Table de hachage répartie et map/reduce en BCM4Java

Cahier des charges du projet CPS 2025

Jacques Malenfant
professeur des universités

version 4.0 du 21/03/2025

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.

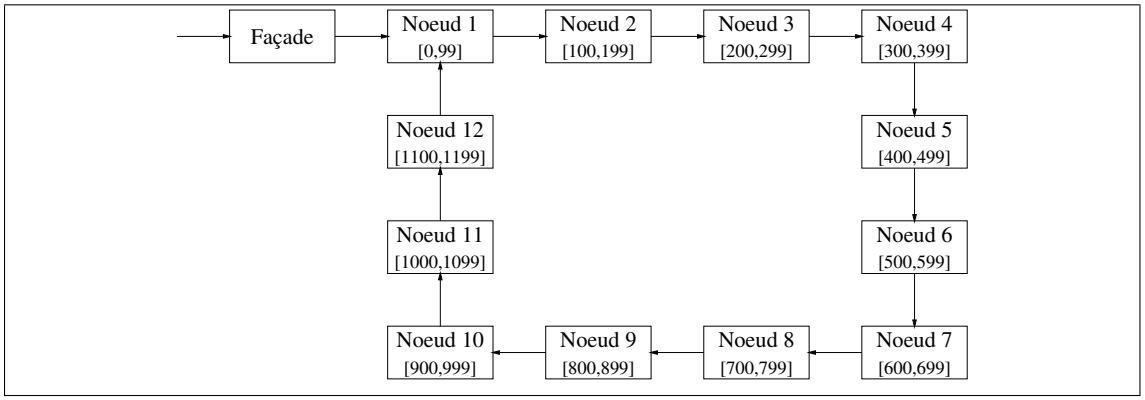

FIGURE 1 – Table de hachage répartie ; les couples présents dans chaque nœud donnent l'intervalle des valeurs de hachage des clés qui sont conservées dans la table de hachage locale du nœud concerné.

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.

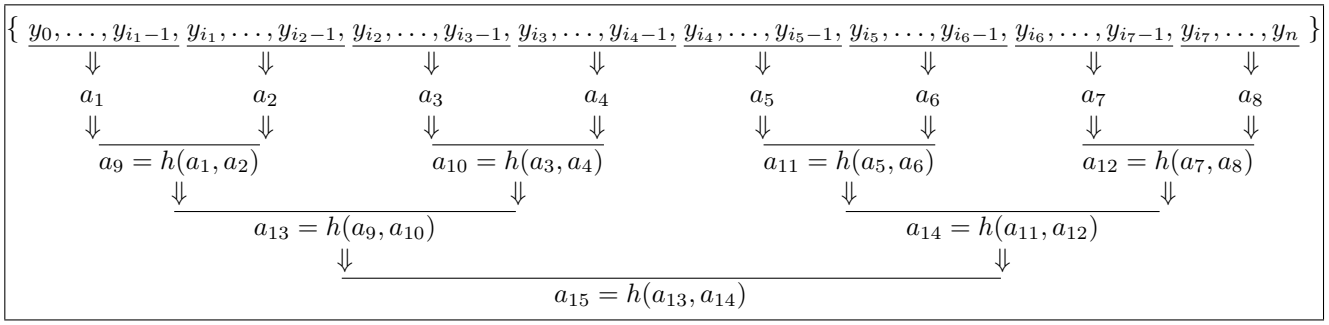

FIGURE 2 – Parallélisation de la réduction, avec utilisation des fonctions de réduction g et de combinaison h. Dans cet exemple, la liste initiale est subdivisée en 8 sous-listes traitées par une réduction par g pour donner les résultats intermédiaires $a_j = reduce(g, a_0, \{y_{i_j}, y_{i_{j+1}-1}\})$, $j = 0, \ldots, 8$ 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 sélectionné entre 1 et 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

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.

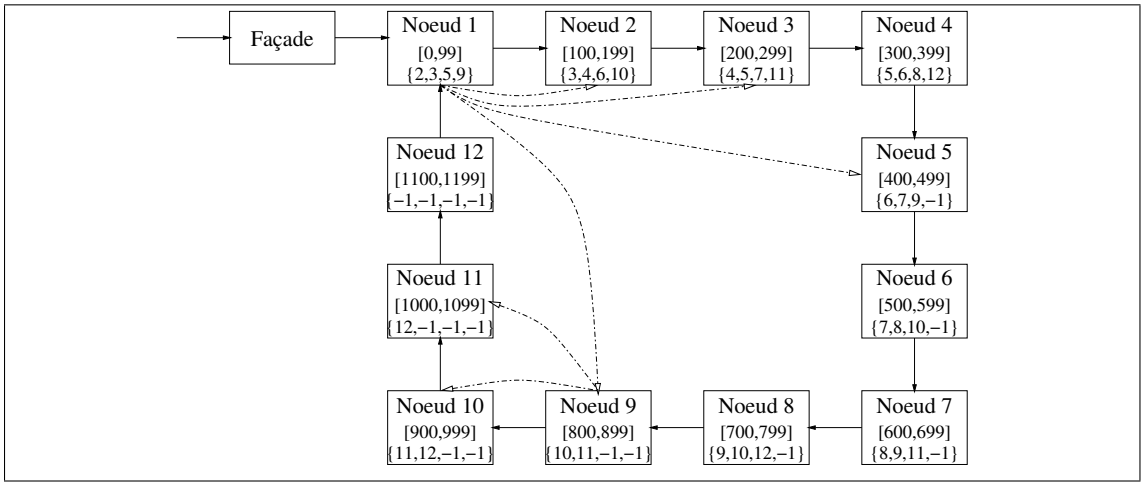

FIGURE 3 – Table de hachage répartie accélérée ; la liste des indexes dans chaque nœud donne les cordes issues du nœud concerné (une valeur -1 indique qu'il n'y a pas de corde pour le degré concerné). Les liens ainsi créés entre les nœuds ne sont indiqués explicitement par les flèches que pour les nœuds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

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.

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.

```
public interface ContentKeyI extends Serializable  { }

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

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

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 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 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`).

```
@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.

```
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.

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

```
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.

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 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

```
public interface DHTServicesCI
extends OfferedCI, RequiredCI, DHTServicesI
{
  @Override
  public ContentDataI get(ContentKeyI key) throws Exception;

  @Override
  public ContentDataI put(ContentKeyI key, ContentDataI value) throws Exception;

  @Override
  public ContentDataI remove(ContentKeyI key) throws Exception;

  @Override
  public <R extends Serializable, A extends Serializable> A mapReduce(
    SelectorI selector, ProcessorI<R> processor,
    ReductorI<A, R> reductor, CombinatorI<A> combinator, A initialAcc
    ) throws Exception;
}
```

FIGURE 9 – Interfaces de composants déclarant les services offerts par la façade de la table de hachage répartie.

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.

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.

```
public interface ContentAccessSyncCI
extends OfferedCI, RequiredCI, 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 MapReduceSyncCI
extends OfferedCI, RequiredCI, 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 10 – Interfaces de composants déclarant les services offerts par les nœuds de la table de hachage répartie.

2.2 Deuxième étape : composants BCM4Java

La deuxième étape du projet se consacre à la transformation des entités formant la table de hachage, façade et nœuds, ainsi que les clients d'objets Java programmés à l'étape 1 à des composants BCM4Java. Pour cela, il faut d'abord passer d'interfaces Java à des interfaces de composants BCM4Java. La figure 9 présente l'interface de composants `DHTServicesCI` qui remplace l'interface `DHTServicesI` alors que la figure 10 présente les interfaces de composants `ContentAccessSyncCI` et `MapReduceSyncCI` qui remplacent les interfaces Java `ContentAccessSyncI` et `MapReduceSyncI` respectivement. Ensuite, il faudra remplacer les points de connexions de type « POJO » par des points de connexion BCM. Et enfin, il faut remplacer les classes représentant la façade, les nœuds et les clients pas des composants BCM4Java.

De ce point de départ, les principales tâches à réaliser sont :

1. Créer les classes de ports entrants et sortants ainsi que celles des connecteurs correspondant aux interfaces de composants fournies.
2. Créer les classes de points de connexion BCM4Java utilisant les ports et connecteurs pour établir les connexions entre les composants client/façade ainsi que façade/nœud et nœud/nœud.
3. Transformer les classes de client, façade et nœuds en composants, ce qui requiert de définir leur cycle de vie et d'utiliser les points de connexion BCM4Java précédents pour les connecter les uns aux autres.
4. Adapter les implantations de ces classes de composants pour utiliser les *threads* prévus par BCM4Java.

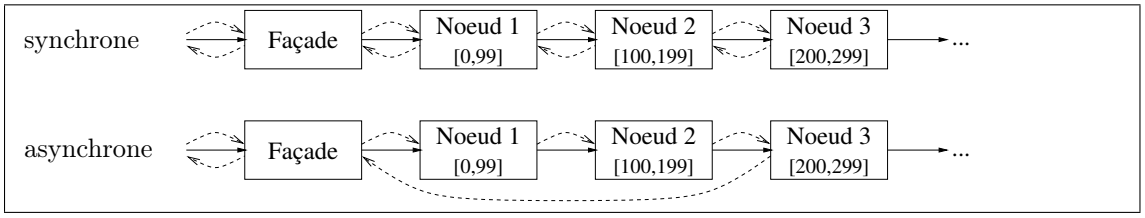

FIGURE 11 – Illustration du flôt de contrôle en appel synchrone comparé à l'appel asynchrone. En haut, l'appel synchrone pour recupérer une donnée contenue dans le nœud 3 passe par un appel au nœud 1 qui appelle le nœud 2 qui lu-même appelle le nœud 3, ces trois appels mobilisant à chaque fois un *thread* dans chacun des composants qui ne sera libéré pour le nœud 3 que lorsqu'il retourne son résultat au nœud 2 qui, à son tour ne libérer son *threrad* que lorsqu'il retourne son résultat au nœud 1 et ainsi de suite. En bas, l'appel asynchrone passe du nœud 1 au nœud 2 et le nœud 1 termine et libère son *thread* immédiatement après. Le nœud 2 fait de même et le nœud 3 utilise le point de connexion au nœud appelant qui lui est passé en paramètre, ici la façade, pour lui envoyer directement le résultat. Un parallélisme entre composants apparaît alors du fait que les *threads* libérés peuvent traiter de novelles requêtes pendant que les précédentes requêtes sont traitées par les nœuds suivants.

5. Programmer des scénarios de tests suffisamment complexes pour démontrer le bon fonctionnement de la table de hachage répartie, en incluant des traces permettant de s'assurer de ce bon fonctionnement lors des démonstrations.

À cette étape, les appels sont encore réalisés par une sémantique d'appel synchrone, c'est à dire que les composants appelants sont bloqués lors de leurs appels jusqu'à ce que les coposants appelants aient terminé l'exécution de leurs méthodes et retourner les résultats.

2.3 Troisième étape : parallélisme et concurrence

Dans cette troisième étape, l'objectif est d'introduire du parallélisme dans l'exécution de requêtes sur la table de hachage répartie, à la fois entre les requêtes reçues par la façade et, pour chaque requête, entre les nœuds et au sein des nœuds (dans ce dernier cas, pour les requêtes map/reduce). Mais ce parallélisme va engendrer des problèmes de cohérence dans les données partagées ce qui va nécessiter de traiter la concurrence entre les *threads* par des sections critiques et des synchronisations.

Déjà, observez qu'il est possible d'obtenir du parallélisme entre les requêtes reçues par la table de hachage répartie simplement en utilisant plusieurs *threads* dans la façade ainsi que dans les nœuds. Toutefois, l'utilisation d'appels synchrones limite beaucoup le parallélisme potentiel car chaque requête serait exécutée séquentiellement au sein de l'anneau. Pour les accès au contenu de la table de hachage, cela paraît anodin, mais pour les requêtes map/reduce c'est beaucoup moins intéressant. Ainsi, il faut aussi introduire du parallélisme entre les nœuds de l'anneau, ce qui requiert de passer à la programmation asynchrone. Cela va nécessiter de revoir les interfaces de composants des nœuds pour tenir compte des spécificités de ce type de programmation. Toutefois, comme il n'est pas souhaitable de complexifier le code des clients, l'interface de composants `DHTServicesCI`, elle, ne sera pas modifiée et nous maintiendrons des appels synchrones entre les clients et la façade de la table de hachage répartie.

Comme expliqué dans le cours, la programmation asynchrone se base sur une idée assez simple. Plutôt que d'attendre le résultat d'un appel de méthode, l'appelant passe en paramètre un point de connexion par lequel l'appelé pourra « envoyer » le résultat de l'appel initial lorsqu'il aura terminé son exécution. Par analogie, l'appel synchrone se compare à un appel téléphonique. Si vous appelez par exemple un service client d'une entreprise pour poser une question, votre interlocuteur cherche la réponse, temps pendant lequel vous attendez, puis vous la donne directement puis vous raccrochez. L'appel asynchrone s'assimile plutôt à un courriel. Vous envoyez un courriel avec votre question puis vous vaquez à vos autres occupations. Le service client cherche la réponse et vous l'envoie par courriel en utilisant votre adresse électronique. En plus, si votre interlocuteur délègue le travail à quelqu'un d'autre, cette personne pourra vous renvoyer directement la réponse sans repasser par votre interlocuteur initial.

La figure 11 reprend cette idée mais sur les appels synchrones versus asynchrones en BCM4Java. En appel synchrone, un composant appelant est bloqué en attente du résultat calculé côté appelé. En appel asynchrone, l'appelant lance la requête puis continue son exécution. Lorsqu'un composant appelé a le résultat, il le renvoie au composant appelant initial (ici la façade) grâce à un appel de méthode spécifiquement prévue pour cela. Côté nœuds, si un nœud ne détient pas une entrée par exemple, il peut passer la requête d'accès au nœud suivant et terminer son exécution de cette requête sans plus s'en soucier puisque le nœud suivant (ou un autre encore plus loin) se chargera de renvoyer la réponse directement à la façade. Pour les requêtes map/reduce, les opérations pourront se faire en parallèle entre les nœuds ainsi qu'entre les map et les reduce.

La figure 12 présente les interfaces utilisées en appel asynchrone pour l'accès au contenu des nœuds. Chaque méthode asynchrone, comme `get`, est déclarée avec un type de retour `void` puisqu'elle ne retourne pas de résultat (elles seront appelées par une `runTask` dans le port entrant). Elle a cependant un paramètre supplémentaire `caller` qui est un point de connexion qui sera utilisé par le composant nœuds en tant que client pour retourner le résultat à l'appelant (façade) jouant alors le rôle de serveur. Pour les requêtes d'accès au contenu, le point de connexion `caller` utilise l'interface de composants `ResultReceptionCI`. Lors le nœud veut retourner un résultat, il appelle la méthode `acceptResult` avec l'URI du calcul et son résultat. En faisant cela, il utilise le point de connexion `caller` en tant que client. Le composant appelant initial doit donc prévoir une méthode qui va traiter ce retour c'est à dire, pour la façade, retrouver le client en attente d'un résultat pour le lui transmettre comme expliqué à la figure 11.

Le code fourni pour le projet contient également les interfaces construites sur le même principe pour la partie map/reduce, c'est à dire les interfaces `MapReduceI`, `MapReduceCI`, pour les méthodes elles-mêmes ainsi que `MapReduceResultReceptionI` et `MapReduceResultReceptionCI` pour le retour des résultats (reduce). Vous noterez toutefois que comme la méthode `map` ne retourne pas de résultat, elle peut être exécutée de manière asynchrone sans avoir à lui ajouter un paramètre pour retourner un résultat. La méthode `reduce` retournant un résultat, elle, nous devons lui ajouter ce paramètre de point de connexion de retour de résultat. Le code des méthodes d'implantation devra prendre en compte cette nouvelle technique asynchrone.

La passage à la programmation asynchrone accroît considérablement le potentiel de parallélisme dans la table de hachage répartie car il peut alors y avoir du parallélisme entre les nœuds sur une même requête. Mais, du coup, cela va poser pas mal de questions sur la gestion de ce parallélisme. Assez souvent, dans les textes introduisant le parallélisme en Java, la question de la gestion des ressources, et en particulier des nombres de *threads* à utiliser, est passée sous silence. En fait, cela découle de deux utilisations de la notion de *threads* bien distinctes en programmation concurrente et un peu antagoniste :

1. Une facilité de programmation des tâches indépendantes où le *thread* est vu comme un processus virtuel créé, utilisé puis détruit à volonté. Ils ne s'exécutent pas nécessairement en parallèle réellement, mais plutôt dans un pseudo-parallélisme où il s'agit de passer d'un *thread* à l'autre pour faire progresser les calculs. Dans cette vision, par exemple, il n'est pas illogique d'utiliser plusieurs *threads* sur un ordinateur qui n'a qu'un seul processur et pas de cœur capables d'exécuter plusieurs choses en même temps.
2. Une deuxième vision des *threads* consiste à les voir comme représentant directement les processeurs et cœurs de processeurs disponibles et donc le parallélisme physique réel de l'ordinateur sous-jacent qui correspond au nombre de processeurs et de cœurs de processeurs pouvant effectivement exécuter des tâches en parallèle. Les *threads* sont alors vu comme une représentation de ces capacités physiques et le nombre de *threads* exécutables réellement en parallèle est limité par le nombre de ces processeurs et cœurs disponibles.

Depuis plus de 50 ans, l'histoire de la programmation a surtout cherché à éloigner le programmeur des questions de gestion de ressources, d'où la prévalence de la première interprétation pour les *threads i.e.*, le *thread* comme processeur ou cœur de processeur *purement virtuel*. Malheureusement, cette vision est mise à mal par la programmation pour des plates-formes à ressources limitées (téléphones, tablettes, systèmes embarqués, IoT, etc.) mais également par le passage à l'échelle des applications sur le *cloud* qui doivent s'assurer de la bonne marche des choses malgré une demande qui peut être largement plus grande que ce que peuvent absorber les ressources disponibles.

Si on prend la seconde interprétation, le nombre total de *threads* permettant une exécution *réellement parallèle* est borné. Il devient alors intéressant de choisir une bonne répartition de ces *threads* à affecter aux différents types de tâches qu'un programme est amené à exécuter. Ici,

```
public interface ContentAccessI extends ContentAccessSyncI
{
  public <CI extends ResultReceptionCI> void get(
    String computationURI, ContentKeyI key, EndPointI<CI> caller
    ) throws Exception;

  public <CI extends ResultReceptionCI> void put(
    String computationURI, ContentKeyI key, ContentDataI value,
    EndPointI<CI> caller
    ) throws Exception;

  public <CI extends ResultReceptionCI> void remove(
    String computationURI, ContentKeyI key, EndPointI<CI> caller
    ) throws Exception;
}

public interface ContentAccessCI extends ContentAccessSyncCI, ContentAccessI
{
  @Override
  public <CI extends ResultReceptionCI> void get(
    String computationURI, ContentKeyI key, EndPointI<CI> caller
    ) throws Exception;

  @Override
  public <CI extends ResultReceptionCI> void put(
    String computationURI, ContentKeyI key, ContentDataI value,
    EndPointI<CI> caller
    ) throws Exception;

  @Override
  public <CI extends ResultReceptionCI> void remove(
    String computationURI, ContentKeyI key, EndPointI<CI> caller
    ) throws Exception;
}

public interface ResultReceptionI
{
  public void acceptResult(String computationURI, Serializable result)
  throws Exception;
}

public interface ResultReceptionCI
extends OfferedCI, RequiredCI, ResultReceptionI
{
  @Override
  public void acceptResult(String computationURI, Serializable result)
  throws Exception;
}
```

FIGURE 12 – Interfaces de composants déclarant la version asynchrone des services d'accès au contenu offerts par les nœuds de la table de hachage répartie ainsi que les interfaces utilisées côté façade et nœuds pour recevoir les résultats des calculs.

combien de *threads* à affecter aux opérations d'accès au contenu ? Combien aux opérations de *map/reduce* ? Et parmi les opérations de ces deux ensembles, doit-on les répartir à nouveau entre les opérations (par exemple, entre les *map* et les *reduce*) ou les partager ?

Pour compléter l'étape 3, vous devrez donc réfléchir et doter votre table de hachage répartie d'une politique de gestion du parallélisme. En partant de l'idée que vous disposez d'un nombre

total de *threads* limité :

— Il s'agit pour vous d'analyser les opérations de chacun des composants pour séparer leurs méthodes en groupes où les requêtes concernant chaque groupe seront traitées par un *pool* de *threads* distinct.

— Ensuite, il faudra définir des choix de nombre de *threads* à allouer à chaque groupe sous la forme d'une politique (définie par une classe et représentée par ses instances) que vous utiliserez dans la création des composants pour créer les *pools* de *threads* nécessaires.

Comme nous l'avons vu en cours, le parallélisme engendre des problèmes de concurrence. Il faudra donc aussi ajouter à vos composants une gestion interne de la concurrrence. Cela se développe en trois points à appliquer à chaque composant :

1. Déterminez les structures de données qui doivent être accédées conjointement par les opérations.
2. Choisissez une technique de synchronisation à appliquer pour protéger conjointement ces structures de données.
3. Identifiez dans le code des opérations du composant les sections critiques où vous ajouterez le code de synchronisation nécessaire pour garantir l'exclusion mutuelle.

2.4 Quatrième étape : cordes, dynamicité et répartition

Dans cette quatrième et dernière étape du projet, deux objectifs principaux seront poursuivis :

— calculer les cordes (introduites à la section 1.3 et illustrées à la figure 3) de manière à augmenter la performance des accès au contenu et le parallélisme des calculs map/reduce ;

— introduire de la dynamicité dans l'anneau en permettant de scinder ou fusionner les nœuds pendant l'exécution afin d'équilibrer la charge des données à stocker dans l'ensemble de la table de hachage répartie, dont les concepts ont été introduits à la section 1.4.

En bonus, l'objectif ultime sera d'exécuter la table de hachage répartie sur plusieurs machines virtuelles Java.

2.4.1 Calcul des cordes

Le premier objectif de cette quatrième étape est le calcul des cordes. Les signatures de méthodes nécessaires sont déclarées dans les interfaces de gestion de la table de hachage répartie `DHTManagementI` et `DHTManagementCI` qui apparaissent à la figure 13. L'interface de composant `DHTManagementCI` est offerte et requise par les composants nœuds.

Pour le calcul des cordes, la signature `computeChords` devra être implantée comme méthode des composant nœuds pour effectuer ce calcul globalement par un passage sur l'ensemble des nœuds de l'anneau. Elle prend en paramètres une URI de calcul, comme la plupart des signatures précédentes, et un nombre de cordes à calculer pour chacun des nœuds. Pour mémoire, les cordes d'un nœud en position i dans l'anneau sont les nœuds à distance d'une puissance de 2 de ce nœud, c'est à dire :

— le nœud [4] $i + 1$ pour la corde 0 ($i + 2^0$),

— le nœud $i + 2$ pour la corde 0 ($i + 2^1$),

— le nœud $i + 4$ pour la corde 0 ($i + 2^2$),

— le nœud $i + 8$ pour la corde 0 ($i + 2^3$),

— etc.

Si le nombre de cordes à calculer pour chaque nœud est 4, les cordes 0 à 3 seront à construire. Le bon nombre de cordes à calculer dépend essentiellement de deux critères : l'amélioration de performance que ces cordes permettent et le nombre total de nœuds dans l'anneau. Généralement, moins de quatre cordes n'offrent pas une amélioration justifiant leur complexité, ce qui est un nombre minimal. Pour un anneau de très grande taille, choisir la plus grande puissance de 2 inférieure au nombre N de nœuds est possible, car l'espace mémoire nécessaire pour stocker les cordes demeurent dans l'ordre de $\log N$, mais dans ce cas il n'est pas toujours nécessaire de conserver toutes les cordes.

4. La corde 0 correspond au nœud suivant dans l'anneau, elle fait donc doublon avec la connexion déjà réalisée par l'anneau. Pour simplifier les choses et aussi garantir une connectivité complète entre les nœuds de l'anneau, il faudra maintenir les deux : le point de connexion vers le nœud suivant et la corde 0, même s'ils sont connectés au même nœud.

```
public interface DHTManagementI
{
  public interface NodeStateI extends Serializable { }
  public interface NodeContentI extends Serializable { }

  @Override
  public void   initialiseContent(NodeContentI content) throws Exception;

  @Override
  public NodeStateI getCurrentState() throws Exception;

  @Override
  public NodeContentI suppressNode() throws Exception;

  @Override
  public <I extends ResultReceptionCI> void split(
    String computationURI, LoadPolicyI loadPolicy, EndPointI<I> caller
    ) throws Exception;

  @Override
  public <I extends ResultReceptionCI> void merge(
    String computationURI, LoadPolicyI loadPolicy, EndPointI<I> caller
    ) throws Exception;

  @Override
  public void computeChords(String computationURI, int numberOfChords)
  throws Exception;

  @Override
  public SerializablePair<
             ContentNodeCompositeEndPointI<
                 ContentAccessCI,
                 ParallelMapReduceCI,
                DHTManagementCI>,
             Integer>    getChordInfo(int offset) throws Exception;
}
```

FIGURE 13 – Interfaces Java déclarant les services de gestion offerts par les nœuds de la table de hachage répartie et les interfaces Java qui y sont utilisées. L'interfaces de composants `DHTManagementCI` reprend exactement les mêmes signatures.

Lorsqu'on calcule les cordes pour un nœud donné, il faut aller chercher des informations sur un nœud qui le suit dans l'anneau. La signature `getChordInfo` servira à cela. Cette signature fait apparaître un paramètre `offset` qui est le décalage en nombre de nœuds à partir du nœud appelé dont on cherche les informations : si `offset` vaut 0, ce sont les informations du nœud appelé qui sont requises, s'il vaut 1 c'est le suivant et ainsi de suite.

Deux informations sont nécessaires et donc sont retournées sous la forme d'une paire [5] à l'appelant. D'abord, il faut obtenir un point de connexion côté client permettant de se connecter à ce nœud (le point de connexion côté client qui se connecte au même point de connexion côté serveur qui est utilisé pour connecter un nœud à son suivant). Ensuite, pour des raisons expliquées ci-après (§2.4.2), il faut également obtenir la plus petite valeur de hachage des clés détenues par le nœud, ce qui est l'entier (`Integer`) second membre de la paire retournée par `getChordInfo`.

Pour simplifier les choses, ces deux méthodes sont à implanter en mode synchrone et elles pourront s'exécuter en exclusion mutuelle avec toute autre opération sur l'ensemble de la table de hachage répartie (ce dont vous devrez toutefois vous assurer). Leur scénario d'utilisation consiste à exécuter un premier calcul au lancement (on suppose qu'il y aura un nombre de nœuds dans

5. La classe `SerializablePair` est fournie dans le répertoire `provided`.

```
public interface ParallelMapReduceI
{
  public interface ParallelismPolicyI extends Serializable { }

  @Override
  public <R extends Serializable> void parallelMap(
    String computationURI,
    SelectorI selector,
    ProcessorI<R> processor,
    ParallelismPolicyI parallelPolicy
    ) throws Exception;

  @Override
  public <A extends Serializable, R, I extends MapReduceResultReceptionCI>
                      void parallelReduce(
    String computationURI,
    ReductorI<A, R> reductor,
    CombinatorI<A> combinator,
    A identityAcc,
    A currentAcc,
    ParallelismPolicyI parallelismPolicy,
    EndPointI<I> caller
    ) throws Exception;
}
```

FIGURE 14 – Interfaces de composants déclarant la version parallèle des services map/reduce offerts par les nœuds de la table de hachage répartie.

l'anneau dès le démarrage suffisant pour justifier le calcul des cordes), puis de le réexécuter après avoir fusionner ou scinder des nœuds (voir ci-après §2.4.3) pour mettre à jour les informations suite à une modification de l'anneau.

Si vous arrivez à faire exécuter ce calcul sans imposer une exclusion mutuelle globale sur l'ensemble de l'anneau mais seulement une exclusion mutuelle au niveau des nœuds, vous pourrez bénéficier d'un bonus à condition d'expliquer comment vous avez fait lors de la soutenance finale.

2.4.2 Exploitation des cordes

Exploiter les cordes se fait de deux manière différentes :

— Pour les méthodes d'accès au contenu, lorsqu'un nœud ne détient pas le couple clé/valeur, il peut utiliser les informations sur les cordes pour trouver la corde qui détient la valeur de hachage la plus proche de celle de la clé reçue sans la dépasser pour appeler directement cette corde plutôt que de passer nœud par nœud.

— Pour les méthodes `map` et `reduce`, il s'agit de lancer en parallèle les calculs vers les cordes plutôt que de simplement vers le nœud suivant, mais cela mérite quelques explications supplémentaires.

L'exploitation des cordes pour les accès au contenu de la table de hachage répartie a été expliqué à la section 1.3. Par contre, pour les calculs map/reduce, le fait de pouvoir lancer les map sur plusieurs des cordes va permettre d'obtenir du parallélisme supplémentaire entre les nœuds, mais exploiter ce parallélisme n'est pas si simple. À l'étape 3, il a été possible de s'appuyer sur l'appel asynchrone pour lancer le calcul sur le nœud suivant en parallèle avec le calcul sur le nœud courant. Utiliser les cordes permet d'augmenter plus rapidement le nombre de nœud exécutant leurs calculs locaux en parallèle avec les autres nœuds. Cela dit, il faut bien parcourir tous les nœuds, donc il faut établir une bonne stratégie de parallélisme.

Déjà, on constate facilement que la corde 0 ne servira pas à produire plus de parallélisme qu'à l'étape 3 puisque le calcul pouvait déjà être lancé sur le nœud suivant (qui est le même que celui de la corde 0). Que se passe-t-il si on utilise la corde 1 du nœud i pour lancer en parallèle le calcul sur le nœud $i + 2$? Dans ce cas, le nœud i lancerait le calcul localement sur sa table de hachage

et en parallèle sur le nœud $i+2$ (corde 1). Mais il ne faut pas sauter le nœud $i+1$, donc pour la stratégie exploitant la corde 1, le nœud i doit aussi lancer en parallèle le calcul sur le nœud $i+1$ en plus du nœud $i+2$. Alors, le nœud $i+1$ étant appelé, il va lui-même chercher à lancer le calcul sur son suivant, le nœud $i+2$, et sur celui de sa corde 1, le nœud $i+3$. Mais le nœud $i+2$ aura déjà été appelé par le nœud i (via sa corde 1), donc l'appel venant du nœud $i+1$ ne servira à rien (il faudra reconnaître que le calcul a déjà été fait lors du second appel pour ne pas le refaire une deuxième fois).

De cette explication, on voit que la stratégie pour obtenir du parallélisme dans les calculs map/reduce via les cordes n'est pas si simple. Vous devrez donc concevoir une stratégie et l'implanter. Pour vous aider, deux nouvelles signatures pour map/reduce sont ajoutées par l'interface Java `ParallelMapReduceI` et l'interface de composants `ParallelMapReduceCI` (figure 14) : `paralleMap` et `parallelReduce`. Ces deux signatures sont identiques à la version asynchrone des deux signatures sauf pour un paramètre supplémentaire `parallelismPolicy` qui doit vous servir à exprimer votre stratégie de parallélisme. Le type de ce paramètre est `ParallelismPolicyI` qui est une simple interface marqueur étendant `Serializable`. Ainsi, vous pouvez créer une classe exprimant votre stratégie et lui faire implanter cette interface pour en passer une instance lors des appels aux méthodes `paralleMap` et `parallelReduce`.

Bien sûr, le parallélisme sur `parallelReduce` va engendrer plusieurs résultats de réduction qu'il va falloir combiner pour obtenir le résultat de la réduction sur l'ensemble de la table de hachage répartie. Il faudra également choisir une solution pour cela et la mettre en œuvre.

2.4.3 Fission et fusion de nœuds

Pour gérer la dynamicité de l'anneau (fission et fusion de nœuds), les interfaces `DHTManagementI` et `DHTManagementCI` déclarent deux principales signatures à implanter par des méthodes qui vont balayer l'anneau en parcourant successivemtn les nœuds :

`split :` elle prend trois paramètres, une URI de calcul, une politique de gestion de la charge (`LoadPolicyI` et un point de connexion de l'appelant initial (ici, la façade). L'idée est de lancer un balayage sur l'anneau du premier au dernier nœud qui regarde successivement chacun des nœuds, décide s'il faut le scinder en deux et le cas échéant crée un nouveau nœud suivant, l'intègre dans l'anneau et lui passe une partie de ses données en utilisant la signature `initialiseContent` dont le paramètre de type `NodeContentI` vous laisse toute liberté de concevoir une classe contenant toutes les informations à transférer au nouveau noeud. Lorsque le balayage complet est terminé, le point de connexion est utilisé pour signaler à l'appelant initial la terminaison de la passe de fission.

`merge :` elle prend les trois mêmes paramètres. L'idée est aussi de lancer un balayage sur l'anneau du premier au dernier nœud mais qui regarde successivement les nœuds deux par deux (un nœud et son successeur), décident s'il faut les fusionner (en utilisant la signature `getCurrentState` retournant un résultat de type `NdeStateI` vous laissant toute latitude de décider quelles informations sont nécessaires pour décider) et le cas échéant détruit l'un des nœuds en récupérant ses données dans l'autre (en utilisant la signature `suppressNode` retournant un résultat de type `NodeContentI` vous laissant toute latitude de décider quelles informations doivent être récupérées). Lorsque le balayage est terminé, le point de connexion est utilisé pour signaler à l'appelant initial la terminaison de la passe de fusion.

Le paramètre de type `LoadPolicyI` exprime la stratégie de gestion de la charge des nœuds selon une des approches proposées dans la section 1.4. La présence d'un point de connexion dans les paramètres de ces deux signatures indiquent une exécution en programmation asynchrone. Toutefois, comme pour le calcul des cordes c'est à dire pour simplifier les choses, les balayages sur ces deux opérations sont réalisés en exclusion mutuelle avec toute autre opération sur la tables de hachage répartie. Cela dit, si vous arrivez à faire exécuter ce calcul sans imposer une exclusion mutuelle globale sur l'ensembke de l'anneau mais seulement une exclusion mutuelle au niveau des nœuds, vous pourrez bénéficier d'un bonus à condition d'expliquer comment vous avez fait lors de la soutenance finale.

Le scénario d'utilisation de ces opérations doit être pour vous l'occasion de réfléchir à la gestion des ressources dans une application répartie. On constate que ces opérations sont coûteuses car elles sont exécutées en exclusion mutuelle et sur une table de hachage répartie qui peut contenir

beaucoup de nœuds. Il faut donc les lancer parcimonieusement, d'autant qu'elles vont devoir être suivies d'un recalcul global des cordes (un recalcul local serait possible, mais pose à la fois des problèmes de mise en œuvre et de performance). Il faut donc guider leur lancement par une politique de gestion fondée sur des critères précis, comme par exemple :

— à une certaine fréquence fixée à l'avance, ce qui est la politique la plus simple à mettre en œuvre ;
— en faisant un bilan des ajouts et retraits pour effectuer les balayages utiles au moments où on pense qu'ils seront vraiment utiles (ce qui n'est possible que si toutes les requêtes des clients passent par un point d'entrée unique, ici la façade) ;
— selon la charge courante en nombre de requêtes par seconde reçues des clients, avec l'idée que ces opérations en exclusion mutuelle devraient être exécutées aux moments les moins gênants pour les clients ;
— etc., y compris une combinaison des critères précédents.

Il vous faudra choisir un de ces critères et le mettre en œuvre dans votre projet.

2.4.4 Bonus : exécution répartie

Le passage à une exécution répartie en BCM4Java est le test ultime de la bonne programmation d'une application montrant que le code permet bien de passer sans erreur d'une exécution sur une seule machine virtuelle à une exécution sur plusieurs machines virtuelles utilisant les appels RMI. Bien que cet objectif soit important, ce passage exige passablement de travail au départ (programmation des classes de déploiement étendant `AbstractDistributedCVM`, rédaction du fichier de configuration en XML, déploiement du code). Et lorsque des erreurs se manifestent, la mise au point est encore plus lente que dans le cas d'un programme *multi-threadé* s'exécutant dans une seule machine virtuelle.

Pour toutes ces raisons, le passage à une exécution raprtie est considérée comme un bonus dans l'évaluation finale. En clair, cela veut dire qu'il demeure possible d'obtenir la note maximale sans faire ce passage, mais que si vous le faites cela pourra compenser des points perdus sur d'autres aspects du projet.

Pour être convaincant du point de vue vérification et validation, une exécution répartie doit s'assurer que pour tous les types de connexions et tous les types de composants, au moins une connexion se fasse à distance, via RMI. Dans le projet, il y a deux types de connexions et trois types de composants, donc il faut qu'au moins une connexion RMI soit faite :

— entre au moins un client et la façade,
— entre la façade et le premier nœud dans l'anneau,
— entre au moins deux nœuds.

Pour satisfaire ces contraintes, il faut un déploiement sur au moins deux machines virtuelles MV_1, MV_2 et une répartition des composants où au moins un client se trouve dans MV_1, la façade dans MV_2 et le premier nœud de l'anneau dans MV_1. Au delà de ce déploiement minimal, un déploiement sur quatre machines virtuelles, une pour les clients, une pour la façade et deux pour les nœuds seraient approprié. La répartition des nœuds entre les deux machines virtuelles dédiées devrait faire en sorte de favoriser le plus grand nombre possible de connexions RMI entre eux, ce qui permettrait de mieux se rendre compte de l'impact de ces connexions sur la performance des opérations.

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) : Introduction des cordes dans la table de hachage répartie.

   Semaine 9 (31/03-1/04) : (en parallèle avec l'audit 2) 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 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 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

L'ensemble des étape 1 et 2 feront l'objet de l'évaluation de mi-semestre qui prendra la forme d'une soutenance avec rendu préalable de code (voir plus loin les modalités d'évaluation). Pour cette évaluation, les critères de notation porteront d'abord sur l'état d'avancement de votre code :

— une implantation complète des fonctionnalités des composants façade et nœuds ;
— une implantation de tous les ports, connecteurs et points de connexion correspondant aux interfaces de composants fournies.

Ils porteront également sur une exécution d'un test d'intégration sur une seule machine virtuelle Java où seront exécutés couvrant l'ensemble des fonctionnalités et de la table de hachage répartie et des cas aux limites comme le retrait d'une entrée inexistante.

Vous devrez rendre votre code le dimanche soir précédant la soutenance (voir les modalités décrites ci-après). Pour cette étape, en plus du taux de couverture des réalisations demandées (étapes 1 et 2) et de leur bon fonctionnement en exécution démontré par les scénarios de test demandés, la qualité de l'ensemble de votre code sera également un critère d'évaluation :

— lisibilité et compréhensibilité ;
— pertinence des choix de conception ;
— qualité de la programmation Java ;
— pertinence des choix d'implantation.

3.4 Audit 2 : exécution asynchrone, parallélisme et concurrence

L'objectif principal de la troisième étape sera d'ajouter l'exécution asynchrone des requêtes, assurer une gestion explicite du parallélisme par utilisation de *pools* de *threads* créés explicitement et maîtriser la concurrence par l'introduction de sections critiques dans le code.

L'audit 2 se déroulera pendant la neuvième séance de TME et les résultats attendus sont :

— l'ajout des appels aynchrones pour l'exécution des requêtes ;
— l'introduction de *pools* de *threads* distincts pour l'exécution parallèle des requêtes sur la façade et les nœuds ;
— la gestion de la concurrence ;
— la capacité à expliquer vos choix de conception et d'implantation ;
— les scénarios de tests d'intégration permettant de vérifier le bon fonctionnement des appels asynchrones, du parallélisme et de la concurrence en faisant s'exécuter plus d'une requête à la fois sur les mêmes composants nœuds.

Concernant les scénarios des tests, vous allez être confrontés probablement pour une première fois au besoin de monter des scénarios de tests *temporisés*, c'est à dire où il faut maîtriser les moments en temps réel où les actions de tests sont exécutés. ici, il s'agira d'utiliser plusieurs composants clients qui vont faire des appels à la façade au même instant pour forcer une exécution parallèle. L'annexe A explique comment utiliser un outil fourni par BCM4Java pour monter de tels tests : l'horloge accélérée. Il s'agit d'une horloge partagée entre composants qui permet de planifier des actions à des instants précis. Grâce à cet outil, vous pourrez synchroniser les clients pour leur faire envoyer des requêtes au même instant.

Pour cet audit également, le principal critère d'évaluation sera le degré de réalisation des développements demandés et le bon fonctionnement des tests d'intégration.

3.5 Soutenance finale : table de hachage dynamique et répartition

Bien que l'évaluation finale soit récapitulative sur l'ensemble du projet, la nature cumulative des développements fait que cette évaluation portera en réalité sur le résultat final et dépendra donc en bonne partie de l'état d'avancement de votre travail par rapport aux développements demandés. En gros, les grands jalons approchant graduellement le résultat final attendu sont :

1. Le résultat de l'étape 3 déjà évalué par l'audit 2 mais dont le code n'aura pas été examiné complètement, c'est à dire les calculs sur les nœuds implantés en programmation asynchrone, effectués en parallèle avec gestion de la concurrence.
2. Calculs des cordes et leur exploitation dans les méthodes d'accès au contenu.

3. Exploitation des cordes pour améliorer le parallélisme dans les calculs map/reduce.
4. Implantation de la gestion dynamique des nœuds avec fusion et fission de nœuds.
5. Et, en bonus, exécution répartie sur plusieurs machines virtuelles.

Pour réussir au mieux l'évaluation finale, il est vivement suggéré d'organiser votre travail de manière à sécuriser chacun de ces jalons un à un, en prenant soin de conserver un état instantané de votre code au dernier jalon terminé auquel vous pourrez revenir pour l'exécution du test d'intégration si vous n'avez pas le temps de tout compléter jusqu'au dernier jalon. Il faut bien sûr surtout éviter de débuter le code pour plusieurs de ces jalons en même temps sans pouvoir aller au bout d'aucun.

La soutenance finale se déroulera pendant la semaine des examens de fin de semestre et elle visera à évaluer cumulativement l'ensemble des réalisations du projet (étapes 1 à 4). Lors de celle-ci, les échanges se concentreront sur :
— l'ensemble des développements que vous aurez réalisés en fonction de l'état d'avancement de votre projet qui sera évalué en parcourant des éléments de votre code ;
— des scénarios de tests d'intégration sous la forme de déploiements comportant 5 composants clients et au moins 16 composants nœuds avec 4 cordes calculées par nœud, que vous devrez expliquer et en faire exécuter au moins un.

Lors de la soutenance finale, il sera possible de préparer une petite présentation si cela vous aide à expliquer vos choix de conception et vos réalisations.

Vous devrez rendre votre code en amont de la soutenance (voir les modalités décrites ci-après). Si vous n'êtes pas arrivés au dernier jalon, il vous sera possible de rendre à la fois le code du dernier jalon complété et le code incomplet du jalon suivant. Vous prendrez alors soin de bien indiquer les jalons concernés pour chaque rendu. Pour cette étape, en plus du taux de couverture des réalisations demandées et de leur bon fonctionnement en exécution, la qualité de l'ensemble de votre code selon les critères donnés pour la soutenance de mi-semestre auxquels va s'ajouter la qualité de votre documentation.

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 30 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 devant l'écran sous Eclipse et une démonstration (exécution) des tests. 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 30 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 et comportera une discussion sur vos choix de conception et d'implantation ainsi que sur les réalisations suivie d'une démonstration (exécution) des tests. 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 et, le cas échéant, scénarios de tests pour les composants) ;
  — l'exécution correcte de tests d'intégration mettant en œuvre des composants de tous les types ;
  — la qualité des scénarios de tests, conçus pour mettre à l'épreuve les choix d'implantation ;
  — la qualité de votre code autant dans la technicité que dans la maintenabilité (votre code doit être clair, commenté, lisible – choix pertinents des identifiants, . . . – et correctement présenté – indentation, . . . – etc.) ;
  — la qualité de la documentation (vos rendus pour la soutenance finale devront inclure une *documentation Javadoc* des paquetages et classes de votre projet générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que vos répertoires de code source.
— Bien que les audits et soutenances se fassent par équipe, l'évaluation reste **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 de membres de l'équipe de plus d'un tiers de la durée de la soutenance 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 sans les retardataires.
— 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 les répertoires de code source — `src` ou autre répertoires que vous aurez créés — 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.

A Scénarios de tests et horloge accélérée

Le test des applications avec interfaces graphiques et encore plus des applications parallèles et réparties exige de pouvoir planifier des actions de différentes entités logicielles à des instants précis les unes par rapport aux autres pour construire des scénarios d'exécution corrects et réalistes. Cela demande donc de synchroniser ces entités de telle manière à ce qu'elles exécutent les actions dans un ordre précis. Plusieurs techniques sont envisageables pour ce faire, dont celle d'une synchronisation *dirigée par le temps* que nous allons utiliser dans le cadre du projet CPS.

A.1 Scénarios de test temporisés

Pour le projet, par exemple, vous comprendrez vite que lors du démarrage, il faut que les composants nœuds du réseau de capteurs s'inscrivent auprès du registre (action 1) avant que les composants clients puissent récupérer auprès du registre (action 2) les informations de connexion sur ces nœuds pour leur envoyer des requêtes (action 3). Si on ne synchronise pas ces actions, des conditions de course (*race conditions*) vont prévaloir entre les composants et selon l'ordre dans lequel la machine virtuelle Java va exécuter les choses, on pourra très bien avoir l'action 2 qui sera exécutée avant l'action 1, ce qui mènera à une erreur.

BCM4Java offre une classe appelée `AcceleratedClock` ainsi qu'un composant `ClocksServer` pour simplifier la construction de scénarios de test temporisés. Premièrement, pour synchroniser les actions des composants de manière dirigée par le temps, il faut que tous les composants puissent accéder à une unique horloge qui va établir un temps global suffisamment précis pour qu'on puisse faire exécuter une série d'actions par différents composants dans un ordre choisi par le programmeur. Que l'on soit sur un seul ordinateur ou sur plusieurs ordinateurs dont les horloges matérielles sont synchronisées avec suffisamment de précision, l'horloge matérielle est une bonne candidate pour servir de temps global.

En Java, la méthode `System.currentTimeMillis()` retourne la valeur de l'horloge matérielle de l'ordinateur sous-jacent exprimée en ce qui est appelé l'*Unix epoch time*, c'est à dire en temps écoulé depuis le 01/01/1970 en millisecondes. Par ailleurs, les *pools* de *threads* ordonnançables de l'`ExecutorService` de Java offrent la possibilité de faire exécuter du code après un certain délai. BCM4Java s'appuie sur cette implantation pour offrir la méthode `scheduleTask` :

```
final AbstractComponent c = this;
this.scheduleTask(
  o -> { c.traceMessage("Ceci sera exécuté après le délai d nanosecondes.\n"); },
  d, TimeUnit.NANOSECONDS);
```

Attention, pour pouvoir utiliser la méthode `scheduleTask` (et les autres méthodes du même genre), il faut que le composant possède au moins un *thread* dans un *pool* de *threads* ordonnançable.

A.2 Horloge accélérée

Bien que l'horloge matérielle donne une base fixe et fiable sur plusieurs ordinateurs, planifier des actions de test dans cette référence temporelle n'est pas aisée. Pour faciliter la planification des actions, la classe `AcceleratedClock` de BCM4Java offre la possibilité d'utiliser une autre référence temporelle pour planifier les actions : celle de la classe `Instant` de Java. Sans entrer dans tous les détails, il est facile de créer un instant « virtuel » grâce à la méthode statique `Instant#parse` qui prend en paramètre une chaîne de caractères où un instant est exprimé selon la norme internationale suivante :

"2025-01-31T09:00:00.00Z"

c'est à dire année-mois-jour puis heure, minutes, secondes et centièmes de secondes, le Z indiquant le fuseau horaire. L'expression `Instant.parse("2024-01-31T09:00:00.00Z")` retourne donc l'objet instant représentant le 31 janvier 2025 à 9 heures du matin.

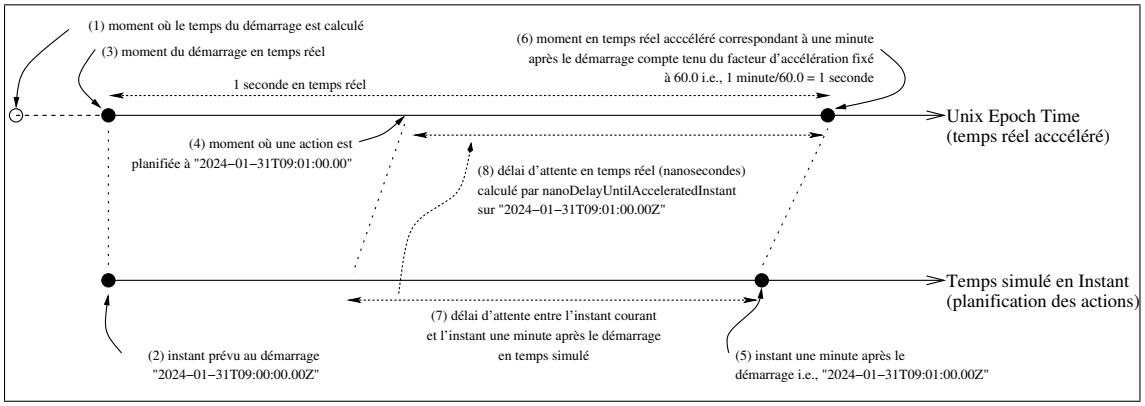

FIGURE 15 – Les deux références temporelles mises en synchronisation par `AcceleratedClock` : le temps réel accéléré s'appuyant sur l'horloge matérielle de l'ordinateur via le système d'exploitation et le temps virtuel des actions du scénario de test représenté par des `Instant` de Java.

Supposons que l'instant précédent représente l'instant de démarrage des actions du scénario de test, grâce aux instants Java on peut planifier une action devant se dérouler une minute après le démarrage à 9h01. Soit on crée un nouvel instant avec :

```
Instant.parse("2024-01-31T09:01:00.00Z")
```

Soit on crée de manière équivalente un nouvel instant par rapport à un instant précédent :

```
Instant i0 = Instant.parse("2025-01-31T09:00:00.00Z");
Instant i1 = i0.plusSeconds(60);
```

On constate qu'il sera beaucoup plus facile et lisible de travailler avec des instants Java pour planifier les actions d'un scénario de test qu'avec l'*Unix epoch time* en millisecondes.

Maintenant, la question qui se pose est comment mettre en relation un temps simulé exprimé en `Instant` avec le temps Unix qui sert de base pour ordonnancer l'exécution de tâches à des moments précis en temps réel. L'idée va être de créer une horloge qui va maintenir une synchronisation entre les deux références temporelles. Cette classe, `AcceleratedClock`, va permettre de créer une horloge en lui passant un moment précis en temps réel Unix pour le démarrage des actions du scénario de test disons `t0` et un instant en temps simulé du scénario exprimé en `Instant` disons `i0`. Après le démarrage du scénario (après `t0`), l'horloge va permettre de convertir des temps réel Unix (postérieurs à `t0`) en instants de même que des instants (postérieurs à `i0`) en temps réel Unix.

Un autre aspect intéressant à prendre en compte est la possibilité de planifier des actions selon une échelle de temps commode mais de pouvoir ensuite exécuter le scénario beaucoup plus rapidement. Par exemple, on pourrait trouver plus simple de planifier un scénario de test avec des actions se déclenchant toutes les minutes, mais ensuite d'accélérer ce temps pour faire en sorte qu'une minute planifiée s'exécute en une seconde en temps réel. Pour ce faire, `AcceleratedClock` va permettre de définir à la création d'une horloge un facteur d'accélération entre le déroulement du temps simulé et le déroulement du temps réel.

Pour comprendre, considérez la figure 15. En (1), on va calculer et programmer le temps de démarrage du scénario de test. Souvent, ce sera fait dès le début de l'exécution de l'application et il faudra laisser suffisamment de temps pour l'initialisation et le démarrage de tous les composants. En définissant un délai initial de démarrage, on pourrait définir le moment de démarrage de la manière suivante :

```
public static final long START_DELAY = 3000L; // 3000 millisecondes
long unixEpochStartTimeInNanos =
  TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis() + START_DELAY);
```

En (2), apparaît l'instant de départ matérialisé sur la référence temporelle du scénario de test et en (3) le moment du démarrage en temps réel est matérialisé sur la référence temporelle du temps réel Unix.

Pour pouvoir se référer à une horloge, on lui attribue à sa création une URI. Ainsi, la création d'une horloge accélérée peut se faire de la manière suivante :

```
public static final String CLOCK_URI = "horloge-101";
AcceleratedClock ac =
  new AcceleratedClock(
      CLOCK_URI,        // URI attribuée à l'horloge
      t0,               // moment du démarrage en temps réel Unix
      i0,               // instant de démarrage du scénario
      60.0)             // facteur d'acccélération
```

Avant de pouvoir faire des calculs avec cette horloge, il faut attendre le moment du démarrage en temps réel. Pour cela, la classe `AcceleratedClock` définit une méthode `waitUntilStart()` qui, comme son nom l'indique bloque le *thread* qui l'appelle jusqu'à ce que l'on arrive en temps réel au temps `t0` choisi pour le démarrage du scénario.

Supposons maintenant qu'à un certain moment dans l'exécution (voir (4) sur la figure) se situant avant le moment en temps réel où elle doit être déclenchée, le programme planifie une action devant se déclencher à l'instant `i1`, c'est à dire une minute après le démarrage du scénario de test qui est matérialisé en (5). Grâce à la synchronisation modulo le facteur d'accélération, `AcceleratedClock` peut reporter cet instant `i1` sur la référence temporelle du temps réel Unix en (6). Elle peut alors calculer le délai en termes d'instants à attendre depuis le moment où on veut planifier cette action jusqu'à `i1` en (7). Elle peut aussi calculer le même délai mais cette fois-ci en temps réel Unix, indiqué en (8). Ce délai en temps réel est celui qu'il faut utiliser dans la méthode `scheduleTask` pour faire s'exécuter l'action au bon moment :

```
long d = ac.nanoDelayUntilInstant(i1); // délai en nanosecondes
final AbstractComponent c = this;
this.scheduleTask(
  o -> { c.traceMessage("Le code de l'action 1 doit être placé ici.\n"); },
  d, TimeUnit.NANOSECONDS);
```

Le moment où la planification est faite est le moment auquel l'expression `ac.nanoDelayUntilInstant(i1)` est exécutée et le délai `d` est le délai en temps réel Unix à attendre avant de déclencher l'exécution de l'action 1.

A.3 Partage d'horloges accélérées : le serveur d'horloge

Maintenant que l'on a vu les principales fonctionnalités de l'horloge accélérée, [6] se pose un nouveau problème : comment faire en sorte que plusieurs composants, s'exécutant potentiellement sur plusieurs ordinateurs, puissent partager une *même* horloge accélérée afin que leurs actions planifiées soient effectivement synchronisées selon le temps qui s'écoule sur cette horloge.

Comme on l'a répété en cours, on ne peut pas partager une référence Java à un objet horloge entre plusieurs machines virtuelles Java. La solution consiste à fournir un composant serveur d'horloge sur lequel on va créer des horloges avec leurs URIs puis les composants vont pouvoir appeler ce composant pour récupérer des copies de cette horloge en utilisant l'interface `ClocksServerCI` offerte par `ClocksServer` et requise par les composants qui souhaitent l'utiliser. Toutes les copies d'une même horloge seront synchronisées entre elles par le fait qu'elles utilisent l'horloge matérielle comme référence temporelle. Que ces composants soient tous sur le même ordinateur et donc se réfèrent à la même horloge matérielle par définition offrant une synchronisation parfaite, ou qu'ils soient sur différents ordinateurs donc les horloges matérielles de ces ordinateurs seront supposées suffisamment synchronisées entre elles pour que les horloges accélérées le soient également.

Dans le cadre du projet, il suffira de créer un composant serveur d'horloge à partir de la classe `ClocksServer`. Ce composant permet de créer une horloge en passant les paramètres nécessaires à l'un de ses constructeurs. Par exemple, pour l'horloge précédente, cela donne :

6. Consultez la documentation javadoc de BCM4Java pour plus de détails.

```
public static final String TEST_CLOCK_URI = "test-clock";
public static final Instant START_INSTANT =
                              Instant.parse("2024-01-31T09:00:00.00Z");
protected static final long START_DELAY = 3000L;
public static final double ACCELERATION_FACTOR = 60.0;

long unixEpochStartTimeInNanos =
  TimeUnit.MILLISECONDS.toNanos(System.currentTimeMillis() + START_DELAY);

AbstractComponent.createComponent(
  ClocksServer.class.getCanonicalName(),
  new Object[]{
        TEST_CLOCK_URI,             // URI attribuée à l'horloge
        unixEpochStartTimeInNanos,  // moment du démarrage en temps réel Unix
        START_INSTANT,              // instant de démarrage du scénario
        ACCELERATION_FACTOR});      // facteur d'acccélération
```

Ensuite, chaque composant qui veut se synchroniser sur cette horloge va devoir se connecter au serveur d'horloge, récupérer une copie de l'horloge en utilisant son URI puis l'utiliser pour faire sa planification :

```
ClocksServerOutboundPort p = new ClocksServerOutboundPort(this);
p.publishPort();
this.doPortConnection(
        p.getPortURI(),
        ClocksServer.STANDARD_INBOUNDPORT_URI,
        ClocksServerConnector.class.getCanonicalName());
AcceleratedClock ac = p.getClock(TEST_CLOCK_URI);
this.doPortDisconnection(p.getPortURI());
p.unpublishPort();
p.destroyPort();
// toujours faire waitUntilStart avant d'utiliser l'horloge pour
// calculer des moments et instants
ac.waitUntilStart();
```

A.4 Précautions à prendre

La précision de l'ordonnancement sur un ordinateur utilisant un système d'exploitation comme Unix n'est pas très élevée. Par expérience sur mon Unix, la précision est d'environ 10 millisecondes en pratique. Cela veut dire que si deux composants appellent `scheduletask` au même moment avec des délais différant de moins de 10 millisecondes, on ne peut plus prévoir dans quel ordre les deux tâches seront exécutées.

En pratique, cela veut dire que le délai en temps réel séparant deux tâches à ordonnancer ne doit pas être inférieur à 10 millisecondes pour obtenir l'ordre que l'on souhaite dans un scénario. Et bien sûr, il faut prendre en compte le facteur d'accélération qui aboutit à réduire les délais en temps réel entre les actions planifiées à deux instants. Il faut donc éviter de produire des scénarios où le délai entre deux actions selon la référence temporelle des instants divisé par le facteur d'accélération ne donne une valeur inférieure à 10 millisecondes. Par exemple, deux actions planifiées à une minute d'intervalle dans le temps simulé avec un facteur d'accélération de 600 vont se produire à un dixème de secondes de distance (60 secondes/600) en temps réel, donc ne devrait pas poser de problème. Par contre, avec un facteur d'accélération de 6000, on tomberait à une différence de 10 millisecondes, donc pouvant potentiellement poser des problèmes.

Pour votre culture personnelle, sachez qu'il existe des horloges matérielles et des systèmes d'exploitation beaucoup plus précis dans l'ordonnancement des tâches, c'est à dire pouvant descendre au millionnième voire au milliardième de seconde. Ce sont des systèmes d'exploitation temps réel utilisés dans les applications temps réel.