Réseau de capteurs en BCM4Java
Cahier des charges du projet CPS 2024

Jacques Malenfant
professeur des universités

version 1.3 du 6/03/2024

Résumé

Ce document définit le cahier des charges du projet de l'unité d'enseignement Composants (CPS) pour son instance 2024. 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 supervision (*monitoring*) réparti fondé sur la notion de réseau de capteurs, très librement inspiré d'un prototype existant [BSVV20]. Le système s'appuie sur des entités possédant des attributs valués par des capteurs physiques (simulés pour les besoins du projet) pour exécuter de manière répartie des requêtes sur les valeurs courantes de ces capteurs. Dans le cadre de ce projet, il s'agit de reprendre cette idée en adoptant les composants BCM4Java comme entités définissant des nœuds d'un réseau de capteurs et s'appuyant sur d'autres composants BCM4Java comme clients lançant l'exécution de requêtes. La démonstration de ce projet se fera sur une application simulant un réseau de capteurs environnementaux.

1 Généralités

Les réseaux de capteurs servent à récolter des mesures de propriétés physiques sur une zone géographique donnée en répartissant un certain nombre de blocs de capteurs spécifiques (comme une série de capteurs météo, vitesse et direction du vent, des capteurs de chaleur et de fumée pour surveiller des forêts ou encore des capteurs biologiques sur le réseau d'égouts d'une ville). Outre la récolte des données, un réseau de capteurs sert à émettre des alertes (comme le départ d'un feu de forêt), parfois en déterminant d'autres propriétés (comme la direction du feu) à partir des données capteurs brutes. Les capteurs sont disséminés géographiquement et leur localisation est un aspect important pour l'identification des alertes. Pour transmettre les informations, les capteurs sont constitués en réseau, qui peut être basé sur un réseau physique filaire, c'est à dire un réseau sans fil externe accessible à tous les capteurs (comme par exemple le réseau 5G) ou un réseau sans fil interne auto-organisé par communication directe entre les capteurs suffisamment proches géographiquement les uns des autres pour rester dans leurs portées respectives.

L'architecture traditionnelle pour la détection d'alertes sur un réseau de capteurs consiste à construire les clients émetteurs des requêtes comme des contrôleurs centralisés qui récoltent directement toutes les données de tous les capteurs concernés puis les examine pour détecter les alertes. Cette approche traditionnelle présente plusieurs inconvénients lorsqu'on s'intéresse à des systèmes temps réel en réseau, qui plus est à grande échelle. Dans ces systèmes, le recueil des données par un contrôleur centralisé souffre des délais de transmission par réseau et de la quantité croissante de données à examiner à chaque cycle de détection. Si les contraintes de temps pour la détection des alertes sont fortes et à fréquence élevée, cette approche centralisée peut arriver à ses limites et ne plus pouvoir répondre dans les temps.

Une alternative consiste à décentraliser et à exécuter de manière asynchrone la vérification des conditions à surveiller. L'idée générale est la suivante. Conceptuellement, l'alerte correspond à une

requête combinant des conditions élémentaires portant sur des valeurs de capteurs précis, lesquelles vont s'exécuter localement sur chaque entité possédant ces capteurs. Par exemple, une condition peut être que le capteur de température indique plus de 50° et que le capteur de fumée indique un niveau supérieur à 3. Une requête peut contenir des conditions s'appliquant sur différents capteurs du même nœud mais le plus souvent aussi sur différents nœuds à proximité. L'exécution d'une requête sur le graphe des nœuds consiste donc à passer la requête en cours d'exécution d'abord au premier nœud concerné qui va évaluer localement les parties qui lui sont locales puis passer le reste du calcul (une forme de continuation) de la requête à un ou plusieurs nœuds suivants concernés et ainsi de suite jusqu'à ce que la requête ait été entièrement évaluée et alors le résultat peut être retourné à l'émetteur de la requête. Cette forme d'exécution peut donner lieu à de l'asynchronisme et à du parallélisme qui, en plus de la répartition sur les entités, permet d'espérer une durée totale d'exécution plus courte que l'approche centralisée.

Mise en réseau des capteurs et propagation des requêtes

Comme indiqué précédemment, dans un réseau de capteurs, la fonction communication utilise un réseau physique reliant les capteurs et les clients (eux toujours connectés à Internet). Dans le cas de réseaux de capteurs déployés dans des environnements où il n'y a pas d'infrastructure réseau, il est possible de créer un réseau à l'aide des émetteurs sans fil des capteurs en le reliant les uns avec les autres lorsqu'ils sont suffisamment proches géographiquement. On parle alors de création d'un *réseau logique* reliant directement les nœuds à leurs voisins géographique, avec une forme de routage simplifiée. Cette approche, du reste très intéressante à étudier, ne sera pas celle du projet de cette année.

Nous allons plutôt supposer que notre réseau de capteurs a accès à une infrastructure réseau déjà en place, par exemple un réseau sans fil de type 5G. Néanmoins, le maillage géographique des capteurs doit être restitué, car il peut êtte utilisé dans les requêtes. Considérons le scénario suivant :

— Le superviseur envoie régulièrement aux nœuds une requête à exécuter individuellement portant sur les capteurs de température et de fumée pour détecter le départ potentiel d'un feu de forêt.
— Lorsqu'un nœud renvoie un résultat positif à la requête précédente, une nouvelle requête est envoyée pour exécution locale sur ce nœud afin de récupérer les valeurs de direction et vitesse du vent.
— En fonction de la direction du vent, le superviseur envoie une nouvelle requête répartie pour évaluer l'étendue du feu dans la direction du vent. La requête va donc évaluer les conditions de détection du feu de forêt en commençant par le nœud où il a été initialement détecté puis sur les voisins dans la direction donnée. Les voisins pourront transmettre la requête à leurs propres voisins et ainsi de suite jusqu'à ce que l'on atteigne un critère d'arrêt, soit un nombre maximal de sauts, soit une distance géographique maximale, depuis le nœud ayant initialement reçu la requête.

À la création du réseau, les capteurs vont donc devoir identifier leurs voisins de manière à se connecter à eux pour pouvoir ensuite leur passer des requêtes. Dans un réseau de capteurs utilisant la communication sans fil, les voisins se détectent et se connectent via le protocole du réseau sans fil, comme lorsque vous connectez un accessoire en Bluetooth à votre téléphone portable. Dans le cadre du projet, nous allons simuler cette fonctionnalité en utilisant un composant agissant comme registre global des nœuds auprès duquel ceux-ci vont devoir s'enregistrer. Lorsqu'un nœud se joint au réseau (on vient de l'allumer ou il retrouve un niveau de batterie suffisant grâce à ses panneaux solaires, par exemple), il va contacter directement le registre en lui fournissant des informations sur lui (identité, position, portée de ses émetteurs et informations de connexion dans le cas de ce projet). En retour, le registre va lui fournir une liste des informations sur les nœud qui lui sont directement adjacents ou voisins géographiquement. À la réception de ces informations, le nouveau nœud se connecte à ses voisins et les notifie de cette connexion afin qu'ils se connectent à leur tour à lui pour former le maillage géographique du réseau de capteurs.

Par exemple, pour le réseau de capteurs illustré à la figure 1, supposons que le nœud n7 se joint au réseau alors que tous les autres nœuds sont déjà présents. Le registre retournera à n7 les voisins n4, n5, n9 et n10 associés chacun à l'une des directions nord-ouest, nord-est, sud-ouest et sud-est respectivement. Le nœud n7 va alors se connecter à ces nœuds puis les notifier un par un

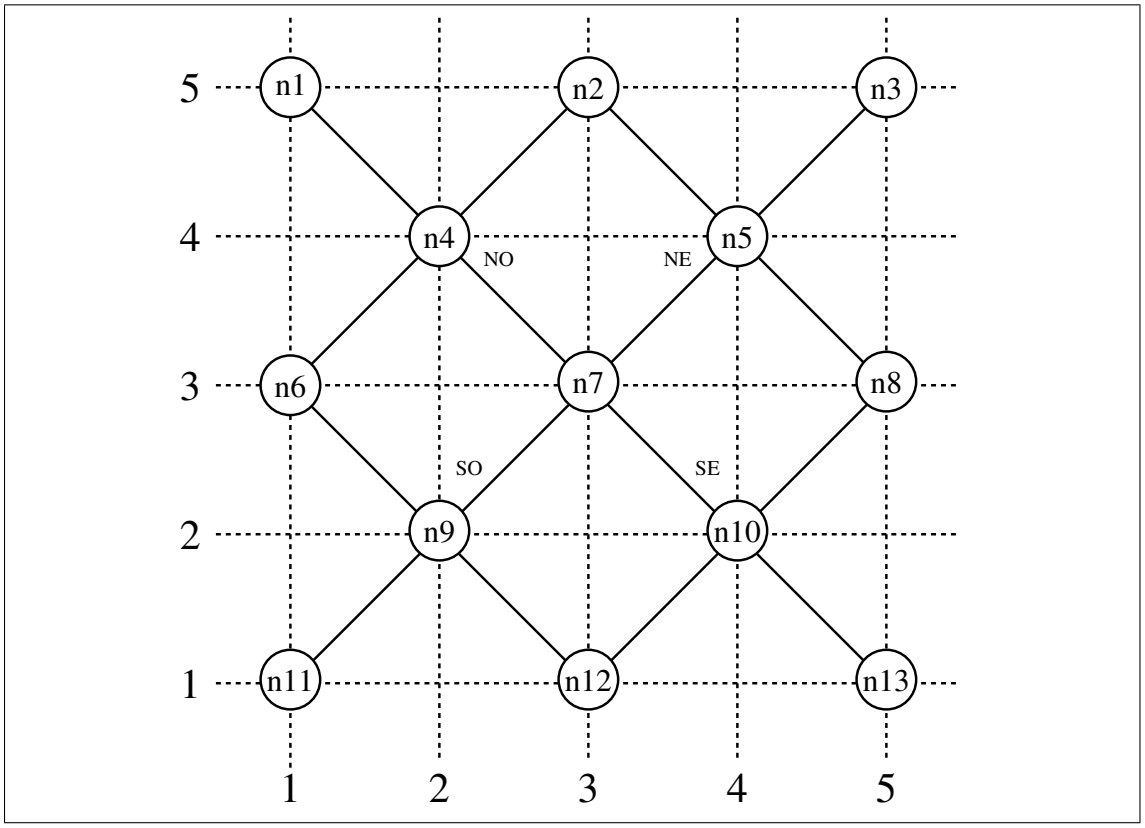

FIGURE 1 – Réseau de capteurs. Notez les directions nord-ouest (NO), nord-est (NE), sud-ouest (SO) et sud-est (SE) indiquées par rapport au nœud n7 ; ainsi, par exemple, le nœud n5 est-il au nord-est de n7. Les indices en ordonnée et en abcisse peuvent être interprétés comme des kilomètres, et donc donne la distance entre les nœuds. Par exemple, la distance entre n7 et n5 est 1,4142 km ($\sqrt{2}$) alors qu'entre n7 et n2 elle est de 2 km.

pour que ces derniers se connectent à lui. Notez que plusieurs nœuds existent ici pour chacune des directions. Dans la direction nord-est, le nœud n3 est aussi au nord-ouest de n7. Le registre retourne donc pour chaque direction le nœud le plus proche géographiquement du nouveau nœud dans la portée de ce dernier. [1] Notez également que lors de la construction du réseau, l'arrivée d'un nouveau nœud peut remettre en cause les liaisons des nœuds déjà présents dans le réseau. Par exemple, si n9 était le nœud sud-ouest de n5, après l'arrivée de n7, n5 changera son nœud à son sud-ouest pour n7 et n9 changera son nœud au nord-est pour n7 également. [2]

En utilisant ce réseau de capteur, le scénario précédent s'illustrerait de la manière suivante :

— Supposons que la requête initiale (expression booléenne sur les capteurs `température` et `fumée`) soit envoyée aux nœuds n1, n3, n7, n11 et n13, et alors que ce soit le nœud n7 qui réponde avec des valeurs laissant croire à un feu dans son secteur.
— Le superviseur envoie la seconde requête (valeurs des capteurs `vitesse-vent` et `direction-vent`), à laquelle les résultats indiqueraient par exemple la direction nord-est.
— Le superviseur envoie alors une troisième requête à n7 (la même que la première mais avec comme continuation la direction nord-est) qui parcourerait les nœuds n7 (receveur initial de la requête) puis n5 et n3 pour essayer de délimiter la zone du feu.

Notez que si la vitesse du vent est très faible, sa direction aura peu d'influence sur l'extension du

1. Parmi les informations sur les nœuds, il y aura la portée de son émetteur réseau sans fil, ce qui limitera les liaisons possibles aux nœuds qui sont à l'intérieur de cette portée et pour lesquels le nouveau nœud est également dans leur portée.

2. Pour éviter les ambiguïtés, il faut que chanque nœud ne soit jamais que dans u seul des quadrants par rapport à un autre nœud. Il faut donc définir les choses strictement. On considérera que est au nord un nœud dont la position en ordonnée est supérieure ou égale à celle du nœud auquel on se réfère et au sud celui dont l'ordonnée est strictement inférieure. De même, on considérera qu'un nœud est à l'est si sa position en abcisse est supérieure ou égale à celle du nœud auquel on se réfère et à l'ouest si elle est strictement inférieure. Notez aussi qu'en degrés, le nord-est correspond au quadrant allant de 0 (inclus) à 90° (inclus), le nord-ouest va de 90 (exclus) à 180° (inclus), le sud-ouest va de 180 (exclus) à 270° (exclus) et enfin le sud-est va de 270 (inclus) à 360° (exclus).

feu. Le superviseur pourra alors utiliser une deuxième forme de continuation consistant à donner une distance maximale depuis le nœud recevant initialement la requête. Dans notre exemple, si la requête fournit comme continuation la distance maximale 2 km à partir du nœud initial n7, la requête sera passée à tous ses nœuds voisins (n4, n5, n9 et n10) qui eux-mêmes repasseront la requête à leurs propres nœuds voisins jusqu'à ce que la distance de 2 km par rapport à n7 soit dépassée. Dans cet exemple, les nœuds n2, n6, n8 et n12 seront touchés mais pas les nœuds n1, n3, n11 et n13 situés à plus de 2 km de n7.

Données capteurs, requêtes et résultats de requêtes

Le réseau de capteurs va permettre de récolter des données des capteurs et d'exécuter des requêtes booléennes servant à détecter des alertes qui dépendent des données capteurs. Notons d'abord que nous allons supposer que tous les nœuds dans le réseau possèdent un identifiant unique et qu'une nomenclature des différents capteurs disponibles nous permet de les désigner également par des identifiants uniques (par exemple, `température` ou `direction-vent`. Ainsi, les requêtes qui vont être transmises sur le réseau sont des types suivants :

Requêtes de collecte : ces requêtes fournissent une liste d'identifiants de capteurs détenus par les nœuds et dont les valeurs courantes doivent être retournées au client émetteur de la requête.

Requêtes booléennes : ces requêtes comportent une expression booléenne portant sur les valeurs de capteurs détenus par les nœuds ; cette expression booléenne est évaluée sur les nœuds et son résultat est la liste des identifiants de nœuds où l'expression booléenne a été évaluée à vrai.

Les deux types de requêtes vont aussi avoir une partie qu'on va appeler continuation qui identifie les nœuds suivants auxquels la requête doit être propagée après exécution sur le nœud courant. L'implantation de ce langage de requêtes sera le premier objectif à atteindre dans le développement du projet. Il s'agira d'implanter un interpréteur qui va servir aux composants représentant les nœuds capteurs à exécuter les requêtes. Nous revenons à la section suivante sur sa définition précise.

Déroulement général du projet

Le projet va 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 à une exécution réellement parallèle et répartie dans un second temps. L'idée est de vous permettre d'appréhender ce type de programmation parallèle et répartie, et ce qui la 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.

En gros, le projet va se dérouler sur tout le semestre de la manière suivante :

1. **Première et deuxième étapes** :
   (a) Implantation de l'interpréteur du langage de requête.
   (b) Implantation du réseau de capteurs et des interconnexions entre composants.
   (c) Implantation de l'exécution synchrone des requêtes sur les nœuds dans un style direct, similaire à une exécution séquentielle classique.
2. **Troisième et quatrième étapes** :
   (a) Passage à une exécution asynchrone des requêtes en style passage à la continuation exploitant mieux les ressources des nœuds grâce au parallélisme.
   (b) Introduction de l'abstraction de comportement et de la réutilisation par l'utilisation de greffons (*plug-ins*).

(c) Recherche de performance et de passage à l'échelle par l'utilisation de *threads* multiples, ce qui nécessitera de bien gérer les accès concurrents aux données par ces *threads* s'exécutant en « parallèle ».

La fin de chaque étape sera marquée par une évaluation (audit ou soutenance). Nous allons détailler ces différents points dans les sections suivantes.

2 Langage de requêtes

La première tâche du projet va être de développer un interpréteur por le langage de requêtes. Voici la syntaxe abstraite de ce langage [3] :

query	::=	**GQuery** *gather cont* || **BQuery** *bexp cont*
gather	::=	**RGather** `sensorId` *gather* || **FGather** `sensorId`
bexp	::=	**AndBExp** $bexp_1$ $bexp_2$ || **OrBExp** $bexp_1$ $bexp_2$ || **NotBExp** *bexp* || **SBExp** `sensorId` || **CExpBExp** *cexp*
cexp	::=	**EqCExp** $rand_1$ $rand_2$ || **LCExp** $rand_1$ $rand_2$ || **LeqCExp** $rand_1$ $rand_2$ || **GCExp** $rand_1$ $rand_2$ || **GeqCExp** $rand_1$ $rand_2$
rand	::=	**SRand** `sensorId` || **CRand** `double`
cont	::=	**ECont** || **DCont** *dirs* `int` // nombre maximal de sauts autorisés || **FCont** *base* `double` // distance maximale de la base
dirs	::=	**RDirs** *dir dirs* || **FDirs** *dir*
dir	::=	`NE` || `NW` || `SE` || `SW`
base	::=	**ABase** `position` || **RBase** `this`

Les requêtes (non terminal *query*) ont deux formes : les requêtes de collecte de données capteur **GQuery** et les requêtes booléennes **BQuery**. Pour les premières, elles se définissent par une liste d'identifiants de capteurs dont on veut obtenir les valeurs (non terminal *gather*) et d'une continuation (non terminal *cont*). Pour les secondes, elles sont constituées d'une expression booléenne (*bexp*) et d'une continuation. Les requêtes booléennes retournent une liste d'identifiants de nœuds, identifiants qu'on implantera comme des chaînes de caractères. Les requêtes de collecte vont retourner une liste de triplets comportant l'identifiant du nœud, l'identifiant du capteur et la valeur du capteur.

Le non terminal *gather* est défini par deux règles dont une récursive pour donner une liste d'identifiants de capteurs. Le non terminal *bexp* est défini de manière très classique. Les étiquettes **AndBExp**, **OrBExp** et **NotBExp** permettent de distinguer les expressions et, ou et non habituelles. On peut toutefois y utiliser un identifiant de capteur par **SBExp** et dans ce cas, le capteur devra être du type booléen pour respecter les types attendus par les opérateurs booléens (et, ou non). Un expression booléenne peut avoir des opérandes qui sont des expressions de comparaison (non terminal *cexp*). Les expressions de comparaison sont définies de manière tout aussi classique avec les opérateurs de comparaison sur les réels indiqués par les étiquettes **EqCExp** (égal), **LCExp** (plus petit), **LeqCExp** (plus petit ou égal), **GCExp** (plus grand) et **GeqCExp** (plus grand ou égal). Ici aussi, les deux seuls cas d'opérandes possibles (non terminal *rand*) sont une constante réelle (**CRand**) ou un identifiant de capteur (**SRand**), auquel cas il devra s'agir d'un capteur à valeur réelle. [4]

Les continuations (non terminal *cont*) sont de trois formes : les continuations vides (**ECont**, pour les requêtes s'exécutant que sur un nœud), les continuations directionnelles (**DCont**) et les continuations d'inondation (**FCont**). Les continuations directionnelles énumèrent une liste de directions à suivre (non terminal *dirs*) et un nombre maximal de sauts autorisés (constante `int`). Les continuations de type inondation définissent la position de référence (non terminal *base*) et la portée maximale à partir de cette position de référence (constante `double`). La position de base peut être fournie par une constante de type position (`position`) ou par le mot-clé `this` auquel cas on prendra pour position de référence celle du nœud recevant initialement la requête.

L'évaluation d'une continuation peut se faire progressivement, en prenant d'abord les voisins

3. Cette syntaxe abstraite produit des arbres de syntaxe abstraite. Elle est définie en règles au format Backus-Naur. Les non terminaux sont en *italique*, les terminaux sont en police `machine à écrire` et les étiquettes permettant de distinguer les différentes définitions en partie droite des règles sont en **petites capitales et en gras**.

4. Pour simplifier le projet, nous allons supposer que les capteurs n'ont des valeurs que des types booléen et réel.

du nœud ayant reçu initialement la requête, puis les voisins des voisins, etc., en s'arrêtant lorsque le critère (nombre de sauts autorisés ou portée maximale) est atteint.

Implantation du langage de requêtes

Pour vous guider dans l'implantation du langage de requête, une sémantique précise pour ce langage de requêtes est fournie en annexe A à la fois sous la forme d'une sémantique formelle (dénotationnelle) et d'une sémantique opérationnelle expliquée en langue naturelle (français...) abordant comment il faudra passer de la sémantique formelle à une exécution répartie sur les nœuds du réseau de capteurs implantés par des composants BCM4Java qui sera conforme à la sémantique.

Comment utiliser concrètement cette définition formelle dans le cadre du projet ? D'abord, il faut implanter un interpréteur du langage. L'approche classique consiste à faire une implantation *dirigée par la syntaxe*[5] qui revient à créer sous la forme d'objets, donc définir les classes correspondant à chaque non terminal de la grammaire pour les organiser en arbre de syntaxe abstraite, où chaque classe définit des variables qui contiennent la référence à chacune des sous-éléments en partie gauche de la règle de syntaxe. Une fois cette organisation obtenue, il faut définir une méthode `eval` pour chacune des ces classes qui parcourt l'arbre de syntaxe abstraite pour évaluer le non terminal que cette classe représente à l'aide de l'évaluation des non terminaux de ses sous-éléments, et ce conformément à la sémantique.

Le seul point un peu délicat de cette partie de l'implantation sera d'assurer la liaison entre l'interpréteur et les capteurs détenus par chaque composant représentant un nœud du réseau logique. En effet, selon la description précédente, on voit que dans les expressions de collecte ainsi que dans les expressions booléennes, il faut à certains endroits aller chercher les valeurs courantes des capteurs du nœud courant. Il faudra donc définir un protocole valide sur tous ces composants permettant à l'interpréteur d'obtenir la valeur d'un capteur à partir de son identifiant.

Dans un premier temps, l'implantation doit suivre le style direct, c'est-à-dire qu'un nœud recevant une requête calcule son résultat local à la requête (partie *gather* ou *bexp*), puis il identifie ses voisins faisant partie de la continuation et appelle tour à tour chacun de ces voisins pour qu'il exécute la suite de la requête et en retourne le résultat. Le nœud récupère tous les résultats de ses voisins et les assemble avec son propre résultat local pour retourner le résultat complet à son propre appelant, qui peut être soit un nœud du réseau qui le précéde dans l'exécution de cette requête, soit le composant superviseur ayant initié la requête.

Pour ce qui est de la représentation des requêtes, nous n'avons pas défini de syntaxe concrète qui permettrait d'écrire un programme sous la forme d'un texte qui passerait par une analyse lexicale puis une analyse syntaxique pour produire un arbre de syntaxe abstraite, comme avec un compilateur. Pour contourner le besoin d'analyseurs lexical et syntaxiques, il sera plus rapide pour les besoins du projet de construire les programmes directement sous la forme d'arbre de syntaxe abstraite. Par exemple, voici à quoi pourrait ressembler le code Java qui créerait une requête complète vérifiant la condition du départ d'un feu de forêt susceptible de se propager dans la direction nord-est :

```
new BQuery(
    new AndBExp(new GeqCEXP(
                    new SRand("température"),
                    new CRand(50.0)),
                new GeqCEXP(
                    new SRand("fumée"),
                    new CRand(3.0))),
    new DCont(new FDirs(Directions.NE), 2))
```

Comme nous allons le voir plus loin, vous allez implanter ce langage en deux temps. Dans un premier temps, vous allez développer une première version du réseau de capteur où les requêtes seront évaluées en style direct, c'est à dire par des appels synchrones entre composants qui retournent à leur appelant le résultat de cette requête comme on le fait habituellement en programmation

5. C'est l'approche utilisée dans l'UE DLP.

```
public interface ConnectionInfoI extends Serializable {
  public String nodeIdentifier();
  public EndPointDescriptorI endPointInfo();
}

public interface NodeInfoI extends ConnectionInfoI {
  public PositionI nodePosition();
  public double nodeRange();
  public EndPointDescriptorI p2pEndPointInfo();
}

public interface EndPointDescriptorI extends Serializable { }

public interface BCM4JavaEndPointDescriptorI extends EndPointDescriptorI {
  public String getInboundPortURI();
  public boolean isOfferedInterface(Class<? extends OfferedCI> inter);
}

public enum Direction {NE, NW, SE, SW}

public interface PositionI extends Serializable {
  public double distance(PositionI p);
  public Direction directionFrom(PositionI p);
  public boolean northOf(PositionI p);
  public boolean southOf(PositionI p);
  public boolean eastOf(PositionI p);
  public boolean westOf(PositionI p);
}
```

FIGURE 2 – Informations sur les nœuds. Un nœud est d'abord défini par son identifiant eet ses informations de connexion par l'interface `ConnectionInfoI`. S'ajoutent la position et la portée qui peuvent différer selon les implantations et définis par l'interface `NodeInfoI`. Pour être plus de généralité, l'interface `EndPointDescriptorI` ne force aucune implantation particulière pour la connexion entre composant qui pourrait donc aussi varier d'une implantation à l'autre, mais l'interface `BCM4JavaEndPointDescriptorI` précise que pour des composants BCM4Java, les informations de connexion sont l'URI du port entrant qui expose une certaine interface offerte. Les positions sont implantées par une classe implantant l'interface `PositionI`, dont vous noterez qu'elle ne prescrit pas une représentation précise mais uniquement les opérations nécessaires pour les besoins du réseau logique.

séquentielle. Dans un deuxième temps, vous allez ajouter une seconde implantation passant à une évaluation asynchrone qui va permettre de mieux utiliser le parallélisme et la répartition, ce qui permettrait aussi potentiellement de passer à l'échelle.

3 Construction du réseau de capteurs

Nous avons expliqué dans la première section du cahier des charges que les nœuds du réseau de capteurs vont être implantés par des composants BCM4Java. Chacun de ces composants vont détenir un ensemble de capteurs et vont faire évoluer les valeurs de ces capteurs par des mesures qui seront simulées pour construire des scénarios d'exécution en vue des démonstrations d'exécution du système. Ces composants vont devoir s'appeler les uns les autres à travers des ports ert des connecteurs implantant des interfaces de composants. Cette section couvre les spécifications qui vont vous permettre de développer composants, ports et services pour ces composants et pour les composants clients du réseau de capteurs.

```
public interface SensorNodeP2PImplI
{
  public void ask4Connection(NodeInfoI newNeighbour) throws Exception;
  public void ask4Disconnection(NodeInfoI neighbour) throws Exception;
  public QueryResultI execute(RequestContinuationI request) throws Exception;
  public void executeAsync(RequestContinuationI requestContinuation)
  throws Exception;
}
```

FIGURE 3 – `SensorNodeP2PImplI` est l'interface d'implantation, c'est à dire qu'elle est implantée au sens Java par la classe définissant les composants nœuds, qui définit les méthodes implantées par les nœuds pour assurer leur interconnexion et la propagation des requêtes d'un nœud à l'autre tel que requis par leur continuation.

3.1 Les nœuds et leur contenu

D'abord chaque nœud est décrit pas des informations qui sont rassemblées dans un objet descripteur du nœud dont la classe implantant (au sens Java) l'interface `NodeInfoI` qui elle-même utilise d'autres interfaces apparaissant dans la figure 2. [6] Un tel descripteur sera fourni à la création de chaque composant BCM4Java qui représente le nœud dans l'application.

L'interface `ConnectionInfoI` permet d'accéder aux informations nécessaires à un composant client pour se connecter à un composant nœud. Le descripteur du point d'accès (`EndPointDescriptorI`) est défini d'abord de manière totalement générique et indépendant des technologies d'interconnexion employées. L'interface `BCM4JavaEndPointDescriptorI` étend cette interface pour spécifier les informations qui s'appliquent au cas des connexions BCM4Java à savoir l'URI du port entrant et l'interface de composants exposée par ce port entrant. Dans le cas du résultat de la méthode `ConnectionInfoI#endPointInfo()`, ce sera les informations de connexion au composant nœud via l'interface de composants `RequestingCI` (figure 6), ou une interface étendant cette dernière, à utiliser par les composants clients (voir la sous-section 3.3). Les informations de connexions représentées par `ConnectionInfoI` sont minimales et ce sont celles qui sont échangées avec les composants clients depuis le registre (voir à nouveau la sous-section 3.3).

Chaque nœud devant s'interconnecter avec d'autres nœuds, l'interface `NodeInfoI` étend et complète l'interface `ConnectionInfoI` pour représenter des informations nécessaires au registre et aux interconnexions entre nœuds : la postion du nœud, sa portée de connexion sans fil et le descripteur d'un second point d'accès pour connecter les composants nœuds entre eux pour former le réseau de capteurs. Ce deuxième point d'accès est défini par l'URI du port entrant qui expose l'interface de composants `SensorNodeP2PCI` (figure 7) ou une interface qui étend cette dernière.

La classe définissant les composants représentant les nœuds du réseau de capteur va donc prévoir de conserver les informations sur le nœud (`NodeInfoI`) de manière à les envoyer au registre lors de son inscription dans le réseau de capteurs (voir sous-section 3.2). Cette classe va aussi devoir implanter (au sens Java) l'interface `SensorNodeP2PImplI` (figure 3). Cette interface définit trois méthodes. `ask4Connection` est utilisée lors de l'entrée du nœud dans le réseau pour s'interconnecter avec ses voisins (voir sous-section 3.2). Les deux autres méthodes sont utilisées pour propager les requêtes en fonction de leur continuation (voir sous-section 3.3).

3.2 Insertion dans le réseau de capteurs : registre global et interconnexions

Lorsqu'un nouveau nœud veut s'insérer dans le réseau de capteur, soit parce qu'il vient d'être installé ou qu'il vient d'être (r)allumé, il va devoir :

1. S'inscrire auprès du registre du réseau.
2. S'interconnecter avec les nœuds voisins géographiquement identifiés par le registre.

6. Ces interfaces vont sont fournies dans une archives comme point de départ de votre projet à intégrer dans Eclipse. Elles sont dument commentées en Javdoc en complément des descriptions faites dans ce document.

```
public interface RegistrationCI extends OfferedCI, RequiredCI
{
  public boolean registered(String nodeIdentifier)throws Exception;
  public Set<NodeInfoI> register(NodeInfoI nodeInfo) throws Exception;
  public NodeInfoI findNewNeighbour(NodeInfoI nodeInfo) throws Exception;
  public void unregister(String nodeIdentifier) throws Exception;
}
```

FIGURE 4 – `RegistrationCI` est offerte par le registre et requise par les nœuds du réseau de capteurs pour s'enregistrer.

Pour s'inscrire auprès du composant registre, le composant nœud va devoir d'abord se connecter (au sens BCM4Java) à celui-ci via l'interface de composants `RegistrationCI` (figure 4). Au tout début de son exécution, le composant nœud doit appeler la méthode `register` en passant ses informations (`NodeInfoI`), et le registre va lui retourner l'ensemble des informations sur les nœuds qui pourront devenir ses voisins dans le réseau. Au plus quatre voisins lui seront attribués, un au plus pour chaque direction (nord-ouest, nord-est, sud-ouest et sud-est) à condition qu'il en existe et que le nouveau nœud et le voisin soient dans leur portée d'émission mutuelle.

Sur réception de cet ensemble, le nouveau nœud X va se connecter à chacun de ces voisins Y via l'interface de composants `SensorNodeP2PCI` (figure 7). Cette connexion faite, X va appeler chacun de ses nouveaux voisins Y par la méthode `ask4Connection` à laquelle il passe ses propres informations de connexion pour que chacun des voisins puisse se connecter à lui en retour. Chaque voisin Y va alors se connecter en retour à X via `SensorNodeP2PCI`. De cette manière, on obtient une interconnexion dans les deux sens entre les nœuds.

Si un nœud Y reçoit une demande de connexion d'un nouveau nœud X dans une direction où Y avait déjà un voisin Z, Y doit d'abord se déconnecter de Z. Pour cela, Y appelle d'abord la méthode `ask4Disconnection` sur Z pour que celui-ci se déconnecte de lui, puis Y se déconnecte effectivement de Z avant de se connecter avec X. Lorsqu'un nœud se retrouve sans voisin dans une certaine direction, il appelle le registre avec la méthode `findNewNeighbour` en fournissant en paramètre ses informations de nœud et la direction dans laquelle il n'a plus de voisin ; le registre retourne les informations de connexion d'un nouveau voisin ou `null` s'il n'existe plus de voisin possible dans cette direction.

Lorsque le nœud quitte le réseau de capteur, il doit d'abord se déconnecter de ses voisins puis se désenregistrer auprès du registre en appelant la méthode `unregister`.

3.3 Exécution des requêtes

Pour faire éxécuter une requête par un certain composant nœud, un composant client doit d'abord se connecter à lui (au sens de BCM4Java). Pour cela, il doit avoir les informations nécessaires, que le client va récupérer auprès du registre. La séquence est donc :

1. Le composant client se connecte au composant registre via l'interface de composant `LookupCI` (figure 5).
2. Le client appelle le registre soit par la méthode `findByIdentifier`, auquel cas il va récupérer les informations de connexion à ce nœud, ou encore par la méthode `findByZone`, auquel cas il va récupérer un ensemble d'informations de connexion aux nœuds dans cette zone, parmi lesquels il va en sélectionner un.
3. Avec les informations de connexion reçues du registre, le client se connecte au composant nœud via l'interface de composants `RequestingCI` (figure 6) qui lui permettra d'envoyer sa requête.

3.3.1 Exécution synchrone en style direct

L'interface de composants `RequestingCI` (figure 6) définit deux méthodes pour envoyer une requête à un composant nœud :

— `execute` est utilisée en style direct synchrone ;

```
public interface LookupCI extends OfferedCI, RequiredCI
{
  public ConnectionInfoI findByIdentifier(String sensorNodeId)
  throws Exception;
  public Set<ConnectionInfoI> findByZone(GeographicalZoneI z)
  throws Exception;
}

public interface GeographicalZoneI extends Serializable
{
  public boolean in(PositionI p);
}
```

FIGURE 5 – `LookupCI` est offerte par le registre et requise par les composants clients (superviseurs) pour récupérer les informations de connexions avec les nœuds du réseau de capteurs afin de leur envoyer des requêtes à exécuter.

```
public interface RequestI extends Serializable
{
  public String requestURI();
  public Query getQueryCode();
  public boolean isAsynchronous();
  public ConnectionInfoI clientConnectionInfo();
}

public interface QueryResultI extends Serializable
{
  public boolean isBooleanRequest();
  public ArrayList<String> positiveSensorNodes();
  public boolean isGatherRequest();
  public ArrayList<SensorDataI> gatheredSensorsValues();
}

public interface RequestingImplI
{
  Public QueryResultI execute(RequestI request) throws Exception;
  public void executeAsync(RequestI request) throws Exception;
}

public interface RequestingCI extends OfferedCI, RequiredCI, RequestingImplI
{
  @Override
  public QueryResultI execute(RequestI request) throws Exception;
  @Override
  public void executeAsync(RequestI request) throws Exception;
}
```

FIGURE 6 – Cette figure présente les interfaces Java et l'interface de composants liées à l'exécution des requêtes. `RequestI` sert à la représentation des requêtes échangées, qu'elles soient de type synchrone ou asynchrone. `QueryResultI` sert à la représentation du résultat des requêtes à retourner au client, soit des valeurs de capteurs, soit des identifiants de composants nœuds pour lesquels la requête booléenne est vraie. `RequestingCI` est l'interface de composant offerte par les composants nœuds et requise par les composants clients permettant de faire exécuter des requêtes sur le réseau de capteurs. La méthode `execute` implante le style direct synchrone alors qu'`executeAsync` implante le style passage à la continuation en appel asynchrone. Notez que dans le cas d'`executeAsync`, le paramètre de type `RequestI` doit représenter une requête asynchrone (*i.e.*, `request.isAsynchronous()` retourne vrai). L'interface d'implantation `RequestingImplI` est implantée au sens de Java par les composants nœuds.

— `executeAsync` est utilisée en style passage à la continuation asynchrone.

Commençons par le style direct synchrone. Le client appelle `execute` en lui passant une requête représentée par une instance d'une classe implantant l'interface `RequestI`. Une requête possède une URI, un code sous la forme d'un arbre de syntaxe abstrait (`isAsynchronous` et `clientConnectionInfo` sont discutées dans la sous-sous-section suivante). Le composant nœud va utiliser ce paramètre pour récupérer le code de la requête et exécuter d'abord localement la partie collecte ou booléenne, puis il devra propager la requête vers la continuation.

En style direct, que la continuation soit directionnelle ou inondation, elle doit d'abord identifier les nœuds voisins qui sont dans cette continuation puis leur propager la requête et enfin récupérer les résultats retournés par ces appels aux voisins de manière à composer le résultat complet de la requête qui sera retourné au client sous la forme d'un objet présentant l'interface `QueryResultI`. La propagation va se faire par un appel aux voisins via l'interface de composants `SensorNodeP2PCI` (figure 7).

En style direct synchrone, la propagation utilise la méthode `execute` à laquelle est passée une continuation de requête définie par l'interface `RequestContinuationI`. Cette continuation est une simple extension de `RequestI` qui permet de récupérer l'état d'exécution courant de la requête. Cet état d'exécution est fait pour être passé de composant nœuds en composant nœuds pour être mis à jour au fil de l'exécution de la requête. L'état d'exécution permet de récupérer :

— par `getProcessingNode()` une référence sur le composant nœud courant de type `ProcessingNodeI` ;
— par `getCurrentResult()` le résultat cumulé depuis le début de l'exécution de la requête ;
— par `addToCurrentResult` d'ajouter le résultat obtenu localement au résultat cumulé ;
— par `isDirectional` de savoir si la continuation est directionnelle et dans ce cas de savoir par `noMoreHops` s'il reste des sauts autorisés ainsi que par `incrementHops` d'incrémenter le nombre de sauts déjà faits ;
— par `isFlooding` de savoir si la continuation est de type inondation et dans ce cas de tester si la psition d'un nœud est dans la portée prévue pour cette inondation.

Les méthodes `updateProcessingNode`, `getCurrentResult()` et `addToCurrentResult` servent à mettre à jour l'état d'exécution en modifiant le nœud qui exécute la requête actuellement et en ajoutant des résultats locaux aux résultats cumulés jusque-là dans l'exécution de la requête. La méthode `updateProcessingNode` permet de mettre à jour la référence sur le nœud courant après la propagation vers un voisin. Les méthodes `getCurrentResult()` et `addToCurrentResult` ne sont pas utilisées dans le cas d'appels synchrones mais seulement dans le cas asynchrone qui sera décrit à la sous-sous-section suivante. Les autres méthodes de `ExecutionStateI` sont utilisées pour la propagation de la requête. Les méthodes `isDirectional` et `isFlooding` permettent d'identifier le type de continuation que la requête utilise.

Lorsque le nœud qui a initialement reçu la requête a terminé l'exécution de la partie locale de cette dernière, il vérifie les conditions de terminaison (par `ExecutionStateI#noMoreHops` ou `ExecutionStateI#withinMaximalDistance`). S'il doit arrêter, il retourne un résultat qui ne contient que ses résultats locaux à l'appelant. S'il doit poursuivre, il va devoir créer un objet implantant l'interface `RequestContinuationI` avec les informations sur la continuation qui ont été obtenues de l'exécution locale de la requête. Il appelle ensuite ses voisins faisant partie de la continuation avec la méthode `SensorNodeP2PCI#execute` (fig. 7) en lui passant l'objet continuation qu'il vient de créer. Il attend ensuite les résultats de ces appels pour les fusionner ensemble et avec ses résultats locaux avant de retourner le tout à son appelant.

Dès qu'un nœud reçoit une continuation par la méthode `SensorNodeP2PCI#execute`, il met à jour la référence sur le nœud courant dans l'état d'exécution puis exécute de la partie collecte ou booléenne localement. Avant de relancer la propagation de cette requête vers ses propres voisins, il vérifie les conditions de terminaison et s'il doit arrêter la propagation, il retourne son résultat local comme résultat de la requête à son appelant. Et s'il doit poursuivre, il exécute localement la requête, met à jour la condition de terminaison par `ExecutionStateI#incrementHops` si la continuation est de type directionnelle et propage ensuite la continuation de requête vers ses propres voisins faisant partie de la continuation par la méthode `SensorNodeP2PCI#execute`. Il attend alors les résultats pour les fusionner avec ses résultats locaux avant de renvoyer les résultats fusionnés à son appelant.

```
public interface SensorNodeP2PCI extends OfferedCI, RequiredCI, SensorNodeP2PImplI
{
  @Override
  public void ask4Connection(NodeInfoI newNeighbour) throws Exception;
  @Override
  public void ask4Disconnection(NodeInfoI neighbour) throws Exception;
  @Override
  public QueryResultI execute(RequestContinuationI request) throws Exception;
  @Override
  public void executeAsync(RequestContinuationI requestContinuation)
  throws Exception;
}

public interface RequestContinuationI extends RequestI
{
  public ExecutionStateI getExecutionState();
}

public interface ExecutionStateI extends Serializable
{
  public ProcessingNodeI getProcessingNode();
  public void updateProcessingNode(ProcessingNodeI pn);
  public QueryResultI getCurrentResult();
  public void addToCurrentResult(QueryResultI result);
  public boolean isDirectional();
  public Set<Direction> getDirections();
  public boolean noMoreHops();
  public void incrementHops();
  public boolean isFlooding();
  public boolean withinMaximalDistance(PositionI p);
}

public interface ProcessingNodeI
{
  public String getNodeIdentifier();
  public PositionI getPosition();
  public Set<NodeInfoI> getNeighbours();
  public SensorDataI getSensorData(String sensorIdentifier);
}
```

FIGURE 7 – `SensorNodeP2PCI` est l'interface de composants offerte et requise par les composants nœuds pour assurer leur interconnexion. L'interface `SensorNodeP2PImplI` est définie à la figure 3.

3.3.2 Exécution asynchrone en style passage à la continuation

L'un des premiers points limitant la performance de l'implantation en style direct 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 externe n'a pas retourné son résultat, alors que dans les appels asynchrones, le *thread* du composant appelant peut être libéré immédiatement après l'appel externe en faisant

```
public interface RequestResultCI {
  public void acceptRequestResult(
    String requestURI,
    QueryResultI result
    ) throws Exception;
}
```

FIGURE 8 – Interface de composant offerte par les composants clients et requise par les nœuds du réseau de capteurs permettant l'envoi et la réception des résultats des requêtes.

en sorte de terminer la tâche immédiatement après cet appel. Dans notre système, par exemple, un appel à `execute(RequestContinuationI)` par un nœud va demander au composant nœud d'occuper un de ses *threads* tant que le composant nœud voisin n'aura pas retourné de résultat. Pire, pour retourner son résultat, ce voisin va devoir aussi appeler ses propres voisins et attendre leurs propres résultats, ce qui va également occuper l'un de ses *threads*, et ainsi de suite. Tout cela bloque et occupe beaucoup trop de ressources sur de nombreux nœuds, ce qui limite très sensiblement 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 d'un nœud à son voisin, mais alors se pose le problème du retour du résultat, car l'appel asynchrone ne retourne pas de résultat. Pour résoudre cette difficulté, il faut utiliser un patron de conception [7] où l'appel asynchrone connaît les informations de connexion lui permettant de renvoyer le résultat directement au composant qui a lancé la requête quand le résultat est complet. Le composant client qui doit recevoir le résultat doit alors offrir une interface de composants avec une méthode à appeler pour lui envoyer ce résultat. La figure 8 présente l'interface de composants `RequestResultCI` avec la méthode `acceptRequestResult` permettant de passer le résultat d'une requête à son appelant qui offrira `RequestResultCI`.

Après ces considérons générales, reprenons le déroulement de l'exécution d'une requête mais cette fois-ci en utilisant l'approche asynchrone. Le composant client connecté au composant nœud, comme vu précédemment, va d'abord appeler la méthode `RequestingCI#executeAsync` (fig. 6) en lui passant une requête où la méthode `RequestI#isAsynchronous` retourne vrai. Dans ce cas, le méthode `RequestI#clientConnectionInfo` de la requête va devoir retourner les informations de connexion du point d'entrée du composant client offrant l'interface de composants `RequestResultCI` qui permettra aux composants nœuds (requérant cette même interface) de lui retourner le résultat de la requête. On comprend ici l'intérêt d'avoir un identifiant unique de requête car si le composant client lance plusieurs requêtes en parallèle, c'est cet identifiant (URI) qui va lui permettre de faire le lien entre le résultat qu'il reçoit et la requête à laquelle il s'applique.

Le premier composant nœud recevant la requête via la méthode `RequestingCI#executeAsync` commence par exécuter le code de la requête, comme dans le cas synchrone. C'est après l'exécution locale et lors du passage à la continuation que le déroulement va changer. D'abord, le premier composant nœud va vérifier les conditions de terminaison et si l'exécution doit se terminer, il envoie au client ses résultats locaux comme résultat de la requête. Pour cela, il doit se connecter au composant client via l'interface de composants `RequestResultCI` grâce aux informations de connexion obtenue par `RequestI#clientConnectionInfo` puis lui envoyer le résultat par `RequestResultCI#acceptRequestResult`.

Si l'exécution doit se poursuivre dans l'un ou plusieurs de ses voisins, le nœud initial crée l'objet implantant l'interface `RequestContinuationI` comme dans le cas synchrone mais cette fois-ci en ajoutant le résultat local de la requête comme son résultat obtenu jusqu'ici. Il va ensuite propager la requête à ses voisins faisant partie de la continuation mais cette fois en utilisant la méthode `SensorNodeP2PCI#executeAsync`. Un nœud voisin recevant cette continuation de requête va procéder similairement au cas synchrone décrit précédemment, sauf pour ce qui concerne l'envoi du résultat (via `RequestResultCI#acceptRequestResult` comme décrit ci-haut) et la propagation à ses propres voisins. Pour construire les résultats courants de la requête à partir des résultats locaux, le nœud doit d'abord ajouter ses résultats locaux à ceux déjà obtenus qui sont présents dans l'état d'exécution avec la méthode `ExecutionStateI#addToCurrentResult`. Après la vérification

7. 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.

des conditions de terminaison, si l'exécution doit s'arrêter, il récupère le résultat fusionné avec la méthode `ExecutionStateI#getCurrentResult` pour l'envoyer au composant client. Si le nœud doit propager la requête, il se contente de mettre à jour les conditions de terminaison puis de propager la continuation de requête mise à jour (résultats et conditions de terminaison) à ses propres voisins faisant partie de la continuation.

Si on considère les appels successifs, à la méthode `RequestingCI#executeAsync` pour le nœud initial puis `SensorNodeP2PCI#executeAsync` pour les nœuds dans la continuation, qui vont se déployer lors de l'exécution asynchrone d'une requête, on constate d'abord que dès qu'un nœud a propagé la requête à ses voisins faisant partie de la continuation, il peut terminer la tâche d'exécution de cette requête, ce qui va libérer ses ressources pour traiter d'autres requêtes. Ensuite, on constate que lors de la propagation d'une requête à plusieurs voisins, cette propagation va se faire en *parallèle* à tous ses voisins. Eux-mêmes vont aussi la propager en parallèle à tous leurs propres voisins et ainsi de suite pour former non pas une séquence linéaire d'appels mais une *arborescence* d'appels qui possède *plusieurs feuilles*. On constate aussi que ce sont les nœuds feuilles de cette arborescence d'appels asynchrones qui vont envoyer les résultats partiels cumulés le long de leur chaîne d'appels au composant client.

Le composant client va donc recevoir autant de résultats partiels à sa requête qu'il y a de feuilles dans l'arborescence d'appels, résultats partiels qu'il va devoir fusionner pour obtenir les résultats complets de sa requête. En plus de cette fusion, va se poser le problème de savoir quand le composant client peut considérer qu'il a reçu tous les résultats partiels de sa requête. Il ne connaît effectivement pas la topologie du réseau de capteurs et donc il ne peut pas savoir combien il y aura de feuilles dans l'arborescence des appels que sa requête va engendrer. [8] Pour résoudre ce problème, il va falloir utiliser une temporisation, c'est à dire fixer un délai maximal d'attente des résultats partiels à l'issue duquel, le composant client va fusionner tous les résultats partiels reçus et retourner les résultats fusionnés au composant pour traitement.

3.4 Parallélisme et gestion de concurrence

L'introduction du traitement asynchrone des requêtes au sein des composants nœuds du réseau de capteur vise non seulement à éviter les situations d'inter-blocages dues à la sémantique des appels synchrones mais également à introduire du parallélisme dans l'exécution des requêtes pour augmenter la performance du réseau de capteurs. Pour cela, il va falloir accompagner de l'introduction de l'asynchronisme d'une gestion du parallélisme et par conséquent d'une gestion de la concurrence au sein des composants :

- Pour gérer le parallélisme, 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.
- Pour gérer la concurrence, 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.

3.5 Réutilisabilité : utilisation de greffons

Sur la base des développements précédents, le passage à des greffons consiste à regrouper le code nécessaire aux différents rôles joués par les composants, comme celui de nœud dans un réseau de capteur, dans des greffons, ce qui va permettre une meilleure réutilisation logicielle et augmenter la modularité du système.

Le passage aux greffons consistera à implanter les deux principaux rôles des composants :

8. En fait, on peut construire une solution à ce problème de dénombrement sans que le client ne connaisse la topologie du réseau de capteurs. Si une équipe arrive à une solution, je serai heureux de lui accorder un bonus sur sa note finale d'UE...

1. Un greffon pour le rôle nœud du réseau de capteur qui va implanter les fonctionnalités supportant les interfaces de composant `RequestingCI`, `SensorNodeP2PCI` et `RegistrationCI`.
2. Un greffon pour le rôle de client du réseau de capteurs qui va implanter les fonctionnalités supportant les interfaces de composant `RequestingCI` et `LookupCI`.

3.6 Analyse de performance

Pour compléter le projet, il faudra procéder à des expérimentations sur le réseau de capteurs. Pour cela, le nombre de composants nœuds et clients dans la démonstration finale sera de l'ordre de 5 pour les clients et de 50 pour les nœuds.

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 pour voir jusqu'à quel point une augmentation du nombre de *threads* alloués à chaque composant pour trainter les requêtes influe sur la performance globale du réseau. Pour cela, il faudra jouer sur les deux paramètres suivants :

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

Le nombre de *threads* alloué à l'exécution des requêtes : il faudra mesurer les performances en allouant d'abord un *thread* dans le *pool* de *threads* dédié à l'exécution des requêtes de chaque composant nœud (y compris le traintement des propagations), puis 2, 5 et 10, l'objectif étant de déterminer à partir de quel nombre la performance globale du réseau de capteurs cesse de s'améliorer.

Les résultats croisés (rythme et nombre de *threads* alloués) de ces expérimentations seront présentés sous forme de tableaux lors de la soutenance finale.

4 Démonstration et simulation

4.1 Scénario initial

Pour le scénario initial représentant le seuil minimal pour l'audit 1, il s'agit de réaliser une première connexion entre composants client et nœud de manière à valider l'acquisition des compétences les plus basiques de programmation en BCM4Java : créer des composants, des ports, des connecteurs et un déploiement sur la machine virtuelle à composants d'un composant client et d'un composant nœud, de les connecter directement de manière statique puis, à l'exécution, de faire en sorte que le composant client envoie une requête à continuation vide au composant nœud qui va l'exécuter de manière synchrone et retourner le résultat au composant client.

Pour ce scénario initial, il suffira pour le composant nœud de fixer une valeur initiale à chacun de ses capteurs. Ces valeurs seront retournées au composant client en réponse à sa requête de type collecte de données de capteurs.

4.2 Scénarios de base

Pour les scénarios de test de base, il faudra ajouter deux grandes fonctionnalités : l'exécution synchrone de requêtes ayant une continuation non vide (directionnelles et d'inondation) ainsi que d'intégrer l'utilisation du registre des nœuds du réseau de capteurs (à la fois pour gérer la construction du réseau et pour l'envoi de requêtes par les composants clients). Ces scénarios vont aussi exiger de gérer le déroulement des actions (arrivées et départs des nœuds dans le réseau, envoi des requêtes par des composants clients).

Les tests d'intégrations des applications parallèles et répartis sont beaucoup plus complexes que les tests des applications séquentielles classiques. Dans une application séquentielle classique, sans interface graphique, il suffit de construire un ensemble de cas de tests et de les dérouler les uns après les autres ans un ordre quelconque sur l'application. Dans une applicaiton avec interface

graphique, c'est déjà plus complexe car les tests doivent prendre en compte un déroulement dans le temps des actions de l'utilisateur. Il faut donc simuler les actions de l'utilisateur dans un ordre précis. De plus, comme plusieurs séquences d'interaction avec l'utilisateur sont possibles, il va falloir définir différents scénarios d'interaction pour tester toutes les séquences d'actions possibles afin de couvrir toutes les facettes de l'application. Chacun de ces scénarios d'interaction devra être repris sur un ensemble de jeux de données pour couvrir également la plus grande part possible des données d'entrées potentielles pouvant être soumises à l'application.

Dans les applications parallèles et réparties, s'ajoutent les difficultés liées à la temporisation des actions. En effet, dans ces applications, non seulement les délais entre les actions sont importantes mais également plusieurs séquences d'actions se déroulent en parallèle et l'entrelacement de leurs actions dans le temps va également avoir une grande importance pour vérifier toutes les séquences globales d'actions possibles. Dans les applications séquentielles à interface graphique, on peut souvent se contenter de séquences d'actions faite dans un certain ordre, donc d'un notion de temps qui peut se comparer à celle des automates ou des langages synchrones en programmation temps réel. Dans les scénarios pour les applications parallèles et réparties, il faut une notion de temps similaire au temps physique dans lequel nous vivons, c'est à dire une horloge globale avec des instants ponctuels où on a un notion de distance (délai) entre deux instants. Les actions à exécuter dans les scénarios de tests vont devoir être planifiées et déclenchées à des instants précis selon une horloge qui pourront varier d'un scénario à l'autre de manière à couvrir toutes les séquences et tous les entrelacement possibles entre elles.

En BCM4Java, deux outils sont fournis pour faciliter l'écriture de tels scénarios : une horloge globale synchronisée sur l'horloge physique de l'ordinateur sous-jacent et des *threads* dits *ordonnançables* permettant de planifier et déclencher des tâches à exécuter à des instants précis spécifiés dans le temps de l'horloge globale. L'annexe B présente dans le détail la classe `AcceleratedClock` ainsi que le composant `ClocksServer` et détaille l'utilisation de ces dernières pour produire des scénarios de tests temporisés. Pour l'utilisation des *pools* de *threads* ordonnançables, la méthode `scheduleTask` utilisée dans l'annexe doit vous suffire pour réaliser les tests temporisés de l'évaluation de mi-semestre. Vous pouvez toutefois consulter la documentation d'`AbstractComponent` pour connaître et utiliser les autres méthodes permettant de faire exécuter des tâches par des composants à des instants choisis dans le temps.

Scénarios de tests du projet pour l'évaluation de mi-semestre

Pour les scénarios de base du projet, voici les actions qui vont devoir faire l'objet d'une planification et d'un déclenchement dirigés par l'horloge pour l'évaluation de mi-semestre :

— les arrivées des nœuds dans le réseau de capteurs ;
— la récupération des données de connexion auprès du registre par les composant clients ;
— les mises à jour des données des capteurs sur les nœuds, ce qui va simuler les prises de mesures dans un réseau de capteurs réel ;
— les envois de requêtes depuis les composants clients.

L'objectif pour les tests lors de la soutenance de mi-semestre sera d'arriver à construire un réseau de cinq composants nœuds, un central et les quatre autres étant ses voisins, se joignant au réseau en passant par le registre puis d'avoir un composant client qui récupère les informations de connexion du composant nœud central et s'y connecte pour lui envoyer et faire exécuter deux requêtes, l'une avec continuation directionnelle et l'autre avec continuation inondation, à partir du composant nœud central.

4.3 Scénarios avancés et de tests de performance

Les scénarios avancés ont pour objectif de vérifier :

— le bon fonctionnement des requêtes avec des continuations à plus d'un niveau (voisins de voisins, voisins de voisins de voisins, etc.) ;
— le bon fonctionnement de la gestion du parallélisme et de la concurrence dans les composants ;
— le bon fonctionnement de l'évolution dynamique du réseau de capteurs en échelonnant les arrivées et les départs de nœuds pendant les exécutions de requêtes sur le réseau.

Pour cela, il faudra d'abord augmenter le nombre des composants nœuds et la complexité du réseau de capteur puis de modifier les requêtes pour exploiter des continuations à plus d'un niveau. Ensuite, il faudra exploiter la capacité du composant client à lancer plusieurs requêtes sans en attendre les résultats (exécution asynchrone) pour faire exécuter plusieurs requêtes en même temps sur le réseau de capteurs.

Les scénarios de tests de performance ont pour objectif d'évaluer la performance du réseau de capteurs sous stress en termes de nombre de requêtes par seconde qui peuvent être exécutées. Comme expliqué en 3.6, il s'agit :

— d'augmenter le nombre de composants clients, le nombre de requêtes envoyées et la fréquence d'envoi des requêtes pour mesurer le temps nécessaire pour compléter l'exécution de toutes les requêtes et ainsi calculer le nombre de requêtes exécutées par seconde ;
— de reprendre les mesures précédentes avec différentes allocations de *threads* aux composants nœuds.

5 Déroulement du projet et résultats attendu

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

5.1 Audit 1 : langage de requêtes

La première étape va être centrée sur l'implantation du langage de requêtes tel que décrit dans la section 2 puis sur l'exécution de requêtes sur un unique composant nœud dans le style direct décrit à la sous-section 3.3.1, ce qui demandera une prise en main de BCM4Java suffisante pour :

— implanter un composant nœud offrant l'interface de composant `RequestCI` (figure 6) et un composant client requérant la même interface ;
— de connecter *directement* les deux composants (sans passer par le registre pour l'instant) avec les ports et connecteurs appropriés ;
— de monter une exécution avec des deux composants où le composant client demandera l'exécution d'une requête de type collecte de données de capteurs avec une continuation vide selon le scénario de base décrit en 4.1.

Pour cet audit, le principal critère d'évaluation sera le degré de réalisation des développements décrits dans les sections 2, 3.1, 3.3.1 (avec continuations vides) et 4.1). L'exécution correcte du scénario de test précédent sera une condition minimale de succès à l'audit.

5.2 Soutenance de mi-semestre : exécution synchrone en style direct

L'étape 2 va demander de développer une première version complète du réseau de capteurs, avec au moins quelques composants nœuds qui vont se joindre au réseau en passant par le registre. Les nœuds devront être organisés en un réseau suffisamment complexe, où au moins deux nœuds seront à une distance de deux sauts l'un de l'autre de manière à pouvoir tester des requêtes ayant les deux types de continuation, directionnelles et inondation géographique. À cette étape, seule l'exécution synchrone en style direct devra être implantée.

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 :

— un interprète de requêtes complet ;
— tous les ports entrants et sortants ainsi que les connecteurs correspondant aux interfaces de composants fournies ;
— les trois types de composants attendus : nœuds, clients, registre.

Ils porteront également sur une exécution d'un test d'intégration sur une seule machine virtuelle Java où seront exécutés :

— différentes requêtes (de collecte et booléennes) avec différentes continuations (vide, directionnelle et inondation), toutes exécutées de manière synchrone ;

— des scénarios de tests, comme expliqué dans la sous-section 4.2, où les données des capteurs des différents nœuds vont évoluer dans le temps, ce qui permettra de lancer les requêtes entre les changements de valeurs des capteurs pour valider le bon fonctionnement global du système.

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 (sections 2, 3.1, 3.2 et 3.3.1 avec continuations non vides et 4.2) 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 ;
— qualité de la programmation Java ;
— pertinence des choix d'implantation.

5.3 Audit 2 : exécution asynchrone, parallélisme et concurrence

L'objectif principal de la troisième étape sera d'ajouter l'exécution asynchrone des requêtes, telle qu'expliquée dans la sous-section 3.3.2. Ce faisant, il faudra également assurer une gestion explicite du parallélisme par utilisation de *pools* de *threads* créés explicitement et une gestion de la concurrence par l'introduction de sections critiques dans le code (voir la sous-section 3.4).

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 ;
— introduction de *pools* de *threads* distincts dans les composants nœuds et registre ;
— la gestion de la concurrence sur l'accès aux données des capteurs dans les composants nœuds et dans le composant registre ;
— les mêmes scénarios de tests d'intégration que ceux utilisés pour la soutenance de mi-semestre seront utilisés pour vérifier le bon fonctionnement des appels asynchrones, à ceci près qu'il faudra aussi vérifier le bon fonctionnement du parallélisme et de la concurence au sein des composants nœuds en faisant s'exécuter plus d'une requête à la fois sur les mêmes composants nœuds.

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.

5.4 Soutenance finale

Pour la dernière étape du projet, bien que l'évaluation soit récapitulative sur l'ensemble du projet, le travail à réaliser sera d'abord de réorganiser le code des composants autour des greffons, tel qu'expliqué dans la sous-section 3.5 puis de monter des scénarios de test et d'évaluation de performance où le réseau de capteurs devra comporter plus de nœuds, où seront utilisés plusieurs composants clients et où de plus en plus de requêtes seront exécutées dans des délais assez courts. Ces scénarios devront servir à évaluer la performance avec différentes tailles des *pools* de *threads* (voir sous-section 3.6).

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). Les résultats attendus comportent :

— l'ensemble des développements demandés, et en particulier l'utilisation des greffons ;
— des scénarios de tests d'intégration et de performance sous la forme de déploiements comportant 5 composants clients, et 50 composants nœuds ;
— un plan de tests pour les évaluations de la performance avec des tableaux présentant les mesures obtenues.

Pour obtenir la meilleure note possible au projet, les tests de performance devraient être réalisés en déploiement réparti sur 5 machines virtuelles Java, chacune ayant un composant client et 10 composants nœuds et des requêtes étant envoyées et exécutées entre les machines virtuelles. À défaut, un déploiement s'exécutant correctement sur une seule JVM sera votre objectif minimal à atteindre pour obtenir la note de passage.

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 l'ensemble de votre code selon les critères donnés pour la soutenance de mi-semestre auxquels va s'ajouter la qualité de votre documentation.

6 Modalités générales de réalisation et calendrier des évaluations

— Le projet se fait **obligatoirement** en **équipe de deux ou trois 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 **30 janvier 2024**.
— 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 (**12 février 2024**) et il portera sur votre avancement de l'étape 1. Le second audit se déroulera lors de la séance 9 (**22 avril 2024**) et il portera sur votre avancement de l'étape 3. Ils compteront chacun pour 5% de la note finale de l'UE.
2. La **soutenance à mi-parcours** d'une durée de 20 minutes portera sur l'atteinte des objectifs de l'ensemble des première et deuxième étapes du projet. Elle se tiendra pendant la semaine des premiers examens répartis du **4 au 8 mars 2024** selon un ordre de passage et des créneaux qui seront annoncés au préalable. Elle comptera pour 35% de la note finale de l'UE. Elle comportera une discussion des réalisations pendant une quinzaine de minutes (devant l'écran sous Eclipse) et une courte démonstration de cinq minutes. Les rendus à mi-parcours se feront le **dimanche 3 mars 2024 à minuit** au plus tard (des pénalités de retard seront appliquées).
3. La **soutenance finale** d'une durée de 20 minutes portera sur l'ensemble du projet mais avec un accent sur les troisième et quatrième étapes. Elle aura lieu dans la semaine des seconds examens répartis du **13 au 17 mai 2024** (et plus probablement le lundi 13/05/2024 et mardi 14/05/2024 selon un ordre de passage et des créneaux qui seront annoncés au préalable. Elle comptera pour 55% de la note finale de l'UE. Elle comportera une discussion d'une quinzaine de minutes sur les réalisations suivie d'une démonstration. Les rendus finaux se feront le **dimanche 12 mai 2024 à minuit** au plus tard (des pénalités de retard seront appliquées).

— Lors des soutenances, les points suivants seront évalués :
  — le respect du cahier des charges et la qualité de votre programmation ;
  — l'exécution correcte de tests unitaires (JUnit pour les classes et les objets Java, scénario de tests pour les composants) ;
  — l'exécution correcte de tests d'intégration mettant en œuvre des composants de tous les types ;
  — la qualité et l'exécution correcte des scénarios de tests de performance, conçus pour mettre à l'épreuve les choix d'implantation versus la montée en charge ;
  — la qualité de votre code (votre code doit être commenté, être lisible – choix pertinents des identifiants, ... – et correctement présenté – indentation, ... – etc.) ;
  — la qualité de votre plan d'expérimentation et tests de performance (couverture de différents cas, isolation des effets pour identifier leurs impacts sur la performance globale, etc.) ;
  — la qualité de la documentation (vos rendus pour la soutenance finale devront inclure une *documentation Javadoc* des différents paquetages et classes de votre projet générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que votre répertoire `src`).

— Bien que les audits et les soutenances se fassent par équipe, l'évaluation reste à chaque fois **individuelle**. Lors des audits et des soutenances, *chaque membre* devra se montrer capable d'expliquer différentes parties du projet, et selon la qualité de ses explications et de ses réponses, sa note peut être supérieure, égale ou inférieure à celle des autres membres de son équipe.
— Lors des soutenances, **tout retard** non justifié d'un des membres de l'équipe de plus d'un tiers de la durée de la soutenance (7 minutes) entraînera une **absence** et une note de 0 attribuée aux membre(s) retardataire(s) pour l'ensemble de l'épreuve concernée (y compris le rendu préalable). Si une partie des membres d'une équipe arrive à l'heure ou avec un retard de moins d'un tiers de la durée de la soutenance, les présents passeront l'épreuve seul.
— Le rendu à mi-parcours et le rendu final se font sous la forme d'une archive `tgz` si vous travaillez sous Unix ou `zip` si vous travaillez sous Windows ***à l'exclusion de toute autre format*** (n'inclure *que les sources*, c'est à dire uniquement le répertoire `src` de votre projet *sans les binaires* et les jars pour réduire la taille du fichier) envoyé à `Jacques.Malenfant@lip6.fr` comme attachement fait proprement avec votre programme de gestion de courrier préféré ou encore par téléchargement avec un lien envoyé par courrier électronique (en lieu et place du fichier). Donnez pour nom au répertoire de projet et à votre archive celui de votre équipe (ex. : équipe LionDeBelfort, répertoire de projet `LionDeBelfort` et archive `LionDeBelfort.tgz`).
— **Tout manquement à ces règles élémentaires entraînera une pénalité dans la note des épreuves concernées !**
— Pour la **deuxième session**, si elle s'avérait nécessaire, elle consiste à poursuivre le développement du projet pour résoudre ses insuffisances constatées à la première session et donnera lieu à un rendu du code et de documentation puis à une soutenance dont les dates seront déterminées en fonction du calendrier du master. Les critères d'évaluation sont les mêmes que lors de la soutenance finale, mais modulés selon qu'une seule personne ou deux de la même équipe doivent passer la seconde session. Sur demande faite en avance (dès les notes de première session connues), une personne peut choisir de passer la seconde session seule même si d'autres membres de l'équipe doivent également le passer ; en cas de désaccord entre les membres de l'équipe, la demande de passage en solitaire prévaudra.

Références

[BSVV20] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries over models@runtime for runtime monitoring of cyber-physical systems. *International Journal on Software Tools for Technology Transfer*, (22) :79–102, 2020.

A Sémantique dénotationelle du langage de requêtes

Rappelons la syntaxe abstraite du langage de requêtes déjà décrite précédemment :

query	::=	**GQuery** *gather cont* || **BQuery** *bexp cont*
gather	::=	**RGather** `sensorId` *gather* || **FGather** `sensorId`
bexp	::=	**AndBExp** $bexp_1$ $bexp_2$ || **OrBExp** $bexp_1$ $bexp_2$ || **NotBExp** *bexp* || **SBExp** `sensorId` || **CExpBExp** *cexp*
cexp	::=	**EqCExp** $rand_1$ $rand_2$ || **LCExp** $rand_1$ $rand_2$ || **LeqCExp** $rand_1$ $rand_2$ || **GCExp** $rand_1$ $rand_2$ || **GeqCExp** $rand_1$ $rand_2$
rand	::=	**SRand** `sensorId` || **CRand** `double`
cont	::=	**ECont** || **DCont** *dirs* `int` // nombre maximal de sauts autorisés || **FCont** *base* `double` // distance maximale de la base
dirs	::=	**RDirs** *dir dirs* || **FDirs** *dir*
dir	::=	`NE` || `NW` || `SE` || `SW`
base	::=	**ABase** `position` || **RBase** `this`

A.1 Considérations introductives

Nous allons développer une sémantique dénotationnelle pour ce langage. Une sémantique dénotationnelle est une forme de sémantique formelle pour les langages de programmation qui décrit le résultat d'un programme sous la forme d'une fonction mathématique. Pour l'essentiel, c'est un λ-calcul dont les valeurs manipulées sont définies par des *domaines*, c'est à dire des ensembles munis d'une relation d'ordre.

Toutefois, il n'est pas nécessaire de se plonger profondément dans la théorie pour lire et comprendre une sémantique dénotationnelle. Le λ-calcul se comprend comme un programme en langage fonctionnel, à ceci près que le λ-calcul n'admet pas de récursivité dans la définition des fonctions. Pour définir l'équivalent d'une fonction récursive, il faut utiliser ce qu'on appelle l'*opérateur de point fixe* **fix**. Cet opérateur prend une fonction et l'applique à elle-même pour obtenir l'effet de la récursivité. Là encore, point n'est besoin de se plonger dans la théorie, il suffit de comprendre comment passer d'une définition récursive à une définition par point fixe. Supposons une fonction récursive simple qui parcourt une liste d'entiers et retourne une liste d'entiers où chaque valeur est incrémentée de 1. Dans un pseudo λ-calcul avec définitions cela donnerait (voir les notations utilisées à la figure 9) :

$$\textbf{def}\ incr \quad = \quad \lambda\, l.\textbf{if}\ \varepsilon(l)\ \textbf{then}\ \epsilon\ \textbf{else}\ (\boxdot l + 1) \boxdot incr(\boxplus l)$$

Cette définition dit simplement que si la liste l est vide ($\varepsilon(l)$) alors on retourne la liste vide (ϵ) sinon, on retourne la liste obtenue de la concaténation du premier élément de l ($\boxdot l$) incrémenté de 1 avec la liste retournée par l'appel récursif à *incr*. La version avec opérateur de point fixe de cette fonction est :

$$\textbf{def}\ incr \quad = \quad \lambda l.((\textbf{fix}\ \lambda f.\ \lambda ll.\textbf{if}\ \varepsilon(ll)\ \textbf{then}\ \epsilon\ \textbf{else}\ (\boxdot ll + 1) \boxdot f(\boxplus ll))\ l)$$

Sans entrer dans les détails, l'appel récursif à *incr* est remplacé par un appel à une fonction f sur laquelle est appliqué l'opérateur **fix** qui «réalise» l'effet récursif équivalent sur cette dernière.

Pour les domaines sémantiques [9], c'est à dire les types de valeurs valeurs utilisées dans les fonctions définissant la sémantique, on utilise des domaines simples dont les réels ($\mathbb{R}$), les entiers ($\mathbb{N}$), les booléens ($\mathbb{B}$). On utilise aussi des domaines simples mais moins courants comme les identifiants ($\mathbb{I}$ pour les nœuds et $\mathbb{I}_s$ pour les capteurs sur les nœuds), les positions ($\mathbb{P}$) qui ne sont pas spécifiés précisément. On utilise aussi des domaines complexes, essentiellement les listes (T^* est le type liste de valeurs de type T), les domaines sommes ($T_1 \oplus T_2$ dénote l'ensemble des valeurs soit de type T_1 soit de type T_2) et les produits ($T_1 \otimes T_2$ dénote l'ensemble des valeurs tuples dont le premier élément est de type T_1 et le second de type T_2). Là encore, la figure 9 donne quelques notations d'opérations permettant de manipuler les valeurs de ces types complexes.

9. Pour ne pas trop formaliser le discours, la définition des domaines sémantiques omet la question de la relation d'ordre nécessaire pour obtenir une sémantique dénotationnelle, mais les domaines décrits sont soit bien définis soit sont facilement modifiables pour obtenir de «vrais» domaines.

ϵ	liste vide.
$e \boxdot l$	liste obtenue après ajout d'un élément e au début de la liste l.
$l1 \boxplus l2$	concaténation des listes *l1* et *l2*.
$l1 \boxminus l2$	la liste *l1* moins les éléments présents dans *l2*.
$l1 \cup l2$	concaténation des listes *l1* et *l2* avec retrait des entrées doublons.
$\varepsilon(l)$	vrai si l est vide et faux sinon.
$\boxdot l$	premier élément de la liste l (non vide).
$\boxplus l$	liste obtenue après avoir enlevé le premier élément de la liste l (non vide).
$\langle\,\rangle_T$	création d'une valeur du domaine produit T.
$\pi\vert_n^T$	extraction du nième élément d'une valeur du domaine produit T.
$\downarrow_T^S$	extraction de la valeur de type T d'une valeur du domaine somme S.
$\uparrow_T^S$	injection d'une valeur de type T dans le domaine somme S.

FIGURE 9 – Notations.

A.2 Domaines et fonctions sémantiques

Les domaines sémantiques eux-mêmes nécessaires pour définir la sémantique de notre langage de requêtes sont les suivants :

$\mathbb{I}$		ensemble des identifiants de nœuds capteurs (non spécifié).	(1)
$\mathbb{I}_s$		ensemble des identifiants de capteurs (non spécifié).	(2)
$\mathbb{P}$		ensemble des positions géographiques (non spécifié).	(3)
$\mathbb{SV} : \mathbb{B} \oplus \mathbb{R}$		ensemble des valeurs de capteurs : booléen ou réel.	(4)
$\mathbb{SN} : \mathbb{I} \otimes \mathbb{P} \otimes (\mathbb{I}_s \otimes \mathbb{SV})^* \otimes \mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I}$		ensemble des nœuds capteurs : identifiant, position, capteurs, voisins dans les quatre directions (*ne, nw, se, sw*).	(5)
$\mathbb{D}$	$=$	$\{ne, nw, se, sw\}$	(6)
$\Sigma : \mathbb{SN} \otimes \mathbb{SN}^*$		le nœud courant, liste des nœuds existants.	(7)
$\Phi : \mathbb{I} \otimes \mathbb{I}_s \otimes \mathbb{SV}$		valeur d'un capteur d'un nœud	(8)
$\Gamma : \Phi^* \oplus \mathbb{I}^*$		liste de valeurs de capteurs ou liste d'identifiants de nœuds	(9)

Pour les domaines simples, nous avons déjà mentionné les identifiants de nœuds (1) et de capteurs (2) ainsi que les positions (3). Les capteurs peuvent prendre des valeurs soit réelles soit booléennes, d'où la définition du domaine somme $\mathbb{SV}$ (4). Un nœud du réseau est défini ici comme une valeur du domaine produit $\mathbb{SN}$ (5) contituée de l'identifiant du nœud, de sa position, d'une liste de couples identifiant et valeur de capteur donnant les valeurs courantes de ses capteurs puis quatre identifiants de nœuds voisins respectivement dans la direction nord-est, nord-ouest, sud-est et sud-ouest. Les directions sont définies par un domaine simple $\mathbb{D}$ (6) discret et fini.

Pour les calculs, il faut définir des domaines spécifiques. À tout instant le calcul se fait dans un contexte défini par le domaine produit Σ (7) contenant la valeur du domaine $\mathbb{SN}$ définissant le nœud courant sur lequel on s'exécute et la liste des valeurs de $\mathbb{SN}$ définissant tous les nœuds présents dans le réseau à l'instant courant. Les requêtes peuvent retourner deux types de résultats : si c'est une requête booléenne, une liste d'identifiants des nœuds parcourus pour lesquels la requête est vraie, si c'est une requête de collecte, une liste des valeurs des capteurs collectées sous la fome d'un domaine produit Φ (8) contenant l'identifiant du nœud, l'identifiant du capteur et la valeur de ce dernier. Enfin, le résultat d'une requête est une valeur du domaine somme Γ (9), c'est à dire soit une liste de valeurs de capteurs du domaine Φ^*, soit une liste d'identfiants de nœuds du domaine $\mathbb{I}^*$

Avant d'aborder les fonctions sémantiques en tant que telles, nous définissons quelques fonctions

sémantiques auxiliaires qui vont faciliter ensuite l'écriture des précédentes :

$$sn : \mathbb{I} \otimes \mathbb{SN}^* \to \mathbb{SN} \quad \text{nœud capteur correspondant à un identifiant (non spécifiée).} \quad (10)$$

$$sv : \mathbb{SN} \otimes \mathbb{I}_s \to \mathbb{SV} \quad \text{valeur d'un capteur sur un nœuds capteur (non spécifié).} \quad (11)$$

$$nd : \mathbb{SN} \otimes \mathbb{D} \to \mathbb{I} \quad \text{identifiant du voisin d'un nœud dans la direction donnée (non spécifiée).} \quad (12)$$

$$ns : \mathbb{I}^* \otimes \mathbb{D}^* \otimes \mathbb{SN}^* \to \mathbb{I}^* \quad \text{liste des identifiants de voisins (sans répétition) d'une liste de nœuds donnés par leur identifiant (non spécifiée).} \quad (13)$$

$$\delta : \mathbb{P} \otimes \mathbb{P} \to \mathbb{R} \quad \text{distance entre deux positions (non spécifiée).} \quad (14)$$

L'état du calcul défini par le domaine Σ comporte la liste des nœuds présents dans le réseau du type $\mathbb{SN}^*$. La fonction sn (10) prend un identifiant de nœud et la liste des nœuds présents pour retourner le nœud ayant l'identifiant donné. La fonction sv (11) prend un nœud et un identifiant de capteur pour retourner la valeur de ce capteur pour le nœud donné. La fonction nd (12) prend un nœud et une direction pour retourner l'identifiant du nœud voisin dans la direction donnée. La fonction ns (13) quant à elle prend une liste des identifiants de plusieurs nœuds, une liste de directions et la liste des nœuds présents dans le réseau pour retourner la listes des voisins de tous les nœuds donnés dans les directions données sans répétition.

Les quatre fonctions précédentes ne sont pas spécifiées car suffisamment évidentes pour ne pas alourdir nos explications. Par contre le domaine $\mathbb{P}$ des positions de même que la fonction δ ne sont pas spécifiés volontairement pour laisser à l'implanteur du langage le choix de la réprésentation des positions et le choix de la distance qui seront utilisés. Un telle non spécification dans une sémantique définit en quelque sorte un type abstrait laissant champ libre à l'implanteur. Libre à lui de le spécifier formellement selon son choix.

A.3 Fonctions d'évaluation

Abordons maintenant les fonctions d'évaluation, c'est à dire les fonctions qui associent à chaque construction de la syntaxe abstraite une fonction mathématique permettant de spécifier le résultat d'un programme en fonction de son contenu et des données d'exécution. L'organisation de ces fonctions suit un standard où pour chaque non terminal de la sémantique abstraite est défini une fonction d'évaluation qui elle-même possède des définitions distinctes pour chaque choix de la partie droite de la définition de cet élément syntaxique. Par exemple, pour l'élément syntaxique *query*, on définit la fonction d'évaluation $\mathcal{Q}$ qui possède deux définitions, la première pour la cas d'un requête de collecte et le second pour le cas de la requête booléenne.

Une fonction d'évaluation prend au moins un paramètre venant de la syntaxe abstraite puis peut prendre 0, 1 ou plusieurs paramètres supplémentaires selon les besoins. Pour bien marquer la différence de traitement du premier paramètre (syntaxe abstraite) qui donne lieu à une définition par cas avec appariement structurel et possibilité d'appel récursif sur la structure de la syntaxe abstraite [10], il est de tradition de le mettre entre double crochets $[\![\ ]\!]$. Pour les autres paramètres passés à la fonction d'évaluation selon un mode traditionnel, il sont simplement ajoutés après le paramètre syntaxique. Pour la sémantique de notre langage de requêtes, nous avons dit que l'évaluation se fait dans un contexte défini par le domaine Σ, d'où l'utilisation d'un paramètre σ à chaque fois que nécessaire dans les différentes fonctions d'évaluation. Enfin, pour plus de clarté, nsou ajoutons le type du résultat de chacune des fonctions d'évaluation ; par exemple, la fonction $\mathcal{Q}$ retourne un résultat du domaine Γ.

Il est plus facile d'expliquer et de comprendre la sémantique si on la construit depuis les non terminaux de base de la syntaxe abstraite en remontant vers les non terminaux plus complexes pour terminer par le non terminal de départ, ici *query*. Prenons donc d'abord les non terminaux de la fin de la syntaxe abstraite :

10. Cette forme de récursivité est autorisée pour les fonctions d'évaluation car elle est bien fondée si la syntaxe abstraite est elle-même bien fondée. Il y a cependant une règle importante : tous les appels à des fonctions d'évaluation dans la partie droite définissant cette fonction d'évaluation doit s'appliquer à un *sous-élément* de l'élément syntaxique en partie gauche. Par exemple, il est interdit d'écrire quelque chose comme $\mathcal{E}[\![s]\!] = ...\mathcal{E}[\![s]\!]...$ car alors la récursivité ne serait plus bien fondée. En effet, l'appel récursif porte exactement sur le même élément syntaxique qui apparaît en partie gauche de la définition de la fonction d'évaluation, ce qui n'assure plus que cette récursivité s'arrête nécessairement (*i.e.*, qu'elle est bien fondée).

$$\mathcal{DL}[\![\textbf{RDirs } dir \ dirs]\!] : \mathbb{D}^* = \mathcal{D}[\![dir]\!] \boxdot \mathcal{DL}[\![dirs]\!] \tag{15}$$

$$\mathcal{DL}[\![\textbf{FDirs } dir]\!] : \mathbb{D}^* = \mathcal{D}[\![dir]\!] \boxdot \epsilon \tag{16}$$

$$\mathcal{D}[\![\texttt{NE}]\!] : \mathbb{D} = ne \tag{17}$$

$$\mathcal{D}[\![\texttt{NW}]\!] : \mathbb{D} = nw \tag{18}$$

$$\mathcal{D}[\![\texttt{SE}]\!] : \mathbb{D} = se \tag{19}$$

$$\mathcal{D}[\![\texttt{SW}]\!] : \mathbb{D} = sw \tag{20}$$

$$\mathcal{B}[\![\texttt{position}]\!]\sigma : \mathbb{P} = \mathbb{P}[\![\texttt{position}]\!] \tag{21}$$

$$\mathcal{B}[\![\texttt{this}]\!]\sigma : \mathbb{P} = (\sigma|_1^\Sigma)|_2^{SN} \tag{22}$$

$$\mathcal{P}[\![\texttt{position}]\!] : \mathbb{P} = p \quad \text{(non spécifié)} \tag{23}$$

$$\mathcal{N}[\![\texttt{int}]\!] : \mathbb{N} = n \quad \text{(non spécifié)} \tag{24}$$

$$\mathcal{R}[\![\texttt{double}]\!] : \mathbb{R} = r \quad \text{(non spécifié)} \tag{25}$$

Pour les fonctions $\mathcal{P}$ (23), $\mathcal{N}$ (24) et $\mathcal{R}$ (25), elles transforment les constantes syntaxiques de position, entières et réelles dans des valeurs des domaines correspondant $\mathbb{P}$, $\mathbb{N}$ et $\mathbb{R}$; elles ne sont pas spécifiées à la fois parce que c'est fastidieux, simpliste et, pour les positions, volontaire, comme expliqué précédemment. La fonction d'évaluation $\mathcal{B}$ (21 et 22) évalue la base servant à définir une continuation de type inondation (avec une position de base et une portée maximale). Elle possède deux définitions, l'une pour une base donnée sous forme de position absolue, l'autre pour une base utilisant la position du nœud courant. Dans le premier cas (21), il suffit d'évaluer la position absolue donnée. Dans le second cas (22), il faut retourner la position du nœud courant en extrayant la valeur de celui-ci de σ, puis en extrayant sa position. La fonction $\mathcal{D}$ (17 à 20) transforme directement les positions exprimées dans la syntaxe abstraite en des valeurs du domaine $\mathbb{D}$.

La fonction d'évaluation $\mathcal{DL}$ s'attaque aux listes de directions définies par une règle récursive à droite dans la syntaxe abstraite. Elle possède deux définitions. La seconde (16) traite le cas terminal où il n'y a plus qu'une direction et elle retourne la direction en question mise dans une liste en la plaçant en premier grâce à l'opérateur $\boxdot$ dyadique appliqué à la liste vide. La première définition (15) calcule la première direction et la place en premier sur la liste retournée par l'appel récursif à $\mathcal{DL}$ sur le reste de la liste.

La fonction d'évaluation $\mathcal{C}$ traite le non terminal *cont*, les continuations :

$$\begin{aligned}\mathcal{C}[\![\textbf{FCont } base \ \texttt{double}]\!]\sigma : \mathbb{I}^* = \ & \textbf{let } p = \mathcal{B}[\![base]\!]\sigma \textbf{ and } r = \mathcal{R}[\![\texttt{double}]\!] \\ & \textbf{in } (\textbf{fix } \lambda\, g.\lambda\, l. \\ & \qquad \textbf{if } \varepsilon(l) \textbf{ then } \epsilon \\ & \qquad \textbf{elseif } \delta((\boxdot l)|_2^{\mathbb{SN}}, p) \leq r \textbf{ then } (\boxdot l)|_1^{\mathbb{SN}} \boxdot g(\boxplus l) \\ & \qquad \textbf{else } g(\boxplus l))\, \sigma|_2^\Sigma\end{aligned} \tag{26}$$

$$\begin{aligned}\mathcal{C}[\![\textbf{DCont } dirs \ \texttt{int}]\!]\sigma : \mathbb{I}^* = \ & \textbf{let } d^* = \mathcal{DL}[\![dirs]\!] \textbf{ and } j = \mathcal{N}[\![\texttt{int}]\!] \\ & \textbf{in } (\textbf{fix } \lambda\, g.\lambda\, (k, l, f). \\ & \qquad \textbf{if } k = 0 \textbf{ then } f \\ & \qquad \textbf{else let } nl = ns(l, d^*, \sigma|_2^\Sigma) \boxminus f \\ & \qquad\qquad \textbf{in } g(k-1, nl, nl \cup f))\ (j, (\sigma|_1^\Sigma)|_1^{\mathbb{SN}} \boxplus \epsilon, \epsilon)\end{aligned} \tag{27}$$

$$\mathcal{C}[\![\textbf{ECont}]\!]\sigma : \mathbb{I}^* = \epsilon \tag{28}$$

Cette fonction est déjà nettement plus complexe que les précédentes. Notons d'abord que dans notre langage de requêtes, la continuation indique les nœuds où doit se poursuivre l'exécution de la requête. Notre choix sera donc que la fonction $\mathcal{C}$ retourne la liste de tous les identifiants des nœuds qu'il faudra visiter pour évaluer complètement la requête. Pour une continuation vide (28), on retourne simplement la liste vide. Pour les deux autres cas, il y aura une forme de récursivité impliquée, puisque dans le cas base plus portée, il faut trouver tous les identifiants des nœuds présents dans le réseau à l'intérieur de la portée requise alors que dans le cas directions et nombre de sauts autorisés, il faudra aller chercher les voisins du nœud courant, puis les voisins de ces voisins et ainsi de suite jusqu'à avoir atteint le nombre de sauts autorisés. Dans les deux cas, on

calcule d'abord les informations nécessaires à partir des sous-éléments syntaxiques : position de base p et portée r (26) ou directions d^* demandées et nombre de sauts autorisés j (27).

Dans le premier cas (26), la fonction «récursive» g parcourt la liste des nœuds présents dans le réseau récupéré de σ pour ne retenir que les identifiants des nœuds qui sont dans la portée (avant-dernière ligne, appel à la fonction δ). Le point fixe de la fonction g est appliqué à la liste des nœuds présents dans le réseau extraite de σ.

Dans le secon cas (27), la fonction g est un peu plus complexe encore puisqu'elle fait une double récursivité sur le nombre de sauts restant à faire par son premier paramètre k et sur les nœuds à parcourir par son second paramètre l. Elle utilise un troisième paramètre f qui est la liste des identifiants des nœuds déjà trouvés, ce qui permet de filtrer les nouveaux voisins trouvés à chaque étape pour éviter de les reparcourir récursivement ce qui risquerait de ne jamais s'arrêter. Lorsque k est égal à zéro, on a terminé et on retourne la liste des identifiants déjà trouvés f. Sinon, on appelle la fonction ns qui retourne les voisins de tous les nœuds de l dont les nœuds déjà trouvés sont éliminés (opérateur dyadique $\boxminus$) puis on appelle récursivement avec un saut en moins restant, la liste des nouveaux nœuds à parcourir et la liste des nœuds déjà trouvés enrichie des voisins trouvés à cette étape nl mais sans qu'il y ait de répétition (opérateur dyadique $\cup$). Le point fixe de la fonction g est appliqué au tuple j, nombre de sauts autorisés initial et identifiant du nœud courant mis dans une liste et la liste vide comme liste des nœuds déjà trouvés.

Remontons encore d'un cran pour aborder la partie des expressions définissant les requêtes booléennes :

$$\mathcal{BE}[\![\textbf{AndBExp } bexp_1\ bexp_2]\!]\sigma : \mathbb{B} = \mathcal{BE}[\![bexp_1]\!]\sigma \wedge \mathcal{BE}[\![bexp_2]\!]\sigma \tag{29}$$

$$\mathcal{BE}[\![\textbf{OrBExp } bexp_1\ bexp_2]\!]\sigma : \mathbb{B} = \mathcal{BE}[\![bexp_1]\!]\sigma \vee \mathcal{BE}[\![bexp_2]\!]\sigma \tag{30}$$

$$\mathcal{BE}[\![\textbf{NotBExp } bexp]\!]\sigma : \mathbb{B} = \neg\mathcal{BE}[\![bexp]\!]\sigma \tag{31}$$

$$\mathcal{BE}[\![\textbf{SBExp } \texttt{sensorId}]\!]\sigma : \mathbb{B} = sv(\sigma|_1^\Sigma, \mathcal{I}_s[\![\texttt{sensorId}]\!]) \downarrow_{\mathbb{B}}^{\mathbb{SV}} \tag{32}$$

$$\mathcal{BE}[\![\textbf{CExpBExp } cexp]\!]\sigma : \mathbb{B} = \mathcal{CE}[\![cexp]\!]\sigma \tag{33}$$

$$\mathcal{CE}[\![\textbf{EqCExp } rand_1\ rand_2]\!]\sigma : \mathbb{B} = \mathcal{O}[\![rand_1]\!]\sigma = \mathcal{O}[\![rand_2]\!]\sigma \tag{34}$$

$$\mathcal{CE}[\![\textbf{LCExp } rand_1\ rand_2]\!]\sigma : \mathbb{B} = \mathcal{O}[\![rand_1]\!]\sigma < \mathcal{O}[\![rand_2]\!]\sigma \tag{35}$$

$$\mathcal{CE}[\![\textbf{LeqCExp } rand_1\ rand_2]\!]\sigma : \mathbb{B} = \mathcal{O}[\![rand_1]\!]\sigma \leq \mathcal{O}[\![rand_2]\!]\sigma \tag{36}$$

$$\mathcal{CE}[\![\textbf{GCExp } rand_1\ rand_2]\!]\sigma : \mathbb{B} = \mathcal{O}[\![rand_1]\!]\sigma > \mathcal{O}[\![rand_2]\!]\sigma \tag{37}$$

$$\mathcal{CE}[\![\textbf{GeqCExp } rand_1\ rand_2]\!]\sigma : \mathbb{B} = \mathcal{O}[\![rand_1]\!]\sigma \geq \mathcal{O}[\![rand_2]\!]\sigma \tag{38}$$

$$\mathcal{O}[\![\textbf{SRand } \texttt{sensorId}]\!]\sigma : \mathbb{R} = sv(\sigma|_1^\Sigma, \mathcal{I}_s[\![\texttt{sensorId}]\!]) \downarrow_{\mathbb{R}}^{\mathbb{SV}} \tag{39}$$

$$\mathcal{O}[\![\textbf{CRand } \texttt{double}]\!]\sigma : \mathbb{R} = \mathcal{R}[\![\texttt{double}]\!]) \tag{40}$$

$$\mathcal{I}_s[\![\texttt{sensorId}]\!] : \mathbb{I}_s = i_s \quad \text{(non spécifié)} \tag{41}$$

Au dernier niveau de ces expressions se trouvent les opérandes simples qui sont ici soit des constantes réelles soit des identifiants de capteur qui vont servir à aller cherche leur valeur dans le nœud courant. La fonction d'évaluation $\mathcal{O}$ traite ces deux cas d'opérandes en appelant les fonctions $\mathcal{R}$ (40) et $\mathcal{I}_s$ (39) pour obtenir les valeurs des domaines $\mathbb{R}$ ou $\mathbb{I}_s$, cette dernière étant utilisé dans l'appel à la fonction sv pour obtenir la valeur du capteur correspondant dans le nœud courant (contenu dans σ). Notez que dans ce cas, seules les valeurs de capteurs réelles sont admises, d'où l'extraction de la valeur du domaine $\mathbb{SV}$ mais de type $\mathbb{R}$.

Pour les expressions de comparaison, la fonction d'évaluation $\mathcal{CE}$ est constituée de cinq définitions correspondant à chacun des opérateurs de comparaison. Ces définitions évaluent les deux opérandes puis applique la comparaison requise entre les deux résultats. Notez que les opérateurs en partie droite sont les opérations définies sur $\mathbb{R}$ et qui retournent un résultat dans $\mathbb{B}$. Les expressions booléennes sont traitées par la fonction d'évaluation $\mathcal{BE}$ de manière assez similaire aux expressions de comparaison. Le cas d'un identifiant de capteur donne aussi lieu à l'obtention de sa valeur dans le nœud courant (32), mais cette fois l'extraction se fait du domaine $\mathbb{SV}$ vers le domaine $\mathbb{B}$ puisqu'il s'agit ici d'avoir nécessairement une valeur booléenne. Le cas d'une expression de comparaison (33) fait appel à la fonction $\mathcal{CE}$ pour l'évaluer.

Passons maintenant aux requêtes de collecte de données de capteurs :

$$\mathcal{G}[\![\textbf{RGather } \texttt{sensorId } \textit{gather}]\!]\sigma : \Phi^* = \mathcal{V}[\![\texttt{sensorId}]\!]\sigma \boxdot \mathcal{G}[\![\texttt{gather}]\!]\sigma \tag{42}$$

$$\mathcal{G}[\![\textbf{FGather } \texttt{sensorId}]\!]\sigma : \Phi^* = \mathcal{V}[\![\texttt{sensorId}]\!]\sigma \boxdot \epsilon \tag{43}$$

$$\mathcal{V}[\![\texttt{sensorId}]\!]\sigma : \Phi = \textbf{let } i_s = \mathcal{I}_s[\![\texttt{sensorId}]\!] \textbf{ in } \langle (\sigma|_1^\Sigma)|_1^{\mathbb{SN}}, i_s, sv(\sigma|_1^\Sigma, i_s)\rangle_\Phi \tag{44}$$

Comme indiqué précédemment, les requêtes de collecte de données de capteurs vont parcourir un certain nombre de nœuds (nœud recevant initialement la requête puis les nœuds identifiés dans la continuation) et pour chaque nœud visité, collecter les valeurs des capteurs indiqués dans une liste d'identifiants de capteurs énumérés par la forme syntaxique *gather*. La fonction d'évaluation $\mathcal{G}$ gère cette liste en deux définitions, la première pour le cas récursif qui évalue le capteur indiqué et fait l'appel récursif à $\mathcal{G}$ sur le reste de la liste alors que la seconde se contente d'évaluer le capteur indiqué et mettre le résultat dans un liste. Dans les deux cas, l'évaluation de la valeur d'un capteur à partir de son identifiant est faite par la fonction $\mathcal{V}$ qui construit une valeur du domaine Φ à partir de l'identifiant du nœud courant, de celui du capteur et de sa valeur du domaine $\mathbb{SV}$.

Enfin, la fonction d'évaluation $\mathcal{Q}$ traite les deux cas de requêtes, de collecte de données de capteurs et booléennes :

$$\begin{aligned}\mathcal{Q}[\![\textbf{GQuery } \textit{gather cont}]\!]\sigma : \Gamma = \textbf{let } i^* &= C[\![\textit{cont}]\!]\sigma \\ \textbf{in } &\uparrow^\Gamma_{\Phi^*} ((\textbf{fix } \lambda\, f.\lambda(i^*, \textit{sigma}). \\ &\qquad \textbf{if } \varepsilon(i^*) \textbf{ then } \epsilon \\ &\qquad \textbf{else let } \sigma' = \langle sn(\boxdot i^*, \sigma|_2^\Sigma)\rangle_\Sigma \\ &\qquad\quad \textbf{in } \mathcal{G}[\![\textit{gather}]\!]\sigma' \boxplus f(\boxplus i^*, \sigma')) \\ &\quad (((\sigma'|_1^\Sigma)|_1^{\mathbb{SN}}) \boxdot i^*, \sigma))\end{aligned} \tag{45}$$

$$\begin{aligned}\mathcal{Q}[\![\textbf{BQuery } \textit{bexp cont}]\!]\sigma : \Gamma = \textbf{let } i^* &= C[\![\textit{cont}]\!]\sigma \\ \textbf{in } &\uparrow^\Gamma_{I^*} ((\textbf{fix } \lambda\, f.\lambda(i^*, \textit{sigma}). \\ &\qquad \textbf{if } \varepsilon(i^*) \textbf{ then } \epsilon \\ &\qquad \textbf{else let } \sigma' = \langle sn(\boxdot i^*, \sigma|_2^\Sigma)\rangle_\Sigma \\ &\qquad\quad \textbf{in } (\textbf{if } \mathcal{BE}[\![\textit{bexp}]\!]\sigma' \textbf{ then } (\sigma'|_1^\Sigma)|_1^{SN} \boxdot \epsilon \\ &\qquad\qquad \textbf{else } f(\boxplus i^*, \sigma')) \\ &\quad (((\sigma'|_1^\Sigma)|_1^{\mathbb{SN}}) \boxdot i^*, \sigma))\end{aligned} \tag{46}$$

Ici, encore, pour traiter la continuation, on utilise des points fixes de fonction pour obtenir l'effet d'un appel récursif. Dans les deux cas, on commence par évaluer la continuation, ce qui donne une liste d'identifiants de nœuds à parcourir. On ajoute l'identifiant du nœud courant au début de cette liste puis on applique à cette nouvelle liste le point fixe d'une fonction f qui dépend du cas à traiter. Dans le cas des requêtes de collecte, on doit retourner une liste de valeurs du domaine Φ ; cette liste est vide si on a une liste d'identifiants de nœuds vide, et si la liste d'identifiants de nœuds n'est pas vide, on évalue l'expression de collecte par appel à $\mathcal{G}$ et on ajoute le résultta au début de la liste obtenue par l'appel «récursif» sur le reste de la liste d'identifiants de nœuds. Dans le cas des requêtes booléennes, le traitement est similaire, sauf qu'il faut tester le résultat de l'évaluation de l'expression booléenne sur le nœud courant et n'ajouter son identifiant au résultat que si le résultta de celle-ci est vrai. Notez dans les deux cas que le passage d'un nœud au suivant requiert de modifier la valeur de σ pour obtenir un σ' où le nœud courant devient celui correspondant à l'identifiant suivant dans la liste des identifiants des nœuds à traiter.

A.4 Implantation et passage à une sémantique opérationnelle répartie

Pour implanter un langage dont on a une sémantique formelle, il faut faire des choix d'implantation puis réaliser cette implantation elle-même. La sémantique formelle sert à la fois de guide pour structurer cette implantation et de référence pour en vérifier la conformité à la spécification du langage.

Pour l'évaluation, les parties expression de collecte (*gather*) et expression booléenne (*bexp*) pourront être évaluées en suivant directement la sémantique fournie. Un point important sera de pouvoir aller chercher les valeurs des capteurs du nœud sur lequel la requête est exécutée. L'implantation de cette partie du langage ne diffèrera pas d'une version à l'autre des versions qui devront être développées au long du semestre.

Pour ce qui concerne le traitement de la continuation par contre, il ne sera pas réaliste ni performant de suivre la sémantique dans notre contexte réparti sous composants BCM4Java car elle exigerait d'avoir un accès direct à tous les nœuds du réseau de capteur à partir d'un évaluateur unique centralisé, alors qu'en pratique on ne peut accéder ces nœuds que via les connexions entre composants BCM4Java. L'idée générale pour traiter le passage de la requête à la continuation va être d'accumuler progressivement les résultats de la requête et passer aux nœuds suivants par un appel entre composants BCM4Java. Pour identifier les composants auxquels il faut passer le requête pour poursuivre son exécution, il faudra simplement identifier les voisins du nœud courant qui font partie de la continuation et leur passer la requête pour qu'ils en poursuivent l'exécution. Le principal souci de cette approche sera d'éviter d'évaluer plusieurs fois la requête sur un même nœud accédé par des chemins différents. Par exemple, si une requête à continuation directionnelle part dans les directions nord-est et sud-est est envoyée au nœud n7 (figure 1), elle sera d'abord envoyée aux nœuds n5 et n10. Ces derniers devraient ensuite l'envoyer tous les deux au nœud n8, mais il faudra que n8 n'exécute qu'une seule fois cette requête.

De plus, entre la première et la seconde partie du projet, deux implantations différentes seront utilisées pour évaluer la continuation de la requête. Dans la première partie, vous utiliserez ce qu'on appelle un style directe par appel synchrone. C'est tout simplement une approche similaire à ce qui se fait en programmation séquentielle classique : le nœud courant appelle chacun de ses voisins faisant partie de la continuation puis récupérer les résultats, les concatène en un seul résultat qui est retourné à son appelant. Dans ce style direct, lorsqu'un nœud n'a pas de voisin qui fait partie de la continuation (parce qu'on est arrivé au bout du nombre de sauts autorisés, par exemple), il calcule son résultat local et le retourne simplement à son appelant.

Dans la seconde partie, comme expliqué dans le cahier des charges précédemment, il faudra utiliser un style dit de passage à la continuation par appel asynchrone où le nœud courant reçoit du nœud précédent le résultat accumulé avant d'arriver à lui, calcule son résultat local de la requête, l'ajoute au résultat précédent et passe ce nouveau résultat à chacun de ses voisins qui font partie de la continuation avec la requête pour que ces derniers poursuivent le calcul. Dans ce style, lorsqu'un nœud n'a pas de voisin qui fait partie de la continuation, il ajoute son résultat local aux résultat précédent qu'il a reçu et il renvoie le tout directement au composant qui a lancé initialement la requête. On constate que dans ce cas, à partir du moment où un nœud a plusieurs voisins faisant partie de la continuation, plusieurs évaluations de cette requête vont être lancées dans différentes directions et donc qu'une arborescence d'appels va se développer jusqu'à arriver aux feuilles (les nœuds qui n'ont pas de voisin qui fait partie de la continuation). Par exemple, la requête décrite précédemment reçue par n7 va devoir être envoyée à la fois à n5 et à n10, puis n5 l'enverra à n3 et n8 alors que n10 l'enverrra à n8 et à n13 ; en supposant que n8 n'exécute la requête qu'une seule fois, les trois nœuds sont terminaux dans l'arbre des appels et vont dont tous les trois renvoyer le résultat de la requête au composant client. En conséquence, ce composant client va recevoir autant de résultats qu'il y a de feuilles dans cet arborescence d'appels qu'il lui faudra recomposer en un unique résultat avant de l'utiliser. C'est là aussi un point critique de la réalisation du projet.

B 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.

B.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.

B.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 :

"2024-01-31T09:00:00.00Z"

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 2024 à 9 heures du matin.

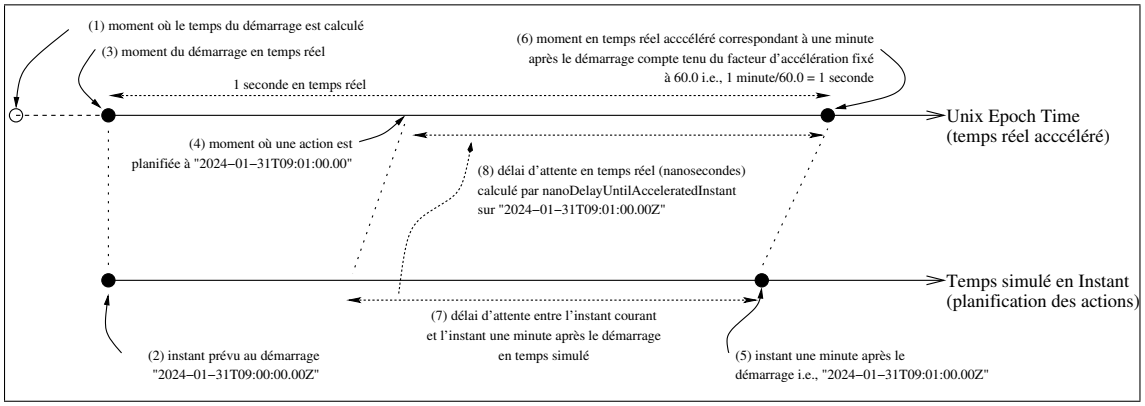

FIGURE 10 – Les deux références temporelles mises en synchronisation par `AcceleratedClock` : le temps réel accéléré s'appuyant sur l'horloge matérielle de l'ordinateur via le système d'exploitation et le temps virtuel des actions du scénario de test représenté par des `Instant` de Java.

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("2024-01-31T09:00:00.00Z");
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 10. 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.

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

Maintenant que l'on a vu les principales fonctionnalités de l'horloge accélérée, [11] 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 :

11. 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();
```

B.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.