Simulation d'algorithmes distribués avec **Sinalgo**

Stéphane Devismes (adapté pour les RICM4 par K. Altisen)

Résumé

Sinalgo est une plateforme de simulation qui permet de tester et de valider ≪ à haut niveau ≫ des algorithmes distribués. Cette platefome est écrite en JAVA. Elle est initialement dédiée aux protocoles pour réseaux de capteurs sans fil, mais nous l'utiliserons pour implémenter les protocoles du cours.

1 Installation

1.1 Pré-requis

Sinalgo nécessite que **Java 5.0** ou une version supérieure soit installé sur votre machine. Pour développer dans **Sinalgo** nous vous conseillons d'utiliser un environnement de développement intégré tel que **Eclipse**.

1.2 Installation

Les ressources à télécharger se trouvent sur le site du cours :

`http://www-verimag.imag.fr/~altisen/APD/`

1) Téléchargez le fichier *sinaligo-XXX.zip*. Décompressez le dans un sous-répertoire *sinalgo* de votre répertoire de travail. Ce répertoire contient notamment *les sources* dans *src*.
2) Téléchargez le fichier *FifoRandom.java* et copiez le dans le répertoire *sinalgo/src/projects/defaultProject/models/messageTransmissionModels/*.
3) Téléchargez le fichier *ConnectByHand.java* et copiez le dans le répertoire *sinalgo/src/projects/defaultProject/models/connectivityModels/*.
3) Téléchargez le fichier *walker.zip* et décompressez le dans le répertoire *sinalgo/src/projects/*. Vous aurez alors dans *sinalgo/src/projects/* un nouveau répertoire appelé *walker*.

Deux choix s'offrent alors à vous : soit travailler en ligne de commande, soit utiliser un environnement de développement comme **Eclipse**.

1.3 En ligne de commande

Commencer par recompiler tout le logiciel en tapant *ant*. La compilation devrait se terminer par le message *BUILD SUCCESSFUL*. Puis lancer le script *sinalgo* ou *sinalgo.bat* selon votre système d'exploitation.

1.4 Avec Eclipse

1. Lancez **Eclipse**.
2. Créez un nouveau projet java (File→New→Java Project). Nommez-le *sinalgo*. Décochez la case ≪ use default location ≫ et sélectionnez le répertoire *sinalgo* créé précédemment. Cliquez sur *finish*.
3. Commencez par reconstituer les *.class* : faites un clic droit sur *build.xml* dans le navigateur de package et sélectionnez ≪ *Run as→Ant Build* ≫.
4. Maintenant, on peut lancer l'exécution : il suffit de faire un clic droit sur *src* dans le navigateur de package et de sélectionner ≪ *Run as→Java Application* ≫ pour démarrer l'application.

2 Organisation

Le coeur du logiciel se trouve dans *src/sinalgo* ; vous n'aurez *a priori* pas à toucher à cette partie du code. **Sinalgo** propose ensuite de développer des protocoles dans le répertoire *src/projects*. Ce répertoire contient divers exemples fournis par la distribution standard ainsi que l'exemple de protocole *walker* que vous venez d'ajouter. Il contient aussi un projet par défaut (*defaultProject*) et un projet « vide » *template*. L'organisation de chaque projet est identique et comporte :

- Un fichier *Config.xml* qui permet de paramétrer le modèle d'exécution.
- Un fichier *description.txt* ; l'ensemble des informations de ce fichier est affiché dans le menu où l'on sélectionne la simulation à exécuter.
- Un fichier *CustumGlobal.java* qui permet de modifier l'interface de simulation.
- Deux répertoires : *models* et *nodes* qui contiennent les implémentations des modèles et des noeuds (*cf.* ci-dessous).

Les fichiers et répertoires importants sont détaillés ci-dessous. Pour plus d'informations, voir `www.disco.ethz.ch/projects/sinalgo/`.

2.1 Noeuds

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

- Le répertoire *messages* contient la description de chaque type de message utilisé dans la simulation.
- Le répertoire *nodeImplementations* contient le code pour le comportement des noeuds.
- Le répertoire *timers* contient le code des *timers* utilisés dans la simulation.

2.2 Modèles

Les modèles décrivent l'environnement dans lequel votre protocole est simulé. Ils sont définis dans *sinalgo/src/projects/defaultProject/models*. Vous pouvez implanter vos propres modèles ou utiliser des modèles prédéfinis. Ici, il ne vous sera pas demandé de développer vos propres modèles. Pour chaque simulation, il faut choisir 6 types de modèles :

- Le *modèle de connectivité* qui décide comment évaluer le voisinage d'un noeud. Nous utiliserons uniquement le modèle *ConnectByHand* qui permet de programmer des réseaux avec des formes particulières. Des exemples d'utilisation (réseaux en grille, en étoile, en arbre et en anneaux) sont fournis dans le fichier *sinalgo/src/projects/walker/CustomGlobal.java*.
- Le *modèle de distribution* qui décrit comment les noeuds sont placés au démarrage de l'application quand on utilise un modèle de connectivité automatique. Nous n'utiliserons pas cela dans ce TP.
- Le *modèle d'interférence* qui décide si un message est perdu suite à une interférence. Nous paramètrerons toujours le modèle pour qu'il n'y ait pas d'interférence.
- Le *modèle de transmission* qui décide du temps d'acheminement des messages.
  - *ConstantTime* permet d'exécuter le protocole en mode synchrone : chaque message est délivré en temps constant.
  - *RandomTime* permet d'exécuter le protocole en mode asynchrone, mais l'ordre FIFO n'est pas respecté.
  - Nous avons développé un modèle de transmission *FIFO* et *asynchrone*, appelé *FifoRandom* que nous utiliserons dans le TP.
- Le *modèle de mobilité* qui gère le mouvement des noeuds. Nous paramétrerons toujours le modèle pour que les nœuds restent immobiles.
- Le *modèle de fiabilité* qui gère la perte de messages.
  - *ReliableDelivery* assure que les canaux sont fiables.
  - *LossyDelivery* permet d'obtenir des canaux avec un taux de perte défini.

L'utilisation de ces modèles est paramétrée à l'aide du fichier *Config.xml*, que nous décrivons ci-après.

2.3 Config.xml

Nous détaillons maintenant les principales balises d'un fichier *Config.xml*.

- `asynchronousMode` : définit que les nœuds s'exécutent de façon asynchone.
- `mobility` : définit que les noeuds sont mobiles ou pas.

- `interference` : définit comment les communications sont sujettes aux interférences.
- `edgeType` : définit le type de lien de communications.
- `DefaultMessageTransmissionModel` : définit le modèle de transmission (synchrone/asynchrone, FIFO ou pas). Plusieurs valeurs sont possible :
  - `"ConstantTime"` (communication synchrone). Si vous l'utilisez, alors il faut redéfinir la balise `MessageTransmission` en `<MessageTransmission ConstantTime="1"/>`. Dans ce cas, les messages sont tous reçus après une unité de temps.
  - `"RandomTime"` (communication asynchrone mais pas FIFO). Si vous l'utilisez, alors il faut redéfinir la balise `MessageTransmission` en `<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.
  - `"FifoRandom"` (communication asynchrone et FIFO). Si vous l'utilisez, alors il faut remplacer la balise `MessageTransmission` par `<FifoRandom distribution="Exponential" lambda="1" />`.
- `DefaultConnectivityModel` : définit la relation de voisinage entre les nœuds.
- `DefaultMobilityModel` : définit le modèle de mobilité si on a choisi des nœuds mobiles. Ici sa valeur devra être `"NoMobility"`.
- `DefaultReliabilityModel` : définit la fiabilité des liens.

3 Le projet *Walker*

Lancez une exécution du projet *walker*. Vous pouvez le lancer *via* **Eclipse** ou en ligne de commande en utilisant un des deux scripts de lancement. Pour éviter de passer par la fenêtre de choix de projet, vous pouvez utiliser la commande : *sinalgo[.bat] -project walker*
Pour lancer la simulation, cliquez ensuite sur *Build XX Network* où *XX* est la forme du réseau que vous aurez choisi. Cliquez sur l'icone verte pour faire avancer la simulation pas par pas.
Pour comprendre le protocole que vous simulez, n'oubliez pas de lire le fichier *description.txt* (qui s'affiche aussi dans l'invite, avant le lancement de la simulation).

Voici maintenant comment est programmé le projet et comment vous devrez procéder pour créer nouveau projet : il faut créer de nouveaux types de noeuds, messages et timers. Ils se trouvent dans le répertoire de projet *walker*.
- A la racine du répertoire, il y a les fichiers *Config.xml*, *description.txt* et *CustomGlobal.java*.
- Dans le répertoire *nodes/messages*, le fichier *WalkerMessage.java* décrit le (seul type de) message utilisé dans le projet.
- Dans le répertoire *nodes/nodeImplementations*, les fichiers *InitNode.java* décrit un noeud initiateur (noeud vert dans la simulation), *EndNode.java* décrit le noeud qui arrête le message (noeud bleu) et *WalkerNode.java* décrit les autres noeuds (jaunes).
- Dans le répertoire *nodes/timers*, le fichier *InitTimer.java* programme le *timer* utilisé par le noeud initiateur pour déterminer quand il démarre son exécution.

Lancez plusieurs fois l'exécution du protocole sur les topologies fournies puis étudiez le code.

4 Programmer d'autres projets

Pour créer un autre projet, copiez-collez le répertoire *template* ou plus simplement si vous voulez aussi une base de code à modifier le répertoire *walker* et renommez-le. Vous pouvez ensuite compléter et modifier les fichiers de votre nouveau projet.

4.1 Inondation

Programmez un nouveau projet dont le protocole serait le suivant. Comme pour le *walker*, il y a 3 sortes de noeuds : un noeud initiateur *InitNode*, un noeud final *EndNode* et des noeuds intermédiaires *FloodingNode*.

Le noeud initiateur envoie un message de type *Flood* à tous ses voisins. Un noeud intermédiaire qui reçoit un message de type *Flood* pour la première fois change de couleur et renvoie le message à tous ses voisins. Un noeud intermédiaire qui aurait déjà reçu un tel message et qui en reçoit de nouveau ne fait rien. Le noeud final quand il reçoit un message de type *Flood* change simplement de couleur.

1. Ecrivez d'abord le protocole dans le formalisme du cours le protocole ci-dessus (vous pourrez l'écrire dans le fichier *description.txt*).
2. Programmez ce protocole et simulez-le.

4.2 Election sur anneau

De façon similaire programmez et testez l'algorithme d'élection sur anneau de Le Lann, vu en cours. Cette étape a aussi pour but de vous faire développer les fonctions qui vous seront nécessaires pour votre devoir à la maison.

Rappel de l'algorithme :

Algorithme 1 Élection de Le Lann pour un processus p

Variables

$Id_p \in \mathbb{Z}$ (identifiant unique du processus)
$Leader \in \{Vrai, Faux\}$ (représente le résultat de l'élection : le processus est-il le leader ?)
$List \in \mathbb{Z}^*$ (liste des identifiants collectés)

Algorithme pour un processus initiateur

$List \leftarrow \{Id_p\}$
`Envoyer` $\langle Election, Id_p \rangle$ `à` D
Pour toujours
    `Réceptionner` $\langle Election, Id \rangle$ `de` G
    Si $Id_p \neq Id$ **alors**
        $List \leftarrow List \cup \{Id\}$
        `Envoyer` $\langle Election, Id \rangle$ `à` D
    Sinon
        $Leader \leftarrow Id_p = \min(List)$
    Fin Si
Fin Pour toujours

Algorithme pour un processus non-initiateur

$Leader \leftarrow Faux$
Pour toujours
    `Réceptionner` $\langle Election, Id \rangle$ `de` G
    `Envoyer` $\langle Election, Id \rangle$ `à` D
Fin Pour toujours

Amélioration de l'algorithme précédent :

On cherche maintenant à améliorer la solution précédente. On part de deux observations simples.

1) On peut faire le calcul du minimum au fur et à mesure. Il n'est pas nécessaire de mémoriser une liste car, à chaque fois qu'un processus voit passer un message, il peut mettre à jour sa valeur du minimum. Stocker une seule valeur entière est alors nécessaire pour calculer, au lieu de n, dans la solution précédente.

2) On peut aussi réduire le nombre de messages échangés. Quand un message arrive avec un identifiant plus grand que l'identifiant du nœud, il ne porte certainement pas l'identité du leader et on peut le supprimer. Dans ce cas, le seul message à réussir à faire le tour est le message qui porte l'identité du leader. Autrement dit, un nœud qui reçoit au bout d'un tour le message qu'il avait envoyé peut se considérer comme leader !

En utilisant cette base d'observations proposez un algorithme qui améliore celui de Le Lann (il devra répondre à la même spécification).

1. Ecrivez votre algorithme en utilisant la syntaxe du cours.
2. Puis programmez le, pour le tester.
3. Quelle est la complexité en mémoire et en temps de votre algorithme ?
4. Que pensez vous de l'hypothèse sur l'ordre FIFO des canaux ?