Compte rendu projet CHORD (AR)

- Maxime DERRI
- Damien ALBISSON

Exercice 2 – Calcul des finger tables :

Question 1 :

Principe de l'algorithme :

Nous avons choisi de baser notre algorithme sur celui de Hirschberg & Sinclair, afin de simplifier nos preuves. Notre algorithme élit un processus leader qui s'occupe comme dans la question 1 de la création des tables et de la distribution à chacun des processus pairs, il se charge également de l'organisation en anneau unidirectionnel. Pour que le processus leader puisse gérer ses deux tâches, il a besoin d'avoir connaissance des identifiants de chacun de ses pairs. Il y aura donc un potentiel long message qui sera envoyé lors de l'algorithme (une liste compose de tous les identifiants des pairs) cependant la complexité en nombre de message est celle de l'algorithme de Hirschberg & Sinclair et donc en O(log2(n) * n) qui reste inférieur à la complexité quadratique. Ce long message qui récupère la liste de tous les identifiants des processus pairs sera envoyé par le leader une fois qu'il deviendra élu. Donc 1 message de plus est envoyé à la fin de l'algorithme de Hirschberg & Sinclair, celui parcourera tout l'anneau 1 fois. Ensuite comme dit précédemment le calcul des finger tables deviendra comme pour la question 1 centralisé (le processus leader) puis distribué à chacun des autres processus pairs.

Note pour amélioration taille des messages : On envoie un message pour avoir tous les id processus de notre anneau a la fin lors du message TERM de terminaison en même temps qu'on informe à tous les processus l'identité du leader, ce qui nous évite de renvoyer plusieurs fois inutilement une partie de ce tableau des id des processus comme ça aurait été le cas si nous avions tenté de l'envoyer dans les messages de type OUT.

Algorithme :

3 types de messages : IN, OUT, TERM (pour la terminaison et l'annonce du leader)
6 primitives :

- envoyer_VG (<msg>) // envoyer un message à son voisin gauche, contenu du message libre
- envoyer_VD (<msg>) // envoyer un message à son voisin droite, contenu du message libre
- init_list(list) // Initialise notre liste

- ajout_elt (list, elt) // ajoute l'élément elt dans la liste

1 fonction de traitement :

- creation_distribution_finger_table() // Création et distribution des finger tables comme dans l'exercice 1, de manière centralisé

constantes :

- ID(i) : identifiant

variables :
- VD(i) = identifiant voisin droite
- VG(i) = identifiant voisin gauche
- etat(i) = NSP // NSP (ne sait pas), ELU, BATTU sont les 3 états possibles
- k(i) = 0 // Numéro de l'étape de l'algorithme à laquelle nous sommes
- is_initiateur(i) = faux // Si vrai le processus est initiateur sinon faux
- nb_in(i) = 0 // Nombre de message IN reçu pour passer à l'étape suivante il faut en avoir reçu 2 (1 de chaque côté de l'anneau)

```
recevoir (<emetteur, initiateur, OUT, distance>) {
        SI (is_initiateur(i) == faux OU ID(i) < initiateur) ALORS
                etat(i) = BATTU
                SI (distance > 1) ALORS
                        SI (emetteur == VD(i)) ALORS
                                envoyer_VG(<ID(i), initiateur, OUT, distance-1>)
                        SINON
                                envoyer_VD(<ID(i), initiateur, OUT, distance-1>)
                SINON
                        SI (emetteur == VD(i)) ALORS
                                envoyer_VD(<ID(i), initiateur, IN>)
                        SINON
                                envoyer_VG(<ID(i), initiateur, IN>)
        SINON SI (initiateur == ID(i)) ALORS
                etat(i) = ELU

                // Ici pour que tous les autres processus sachent que c'est ce processus qui
                // a gagné on va envoyer un dernier message TERM, et surtout pour que le
                // processus leader puisse récupérer la liste de tous les identifiants des pairs
                list = init_list(list)
                envoyer_VG(<ID(i), TERM, list>)
        FIN SI
}
```

```
recevoir (<initiateur, TERM, list>) {
	// list[index_list] = ID(i)
	ajout_elt (list, ID(i))
	SI (ID(i) != initiateur) ALORS
		envoyer_VG(<ID(i), initiateur, TERM, list>)
	SINON ALORS
		creation_distribution_finger_table()
	FIN SI
}

recevoir (<emetteur, initiateur, IN>) {
	SI (ID(i) == initiateur) ALORS
		nb_in(i) ++
		SI (nb_in(i) == 2) ALORS
			k(i) ++
			initier_etape_suivante()
		FIN SI
	SINON
		SI (emetteur == VD(i)) ALORS
			envoyer_VG(<ID(i), initiateur, IN>)
		SINON
			envoyer_VD(<ID(i), initiateur, IN>)
	FIN SI
}

// Les processus exécutant cette fonction d'eux même sont nos initiateurs
initier_etape_suivante() {
	nb_in = 0
	envoyer_VG(<ID(i), ID(i), OUT, 2^k(i)>)
	envoyer_VD(<ID(i), ID(i), OUT, 2^k(i)>)
}
```

Justification de sa correction :

La preuve du bon fonctionnement de notre algorithme vient en grande partie des briques de base sur lesquelles nous nous appuyons. Premièrement on est sûr d'obtenir un seul leader même si on a plusieurs initiateurs dans notre réseau, en effet la réutilisation en grande partie de l'algorithme de Hirschberg & Sinclair nous l'assure (prouvé en TD). Il n'y a que 2 parties que nous ajoutons à cet algorithme, la première est l'envoie du message de fin qui récolte les identifiants de chacun des processus pairs de notre réseau. La seconde est la est la distribution des finger tables.

L’envoie du message de fin par le processus leader est un simple message utilisant l’algorithme à vague de l’anneau, nous sommes bien sur un anneau certes bidirectionnel mais ici nous n’utilisons qu’un seul de ses côtés, nous avons donc l’assurance que tous les pairs ont bien rajouté leurs identifiants dans la liste et que celui ci s’arrêtera au bout d’un tour exactement, cette brique de base nous l’assure également.

Ici la distribution des finger tables est simplement effectuée car le leader possède au moment de cette distribution l’ensemble des identifiants de chacun des pairs (prouvé par le point précédent), l’envoie d’un message contenant la finger table calculé pour un processus pair est donc trivial.

Preuve de complexité :

N : le nombre de noeud pair.

On sait que l’algorithme de Hirschberg & Sinclair a une complexité en message de l’ordre du O(log2(N) * N), l'algorithme étant une brique de base ainsi que le fait que nous l'avons déjà prouvé en TD me semble suffisant pour ne pas refaire la preuve de sa complexité en nombre de messages.

Notre algorithme, en plus de l’élection de leader, comprend une utilisation de l’algorithme à vague de l’anneau, pour la construction de la liste des identifiants des pairs. Cette brique de base a été vu et prouvé en cours, elle possède une complexité en nombre de message égal à N.

Notre algorithme comprend finalement une distribution de chacune des finger tables aux différents processus pairs. Chaque processus pair possède une finger table construite par le leader, celui-ci doit donc envoyer un message associant la finger table à chaque processus sauf lui donc N-1 messages sont envoyés, il a connaissance de chacun des identifiants des autres pairs donc 1 message est suffisant envoyé une finger table a 1 pair.

La complexité de notre algorithme est donc en O(log2(N) * (3N-1)). Dans le pire des cas la complexité de notre algorithme reste au même niveau que celle de Hirschberg & Sinclair en O(log2(N)*N) en nombre de messages, notre algorithme est bien sous quadratique.

Exercice 3 – Insertion d'un pair :

Question 1 :

Principe de l'algorithme :

Notre algorithme va être découpé en plusieurs étapes, premièrement une étape d'insertion ou le nouveau nœud voulant s'insérer va devoir trouver sa place dans la DHT en utilisant la recherche dans une DHT abordé dans l'exercice 1. Par la suite, nous construisons la finger table de ce nouveau nœud encore une fois en utilisant l'algorithme de recherche sur notre DHT pour la construire entrée par entrée et en profitant pour mettre à jour la liste inverse des nœuds sur lesquelles la finger table de notre nouveau nœud vient taper. Finalement l'algorithme termine par réellement ajouter le nœud à l'endroit calculé précédemment puis met à jour la finger table des autres nœuds concerné par son ajout, en utilisant en autre cette fameuse liste inverse que nous avons mis à jour plus tôt pour réduire la complexité en message de notre algorithme.

Algorithme :

8 types de messages : SRCH, NOTIF_KEY, SRCH_END, UPDATE_REVERSE, INSERT_NODE, UPDATE_PRED, UPDATE_FINGER

6 primitives :

- SEND (<msg>) à id_proc // Envoyer un message avec le contenu <msg> à id_proc
- compute_finger(cle, i) : Permet de calculer à partir d'une clé dans la DHT (emplacement futur du nouveau noeud), une ième entré dans la finger table calculé donc avec une puissance de 2 et un modulo en fonction du nombre de clé dans la DHT comme vu en cours
- rand_target() : Choisit une cible aléatoirement en fonction de celle compris à l'intérieur de notre DHT
- copy(destination, source, nbr_a_copie): fonction qui nous copie une zone mémoire dans destination, en partant de source jusqu'à source + nbr_a_copie
- wait(): met en attente le processus d'un stop_waiting()
- stop_waiting(): reveille le processus si un wait a été fait précédemment

constantes :

- ID(i) : identifiant
- bits_of_keys : nombre d'entrée dans la table des fingers

- nbr_keys : nombre de clé dans la DHT

variables :

- succ : Noeud successeur dans la DHT

- finger_table[ ][ ] : Table des finger pour un processus au début de l'algorithme une DHT est déjà en activité donc les nœuds déjà présent ont déjà un finger_table initialisé. Le premier indice représente l'entrée de la table (0 pour une longueur de 1, 1 pour une longueur de 2, 3 pour une longueur de 4 …). Chaque entrée possède 2 informations, la première l'identifiant (identifiant mpi par exemple) pour contacter le node, le second son identifiant dans la DHT.

- reverse_table: une table possédant toutes les clés qui référence ce nœud avec leur finger_table, ici encore potentiellement à jour car DHT déjà en activité.

- finger_updt: une table qui nous simplifiera la mise à jour de la finger table

- id: numéro d'identifiant dans la dht

```
// La recherche classique dans la DHT
// La première recherche qui sera reçu viendra du processus qui tente de se rajouter dans la
// DHT
recevoir(<SRCH, cle, initiateur>) {
        SI (ID(i) == cle) ALORS
                // Trouvé on s'arrête
                Send(<SRCH_END, ID(i), 1>) a initiateur
        FIN SI
        next = -1 // Faut il forward la requête ou nous sommes à destination
        POUR k allant de bits_of_keys à 1 FAIRE
                // Cas où l'intervalle est "normal" pas de problème avec l'indice 0 de la DHT
                SI (ID(i) < finger_table[k][1]) ALORS
                        SI (cle > id ET id <= finger_table[k][1]) ALORS
                                next = k
                                break
                        FIN SI
                // Cas ou il faut décider entre un des 2 intervalles
                SINON SI (not((cle > ID(i) ET cle < nbr_keys) OU (cle >= 0 ET cle <=
                                                        finger_table[k][1]))) ALORS
                        next = k
                        break
                FIN SI
        FIN POUR
        // Aucun next trouvé c'est donc le noeud suivant
        SI (next < 0) ALORS
                SEND(<NOTIF_KEY, cle, initiateur>) à finger_table[0][0]
        // Sinon on forward la requête et on continue
        SINON ALORS
```

```
            SEND(<SRCH, cle, initiateur>) à finger_table[next][0]
        FIN SI
}


// Le recevoir qu'on utilise pour ce type de message qu'une seule fois au moment de la fin du calcul pour trouver la nouvelle place du pair qu'on veut insérer.
recevoir(<SRCH_END, cle, ID(j), type>) {

        SI (type == 0) ALORS
            finger_table[k][0] = ID(j)
            finger_table[k][1] = cle
            SEND(<UPDATE_REVERSE, id_dht, ID(i)>) à ID(j)
            stop_waiting()
        SINON ALORS

            succ = ID(j)
            target = rand_target() // La cible qui va s'occuper de la première étape de la
                                   recherche

            id_dht = ID(j)
            // Phase de construction de la finger table du nouveau nœud.
            POUR k allant de 0 à bit_of_keys FAIRE
                // Pour une entrée de la table faire une recherche
                SEND(<SRCH, compute_finger(ID(j), k), ID(i)>) à target
                wait()
            FIN POUR

            SEND(<INSERT_NODE, id_dht, ID(i)>) à succ
        FIN SI
}

// Met à jour la liste inverse
recevoir (<UPDATE_REVERSE, id_dht, ID(j)>) {
        reverse_table[id_dht] = ID(j)
}
```

```
// Le moment ou on va réellement intégrer le nouveau noeud à notre DHT
recevoir(<INSERT_NODE, id_new_node, ID(j)>) {
        other_node_id = -1
        k = id - 1
        SI (id == 0) ALORS
                k = nbr_keys - 1
        FIN SI

        // On cherche le premier prédécesseur
        TANT QUE k != id FAIRE
                // Si c'est pas le nouveau noeud et qu'il y a bien un noeud existant, alors on a
                // trouvé notre prédécesseur
                SI (reverse_table[k] != -1 ET k != id_new_node) ALORS
                        other_id = reverse_table[k]
                        other_node_id = k
                        break
                FIN SI
                SI (k == 0) ALORS
                        k = nbr_keys
                FIN SI
                k -= 1
        FIN TANT QUE

        // On notifie notre prédécesseur du nouveau successeur
        SEND(<UPDATE_PRED, id_new_node, ID(j)>) à other_id
        reverse_table[other_node_id] = -1

        // On prépare des infos pour mettre à jour la finger table, en sauvegardant la reverse
        // table dans finger_updt
        copy(finger_updt, reverse_table, nbr_keys)
        finger_updt[id_new_node] =  -1
        reverse_table[id_new_node] = ID(j)
        // Pour garder les informations sur le nouveau noeud
        finger_updt[nbr_keys] = id
        finger_updt[nbr_keys+1] = ID(j)

        // On cherche le premier noeud qui est référencé dans la reverse table et qu'on doit
        // donc mettre à jour
        POUR k allant de 0 à nbr_keys FAIRE
                SI (finger_updt[k] > -1) ALORS
                        tmp = finger_updt[k]
                        finger_updt[k] = -1
                        SEND(<UPDATE_FINGER, finger_updt, other_id>) à tmp
                        break
                FIN SI
        FIN POUR
}
```

```
// On va voir si dans les prédécesseur on a besoin de mettre à jour la finger table
recevoir(<UPDATE_PRED, id_new_node, ID(j)>) {
        next = finger_table[0][1]

        POUR k allant de 0 à nbr_keys FAIRE
                SI (finger_table[k][1] != next) ALORS
                        continue
                FIN SI
                // tmp contient l'entrée de finger au rang k
                tmp = compute_finger(id, k)
                // Si l'entrée de finger a bien changé du au new node on change la
                // finger_table en ajoutant celui-ci
                SI (id < tmp ET tmp <= id_new_node) ALORS
                        finger_table[k][0] = ID(j)
                        finger_table[k][1] = id_new_node
                // Encore une fois une deuxième comparaison pour faire la comparaison vu
                // que l'indice 0 peut changer la donne
                SINON SI ((id < tmp ET tmp < nbr_keys) OU  (tmp >= 0 ET tmp <=
                id_new_node)) ALORS
                        finger_table[k][0] = ID(j)
                        finger_table[k][1] = id_new_node
                FIN SI
        FIN POUR
}

/ On va voir si on besoin de mettre à jour notre finger table
recevoir(<UPDATE_FINGER, finger_updt(j), pred_id>) {
        finger_updt = finger_updt(j)
        // Nouveau finger
        other_id = finger_updt[nbr_keys]
        other_node_id = finger_updt[nbr_keys+1]
        tmp = finger_table[0][0]

        POUR k allant de 0 à nbr_keys FAIRE
                SI (finger_table[k][1] == other_node_id) ALORS
                        tmp = compute_finger(id, k)

                        // Ici on vérifie si on a besoin de changer par rapport celui qui se
                        // possédait l'entrée dans la finger table précédemment (le
                        // prédécesseur de notre nouveau nœud).
                        // Les 2 cas à gérer dans l'intervalle comme à chaque fois
                        SI ((pred_id < tmp ET tmp <= other_id) OU ((pred_id < tmp ET tmp <
                nbr_keys) OU  (tmp >= 0 ET tmp <= other_id)) ALORS
                                SI ((other_id != finger_table[k][1] ET ) ALORS
                                        // On notifie l'ancien finger, on enlève son entré d'où le
```

```
                                    // -1
                                    SEND(UPDATE_REVERSE, id, -1) à finger_table[k][0]
                                    // Puis on notifie le nouveau, on lui met la nouvelle
                                    // entré
                                    SEND(UPDATE_REVERSE, id, ID(i)) à other_id
                                    finger_table[k][0] = other_id
                                    finger_table[k][1] = other_node_id
                            FIN SI
                    FIN SI
        FIN POUR

        // Ici on vérifie si d'autre noeud peuvent potentiellement être changé en se basant sur
        // qui sont référencé par finger_updt, on met un -1 pour le noeud traité puis on passe
        // au suivant
        tmp = -1
        POUR k allant de 0 à nbr_keys FAIRE
                SI (finger_updt[k] > -1) ALORS
                        tmp = finger_updt[k]
                        finger_updt[k] = -1
                        SEND(<UPDATE_FINGER, finger_updt, other_id>) à tmp
                FIN SI
        FIN POUR
}
```

Justification de sa correction :

Pour s'assurer du bon fonctionnement de notre algorithme nous pouvons nous baser sur nos briques de base encore et toujours. Ici celle que nous utilisons majoritairement est la recherche dans une DHT en suivant le protocole CHORD, nous appuyons les 3 grandes de notre algorithme sur cette recherche que nous considèreront comme correct.

La première pour trouver notre place dans cette nouvelle DHT, se solde par la simple application d'une recherche dans la table sans aucun ajout, cette partie peut donc être considérée comme fonctionnelle.

Vient la seconde étape de création de la finger table, ici nous utilisons encore une fois cette opération de base autant de fois que d'entrée que compose la finger table. Ici nous mettons à jour pour chaque entrée la reverse table de l'autre nœud concerné, une opération simple d'un envoie de message et de changement d'une valeur d'un indice d'un tableau.

Vient pour finir l'étape d'ajout réel et de mise à jour des nœuds. On commence par chercher notre premier successeur en parcourant chacune des clés de notre DHT en sens inverse en partant de notre nouveau nœud, nous trouverons forcément un prédécesseur si notre réseau est composé d'au moins 2 nœuds car ici la recherche s'apparente à celle faite sur un anneau. Une fois ce prédécesseur trouvé, on parcourra chacune des entrées pour voir si

celle-ci n'ont pas changé dû à l'ajout du successeur, le prédécesseur aura donc bien sa table à jour.

Il ne reste plus qu'à modifier les potentielles autres nœuds, pour ce faire nous allons parcourir la sauvegarde notre finger_updt qui nous indique les nœuds qui peuvent maintenant potentiellement pointer vers ce nouveau nœud. A chaque nœud concerné cette nouvelle table se remplit de -1 et lorsqu'elle deviendra entièrement composée de -1, tous les nœuds seront à jour. Cette dernière étape a donc bien mis à jour chacun des nœuds qui pouvait changer. Nous retournons donc dans un état stable où chaque table a été mise à jour et ou le nouveau nœud possède lui-même une table conforme à notre protocole.

Preuve de complexité :

N : le nombre de nœuds pair, K : le nombre de clés dans la DHT.

Dans cet algorithme nous nous basons en grande partie sur le fait qu'une recherche dans une DHT avec un protocole CHORD comme ici présent possède une complexité dans le pire cas en nombre de messages de O(log(N)).

Nous commençons par faire une première recherche de clé pour savoir ou ranger notre nouveau nœud, cette recherche s'effectue donc comme précédemment en O(log(N)).

Vient par la suite le calcul de la finger table de ce nouveau nœud. Ici nous envoyons autant de message qui provoqueront une recherche en O(log(N)) nous pouvons dire que ce nombre de recherche sera borné à log(K) car nous savons qu'une finger table contient log(K) d'entrée dans celle-ci, ici nous faisons une recherche par entrée. Nous sommes donc ici en O(log(K) * log(N)) pour construire la finger table de ce nouveau nœud.

Nous allons par la suite devoir mettre à jour les finger table des différents noeuds déjà présents dans la DHT, on commence par envoyer 1 unique message UPDATE_PRED pour prévenir le prédécesseur que le successeur a changé. Par la suite nous envoyons également 1 seul message UPDATE_FINGER.

Le message UPDATE_PRED ne provoquera par la suite aucun autre envoie de messages.

Le message UPDATE_FINGER quant à lui dans un cas ou les noeuds dans la DHT sont plutôt mal répartis (toutes nos noeuds collé aux même endroits dans notre DHT) alors il est possible que chaque entrée de la finger table viennent tapé sur le même noeuds dans ce cas ci, notre boucle qui fait log2(K) itérations enverra des messages à chacune d'entre elles. 2 messages par itération donc du 2*log2(K) messages. On se retrouve alors avec du O(2*log(K)).

Pour conclure ici en calculant chacune de ses complexité nous obtenons du :

O(2log(K) + log(N) + 2) en nombre de message moyen. Ici K nous pose problème car celui-ci peut être choisi arbitrairement. Mais si on se base sur l'hypothèse que K est bien choisi raisonnablement par rapport à N alors on peut assurer une complexité en message arrondis de l'ordre du O(log(N)).