TME ARA
PeerSim : Prise en main

Jonathan Lejeune

Objectifs

L'objectif de ce TME est de vous familiariser avec l'API de l'outil PeerSim en implémentant un protocole simple. Vous apprendrez à coder de protocole et à configurer une simulation en définissant ses différents paramètres.

1 Aperçu général de PeerSim

Le simulateur Peersim est un simulateur à événements discrets développé par une équipe d'académiques. Il est codé en Java, son implantation et son API sont relativement simples à comprendre (environ 19000 lignes de code dans sa globalité) et à utiliser (on utilise en général 5 ou 6 classes/Interfaces de son API). La figure 1 schématise le fonctionnement de peersim.

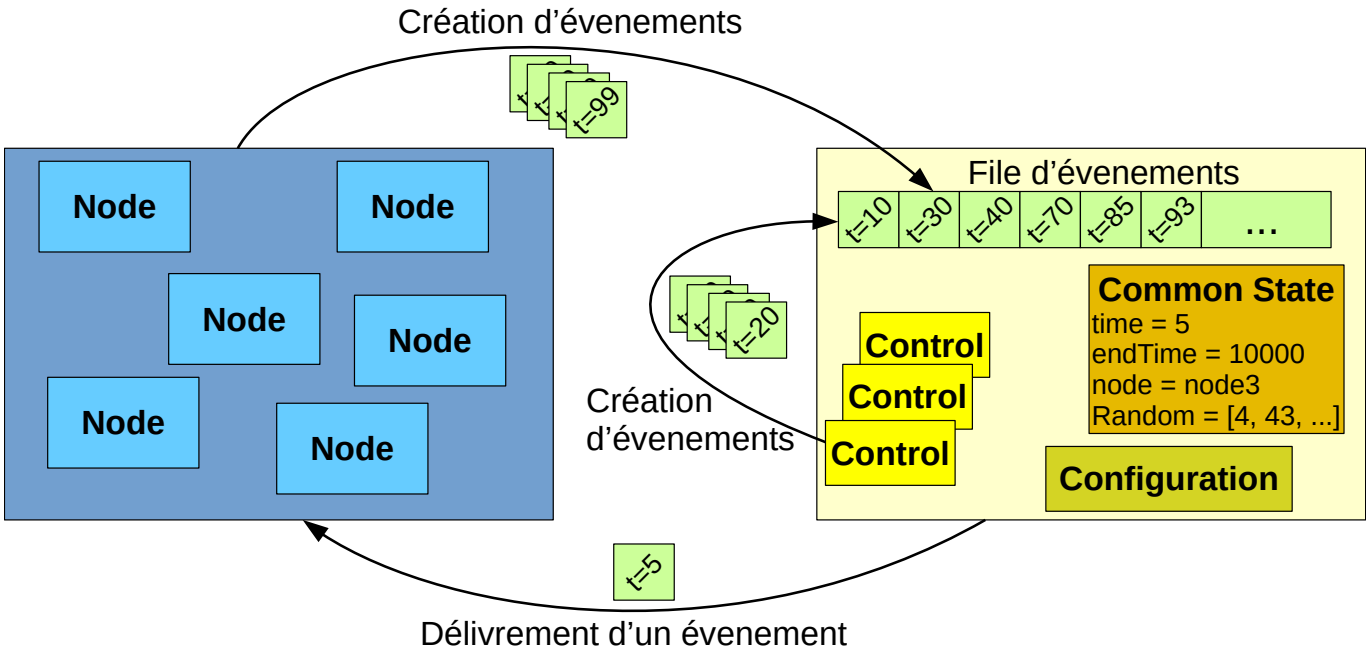


FIGURE 1 – Vue d'ensemble du simulateur Peersim

On peut remarquer deux parties :

- une partie bleu pour le code décrivant et implantant le système à étudier
- une partie jaune pour le code contrôlant la simulation.

Ainsi la partie jaune maintient la file des événements à délivrer dans la partie bleu. Cette file est triée par estampille croissante permettant ainsi de respecter l'ordre chronologique de la délivrance des événements. La partie bleu peut générer de nouveaux événements qui sont internes au système (ex : l'envoi d'un message). La partie jaune peut également via des contrôleurs créer de nouveaux événements notamment pour simuler des événements extérieurs au système ou non contrôlable par le système (ex : panne d'un nœud). On remarque également que la partie jaune maintient :

- les paramètres de configuration de la simulation
- un état courant global notamment pour la date courante, le date de la fin de simulation, le nœud sur lequel on exécute l'événement courant et un objet permettant de tirer des valeurs aléatoires à partir d'une graine (Random).

2 Architecture logicielle principale de Peersim

La figure 2 décrit sous le formalisme UML l'architecture logicielle principale de PeerSim en gardant le code couleur de la figure 1.

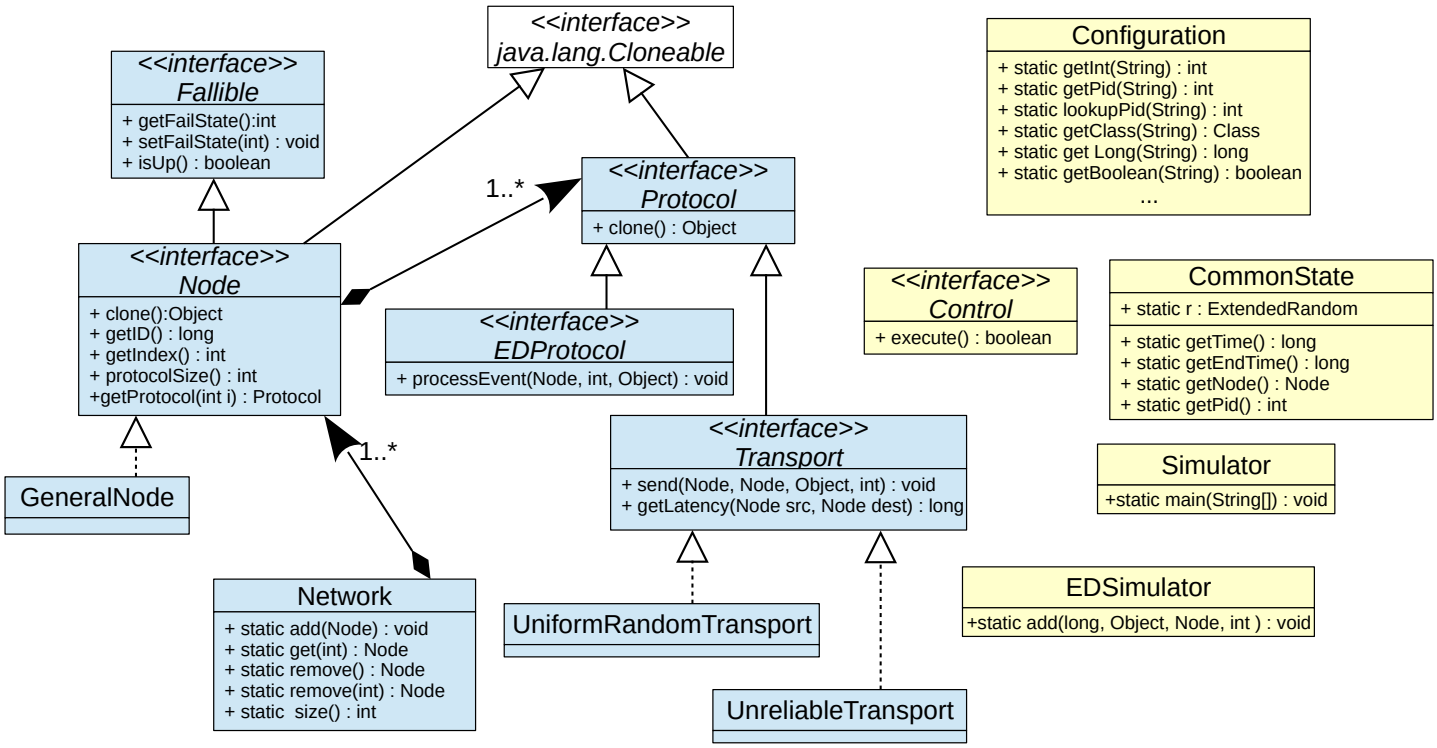


FIGURE 2 – Architecture logicielle principale du simulateur Peersim

Nous allons décrire brièvement chaque classe/interface de l'API. Afin de mieux appréhender ceci il vous est conseillé de regarder en même temps le code source.

2.1 Les classes/interfaces du système

2.1.1 La classe `peersim.Network`

Cette classe ne contient que des champs statique (= classe singleton) et maintient un tableau de Node qui représente l'ensemble des nœuds du système. On notera que chaque nœud est référencé à partir de cette classe par un index (i.e. l'indice dans le tableau). Il est possible d'avoir une référence

sur une instance de nœud grâce à la méthode `Network.get(int index)` en renseignant son indice dans le tableau.

N.B. : Dans un réseau statique (ensemble constant de nœuds tout au long de la simulation), l'indice peut faire office d'identifiant. En revanche si le réseau est dynamique (apparition/disparition de nœuds durant la simulation), l'indice ne peut pas être considéré comme un identifiant. PeerSim assure en revanche que chaque nœud a un identifiant unique (ID de type `long`) tout au long de la simulation. L'accès à un nœud depuis la classe `Network` à partir de son identifiant n'est malheureusement pas implanté.

2.1.2 L'interface `peersim.core.Node`

Cette interface permet de représenter un nœud du système. Un objet `Node` a un ID, un index dans le réseau, une liste de protocoles identifiés chacun par un indice (cf. section suivante) et doit avoir la capacité de se cloner. L'accès à une instance d'un protocole du nœud se fait par la méthode `getProtocol(int)` pour un indice de protocole. On notera que l'index d'un protocole au sein d'un nœud est le même sur tous les nœuds.

L'interface `Node` étend l'interface `peersim.Fallible` qui permet de contrôler l'"état de santé" d'un nœud. Les états de santé possible d'un nœud sont :

- `Fallible.OK` : nœud opérationnel et accessible
- `Fallible.DEAD` : nœud inaccessible définitivement
- `Fallible.DOWN` : nœud inaccessible temporairement

Il existe la classe `peersim.core.GeneralNode` qui est une implantation par défaut de l'interface `Node`. Cette implantation est généralement suffisante pour la plupart des simulation.

2.1.3 L'interface `peersim.core.Protocol`

Cette interface permet de décrire une couche protocolaire du système. C'est cette interface qui est implantée lorsque l'on programme un protocole/un algorithme distribué. On note l'existence de deux sous-interfaces :

- `EDProtocol` qui permet de programmer un protocole sur le modèle événementiel. Elle offre la méthode `processEvent` qui est appelée par le simulateur lorsqu'un événement doit être délivré sur un nœud pour ce protocole. Ainsi cette méthode prend trois arguments :
  - ◇ `Node` : l'instance du nœud sur lequel l'événement est délivré
  - ◇ `int` : entier égal à l'index du protocole sur lequel l'événement est délivré
  - ◇ `Object` : l'objet de l'événement
- `Transport` : qui permet de simuler la communication point à point entre deux nœuds du réseau. Cette interface offre deux méthodes :
  - ◇ `void send(Node src, Node dest, Object msg, int pid)` permettant d'envoyer un message `msg` du nœud `src` sur le protocole `pid` du nœud `dest`.
  - ◇ `long getLatency(Node src, Node dest)` permettant de savoir la latence d'un message que l'on souhaite simuler entre les nœuds `src` et `dest`.

  Il existe deux implantations de cette interface :
  - ◇ `UniformRandomTransport` qui définit une valeur de latence aléatoire uniforme compris entre deux paramètres min et un max à chaque envoi de message.
  - ◇ `UnreliableTransport` qui est un décorateur permettant de simuler la perte de messages avec une probabilité donnée.

Important : toute classe d'implantation de protocole doit avoir un constructeur prenant un argument de type `String` et doit implanter la méthode `Object clone()`.

2.2 Les classes/interfaces du contrôle de simulation

La plupart des classes contrôlant la simulation sont la plupart des classes singletons (que des champs statique) représentant ainsi l'ensemble des données et des paramètres de la simulation.

2.2.1 L'interface `peersim.core.Control`

L'interface `Control` représente un module de contrôle du simulateur qui peut être invoqué :
- à l'initialisation du système (à t=0). Ceci peut être utile pour par exemple construire la topologie du système, amorcer l'application, etc.
- pendant la simulation périodiquement (tous les x unités de temps) ou ponctuellement (à une date précise)
- à la fin de la simulation. Ceci peut être utile pour par exemple faire des statistiques sur des métriques de la simulation.

Un module de contrôle permet généralement de :
- simuler des événements périodiques du système, l'activité de la couche applicative ou des événements extérieurs comme les pannes, les départs de nœuds ou de liens, etc.
- monitorer le système

Un module de contrôle offre la méthode `boolean execute()` qui appelée par le simulateur lorsque le module doit être invoqué. Le retour de cette méthode permet d'indiquer au simulateur si la simulation doit être stoppée (résultat `true`) ou pas (résultat `false`)

Important : À l'instar des classes implantant les protocoles, le constructeur doit avoir un seul paramètre de type `String`.

2.2.2 La classe `peersim.core.CommonState`

Cette classe singleton permet de maintenir les variables globales de la simulation à savoir :
- la date courante
- la date maximum de simulation
- le nœud courant de la simulation, i.e. le nœud sur lequel on délivre l'événement courant
- l'identifiant du protocole courant, i.e. le protocole sur lequel on délivre l'événement courant

Cette classe expose un objet de type `peersim.util.ExtendedRandom` en attribut statique publique pour tirer des valeurs aléatoires pendant la simulation.

2.2.3 La classe `peersim.Simulator`

C'est la classe contenant le point d'entrée du programme java (`public static void main`). Le simulateur prend un paramètre le chemin vers un fichier de configuration décrivant les différents paramètres de la simulation (ex : nombre de nœuds, latence, temps de simulation, classe des protocoles, paramétrage des protocoles, etc.).

2.2.4 La classe `peersim.edsim.EDSimulator`

Cette classe est le moteur d'exécution d'une simulation à événements discrets. Elle maintient la file des événements, la liste des modules de contrôle à exécuter et définit la boucle principale du simulateur (méthode `executeNext`). Elle offre également la méthode `void add(long delay, Object event, Node node, int pid)` qui permet d'ajouter l'événement `event` dans la file. Un appel à cette méthode permet de délivrer l'événement `event` dans `delay` unités de temps au nœud `node` pour le protocole d'identifiant `pid`.

2.2.5 La classe `peersim.config.Configuration`

Cette classe permet d'accéder depuis le programme java aux paramètres définis dans le fichier de configuration renseigné en entrée de la fonction `main`. Elle offre notamment :

- les méthodes `int getInt(String k)`, `long getLong(String k)`, `double getDouble(String k)`, `boolean getBoolean(String k)`, `String getString(String k)` qui renvoient la valeur du paramètre associée à la clé `k`.
- la méthode `int lookupPid(String name)` renvoie l'identifiant du protocole de nom `name`
- la méthode `int getPid(String k)` renvoie l'identifiant d'un protocole si la clé `k` est associé à un nom de protocole dans le fichier de configuration . C'est équivalent à faire `lookupPid(getString(k))`

3 Le fichier de configuration

Le fichier de configuration permet de décrire textuellement les paramètres de la simulation. Le format de ce fichier se base sur une association de clé (nom du paramètre) et de valeur (valeur du paramètre). Concrètement une ligne du fichier définit un paramètre où la clé et la valeur sont séparé par un espace.

```
<nom du parametre> <valeur du parametre>
```

Les noms des paramètres sont normalisés avec une syntaxe hiérarchique (à l'instar des noms de package java) afin de les catégoriser (une simulation peut contenir plusieurs dizaines de paramètre) et ainsi de plus facilement définir sur quels objets ils agissent. Dans un fichier de configuration il possible de commenter des lignes avec le caractère d'échappement #.

3.1 Les paramètres globaux

La tableau suivant résume les paramètres globaux d'une simulation

Nom	Valeur par défaut	Commentaire
`simulation.endtime`	ENTIER REQUIS	date de fin de la simulation
`network.size`	ENTIER REQUIS	taille du réseau en nombre de nœuds
`random.seed`	date courante de la JVM	valeur de la graine du générateur aléatoire
`simulation.experiments`	1	nombre de fois où l'expérience est exécutée. À chaque expérience tous les paramètres sont réinitialisés sauf `random.seed`

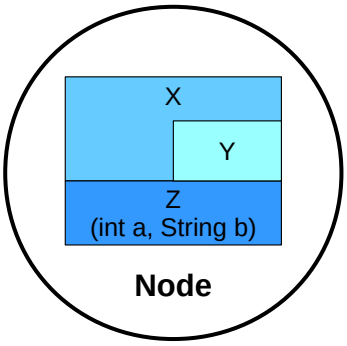
3.2 Paramétrer les protocoles et la pile protocolaire

Pour définir, instancier, paramétrer et lier les différents protocoles des nœuds du système il faut définir pour chaque protocole un nom en définissant une clé préfixée par `"protocol."`. La valeur de cette clé doit être le nom de la classe d'implantation du protocole, i.e la classe Java qui définit les différentes primitive du protocole et qui implante l'interface `Protocol`. Pour renseigner les différents attributs du protocole nécessaires à son instanciation, on définit une clé en la préfixant par la clé du protocole (`protocol.nom_protocol`) et en la suffixant par le nom du paramètre. Ceci donnera donc la ligne suivante pour chaque paramètre d'initialisation :

```
protocol.nom_protocol.nom_attibut valeur_attribut
```

Illustrons ceci par un petit exemple. On considère que le système est composé de 3 protocoles `px`, `py`, `pz` implantés respectivement par les classes `Px`, `Py` et `Pz`.

- Le protocole `px` a besoin des services du protocole `py` et du protocole `pz` pour fonctionner et nécessite l'initialisation de deux IDs de protocole désignés respectivement par les nom `protoy` et `protoz`. Il ne possède pas de paramètre à initialiser à la construction.
- Le protocole `py` a besoin des services du protocole `pz` pour fonctionner et nécessite l'initialisation d'un ID de protocole désigné par le nom `protoz` . Il ne possède pas de paramètre à initialiser à la construction.
- Le protocole `pz` est indépendant de tout protocole et ne possède pas de paramètre à initialiser à la construction. Il possède deux attributs `a` de type `int` et `b` de type `String` qu'il est nécessaire d'initialiser à la construction d'une instance du protocole.




```
protocol.pz Pz # nom de la classe àcharger pour pz
protocol.pz.a 45 # valeur de l'attribut a de pz
protocol.pz.b bonjour # valeur de l'attribut y de pz

protocol.py Py # nom de la classe àcharger pour py
protocol.py.protoz pz # le nom du protocole pour initialiser protoz
    de py

protocol.px Px # nom de la classe àcharger pour px
protocol.px.protoy py
protocol.px.protoz pz
```

3.3 Gestion des modules de contrôle

Pour définir, instancier et paramétrer les différents modules de contrôle de la simulation il faut définir pour chaque instance de module un nom en définissant une clé préfixée par `"control."` ou bien `"init."` si on souhaite que le module soit un module d'initialisation. La valeur de cette clé doit être le nom de la classe d'implantation du module de contrôle, i.e la classe Java qui implante l'interface `Control`. Le passage de paramètre de la construction se fera de la même manière que les protocoles.

La tableau suivant résume les différents paramètres pour configurer les moments de déclenchement des modules de contrôle.

Nom	Valeur par défaut	Commentaire
`init.<nom_du_module>`	CLASSE REQUISE	Déclare une instance de module de contrôle d'initialisation et prend comme valeur sa classe d'instanciation.
`control.<nom_du_module>`	CLASSE REQUISE	Déclare une instance de module de contrôle et prend comme valeur sa classe d'instanciation.
`control.<nom_du_module>.from`	0	Définit une date de début à partir de laquelle le module sera exécuté périodiquement
`control.<nom_du_module>.until`	`Long.MAX_VALUE`	Définit une date de fin de l'exécution périodique
`control.<nom_du_module>.step`	1	Définit le pas de déclenchement périodique du module
`control.<nom_du_module>.at`	-1	Définit une date ponctuelle à laquelle le module sera exécuté. La valeur -1 signifie qu'il ne sera pas exécuté ponctuellement.
`control.<nom_du_module>.FINAL`	aucune	Pas de valeur associée à cette clé. Sa déclaration signifie que le module sera exécuté une fois la simulation terminée. Incompatible avec `from`, `until`, `step` et `at`$\neq$ -1

Le fichier ci-dessous est un exemple déclarant un module d'initialisation, un module périodique, un module ponctuel, et un module de finalisation. La figure 3 illustre l'invocation de ces contrôles au cours du temps.

```
#definition des modules de controle d'initialisation
init.initialisation Initialisation # nom de la classe àcharger
init.initialisation.attributZ 78

#controles de simulation
control.monitor Monitor # nom de la classe àcharger
control.monitor.from 0 # date de début de repétition
control.monitor.until 10000 # date de fin de répétition
control.monitor.step 50 # pas de répétition

control.panne PanneAleatoire # nom de la classe àcharger
control.panne.at 5000 # module de controle ponctuel

#controle de finalisation
control.endcontroler EndControler # nom de la classe àcharger
control.endcontroler.at −1 # pas d'execution pendant la simu
control.endcontroler.FINAL # le controle doit s'executer après la fin
```

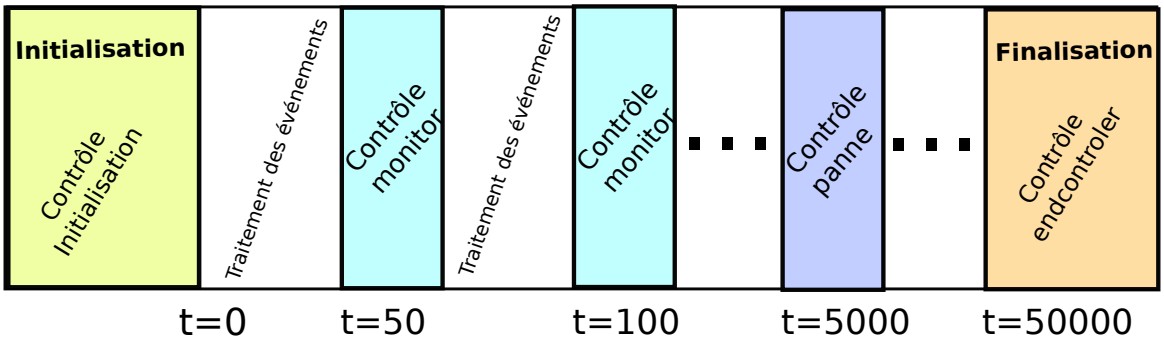


FIGURE 3 – Illustration chronologique de l'exemple

4 Déploiement et initialisation de la simulation

Lors du lancement de la classe `Simulator`, le simulateur charge le fichier de configuration et amorce l'initialisation du système. Lors de l'initialisation du système il construit les instances de nœud suivant le design pattern *prototype*. Dans ce design pattern le principe est de construire une instance d'objet prototype et d'appliquer un clonage de cet objet en profondeur pour générer les instances de nœud du système. C'est la raison pour laquelle il faut que les objets `Node` et `Protocol` soient `Cloneable`. Le prototype ne fait pas partie du système. Pour construire ce prototype il est nécessaire d'instancier l'ensemble des protocoles du système. Une fois que tous les objets représentant le système sont instanciés, le simulateur exécute les modules d'initialisation.

La figure 4 représente un aperçu de la mémoire de la JVM une fois les objets instanciés dans un système à 3 nœuds.

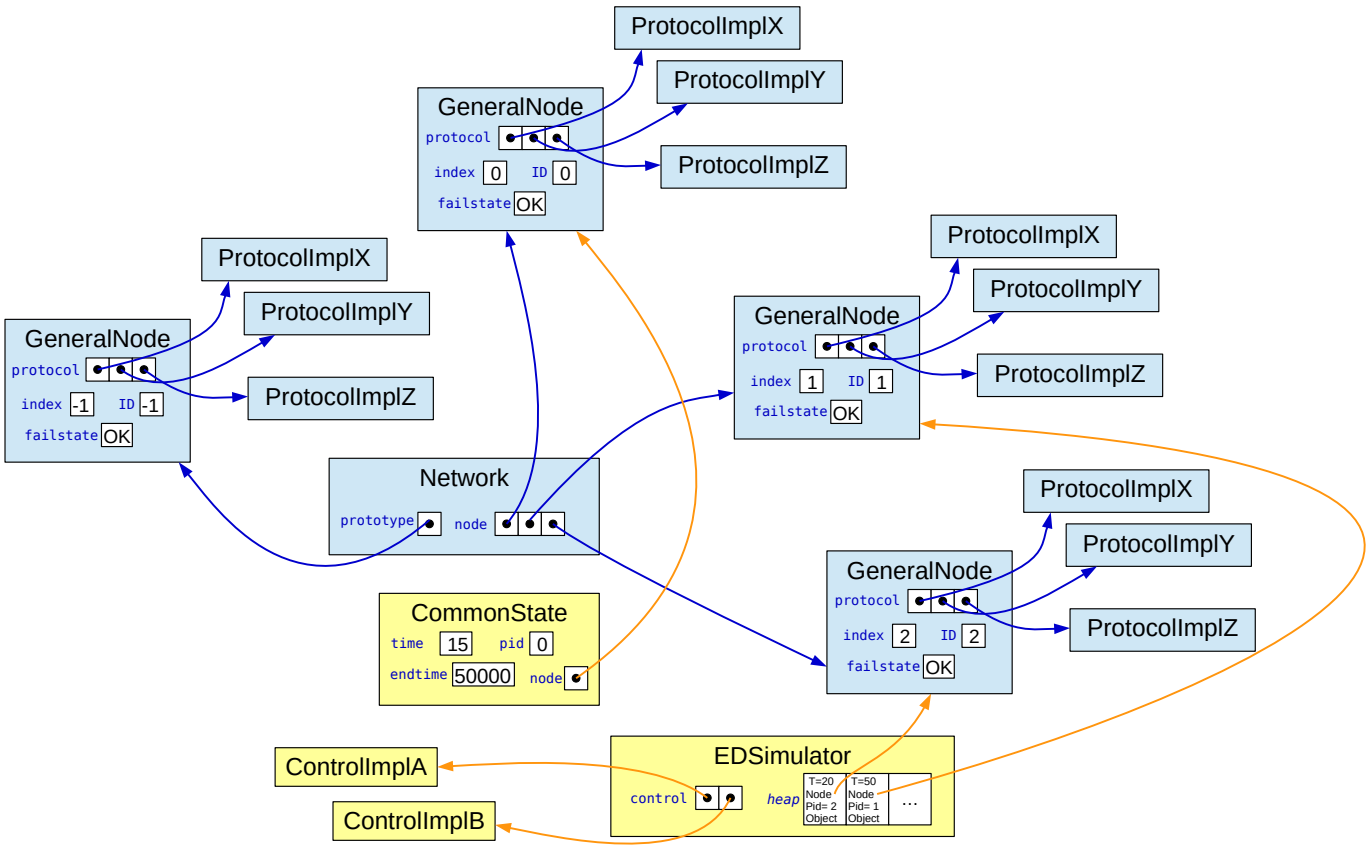


FIGURE 4 – Diagramme d'objets en mémoire dans un système à 3 nœuds

Tuto 1 – Installation de l'environnement de travail

Étape 1

Téléchargez le simulateur PeerSim et désarchivez-le :

```
wget http://downloads.sourceforge.net/project/peersim/peersim-1.0.5.zip

unzip peersim-1.0.5.zip
```

Étape 2

Ouvrez Eclipse et créez un nouveau projet Java (nommons-le par exemple TMEPeerSim).

Étape 3

Pour compiler et exécuter vos programmes PeerSim dans Eclipse, vous devez ajouter au Build-Path de votre projet les jars les différents fichiers jar présents à la racine de votre dossier d'installation PeerSim. Pour cela :

- Sélectionnez *Build Path* dans le menu contextuel (clic droit sur le dossier) du projet puis *Configure Build Path*
- Dans l'onglet *Libraries*, cliquez sur *Add External JARs....*
- Ajoutez l'ensemble des *Jar* présents à la racine de votre dossier Peersim
- Vérifiez que les Jar ont bien été ajoutés dans le répertoire *Referenced Libraries* dans votre projet Eclipse.

Étape 4

Afin de pouvoir parcourir le code source de PeerSim dans Eclipse (ceci peut être utile pour comprendre davantage son fonctionnement) :

- Zippez le dossier src de votre installation
  `zip -r src.zip src`
- Dans Eclipse, liez cette archive de fichiers sources au jar peersim-1.0.5.jar. Pour ceci :
  - ◇ Clic droit sur le fichier dans les *Referenced Libraries* puis *Properties*
  - ◇ Dans le menu *Java Source Attachment* sélectionnez *External Location* et indiquez l'archive zip en cliquant sur le bouton *External File*

Tuto 2 – Tutoriel : Implantation d'un protocole simple

On considère N nœuds d'un système distribué organisés en une topologie quelconque mais connexe. Chaque nœud possède une liste de taille aléatoire (paramétrable) d'entier tiré aléatoirement entre 0 et 128. À l'initialisation, le nœud d'identifiant 0 diffuse un message "hello" à tout le monde. Un message hello contient la liste aléatoire de son expéditeur. À la réception d'un message "hello", un nœud affiche le contenu du message (i.e., la liste) avec l'id de son expéditeur. Si le récepteur n'a jamais envoyé de message "hello", il diffuse à tout le monde son message "hello".

Étape 1

Nous allons coder la classe `HelloMessage` qui étend la classe abstraite `Message` suivante :

```
public abstract class Message {

	private final long idsrc;
	private final long iddest;
	private final int pid;

	public long getIdSrc() {return idsrc;}
	public long getIdDest() {return iddest;}
	public int getPid(){return pid;}

	public Message(long idsrc, long iddest, int pid){
		this.iddest=iddest;
		this.idsrc=idsrc;
		this.pid=pid;
	}
}
```

Cette classe `Message` permet de factoriser les informations communes à tout message à savoir l'identifiant de l'expéditeur, l'identifiant du destinataire et l'identifiant de protocole auquel le message est associé. Notre classe `HelloMessage` va donc rajouter comme attribut une liste d'entiers. Pour éviter les incohérences qui peuvent être liées à la mémoire partagée, il est préférable de déclarer tous les attributs constants (mot clé `final`) rendant ainsi toute instance de message immuable.

```
package ara;

import java.util.List;
import ara.util.Message;

public class HelloMessage extends Message {

	private final List<Integer> info;

	public HelloMessage(long idsrc, long iddest, int pid, List<Integer> info) {
		super(idsrc, iddest, pid);
		this.info=info;

	}
	public List<Integer> getInfo() {
		return info;
	}

}
```

Étape 2

Nous allons déclarer une classe `HelloProtocol` implantant `EDProtocol` L'en-tête de la classe est donc

```
public class HelloProtocol implements EDProtocol{
 //corps de la classe
}
```

Étape 3

Nous allons maintenant y coder les attributs. Tout d'abord il faut identifier les données qui seront renseignées dans le fichier de configuration : ici nous avons besoin de savoir quel est l'id du protocole de transport (pour l'envoie de message) et la taille maximum de la liste. On a donc besoin de définir deux constantes de type String pour définir le nom du paramètre dans le fichier de configuration et deux attributs d'instance de type int.

```
        private static final String PAR_TRANSPORT = "transport";
        private static final String PAR_MAXSIZELIST = "maxsizelist";

        private final int pid_transport;
        private final int maxsizelist;
```

À cela nous ajoutons les attributs internes au protocole :

```
private final int my_pid;// pour stocker l'identifiant du protocole
private List<Integer> mylist; //pour la liste propre àchaque noeud
private boolean deja_dit_bonjour=false;// pour indiquer si on a déjà envoyer les
    messages
```

On notera qu'on ne stockera pas de référence vers le nœud hôte dans les attributs. Cette référence sera en général renseignée en paramètre de chaque primitive du protocole

Étape 4

Nous allons maintenant coder le constructeur ainsi que la méthode `clone`. Le constructeur prend un paramètre de type `String` qui est égale à la clé du paramètre qui déclare le protocole dans le fichier de configuration. Cette chaine fais donc office de préfixe pour récupérer les clés des paramètres du protocole.

```
public HelloProtocol(String prefix) {
        String tmp[]=prefix.split("\\.");
        my_pid=Configuration.lookupPid(tmp[tmp.length-1]);
        pid_transport = Configuration.getPid(prefix+"."+PAR_TRANSPORT);
        maxsizelist=Configuration.getInt(prefix+"."+PAR_MAXSIZELIST);
        mylist=new ArrayList<>();
}
```

Rappelons que ce constructeur sera appelé pour construire le nœud prototype et que la méthode `clone` sera appelée pour construire les différentes instances de nœud. Dans la méthode `clone` il faut donc penser à cloner également les attributs référençant des objets afin que les nœuds ne partagent pas d'objet commun et éviter ainsi les bugs (n'oublions pas que l'on teste un système à mémoire distribuée dans un environnement à mémoire partagée).

```
public Object clone() {
        HelloProtocol ap = null;
        try { ap = ( HelloProtocol) super.clone();
```

```
            ap.mylist=new ArrayList<>();
        }
        catch( CloneNotSupportedException e ) {} // never happens
        return ap;
}
```

Étape 5

Nous allons coder une première primitive qui implante le traitement à faire lorsqu'un nœud souhaite faire sa diffusion du message `Hello`.

```
public void direBonjour(Node host) {
 Transport tr = (Transport) host.getProtocol(pid_transport);
 for(int i = 0 ; i < Network.size(); i++) {
    Node dest= Network.get(i);
    Message mess = new HelloMessage(host.getID(), dest.getID(), my_pid, new
        ArrayList<>(mylist));
    tr.send(host, dest,mess, my_pid);
 }
 deja_dit_bonjour=true;
}
```

Étape 6

Nous allons coder la primitive de traitement lorsque l'on reçoit un `HelloMessage`.

```
private void receiveHelloMessage(Node host, HelloMessage mess) {
 System.out.println("Noeud "+host.getID() +" : recu Hello de "+mess.getIdSrc()+ "
     sa liste = "+mess.getInfo());
 if(!deja_dit_bonjour) {
    direBonjour(host);
 }
}
```

Étape 7

Nous allons coder une primitive d'initialisation pour initialiser la liste aléatoire et permettre au nœud zéro d'initier la première diffusion.

```
public void initialisation(Node host) {
 int size_list= CommonState.r.nextInt(maxsizelist);
 for(int i=0;i<size_list;i++) {
    mylist.add(CommonState.r.nextInt(128));
 }
 if(host.getID() == 0) {
    direBonjour(host);
 }
}
```

Étape 8

Enfin il reste l'implantation de la méthode `processEvent` pour respecter l'interface `EDProtocol`. Dans notre cas cette méthode fait office d'aiguillage vers la bonne primitive du protocole une fois la vérification des arguments.

```
@Override
public void processEvent(Node host, int pid, Object event) {
```

```
 if(pid != my_pid) throw new IllegalArgumentException("Incohérence sur l'
    identifiant de protocole");
 if( event instanceof HelloMessage) {
   receiveHelloMessage(host,(HelloMessage) event);
 }else {
   throw new IllegalArgumentException("Evenement inconnu pour ce protocole");
 }
}
```

Étape 9

Nous allons coder maintenant le module d'initialisation qui permettra d'amorcer le protocole. Nous avons besoin d'une classe implantant `Control`

```
public class Initialisateur implements Control {
 ...
]
```

Étape 10

Le module d'initialisation a besoin de connaître l'id du protocole `HelloProtocole` pour pouvoir appeler sur chaque nœud la méthode `initialisation` codée précédemment. Cette information sera renseignée dans le fichier de configuration. Il est donc nécessaire de définir une constante de type String pour définir le nom du paramètre dans le fichier de configuration et un attribut de type int qui indiquera l'identifiant du protocole.

```
 private static final String PAR_PROTO_HELLO="hellopid";

 private final int hellopid;

 public Initialisateur(String prefix) {
   hellopid=Configuration.getPid(prefix+"."+PAR_PROTO_HELLO);
 }
```

Étape 11

Pour terminer la classe, nous allons enfin coder la méthode `execute`

```
@Override
public boolean execute() {
  for(int i = 0 ; i < Network.size() ; i++) {
    Node node = Network.get(i);
    HelloProtocol hello = (HelloProtocol) node.getProtocol(hellopid);
    hello.initialisation(node);
  }
  return false;
}
```

Étape 12

Pour terminer ce tutoriel, voici le contenu du fichier de configuration.

```
network.size 5
simulation.endtime 50000
random.seed 20

protocol.transport UniformRandomTransport
```

```
protocol.transport.mindelay 20                                  6
protocol.transport.maxdelay 30                                  7
                                                                8
                                                                9
protocol.hello HelloProtocol                                    10
protocol.hello.transport transport                              11
protocol.hello.maxsizelist 5                                    12
                                                                13
                                                                14
init.i Initialisateur                                           15
init.i.hellopid hello                                           16
```

Étape 13

Lorsque vous lancez plusieurs fois la simulation vous pourrez remarquer que l'affichage reste le même tant que la graine ne change pas. On arrive donc à avoir une exécution déterministe.