Contrôle Final 1 - CSC3101 - mardi 27 Janvier 2026 - 2h

Tout document écrit est autorisé. Pour répondre à une question, fournissez le code correspondant (en évitant les répétitions). Les classes sont à placer dans le paquetage par défaut. Sauf indication contraire, la visibilité des méthodes et attributs est celle par défaut. Il n'y a pas à écrire les directives `import`.

Le but de cet examen est d'écrire une table de hachage distribuée (ou DHT en anglais, de Distributed Hash Table). Une DHT est un système pair-à-pair structuré sous la forme d'un anneau orienté. Chaque pair est une machine pouvant stocker de l'information située sur l'anneau. Les DHTs sont utilisés dans de nombreux systèmes, comme BitTorrent ou IPFS.

Dans une première partie, nous verrons les principes d'une DHT. Ensuite, nous regarderons comment acheminer une requête entre les pairs afin d'y stocker de l'information.

La fin du sujet contient un extrait de code qui illustre l'usage des classes et méthodes développées dans cet examen.

1. Principes d'une table de hachage distribuée *(~60 minutes, 11pt)*

Une DHT est une table de hachage, c'est à dire une structure de données de type "clé-valeur". Chaque donnée est associée à une clé et est distribuée sur le réseau. Les clés sont organisées sous la forme d'un anneau orienté. La Figure 1 illustre une table de hachage distribuée. Dans cette figure, la DHT stocke les clés de l'intervalle $[0, 2^6 - 1]$.

Chaque pair (machine) a une position sur l'anneau qui correspond à son identifiant. Il connait le pair qui le succède dans la DHT, et celui qui le précède. Par exemple, la Figure 1 illustre 4 pairs respectivement d'identifiants 4, 12, 23 et 42. Le prédécesseur (respectivement, successeur) d'un pair est le pair qui précède (resp., suit) immédiatement ce dernier dans l'anneau. Par exemple, le prédécesseur de 12 est 4. Son successeur est 23.

[Q1a] *(1pt)* Quel est le prédécesseur de 4 dans la Figure 1 ?

[Q1b] *(2pt)* Donnez le code de la classe `Peer`. Cette classe a trois attributs: l'identifiant du pair (`int id`), son prédécesseur (`Peer pred`), et son successeur dans l'anneau (`Peer next`). La classe comporte un constructeur prenant en entrée l'identifiant du pair. Initialement, les champs `pred` et `next` sont mis à la valeur `this`.

Un pair p est en charge des clés allant de $p.pred.id + 1$ à $p.id$. Ainsi dans la figure 1, 23 est en charge de l'intervalle $[13, 23]$. Quant au pair 4, il gère les clés de l'intervalle $[43, 4]$.

[Q1c] *(2pt)* Écrivez une méthode `boolean own(int x)` qui indique si le pair est en charge de la clé x. Vous serez attentif au cas épineux où x est situé dans l'intervalle du premier pair (i.e., la situation du pair 4 en Figure 1).

Pour trouver qui détient la clé x, on exécute une requête de `lookup(x)` (comme dans une table de hachage classique). Le routage dans la DHT se fait de proche en proche en suivant le sens de l'anneau. Afin d'illustrer ceci, considérons à nouveau la Figure 1. Si on cherche qui détient la clé 15 à partir du pair 4 alors on va traverser successivement les pairs 4, 12 puis 23.

[Q1d] *(1pt)* Écrivez une méthode `Peer lookup(int x)` qui retourne le pair en charge de la clé x. Vous utiliserez la méthode `own(int x)` écrite à la question précédente. Votre code doit gérer le cas où la DHT comporte un seul pair. Pour rappel, dans cette situation on a `succ=pred=this` sur ce pair.

[Q1e] *(1pt)* Quelle est la complexité du routage dans la DHT ?

Quand un pair p rejoint la DHT, il doit être placé au bon endroit sur l'anneau. A savoir, p est ajouté juste après le pair ayant le plus grand des identifiants plus petit que $p.id$.

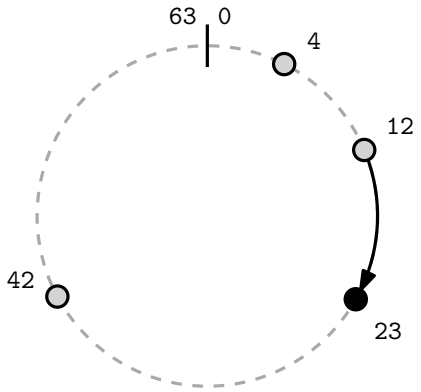

Figure 1: Illustration d'une table de hachage distribuée. La flêche noire correspond à l'intervalle $[13, 23]$, à savoir les clés du pair 23.

[Q1f] *(1pt)* Supposons qu'on ajoute le pair 36 à la DHT de la Figure 1. Quels sont le successeur et le prédécesseur de 36 après cet ajout ?

Après un ajout, on doit mettre à jour les champs de tous les pairs impactés. En détail, le pair suivant le nouveau pair doit avoir son champ `pred` à la bonne valeur. C'est aussi le cas du champ `succ` du pair étant prédécesseur du nouveau pair.

[Q1g] *(2pt)* Écrivez une méthode `void join(Peer p)` qui permet d'ajouter à la DHT le pair qui reçoit l'appel. Le pair `p` passé en argument est un pair qui appartient déjà à la DHT. Par exemple, losrque `Peer q = new Peer(36)` souhaite rejoindre la DHT, on effectue l'appel `q.join(p)`.

La DHT permet aussi de supprimer un pair. Dans ce cas là, le prédécesseur du pair retiré devient le prédécesseur de son successeur.

[Q1h] *(1pt)* Écrivez une méthode `void leave()` qui permet au pair qui reçoit l'appel de quitter la table de hachage distribuée.

2. Création de raccourcis *(~30 minutes, 7pt)*

On souhaite maintenant améliorer le routage dans la DHT en ajoutant des raccourcis (nommés fingers en anglais). Sur un pair, ces fingers sont au nombre de $log(N)$, où N est le nombre de pairs dans la DHT. Ils sont stockés dans un ensemble ordonné de type `TreeSet<Peer>`. Pour utiliser cette structure de données, la classe `Peer` doit mettre en œuvre l'interface `Comparable<Peer>`.

[Q2a] *(1pt)* Quelle est la signature de la classe `Peer` pour mettre en œuvre `Comparable<Peer>` ?

L'interface `Comparable<Peer>` définit la méthode `public int compareTo(Peer other)`. Soient deux pairs p et q, `p.compareTo(q)` doit retourner 1 si l'identifiant de p est plus grand que celui de q. Dans le cas inverse, l'appel retourne -1. En cas d'égalité, la méthode retourne 0.

[Q2b] *(2pt)* Donnez le code de la méthode `compareTo(Peer p)`.

Pour construire les fingers, on applique le pseudo-code suivant sur le pair `p`: Soit i valant 0 à $N-1$. Pour chaque i, on retrouve le pair en charge de la clé $p.id + 2^i$ en utilisant `lookup`. Ce pair est ajouté à l'ensemble des fingers de p.

[Q2c] *(2pt)* On suppose désormais que l'on a un champ `TreeSet<Peer>` dans `Peer`, initialement vide, permettant de stocker les fingers. Écriver le code de la méthode `void computeFingers()` qui permet de créer les fingers.

Quand on route une clé x dans la DHT depuis un pair p, on regarde la table de fingers. On considère le plus grand des fingers f ayant un identifiant plus petit que x. Si x n'est pas géré par p alors *(i)* si f existe, la requête est envoyée f, et sinon *(ii)* elle est envoyée à $p.next$.

[Q2d] *(2pt)* Modifiez le code de la méthode `lookup` afin de prendre en compte les fingers du pair. Vous aurez besoin de la méthode `TreeSet.floor(Peer p)` qui retourne le plus grand des éléments plus petit ou égal à p dans la structure de données. Quel est la complexité du routage ainsi obtenu ?

3. Stockage des données *(~20 minutes, 2pt)*

Nous pouvons désormais ajouter la gestion des données à notre code. Pour ce faire, on considère que chaque pair est muni d'un attribut `Map<Integer, String> store` qui permet de stocker les données sous forme de chaîne de caractères en utilisant une clé de type entier.

[Q3a] *(1pt)* La table de hachage distribuée est codée dans la classe `DistributedHashTable`. Cette classe contient un pair (champ `Peer peer`) permettant d'accéder à la DHT. Les méthodes `void put(int k, String v)` et `String get(int k)` offre la possibilité de stocker et retrouver des données. Donnez le code de ces méthodes.

[Q3b] *(1pt)* Ajoutez une méthode `boolean verify()` à la classe `DistributedHashTable`. Cette méthode doit vérifier que les pairs de la DHT sont bien dans le bon ordre: à savoir, le successeur d'un pair (sauf le dernier) a toujours un identifiant plus grand que celui-ci.

Voici un exemple d'utilisation du code écrit dans cet examen:

```
// build DHT with N peers
int N = Integer.parseInt(args[0]);
Random rand = new Random();
Peer p = null;
for (int i=0; i<N; i++) {
    Peer q = new Peer(rand.nextInt(Integer.MAX_VALUE));
    q.join(p);
    p = q;
}
DistributedHashTable dht = new DistributedHashTable(p);

// check it
assert dht.verify();

// store some web pages
URL[] urls = {
    new URL("https://telecom-sudparis.eu"),
    new URL("https://www.ip-paris.fr/")};

for (URL url: urls) {
   InputStream is = url.openStream();
   StringBuffer buffer = new StringBuffer();
   int ptr = 0;
   while ((ptr = is.read()) != -1) {
       buffer.append((char)ptr);
   }
   dht.put(Math.abs(url.hashCode()%Integer.MAX_VALUE), buffer.toString());
}
```