Master 2 Réseaux - NRES - TP : réseaux de capteurs

Stéphane Devismes Sébastien Tixeuil

Résumé

L'objet de ce TP est la découverte de la plateforme de simulation **Sinalgo**. **Sinalgo** est une plateforme de simulation qui permet de tester et de valider "à haut niveau" des protocoles pour réseaux de capteurs.

1 Installation

Les sources et le tutorial de **Sinalgo** se trouvent à l'adresse suivante (télécharger la version "Regular Release") :

`http://dcg.ethz.ch/projects/sinalgo/`

1.1 Pré-requis

Sinalgo nécessite que **Java 5.0** ou une version supérieure soit installé sur votre machine. **Sinalgo** fonctionne aussi avec des environnnement de développement de type **Eclipse**.

1.2 Installation

Téléchargez la version "Regular Release" de **Sinalgo** et décompressez la dans un sous-répertoire *sinalgo* de votre répertoire de travail. Ce répertoire contient entre autres *les sources* dans *src* et les fichiers binaires dans *binaries/bin*.

Deux choix s'offrent alors à vous :

- Soit vous lancez **Sinalgo** à partir de la ligne de commande en tapant : `java -cp binaries/bin sinalgo.Run` à partir du répertoire *sinalgo*.
- Soit vous utilisez un environnement de développement type **Eclipse** (*cf.* ci-dessous).

1.3 Avec Eclipse

1. Lancez **Eclipse**.
2. Créez un nouveau projet java (File→New→Java Project). Nommez-le *sinalgo*. Cochez la case "créer un projet à partir d'un source existant" et sélectionnez le répertoire *sinalgo* créé précédemment. Cliquez sur *terminer*.
3. Vérifiez qu'**Eclipse** est configuré pour utiliser **Java 5.0** : dans le menu *Preferences*, selectionnez *Java→Compiler*. Le niveau du compilateur doit être à 5.0 ou plus.

Une fois ces actions effectuées, il suffit de faire un clic droit sur *src* dans le navigateur d'**Eclipse** et de sélectionner *Run as→Java* Application pour démarrer l'application.

2 Organisation

Sinalgo propose un ensemble de sources de projets : des exemples (*sample*), un projet par défaut (*defaultProject*) et un projet "vide" (*template*). Chaque source est stocké dans un répertoire situé dans `sinalgo/src/projects`. L'organisation de chaque projet est identique et comporte :

- Un fichier *Config.xml* qui permet de paramètrer le modèle d'exécution.
- Un fichier *description.txt* ; l'ensemble des informations de ce fichier sont affichés dans le menu où l'on sélectionne la simulation à exécuter.
- Un fichier *CustumGlobal.java* qui permet de modifier l'interface de simulation.
- Un fichier *LogL.java* qui permet de développer des fonctions de "monitoring".
- Optionnellement, un fichier *run.pl* ; ce fichier *Perl* permet de gérer le paramètrage d'une exécution.
- Un répertoire *images* ; ces images peuvent être utilisées pour personnaliser les menus, etc.
- Deux répertoires : *models* et *nodes* qui contiennent les implémentations des modèles et des noeuds (*cf.* ci-dessous).

2.1 Modèles

Les modèles décrivent l'environnement dans lequel votre protocole est simulé. Vous pouvez implanter vos propres modèles ou utiliser des modèles prédéfinis. Ici, nous utiliserons et paramètrerons uniquement des modèles prédéfinis. Pour chaque simulation, il faut définir 6 modèles différents :

- Le *modèle de connectivité* qui décide comment évaluer le voisinage d'un noeud.
- Le *modèle de distribution* qui décrit comment les noeuds sont placés au démarrage de l'application.
- Le *modèle d'interférence* qui décide si un message est perdu suite à une interférence.
- Le *modèle de transmission* qui décide du temps d'acheminement des messages.
- Le *modèle de mobilité* qui gére le mouvement des noeuds.
- Le *modèle de fiabilité* qui gére la perte de messages.

2.2 Noeuds

Le répertoire *nodes* contient le code du protocole. Ce code est subdivisé en plusieurs fichiers répartis dans plusieurs répertoires :

- *edges*. Contient le code relatif aux liens de communication (ici, nous utiliserons uniquement l'implantation par défaut donc ce répertoire ne contiendra aucun fichier).
- *messages*. Contient la description de chaque type de message utilisé dans la simulation.
- *nodeImplementations*. Contient le code pour le comportement des noeuds.
- *timers*. Contient le code des *timers* utilisés dans la simulation.

3 Un petit exemple...

Lancez une exécution de l'exemple *sample1*. Vous pouvez le lancer via **Eclipse** ou directement en tapant :

```
java -cp binaries/bin sinalgo.Run -project sample1
```

Une fenêtre apparait alors avec un plan en 3 dimensions. Pour lancer la simulation, allez dans le menu *Simulation* et cliquez sur *Generate Nodes*. Créez 100 noeuds puis cliquez sur l'icone verte pour démarrer la simulation.

Vous pouvez ensuite paramétrer la simulation directement à partir de la ligne de commande. Par exemple, taper :

```
java -cp binaries/bin sinalgo.Run -project sample1 -gen 1000 sample1 :S1Node
RandomC=UDG -rounds100.
```

Cette commande lance la simulation *sample1* avec 1000 noeuds de type $S1Node$, ces noeuds sont placés de manière aléatoire, le modèle de connectivté utilisé est UDG, enfin la simulation s'arrête après 100 unités de temps.

4 1er protocole : coloriage

Le premier protocole que nous allons écrire est très simple : au début chaque noeud tire au hasard une couleur parmi k couleurs possible. Ensuite, chaque noeud envoie périodiquement sa couleur à tous ses voisins. Lorsqu'un noeud détecte que l'un de ses voisins à une couleur identique à la sienne, il tire au hasard une nouvelle couleur parmi celles non-utilisées dans son voisinage.

Pour créer notre simulation, nous allons tout d'abord faire une copie du répertoire *template* situé dans le répertoire *src/ projects* dans le même répertoire. Cette copie sera re-nommée *coloriage*.

Nous allons ensuite créer de nouveau type de noeuds, messages, et timers. Pour cela, téléchargez les fichiers *CMessage.java*, *CNode.java CTimer.java* situés à l'adresse du cours NRES.

Copiez le fichier *CMessage.java* dans le répertoire *nodes/messages*. Ce fichier contient le code suivant :

```
package projects.coloriage.nodes.messages;
import sinalgo.nodes.messages.Message;

/* description de l'unique type de message utilisé dans l'application */
public class CMessage extends Message {
```

```
public int id;
public int couleur;

public CMessage(int id, int couleur) {
this.id=id;
this.couleur = couleur;
}

public Message clone() {
return new CMessage(id,couleur);
}
}
```

Copiez le fichier *CTimer.java* dans le répertoire *nodes/timers*. Ce fichier contient le code suivant :

```
package projects.coloriage.nodes.timers;
import projects.coloriage.nodes.nodeImplementations.CNode;
import projects.coloriage.nodes.messages.*;
import sinalgo.nodes.timers.Timer;

/* Description de l'unique timer utilisé dans l'application */
public class CTimer extends Timer {
CNode sender;
int interval;

public CTimer(CNode sender, int interval) {
this.sender = sender;
this.interval = interval;
}

/* La fonction "fire" est appelée lorsque le timer expire */
public void fire() {
// le noeud crée un message contenant sa couleur
CMessage msg= new CMessage(sender.ID,sender.getCouleur());
// le noeud envoie le message à tous ses voisins
    sender.broadcast(msg);
    // le noeud relance un nouveau timer
this.startRelative(interval, node); // recursive restart of the timer
}
}
```

Copiez le fichier *CNode.java* dans le répertoire *nodes/nodeImplementations*. Ce fichier contient le code suivant :

```
package projects.coloriage.nodes.nodeImplementations;
import java.awt.Color;
import java.awt.Graphics;
import java.util.*;
import sinalgo.configuration.WrongConfigurationException;
import sinalgo.gui.transformation.PositionTransformation;
import sinalgo.nodes.Node;
import sinalgo.nodes.edges.Edge;
import sinalgo.nodes.messages.Inbox;
import projects.coloriage.nodes.timers.*;
import projects.coloriage.nodes.messages.*;
import sinalgo.nodes.messages.Message;

/* La classe "donnee" ci-dessous est utilisée
 * pour stocker l'état d'un voisin, ici sa couleur.
```

```
 * Les états de tous les voisins
 * seront ensuite stockés dans une table de hachage */
class donnee
{
int couleur;

donnee(int couleur){
this.couleur=couleur;
}
}

/* La classe "CNode" ci-dessous implémente le code de chaque noeud */
public class CNode extends Node {

private int couleur; // la couleur du noeud
/*  ci-dessous "nb" représente le nombre de couleurs total
 *  le tableau "tab" stocke les codes couleur */
private final int nb = 10;
private final Color tab[] = {Color.BLUE,Color.CYAN,Color.GREEN,
        Color.LIGHT_GRAY,Color.MAGENTA,Color.ORANGE,Color.PINK,Color.RED,
        Color.WHITE,Color.YELLOW};

/* La table de hachage "etatvoisin" stocke le dernier état connu
 * de chaque voisin */
private Hashtable<Integer,donnee> etatvoisin;

public int getCouleur(){
return couleur;
}

public Color RGBCouleur(){
return tab[getCouleur()];
}

public void setCouleur(int c) {
this.couleur=c;
}


/* La fonction ci-dessous
 * est utilisée pour tirer au hasard une couleur parmi les nb disponibles */
public void initCouleur(int range){
setCouleur((int) (Math.random() * range) % range);
}

/* La fonction "compute" est lancée à chaque réception
 * de message. Elle permet de changer la couleur du noeud si nécessaire */
public void compute(){
boolean same=false;
Iterator<Edge> it=this.outgoingConnections.iterator();
boolean SC[]=new boolean[nb];

for (int i=0;i<SC.length;i++)
SC[i]=false;

while(it.hasNext()){
Edge e=it.next();
donnee tmp=etatvoisin.get(new Integer(e.endNode.ID));
```

```
if(tmp!=null){
if(tmp.couleur==this.getCouleur()){
same=true;
}
SC[tmp.couleur]=true;
}
}

if (same){
int dispo=0;
for (int i=0;i<SC.length;i++)
if(SC[i]==false) dispo++;
if (dispo == 0) return;
int choix= ((int) (Math.random() * 10000)) % dispo + 1;
int i=0;
while(choix > 0){
if(SC[i]==false)
choix--;
if(choix>0) i++;
}
this.setCouleur(i);
}
}

/* La fonction ci-dessous est appelée à chaque réception de message */
public void handleMessages(Inbox inbox) {

if(inbox.hasNext()==false) return;

while(inbox.hasNext()){

Message msg=inbox.next();

if(msg instanceof CMessage){
/* Chaque message contient l'état d'un voisin.
 * On mets alors la table de hachage à jour
 * Puis on réévalue la couleur du noeud */
donnee tmp=new donnee(((CMessage) msg).couleur);
etatvoisin.put(new Integer(((CMessage) msg).id),tmp);
compute();
}
}
}

public void preStep() {}

/* La fonction ci-dessous est appelée au démarrage uniquement
 * On initialise la couleur du noeud au hasard
 * On charge le premier timer. On crée la table de hachage */
public void init() {
initCouleur(nb);
(new CTimer(this,50)).startRelative(50,this);
this.etatvoisin=new Hashtable<Integer,
donnee>(this.outgoingConnections.size());
}

public void neighborhoodChange() {}
```

```
public void postStep() {}

public String toString() {
String s = "Node(" + this.ID + ") [";
Iterator<Edge> edgeIter = this.outgoingConnections.iterator();
while(edgeIter.hasNext()){
Edge e = edgeIter.next();
Node n = e.endNode;
s+=n.ID+" ";
}
return s + "]";
}

public void checkRequirements() throws WrongConfigurationException {}

/* La fonction ci-dessous affiche le noeud */
public void draw(Graphics g, PositionTransformation pt, boolean highlight) {
Color c;
this.setColor(this.RGBCouleur());
String text = ""+this.ID;
c=Color.BLACK;
super.drawNodeAsDiskWithText(g, pt, highlight, text, 20, c);
}
}
```

La dernière étape consiste à définir le modèle en nous basant sur les modèles existants. Pour cela, il suffit de personnaliser le fichier *Config.xml* situé dans le répertoire *coloriage*. Ouvrez le fichier *Config.xml* et modifiez les paramètres suivants :

- Interdisez la mobilité et les interférences.
- Changez le type de noeuds par défaut : remplacez ”DummyNode” par ”coloriage :CNode”.
- Changez le type d'arête en arête bidirectionnelle (*sinalgo.nodes.edges.BidirectionalEdge*).
- Initialisez les connections dès le démarrage de la simulation.
- Changez le mode de transmission de synchrone à asynchrone. Pour cela, modifiez la balise de mode de transmission de *ConstantTime* à *RandomTime*. Puis, remplacez la ligne <MessageTransmission ConstantTime=”1”/> par <RandomMessageTransmission distribution=”Uniform” min=”1” max=”5”/>. Dans cette balise, *min* et *max* définissent la borne inférieure et supérieure du temps de transmission des messages.
- Pour augmenter la connectivité du réseau, changez le paramètre *rMax* de *GeometricNodeCollection* et *UDG* en passant de 100 à 150. Dans *GeometricNodeCollection*, *rMax* correspond à la distance de transmission maximum des messages. Dans *UDG*, *rMax* définit la distance maximum où se trouve les voisins.

Lancez une exécution avec 100 noeuds.

5 Protocole de clustering

En vous basant sur le protocole de coloriage précédent, vous allez maintenant écrire un protocole de clustering. Dans ce protocole, chaque noeud va choisir un chef de cluster. Chaque cluster sera identifié par l'identité de son chef de cluster. Chaque noeud p va se baser sur deux informations pour choisir son chef de cluster : sa densité d_p et sa couleur c_p. La densité d_p du noeud p est égale au nombre d'arêtes liant p ou l'un de ses voisins à un autre voisin de p divisé par le nombre de voisin de p. En utilisant d_p et c_p, on définit la relation d'ordre suivante : $p \prec q$ *si et seulement si* $(d_p < d_q) \vee (d_p = d_q \wedge c_q < c_p)$. La fonction de choix du chef est alors la suivante :

$$\texttt{ClusterHead}(p) = \begin{cases} p & \text{si } \forall q \in N_p, q \prec p \\ max_{\prec}\{q \in N_p\} & \text{sinon} \end{cases}$$

avec N_p, l'ensemble des voisins de p.