Évaluation de systèmes répartis à large échelle

Sergey Legtchenko

LIP6-INRIA

Besoin de concevoir des systèmes massivement répartis.

Motivation :

- Tolérance aux pannes
- Stockage de données critiques
- Coût réduit
- Seul moyen de réellement passer à l'échelle

Comment tester le système ?

Problème :

Si on veut tester l'application dans les conditions réelles,

- Il faut trouver un grand nombre de sites répartis pendant la phase de test.
- Coût élevé
- Résultat non garanti

Peu optimal !

Solution

Concevoir des plates-formes d'évaluation pour pouvoir :

- Émuler un grand nombre de nœuds (au moins supérieur à 1000)
- Injecter des fautes pour observer la réaction du système
- Vérifier des propriétés algorithmiques

Plan

Plates-formes d'évaluation réparties

- Un ensemble de sites geographiquement éloignés
- Chaque machine émule plusieurs nœuds virtuels (selon sa capacité)
- Les communications entre nœuds virtuels d'un même site sont ralenties pour simuler le réseau

On obtient un très grand nombre de nœuds virtuels pour faire tourner l'application.

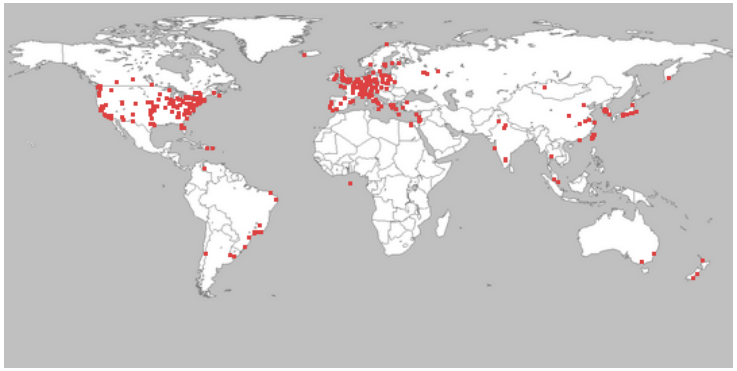
PlanetLab

PLANETLAB
An open platform for developing, deploying, and accessing planetary-scale services

À l'heure actuelle :

- 1132 machines connectées par Internet
- 516 sites répartis sur toute la planète

Répartition de PlanetLab



Grid5000

- Plate-forme française, 3200 processeurs
- Réseau par fibre optique

Processors \ Sites	Orsay	Grenoble	Lyon	Rennes	Sophia
AMD Opteron	684		260		356
Intel Xeon EM64T		68		326	
Sites total	684	68	260	326	356

Processors \ Sites	Bordeaux	Lille	Nancy	Toulouse
AMD Opteron	322	198	94	276
Intel Xeon EM64T	102	92	424	
Sites total	424	290	518	276

Avantages des plates-formes réparties

- Bonne montée en charge
- On peut faire tourner la vraie application sur les nœuds virtuels
- L'application est réellement distribuée entre des sites physiquement distants

Proche d'un essai grandeur nature

Inconvénients des plates-formes réparties

- Il faut être enregistré pour pouvoir y accéder
- Il faut réserver les nœuds par avance et pour un temps limité
- Difficile à prendre en main
- Des systèmes de déploiement pas toujours performants
- Debug difficile

Plates-formes centralisées : les simulateurs

Idée :

- On met au point un modèle simplifié du système original
- Le simulateur tourne sur une seule machine qui simule l'ensemble des nœuds.

La memoire d'une machine est en général suffisante pour simuler quelques milliers de nœuds simplifiés.

Les simplifications du système

Simplifications faites sur :

- Le code applicatif
- Les couches réseau
- Les contraintes physiques (pannes, latence, etc.)

Objectif : Tester l'intéraction entre les nœuds du système

- Les nœuds de l'application sont considérés comme des modules qui échangent des messages
- On ne simule pas (ou peu) le fonctionnement interne de chaque nœud

Network Simulator (ns-{1,2,3})

Avantages

- Open Source (en C++)
- Simule précisement les protocoles réseau (TCP, WiFi, etc.)
- Très répandu dans la communauté réseau

Inconvénient

Relativement difficile d'accès

Omnet

Utilisé pour simuler des réseaux, mais aussi des architectures multi-processeurs, des applications multithreadées, etc.

Avantages :

- Très générique : simule des modules qui échangent des messages
- Permet de générer des graphes de séquence, une représentation graphique de la topologie, etc.

Inconvénient :

Trop générique : incompatibilités entre les versions et les modules

Peersim

Avantages :

- Moins de 20000 lignes de code Java (Javadoc comprise)
- API très simple d'utilisation (comparé aux autres simulateurs)

Inconvénients :

- Parfois un peu trop simpliste (pas de simulation fine des protocoles réseau, ni même de gestion de bande passante)
- Relativement lent

En résumé :

- Soit on teste le vrai système sur une plate-forme répartie
- Soit on conçoit un modèle simplifié et on teste dans un simulateur

La deuxième solution peut être satisfaisante dans de très nombreux cas.

Plan

Simulation événementielle

Toutes les plates-formes de simulation sont basées sur le modèle à événements discrets.

Idée : Discrétiser le temps

- Deux entités : les nœuds et les messages
- On considère que le temps évolue seulement lorsqu'un événement survient sur un nœud

Évolution du temps global au sein de la simulation en fonction des messages reçus

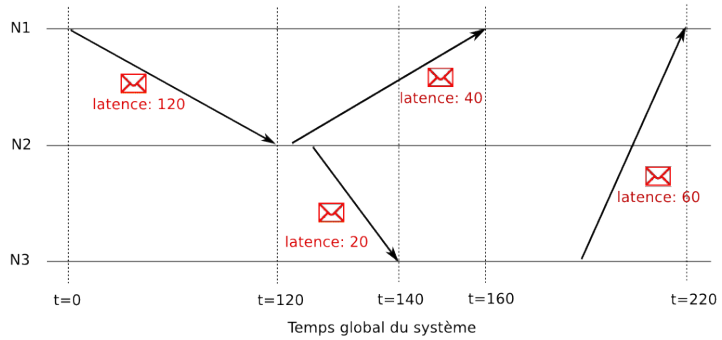
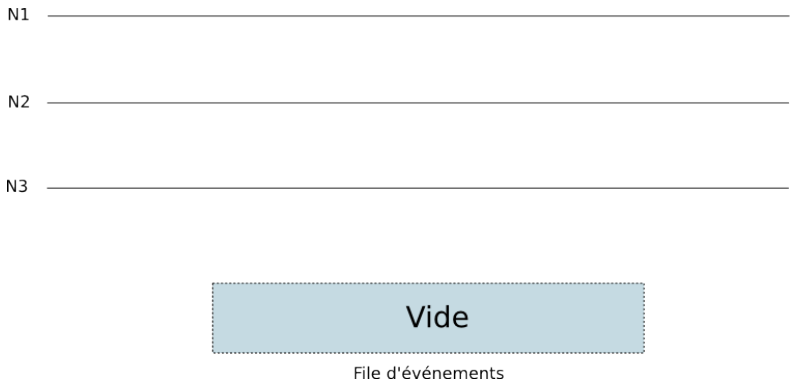


FIGURE:

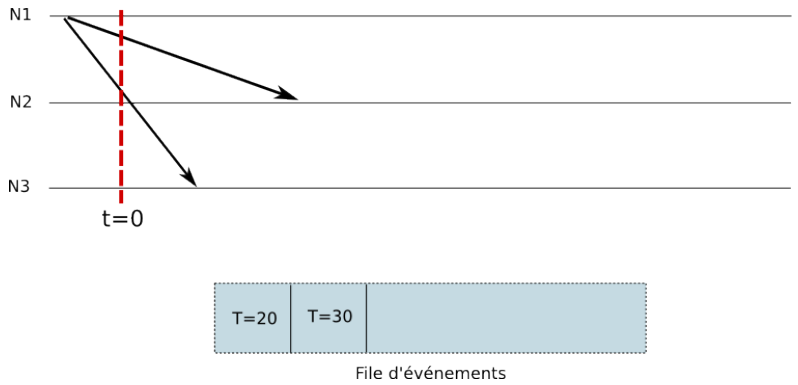
Gestion des messages (1)

- Chaque message envoyé est estampillé avec son temps de *récéption*
- Lors de l'envoi, les messages sont insérés dans une file à priorités en fonction de leur estampille.
- Lorsque le simulateur a calculé l'état de l'application à un instant t, il récupère un nouveau message en tête de file, et le délivre au nœud déstinataire.

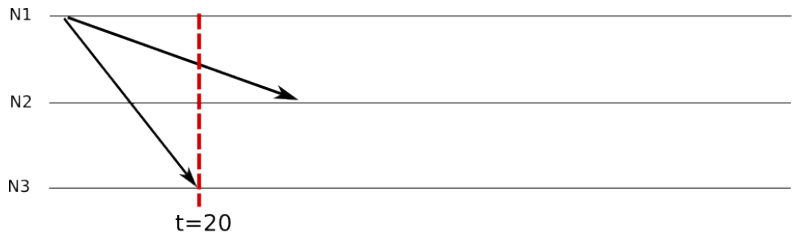
Gestion des messages (2)



Gestion des messages (2)

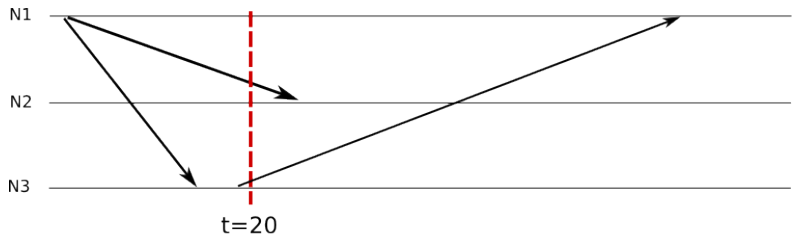


Gestion des messages (2)



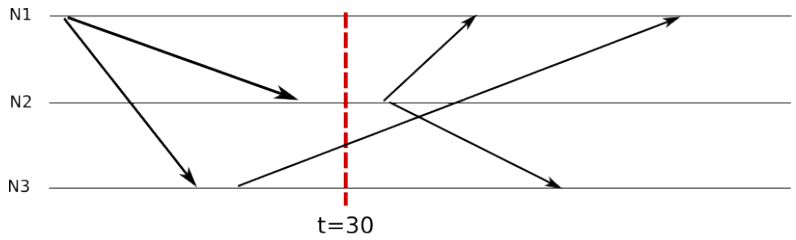
File d'événements

Gestion des messages (2)



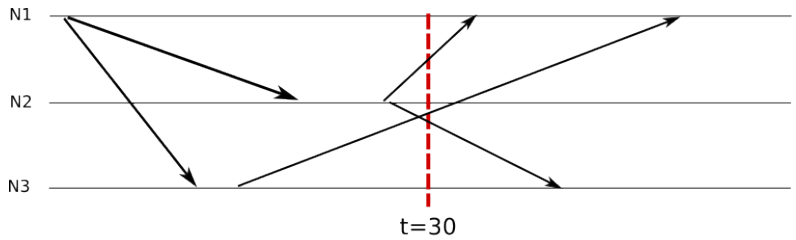
File d'événements

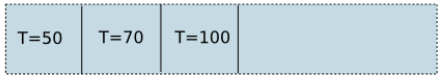
Gestion des messages (2)



File d'événements

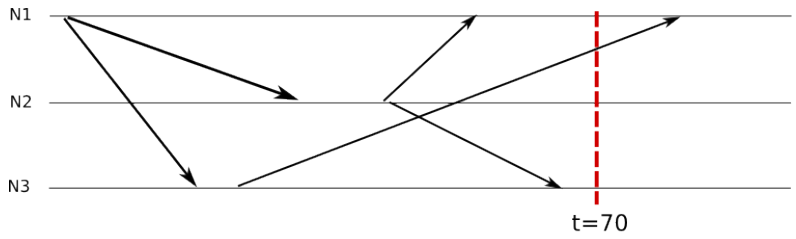
Gestion des messages (2)





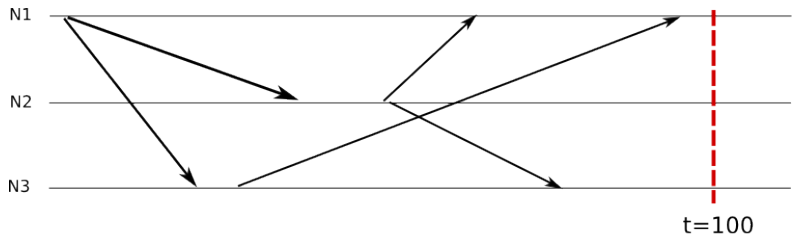
File d'événements

Gestion des messages (2)



File d'événements

Gestion des messages (2)



File d'événements

Gestion des messages

On simule en fait la réaction du système aux événements :

- Récéption de messages
- Événements internes aux nœuds

Mais pas le comportement du système entre les événements.

Avantages

- La charge de calcul est allégée : on ne simule pas le comportement du système entre les événements.
- Il est possible de faire des sauvegardes d'un état du système, il est donc très simple de faire des arrêts du système pour reprendre la simulation à un autre moment.
- La simulation est reproductible : on peut redéclencher le même bug encore et encore jusqu'à sa résolution.
- Le debug est facilité : on connaît à tout moment les messages en transit et leur temps de délivrement
- On ne code que les traitants des événements

Inconvénients

On fait des hypothèses de simplification par rapport au système réel

On peut perdre du temps par rapport à la réalité si le système qu'on simule génère trop d'événements.

Il faut correctement choisir le grain des événements :

- Un grain trop fin ralentit excessivement la simulation (et ne passe pas à l'échelle)
- Un grain trop gros risque de fausser le résultat de la simulation.

Économie de la mémoire (1)

La mémoire est occupée par :

- Les nœuds simulés (gros objets *relativement* peu nombreux).
- Les messages envoyés (multitude de petits objets).

Il faut limiter au maximum le nombre de messages envoyés tout en respectant au maximum les spécifications du système simulé.

Économie de la mémoire (2) : exemple

Scénario :

- À chaque cycle, tout nœud envoie un ping à ses voisins logiques.
- Au cycle suivant, le nœud supprime les voisins qui n'ont pas répondu au ping.

Problème :

- Le protocole est coûteux : à chaque cycle, chaque nœud inonde tout son voisinage avec ses messages.
- Comment éviter des envois de messages sans modifier le résultat de la simulation ?

Économie de la mémoire (3) : simplification du système

Si le passage à (très) grande échelle est vraiment nécessaire :

- Aggregation de protocoles.
- Approximation des temps de delivrance des messages.
- Suppression de la simulation de certaines couches protocolaires.

Possibilité d'atteindre plusieurs centaines de milliers de nœuds.

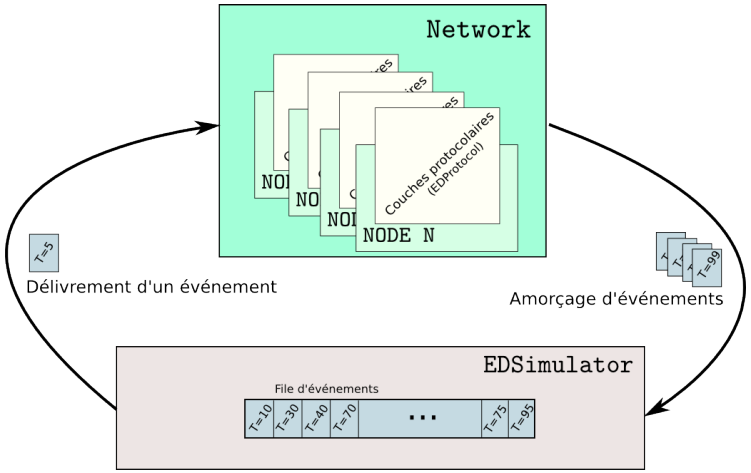
Attention aux effets de bord !

Plan

Entités principales

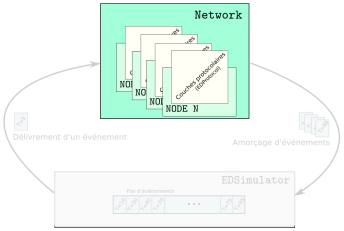
- `Node` (classe) : représente un nœud
- `EDProtocol` (interface) : représente un protocole
- `Network` (classe statique) : classe pour manipuler les nœuds
- `EDSimulator` (classe statique) : gère les événements

Vue d'ensemble du simulateur Peersim



Classe Network

Il s'agit d'un tableau qui contient l'ensemble des nœuds



Methodes utiles :

- `Network.add` : ajoute un nœud
- `Network.get` : retourne le nœud dont l'index est en paramètre
- `Network.remove` : supprime un nœud

Classe Node

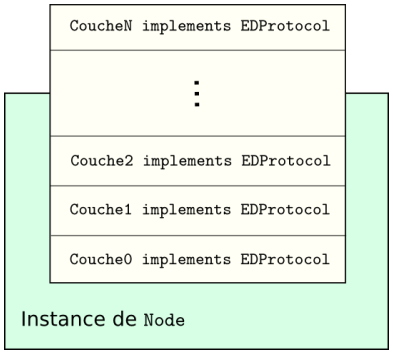


FIGURE: *Chaque objet Node contient une instance de chaque protocole. Il s'agit simplement d'un objet conteneur de protocoles.*

Classe Node (2)

Méthodes utiles :

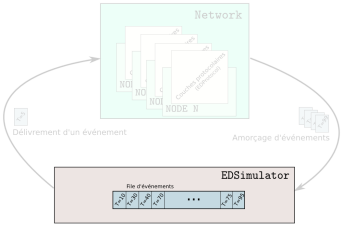
- `setFailState` : Activer/Désactiver un nœud (simuler une connexion/déconnexion)
- `getFailState` : Information sur l'état du nœud (activé ou désactivé)
- `getProtocol` : retourne l'objet protocole dont l'identifiant est passé en paramètre.

Interface EDProtocol

- Symbolise une couche réseau.
- Un objet avec l'interface EDProtocol est associé à un identifiant de protocole.
- Deux methodes obligatoires : `processEvent` et `clone`

Classe EDSimulator

- Contient la file d'événements
- Extrait les événements en tête de file et les délivre aux nœuds déstinataires



Méthodes utiles :

`EDSimulator.add` : ajoute un message dans la file

Envoi d'un message

On invoque `EDSimulator.add(t,event,N,pid);`

Le message mis en file d'attente possède donc :

- `t` : Un temps de délivrement
- `N` : Un nœud cible sur lequel délivrer le message
- `pid` : L'identifiant d'un protocole

Au temps `t`, la méthode `processEvent` du protocole ayant l'identifiant `pid` situé sur le nœud `N` sera invoquée par le simulateur.

Modules d'initialisation

- Classe implémentant l'interface `peersim.core.Control`
- Possède une méthode `execute` invoquée une seule fois au début de la simulation

Nécessaire pour le bootstrap du système :

- Construction de la topologie du système
- Amorçage de la couche applicative

Modules de contrôle

- Classe implémentant l'interface `peersim.core.Control`
- Sa méthode `execute` est périodiquement invoquée pendant toute la durée de la simulation

Permet de simuler :

- Des événements périodiques du système (protocoles de maintenance)
- L'activité de la couche applicative
- Des événements extérieurs : pannes, départs de nœuds, etc.

Diagramme type d'exécution de la simulation

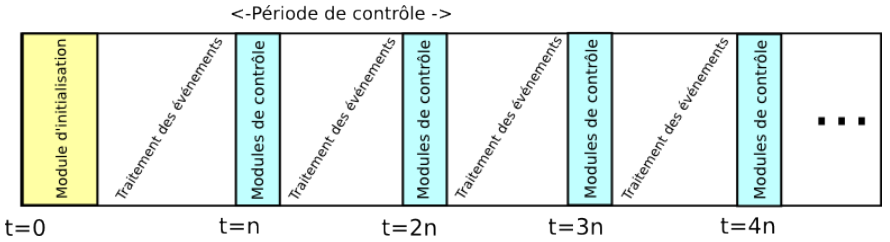


FIGURE: *Après la phase d'initialisation, les modules de contrôle sont exécutés de façon périodique*

Le fichier de configuration

Contient des couples entrées/valeurs

Permet de :

- Spécifier des paramètres de simulation (taille du réseau, temps de simulation, etc.)
- Fixer les paramètres des différents protocoles (intervalle de maintenance, temps de latence, etc.)
- Faire le lien entre les différents protocoles (initialiser le modèle en couches)

Spécification des couches protocolaires

Soit un modèle avec deux couches :

- Couche transport : la classe `MyTransport`
- Couche applicative : la classe `MyApplicative`

- La couche applicative utilise la couche transport pour propager ses messages.
- Une instance de `MyApplicative` doit donc posséder une instance de `MyTransport`.

Spécification des couches protocolaires (2)

Comment faire ?

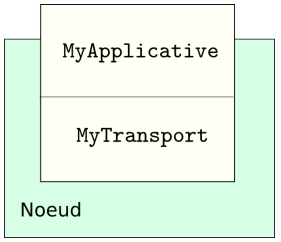
- On peut créer un objet MyTransport à l'initialisation de `MyApplicative`.

Problème : Si on veut remplacer `MyTransport` par `MyTransport2`, il faut changer le code de `MyApplicative`.

- On peut spécifier la dépendance entre MyApplicative et MyTransport dans le fichier de configuration.

On conserve ainsi la généricité des couches protocolaires.

Spécification des couches protocolaires (3)



Fichier de configuration :

```
protocol.tr MyTransport

protocol.appli MyApplicative
protocol.appli.transport tr
```

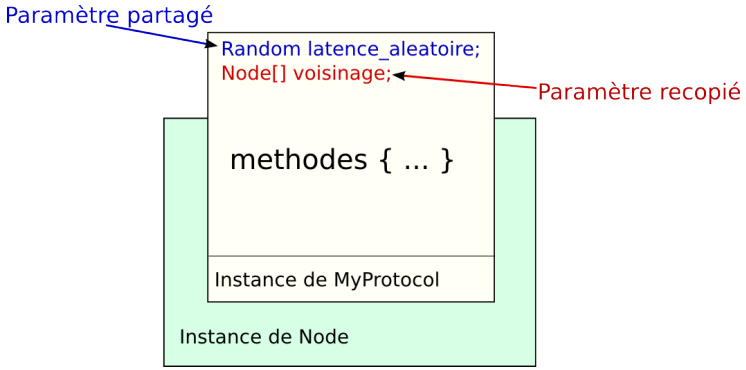
Clonage des nœuds

Au lancement de la simulation :

- Un objet `Node` contenant une instance de `MyTransport` et `MyApplicative` est crée.
- L'objet est cloné pour obtenir l'ensemble des nœuds.

- Il faut differencier les variables d'instance qui peuvent être partagées de celles qui doivent être copiées.
- Tout copier prend beaucoup de mémoire.
- Tout partager fausse la simulation.

Clonage des nœuds : exemple



Spécification d'un module d'initialisation

Soit une classe `MyBootstrap` initialisant le système.
Dans le fichier de configuration :

init.mondemarrage MyBootstrap

- Le mot clé init précise qu'il s'agit d'un module d'initialisation.
- mondemarrage est le nom de l'entrée.
- MyBootstrap est la valeur de l'entrée (ie. le nom de la classe).
- Au lancement du simulateur, un chargeur de classe est invoqué pour charger les classes spécifiées dans le fichier de configuration.

Mots clés importants du fichier de configuration :

- **init**.monmodule MaClasse charge MaClasse en tant que module d'initialisation
- **control**.monmodule MaClasse charge MaClasse en tant que module de contrôle
- **protocol**.monmodule MaClasse charge MaClasse en tant que protocole
- **simulation.experiments** permet de spécifier le nombre d'expériences consécutives (en général une seule)
- **simulation.endtime** permet de spécifier le temps de terminaison de l'expérience
- **network.size** spécifie la taille du réseau (en nombre de nœuds)

Plan

Système simulé

Scénario :

- On initialise 10 nœuds.
- À l'initialisation, le nœud d'identifiant 0 diffuse un message "hello" à tous les autres.
- À la réception, un nœud affiche le message à l'écran.

Le système est composé de :

- Un protocole de transport (juste pour modéliser la latence).
- Un protocole de helloWorld (qui sert à émettre et afficher le HelloWorld).
- Un module d'initialisation (qui déclenche l'envoi par le nœud 0 du message "hello").

Protocole de transport

```
//envoi d'un message: il suffit de l'ajouter a la file d'evenements
public void send(Node src, Node dest, Object msg, int pid) {

    long delay = getLatency(src,dest);
    EDSimulator.add(delay, msg, dest, pid);

}


//latence random entre la borne min et la borne max
public long getLatency(Node src, Node dest) {

    return (range==1?min:min + CommonState.r.nextLong(range));

}
```

Protocole de HelloWorld

```
//methode appelee lorsqu'un message est recu par le protocole HelloWorld du noeud
public void processEvent( Node node, int pid, Object event ) {
    this.receive((Message)event);
}

//envoi d'un message (l'envoi se fait via la couche transport)
public void send(Message msg, Node dest) {
    this.transport.send(getMyNode(), dest, msg, this.mypid);
}

//affichage a la reception
private void receive(Message msg) {
    System.out.println(this + ": Received " + msg.getContent());
}
```

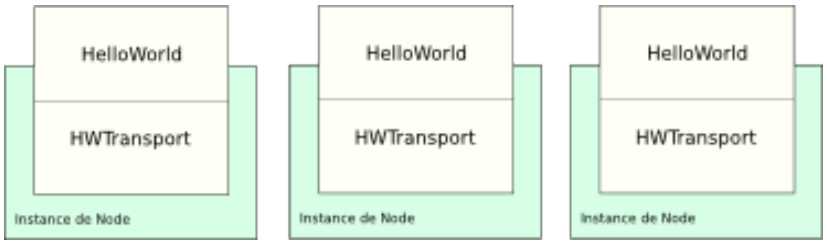

Fichier de configuration : définition des couches

```
#definition de la couche transport
protocol.transport helloWorld.HWTransport

#definition de la couche applicative (le hello world)
protocol.applicative helloWorld.HelloWorld
```

- Le chargeur de classes de la JVM est invoqué par Peersim pour charger les classes déclarées comme protocole.
- Il attribue ensuite un identifiant (pid) à chaque protocole.

Vue d'ensemble des couches



Liaison entre les couches

```
#definition de la couche transport
protocol.transport helloWorld.HWTransport

#definition de la couche applicative (le hello world)
protocol.applicative helloWorld.HelloWorld
#liaison entre la couche applicative et la couche transport
protocol.applicative.transport transport
```

- Sur chaque nœud il y a une instance de HWTransport et de HelloWorld.
- L'instance de HelloWorld connaît l'identifiant (pid) du protocole HWTransport.

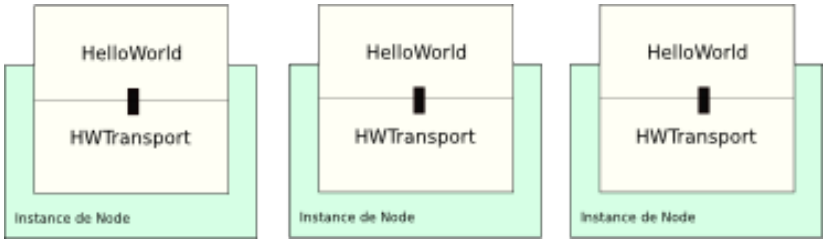
Liaison entre les couches (2)

Il suffit à l'objet HelloWorld de récupérer une référence vers l'objet HWTransport situé sur le même nœud.

Dans l'objet HelloWorld situé sur le nœud i appeler :

```
//liaison entre un objet de la couche applicative et un
//objet de la couche transport situes sur le meme noeud
public void setTransportLayer(int nodeId) {
    this.nodeId = nodeId;
    this.transport = (HWTransport) Network.get(this.nodeId).getProtocol(this.transportPid);
}
```


Vue d'ensemble des couches



Fichier de configuration : paramètres du simulateur

```
#Nombre de simulations consecutives
simulation.experiments 1

#date de fin de la simulation
simulation.endtime 3000

#taille du reseau
network.size 10
```


Passage de paramètres via le fichier de configuration

```
#latence minimale
protocol.transport.mindelay 80

#latence maximale
protocol.transport.maxdelay 120
```

FIGURE: Ecriture des paramètres dans le fichier de configuration.

```
public HWTransport(String prefix) {
    System.out.println("Transport Layer Enabled");

    //recuperation des valeurs extremes de latence depuis le fichier de configuration
    min = Configuration.getInt(prefix + ".mindelay");
    max = Configuration.getInt(prefix + ".maxdelay");
```

FIGURE: Récupération dans le code java.

Fichier de configuration : module d'initialisation

```
# ::::: INITIALIZER :::::

#declaration d'un module d'initialisation
init.initializer helloWorld.Initializer

#pour que le module connaisse le pid de la couche applicative
init.initializer.helloWorldProtocolPid applicative
```


Module d'initialisation (méthode execute)

```
public boolean execute() {
    int nodeNb;
    HelloWorld emitter, current;
    Node dest;
    Message helloMsg;

    //recuperation de la taille du reseau
    nodeNb = Network.size();
    //creation du message
    helloMsg = new Message(Message.HELLOWORLD,"Hello!!");
    if (nodeNb < 1) {
        System.err.println("Network size is not positive");
        System.exit(1);
    }

    //recuperation de la couche applicative de l'emetteur (le noeud 0)
    emitter = (HelloWorld)Network.get(0).getProtocol(this.helloWorldPid);
    emitter.setTransportLayer(0);

    //pour chaque noeud, on fait le lien entre la couche applicative et la couche transport
    //puis on fait envoyer au noeud 0 un message "Hello"
    for (int i = 1; i < nodeNb; i++) {
        dest = Network.get(i);
        current = (HelloWorld)dest.getProtocol(this.helloWorldPid);
        current.setTransportLayer(i);
        emitter.send(helloMsg, dest);
    }

    System.out.println("Initialization completed");
    return false;
}
```


Pour finir

- Le code de Peersim est bien documenté.
- La Javadoc est disponible ici :
  http ://peersim.sourceforge.net/doc/index.html