Notre implémentation du simulateur

MERLET Benoît
ZUREK Pierre

16 août 2005

Résumé

Notre participation au projet RARE avait essentiellement pour but d'aider Jérôme dans la tâche de programmation d'un simulateur de réseau *peer-to-peer* dont chaque noeud serait capable de rechercher des documents sur le réseau. Le but final étant d'intégrer les optimisations résultantes de la réflexion des chercheurs sur le projet pour pouvoir les évaluer quantitativement.

Il a été décidé de se baser sur un simulateur existant : PeerSim. Le problème était qu'il ne répondait pas au cahier des charges fixé en ce sens qu'il se cantonne plus ou moins à la gestion du réseau, mais pas des différents protocoles implémentables.

Voici donc la documentation de notre implémentation.

Table des matières

1 Le simulateur peersim

1.1 Fonctionnement général

Le simulateur peersim est un ensemble de classes java qui implémentent un simulateur de réseau pair à pair. Plusieurs types de simulations sont possibles, mais celle qui nous intéresse ici est la simulation par cycles : nous allons laisser évoluer le réseau pendant un nombre prédéfini de cycles. Il nous suffira d'analyser l'état du réseau à la fin de la simulation pour en tirer les bonnes conclusions.

De façon à simuler et observer tout type de réseau P2P, tout est paramétrable : les développeurs de PeerSim mettent ainsi à notre disposition trois types de composants :

des protocoles : ils définissent un comportement de noeud.

des dynamics : ils permettent de définir le réseau.

des observers : ils permettent d'observer la simulation.

Toutes les données utiles au simulateur sont regroupées dans un fichier de configuration où l'on définit les variables du réseau, déclare la structure d'un noeud (les protocoles qui vont s'y exécuter à chaque cycle), les protocoles, les dynamics et les observers que l'on veut utiliser.

1.2 Les protocoles

Comme vu plus haut, un protocole est un comportement de noeud. En effet, dans un réseau P2P que l'on veut simuler avec PeerSim, chaque noeud dispose d'une pile de protocoles qui vont être exécutés à chaque cycle de simulation, et ceci dans leur ordre de déclaration dans le fichier de configuration. Bien évidemment, tout est mis à notre disposition pour créer nos propres protocoles.

Chaque protocole est décrit dans la classe java qui porte son nom. Pour les simulations par cycles, celles qui nous intéressent, cette classe doit implémenter l'interface `CDProtocol` (pour Cycle Driven Protocol) fournie dans PeerSim. Elle doit donc redéfinir les méthodes `clone()` et `nextCycle(Node, int)`.

Lors du lancement du simulateur, PeerSim crée un squelette de noeud avec une instance de tous les protocoles déclarés dans le fichier de configuration, puis clone ce squelette autant de fois que le nombre de noeuds dans le réseau. C'est à cela que sert cette première fonction `clone()`. Il faut à ce propos faire attention à bien réinitialiser les attributs des protocoles dans cette fonction, sinon, lors du clonage, les références des attributs vont être clonées et tous les protocoles auront, dans les faits, les mêmes attributs.

Quant à la fonction `nextCycle(Node, int)`, c'est elle qui définit le comportement du protocole que l'on souhaite implémenter. Elle prend en paramètre le noeud sur lequel se trouve le protocole, et un entier identifiant le numéro du protocole dans la pile de protocole du noeud squelette. C'est cette fonction qui sera appelée à chaque cycle et qui simulera l'exécution de ce protocole sur le noeud courant.

1.3 Les dynamics

Leur but est d'introduire un dynamisme dans le réseau, essentiellement sous deux formes : d'une part initialiser le réseau, c'est à dire initialiser les protocoles présents sur les noeuds du réseau, et d'autre part introduire des biais en cours de simulation, par exemple des suppressions de liens entre des noeuds, ou des suppressions de noeuds.

Tout comme pour les protocoles, nous pouvons créer nos propres dynamics. Il suffit de créer une classe Java implémentant l'interface `Dynamics` fournie dans PeerSim, et qui doit donc redéfinir la méthode `modify()`. C'est cette méthode qui sera appelée par le simulateur lors de l'exécution d'un dynamics qui est censé *modifier* l'état du réseau.

Le fichier de configuration peut contenir deux types de fichiers de dynamics *(cf. infra)* :
- le type initializer sera exécuté une unique fois avant le début de la simulation
- le type standard sera exécuté au début de chaque cycle de simulation

1.4 Les observers

Comme leur nom l'indique, ils ont pour but de surveiller le réseau. Ils sont exécutés au début de chaque cycle de simulation et peuvent être utilisés pour de multiples traitements : affichage brut de l'état d'un noeud, collection de données pour un affichage résumé ultérieur, calcul d'indicateurs sur l'état du réseau, statistiques sur ces indicateurs, ...

De la même manière que les deux autres composants, PeerSim nous facilite le rajout d'un observer : c'est toujours une classe Java, mais qui implémente cette fois-ci l'interface `Observer`. Nous devons donc redéfinir la méthode `analyze()` qui sera appelée par le simulateur lors de l'exécution de notre observer.

1.5 Un exemple de fichier de configuration

De façon à synthétiser les paragraphes précédents, détaillons l'explication d'un exemple du fichier de configuration de PeerSim. Ce fichier est une configuration d'un des exemples distribués avec PeerSim : chaque noeud possède

une valeur et la simulation consiste, pour chaque noeud et à chaque cycle, à faire la moyenne de la valeur du noeud avec celles de ses voisins. Si nous lançons cette simulation sur un nombre suffisant de cycles, nous voyons que la valeur de tous les noeuds s'approche de la moyenne des valeurs initiales.

```
# PEERSIM EXAMPLE 1
simulation.cycles 30
simulation.shuffle

overlay.size 50000
overlay.maxsize 100000
```

Dans un premier temps, on renseigne le simulateur du type de simulation que nous voulons faire : un réseau initial de 50000 noeuds (100000 au maximum) sur 30 cycles de simulation sans suivre l'ordre des noeuds (`shuffle`).

```
protocol.0 peersim.core.IdleProtocol
protocol.0.degree 20

protocol.1 example.aggregation.AverageFunction
protocol.1.linkable 0
```

On déclare ensuite la pile de protocoles présente sur un noeud : ici, le protocole 0 qui se charge de gérer une liste de 20 voisins (`degree`), et le protocole 1 qui se charge de calculer la moyenne des valeurs sur un ensemble de voisins qu'il récupère grâce au protocole 0 (`linkable`).

```
init.0 peersim.dynamics.WireRegularRandom
init.0.protocol 0
init.0.degree 20

init.1 example.loadbalance.LinearDistributionInitializer
init.1.protocol 1
init.1.max 100
init.1.min 1
```

Il est temps de passer à la déclaration des dynamics, qui sont ici des initializers (`init.*`). Le premier crée une topologie aléatoire en agissant sur le protocole 0 (chaque noeud a donc 20 voisins aléatoires), alors que le second distribue linéairement entre `min` et `max` les valeurs initiales des noeuds du réseau (il agit sur le protocole 1 de chacun des noeuds).

```
observer.0 example.aggregation.AverageObserver
observer.0.protocol 1
```

Enfin, on déclare un observer qui affiche, à chaque cycle, des statistiques sur les différentes valeurs des noeuds. Cela permet de voir l'évolution globale des valeurs. Remarquons qu'on lui passe en paramètre le protocole 1 de façon à ce qu'il sache où aller chercher l'information.

2 Notre implémentation

2.1 Principes généraux

Le but initial de notre participation dans ce projet était de fournir un simulateur capable de réaliser des simulations d'un réseau de pairs implémentant un protocole P2P doté d'un système de recherche de documents sur le réseau. Le choix fut vite fait étant données les documentations trouvées sur Internet : le protocole Gnutella. Notre implémentation est d'ailleurs très largement basée sur la spécification de ce protocole.

Le principe général du système de requêtage de Gnutella est un parcours en largeur du graphe du réseau de pairs. Typiquement, lorsqu'un pair initie une requête, il l'envoie à 7 de ses 20 voisins qui vont traiter la requête en local, envoyer une réponse au noeud parent (celui qui vient de transférer la requête) s'il y a effectivement des documents pertinents, puis transférer la requête à 7 autres voisins (tous différents du noeud parent).

Les concepteurs du protocole ont tout de même apporté certaines optimisations par rapport au simple parcours en largeur :

- Comme indiqué ci-dessus, seuls 7 voisins sur 20 sont visités à chaque itération du parcours.
- Pour ne pas être obligé de garder dans la requête le noeud initiateur, une réponse va suivre le chemin inverse de la requête associée.
- Un système de *Time To Live* existe de manière à ne pas augmenter artificiellement le temps global de réponse pour le noeud initiateur.

Une de nos principales préocupations a été de rendre le code le plus modulaire possible, ceci nous ayant été grandement facilité par l'utilisation de PeerSim. Ainsi chaque traitement important est encapsulé dans un protocole que nous rajoutons à la pile de protocoles du noeud squelette (grâce au fichier de configuration). Cette méthode permet une plus grande réutilisabilité du code, ainsi qu'une maintenance plus aisée.

Nous utilisons par ailleurs 2 classes ne contenant que des attributs et méthodes statiques : c'est une manière de recourir à des variables globales. La première se nomme `Constants` et contient les constantes de la simulation courante : le *verbose* des sorties du programme, les différents identifiants des protocoles sur les noeuds, la liste complète des documents du réseau, la pertinence de ces documents en tant que réponse à chaque requête et des accesseurs sur ces attributs. La seconde se nomme `Variables` et ne contient pas grand chose, seulement pour l'instant le nombre de messages envoyés sur le réseau, grandeur variable sur la durée de simulation.

2.2 Le fichier de configuration

Voici ce fichier :

```
simulation.cycles 600
overlay.size 134

# Protocols
protocol.0 peersim.core.IdleProtocol
protocol.0.capacity 30
protocol.1 myGnutella.handlers.OverlayHandler
protocol.2 myGnutella.handlers.QueryHandler
protocol.3 myGnutella.handlers.QueryHitHandler
protocol.3.distanceLimit 0.5
protocol.3.maxDocumentsToDownload 2
protocol.4 myGnutella.handlers.LogHandler
protocol.5 myGnutella.Gnutella
protocol.5.degree 7

# Initializers
init.0 myGnutella.dynamics.ProtocolsInitializer
init.0.protocol 4
init.0.verbose 0
init.0.linkableProtocolID 0
init.0.overlayHandlerProtocolID 1
init.0.queryHandlerProtocolID 2
init.0.queryHitHandlerProtocolID 3
init.0.gnutellaProtocolID 4
init.0.logHandlerProtocolID 5
init.1 peersim.dynamics.WireRegularRandom
init.1.protocol 0
init.1.degree 20
init.1.pack
init.2 myGnutella.dynamics.DocumentsInitializer
init.2.protocol 4
init.2.fileDocumentsDefinition config/document_definition.xml
init.2.fileDocumentsDistribution config/document_distribution.xml
init.3 myGnutella.dynamics.QueriesInitializer
init.3.protocol 2
init.3.fileQueriesDefinition config/query_definition.xml
init.3.fileQueriesDistribution config/query_distribution.xml

# Observers
observer.0 myGnutella.observers.MessagesObserver
observer.0.name MessagesMeterObserver
observer.1 myGnutella.observers.LogObserver
observer.1.name LogGeneratorObserver
observer.1.file config/simulateur.log.xml
```

Ce fichier de configuration est celui du dernier test que nous avons fait sur le simulateur : une simulation de 600 cycles et sur un réseau de 134 pairs. Le réseau contient une ensemble de 17856 documents répartis sur les noeuds qui vont initier quelques 683 requêtes. Voyons plus en détail chaque partie.

Chaque noeud comporte un pile de 6 protocoles :

IdleProtocol : c'est un protocole fourni avec PeerSim et dont le but est de gérer une liste de `capacity` voisins.

OverlayHandler : ce protocole génère, étant donné le voisinage d'un noeud et une requête, une liste de voisins à qui transférer cette requête.

QueryHandler : c'est le protocole qui gère le système de requêtage du pair sur lequel il se trouve.

QueryHitHandler : celui-là s'occupe de la réception des réponses et de la prise de décision du téléchargement d'un document grâce à ses deux paramètres `distanceLimit` et `maxDocumentsToDownload`.

LogHandler : toutes les actions importantes sont notifiées à ce protocole qui se chargera en temps voulu de générer un log XML correctement formaté.

Gnutella : c'est le protocole principal qui s'occupe d'interfacer les modules précédents. Il comporte un paramètre `degree` qui est le nombre de voisins à qui le pair va devoir transférer ses requêtes.

On déclare ensuite 4 initializers :

ProtocolsInitializer : il va se charger d'associer les 5 premiers protocoles d'un noeud donné au protocole `Gnutella` de ce même noeud, mais aussi d'initialiser les constantes de la simulation comme les identifiants des protocoles.

WireRegularRandom : cet initializer est fourni avec PeerSim et se charge de construire une topologie aléatoire où chaque noeud possède `degree` voisins.

DocumentsInitializer : il a pour but de distribuer les documents contenu dans le fichier `fileDocumentsDefinition` sur les pairs grâce au fichier `fileDocumentsDistribution`.

QueryHitHandler : celui-là fait de même, mais sur les requêtes que l'on veut initier sur le réseau.

Enfin, on termine le fichier de configuration par la déclaration des observers. Le premier est plus un test qu'autre chose : il sert à compter le nombre de messages (requête ou réponse sans distinction) envoyés sur le réseau. Le second est plus utile : il teste si le cycle courant est le dernier et si c'est le cas demande au protocole `LogHandler` de génerer le fichier de log de la simulation au format XML.

2.3 Les différents composants

2.3.1 La classe Gnutella

Cette classe est le protocole principal de notre implémentation de Gnutella sur le simulateur PeerSim. Elle s'occupe essentiellement d'interfacer tous les modules de gestion des différents objets du simulateur.

La classe `Gnutella` étend une autre classe nommée `DocumentHolder` qui maintient une liste d'identifiants de documents que possède un pair. C'est éventuellement dans cette classe qu'il faudrait rajouter des systèmes de suppression de documents sur un pair suivant différents comportements (une classe fille par comportement par exemple).

La meilleure méthode pour intégrer les modules est de rajouter un attribut membre de la classe générique pour chaque module. C'est ce qui est fait dans la classe `Gnutella`. Ainsi, à tout moment, le protocole principal connait l'état de ses attributs (donc de ses protocoles de niveau inférieur) et peut utiliser leurs méthodes respectives.

Voici enfin la méthode `nextCycle(Node, int)` du protocole `Gnutella` qui se contente, à chaque cycle de simulation, de faire appel à ses protocoles de niveau inférieur pour traiter les requêtes et les réponses en attente.

```
/**
* defines the behavior of the Gnutella protocol.
* This method need to be implemented because of Protocol abstract class.
* @param node the node where this prtotocol is running.
* @param protocolID the ID of the Gnutella protocol on this node.
*/
public void nextCycle( Node node, int protocolID )
{
    // on récupère le cycle courant
    int currentCycle = CommonState.getCycle();
    // on affiche un petit message
    if( Constants.verbose > 0 )
        System.out.println("\trunning cycle "+currentCycle+" on node "+node.getIndex());
    // on traite et transfère les queries
    this.queryHandler.processAndForwardQueries(this.documentsIdsList,
        currentCycle,
        node,
        this.degree);
    // on transfère les query hits
    this.queryHitHandler.forwardQueriesHits(currentCycle, node);
}
```

2.3.2 les différents gestionnaires

Les gestionnaires sont ce qui est nommé plus haut des protocoles de niveau inférieur. Ce sont aussi des protocoles qui apparaissent dans la pile de protocole d'un noeud, à la différence près que leurs méthodes `nextCycle()` est vide. Ils sont dans le package `handlers` du code.

OverlayHandler

Dans le protocole Gnutella, une requête arrive d'un noeud P sur le noeud courant qui tente d'y répondre puis se charge de la transférer à son voisinage. Le protocole définit clairement ce voisinage : ce sont 7 voisins qui sont choisis au hasard parmi la totalité des voisins privée du noeud P. Le gestionnaire nommé `OverlayHandler` s'occupe de générer cette liste de voisins en utilisant le protocole `IdleProtocol` du noeud (attribut `linkable` de la classe).

QueryHandler

Ce protocole gère l'expédition et la réception des requêtes. Pour cela, la classe admet des attributs : `alreadyProcessedQueries`, une liste de couples (identifiant de requête, noeud parent) qui permet de ne pas traiter une requête qui l'a déjà été sur le noeud, `queriesToForward`, un tableau de FIFO pour chaque cycle de simulation qui permet de savoir quelles requêtes doivent être traitées à chaque cycle et `nextCycleToForward`, un entier qui permet de simuler un temps de traitement d'un cycle par requête. Toutes les méthodes utiles pour pouvoir accéder à ces attributs sont implémentées et suffisament documentées dans le code. Le principe : chaque requête dans la liste associée au cycle courant va être traitée, puis transférée à 7 voisins génèrés par l'`OverlayHandler`.

QueryHitHandler

De la même façon, ce protocole gère l'expédition et la réception des réponses aux requêtes. Mais c'est aussi lui qui prend la décision de télécharger un document quand la réponse est arrivée à destination. L'attribut `queriesHitsToForward` est un tableau de FIFO pour chaque cycle, toujours pour savoir ce que l'on a à traiter au cycle courant : si cette liste est non vide, on récupère, par l'intermédiaire du `QueryHandler` du noeud, le pair à qui transférer la réponse. Si ce pair est `null`, nous sommes sur le père initiateur de la requête. Dans ce cas, les attributs `distanceLimit` et `maxDocumentsToDownload` servent pour la prise de décision de téléchargement : si dans une réponse qui est arrivée à destination se trouve un document dont la distance est inférieure à `distanceLimit`, on le télécharge, dans la limite de `maxDocumentsToDownload` par identifiant de requête. l'attribut `numberOfDocumentsToDownloadByQueryID` tiens une liste de nombre de documents encore à télécharger par identifiant de requête.

LogHandler

Ce protocole permet quant à lui la génération du fichier de sortie du simulateur : c'est un fichier XML qui résume tous les événements importants de la simulation. Le schéma de ce fichier a été discuté avec les Bretons. Pour chaque évènement important, le `LogHandler` du noeud courant est notifié, il se charge alors de construire 3 chaînes de caractères pour chaque identifiant de requête : une pour les informations générales concernant la requête, une pour les documents pertinents reçus dans des réponses à la requête, et enfin, une pour les documents effectivement téléchargés.

2.3.3 Les initializers

Avant de parler en détail des initializers, il faut au préalable décrire la librairie SAX incluse en standard dans la librairie Java et permettant de manipuler les fichiers XML.

La librairie SAX

Elle permet de parser un fichier XML afin d'en récupérer les différents champs et de les stocker dans des structures appropriées, mais aussi de vérifier la validité d'un fichier XML, c'est-à-dire s'il est bien formé (toutes les balises sont bien fermées) et s'il correspond à son schéma, le schéma d'un fichier XML étant un fichier XML (ayant pour extension *xsd*) décrivant de manière précise les différents champs du fichier XML associé, en indiquant leur nom, leur type, le nombre maximal d'occurences d'un champ, etc... (voir plus bas des exemples).

Afin que SAX parse correctement un type de fichier XML, il faut lui définir un comportement. Les classes décrivant le comportement de SAX en fonction d'un type de fichier sont les classes dont le nom commence par `SAXHandler` dans le package dynamics de myGnutella. Il y a donc autant de classes `SAXHandler*` que de fichiers XML à parser (en l'occurence 4).

Les différents fichiers XML

Nous utilisons 4 types de fichiers XML, chacun ayant son propre schéma associé. Nos fichiers XML débutent tous par la balise :

```
<?xml version="1.0" encoding="UTF-8"?>
```

qui indique que le fichier est bien un fichier XML et que l'encodage des caractères est de l'UTF-8. Ensuite tous les fichiers XML utilisés pour les initialiseurs comporteront les attributs :

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='fichier.xsd'
```

dans la balise racine qui permettent d'indiquer l'emplacement du fichier XSD présentant la structure du fichier XML et servant à en vérifier la validité.

Les 4 fichiers XML et leurs schémas sont :

- le fichier de définition des documents, exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<documents xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='DocumentsDefinition.xsd'>
<document document_id="1.txt">
<words>
<word>string000</word>
<word>string001</word>
<word>string002</word>
</words>
</document>
</documents>
```

- le fichier XSD associé :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by abs (fde) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="documents">
<xs:annotation>
<xs:documentation>la liste des documents choisis</xs:documentation>
</xs:annotation>
<xs:complexType>
```

```
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="document">
<xs:complexType>
<xs:sequence minOccurs="0">
<xs:element name="words">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="word" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="document_id" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

– le fichier de distribution des documents, exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<peers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='DocumentsDistribution.xsd'>
<peer peer_id="0">
<document document_id="1.txt"/>
</peer>
</peers>
```

– le fichier XSD associé :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by abs (fde) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="peers">
<xs:annotation>
<xs:documentation>Répartition des documents</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="peer">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="document">
<xs:complexType>
<xs:attribute name="document_id" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="peer_id" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

– le fichier de définition des requêtes, exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<queries xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='QueriesDefinition.xsd'>
<query query_id="1">
<words>
<word>stringone</word>
<word>stringtwo</word>
```

```
</words>
<responses>
<response distance="0.10000000000000000000" document_id="4.txt"/>
<response distance="0.5" document_id="2.txt"/>
</responses>
</query>
</queries>
```

– le fichier XSD associé :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by abs (fde) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="queries">
<xs:annotation>
<xs:documentation>la liste des documents choisis</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="query">
<xs:complexType>
<xs:sequence minOccurs="0">
<xs:element name="words">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="word" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="responses">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="response">
<xs:complexType>
<xs:attribute name="document_id" type="xs:string" use="required"/>
<xs:attribute name="distance" type="xs:float" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="query_id" type="xs:integer" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

– le fichier de distribution des requêtes, exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<peers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='QueriesDistribution.xsd'>
<peer peer_id="1">
<query query_id="1" cycle_number="1" ttl="20"/>
</peer>
</peers>
```

– le fichier XSD associé :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by abs (fde) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="peers">
```

```
<xs:annotation>
<xs:documentation>Répartition des requêtes</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="peer">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="query">
<xs:complexType>
<xs:attribute name="query_id" type="xs:integer" use="required"/>
<xs:attribute name="cycle_number" type="xs:integer" use="required"/>
<xs:attribute name="ttl" type="xs:integer" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="peer_id" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Décrivons maintenant les classes du package dynamics.

DocumentsInitializer

Cette classe permet d'initialiser les documents sur les noeuds. Le constructeur remplit les chemins des deux fichiers XML du fichier de configuration décrivant la définition (*fileDocumentsDefinition*) et la distribution (*fileDocumentsDistribution*) des documents sur les noeuds. Ensuite, on parse d'abord le fichier XML *fileDocumentsDefinition* afin de stocker son contenu sous la forme d'une liste de couples *(documentID, liste de mots)* dans la classe `Constants`. On parse alors *fileDocumentsDistribution* et l'on associe à chaque noeud du réseau la liste des IDs des documents qu'il possède (N.B. : on associe la liste des IDs plutôt que la liste des couples *(ID, liste de mots)* afin de ne pas surcharger les noeuds).

QueriesInitializer

Cette classe permet d'initialiser les requêtes sur les noeuds. De même que pour `DocumentsInitializer`, ceci s'effectue en deux étapes. La première étape consiste à récupérer les chemins des fichiers contenant les données sur la définition (*fileQueriesDefinition*) et la distribution (*fileQueriesDistribution*) des requêtes dans le fichier de configuration. La deuxième étape consiste alors à initier les requêtes sur les noeuds correspondants. Comme précédemment, on parse avec SAX le fichier XML de définition des requêtes en vérifiant sa validité puis on parse le fichier de distribution des requêtes sur les noeuds. A la fin de ce processus, tous les noeuds contiennent leurs requêtes et la liste statique *answersQualityList* de la classe `Constants` est remplie et contient tous les triplets de la forme *(queryId, DocumentId, distance)*.

ProtocolsInitializer
Cette classe initialise les attributs des classes Constants et Variables en fonction des valeurs du fichier de configuration et associe les divers protocoles d'un noeud au protocole Gnutella de ce même noeud.

2.3.4 Les observers

Notre implémentation n'en comporte que 2, mais il pourrait être intéressant de voir plus en détail ce que l'on peut faire. Ils se trouvent dans le package `observers` du code.

LogObserver
Cet observer n'est en fait actif qu'avant la simulation du dernier cycle. Un nom de fichier lui est passé par l'intermédiare du fichier de configuration : il est sauvé dans son attribut `file`. Juste avant la simulation du dernier cycle, cet observer récupère les fichiers journaux partiels sur tous les noeuds du réseau pour les concaténer et créer le fichier de sortie du simulateur.

MessageObserver
Cet observer compte le nombre de messages qui sont envoyés sur le réseau. À chaque fois que l'on simule l'envoi d'un message (requête ou réponse), une variable statique de la classe `Variables` est incrémentée. Cet Observer ne fait qu'afficher cette variable avant la simulation de chaque cycle.

2.4 L'interface graphique ou GUI (Graphical User Interface)

Elle a été créée afin de :
- pouvoir générer et éditer facilement des fichiers de configuration pour le simulateur
- lancer une simulation à partir d'un fichier de configuration nouveau ou donné

Elle utilise la librairie Swing incluse dans la librairie standard Java et qui permet de réaliser des interfaces graphiques.

Comment utiliser la GUI pour lancer une simulation ?

Au lancement de la GUI, on s'aperçoit que les listes situées à gauche de la fenêtre sont vides.

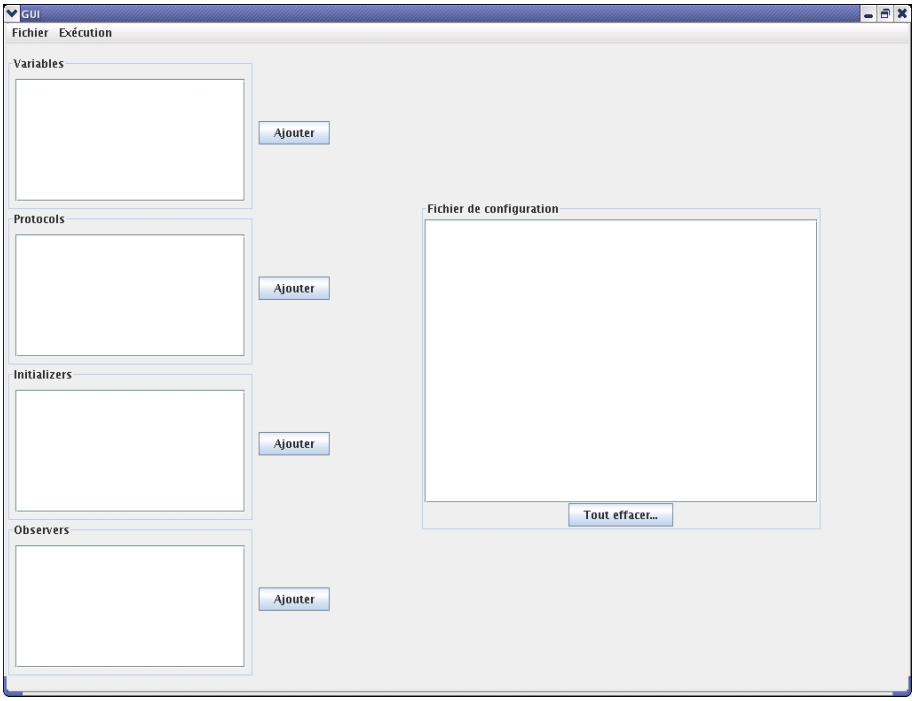

FIG. 1 – Fenêtre de la GUI à son lancement.

Il faut donc ouvrir un fichier XML précisant les contenus des différentes listes : Variables, Protocols, Initializers, Observers. La fenêtre voit alors ces différents champs se remplir si le fichier XML est valide.

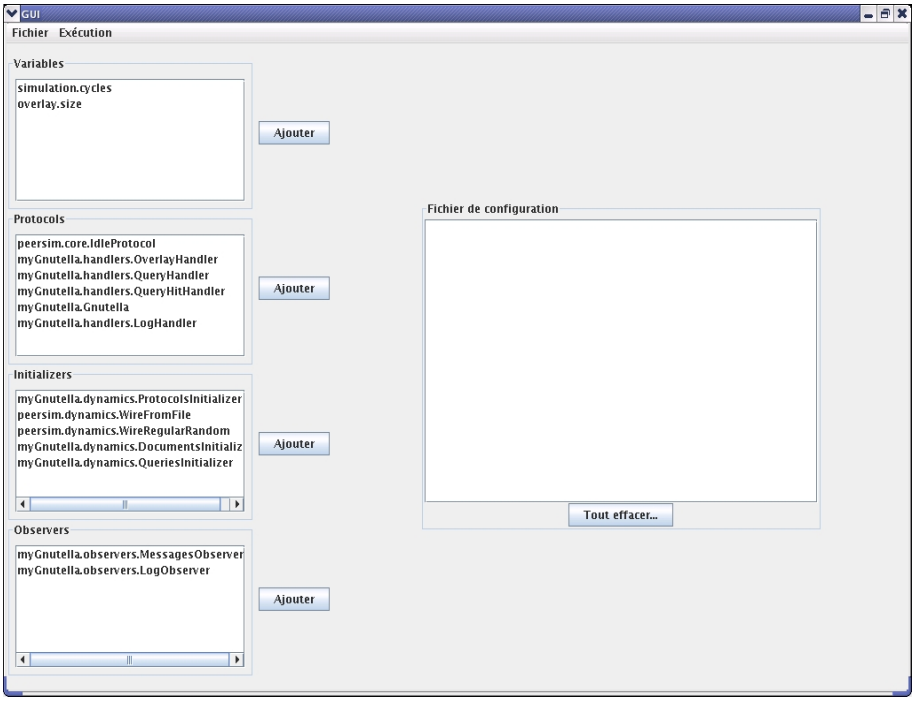

FIG. 2 – Fenêtre de la GUI après ouverture réussie du XML.

Il suffit alors de sélectionner les variables, les protocoles, les initialiseurs et les observers que l'on souhaite inclure dans le fichier de configuration nécessaire à peersim et on clique à chaque fois sur le bouton associé. Une fois le fichier de configuration écrit, il y a moyen de le sauvegarder via le menu Fichier. De même, il est possible d'ouvrir un fichier de configuration dont le contenu s'affichera dans la zone de texte. En cliquant sur le menu Exécuter, une boîte de dialogue apparaît dans laquelle on indique le chemin vers la machine virtuelle java, les fichiers où seront redirigés la sortie standard et la sortie erreur du simulateur, la classe principale du simulateur, le dossier courant du simulateur, les librairies utiles au lancement du simulateur. Une fois que l'on a cliqué sur OK, la fenêtre se gèle et attend la fin de l'exécution de la simulation. Une fois la simulation finie, une boîte de dialogue nous en avertit.

Présentation des classes de la GUI :

- `MainGUI` : c'est la classe principale. Elle lance une fenêtre de type `MyFrame` avec pour titre **GUI**.
- `MyButton` : c'est la classe utilisée pour les différents boutons **Ajouter**. Ces boutons réagissent au clic et permettent d'ajouter la ligne sélectionnée (si elle existe) dans la liste associée à la zone de texte de la fenêtre.
- `MyConstants` : ce sont les constantes utilisées par la GUI.
- `MyDialog` : c'est la fenêtre de dialoque qui s'ouvre au moment où l'on clique sur l'item de menu **Exécuter**. Actuellement, cette classe parse le fichier *.classpath* de type XML généré par Eclipse afin de récupérer des données telles que le chemin vers la librairie peersim. Ceci est à changer car il faut que le programme soit indépendant de la plateforme de développement utilisée (i.e. il faudrait créer un fichier XML *prefs.xml* qui contiennent toutes les données utiles à la bonne exécution de la simulation et qui soit spécifique à chaque utilisateur).

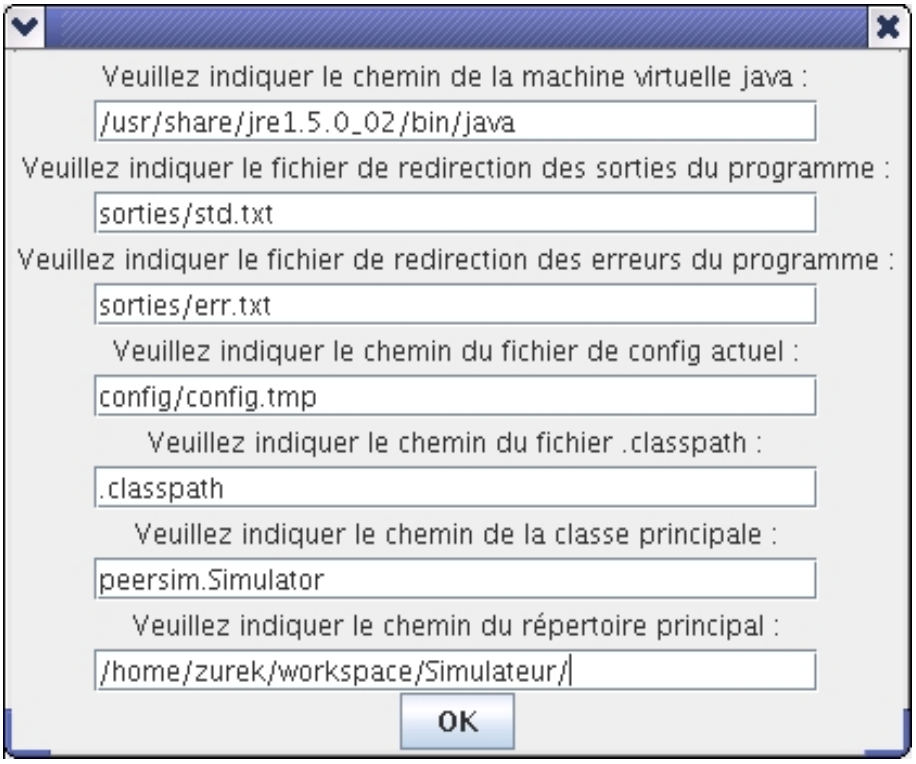

FIG. 3 – Boîte de dialogue affichée avant exécution.

- `MyFrame` : c'est la classe utilisée pour afficher la fenêtre principale de la GUI.
- `MyList` : c'est la classe utilisée pour permettre l'affichage et la sélection dans les listes de Variables, Protocols, Initializers et Observers.
- `MyTextArea` : c'est la classe de gestion de la zone de texte et du bouton **Tout effacer** situés à droite de la fenêtre principale.
- `SAXHandlerClasspathFile` : cette classe est, comme dit précedemment dans la description de la classe `MyDialog`, à supprimer et à remplacer par une classe qui parserait un fichier *prefs.xml* contenant toutes les données nécessaires au lancement de la simulation.
- `SAXHandlerXMLFile` : c'est cette classe qui permet de parser le fichier XML de configuration (généralement appelé *gui.xml*) et qui permet de remplir les champs des différentes listes à gauche dans la fenêtre principale.
- les classes du package types sont juste des classes permettant une meilleure gestion des associations entre variables (ce sont des n-uplets).

Présentation des fichiers nécessaires à la GUI :

- le fichier *gui.xml* : ce fichier XML définit les différents champs des élé-

ments Variables, Protocols, Initializers, Observers de la GUI. Il indique pour chaque champ ses attributs et leur type afin que l'utilisateur ne tape pas n'importe quoi. Il précise aussi les dépendances entre les différents éléments des différentes listes grâce au tag link.
- le fichier *.classpath* d'Eclipse : ce fichier contient certains chemins vers des librairies utiles au lancement du simulateur.
- le fichier de définition des documents (*fileDocumentsDefinition*), le fichier de distribution des documents (*fileDocumentsDistribution*), le fichier de définition des requêtes (*fileQueriesDefinition*), le fichier de distribution des requêtes (*fileQueriesDistribution*) : leurs structures sont détaillées dans la section Initializers de cette documentation.

2.5 Les limites du simulateur

Nous avons envisagé ce petit paragraphe pour parler des performances de notre simulateur. Le seul test de montée en charge que nous avons pu faire est celui de référence de cette documentation : 600 cycles, 133 pairs, 17856 documents, 683 requêtes. En stage, nous avions à notre disposition des stations de travail DELL munies d'Intel Pentium 4 à 2.80GHz et de 1Go de mémoire RAM, et l'exécution de cette simulation prenait environ 2 minutes et utilisait un peu moins de 70Mo de mémoire.

Nous avons pensé le code du simulateur de façon à optimiser le temps de calcul plutôt que l'utilisation de la mémoire. C'est pour cela que sur chaque noeud, dans le `QueryHandler` et le `QueryHitHandler` nous avons opté pour des structures de données bizarres (un tableau de FIFO). Le problème, c'est que la taille de ces tableaux est directement associée au nombre de cycles de simulation, donc il faudra faire attention à ne pas faire une simulation combinant à la fois beaucoup de noeuds et beaucoup de cycles.

2.6 Ce qu'il reste à faire

Voici en vrac une petite liste des fonctionnalités encore à implémenter dans notre version du simulateur, mais aussi des idées dont on avait parlé lors de la réunion RARE :

- Nos amis bretons nous ont demandé s'il était possible de relancer certaines requêtes après la fin d'une simulation (pour évaluer l'état d'apprentissage de leur réseau de neurones). Il serait donc bien de faire un système qui permette de sauver l'état final du simulateur, puis de pouvoir relancer le simulateur en partant de cet état.
- Toujours venant des bretons, il faudrait qu'un document puisse avoir une classe et que, d'une part les fichier du générateur soient générés en conséquence, et d'autre part prendre en compte cette classe dans le simulateur.

- Il faudrait également intégrer un système de feedback c'est à dire une notification du choix effectif de l'utilisateur parmi les propositions (revenues dans les réponses à la requête). Cela sera utile pour les deux points suivants.
- Lors de sa dernière semaine de stage de DEA, Jérôme a implémenté un système de gossiping pour partager les bloom filters entre voisins. Il faudra faire un nouveau protocole pour intégrer ses modifications.
- Pour tout avoir directement dans le simulateur, il faudra intégrer le réseau de neurones pour la mise à jour dynamique du voisinage d'un noeud. C'est dans le `OverlayHandler` qu'il faudra aller mettre son nez.
- Lors de la réunion RARE, nous avions dit qu'il serait souhaitable de prévoir un système de suppression de documents sur un noeud, ceci suivant différents comportements (je-récupère-tout, je-récupère-j'efface, ...). Pour cela, il faudrait faire une classe fille de la classe `DocumentHolder` par comportement, et rajouter un fichier de définition des documents pour choisir le comportement associé à un pair.
- Jérôme nous a parlé d'un système inclus dans le simulateur peersim pour faciliter la génération de statistiques. Il pourrait être bien de voir ce que l'on peut faire avec pour l'appliquer à la génération de statistiques sur le nombre de messages.
- Enfin, pour rendre le code du simulateur le plus réutilisable possible, il faudrait le réorganiser sous forme d'une API en interfaçant chaque type d'objet que nous avons rajouté par rapport à PeerSim (handlers, parsers XML, ...).