Implantation d'un système pair-à-pair de gestion de contenu en BCM4Java

Cahier des charges — version 1.0

23 janvier 2023

Résumé

Ce document définit le cahier des charges du projet de l'unité d'enseignement Composants (CPS) pour son instance 2023. 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 est plus précisément d'implanter un prototype simplifié de système de gestion de contenu en pair-à-pair. Les applications de ce type d'architecture logicielle ont été très populaires il y a une vingtaine d'années pour la partage de fichiers musicaux. Aujourd'hui, on les retrouve dans toutes sortes d'applications, comme les jeux en ligne massivement répartis. Pour ce projet, il ne s'agit évidemment pas de construire une application réaliste. L'objectif sera plutôt de comprendre les notions sous-jacentes à ces architectures grâce à un exemple à la fois intéressant et concret. De nombreuses simplifications seront nécessairement faites pour rendre l'objectif atteignable en un temps restreint.

1 Généralités

> *Le pair-à-pair ou système pair à pair (en anglais peer-to-peer, souvent abrégé « P2P ») est un modèle d'échange en réseau où chaque entité est à la fois client et serveur, contrairement au modèle client-serveur. Les termes « pair », « nœud » et « utilisateur » sont généralement utilisés pour désigner les entités composant un tel système. Un système pair à pair peut être partiellement centralisé (une partie de l'échange passe par un serveur central intermédiaire) ou totalement décentralisé (les connexions se font entre participants sans infrastructure particulière). Il peut servir entre autres au partage de fichier, au calcul distribué ou à la communication.*
>
> Pair-à-pair, Wikipedia, consulté le 13/12/2022

Les systèmes pairs-à-pairs sont des systèmes répartis dont le principe fondamental est le rejet (quasi-total) d'entités centralisées, ce qui permet d'envisager le passage à de très grandes échelles. Dans le grand public, les systèmes pairs-à-pairs ont d'abord été connus par leur application au partage de fichiers, dont principalement les fichiers musicaux (comme Napster), qui a participé de manière importante à l'essor initial de l'Internet chez les particuliers. Aujourd'hui, les architectures pairs-à-pairs se retrouvent dans de nombreuses applications massivement réparties comme les jeux en ligne, la communication (téléphonie sur IP) ou encore le calcul réparti en science participative (projet SETI@home, par exemple).

Le projet consiste donc à implanter un système de gestion de contenu pair-à-pair appliqué au partage de fichiers. Toutefois, il ne s'agira que d'une simulation, puisque la démonstration se fera dans le cadre d'une application BCM4Java déployée principalement sur un seul ordinateur voire sur quelques ordinateurs si les ressources disponibles le permettent. L'idée générale n'est pas d'arriver à une implantation réellement utilisable en pratique mais plutôt de comprendre les principes d'une implantation pair-à-pair ainsi que leurs principaux choix de mise en œuvre et leurs conséquences, en particulier en termes de performance.

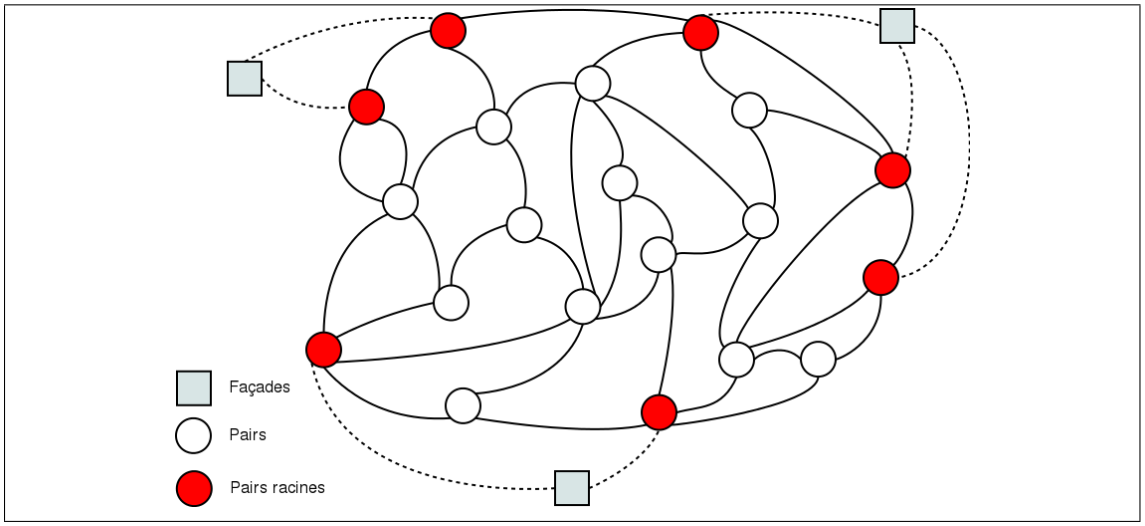

FIGURE 1 – Réseau logique : façades, nœuds pairs et liens.

2 Un système de gestion de contenu simplifié en BCM4Java

2.1 Réseau logique

Dans tout système pair-à-pair, un graphe d'interconnexion des nœuds pairs participants à l'application est créé pour assurer les communications. On appelle ce graphe un *réseau logique*, car la communication réelle va effectivement emprunter un réseau « physique » [1] sous-jacent. Mais en termes d'échanges pairs-à-pairs, les messages seront « routés » d'un nœud pair à un autre via ce réseau logique. Au-dessus de la couche d'interconnexion « physique » sous-jacente va donc se créer un réseau « virtuel » où un relativement petit nombre de connexions directes (liens dans le graphe) seront établies entre nœuds pairs qui deviendront alors *voisins* dans ce réseau.

De ce fait, seuls des nœuds voisins peuvent communiquer *directement* les uns avec les autres. Quand deux nœuds pairs qui ne sont pas voisins veulent communiquer l'un avec l'autre, il faudra faire plusieurs sauts de voisin en voisin jusqu'à atteindre la destination. Le graphe doit donc être suffisamment connecté pour qu'un chemin existe toujours entre deux nœuds pairs quelconques, mais le nombre de voisins de chaque nœud pair doit être suffisamment petit pour que leur gestion ne demande pas une quantité croissante de ressources en fonction de la taille totale du graphe.

La figure 1 illustre un réseau logique où apparaissent les nœuds pairs et les liens entre ces derniers qui définissent des canaux de communication directs entre voisins. Ce réseau logique, comme expliqué précédemment, se projette sur une architecture physique où, dans les applications réelles, les nœuds sont hébergés par des ordinateurs et les liens se réalisent par des connexions Ethernet et Internet. Dans le projet, cette projection se fera plutôt sur des composants BCM4Java (nœuds) et leurs interconnexions (liens) via ports et connecteurs.

Découverte du système

Dans une vision purement pair-à-pair, tous les nœuds dans le réseau logique sont des pairs, c'est à dire qu'ils ont tous exactement la même forme et le même rôle. De plus, le réseau n'est pas créé à l'avance par des moyens externes mais se construit lui-même par insertion et retrait dynamique de nœuds. Se pose alors la question de savoir comment se joindre à un tel réseau sans connaître les identifiants (adresse IP, par exemple) des nœuds existants. Les solutions ne sont pas si simples. Si on désire rester purement pair-à-pair, il faut « découvrir » les adresses des nœuds existants sans aucune information préalable, ce qui suppose à la fois la capacité à forger des adresses parmi les adresses possibles [2] et la capacité à tester les adresses forgées pour filtrer celles qui correspondent

1. Dans les applications du monde réel, c'est l'Internet qui est utilisé, et par extension le WWW qui n'est qu'un protocole particulier au-dessus d'Internet. Dans le cadre du projet, les interconnexions seront des connexions entre ports via de connecteurs entre composants BCM4Java, comme nous le verrons plus loin.

2. Par exemple, sous IP, les adresses se forment d'une adresse IP de 12 chiffres décimaux et d'un numéro de port entre 0 et 65535 sur lequel le processus répond. En pire cas, forger une adresse consiste donc à générer une adresse IP et un numéro de port. Une recherche exhaustive requiert de balayer tous les couples adresse IP/numéro

à des nœuds pairs existants de celles qui ne mènent à rien car elles ne sont pas utilisées.

Une alternative consiste à utiliser des modèles architecturaux mixtes, soit avec un frontal centralisé, porte d'entrée unique à l'application, soit des systèmes hybrides où existent des nœuds distincts *façades*, accessibles selon les méthodes standards (un identifiant connu par les DNS, par exemple). Le nombre de nœuds façades dépend de plusieurs paramètres dont la charge client à soutenir et la robustesse aux pannes de ces nœuds ayant un rôle critique dans l'application. La figure 1 illustre cette option, qui sera choisie dans le cadre du projet.

Pour limiter le nombre de nœuds pairs auxquels ils sont connectés, chaque nœud façade ne doit être connecté qu'à un nombre limité de nœuds pairs parmi ceux qui se sont insérés via lui, comme on peut le voir à la figure 1. Le nombre de ces nœuds dits *racines* est fixé de telle manière à ce qu'ils soient suffisants pour ne pas perdre la connexion au réseau lors des départs de nœuds [3] mais sans croître proportionnellement au nombre total de nœuds dans le réseau sinon le passage à l'échelle serait impossible.

Insertion de nouveaux nœuds

Informellement, quand un nouveau nœud [4] souhaite s'insérer comme pair dans le réseau logique, il va se créer une adresse valide puis il va contacter un des nœuds façades pour lui demander de joindre le réseau. À cette demande, le nœud façade va répondre par un ensemble d'adresses de nœuds pairs valides auxquels le nouveau nœud va se connecter pour s'établir en tant que leur voisin. Le nombre d'adresses fournies dépend du degré de connectivité entre nœuds pairs que le réseau souhaite obtenir. [5]

Pour fournir un ensemble de nœuds voisins potentiels à un nouveau nœud souhaitant s'insérer, deux méthodes seront implantées :

1. Dans un premier temps (étape 1), un seul nœud façade sera utilisé et il va mémoriser toutes les adresse de nœuds existants dans le réseau logique dans une structures de données qui lui est propre. Grâce à cela, il pourra sélectionner n adresses valides au hasard parmi les adresses stockées et les retourner au nouveau nœud.
2. Dans un deuxième temps (étape 2), pour assurer le passage à l'échelle, une approche alternative sera implantée qui se repose sur le réseau logique lui-même en le sondant pour trouver des nœuds pairs dont les adresses seront retournées au demandeur (voir §3.3).

Départ des nœuds

Lorsqu'un nœud pair quitte le réseau normalement (sans qu'il tombe en panne), il doit se déconnecter de tous ses voisins et il peut éventuellement informer le nœud façade de son départ. Dans la première solution pour l'insertion donnée précédemment, le nœud façade pourra supprimer ce nœud de ses propres structures de données. Cette information sert aussi à mesurer ou estimer le nombre de nœuds actuellement présents dans le réseau, ce qui est utile à certains algorithmes.

2.2 Gestion du contenus

L'application qui illustre ici l'approche pair-à-pair est la gestion de contenu représenté par des fichiers musicaux. Les nœuds pairs vont donc stocker des fichiers (partie qui ne sera pas réalisée en pratique, pour simplifier le projet) et les clients pourront rechercher des contenus en lançant des requêtes à partir des nœuds façades. Les nœuds pairs mémorisent des descripteurs de l'ensemble des fichiers qu'ils offrent. Ces descripteurs contiennent le titre du morceau, le titre de l'album, l'ensemble des interprètes, l'ensemble des compositeurs, l'adresse du nœud pair stockant le fichier et la taille de ce dernier.

---

de port possibles pour les filtrer. Filtrer va consister à tenter de contacter un ordinateur à l'adresse IP forgée et communiquer avec un processus sur le numéro de port forgé en utilisant le protocole défini pour cette application pour voir si une réponse est obtenue. Une telle approche est coûteuse en nombre de tentatives de connexions à faire, et peut bien entendu se heurter à des blocages par les politiques de sécurité des systèmes et des réseaux.

3. Lors du départ d'un nœud racine, il sera possible de le remplacer par un autre nœud par sondage, voir plus loin.

4. Dans les applications réelles, il faut généralement installer un logiciel sur son ordinateur qui en se lançant aura toutes les caractéristiques d'un nœud pair, dont une adresse valide, puis il pourra contacter un des nœuds façades pour lui passer une requête d'insertion.

5. Un contrôle du nombre de voisins est possible pour les nœuds existants qui peuvent refuser la connexion du nouveau nœud mais leur fournir l'adresse d'un de leurs propres voisins pour tenter à nouveau une connexion. Cette approche plus complexe ne sera toutefois pas abordée dans le projet.

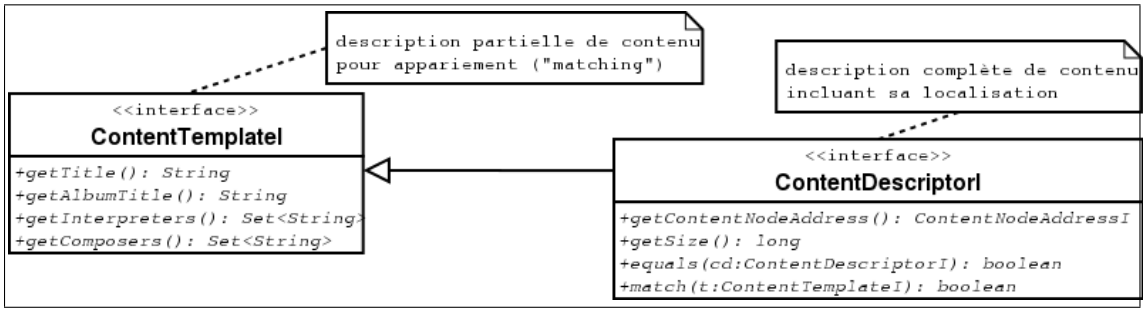

FIGURE 2 – Interfaces utilisées pour représenter le contenu et les requêtes.

Pour faire une requête, les clients utilisent un patron (« template ») qui n'est qu'un descripteur partiel de contenu où seule une partie des informations propres au morceau est fournie. Les nœuds pairs vont conserver des descripteurs complets de leurs contenus de manière à répondre aux requêtes des clients. La figure 2 montre deux interfaces qui devront être utilisées dans le projet pour gérer les contenus. L'interface [6] `ContentTemplateI` déclare les éléments pouvant apparaître dans un patron et qui devront apparaître dans les descripteurs complets. L'interface `ContentDescriptorI` étend la précédente pour ajouter l'adresse du nœud pair (voir plus loin) détenant le fichier correspondant et la taille de ce fichier. Lorsqu'un client cherche un contenu, il créera un patron dans lequel il pourra ne mettre qu'une partie de l'information (que le titre ou que l'un des interprètes, par exemple) qui va former sa « requête ». Pour ce projet, les nœuds pairs vont se joindre au réseau en ayant déjà des contenus à proposer, et donc ils auront déjà créé et stocké les descripteurs complets de tous leurs contenus.

Pour la recherche, l'interface `ContentDescriptorI` déclare une méthode `match` pour apparier un patron à un descripteur de contenu. Utilisant cette méthode `match`, la recherche de contenu à travers le réseau en elle-même se fait selon deux modes :

1. Recherche d'un unique contenu où le premier descripteur de contenu s'appariant à un patron sera retourné.
2. Recherche de plusieurs contenus où un certains nombre de contenus trouvés seront retournés. [7]

Lors d'une requête, le client s'adresse initialement à un nœud façade avec un patron de contenu qu'il recherche. Pour trouver ce contenu, les requêtes vont devoir être passées de voisins en voisins à travers le réseau logique afin de trouver un nœud pair qui détient un ou des contenus correspondants. C'est dans l'algorithme de parcours du réseau logique que prend forme l'idée maîtresse du pair-à-pair : le nœud façade va envoyer la requête vers un ou plusieurs nœuds pairs auxquels il est directement connecté puis, à la réception d'une requête chaque nœud pair va :

— soit ne pas avoir de contenu correspondant auquel cas il va réexpédier la requête à un ou plusieurs de ses voisins choisis aléatoirement ;
— soit répondre à la requête avec un contenu qu'il possède (pour une recherche unique) ou encore réexpédier la requête à un ou plusieurs de ses voisins choisis aléatoirement en ajoutant le ou les contenus qu'il possède dans la réponse à retourner (recherche multiple).

Ce genre de parcours s'appelle une *marche aléatoire* dans le réseau logique. Le problème avec ce genre de parcours est d'arrêter la propagation des requêtes à travers le réseau car il pourrait arriver qu'une requête circule indéfiniment dans le réseau si on ne trouve pas de contenu, par exemple. L'idée généralement utilisée pour ce faire est de limiter le nombre de sauts de pair en pair qu'une requête peut faire et d'arrêter sa propagation lorsque ce nombre est atteint. Donc, une troisième alternative s'ajoute aux deux précédentes :

— soit, lorsque le nombre de sauts maximal est atteint, retourner un descripteur de contenu, ou « `null` » si rien n'a été trouvé dans le dernier nœud atteint (recherche unique), soit les descripteurs de contenus déjà trouvés, ou « `null` » si rien n'a été trouvé au fil des nœuds traversés (recherche multiple).

6. Dans les diagrammes UML, le stéréotype `«interface»` indique une interface Java alors que le stéréotype `«component interface»` indique une interface de composants BCM. De plus, les interfaces Java ont un nom qui se termine par `I` alors que les interfaces de compsoants se terminent par `CI`.

7. Il ne s'agit pas de faire une recherche exhaustive mais une recherche partielle, car une recherche exhaustive demanderait une quantité de ressources croissant très vite en fonction du nombre de nœuds pairs dans le réseau.

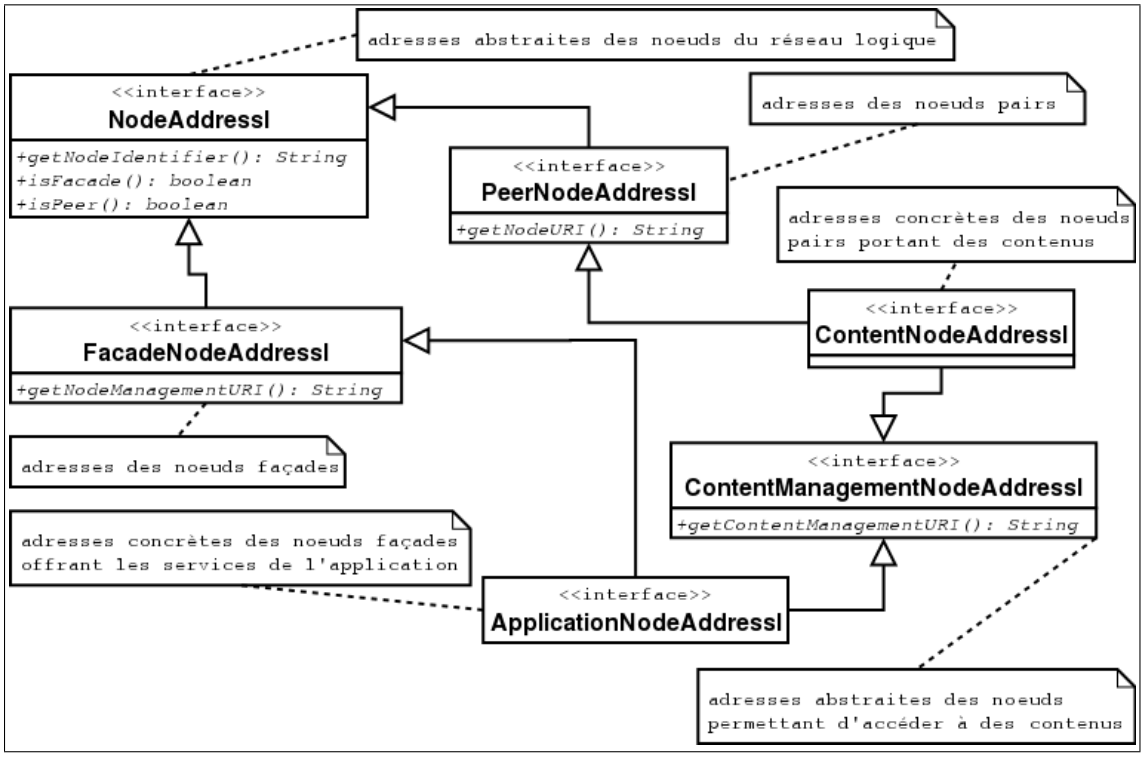

FIGURE 3 – Interfaces utilisées pour définir les adresses des composants façades et pairs en tant que participants au réseau logique et en tant que participants à l'application.

2.3 Première architecture à base de composants

Pour spécifier plus précisément le système, il faut entrer dans la description de l'achitecture logicielle à base de composants. À cette étape du projet, quatre comportements sont nécessaires :

1. Le comportement des nœuds façades comme participant au réseau logique.
2. Le comportement des nœuds pairs comme participant au réseau logique.
3. Le comportement des nœuds façades comme participant à l'application de gestion de contenus.
4. Le comportement des nœuds pairs comme participant à l'application de gestion de contenus.

Dans un premier temps, deux types de composants seront utilisés pour implanter directement ces comportements code de ces composants : les composants façades et les composants pairs.

Par rapport au réseau logique, les nœuds façades sont des entités spécifiques et donc leur comportement en temps que participant au réseau restera implanté directement comme du code de ces composants. Par contre, le comportement des nœuds façades comme application de gestion de contenus et celui des nœuds pairs à la fois comme participant au réseau et comme gestionnaire de contenus ne sont pas spécifiques. Différentes applications pourraient être déployées en pair-à-pair où les composants ont d'autres rôles dépendants du contexte, en plus de ceux d'être nœuds façades ou nœud pairs d'une application pair-à-pair. Un peu plus tard dans le projet (voir §3.1), ces rôles ne seront plus implantés directement comme du code dans les composants mais plutôt par des greffons (« *plug-ins* ») réutilisables qui pourront être ajoutés à des composants quelconques pour leur attribuer ces rôles. L'idée sous-jacente est d'employer de bonnes approches de conception logicielle permettant une meilleure réutilisation logicielle.

Gestion du réseau logique dans les composants

Pour gérer le réseau logique, les nœuds vont être identifiés par des adresses définies à partir des interfaces fournies à la figure 3. Les nœuds sont représentés par des composants BCM4Java donc ces interfaces sont conçues de telle façon à donner pour chaque type de nœuds les informations nécessaires pour se connecter au composant correspondant en BCM4Java :

— `NodeAddressI` factorise les informations communes aux composants d'un réseau logique, c'est à dire leurs identifiants uniques (ici, ce seront les URIs des ports de réflexion entrants

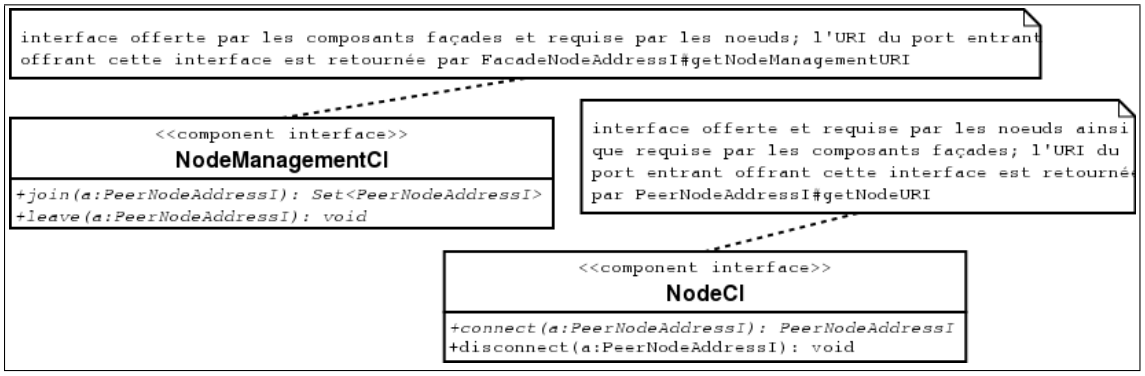

FIGURE 4 – Interfaces de composants offertes par les composants façades et pairs en tant que participants au réseau logique.

des composants) ; les méthodes booléennes servent à filtrer les types de nœuds, façades et pairs, que ces composants représentent.

- `FacadeNodeAddressI` définit les informations utiles pour les composants façades en tant que participant au réseau logique : l'URI du port entrant offrant les services pour se joindre au réseau logique ou le quitter.
- `PeerNodeAddressI` définit les informations utiles pour les composants pairs en tant que participant au réseau logique : l'URI du port entrant permettant de se connecter à ce pair pour construire le réseau logique.
- `ContentManagementNodeAddressI` factorise les informations utiles pour les composants en tant que participants à l'application de gestion de contenus : l'URI du port entrant permettant de faire des requêtes de recherche de contenus.
- `ApplicationNodeAddressI` cumule les informations utiles pour les composants façades vus comme fournisseurs des services de l'application de gestion de contenus en pair-à-pair.
- `ContentNodeAddressI` cumule les informations utiles pour les composants pairs vus comme gestionnaires de contenus en pair-à-pair.

Notez que cette hiérarchie d'interfaces vise à ne mettre dans les adresses que ce qui leur est strictement nécessaire dans l'espace le plus restreint possible. De plus, elle respecte une séparation des préoccupations qui permettrait de réutiliser les interfaces `NodeAddressI`, `FacadeNodeAddressI` et `PeerNodeAddressI` dans une autre application.

En tant que composants, ceux-ci sont interconnectés par des connexions BCM4Java selon des interfaces de composants. Les composants façades du réseau logique offrent l'interface de composants `NodeManagementCI` donnée à la figure 4. Les services déclarés par cette interfaces sont :

`join :` prend en paramètre l'adresse du nœud pair à insérer dans le réseau et retourne à ce dernier un ensemble d'adresses de nœuds pairs auxquels il pourra se connecter comme nouveau voisin.

`leave :` prend en paramètre l'adresse du nœud pair qui quitte le réseau.

Le composants jouant le rôle de nœuds pairs offrent l'interface `NodeCI` avec les services déclarés suivants :

`connect :` reçoit en paramètre l'adresse d'un nouveau pair qui souhaite se connecter à lui en tant que voisin, s'y connecte et lui retourne sa propre adresse comme confirmation.

`disconnect :` reçoit en paramètre l'adresse d'un voisin et se déconnecte de lui.

Gestion du contenu

Pour la gestion de contenu, l'interface `ContentManagementCI` de la figure 5 définit les services qui seront offerts à la fois par les nœuds façades aux clients et par les nœuds pairs pour propage les requêtes dans le réseau logique. Deux méthodes sont définies :

`find :` prend en paramètres un patron de contenu et un nombre maximal de sauts autorisés entre pairs pour retourner le descripteur du contenu s'appariant au patron trouvé en parcourant les pairs ; la façade envoie cette requête à un ou plusieurs de ses racines alors qu'un pair soit retourne le descripteur d'un contenu qu'il détient ou, à défaut décrémente le nombre de sauts autorisés et si ce nombre n'est pas zéro, renvoie cette requête à un ou plusieurs des ses voisins.

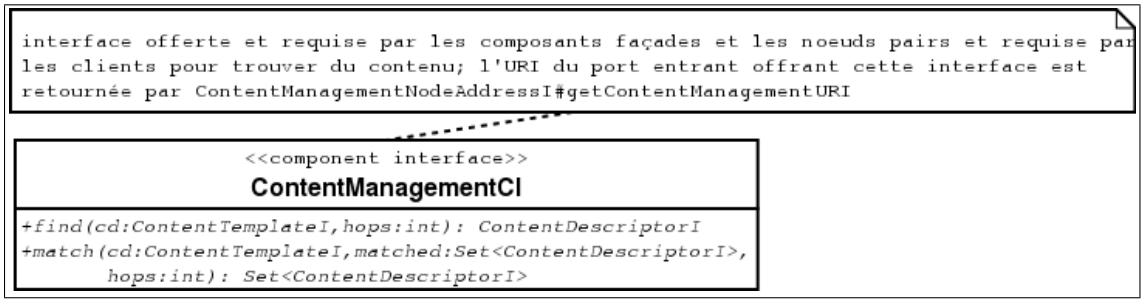

FIGURE 5 – Interfaces utilisées par les composants façades et pairs en tant que participants à l'application de gestion de contenus.

`match :` prend en paramètres un patron de contenu, un ensemble de descripteurs de contenus déjà trouvés et un nombre maximal de sauts autorisés entre pairs pour collecter le résultat de la requête ; la façade envoie cette requête à un ou plusieurs de ses racines avec un ensemble de descripteurs vide au départ alors qu'un pair ajoute d'abord les descripteurs de contenus appariés qu'il détient puis décrémente le nombre de sauts autorisés et s'il est zéro, il retourne l'ensemble obtenu ou s'il n'est pas zéro, renvoie cette requête à un de ses voisins.

L'impact du nombre de coisins auxquels `find` est renvoyé ainsi que celui du nombre de sauts autorisés sur le fonctionnement et la performance du système seront étudiés durant la dernière partie du projet.

2.4 Démonstration et simulation

Après la mise en œuvre des éléments présentés depuis le début de cette section, il sera possible de faire une première démonstration du système de gestion de contenu en pair-à-pair. Cette démonstration sera la pièce maîtresse de la soutenance de mi-semestre et dont l'état d'avancement sera évalué lors de l'audit 1. Pour avoir une démonstration effective, il faut encore répondre à deux points :

— Comment fournir les données (contenus) dans les nœuds pairs ?
— Comment fournir des patrons de requête pour le test d'intégration du système ?

Il serait bien sûr possible de créer les données et les requêtes manuellement, en programmant. Cette option, peut-être valide à cette étape, sera un boulet lorsque plus tard dans le projet il faudra augmenter sensiblement le nombre de nœuds pairs et de requêtes à effectuer.

Les scénarios de tests seront donc construits de manière plus générique en préparant les données dans des fichiers. Voici les étapes à suivre pour exécuter le test d'intégration prévu lors de la soutenance de mi-semestre :

1. Les composants pairs sont créés en leur passant un nom de fichier à lire avec un format prédéfini qui contient les descripteurs de contenus que ces composants vont détenir pendant le test.
2. Ces composants lisent le contenu de ce fichier lors de leur lancement et rangent les descripteurs de contenus dans des structures de données propres.
3. Le composant façade est créé en lui passant le nom d'un fichier à lire avec un format prédéfini qui contient les patrons de contenus qui vont servir à faire des requêtes.
4. Ce composant lit le contenu de ce fichier lors de son lancement et range les patrons de contenus dans des structures de données propres en attendant de lancer les requêtes.
5. Une fois leurs contenus récupérés, les composants pairs vont s'adresser au composant façade pour se joindre au réseau.
6. Chaque composant pair se connecte ensuite aux composants pairs dont les adresses leurs seront fournies pour qu'ils deviennent leur voisin.
7. Enfin, le composant façade lance les requêtes prévues et écrit les contenus récupérés dans sa fenêtre de trace.

3 Réutilisation et passage à l'échelle

Sur la base des développements de la première étape, la deuxième étape va apporter des améliorations visant à renforcer la réutilisation logicielle et à augmenter la performance globale du système. De plus, les démonstrations vont monter progressivement en charge pour évaluer les progrès obtenus.

3.1 Passage aux greffons

La première tâche à réaliser à l'étape 2 est le passage aux greffons pour implanter les rôles des composants :

1. Un greffon pour la gestion du réseau logique pour les composants pairs correspondant aux fonctionnalités définies par l'interface `NodeCI`.
2. Un greffon pour la gestion de contenus pour les composants pairs étendant le greffon de gestion du réseau logique correspondant aux fonctionnalités définies par l'interface `ContentManagementCI`.
3. Un greffon pour la gestion de contenus pour les composants façades correspondant aux fonctionnalités définies par l'interface `FacadeContentManagementCI`.

L'objectif sera de regrouper au sein de chaque greffon toutes les fonctionnalités liées à l'interface correspondante de même que les comportements associés (comme le fait de se joindre au réseau pour les composants pairs).

3.2 Gestion de la performance au niveau des composants

Dans un système pair-à-pair, de nombreuses opérations doivent se faire en même temps, à la fois pour gérer les demandes d'insertion et de départ du réseau logique de même que les requêtes qui arrivent des clients.

Passage en appels asynchrones

L'un des premiers points limitant la performance de l'implantation de l'étape 1 est le recours aux appels synchrones. Pour mémoire (voir le cours), les appels entre composants BCM4Java peuvent être de deux types en termes de gestion du temps :

1. Les appels synchrones, réalisés par des appels aux méthodes de la famille `handleRequest` dans les ports entrants, s'exécutent de telle manière que le composant appelant est bloqué jusqu'à ce que le composant appelé ait retourné le résultat de l'appel.
2. Les appels asynchrones, réalisés par des appels aux méthodes de la famille `runTask` dans les ports entrants, s'exécutent de telle manière que le composant appelant poursuit son exécution sans attendre le résultat et le composant appelé exécute la tâche sans retourner de résultat au composant appelant.

La conséquence des appels synchrones est que le *thread* du composant appelant est occupé tant que son appel n'a pas reçu de résultat, alors que dans les appels asynchrones, le *thread* du composant appelant peut être libéré en terminant la tâche qui a exécuté l'appel. Dans notre système, par exemple, un appel au service `find` par un client va demander au composant façade d'occuper un de ses *threads* tant qu'un composant pair n'aura pas trouvé un contenu et retourner son descripteur. De même, chacun des composants pairs par lesquels la requête `find` passe seront eux aussi bloqués avec un *thread* occupé à attendre le résultat du pair suivant auquel la requête a été relayée. En plus, le résultat va être retourné de proche en proche du composant ayant trouvé à celui qui l'a appelé et ainsi en cascade jusqu'à ce que le contrôle retourne au composant façade avant que ce dernier puisse retourner le résultat au client. Tout cela bloque et occupe beaucoup trop de ressources, ce qui limite énormément le nombre de requêtes que le système peut traiter par seconde.

Le passage à l'appel asynchrone permet de libérer les ressources au fur et à mesure où la requête est relayée à un composant suivant, mais alors se pose le problème du retour du résultat. Pour

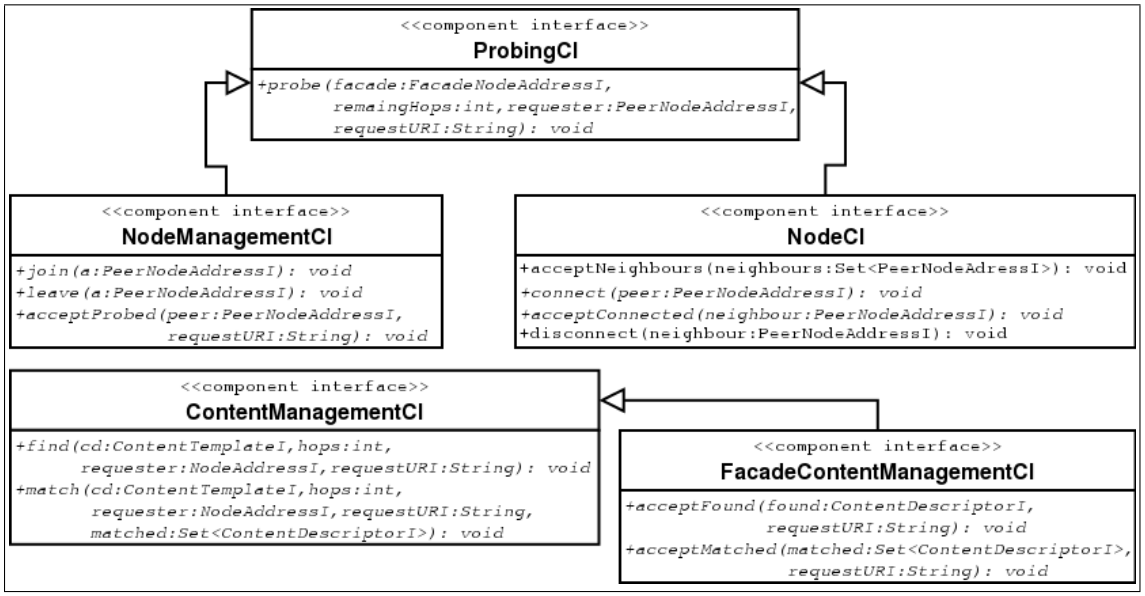

FIGURE 6 – Interfaces de composants utilisées par les composants façades et pairs dans la deuxième étape du projet.

résoudre cette difficulté, il faut utiliser un patron de conception [8] où l'appel asynchrone connaît le composant qui doit recevoir la réponse et ce dernier offre une méthode à appeler pour recevoir cette réponse. La figure 6 présente de nouvelles versions des interfaces `NodeCI`, `NodeManagementCI` et `ContentManagementCI` ainsi qu'une nouvelle interface `FacadeContentManagementCI` qui vont permettre de passer toutes les opérations longues en appels asynchrones.

Considérons la méthode `ContentManagementCI#find`. Cette méthode ne retourne plus de résultat mais prend en paramètre l'adresse du nœud façade qui a lancé la requête pour pouvoir lui retourner le résultat. L'appel à `find` avec ce nouveau paramètre passe de voisin en voisin jusqu'à ce qu'un composant pair trouve un contenu et alors ce composant appelle le composant façade pour lui retourner le résultat en utilisant la méthode `FacadeContentManagementCI#acceptFound`. Reste un petite difficulté : comment un composant façade, qui va recevoir plein de réponses à ses requêtes, pourra-t-il retrouver à quelle requête correspond quelle réponse ? Pour résoudre ce problème, il suffit d'ajouter un paramètre, une URI de requête qui sera générée à chaque requête par les composants façades et passée à la méthode `find` pour être utilisée en bout de chaîne par le composant pair qui répondra afin que le composant façade, recevant cette URI par la méthode `acceptFound`, retrouve la requête concernée et retourne le résultat au client correspondant.

Ce patron est utilisé pour les méthodes suivantes des interfaces présentées à la figure 6 :
- `ContentManagementCI#find` avec la méthode de retour de résultat `FacadeContentManagementCI#acceptFound` ;
- `ContentManagementCI#match` avec la méthode de retour de résultat `FacadeContentManagementCI#acceptMatched` ;
- `NodeManagementCI#join` avec la méthode de retour de résultat `NodeCI#acceptNeighbours` ;
- `NodeCI#connect` avec la méthode de retour de résultat `NodeCI#acceptConnected`.

Pour adopter cette approche, il faut donc remplacer les interfaces précédentes de même nom par celles définies à la figure 6. L'interface `FacadeContentManagementCI` est nouvelle ; elle devient l'interface offerte par les composants façades pour la gestion des contenus. La nouvelle version de l'interface `ContentManagementCI` devient l'interface de gestion de contenus des composants pairs. La nouvelle version de l'interface `NodeManagementCI` devient l'interface de gestion du réseau logique offerte par les composants façades alors que la nouvelle version de l'interface `NodeCI` fait de même pour les composants pairs.

Notez que la méthode `join` est appelée par les composants pairs sur les composants façades pour se joindre au réseau, d'où sa définition sur `NodeManagementCI`, alors que la méthode de retour

8. Ce patron de conception, qui ne fait pas partie des premiers patrons de conception historiques de la bande des quatre, vient de la programmation parallèle et s'inspire aussi de la notion de continuation, ici représentée par le compsaont devant recevoir la réponse et la méthode « accept » utilisée pour recevoir la réponse.

de résultat `acceptNeighbours` est appelée par les composants façades sur les composants pairs et donc définie dans `NodeCI`.

Introduction de pools de threads distincts

Les composants façades doivent répondre aux requêtes liées à la gestion du réseau (`join`, `leave`) en même temps qu'ils répondent aux requêtes de contenus (`find`, `match`). De même, les composants pairs doivent exécuter les requêtes de gestion de réseau de `NodeCI` en même temps qu'ils exécutent les requêtes de contenus de `ContentManagementCI`. Dans un système réel, il faut être en mesure d'exécuter le plus grand nombre possible de telles requêtes en utilisant le minimum de ressources tout en s'assurant que les différentes classes de requêtes ne se nuisent pas les unes les autres. Par exemple, il ne faut pas qu'un grand nombre de requêtes de contenus saturent les nœuds et empêchent les requêtes de gestion du réseau d'être exécutées.

Comme vu dans le cours, la gestion de l'exécution d'un grand nombre de requêtes est mieux réalisée en utilisant des *pools* de *threads* dont on peut adapter le nombre de *threads* à la charge (nombre de requêtes par seconde) et aux ressources disponibles (nombre total maximal de *threads* qu'il est raisonnable de créer sur l'ordinateur sous-jacent). Pour assurer une bonne répartition des ressources entre différentes classes de requêtes, il vaut mieux utiliser des *pools* de *threads* distincts pour chacune des classes.

Pour compléter le passage aux appels asynchrones, il s'agit donc de créer différents *pools* de *threads* dans les composants façades et pairs puis faire en sorte que les ports entrants dirigent bien les différentes classes de requêtes vers les bons *pools* de *threads*. La manière de répartir le nombre maximal total de *threads* alloués entre les *pools* de *threads* sera abordée un peu plus loin.

3.3 Gestion de la performance au niveau du réseau logique

Façades multiples

N'avoir qu'un unique nœud façade, comme dans l'étape 1, n'est pas tenable si on souhaite passer à l'échelle pour deux raisons :

1. Un seul nœud façade est impensable en pratique, car la charge des clients et le risque de perte de connexion au réseau si ce nœud tombait en panne serait trop grand.
2. Mémoriser toutes les adresses des nœuds pairs du réseau demanderait une quantité de ressources croissant linéairement avec la taille de celui-ci, et donc limiterait le passage à l'échelle

À cette étape, plusieurs composants façades seront donc créés et chacun recevra les demandes d'insertion des nœuds pairs qui leur seront affectés. Dans les déploiements mono-JVM, l'affectation des nœuds pairs aux nœuds façades sera faite arbitrairement par la simulation. Dans les exécution réparties sur plusieurs JVM, chaque JVM aura un composant façade et les nœuds pairs déployés dans la même JVM leurs seront affectés.

Insertion des nouveaux nœuds

Pour l'insertion de nouveaux nœuds pairs, il faut modifier la sélection des voisins potentiels car une sélection parmi une liste exhaustive connue n'est plus possible. L'alternative consiste à sonder aléatoirement le réseau logique par marche aléatoire à partir des racines pour trouver des voisins potentiels. Le processus de sondage se fait en envoyant aux nœuds racines des messages relayés de voisin en voisin jusqu'à ce qu'un nombre de sauts préalablement autorisés soit atteint, auquel cas l'adresse du dernier nœud atteint est retournée. Un nœud pair qui reçoit le message doit donc :

— soit le passer à un voisin choisi aléatoirement en décrémentant le nombre de sauts de 1 si ce dernier est plus grand que 0,
— soit retourner au nœud façade sa propre adresse.

Pour chaque sondage, le nœud façade sélectionne un de ses nœuds racines pour initier le processus. En lançant plusieurs sondages, on peut obtenir n adresses valides distinctes à retourner (à condition bien sûr que la taille du réseau soit supérieure à n).

Un nouveau problème apparaît toutefois lorsque plusieurs façades sont utilisées : si une façade sonde toujours les pairs accessibles à partir de ses racines et que ses racines font partie des pairs qu'il a déjà insérés, il n'y aura jamais d'interconnexions entre les pairs insérés par les différentes façades. Pour résoudre ce problème, les façades seront interconnectées entre elles pour se déléguer

les sondages les unes aux autres de manière à mélanger les voisins potentiels entre les pairs insérés par les différentes façades.

La méthode utilisée pour le sondage est déclarée par l'interface `ProbingCI` de la figure 6 et s'appelle `probe`. Elle reçoit en paramètre l'adresse du la façade qui a initié le sondage du réseau et un nombre de sauts restant à faire. Elle utilise aussi l'appel asynchrone avec retour de résultat par la méthode `NodeManagementCI#acceptProbed` car `probe` est appelée sur les composants façades et pairs, d'où l'héritage de `ProbeCI` par `NodeManagementCI` et `NodeCI`, mais le résultat n'est reçu que par les composants façades, d'où le fait que la méthode `acceptProbed` soit définie uniquement sur `NodeManagementCI`. Les implantations de `probe` dans les composants façades et pairs sont :

composant pair : si le nombre de sauts restants est supérieur à 0, le composant passe la main à un de ses voisins choisi aléatoirement, sinon il retourne sa propre adresse à la façade ayant initié le sondage ;

composant façade : soit elle délégue le sondage à une autre façade (selon une politique à déterminer), soit elle sonde ses propres pairs via ses racines selon le processus décrit précédemment.

Les composants façades seront donc interconnectés entre eux via l'interface `NodeManagementCI` pour utiliser ces délégations. Une organisation simple qu'on peut utiliser dans le projet de ces interconnexions consiste à lier les composants façades par un réseau logique en anneau.

Propagation des appels au sein du réseau logique

Dans une application pair-à-pair, la manière dont les requêtes sont propagées dans le réseau logique est un élément crucial de performance. En effet, une diffusion insuffisante des requêtes peut mener à un résultat insatisfaisant (pas de contenu trouvé alors qu'il en existe) mais une diffusion trop importe peut saturer les nœuds de requêtes à propager, ce qui utilise des quantités déraisonnables de ressources pouvant même aller jusqu'à paralyser l'ensemble de l'application.

Trois paramètres jouent un rôle crucial dans le comportement pair-à-pair au niveau du réseau logique :

— la connectivité du réseau, c'est à dire le nombre moyen v de voisins par nœuds, mais l'écart maximal et l'écart-type du nombre de voisins par nœud sontt aussi des propriétés importantes ;

— l'indice i de diffusion des messages ou requêtes, c'est à dire le nombre de voisins auxquels ils sont relayés par les nœuds pairs quand ceux-ci ne peuvent y répondre eux-mêmes ;

— la longueur l des chemins explorés par les messages ou requêtes, c'est à dire le nombre de sauts autorisés avant d'arrêter leur diffusion.

Une bonne connectivité, avec des nombres de voisins bien équilibrés entre les nœuds et des chemins dont la longueur rapportée à la taille du réseau est suffisante garantissent à la fois une bonne probabilité d'atteindre tous les nœuds et une équité dans leur parcours (la probabilité qu'un nœud soit visité par une requête dans un temps raisonnable). L'indice de diffusion i doit être choisi pour augmenter la probabilité de toucher tous les nœuds nécessaires mais sans engendrer une multiplication trop rapide du nombre de messages ou requêtes en cours d'exécution car on arrive à atteindre i^s nœuds à la dernière étape d'une requête effectuant s sauts [9]. Le nombre de sauts autorisés doit être choisi en fonction de la taille N du réseau logique et de sa connectivité, [10] par exemple $\log_c N$, mais il dépend également de l'indice de diffusion. Avec un indice de diffusion plus élevé, on peut adopter une longueur des chemins plus petite.

À cette étape, il faut d'abord modifier l'implantation des requêtes pour paramétrer globalement l'indice de diffusion (une constante publique globale suffira) puis la longueur des chemins devra être fixée grâce à l'estimation de la taille du réseau. Nous avons vu que l'estimation locale du nombre de nœuds insérés via une nœud façade spécifique peut se faire en conservant la différence

9. Avec un $i = 2$, deux nœuds appelés au départ appellent chacun deux nœuds puis ces quatres nøeuds atteints en appellent chacun deux, donc huit nœuds alors atteints, etc. ; le processus est donc exponentiel et ne s'arrête potentiellement que lorsque le nombre de sauts s autorisés est atteint. Notez qu'un indice rationnel est possible : par exemple, un indice 1,25 peut se réaliser en appelant un voisin trois fois sur quatre et deux voisins une fois sur quatre le long du chemin.

10. Sans entrer trop dans les détails, si un chemin hamiltonien existe dans le graphe, c'est à dire un chemin passant par tous les nœuds, alors le nombre de sauts autorisés devrait être au moins la longueur de ce chemin pour espérer un passage par tous les nœuds. Avec un graphe de connectivité c, on peut espérer avec une probabilité non nulle toucher chaque nœud avec des chemins de longueur $\log_c N$ et en multipliant ces chemins, par exemple par un indice de diffusion supérieur à 1.

entre le nombre de nœuds insérés et le nombre de nœuds ayant quitté le réseau. Dans le cas où plusieurs nœuds façades sont utilisés, il faut que ces derniers échangent régulièrement leurs estimations locales pour obtenir une estimation globale pour l'ensemble du réseau. Il faut donc ajouter tout ce qui permettra ces échanges aux composants façades (partie entièrement laissée à votre conception et implantation).

Optionnel : gestion du nombre de voisins par nœud

Pour la connectivité, c'est plus délicat. Elle est partiellement contrôlée par le nombre de voisins poten,tiels fournis aux nouveaux nœuds lors de leut insertion dans le réseau. Toutefois, l'attribution aléatoire des voisins aux nouveaux nœuds peut mener à des déséquilibres dans les nombres de voisins entre les nœuds lorsque certains nøeuds se trouvent à être choisis plus souvent que les autres. Pour rééquilibrer ces nombres, il faudrait pouvoir échanger des voisins entre les nœuds qui en ont le plus vers les nœuds qui en ont le moins. Faire cela optimalement demanderait que les façades connaissetn les nombres de voisins de tous les nœuds, ce qui mènerait à une solution qui ne passe pas à l'échelle. Pour passer à l'échelle, il faut plutôt aller vers des solutions non optimales. Deux possibilités peuvent être explorées :

1. Modifier l'algorithme de sondage pour retenir le nœud rencontré au fil des sauts de voisins en voisins qui a le moins de voisins. Il faut alors ajouter deux paramètres à la méthode `probe`, l'adresse du nœud rencontré ayant le moins de voisins et son nombre de voisins, le mettre à jour au passage de voisins en voisins puis renvoyer le dernier trouvé lorsque le nombre de sauts autorisés est atteint.
2. Modifier la connexion entre nœuds pour permettre à un nœud n_1 sollicité pour devenir le voisin d'un nouveau nœud n_2 de refuser cette connexion mais de présenter une alternative à n_2 c'est à dire l'un de ses propres voisins n_3 (de n_1 donc) qui a strictement moins de voisins que lui. Bien sûr, n_3 pourra à son tour refuser la conenxion et présenter un de ses propres voisins et ainsi de suite jusqu'à ce que l'on tombe sur un nœud dont tous les voisins ont plus de voisins que lui et alors la connexion de n_2 se fera à ce dernier.

On constate que ces protocoles sont significativement plus complexes que ceux mis en place précédemment. L'ajout de l'une ou l'autre de ces extensions est donc optionnel. La conception des interfaces et des algorithmes précis est entièrement à faire dans ce cas.

3.4 Optionnel : gestion de la performance au niveau applicatif

L'utilisation de nœuds pairs uniquement, c'est à dire qui ont tous exactement le même rôle, est une approche qui soulève des questions pratiques. Dans un système de gestion de contenus pair-à-pair, les nœuds peuvent être déployés sur des ordinateurs peu puissants qui ne sont par ailleurs pas toujours connectés, par exemple. D'autre part, les marches aléatoires parmi un très grand nombre de nœuds peuvent devenir très longues et avoir une probabilité de succès faible pour des contenus qui n'apparaissent qu'une seule fois dans le réseau.

Pour pallier ces limitations, plusieurs applications adoptent une solution moins puriste en ayant deux types de nœuds : les nœuds pairs de base qui détiennent tous les contenus offerts via leurs descripteurs associés et des nœuds *super-pairs* déployés sur des ordinateurs plus puissants et connectés en permanence (par exemple, sur le *cloud*. En plus de détenir leurs contenus propres, ces super-pairs vont aussi contenir les descripteurs de contenus d'autres nœuds pairs et ils sauront ainsi répondre à plus de requêtes en indiquant directement sur quel nœud pair il faut aller pour trouver un contenu. [11]

Plus précisément, les nœuds super-pairs vont mémoriser des descripteurs de contenus détenus par des nœuds pairs, donc des couples descripteur/adresse de nœud détenant ce contenu. Lorsqu'ils reçoivent une requête, les nœuds super-pairs peuvent y répondre directement si elle porte sur un contenu dont ils connaissent un nœud pair qui le détient. Et pour « peupler » ces super-pairs, les nœuds pairs vont diffuser leurs descripteurs de contenus avec leur adresse selon l'approche maintenant habituelle des marches aléatoires avec un nombre de sauts autorisés borné et lorsqu'un nœud super-pair est rencontré au long d'une telle marche aléatoire, ce dernier mémorise l'information qu'il reçoit ainsi et arrête la propagation de ce message.

11. Le réseau logique peut être construit de telle manière à favoriser le passage par ces super-pairs, mais cet aspect ne sera pas traité dans le projet.

Une difficulté apparaît dans ce schéma de fonctionnement lorsque les nœuds pairs quittent le réseau, car alors leurs entrées dans les super-pairs deviennent obsolètes. Pour éviter que les super-pairs finissent par contenir des informations qui ne sont plus valides, on peut faire en sorte que les supers-pairs envoient leur adresse aux nœuds pairs dont ils détiennent les descripteurs. Les nœuds pairs, mémorisant les super-pairs détenant des descripteurs de leurs contenus, pourront ainsi les avertir de nettoyer leurs structures de données quand ils quittent le réseau. Une alternative serait toutefois de faire en sorte que ce soit les nœuds super-pairs qui vérifient périodiquement, en tâche de fond, que les nœuds pairs dont ils détiennent des descripteurs de contenus sont toujours présents dans le réseau. Cette dernière méthode est plus robuste aux pannes des pairs qui quittent le réseau sans pouvoir notifier les super-pairs.

L'ajout de super-pairs demande de concevoir de nouvelles interfaces de composants, de nouveaux algorithmes et d'implanter les fonctionnalités nécessaires par de nouveaux greffons, travail relativement long et complexe. Cet ajout est donc optionnel dans le projet. La conception est aussi entièrement à faire dans ce cas.

3.5 Démonstration et analyse de performance

Pour compléter le projet, il faudra procéder à des tests d'intégration et des expérimentations sur le système. Par rapport à la première partie, le nombre de composants façades et pairs dans la démonstration finale sera de l'ordre de cinq pour les façades et de 50 pour les pairs. Comme pour la première partie du projet, des données vous seront fournies sous forme de fichiers à lire par les composants façades et pairs. Pour mener des expérimentations intéressantes, le nombre de requêtes à lancer sera également plus élevé.

Outre une exécution correcte en multi-JVM (ou à défaut en mono-JVM, voir plus loin), il est également demandé de procéder à des expérimentations sur les points suivants :

Le rythme d'exécution des requêtes : le délai entre le lancement des requêtes par les composants façades devra être progressivement accéléré et le délai entre le lancement de chaque requête et le retour du résultat mesuré pour voir comment se comporte la performance du système sous une charge croissante.

La longueur des marches aléatoires et l'indice de diffusion (requêtes `find`) : expérimenter avec différentes longueurs des marches aléatoires et différents indices de diffusion en mesurant à la fois le taux de succès des requêtes (contenu trouvé ou non) ainsi que le délai entre le lancement de chaque requête et le retour du résultat.

La répartition des *threads* entre les classes de services : en supposant que chaque composants peut disposer d'un total de 10 *threads*, mesurer les performances globales du système dans l'exécution des requêtes d'insertion dans le réseau et celles sur les contenus avec différentes répartitions des *threads* entre ces deux classes de services (par exemple, 2 *threads* pour les insertions et 8 *threads* pour les recherches de contenus, puis 5/5, etc.).

Les résultats de ces expérimentations seront présentés sous forme de tableaux lors de la présentation prévue pour la soutenance finale.

4 Déroulement du projet et résultats attendu

4.1 Audit 1 : prise en main de BCM4Java

L'audit 1 se déroulera lors de la quatrième séance de TME et il visera essentiellement à vérifier que vous avez réussi à prendre en main BCM4Java en démarrant le projet. En pratique, cela veut dire avoir implanté au moins les embryons d'un composant façade et des composants pairs avec leurs interfaces de composants, leurs ports et leurs connecteurs. Il faudra aussi qu'une exécution en mono-JVM fonctionne (quitte à connecter manuellement les composants pairs à ce stade), avec au moins deux pairs et un composant façade et qu'une requête de contenu soit exécutée.

En parallèle, il sera attendu que les éléments purement Java de l'étape 1 soient réalisés, comme la représentation des descripteurs et patrons de contenus (figure 2 et §2.2) ainsi que des adresses des nœuds (figure 3 et §2.3), avec des tests unitaires sous JUnit.

Pour cet audit, le principal critère d'évaluation sera le degré de réalisation des développements demandés ainsi que le taux de couverture des cas d'utilisation par vos cas de tests JUnit et leur exécution correcte.

4.2 Soutenance de mi-semestre : étape 1

L'ensemble de l'étape 1 fera 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).

Résultats attendus

Pour la seconde étape, une première version du système devra être développée, ce qui permettra de tester l'ensemble des fonctionnalités du système. Cette version utilisera minimalement les *threads* puisqu'essentiellement séquentialisée.

Pour l'intégration dans le modèle à composants BCM4Java, à partir du diagramme de la figure 3 et les interfaces que ce dernier déclare, le système doit définir :

— tous les ports entrants et sortants ainsi que les connecteurs correspondant aux interfaces de composants de la figure 3 ;
— tous les types de composants et les composants demandés : façades et pairs ;
— des scénarios de tests d'intégration sous la forme de déploiements en mono-JVM dûment tracés comportant un composant façade et 10 composants pairs.

Les tests d'intégration vont s'appuyer sur les données (descripteurs de contenus) et les requêtes qui vous seront fournies en amont. Des tests supplémentaires seront réalisés sur des données non fournies à l'avance.

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 et de leur bon fonctionnement en exécution, la qualité de l'ensemble de votre code sera également un critère d'évaluation.

4.3 Audit 2 : greffons, gestion du parallélisme et de la concurrence

L'audit 2 se déroulera pendant la neuvième séance de TME et il visera à démontrer l'atteinte des objectifs décrits dans les sections 3.1, c'est à dire le passage aux greffons, et 3.2, c'est à dire la gestion du parallélisme et de la concurrence.

Résultats attendus

Les résultats attendus pour l'audit 2 sont :

— les greffons de gestion du réseau logique pour les composants pairs ainsi que les greffons de gestion de contenus pour les composants façades et pairs intégrés et fonctionnels ;
— le passage aux appels aynchrones selon les nouvelles versions des interfaces de compsoants données à la figure 6 ;
— introduction de *pools* de *threads* distincts dans les composants ;
— les mêmes scénarios de tests d'intégration que ceux utilisés pour la soutenance de mi-semestre seront utilisés pour vérifier le bon fonctionnement des modifications apportées.

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.

4.4 Soutenance finale : étape 2

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 et 2).

Résultats attendus

Les résultats attendus pour l'étape 4 comportent :

— l'ensemble des développements demandés à la section 3 (sauf les développements optionnels s'ils sont laissés de côté) ;

— des scénarios de tests d'intégration sous la forme de déploiements en réparti (multi-JVM ou, à défaut et au minimum mais avec une pénalité sur la note finale, sur une seul machine virtuelle) comportant 5 composants façades et 50 composants pairs sur 5 machines virtuelles Java ;
— un plan d'expérimentations pour les évaluations de la performance avec des tableaux présentant les mesures obtenues.

Les tests d'intégration vont s'appuyer sur des données (descripteurs de contenus) et des requêtes qui vous seront fournies en amont. Des tests supplémentaires seront réalisés sur des données non fournies à l'avance.

Vous devrez rendre votre code en amont de la soutenance (voir les modalités décrites ci-après). Pour cette étape, en plus du taux de couverture des réalisations demandées et de leur bon fonctionnement en exécution, la qualité de la présentation orale de même que la qualité de l'ensemble de votre code et de votre documentation seront également des critères d'évaluation.

Développements optionnels

Les développements optionnels (gestion de la connectivité et ajout des super-pairs), sils sont réalisés, seront démontrés et évalués lors de la soutenance finale et dans l'évaluation du rendu de code.

5 Modalités générales de réalisation et calendrier des évaluations

— Le projet se fait **obligatoirement** en **équipe de deux étudiant·e·s**. 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 étudiant·e·s la formant au plus tard le **3 février 2023** à minuit.
— 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 15 minutes au maximum (par équipe) et se dérouleront lors des séances de TME. Le premier audit se tiendra pendant la séance 4 (**20 février 2023**) et il portera sur votre avancement de l'étape 1. Le second audit se déroulera lors de la séance 9 (**17 avril 2023**) et il portera sur votre avancement de l'étape 3. Ils compteront chacun pour 5% de la note finale de l'UE.
2. La **soutenance à mi-parcours** d'une durée de 20 minutes portera sur l'atteinte des objectifs des *deux premières étapes* du projet. Elle se tiendra pendant la semaine des premiers examens répartis du **13 au 17 mars 2023** selon un ordre de passage et des créneaux qui seront annoncés sur le site de l'UE. Elle comptera pour 35% de la note finale de l'UE. Elle comportera une discussion des réalisations pendant une quinzaine de minutes (devant l'écran sous Eclipse) et une courte démonstration de cinq minutes. Les rendus à mi-parcours se feront le **dimanche 12 mars 2023 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 du **11 au 17 mai 2023** (et plus probablement le lundi 15/05/2023 plus le mardi 16/05/2023 si nécessaire) selon un ordre de passage et des créneaux qui seront annoncés sur le site de l'UE. Elle comptera pour 55% de la note finale de l'UE. Elle comportera une présentation d'une douzaine de minutes (en utilisant des transparents), une discussion d'une dizaine de minutes également devant écran sur les réalisations suivie d'une démonstration. Les rendus finaux se feront le **dimanche 14 mai 2023 à minuit** [12] au plus tard (des pénalités de retard seront appliquées).

12. Attention, la remise pourrait être anticipée de quelques jours si l'épreuve CPS devait se tenir les 11 ou 12

— Lors des soutenances, les points suivants seront évalués :
  — le respect du cahier des charges et la qualité de votre programmation ;
  — l'exécution correcte de tests unitaires (JUnit pour les classes et les objets Java, scénario de tests pour les composants) ;
  — l'exécution correcte de tests d'intégration mettant en œuvre des composants de tous les types ;
  — la qualité et l'exécution correcte des scénarios de tests de performance, conçus pour mettre à l'épreuve les choix d'implantation versus la montée en charge ;
  — la qualité de votre code (votre code doit être commenté, être lisible – choix pertinents des identifiants, ... – et correctement présenté – indentation, ... – etc.) ;
  — la qualité de votre plan d'expérimentation et tests de performance (couverture de différents cas, isolation des effets pour identifier leurs impacts sur la performance globale, etc.) ;
  — la qualité de la documentation (vos rendus pour la soutenance finale devront inclure une *documentation Javadoc* des différents paquetages et classes de votre projet générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que votre répertoire `src`).
— Bien que les audits et les soutenances se fassent par équipe, l'évaluation reste à chaque fois **individuelle**. Lors des audits et des soutenances, *chaque étudiant·e* 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 de l'autre membre de son équipe.
— Lors des soutenances, **tout retard** non justifié d'un·e ou des membres de l'équipe de plus d'un tiers de la durée de la soutenance (7 minutes à la soutenance à mi-semestre, 10 minutes à la soutenance finale) entraînera une **absence** et une note de 0 attribuée au(x) membre(s) retardataire(s) pour l'ensemble de l'épreuve concernée (y compris le rendu préalable). Si un·e des membres d'une équipe arrive à l'heure ou avec un retard de moins d'un tiers de la durée de la soutenance, il·elle passera l'épreuve seul·e.
— 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 (n'inclure que les sources, 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 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 son binôme doit également le passer ; en cas de désaccord entre les deux membres du binôme, la demande de passage en solitaire prévaudra.

mai. Surveillez le site de l'UE pour les annonces afférentes.