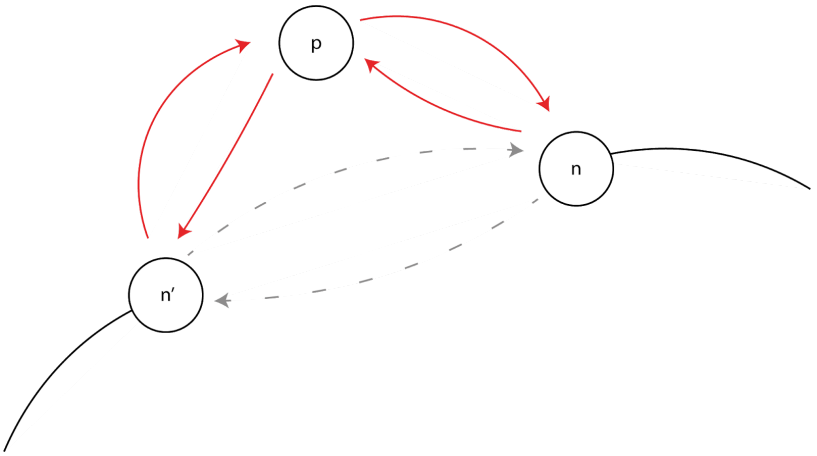

Fig 3 - Mise à jour des relations prédécesseur/successeurs:
les relations en rouge remplace les relations en pointillés grisées

Ensuite, il faut mettre à jour les tables de fingers du système. Pour cela, p a besoin de l'ensemble Π. Il envoie donc un message à p' afin de le récupérer, et initialise sa table avec la fonction init_finger(). Puis, nous mettons à jour les tables de fingers des pairs contenu dans la liste $inverse_n$, car ce sont les seuls qui auront potentiellement p dans leur nouvelle table.

En effet, lorsqu'un pair q de la liste $inverse_n$ recalcule sa table, il met seulement à jour les index de la table des fingers qui pointe vers n. Nous avons alors 2 cas:

Soit $tmp_i = (id_q + 2^i) \bmod 2^M$, si $myFingers_q[i] = id_n$

1. Si $tmp_i \in]id_{n'}, id_p]$, alors $myFinger_q[i] = id_p$
2. Si $tmp_i \in]id_p, id_n]$, alors $myFinger_q[i] = id_n$

Ainsi, les pairs q de la liste changent leurs fingers pointant vers p par n si jamais le responsable a changé. Nous validons le deuxième invariant.

Nous pouvons borner la taille de la liste $inverse_n$: $|inverse_n| <= |\Pi|$. La mise à jour des tables nécessite donc un nombre de message en moyenne inférieur à $|\Pi|$. Ainsi, l'ensemble du protocole d'insertion de p nécessite $O(\log|\Pi|)$ messages pour récupérer le successeur de p, et une complexité inférieur à $|\Pi|$ pour mettre à jour les tables du système.