Rapport PSAR

API générique pour le développement d'applications réparties

Tarik Atlaoui

Nicolas Peugnet

Kimmeng Ly

Max Eliet

08 Juin 2020

Table des matières

1 Introduction

1.1 Les applications réparties

Avant de commencer, il est important de définir ce qu'est une application répartie. Dans notre cas, nous pouvons voir une application répartie comme un ensemble d'entités logicielles, de composants qui peuvent être développés dans différents langages de programmation, s'exécutant sur plusieurs sites et qui sont reliés entre eux par une interface ou un réseau de communication.

1.2 Les difficultés à programmer une application répartie

Aujourd'hui la programmation d'applications réparties est devenue une réalité du monde informatique, cette forme de programmation permet d'augmenter la disponibilité des applications et de diminuer leur temps d'exécution. Cependant, réaliser une application répartie reste une tâche délicate. En effet, nous devons prendre en compte la maintenabilité et la réutilisabilité des programmes. De plus, les accès concurrents peuvent créer des erreurs et des sources d'incohérence. C'est pourquoi il est très important de montrer que ces programmes fonctionnent bien avec des tests unitaires tout en respectant le cahier des charges.

2 Méthode de développement

2.1 API réelle : avantages/inconvénients et exemple (MPI ou autre)

L'API de MPI est avant tout une norme pour le passage de messages entre différents ordinateurs ou au sein d'un même ordinateur. Elle est énormément utilisée pour la communication sur des architectures distribuées. Dans notre cas, elle s'est révélée extrêmement intéressante car MPI a été implémentée sur presque toutes les architectures ainsi elle a été adaptée pour chacune de la façon la plus optimale. Cependant les tests et le debug sur celle-ci restent difficiles dû a l'impossibilité d'avoir un contrôle sur les évenements qui apparaissent, contrairement à PeerSim.

2.2 API simulation à évenements discrets :avantages/inconvénients et exemple(PeerSim ou autre)

Qu'entend-on par *simulation à événements discrets* ? C'est une simulation dont le temps évolue seulement lorsqu'un événement survient sur un noeud, et donc de même l'état du système ne peut être modifié qu'à ces moments là.

On distingue donc deux entités : les noeuds, et les événements qui sont caractérisés par une date de délivrance, un noeud destinataire, et des données.

Les principaux avantages d'un tel type de simulation sont : son déterminisme et donc une capacité à reproduire des bugs, et une charge de calcul réduite aux événements qu'on décide de simuler. Toutefois, il faut savoir trouver un équilibre entre une simulation trop simpliste et une simulation trop précise ralentie par trop d'événements.

Dans notre cas, nous nous sommes dirigés vers PeerSim comme simulateur à événements discrets, car il est codé en Java et possède une API relativement simple d'utilisation.

PeerSim est utilisé pour créer des nœuds et simuler une architecture pair-à-pair en générant des protocoles et des événements qui sont définis par l'utilisateur. Une éxécution est toujours ce qui permet d'accélérer le débug d'une application répartie avant son déploiement ou de reproduire une séquence d'événements qui a provoqué un bug sur une application déja existante.

2.3 Aperçu de l'implémentation d'un anneau avec les deux API

Implémentation en MPI

```
public class RingMpi {

   public static void main(String[] args) {
         MPI.Init(args);
         Comm comm = MPI.COMM_WORLD;
         int size = comm.getSize();
         int rank = comm.getRank();
         int neighbour = (rank + 1) % size;
         int hellotag = 1;
         Integer msg = 0;
            Status status;

         if (rank == 0) {
            comm.send(msg, 0, MPI.INT, neighbour, hellotag);
            status = comm.recv(msg, 0, MPI.INT, MPI.ANY_SOURCE, hellotag);
         } else {
            status = comm.recv(msg, 0, MPI.INT, MPI.ANY_SOURCE, hellotag);
            comm.send(msg, 0, MPI.INT, neighbour, hellotag);
            }

         System.out.println(rank + " Received hello from " + status.getSource());
         MPI.Finalize();
      }
   }
}
```

Implémentation en PeerSim

```
public class HelloProtocol implements EDProtocol {
   //Declarations d'attributs et fonctions retirees pour la clarte du code
   ...
   //Un noeud souhaite faire sa diffusion du message a son voisin
   public void direVoisin(Node host) {
      Transport tr= (Transport) host.getProtocol(pid_transport);
      Node dest=Network.get((int) ((host.getID()+1)%Network.size()));
      Message mess= new Message(host.getID(),(host.getID()+1)%Network.size(),my_pid, new
          ArrayList<>(myList));
      tr.send(host, dest, mess, my_pid);

      deja_dit_voisin=true;
   }

   //Traitement a effectuer lorsqu'on recoit un HelloMessage
   private void receiveHelloMessage(Node host, HelloMessage mess) {
      System.out.println("Noeud "+ host.getID() + " : a recu Hello de "+mess.getIdsrc()+ " sa liste
          = "+mess.getInfo());
      if(!deja_dit_voisin) {
         direVoisin(host);
      }
   }
}
```

3 Motivation et objectif global

3.1 API générique : avantages, modèle de programmation(ici événementiel)

Le but de notre projet est de produire une API générique sur un modèle de programmation événementielle afin de faciliter le développement de futures applications réparties, en permettant de s'abstraire du support d'éxécution au niveau du code métier.

Pouvoir éxécuter le même code métier aussi bien sur une plateforme réelle que sur un simulateur permet au développeur d'une application répartie de passer de l'un à l'autre, sans risquer d'en modifier son comportement en adaptant le code.
Il peut donc profiter des avantages d'un simulateur, comme la vision globale du système et le déterminisme des séquences d'éxécution sans crainte d'y introduire de nouveaux bugs.

3.2 Aperçu de l'implémentation d'un anneau avec l'API générique

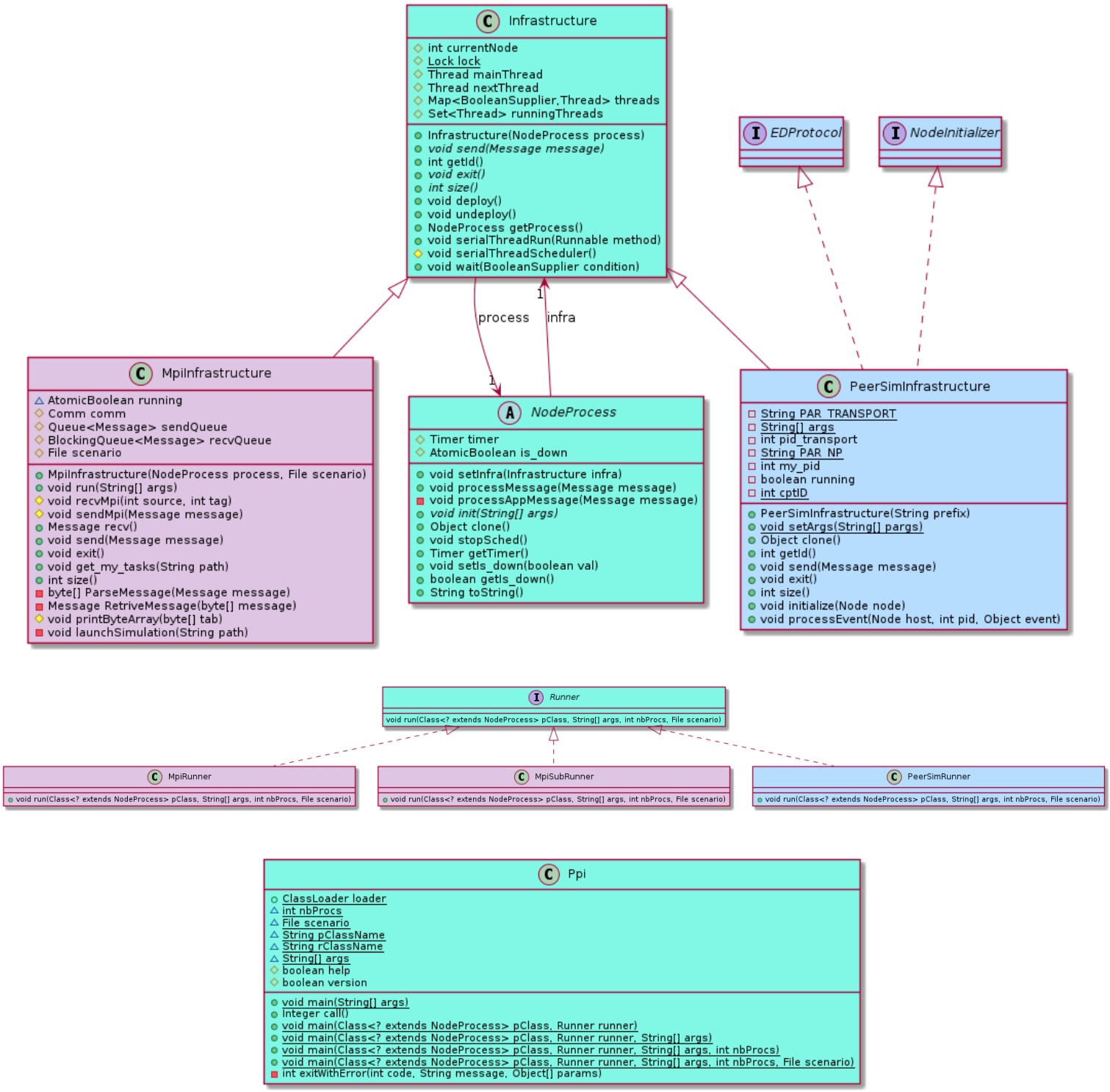
```
public class ExampleNodeProcess extends NodeProcess {

   public static class ExampleMessage extends Message{
      private static final long serialVersionUID = 1L;
      public String content;
      public ExampleMessage(int src, int dest, String content) {
         super(src, dest);
         this.content = content;
      }
   }

   @MessageHandler
   public void processExampleMessage(ExampleMessage message) {
      int host = infra.getId();
      System.out.printf(
         "%d Received '%s' from %d\n",
         host, message.content, message.getIdsrc()
      );
      if (host != 0) {
         int dest = (host + 1) % infra.size();
         infra.send(new ExampleMessage(infra.getId(), dest, "bonjour"));
      }
      infra.exit();
   }

   @Override
   public void init(String[] args) {
      if (infra.getId() == 0) {
         infra.send(new ExampleMessage(infra.getId(), 1, "bonjour"));
      }
   }
}
```

4 Les étapes de réalisation

4.1 Répartition des tâches

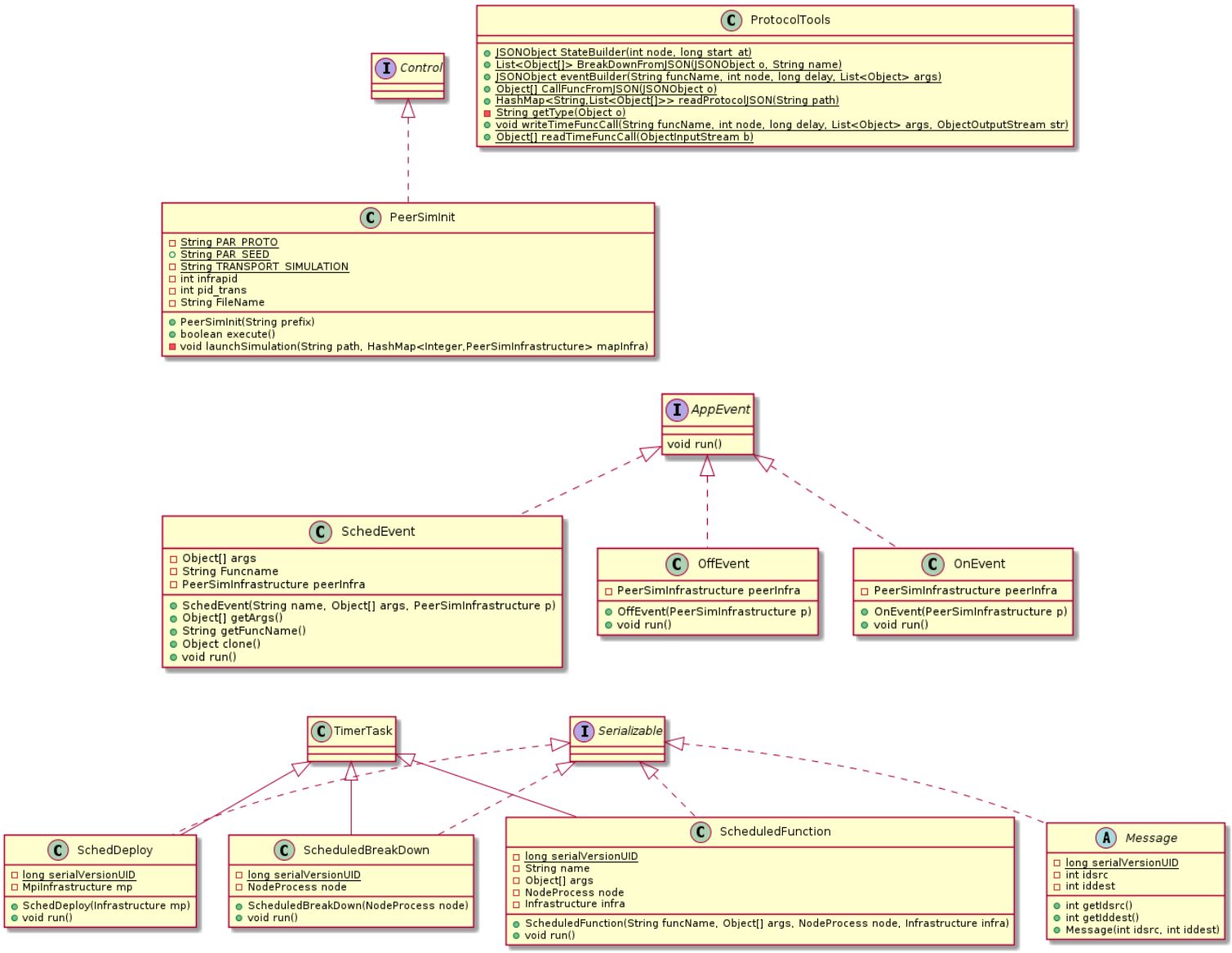
Tâche	Date d'échéance	Fait par
Se familiariser avec MPI et Peersim	17/02	Tous
Proposer une API générique	24/02	Tous
Implanter l'API pour MPI et pour PeerSim	02/03	Tous
Créer une classe abstraite Message	09/03	Tous
Unifier le lancement de l'application	09/03	Tous
Rédiger le carnet de bord	16/03	Tous
Rédiger le pre-rapport	23/03	Tous
Implanter l'aiguillage automatique des messages	23/03	Tous
Ajouter la fonctionnalité de description de scenario	30/03	Tous
Ajouter des fonctionnalités de wait/notify	20/04	Tous
Rédiger le rapport final	08/06	Tous
Soutenance finale	11/06	Tous

4.2 Définition API générique



Nous avons commencé par définir une classe abstraite Infrastructure qui contient toutes les méthodes que nos deux API (MPI et PeerSim) devront implanter et une classe abstraite NodeProcess qui est en lien direct avec Infrastructure.
Le lancement de ces deux applications est complètement différent, c'est pourquoi il est primordial que les deux lancements soient unifiés pour permettre un développement plus rapide et sûr. Pour remédier à cette situation, notre solution consiste à créer une interface Runner qui permet d'instancier les exécutions des deux API avec les mêmes arguments. Il suffit donc d'instancier MPIRunner et/ou PeerSimRunner avec en argument le nom de la classe qui hérite NodeProcess, le nom de la classe du Runner et (facultatifs) le nombre de processus ainsi qu'un scénario.

4.3 Les classes utilitaires



4.4 Définitions des primitives offertes et explication

```
public void start();
```

La primitive start doit etre redéfinie par l'utilisateur et permet de définir le code à éxécuter par le processus lors de son lancement.

```
public void processMessage(Message message);
```

De même pour processMessage qui permet de définir le comportement du processus lors de la réception d'un message.

```
abstract class Message implements Serializable
```

Message est la classe définie afin de permettre une abstraction du traitement des messages et doit etre étendue par l'utilisateur.

```
public void serialThreadRun(Runnable method);
```

Exécuter un nouveau thread qui démarrera immédiatement. Le thread courrant se met en attente jusqu'à la fin d l'exécution de ce dernier ou qu'il se met en attente avant de poursuivre son exécution.

```
protected void serialThreadScheduler();
```

Ordonnanceur qui parcourt les serialThreads en attente et les réveille s'ils peuvent continuer.

```
public void wait(BooleanSupplier condition);
```

Cette primitive wait met le processus appelant en attente jusqu'à que la condition soit vérifiée. La condition doit etre un lambda retournant un booléen.

```
public class Ppi
//utilisation
Ppi.main(new String[] {NodeProcess.class.getName(),Runner.class.getName()});
```

La class Ppi est le main de notre application, elle permet d'executer le code utilisateur en MPI ou en PeerSim.

4.5 Implantation vers MPI

Pour l'implantation de l'adapteur permettant d'utiliser MPI nous avons commencé par implanter un algorithme simple en utilisant cette dernière. Une fois que nous avons compris comment MPI fonctionne, nous avons étendu la classe Infranstructure qui représente l'abstraction des deux adapteurs implantés, pour MPI la classes MpiInfrastructure nous permet

de communiquer avec MPI en ajoutant une couche sur les primitives de MPI. Nous avons, cependant, dû ajouter une boucle afin d'adapter le fonctionnement de MPI à celui de PeerSim.

4.6 Implantation vers PeerSim

Dans le cas de PeerSim, l'implantation de notre API consistait principalement à effectuer un Adapter avec l'API déjà existante de PeerSim, et donc la part la plus importante du travail était de comprendre le fonctionnement des quelques classes de PeerSim nécessaires pour faire le lien entre les deux API. Il a aussi fallu adapter certaines de nos classes pour PeerSim, par exemple rendre notre NodeProcess clonable.

4.7 API pour un scenario

L'utilisateur aura aussi accès à une API permettant de décrire un scenario où il devra spécifier les méthodes qui devront être appeler à un temps donné ou encore la panne d'un nœud.
Si pour Peersim, elle utilise la procédure d'ajout d'événement et ainsi l'évènement est déclenché à la bonne date.
Pour Mpi nous avons utilisé la class Timer de java afin d'avoir pour chaque processus Mpi un démon qui exécutera l'évènement spécifié, les évènements peuvent cependant avoir un décalage.
En premier lieu, chaque évènement est dépendant de l'horloge du processus sur lequel il va être exécuté ou plus précisément l'heure à laquelle ce dernier va démarrer car chaque processus reçoit le fichier de spécification du scénario et y prend que les évênements qui le concernent.
Dans un second temps, la class Timer n'utilise qu'un seul Thread et par conséquent si deux évènements sont trop proches, le second évènement va attendre la fin de l'exécution du premier pour s'exécuter.

4.8 Fonction de wait dans l'infrastructure

4.9 Tests plus poussés des invariants

4.10 Test d'exclusion mutuelle

4.11 Scénario de panne

5 Prise en main de notre API

5.1 Programmer un algorithme

Pour programmer un algorithme à l'aide de Ppi il faut créer un classe qui étend la classe abstraite `NodeProcess`. Par la suite l'ensemble des méthodes disponible seront accessibles par l'intermédiaire de la propriété `infra`.

5.1.1 Etendre le classe `NodeProcess`

La classe abstraite `NodeProcess` contient une methode abstraite `init` qu'il est nécessaire de redéfinir dans votre classe :

```
public void init(String[] args)
```

Cette méthode permet d'initialiser chaque processus à l'aide des arguments qui lui sont passés en paramètre. Elle peut être utile comme point de départ d'un algorithme où bien simplement pour le paramètrer. L'étape suivante sera de créer les différentes classes de message nécessaires au bon fonctionnement de votre algorithme.

5.1.2 Créer des classes de message

Chaque classe de message doit étendre la classe `Message` et l'ensemble de son contenu doit être `Serializable`. De cette manière il sera possible de les envoyer via Ppi et de les réceptionner à l'aide de gestionnaires de messages. Il est conseillé de créer une classe par type de message applicatif et ainsi de créer un gestionnaire par type de message.

5.1.3 Définir des gestionnaires de messages

Pour définir un gestionnaire de message, rien de plus simple. Ajoutez à votre classe héritant de `NodeProcess` une fonction pernant en paramètre un objet d'une des classes de message que vous venez de définir et ajoutez-y l'annotation `@MessageHandler` :

```
@MessageHandler
public void handlerTestMessage(TestMessage msg) {}
```

Il ne doit y avoir qu'un seul gestionnaire de message par classe de message, sinon le premier trouvé sera choisi par Ppi.

5.1.4 Méthodes à dispositions

5.2 Lancer Ppi

5.3 Décrire un scénario

5.3.1 Introduction

Afin de décrire un scenario, l'utilisateur devra spécifier en premier lieu le nombre de noeuds demandés, puis préciser sur quelle API le scénario devra être exécuté, et enfin les événements et leur date qui devront être appelés.

Sachant que MPI travaille sur de vraies machine et que le temps d'arrivée des messages est inconnu, on ne promet pas l'éxécution exacte du scénario demandé.

Afin de demander l'éxecution d'un scénario, il faut aussi étendre la class NodeProcess et y implanter les fonctions qui font office de events, et dont on peut programmer l'appel avec délais données. Pour le faire, il faut un fichier au format Json avec le format suivant.

5.3.2 Exemple de scénario

```
{
   "Off": [
      {
         "node": 0,
         "start": 100
      }
   ],
   "events": [
      {
         "args": [
            { "val": "MonArgument1", "type": "String" },
            { "val": 4, "type": "Integer" }
         ],
         "FunctionName": "End",
         "Node": 1,
         "Delay": 900
      },
      {
         "args": [],
         "FunctionName": "doSomething",
         "Node": 2,
         "Delay": 300
      }
   ],
   "On": {
      "node": 0,
      "start": 10000
   }
}
```

5.3.3 Détails

Les Json object nommé "off","on" et "events" font office de conteneurs pour les différents appels demandés, chaqu'un contient une liste des événements, par exemple "off" sa liste contient un sous objet qui définit
"node" qui indique le noeud sur lequel à la date "start" le noeud arrêtera de répondre au message l'inverse "On" réactive le noeud et permet à eux deux de simuler une panne.

l'objet "events" représente l'appel d'une fonction utilisateur, l'élément "args" de cet objet représente les arguments de la fonction qui doivent être donnés dans le meme ordre que ceux de la fonction. Il faudra aussi spécifier le type de chaque arguments ainsi que sa valeur.

Notons que les types des arguments acceptés sont les types de base de java. Enfin la clé "FunctionName" représente le nom de la fonction à appeler.

Pour aider la construction du fichier de configuration nous proposons les primitives de la class ProtocolTools suivante :
Pour construire un appel de fonction

```
public static JSONObject eventBuilder(String funcName , int node, long delay,
    List<Object> args);
```

Pour construire un appel de fonction

```
public static JSONObject StateBuilder(int node, long start_at);
```

Enfin, pour lancer l'exécution du scénario, il faut juste ajouter le nombre de processus et le chemin vers le fichier Json au runneur souhaité.

5.4 Comment écrire son propre adapter pour un autre support

6 Plan de validation

6.1 Tests de validation

Pour chaque fonctionnalité ajoutée nous ajoutons un test et nous nous assurons que les tests précédents fonctionnent toujours.
La classe BasicTest est la toute première classe de tests que nous avons implémenté, elle s'assure que les fonctionnalités de base de notre API send/receive et le format de description de scénario restent fonctionnels malgré les nouveaux ajouts.

Plusieurs algorithmes distribués seront implantés et testé unitairement dont celui de l'anneau et au moins une exclusion mutuelle.

6.2 Démonstration

Lors de la soutenance, une démonstration sera lancée pour montrer de façon évidente que notre interface fonctionne. Le même code applicatif sera executé sur plusieurs machines (potentiellement virtuelles) une première fois via l'infrastructure MPI puis sur celle de Peersim.

7 Conclusion

7.1 Rappel des objectifs

7.2 Axes principaux de travail

7.3 Axes d'amélioration possibles et ouverture