Projet d'algorithmique répartie

Binôme : Eisha Chen-yen-su et Lydia Rodriguez de la Nava

Note : nous supposerons qu'on ait le type ID = (Hash, Rank), qui est donc une paire entre identifiant Chord (un hash) et rang MPI d'un pair de la DHT.

Exercice 1 :

Question 2 :

Types de messages:

- <FORWARD, key: Hash, caller: ID>
- <SEARCH, key: Hash, caller: ID>
- <RESULT, key: Hash, caller: ID, holder: ID, result: bool>

Variables locales des processus:

- P: Rank
- Id_p: ID = (f(p), p)
- Succ_p: ID
- Finger_p: Map<key: Hash, holder: ID>
- Keys_p: List<key: Hash>

Pseudo code:

```
find_next(k: Hash) -> ID {
	For (key, holder) in finger_p.reverse_iter() {
		If k contained in ]holder, id_p] {
			Return holder;
		}
	}

	Return null;
}

lookup(k: Hash) -> (bool, ID) {
	Let target: ID = find_next(k);

	If target == null {
		Send <SEARCH, k, id_p> to succ_p;
	} else {
		Send <FORWARD, k, id_p> to target;
```

```
        }

        Wait (result, holder);

        Return (result, holder);
}

Recv <FORWARD, key: Hash, caller: ID> {
        Let target: ID = find_next(key);

        If target == null {
                Send <SEARCH, key, caller> to succ_p;
        } else {
                Send <FORWARD, key, caller> to target;
        }
}

Recv <SEARCH, key: Hash, caller: ID) {
        Let result: bool = false;

        If key contained in keys_p {
                Result = true;
        }

        If caller == id_p {
                Notify lookup with (result, id_p);
        } else {
                Let target: ID = find_next(caller);
                If target == null
                        Target := succ_p;
                Send <RESULT, key, caller, id_p, result> to target;
        }
}

Recv <RESULT, key: Hash, caller: ID, holder: ID, result: bool> {
        If caller == id_p {
                Notify lookup with (result, holder);
        } else {
                Let target: ID = find_next(caller);

                Send <RESULT, key, caller, holder, result> to target;
        }
}
```

Exercice 2 :

Question 1 :

Idée de l'algorithme :

1. Election d'un leader.
2. Le leader envoie un message qui circule dans l'anneau où tous les pairs ajoutent leur ID à un tableau.
3. Le leader diffuse le tableau des ID dans un anneau unidirectionnel.
4. Comme tous les nœuds ont à présent la liste des ID des autres noeuds de la DHT, chacun peut facilement et localement calculer sa finger table avec ces ID à disposition.

Types de messages :

- <OUT, sender: Rank, init: Rank, dist: int>
- <IN, sender: Rank, init: Rank, dist: int>

- <COLLECT, init: Rank, peers: List<Rank>>
- <ANN, init: Rank, peers: List<Rank>>

Variables locales des processus :

- P: Rank
- pred_p: Rank
- Succ_p: Rank
- Init_p: bool
- Round_p: int
- Nb_in_p: int
- State_p: {LOSER, LEADER}

- Peers_p: List<Rank>
- Finger_p: Map<key: Hash, holder: ID>

Pseudo code :

{Quand le calcul des finger tables commence et qu'on est initiateur OU quand init_p devient vrai}
init_round();

```
init_round() {
        Nb_in_p = 0;
```

```
        If round_p == 0 {
                Init_p = true;
        }

        Send <OUT, p, p, 2^round_p> to pred_p;
        Send <OUT, p, p, 2^round_p> to succ_p;
        round_p++;
}

Recv <OUT, sender: Rank, init: Rank, dist: int> {
        If init > p {
                State_p = LOSER;

                If dist > 1 {
                        If sender == pred_p {
                                Send <OUT, p, init, dist - 1> to succ_p;
                        } else {
                                Send <OUT, p, init, dist - 1> to pred_p;
                        }
                } else {
                        If sender == pred_p {
                                Send <IN, p, init, 0> to pred_p;
                        } else {
                                Send <IN, p, init, 0> to succ_p;
                        }
                }
        } else if init == p {
                State_p = LEADER;
        } else if init_p == false {
                Init_p = true;
        }
}

Recv <IN, sender: Rank, init: Rank, dist: int> {
        If init != p {
                If sender == pred_p {
                        Send <IN, p, init, dist> to succ_p;
                } else {
                        Send <IN, p, init, dist> to pred_p;
                }
        } else {
                Nb_in_p++;
                If nb_in_p == 2 {
                        init_round();
                }
        }
```

```
}

{Quand le pair devient leader}
init_peers_collect() {
	peers: List<Rank> = {p};

	Send <COLLECT, p, peers> to succ_p;
}

Recv <COLLECT, init: Rank, peers: List<Rank>> {
	If init == p {
		Peers_p = peers;
		Send <ANN, p, peers_p> to succ_p;
	} else {
		peers.add(p);
		Send <COLLECT, init, peers> to succ_p;
	}
}

Recv <ANN, init: Rank, peers: List<Rank>> {
	If init != p {
		Peers_p = peers;
		Send <ANN, init, peers> to succ_p;
	}

	Hashed_peers: List<Hash> = peers_p.hash_mpi_ranks().cyclic_sort();

	For (i = 0; 2^i < floor(dht_size() / 2); i++) {
		Entry: Hash = (f(p) + 2^i) mod 2^dht_size();

		Let (i, j): (Hash, Hash) = entry contained in ]i, j] and j.cyclic_successor_of(i);
		Holder: Hash = j;

		finger_p.insert((entry, (holder, mpi_rank_of(holder))));
	}
}
```

Justification de la correction de l'algorithme :

Pour l'élection, les communications sont fiables et l'exécution est sans faute, donc on est assuré que l'algorithme de Hirschberg & Sinclair fournit à terme un pair leader dans l'anneau.
Le leader envoie un message à un seul de ses voisins. A la réception d'un message, un pair renvoie un message, sans condition, à son autre voisin. Les communications sont sûres et sont FIFO, ainsi le passage du message ne s'arrête pas tant qu'il n'est pas revenu au

leader. On en déduit donc que tous les pairs de l'anneau ont inséré leur identifiant, et qu'ils ont ensuite connaissance pour chacun de tous les identifiants

Complexité de l'algorithme :

Puisque les noeuds sont organisés en anneau bidirectionnel, on utilise l'algorithme d'élection de Hirschberg & Sinclair. Celui-ci a pour complexité en message de O(nlog n). Les deux étapes suivantes vont parcourir chacune une fois la topologie, donc produisent n messages chacunes. On a alors dans l'algorithme entier O(nlog(n)+n) = O(nlog(n)), et on a donc bien une complexité sous-quadratique.

Exercice 3 :

Question 1 :

Idée de l'algorithme :

1. Trouver "successeur": on appelle lookup(hash(id)) et on regarde qui répond (on peut savoir cela, car on a défini que lookup renvoie non seulement si la clef a été trouvée, mais aussi qui en est le responsable).
2. Remplir finger table (l'ID pour la première entrée est le successeur) :
    a. Calculer les clefs de chaque entrée de la finger table.
    b. Tant qu'il y a des entrées manquantes :
        i. lookup(hash de la première entrée vide)
        ii. Enregistrer, dans l'entrée vide, l'ID de qui a répondu.
3. Prévenir les pairs dont la finger table est potentiellement modifiée. Ces pairs sont ceux qui référencent le successeur du nouveau pair dans leur finger table. On peut les retrouver grâce la liste "inverse_p".

Types de messages :

- <NOTIFY, new_peer: ID>
- <JOINED, sender: ID, new_peer: ID>

Variables locales des processus :

- P: Rank
- Id_p: ID = (f(p), p)
- Succ_p: ID
- Finger_p: Map<key: Hash, holder: ID>

Variables locales au nouveau pair :

- Entry_point_p: Rank // Le rang MPI du pair connu par le processus

Variables locales aux pairs déjà présents dans la DHT :

- Inverse_p: List<ID>

Pseudo code :

```
join_dht() {
	Let (_, succ_p): (bool, ID) = lookup(id_p);

	For (i = 0; 2^i < floor(dht_size() / 2); i++) {
		Entry: Hash = (f(p) + 2^i) mod 2^dht_size();

		Let (_, holder): (bool, ID) = lookup(entry);

		finger_p.insert((entry, holder));
	}

	Send <NOTIFY, id_p> to succ_p;
}

Recv <NOTIFY, new_peer: ID> {
	For peer in inverse_p {
		Send <JOINED, id_p, new_peer> to peer;
	}
}

Recv <JOINED, sender: ID, new_peer: ID> {
	For (key, holder) in finger_p {
		If (holder == sender) and (key not contained in ]new_peer, holder]) {
			finger_p.update((key, new_peer));
		}
	}
}
```

Justification de la correction de l'algorithme :

Pour trouver le successeur et pour remplir la finger table du nouveau pair, on utilise simplement le lookup, à la fois pour atteindre le pair recherché, et pour retourner le résultat au nouveau pair. Les communications sont sûres, donc on sait qu'on aura toujours un résultat et donc que la finger table sera correctement initialisée.
Quant à la cohérence de la DHT, puisque pour atteindre le noeud inséré il faut absolument que ce soit son prédécesseur qui lui envoie un message, il suffit de prévenir les pairs de la

liste inverse de ce dernier pour qu'ils calculent à nouveau leur finger table, et la DHT sera correcte.

Complexité de l'algorithme :

Pour chercher le successeur, on fait un lookup donc on a O(log n).
Pour remplir la finger table, on va devoir faire K fois un lookup, donc on a O(log n).
Enfin, pour la dernière étape, on doit envoyer P messages, avec P le nombre de pairs dans la liste inverse du successeur du nouveau pair.
On en conclut que l'insertion du pair se fait en O(log n), et on a bien un complexité sous-linéaire.