Table de hachage répartie et map/reduce en BCM4Java

Cahier des charges du projet CPS 2025

Jacques Malenfant
professeur des universités

version 1.0 du 20/01/2025

Résumé

Ce document définit le cahier des charges du projet de l'unité d'enseignement Composants (CPS) pour son instance 2025. Seul support d'évaluation de l'UE, le projet vise à comprendre les principes d'une architecture à base de composants, l'introduction du parallélisme, la gestion de la concurrence et les problématiques de répartition entre plusieurs ordinateurs connectés en réseau. Des notions d'architecture et de configuration pour la gestion de la performance et le passage à l'échelle seront aussi abordées.

L'objectif du projet 2025 est plus précisément d'implanter un prototype simplifié de table de hachage répartie librement inspiré d'un protocole et d'un algorithme existants appelées Chord [1]. Dans le cadre de ce projet, il s'agit de reprendre les principes de Chord en les simplifiant et en adoptant les composants BCM4Java comme entités définissant les nœuds de la table de hachage répartie, ainsi que les entités périphériques, dont la façade de la table et les clients qui soumettent les requêtes via cette façade. Pour profiter un peu plus de cette structure de données, le projet introduit la possibilité de traiter globalement ces données par un *framework* dit *map/reduce*. Ce sera donc aussi l'occasion de vous familiariser avec deux types de logiciels très courants aujourd'hui, les tables de hachage réparties et le *framework map/reduce*.

Rappel des dates et informations importantes (détails à la fin du document)

	Le projet ***doit*** être réalisé en ***Java SE 8***.
24/01/2025	Date limite pour la formation des équipes (2 personnes).
10/02/2025	Audit 1 (5% de la note finale).
2/03/2025	Rendu de code préalable à la soutenance de mi-semestre (format `tgz` ou `zip` *uniquement*).
3-4/03/2025	Soutenance de mi-semestre (semaine des ER1, 35% de la note finale).
31/03/2025	Audit 2 (5% de la note finale).
27/04/2025	Rendu de code préalable à la soutenance finale (format `tgz` ou `zip` *uniquement*).
28-29/04/2025	Soutenance finale (semaine des ER2, 55% de la note finale).
2-3/06/2025	Soutenance de seconde session (créneau à définir, note remplaçant entièrement celle de 1re session).

1 Généralités

1.1 Table de hachage répartie

Une table de hachage mémorise des couples clé/donnée où l'accès à une donnée d de clé c se fait par hachage de c vers une valeur de hachage h puis accès à la donnée dans la table elle-même par indexation sur h. L'idée maîtresse d'une table de hachage répartie consiste à répartir les données sur plusieurs tables de hachage locales elles-mêmes résidant dans autant de nœuds déployés sur différents ordinateurs. Mais lorsqu'on recherche une donnée d de clé c, se pose alors le problème de savoir sur quel nœud cette donnée a été mémorisée.

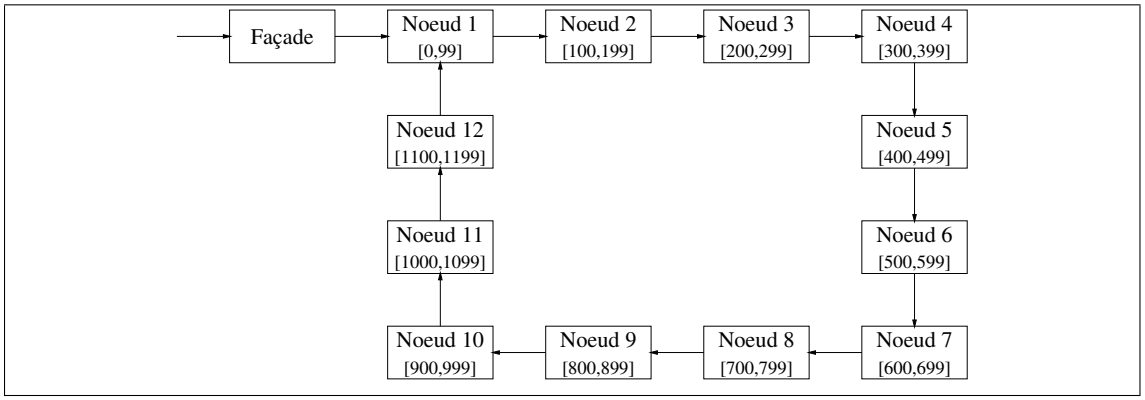

FIGURE 1 – Table de hachage répartie ; les couples présents dans chaque nœud donnent l'intervalle des valeurs de hachage des clés qui sont conservées dans la table de hachage locale du nœud concerné.

Pour résoudre ce problème, l'architecture des tables de hachage réparties de type Chord organise les nœuds en anneau puis répartit les données selon les valeurs de hachage des clés. Chaque nœud dans l'anneau prend en charge les clés dont la valeur de hachage est dans un certain intervalle, les intervalles croissants et disjoints se succédant exactement entre chaque nœud i et leur successeur $i + 1$ pour former une partition de l'ensemble de l'espace de hachage (ensemble des valeurs de hachage possibles).

Pour trouver une donnée dont la clé c_0 a pour valeur de hachage h_0, la requête est passée au premier nœud qui regarde si h_0 est dans son intervalle. Si c'est le cas, le premier nœud accède à la donnée dans sa table de hachage locale, sinon elle passe la requête au nœud 2 qui recommence le processus. En supposant que h_0 est dans le domaine des valeurs admises dans la table de hachage, un seul nœud se charge d'un intervalle contenant h_0 et si ce nœud ne contient pas de donnée correspondant à h_0, alors cette donnée n'est pas dans la table de hachage répartie. La figure 1 illustre cette organisation. Dans cet exemple, les valeurs de hachage des clés sont des entiers dans $[0, 1199]$ et chaque nœud i prend en charge les valeurs de hachage dans le sous-intervalle $[100 \times (i-1), (100 \times (i-1)) - 1]$. Par exemple, pour récupérer la donnée correspondant à une clé dont la valeur de hachage est 450, la requête sera passée successivement aux nœuds 1, 2, 3, 4 et enfin 5 qui trouvera (ou non) la donnée dans sa table de hachage locale.

Le premier objectif du projet sera d'implanter une table de hachage répartie selon ces principes en Java « pur », c'est à dire en utilisant des objets Java et des références d'objets classiques. Une interface offerte par une entité façade permettra aux clients de réaliser les principales opérations attendues sur une table de hachage, c'est à dire l'insertion de couples clé/donnée dans la table, la récupération d'une donnée à partir de sa clé et la suppression d'une donné de la table à partir de sa clé. Les interfaces du projet se conformeront globalement à l'interface `Map` des tables de hachage de la bibliothèque standard de Java modulo quelques adaptations.

Choix d'implantation généraux

Dans le cadre de ce projet, nous faisons deux choix d'implantation préalables qui vont en simplifier la réalisation. D'abord, une entité façade est introduit qui va être le seul point d'accès à la table de hachage répartie pour ses clients. Toutes les tables de hachage réparties ne font pas ce choix en permettant généralement d'initier des requêtes via n'importe quel nœud dans l'anneau. L'avantage de cette liberté d'accès est de mieux supporter la charge des clients qui ne passent pas par une unique façade. Mais cela complexifie la gestion dynamique de l'anneau qui sera introduite dans la dernière étape du projet.

Le second choix est de maintenir un anneau ayant un premier et un dernier nœud. Bien qu'il existe un lien entre le dernier nœud et le premier nœud fermant l'anneau, ce lien est traité de manière distincte dans les algorithmes qui en seront également simplifiés (il devient alors plus facile de savoir quand une requête a fait le tour de l'anneau). En réalité, à partir du moment où on introduit une façade comme seul point d'entrée dans la table, ce second choix s'accorde plus ou moins nécessairement avec le précédent.

1.2 Le modèle de calcul *map/reduce*

Pour offrir des possibilités plus intéressantes, un second objectif du projet sera d'introduire une capacité de traitement global des données insérées dans la table de hachage répartie. Cette capacité sera fondée sur le modèle de calcul *map/reduce* très populaire aujourd'hui dans le monde du traitement de données massives (« *big data* »).

1.2.1 Le modèle de base

Le modèle de calcul *map/reduce* est d'abord apparu dans les langages fonctionnels qui autorisent les fonctions d'ordre supérieur, c'est à dire des fonctions qui prennent des fonctions en paramètre. Dans sa vision la plus simple, le modèle est défini par deux fonctions d'ordre supérieur, *map* et *reduce*, qui travaillent sur les listes. La fonction *map* prend en paramètres une fonction $f : X \to Y$ et une liste $l : X^*$, applique la fonction f à tous les éléments de la liste l pour retourner une liste $r : Y^*$ des résultats obtenus dans le même ordre que la liste en entrée. La fonction *reduce* prend en paramètres une fonction de réduction $g : A \times Y \to A$, un élément neutre $a_0 \in A$ pour la fonction g et une liste $l : Y^*$. Opérationnellement, elle applique successivement g sur a_0 et le premier élément de l rendant un résultat a_1 qui est ensuite utilisé pour appliquer g à a_1 et le deuxième élément de l et ainsi de suite, réduisant donc l progressivement à une valeur finale de type A.

Par exemple, supposons que le type X correspond à des données sur des personnes et que f prend un $x : X$ et retourne son âge. La fonction *map* appliquant f à une liste $l = \{p1, p2, p3, p4, p5\}$ retournerait la liste des âges de ces personnes, par exemple :

$$map(f, l) \Rightarrow \{25, 42, 13, 75, 53\}$$

Supposons maintenant que l'objectif soit de calculer l'âge moyen $\bar{a}$ de ces personnes, la fonction de réduction g prend un couple $[n, \bar{a}]$ contenant le nombre d'entrées traitées et l'âge moyen calculé sur ces n entrées, ainsi qu'un âge a à intégrer dans le calcul pour rendre un nouveau couple tel que :

$$g([n, \bar{a}], a) = [n + 1, n/(n + 1) \times \bar{a} + a/(n + 1)]$$

À l'aide de cette fonction, la réduction de la liste $\{25, 42, 13, 75, 53\}$ en partant du couple neutre $[0, 0]$ pour g donne :

$$reduce(g, \{25, 42, 13, 75, 53\}, [0, 0]) \Rightarrow [5, 41.6]$$

Au total, le traitement complet s'exprime par la composition de *map* et *reduce* :

$$reduce(g, map(f, l), [0, 0]) \Rightarrow [5, 41.6]$$

1.2.2 Parallélisme, répartition, généralisation

Le modèle de calcul *map/reduce* se prête relativement bien à la parallélisation et à la répartition, d'où sa popularité actuelle. D'abord, on constate que *map* peut exécuter en parallèle toutes les applications de la fonction f. Néanmoins, il faudra « construire » la liste résultante [1], ce qui pzeut imposer une forme de synchronisation avec les applications de f pour ranger les résultats dans cette liste et dans le bon ordre, selon le choix de structure de données pour cette liste.

Pour la fonction *reduce*, la parallélisation est moins immédiate. Dans l'exemple précédent, on voit que chaque réduction nécessite d'avoir calculé les réductions précédentes pour obtenir le a_{i-1} à utiliser pour réduire par g l'élément y_i suivant de la liste à réduire, c'est à dire calculer $g(a_{i-1}, y_i)$. La stratégie de parallélisation consiste alors à subdiviser la liste à réduire en sous-listes de manière à pouvoir réduire ces sous-listes en parallèle. Mais dans ce cas, on se retrouve avec plusieurs résultats de réduction (un par sous-liste) qu'il va falloir combiner pour arriver au résultat final de la réduction. C'est ainsi qu'un modèle *map/reduce* légèrement plus complexe est souvent adopté, à savoir par l'ajout d'une fonction de combinaison $h : A \times A \to A$ prenant deux résultats

1. Dans une implantation parallèle, la liste résultant du *map* n'est généralement pas entièrement construite mais plutôt réduite par *reduce* au fur et à mesure.

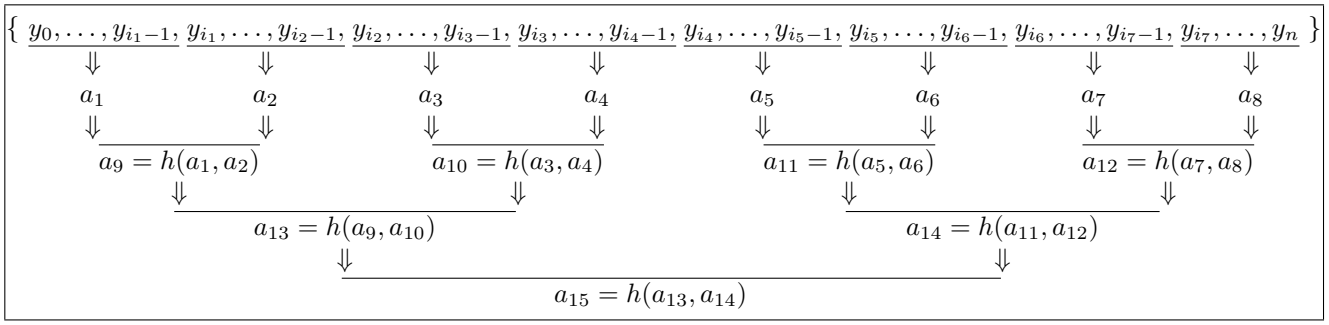

FIGURE 2 – Parallélisation de la réduction, avec utilisation des fonctions de réduction g et de combinaison h. Dans cet exemple, la liste initiale est subdivisée en 8 sous-listes traitées par une réduction par g pour donner les résultats intermédiaires $a_j = reduce(g, a_0, \{y_{i_j}, y_{i_{j+1}-1}\})$, $j = 0, \ldots, 8$ avec $i_0 = 0$ et $i_8 = n$, puis ces résultats sont combinés, par exemple de manière arborescente, par la fonction h pour donner le résultat final a_{15}.

de réduction pour rendre leur combinaison en un seul. Pour l'exemple précédent du calcul de l'âge moyen, la fonction utilisée serait :

$$h([n_1, a_{m,1}], [n_2, a_{m,2}]) = [n_1 + n_2, n_1/(n_1 + n_2) \times a_{m,1} + n_2/(n_1 + n_2) \times a_{m,2}]$$

Cette fonction de combinaison permet d'appliquer un schéma de parallélisation arborescent comme celui illustré à la figure 2 qui utilise une décomposition binaire des sous-listes.

Dans le cadre du projet, l'objectif sera d'appliquer le modèle *map/reduce* aux données de la table de hachage répartie. Il y aura donc deux niveaux dans l'application de *map/reduce* : une répartition des *map* et *reduce* entre les composants de l'anneau et le parallélisme interne à chaque composant dans l'application de *map* et de *reduce* sur les données locales de chaque nœud. Par ailleurs, si le concept de *map/reduce* a d'abord été pensé pour travailler sur des listes, il est généralisable à toute structure de données dont il est possible d'énumérer les éléments dans un ordre fixé, ici l'ensemble des données mémorisées dans la table de hachage répartie.

Notons pour votre culture professionnelle que le modèle *map/reduce*, aussi séduisant qu'il soit pour les traitements à grande échelle, n'est pas toujours si facile à utiliser. En effet, le modèle utilise des hypothèses comme le fait que les fonctions ne font pas d'effets de bord. Pour la parallélisation de *reduce*, on s'appuie également sur l'hypothèse que la fonction de combinaison h est associative (*i.e.*, $h(a_0, h(a_1, a_2)) = h(h(a_0, a_1), a_2)$ pour réaliser les combinaisons dans n'importe quel ordre après les réductions des sous-listes, ce qui simplifie et augmente le parallélisme. En pratique, il devient difficile voire impossible de faire cadrer dans ce modèle des traitements qui cherchent à modifier les données (effets de bord) ou des traitements où il est impossible de fournir une fonction de combinaison associative. Or, si la réduction ne peut utiliser une fonction de combinaison associative, elle doit sérialiser les réductions, ce qui réduit très fortement le parallélisme...

1.3 Table de hachage accélérée : introduction des cordes

Pour accélérer l'accès aux données dans les tables de hachage comportant un grand nombre de nœuds, des liens raccourcis sont ajoutés. L'idée est, pour chaque nœud i, de lui ajouter des liens directs vers les nœuds $i + 2^j$, $j = 0, \ldots, J$ où J est la plus grande puissance de 2 telle que $2^J < N$ *i.e.*, inférieur au nombre total de nœuds dans la table de hachage. La figure 3 reprend l'exemple de la figure 1 en lui ajoutant les cordes. Dans cette table de hachage, il y a 12 nœuds, donc $J = 3$ *i.e.*, $2^3 = 8 < 12 < 2^4 = 16$. Dans ce contexte, le nœud 1 est lié par des cordes aux nœuds $1 + 2^0 = 2$, $1 + 2^1 = 3$, $1 + 2^2 = 5$ et $1 + 2^3 = 9$, Pour le nœud 9, il n'est lié qu'aux nœuds $9 + 2^0 = 10$ et $9 + 2^1 = 11$ car les indexes suivants $9 + 2^2 = 13$ et $9 + 2^3 = 17$ ne donnent pas des nœuds existant dans l'anneau. [2] Notons que le lien vers le nœud $i + 1$ successeur de i est intégré parmi les cordes par la corde d'ordre 0 de chaque nœud i, ce qui rend la structure régulière. Toutefois, ce lien a un rôle privilégié dans le maintien et l'utilisation de l'anneau dont il faudra tenir compte par la suite.

2. Dans une table de hachage répartie où les requêtes peuvent être initiées sur n'importe quel nœud de l'anneau, il serait possible de calculer les indexes modulo N et ainsi obtenir un anneau sans réels début et fin. Pour le projet, nous avons choisis d'introduire une façade et un anneau avec un premier et un dernier nœud.

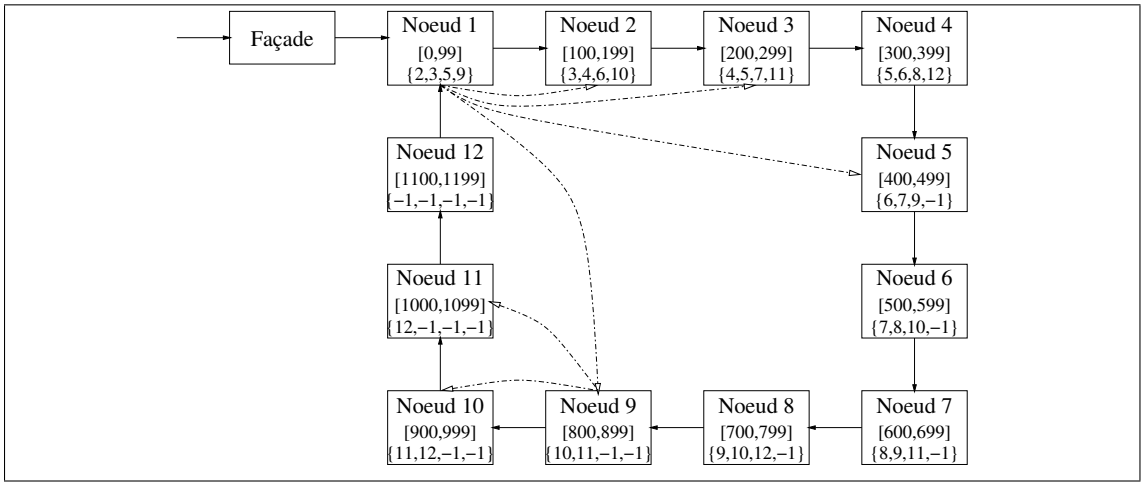

FIGURE 3 – Table de hachage répartie accélérée ; la liste des indexes dans chaque nœud donne les cordes issues du nœud concerné (une valeur -1 indique qu'il n'y a pas de corde pour le degré concerné). Les liens ainsi créés entre les nœuds ne sont indiqués explicitement par les flèches que pour les nœuds 1 et 9 (les autres ne le sont pas pour préserver la lisibilité de la figure).

Les cordes accélèrent l'accès aux données lointaines dans l'anneau de la manière suivante. À chaque corde j dans un nœud i est associé la valeur de hachage minimale $v_{min,j}$ des clés détenues dans le nœud $i+2^j$. Cette valeur est le minimum des valeurs détenues dans le nœud au bout de la corde mais aussi, par construction de l'anneau, de toutes les valeurs détenues par les nœuds du reste de l'anneau à partir de l'index $i+2^j$. Un nœud i peut donc faire passer directement une requête pour une valeur de hachage v qui ne lui est pas destinée à sa corde j telle que $v_{min,j} \leq v < v_{min,j+1}$ (ou simplement $v_{min,j} \leq v$ si la corde j n'existe pas).

Par exemple, si on recherche la donnée associée à la clé dont la valeur de hachage est 950 dans la table de la figure 3, le nœud 1 trouve que sa corde $j = 3$ mène au nœud 9 dont la valeur de hachage minimale est 800. Le nœud 9 procède de la même manière pour se rendre compte que sa corde $j = 0$ mène au nœud 10 dont la valeur de hachage minimale est $900 \leq 950$ alors que sa corde $j = 1$ mène au nœud 11 dont la valeur de hachage minimal est $1000 > 950$. Le nœud 10 peut trouver (ou non, mais alors c'est que la donnée cherchée n'est pas dans la table) la donnée dans sa table de hachage locale (qui contient toutes les valeurs de hachage entre 900 et 999).

1.4 Table de hachage répartie dynamique

Dans les explications précédentes, la table de hachage a été implicitement supposée statique, c'est à dire que tous le nœuds sont créés dès le départ et demeurent dans la même configuration jusqu'à la fin de l'exécution. En pratique, les tables de hachage réparties sont plutôt dynamiques, c'est à dire que le nombre de nœuds présents dans la table varie dans le temps en fonction du nombre total de couples clé/valeur devant être mémorisés. L'idée est donc de démarrer avec un nombre minimal de nœuds puis de les scinder lorsque le nombre de couples insérés dans la table augmente trop, et symétriquement de les fusionner lorsque le nombre de couples diminue.

Dans le contexte réparti à moyenne ou grande échelle, il n'est pas sérieusement envisageable d'optimiser la répartition des couples sur l'ensemble des nœuds. Il serait effectivement très coûteux de réorganiser les intervalles de valeurs de hachage et de transférer massivement des données entre tous les nœuds. Il est plus judicieux d'adopter un critère local et de faire les ajustements entre nœuds adjacents, même si cela ne conduit pas à un nombre de nœuds ni une répartition des données optimaux. Un critère pour modifier le nombre de nœuds ou équilibrer la charge entre ces derniers peut alors être celui du facteur de charge de chaque nœud, c'est à dire le nombre maximal d'entrées dans chaque table de hachage locale. Pour appliquer ce critère à la fois pour scinder et pour fusionner, deux stratégies sont possibles :

1. Une stratégie à **gros grain** qui se limite à scinder ou fusionner les nœuds :

   Gros grain/Scinder : la charge de chaque nœud est examiné et si elle dépasse par exemple deux fois le facteur de charge admis, le nœud est scindé en deux en coupant son intervalle

à la valeur de hachage répartissant en deux sous-ensembles à peu près égaux les couples présents dans la table locale, les données de celui des valeurs de hachage les plus élevées étant transférées dans le nouveau nœud inséré à la suite du premier dans l'anneau.

Gros grain/Fusionner : les charges de deux nœuds successifs i et $i+1$ sont examinées. Si le total du nombre de couples dans les tables locales à ces deux nœuds est inférieur au facteur de charge admis pour un seul nœud alors les deux nœuds sont fusionnés en élargissant l'intervalle de valeurs de hachage du premier à celui du second puis les données du second transférées au premier avant de supprimer le second de l'anneau.

2. Une stratégie à **grain fin** qui d'abord équilibre les charges entre nœuds adjacents et ne scinde ou fusionne qu'en dernier recours :

   Grain fin/Scinder : les charges de deux nœuds successifs sont examinées. Si la charge de l'un ou l'autre des nœuds est supérieure au facteur de charge admis mais que la charge combinée des deux nœuds est inférieure à deux fois le facteur de charge admis et que les écarts entre les deux nœuds le justifie, alors une partie des données du plus chargé est transférée à l'autre pour équilibrer la charge. Par contre, si la charge totale des deux nœuds est supérieure à trois fois le facteur de charge admis, alors on insère un nouveau nœud intermédiaire entre les deux et des données lui sont transférées depuis les deux nœuds préexistants de manière à équilibrer la charge entre les trois nœuds.

   Grain fin/Fusionner : les charges de trois nœuds successifs sont examinées. Si la charge combinée des trois nœuds se situe entre deux et trois fois le facteur de charge admis et que les écarts entre les trois nœuds le justifie, des données sont échangées pour équilibrer la charge entre ces trois nœuds. Si la charge combinée est inférieure à deux fois le facteur de charge admis, alors le nœud intermédiaire est supprimé et ses données sont transférées dans les deux nœuds restants de manière à équilibrer la charge.

Les résultats de toutes ces stratégies en termes de nombre de nœuds dans l'anneau et de charge dans chacun des nœuds après leur application vont différer en fonction de l'ordre dans lequel les nœuds successifs dans l'anneau sont examinés. L'implantation la plus simple consiste à balayer régulièrement les nœuds du premier au dernier, une passe pour fusionner puis une pour scinder. La bonne modulation globale du nombre de nœuds et de la charge entre les nœuds est alors obtenue par l'exécution répétée de ces passes successives dans la durée.

Il est également à noter que la modification dynamique du nombre de nœuds impacte directement les cordes introduites à la section précédente qui doivent être recalculées ou réparées pour refléter les ajouts et retraits de nœuds. Ainsi, lorsque les cordes sont utilisées, la manière dont les données sont échangées entre les nœuds lorsqu'on scinde ou fusionne peut être contrainte pour éviter d'invalider les cordes avant qu'on ne puisse les recalculer ou les réparer, en particulier si on ne bloque pas entièrement la table de hachage répartie pendant sa reconfiguration dynamique.

2 Réalisation

Comme tout projet un tant soit peu complexe, il faut aborder celui-ci de manière systématique. De plus, les solutions aux différents problèmes qui vont se présenter vont être enseignées graduellement au cours du semestre. Il sera donc nécessaire de les intégrer tout aussi graduellement dans le code du projet. Le projet va donc se développer en plusieurs phases qui vont faire passer l'exécution des requêtes d'une approche très classique techniquement similaire à une exécution séquentielle dans un premier temps puis progressivement à une exécution asynchrone, parallèle et répartie, en ajoutant de nouvelles fonctionnalités comme la dynamicité.

L'idée est de vous permettre d'appréhender ces nouveaux types de programmation et ce qui les distingue de la programmation séquentielle qui vous a été enseignée depuis le début de vos études d'informatique. Au fil du déroulement du projet, des notions nouvelles d'architecture logicielle, de structuration et de réutilisation du code ainsi que les méthodes de tests pour ce type d'applications seront aussi abordées. Enfin, nous allons introduire des notions de performance et de passage à l'échelle qui vous permettront de mieux comprendre les techniques et les enjeux des applications à grande échelle comme on en trouve de plus en plus sur Internet aujourd'hui.

Le projet est planifié sur le semestre de manière à rendre votre travail efficace tout en capitalisant sur le travail précédent pour faire évoluer la solution au fur et à mesure de l'intégration

```
public interface ContentKeyI extends Serializable  { }

public interface ContentDataI extends Serializable
{
  default Serializable getValue(String attributeName)
  {
    throw new IllegalArgumentException(
                    "unknown attribute for Person: " + attributeName);
  }
}
```

FIGURE 4 – Interfaces devant être implantées par les clés et les valeurs insérées dans la table de hachage.

des nouvelles possibilités. Ainsi, un premier découpage grossier du projet passe par les étapes suivantes :

1. Premier développement en Java classique mais utilisant la technique des points de connexion explicites (« *end points* ») pour lier les objets représentant les nœuds, la façade et les clients, technique qui sera pérenne ensuite au fil du projet bien que nécessitant des adaptations.
2. Passage des entités représentant les nœuds, la façade et les clients d'objets Java à des composants BCM4Java avec leurs points de connexion particuliers.
3. Introduction du parallélisme et de la concurrence en s'appuyant d'une part sur une technique de programmation asynchrone et d'autre part sur les *pools* de *threads* de Java.
4. Introduction de la dynamicité dans la table de hachage répartie puis introduction de l'exécution répartie s'appuyant sur la capacité de BCM4Java à exécuter les composants dans différentes machines virtuelles Java (déployées sur un seul ordinateur mais qui pourraient l'être sur plusieurs ordinateurs).

2.1 Première étape : développement en Java

2.1.1 Services et façade de la table de hachage

La table de hachage répartie mémorise des couples clé/valeur sous la forme d'objets Java qui implantent (au sens de la clause « `implements` » de Java) les interfaces `ContentKeyI` pour les clés et `ContentDataI` pour les données, interfaces dont les déclarations apparaissent dans la figure 4. [3] `ContentKeyI` étant vide, c'est principalement une interface marqueur qui permet de typer les variables, paramètres et valeurs de retour de manière générique. Toutefois, elle étend l'interface `Serializable` de Java en anticipation de leur utilisation en BCM4Java et en programmation répartie.

L'interface `ContentDataI` introduit une signature `getValue` qui nécessite une explication supplémentaire. Cette signature prévoit une implantation dans tous les objets représentant des données à insérer dans la table de hachage. La méthode ainsi implantée permettra d'écrire des fonctions de transformation et de réduction dans le modèle *map/reduce* sur des données hétérogènes sans avoir à connaître précisément la classe de ces données. À cette fin, elle propose un protocole simple et unique pour accéder aux différentes valeurs apparaissant dans les données. Par exemple, si les données à mémoriser concernent des personnes et que ces dernières ont par exemple un nom et un âge, la méthode `getValue` prévoit que l'objet personne implantant l'interface `ContentDataI` :

1. définira des noms d'attributs, ici des chaînes de caractères, par exemple `"NOM"` et `"AGE"` ;
2. fournira une implantation de `getValue` qui, appelée avec `"NOM"` en paramètre, retournera le nom de la personne alors que si elle est appelée avec `"AGE"`, elle retournera l'âge de la personne.

Pour cette première étape, les services offerts par la table de hachage sont définis par l'interface `DHTServicesI` apparaissant dans la figure 5. Les trois premières signatures correspondent

3. Toutes ces interfaces vous seront fournies en source Java à utiliser directement dans votre projet. Ces sources sont entièrement commentés, en complément des explications données ici.

```
public interface DHTServicesI
{
  public ContentDataI get(ContentKeyI key) throws Exception;
  public ContentDataI put(ContentKeyI key, ContentDataI value) throws Exception;
  public ContentDataI remove(ContentKeyI key) throws Exception;
  public <R extends Serializable,A extends Serializable> A mapReduce(
    SelectorI selector,
    ProcessorI<R> processor,
    ReductorI<A,R> reductor,
    CombinatorI<A> combinator,
    A identityAcc
    ) throws Exception;
}
```

FIGURE 5 – Interface Java déclarant les services offerts par la table de hachage répartie à ses clients.

aux services classiques des tables de hachage, tels qu'ils apparaissent dans l'interface `Map` de la bibliothèque standard de Java. La signature `get` prend une clé en paramètre et retourne la donnée correspondante dans la table ou `null` si la table ne contient pas de donnée attachée à la clé fournie. La signature `put` prend une clé et une donnée qu'elle insére dans la table ; elle retourne la valeur précédente associée à la clé fournie dans la table s'il y en a une ou `null` si c'est la première insertion pour la clé fournie. La signature `remove` prend une clé et retire le couple clé/donnée correspondant de la table ; elle retourne la donnée s'il y a une donnée associée à la clé fournie dans la table ou `null` s'il n'y a pas de telle donnée dans la table.

La signature `mapReduce` fournie le cadre d'implantation du modèle *map/reduce* pour la table de hachage répartie. Elle est définie en spécialisant les éléments utilisés par l'implantation du modèle *map/reduce* par le *framework* des *streams* de Java. La signature `mapReduce` prend cinq paramètres :

`selector :` un prédicat (au sens des interfaces fonctionnelles de Java) qui est utilisé en premier par la phase *map* pour filtrer les données de la table de hachage répartie qui vont être traitées par la suite.

`processor :` une fonction unaire, utilisée après le filtrage dans la phase *map*, prenant une donnée et retournant le résultat à insérer dans la liste des résultats de la phase *map*.

`reductor :` une fonction binaire de réduction permettant de réduire progressivement les résultats de la phase *map*.

`combinator :` une fonction binaire de combinaison permettant de calculer un résultat de réduction à partir de deux résultats de réduction de sous-listes issues de la phase *map*.

`identityAcc :` la valeur neutre pour la fonction de réduction.

Les définitions des types des prédicat et fonctions de cette signature apparaissent à la figure 6. Ce sont des interfaces fonctionnelles qui étendent et spécialisent les interfaces fonctionnelles utilisées par les signatures de l'interface `Stream` de la bibliothèque standard de Java. Elles étendent aussi l'interface standard `Serializable`, également en anticipation de leur utilisation en BCM4Java en programmation répartie.

Pour ce qui concerne la première étape du projet, vous devrez programmer la façade de la table de hachage sous la forme d'une classe Java classique implantant (au sens de `implements`) l'interface `DHTServicesI`. La mise en œuvre des méthodes émanant de cette interface s'appuiera sur les nœuds de la table de hachage qui vont maintenant être introduits. Vous devrez également implanter des clients pour cette table de hachage répartie servant à tester votre implantation.

2.1.2 Nœuds de la table de hachage

Les nœuds de la table de hachage répartie vont fournir les moyens internes pour implanter les services introduits précédemment. Pour cette première étape, ils seront mis en œuvre par des objets Java implantant les interfaces `ContentAccessSyncI` et `MapReduceSyncI`. Deux interfaces

```
@FunctionalInterface
public interface SelectorI extends Predicate<ContentDataI>, Serializable { }

@FunctionalInterface
public interface ProcessorI<R> extends Function<ContentDataI,R>, Serializable
{ }

@FunctionalInterface
public interface ReductorI<A extends Serializable,R>
extends BiFunction<A,R,A>, Serializable
{ }

@FunctionalInterface
public interface CombinatorI<A extends Serializable>
extends BiFunction<A,A,A>, Serializable
{ }
```

FIGURE 6 – Interfaces fonctionnelles Java utilisées dans l'implantation du modèle de calcul *map/reduce* par la table de hachage répartie.

distinctes sont définies pour permettre potentiellement de n'offrir que les services de table de hachage (`ContentAccessSyncI`) sans les services du modèle *map/reduce* (`MapReduceSyncI`).

Comme leurs noms l'indiquent, ces deux interfaces visent à implanter les services de la table de hachage répartie en utilisant une technique d'appels synchrones. Un appel de méthode synchrone est simplement un appel où l'exécution de l'appelant est suspendue dans l'attente du résultat jusqu'à ce que l'appelé lui retourne ledit résultat. Cela correspond à la sémantique habituelle des appels de méthodes en Java séquentiel qui sera utilisé dans cette première étape mais également dans la seconde étape en s'appuyant sur les appels synchrones entre composants.

Bien que cette première implantation se borne à de la programmation séquentielle Java classique, les signatures anticipent sur le passage à une exécution parallèle puis répartie dans les étapes suivantes du projet. Ainsi, les cinq signatures `getSync`, `putSync`, `removeSync`, `mapSync` et `reduceSync` prennent toutes un identifiant unique du calcul global qu'elles font, ici le paramètre `computationURI` de type `String`, c'est à dire un identifiant d'une requête reçue par la façade en attente de résultat. L'idée est que lorsque les appels seront exécutés en parallèle, ce paramètre permettra aux différents nœuds de savoir sur quel calcul les méthodes travaillent lors d'un appel donné. De plus, dans le cas des signatures `mapSync` et `reduceSync`, elles prévoient la possibilité de lancer en parallèle les phases *map* et *reduce*. Le paramètre `computationURI` permettra alors aussi de mémoriser des résultats intermédiaires, comme les résultats de la phase *map* à reprendre par la phase *reduce*.

Les signatures `clearComputation` et `clearMapReduceComputation` sont prévues pour être appelées après que la façade aura reçu le résultat du calcul correspondant (`getSync`, `putSync`, `removeSync` pour la première, `mapSync` et `reduceSync` pour la seconde) pour nettoyer dans tous les nœuds de l'anneau toutes les données transitoires restantes que ce calcul aurait laissées.

2.1.3 Connexions entre les objets de la table de hachage

La première étape va vous permettre de prendre en main le projet en deux temps. Il s'agira essentiellement de programmer trois classes, l'une pour la façade, une autre pour les nœuds et au moins une comme client de test de la table de hachage. Dans un premier temps, qui ne devrait pas prendre plus d'une semaine, vous programmerez ces trois classes en Java séquentiel classique. Cela veut dire que lorsque la classe de la façade se réfère aux nœuds de l'anneau, elle utilisera des variables Java typées par la classe des nœuds, comme vous le faites jusqu'ici en programmant en Java. Cette partie du travail doit vous permettre de développer l'essentiel des algorithmes des différents éléments qui serviront pendant toute la première moitié du projet puis qui seront progressivement modifiés à partir de la mi-semestre pour introduire le parallélisme et la répartition.

Dans un deuxième temps de cette première étape, pour préparer la suite du projet, les références

```
public interface ContentAccessSyncI
{
  public ContentDataI getSync(String computationURI, ContentKeyI key)
  throws Exception;

  public ContentDataI putSync(
    String computationURI,
    ContentKeyI key,
    ContentDataI value
    ) throws Exception;

  public ContentDataI removeSync(String computationURI, ContentKeyI key)
  throws Exception;

  public void   clearComputation(String computationURI)
  throws Exception;
}

public interface  MapReduceSyncI
{
  public <R extends Serializable> void mapSync(
    String computationURI,
    SelectorI selector,
    ProcessorI<R> processor
    ) throws Exception;

  public <A extends Serializable,R> A  reduceSync(
    String computationURI,
    ReductorI<A,R> reductor,
    CombinatorI<A> combinator,
    A initialAcc
    ) throws Exception;

  public void   clearMapReduceComputation(String computationURI)
  throws Exception;
}
```

FIGURE 7 – Interfaces Java déclarant les services offerts par les nœuds de la table de hachage répartie.

Java entre entités de la table de hachage répartie vont devoir être modifiées pour utiliser les points de connexion (*end points*) fournis par BCM4Java. L'utilisation de ces points de connexion vise deux objectifs :

1. Représenter plus explicitement les connexions entre les entités façade et nœuds ainsi qu'entre les nœuds, ce qui permettra ensuite de passer à des connexions entre composants répartis qui peuvent être déployés dans différentes machines virtuelles Java et différents ordinateurs, contexte où de simples références Java ne peuvent plus être utilisées.
2. S'abstraire de la technologie de connexion effectivement utilisée. Aujourd'hui, de nombreuses technologies sont utilisées en pratique dans les architectures logicielles réparties : Java RMI, liaisons de composants CORBA, interfaces REST, liens entre services dans les architectures fondées sur les services, etc. Dans le cadre du projet, vous allez utiliser des références Java classiques et des liaisons BCM4Java avec ports et connecteurs, elles-mêmes s'appuyant sur Java RMI. En utilisant les points de connexion, votre code va pouvoir s'abstraire du choix de technologie. Vous constaterez en fin de projet qu'après avoir passé le code des algorithmes à ces points de connexion, il ne sera plus nécessaire de le modifier lorsque vous changerez la technologie de connexion. Les modifications nécessaires se situeront dans les déclarations de variables et dans les initialisations, mais le code des algorithmes en tant que tel ne sera pas touché.

```
public interface ContentNodeBaseCompositeEndPointI<
                           CAI  extends ContentAccessSyncI,
                           MRI extends MapReduceSyncI>
extends CompositeEndPointI
{
  public EndPointI<CAI> getContentAccessEndpoint();
  public EndPointI<MRI> getMapReduceEndpoint();

  @Override
  public ContentNodeBaseCompositeEndPointI<CAI,MRI> copyWithSharable();
}
```

FIGURE 8 – Interface Java permettant d'implanter le points de connexion composites entre façade et nœuds et entre nœuds lors de la première étape du projet.

Pour l'essentiel, l'introduction des points de connexion va utiliser les interfaces et classes fournies par BCM4Java. En complément de ces dernières, l'interface `ContentNodeBaseCompositeEndPointI` présentée dans la figure 8 qui introduit une spécialisation de l'interface `CompositeEndPointI` fournie par BCM4Java. Un point de connexion composite est simplement le regroupement de plusieurs points de connexion simples en un seul point de connexion multiple. Cela simplifie la construction d'architectures logicielles où les mêmes entités client et serveur sont connectées à travers plusieurs interfaces et donc plusieurs connexions simples distinctes. Dans le cadre du projet, cela concerne les connexions entre la façade et le premier nœud dans l'anneau et les connexions entre les nœuds dans l'anneau.

La spécialisation offerte par `ContentNodeBaseCompositeEndPointI` paramétrise par le polymorphisme paramétrique les types des interfaces via lesquelles les connexions simples sont faites, c'est à dire celle déclarant les services d'accès aux données de la table et l'autre pour le modèle *map/reduce*. Pour la première étape, les interfaces concernées sont celles de la figure 7, mais dans les étapes suivantes ces interfaces seront étendues par de nouvelles interfaces ajoutant d'autres signatures. Il sera donc intéressant d'avoir la possibilité de spécialiser encore plus les interfaces fournies en paramètres génériques de cette interface.

Cette interface introduit aussi deux signatures, `getContentAccessEndpoint` et `getMapReduceEndpoint`, qui vont simplifier l'accès aux points de connexions simples regroupés dans le point de connexion composite tout en spécialisant les types de résultat par rapport aux signatures de l'interface `CompositeEndPointI`. La signature `copyWithSharable` n'est en effet ajoutée que pour spécialiser le type de son résultat, ce qui évitera ensuite de devoir transtyper les résultats de copie.

Enfin, une classe `POJOContentNodeCompositeEndPoint` vous sera fournie. Cette classe devra être utilisée pour mettre en place les points de connexion dans la version avec Java séquentiel de cette première étape. L'autre point de connexion nécessaire pour cette première étape (en fait, ceux entre les clients et la façade) est un point de connexion simple pouvant utiliser directement les interfaces et classes fournies par BCM4Java.

2.2 Deuxième étape : composants BCM4Java

À venir.

2.3 Troisième étape : parallélisme et concurrence

À venir.

2.4 Quatrième étape : dynamicité et répartition

À venir.

3 Déroulement du projet et résultats attendu

Le déroulement du projet s'étale sur quatre étapes échelonnées autour des quatre évaluations successives, audits et soutenances.

3.1 Résumé du déroulement du projet

Dans ses grandes lignes, le projet va se dérouler sur tout le semestre de la manière suivante :

1. **Première étape** (3 semaines)

   Semaine 1-2 (20-21, 27-28/01) : Première implantation en Java « pur » avec les interfaces `DHTServicesI`, `ContentAccessSyncI` et `MapReduceSyncI` ; test à un nœud dans la table de hachage « répartie » puis augmentation à plusieurs nœuds.

   Semaine 3 (3-4/02)) : Modification de la première implantation pour connecter les objets Java à travers des points de connexion (« *end points* ») définis pour les objets Java ; test à au moins deux nœuds dans la table de hachage « répartie »

2. **Deuxième étape** (3 semaines)

   Semaine 4 (10-11/02) : (en parallèle avec l'audit 1) Deuxième implantation avec des composants BCM4Java utilisant des points de connexion pour ces derniers : principales nouvelles interfaces (de composants) à implanter : `DHTServicesCI`, `ContentAccessSyncCI` et `MapReduceSyncCI`.

   Semaine 5 (17-18/02) : Poursuite de la deuxième implantation ; scénarios de test pour la soutenance de mi-semestre.

   Semaine des congés d'hiver (24-25/02) : Fin de la deuxième implantation et rendu de code pour l'évaluation de mi-semestre.

   Semaine des premiers examens répartis (3-4/03) : soutenances de mi-semestre.

3. **Troisième étape** (4 semaines)

   Semaine 6 (10-11/03) : Passage à l'exécution asynchrone des requêtes en style passage à la continuation exploitant mieux les ressources des nœuds grâce au parallélisme qu'il permet d'introduire entre les composants. Principales nouvelles interfaces à implanter : `ContentAccessCI`, `ResultReceptionCI`, `MapReduceCI`, `MapReduceResultReceptionCI`, avec les points de connexion correspondants.

   Semaine 7 (17-18/03) : Introduction de la concurrence et du parallélisme interne aux composants à l'aide des outils fournis par Java.

4. **Quatrième étape** (4 semaines)

   Semaine 8 (24-25/03) : Passage à la gestion dynamique du nombre de nœuds dans la table de hachage répartie ; nouvelle interface de composant à intégrer : `DHTManagementCI`, avec les points de connexion correspondants.

   Semaine 9 (31/03-1/04) : (en parallèle avec l'audit 2) Introduction des cordes dans la table de hachage répartie.

   Semaine 10 (7-8/04) : Passage à une exécution répartie sur plusieurs machines virtuelles Java (sur un même ordinateur).

   Semaines des congés de printemps (12-27/04) : Poursuite du passage à l'exécution répartie et montage de scénarios de test ; rendu de code pour l'évaluation finale.

   Semaine des deuxièmes examens répartis (28-29/04) : soutenances finales.

Ce déroulement est proposé à titre indicatif. Il permet d'assurer les objectifs attendus des évaluations successives. Toutefois, il est fortement suggéré de ne pas hésiter à avancer plus vite si vous le pouvez ; cela réduira votre charge de travail plus tard dans le semestre, lorsque vous serez plus sollicités par vos autres unités d'enseignement. Par exemple, si vous avez terminé ce qui est demandé pour l'audit 1 avant la séance de TD/TME de la semaine 4, n'hésitez pas à commencer le travail attendu pour les semaines suivantes.

3.2 Audit 1 : version Java avec points de connexion

La première étape va être centrée sur l'implantation des algorithmes de la table de hachage, ce qui demandera une prise en main du projet suffisante suffisante pour :

- implanter les trois types d'entités prévues (façade, nœud, client) sous la forme d'objets Java standards mais interconnectés par des points de connexion explicites ;
- d'exécuter la création d'une table de hachage répartie formée d'une façade et d'au moins deux nœuds ;
- de monter un scénario de test par au moins un client soumettant à la façade des requêtes peuplant la table de hachage, vérifiant le bon fonctionnement des méthodes `get`, `put` et `remove` (de la façade) et aussi un calcul *map/reduce* via la méthode `mapReduce`.

Pour cet audit, le principal critère d'évaluation sera le degré de réalisation des développements décrits dans la première partie du projet, leur conformité aux exigences du cahier des charges et l'exécution correcte d'un scénario de test de complexité suffisante pour démontrer le bon fonctionnement de l'ensemble.

3.3 Soutenance de mi-semestre : BCM4Java en exécution synchrone

À venir.

3.4 Audit 2 : exécution asynchrone, parallélisme et concurrence

À venir.

3.5 Soutenance finale : table de hachage dynamique et répartition

À venir.

4 Modalités générales de réalisation et calendrier des évaluations

- Le projet se fait **obligatoirement** en **équipe de deux personnes**. Tous les fichiers sources du projet doivent comporter les noms (balise `authors`) de tous les auteurs en Javadoc. Lors de sa formation, chaque équipe devra se donner un nom et me le transmettre avec les noms des personnes la formant au plus tard le **24 janvier 2025**.
- Le projet doit être réalisé avec **Java SE 8**. Attention, peu importe le système d'exploitation sur lequel vous travaillez, il faudra que votre projet s'exécute correctement sous Eclipse et sous **Mac Os X/Unix** (que j'utilise et sur lequel je devrai pouvoir refaire s'exécuter tous vos tests).
- L'évaluation comportera quatre épreuves : deux audits intermédiaires, une soutenance à mi-semestre et une finale, ces dernières accompagnées d'un rendu de code et de documentation. Ces épreuves se dérouleront selon les modalités suivantes :
  1. Les deux audits intermédiaires dureront 5 à 10 minutes au maximum (par équipe) et se dérouleront lors des séances de TME. Le premier audit se tiendra pendant la séance 4 (**10 février 2025**) et il portera sur votre avancement de l'étape 1. Le second audit se déroulera lors de la séance 9 (**31 mars 2025**) et il portera sur votre avancement de l'étape 3. Ils compteront chacun pour 5% de la note finale de l'UE.
  2. La **soutenance à mi-parcours** d'une durée de 20 minutes portera sur l'atteinte des objectifs de l'ensemble des première et deuxième étapes du projet. Elle se tiendra pendant la semaine des premiers examens répartis du **3 au 7 mars 2025** (très probablement les lundi 3/03 et mardi 4/03)) selon un ordre de passage et des créneaux qui seront annoncés au préalable. Elle comptera pour 35% de la note finale de l'UE. Elle comportera une discussion des réalisations pendant une quinzaine de minutes (devant l'écran sous Eclipse) et une courte démonstration de cinq minutes. Les rendus à mi-parcours

se feront le **dimanche 2 mars 2025 à minuit** au plus tard (des pénalités de retard seront appliquées).

3. La **soutenance finale** d'une durée de 20 minutes portera sur l'ensemble du projet mais avec un accent sur les troisième et quatrième étapes. Elle aura lieu dans la semaine des seconds examens répartis, les lundi 28/04/2025 et mardi 29/04/2025, selon un ordre de passage qui sera annoncé au préalable. Elle comptera pour 55% de la note finale de l'UE. Elle comportera une discussion d'une quinzaine de minutes sur les réalisations suivie d'une démonstration. Les rendus finaux se feront le **dimanche 27 avril 2025 à minuit** au plus tard (des pénalités de retard seront appliquées).

— Lors des soutenances, les points suivants seront évalués :
  — le respect du cahier des charges et la qualité de votre programmation ;
  — l'exécution correcte de tests unitaires (JUnit pour les classes et les objets Java, scénario de tests pour les composants) ;
  — l'exécution correcte de tests d'intégration mettant en œuvre des composants de tous les types ;
  — la qualité et l'exécution correcte des scénarios de tests de performance, conçus pour mettre à l'épreuve les choix d'implantation versus la montée en charge ;
  — la qualité de votre code (votre code doit être commenté, être lisible – choix pertinents des identifiants, ... – et correctement présenté – indentation, ... – etc.) ;
  — la qualité de votre plan d'expérimentation et tests de performance (couverture de différents cas, isolation des effets pour identifier leurs impacts sur la performance globale, etc.) ;
  — la qualité de la documentation (vos rendus pour la soutenance finale devront inclure une *documentation Javadoc* des différents paquetages et classes de votre projet générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que votre répertoire `src`).
— Bien que les audits et les soutenances se fassent par équipe, l'évaluation reste à chaque fois **individuelle**. Lors des audits et des soutenances, *chaque membre* devra se montrer capable d'expliquer différentes parties du projet, et selon la qualité de ses explications et de ses réponses, sa note peut être supérieure, égale ou inférieure à celle des autres membres de son équipe.
— Lors des soutenances, **tout retard** non justifié d'un des membres de l'équipe de plus d'un tiers de la durée de la soutenance (7 minutes) entraînera une **absence** et une note de 0 attribuée aux membre(s) retardataire(s) pour l'ensemble de l'épreuve concernée (y compris le rendu préalable). Si une partie des membres d'une équipe arrive à l'heure ou avec un retard de moins d'un tiers de la durée de la soutenance, les présents passeront l'épreuve seul.
— Le rendu à mi-parcours et le rendu final se font sous la forme d'une archive `tgz` si vous travaillez sous Unix ou `zip` si vous travaillez sous Windows ***à l'exclusion de toute autre format*** (n'inclure *que les sources*, c'est à dire uniquement le répertoire `src` de votre projet *sans les binaires* et les jars pour réduire la taille du fichier) envoyé à `Jacques.Malenfant@lip6.fr` comme attachement fait proprement avec votre programme de gestion de courrier préféré ou encore par téléchargement avec un lien envoyé par courrier électronique (en lieu et place du fichier). Donnez pour nom au répertoire de projet et à votre archive celui de votre équipe (ex. : équipe LionDeBelfort, répertoire de projet `LionDeBelfort` et archive `LionDeBelfort.tgz`).
— **Tout manquement à ces règles élémentaires entraînera une pénalité dans la note des épreuves concernées !**
— Pour la **deuxième session**, si elle s'avérait nécessaire, elle consiste à poursuivre le développement du projet pour résoudre ses insuffisances constatées à la première session et donnera lieu à un rendu du code et de documentation puis à une soutenance dont les dates seront déterminées en fonction du calendrier du master. Les critères d'évaluation sont les mêmes que lors de la soutenance finale, mais modulés selon qu'une seule personne ou deux de la même équipe doivent passer la seconde session. Sur demande faite en avance (dès les notes de première session connues), une personne peut choisir de passer la seconde session seule même si d'autres membres de l'équipe doivent également le passer ; en cas de désaccord entre les membres de l'équipe, la demande de passage en solitaire prévaudra.

Références

[1] Chord (peer-to-peer). Wikipedia entry. last consulted December 19, 2024.

