TP1
Algorithmes de diffusion avec JBotSim

Le but de ce premier TP est de prendre en main la librairie JBotSim, qui nous permettra de simuler un système distribué pour y exécuter différents algorithmes.

1 Création d'un projet Java avec JBotSim

Nous utiliserons l'IDE **VSCode** pour les TPs, et toutes les instructions données ici présument que vous êtes sous VSCode. Si vous êtes sur votre machine personnelle vous pouvez utiliser un autre éditeur, mais vous serez alors livrés à vous-mêmes.

Dans VS Code, commencez par ouvrir un dossier vide à l'emplacement de votre choix (`File -> Open Folder`).

Vous aurez besoin d'installer les extensions suivantes pour faire du Java dans VSCode :

```
Language Support for Java(TM) by Red Hat
Debugger for Java
```

Pour les installer, rendez-vous dans le menu `Extensions` à gauche, puis utilisez la barre de recherche.

Il faut ensuite ajouter la librairie JBotSim au projet. Créez un dossier nommé «lib» dans votre projet (clic droit sur le projet dans le menu à gauche puis `New Folder`). Téléchargez ensuite le fichier .jar de la librairie en copiant l'adresse ci-dessous dans votre navigateur web :

```
https://github.com/jbotsim/JBotSim/releases/download/v1.2.0/jbotsim-standalone-1.2.0.jar
```

Stockez le fichier .jar dans le dossier lib sans le décompresser.

Finalement, créez un dossier «src» dans lequel vous placerez votre code Java.

2 Prise en main de JBotSim

Pour vérifier que tout fonctionne, nous allons créer un programme minimal. Créez un nouveau fichier dans le dossier src (clic droit sur src puis `New File`) et nommez-le «TP1Main.java».

Copiez-collez le code minimal suivant dans votre nouveau fichier :

```java
import io.jbotsim.core.Topology;
import io.jbotsim.ui.JViewer;

public class TP1Main {
    public static void main(String[] args){ // On déclare le programme principal
        Topology tp = new Topology(); // Création d'un nouveau système distribué
        new JViewer(tp); // On active l'interface graphique de JBotSim
        tp.start(); // On démarre le tout
    }
}
```

Si l'indentation est perdue lors du copier-coller dans VSCode, vous pouvez utiliser `Ctrl+A` puis `Shift+Ctrl+I` pour réparer l'indentation automatiquement.

Sauvegardez (`Ctrl+S`) et appuyez sur `F5` pour tester votre programme.

Si tout s'est bien passé, vous devriez voir apparaître une fenêtre grise. Il s'agit de l'interface graphique de JBotSim. Elle dispose des fonctionnalités suivantes :

— **Clic gauche** dans le canevas : créer un nouveau noeud.
— **Clic droit** sur un noeud existant : supprimer le noeud.
— **Clic gauche maintenu** sur un noeud existant : déplacer le noeud.
— Si deux noeuds sont suffisamment proche, un lien de communication est automatiquement créé entre les deux.
— **Ctrl+clic** (ou bien **clic molette**) permet de **sélectionner** un noeud. Pour l'instant cela n'a aucun effet, mais nous utiliserons cette fonctionnalité par la suite pour démarrer nos algorithmes.

Cette interface vous permet de construire graphiquement la topologie de votre système distribué. Familiarisez vous avec elle, nous allons l'utiliser tout au long du TP. Fermez ensuite la fenêtre et retournez sous VSCode.

Pour programmer un système distribué avec JBotSim, le programme principal ne suffit pas : il faut également créer une classe de noeuds. Créez un nouveau fichier (clic droit sur src puis `New File`) et nommez le «NoeudTest.java». Lisez le code suivant puis copiez-collez le dans le nouveau fichier :

```
import io.jbotsim.core.Node;

public class NoeudTest extends Node { // Une classe de noeud doit hériter de Node
    public void onStart() { // Cette méthode est appelée lorsqu'un nouveau noeud  est ajouté
        System.out.println("Le noeud numéro " + getID() + " vient d'être créé!");
    }

    public void onSelection() { // Cette méthode est appelée lors d'un Ctrl+clic sur un noeud
        System.out.println("Vous avez sélectionné le noeud numéro " + getID() + "!");
    }
}
```

Il faut également informer JBotSim que tous les noeuds créés doivent appartenir à cette classe. Dans TP1Main.java, ajoutez la ligne suivante après la création de la topologie :

```
tp.setDefaultNodeModel(NoeudTest.class);
```

Exécutez à nouveau TP1Main. Créez quelques noeuds avec des clics gauche dans le canevas, puis faites un Ctrl+clic sur certains d'entre eux. Vous verrez des messages apparaître dans la console de VSCode.

Nous allons maintenant voir comment les noeuds peuvent communiquer entre eux en s'envoyant des messages. Remplacez le contenu de NoeudTest.java par le code suivant :

```
import io.jbotsim.core.Color;
import io.jbotsim.core.Message;
import io.jbotsim.core.Node;

public class NoeudTest extends Node {
    public void onSelection() {
        System.out.println("Vous avez sélectionné le noeud numéro " + getID() + "!");
        Message m = new Message("Bonjour voisin!");
        // Ici nous utilisons des chaînes, mais un message peut contenir n'importe quel objet.
        sendAll(m); // sendAll envoie le message à tous mes voisins

        if (getNeighbors().size() > 0) { // Si il y a au moins un voisin...
            Node voisinPrefere = getNeighbors().get(0); // Le premier noeud de la liste des voisins
            m = new Message("Tu es mon voisin préféré!");
            send(voisinPrefere, m); // send envoie un message à un seul noeud
        }
    }

    // Cette méthode est appelée lors de la réception d'un message
    public void onMessage(Message m) {
        System.out.println("Le noeud " + getID() + " a reçu le message suivant du noeud "
            + m.getSender() + ": \"" + m.getContent() + "\"");
        if (((String) m.getContent()).contains("préféré"))
            setColor(Color.RED); // Les noeuds rougissent quand ils reçoivent un compliment
    }
}
```

Étudiez ce code pour bien le comprendre, puis testez le.

Finalement, la mémoire locale d'un processus est simulée à l'aide des attributs de la classe, comme dans l'exemple suivant :

```
import io.jbotsim.core.Node;

public class NoeudTest extends Node {
    private int compte; // Chaque noeud garde un entier en mémoire

    public void onStart() {
        compte = 0; // onStart est souvent utilisé pour initialiser les variables
    }

    public void onSelection() {
        compte++;
        System.out.println("Vous avez cliqué sur ce noeud " + compte + " fois!");
        // Le nombre affiché augmente à chaque clic, mais est différent pour chaque noeud
    }
}
```

Exécutez également cet exemple. Vous avez maintenant tous les outils pour programmer un algorithme distribué avec des processus disposant d'une mémoire et capables d'échanger des messages !

3 Diffusion dans un graphe

Dans cet exercice, nous allons programmer un algorithme de diffusion très simple pour propager une information dans un graphe. L'algorithme est le suivant :

- Un processus (sélectionné par Ctrl+clic) démarre la diffusion d'un message en le transmettant à tous ses voisins.
- La première fois qu'un noeud reçoit le message, il change de couleur et retransmet le message à tous ses voisins.

Cet algorithme n'est évidemment pas optimal.

Notez qu'il est nécessaire que chaque noeud se souvienne s'il a déjà reçu le broadcast ou non : il faut donc une mémoire locale.

Question 1. Programmez cet algorithme, puis testez le. Il vous faudra pour ça créer une nouvelle classe de noeud et définir les fonctions onStart(), onSelection() et onMessage().

Le changement de couleur vous permettra de voir l'algorithme progresser. Si l'algorithme se déroule trop vite pour voir la progression, vous pouvez ajouter cette ligne dans votre classe main avant le démarrage de la simulation :

```
tp.setTimeUnit(1500); // Définit l'intervalle de temps entre deux ticks d'horloge dans la simulation
```

Afin de mesurer l'efficacité de l'algorithme, on voudrait maintenant compter le nombre de messages envoyés durant l'exécution. Ce serait très compliqué dans un vrai système distribué, mais grâce au simulateur nous pouvons tricher et utiliser de la mémoire partagée. Dans votre classe noeud, vous pouvez utiliser un attribut statique comme suit :

```
private static int compte = 0; // Cette variable est partagée par tous les noeuds
```

Il suffit alors d'incrémenter la variable partagée à chaque fois qu'on envoie un message. Notez qu'avec un sendAll(), il faut compter plusieurs envois de messages !

Question 2. Modifiez votre code pour compter les messages envoyés.

Pour pouvoir afficher le nombre total de messages envoyés à la fin de l'exécution, il faut pouvoir détecter la terminaison de l'algorithme (le dernier noeud à recevoir le message affiche le compte total). Or, détecter la terminaison d'une diffusion dans un graphe est très coûteux. Nous allons donc tricher une nouvelle fois et utiliser une autre variable partagée pour compter le nombre de noeuds qui ont déjà reçus le message. Vous aurez aussi besoin de la fonction suivante pour savoir si tous les noeuds ont reçu le message :

```
getTopology().getNodes().size() // Retourne le nombre de noeuds total dans le système
```

Après réception d'un message, si je suis le dernier noeud à recevoir le message, alors je peux afficher le compte des messages envoyés.

Question 3. Modifiez votre programme pour afficher le compte des messages envoyés à la fin de l'exécution.

Question 4. Testez votre programme avec un graphe complet (tous les noeuds sont connectés à tous les autres) puis avec un anneau contenant le même nombre de noeuds (chaque noeud est connecté à exactement deux autres). Comptez les messages envoyés dans les deux cas. Que constatez vous ?

Question 5. Comptez les messages envoyés dans un graphe complet de 4 noeuds. À votre avis, quel est le nombre de messages minimal nécessaire pour une diffusion dans ce système ?

4 Construction d'un arbre couvrant

Afin de rendre la diffusion plus efficace, nous allons construire un arbre couvrant en utilisant l'un des algorithmes du cours. Il s'agit de l'algorithme le plus simple, donné sur les slides 16-17 du CM2. Le but est de construire un arbre qui comprends tous les noeuds du système, mais seulement certains des liens. La diffusion est ensuite effectuée sur l'arbre plutôt que sur le graphe, ce qui permet de bien meilleures performances.

L'algorithme utilise trois types de messages différents : JOIN, BACK, et BACKNO. Pour différencier les types de messages, vous pourrez par exemple utiliser le contenu des messages comme suit :

```
Message m = new Message("JOIN");
```

Le récepteur pourra alors vérifier le type du message comme suit :

```
if (m.getContent().equals("JOIN")) { ... }
```

Dans l'algorithme, un message JOIN est une invitation à rejoindre l'arbre qui est en cours de construction. L'émetteur du message propose de devenir le père du récepteur dans le nouvel arbre.

Les messages BACK et BACKNO sont des réponses à un message JOIN. Un message BACK signifie : «j'accepte ton invitation, je suis maintenant ton fils dans l'arbre». Un message BACKNO signifie : «je refuse ton invitation car quelqu'un d'autre m'a invité en premier».

4.1 Étape 1 : JOIN

Pour simplifier les choses, nous allons construire l'algorithme en plusieurs étapes. Pour la première étape, nous allons nous contenter de l'algorithme suivant :

- Le processus qui souhaite diffuser des messages devient jaune puis envoie un message JOIN à tous ses voisins pour débuter la construction d'un arbre couvrant ;
- La première fois qu'un processus reçoit un message JOIN, il rejoint l'arbre et choisit l'émetteur du message comme père dans l'arbre. Il change aussi de couleur et devient jaune. Il envoie ensuite à tous ses voisins (sauf son père) un message JOIN.

Cette première étape est très similaire à l'algorithme de l'exercice précédent, à ceci près que les noeuds n'envoient pas leur message JOIN à leur père.

Chaque processus doit disposer d'un attribut «pere» de type Node qui lui permettra de mémoriser sa place dans l'arbre.

Question 1. Programmez cette première étape de l'algorithme. Vous pouvez ensuite la tester et vérifier que tous les noeuds deviennent jaune.

Bonus : Lorsque le père d'un noeud est déterminé, vous pouvez mettre en valeur cette relation dans le graphe en changeant la couleur du lien entre les deux. Cela vous permettra de visualiser l'arbre au fur et à mesure de sa construction. Vous pouvez accéder à l'objet Link représentant le lien grâce à :

```
getCommonLinkWith(pere); // Retourne l'objet Link qui relie ce noeud et son père
```

La classe Link dispose de méthodes setColor() et setWidth().

4.2 Étape 2 : BACK et BACKNO

À l'issue de l'étape 1, votre programme est déjà capable de construire un arbre couvrant. Cependant, la fin de la construction de l'arbre n'est pas détectée par la racine, qui ne sait donc pas qu'elle peut débuter la diffusion. D'autre part, les noeuds ne savent pas qui sont leurs enfants. Nous allons résoudre ces problèmes avec les messages BACK et BACKNO.

Dans cette seconde étape, nous ajoutons les éléments suivants à notre algorithme :

- Après avoir rejoint l'arbre, si le processus n'a qu'un seul voisin (son père), alors il envoie un message BACK à son père puis devient vert ;
- Lorsqu'un processus reçoit un message JOIN alors qu'il a déjà rejoint l'arbre, il renvoie à l'émetteur un message BACKNO pour indiquer son refus ;
- Une fois qu'un processus a reçu une réponse (BACK ou BACKNO) de tous ceux à qui il avait envoyé un JOIN, il envoie un message BACK à son père (sauf la racine qui n'a pas de père !) et devient vert ;
- Lorsqu'un processus reçoit un message BACK, il ajoute l'émetteur à la liste de ses enfants ;
- Lorsque la racine (l'émetteur du message initial) a reçu une réponse (BACK ou BACKNO) de tous ses voisins, elle considère que la construction de l'arbre est terminée.

De nouveaux attributs doivent être ajoutés à la mémoire de chaque noeud :

- La racine doit se souvenir qu'elle est la racine ;
- Chaque noeud doit mémoriser la liste de ses enfants (il s'agira d'un attribut de type List<Node>) ;
- Chaque noeud doit compter les réponses qu'il a reçu à son message JOIN.

Question 2. Programmez cette seconde étape de l'algorithme. Testez la et vérifiez que tous les noeuds finissent par devenir vert.

4.3 Étape 3 : diffusion

Nous avons désormais un arbre ! Chaque noeud connaît son père et ses enfants, et la racine est prévenue de la fin de la construction de l'arbre. Il ne reste plus qu'à effectuer une diffusion. Afin d'éviter que le message diffusé ne soit confondu avec les messages utilisés pour la construction de l'arbre, nous allons utiliser un type de message supplémentaire : BCAST. L'algorithme pour cette étape est très simple :

- Une fois la construction de l'arbre terminée, la racine devient rouge puis lance la diffusion en envoyant un message de type BCAST à tous ses enfants (et non pas ses voisins !) ;
- Lorsqu'un noeud reçoit un message BCAST, il devient rouge puis retransmet le message à ses enfants.

Question 3. Programmez la diffusion. Testez et vérifiez que tous les noeuds deviennent rouge.

4.4 Dernière étape : comptage des messages

On voudrait maintenant vérifier l'efficacité de notre nouvel algorithme. Comme dans l'exercice précédent, nous allons tricher en utilisant des variables partagées (static) pour compter les messages envoyés, puis détecter la terminaison de la diffusion en comptant le nombre de noeuds ayant reçus le message BCAST.

Il faut compter séparément les messages BCAST (pour la diffusion) et les autres messages (pour la construction de l'arbre couvrant), puis afficher les deux totaux à la fin de l'exécution.

Question 4. Programmez le comptage des messages et testez le.

Question 5. Combien de message faut il à votre programme pour effectuer une diffusion dans un graphe complet de 4 noeuds ? Sachant qu'un même arbre couvrant peut être réutilisé pour plusieurs diffusions, qu'en concluez vous ?