

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	2
1.1	Les applications réparties : qu'est-ce ?	2
1.2	Les difficultés à programmer une application répartie	2
2	Méthode de développement	2
2.1	API réelle : avantages/inconvénients + exemple API(MPI ou autre)	2
2.2	API simulation à événements discrets : qu'est-ce ? + avantages/inconvénients + exemple(PeerSim ou autre)	2
2.3	Aperçu de l'implémentation d'un anneau avec les deux API	3
3	Motivation et objectif global	4
3.1	API générique : avantages, modèle de programmation(ici événementiel)	4
3.2	Aperçu de l'implémentation d'un anneau avec l'API générique	5
4	Les étapes de réalisation	6
4.1	Définition API générique	6
4.2	Définitions des primitives offertes et explication	7
4.3	Implantation vers MPI	8
4.4	Implantation vers PeerSim	8
4.5	API pour un scénario	8
5	Déploiement	9
5.1	Déploiement en MPI	9
5.2	Déploiement en Peersim	9
6	Comment écrire un adaptateur sur un autre API	9
7	Plan de validation	9
7.1	Tests de validation	9
7.2	Démonstration	9
8	Conclusion	9
8.1	Les principales tâches de chacun	9
8.2	Améliorations possibles	9

1 Introduction

1.1 Les applications réparties : qu'est-ce ?

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 + exemple API(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 à l'impossibilité d'avoir un contrôle sur les événements qui apparaissent, contrairement à PeerSim.

2.2 API simulation à événements discrets : qu'est-ce ? + avantages/inconvénients + 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 exécution est toujours ce qui permet d'accélérer le debug d'une application répartie avant son déploiement ou de reproduire une séquence d'événements qui a provoqué un bug sur une application déjà 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'exécution au niveau du code métier.

Pouvoir exécuter 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'exécution 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;
        private String s;
        public ExampleMessage(int src, int dest, String s) {
            super(src, dest);
            this.s = s;
        }

        public String getS() {
            return s;
        }
    }

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

    @Override
    public void start() {
        if (infra.getId() == 0) {
            infra.send(new ExampleMessage(infra.getId(), 1, "bonjour"));
        }
    }
}

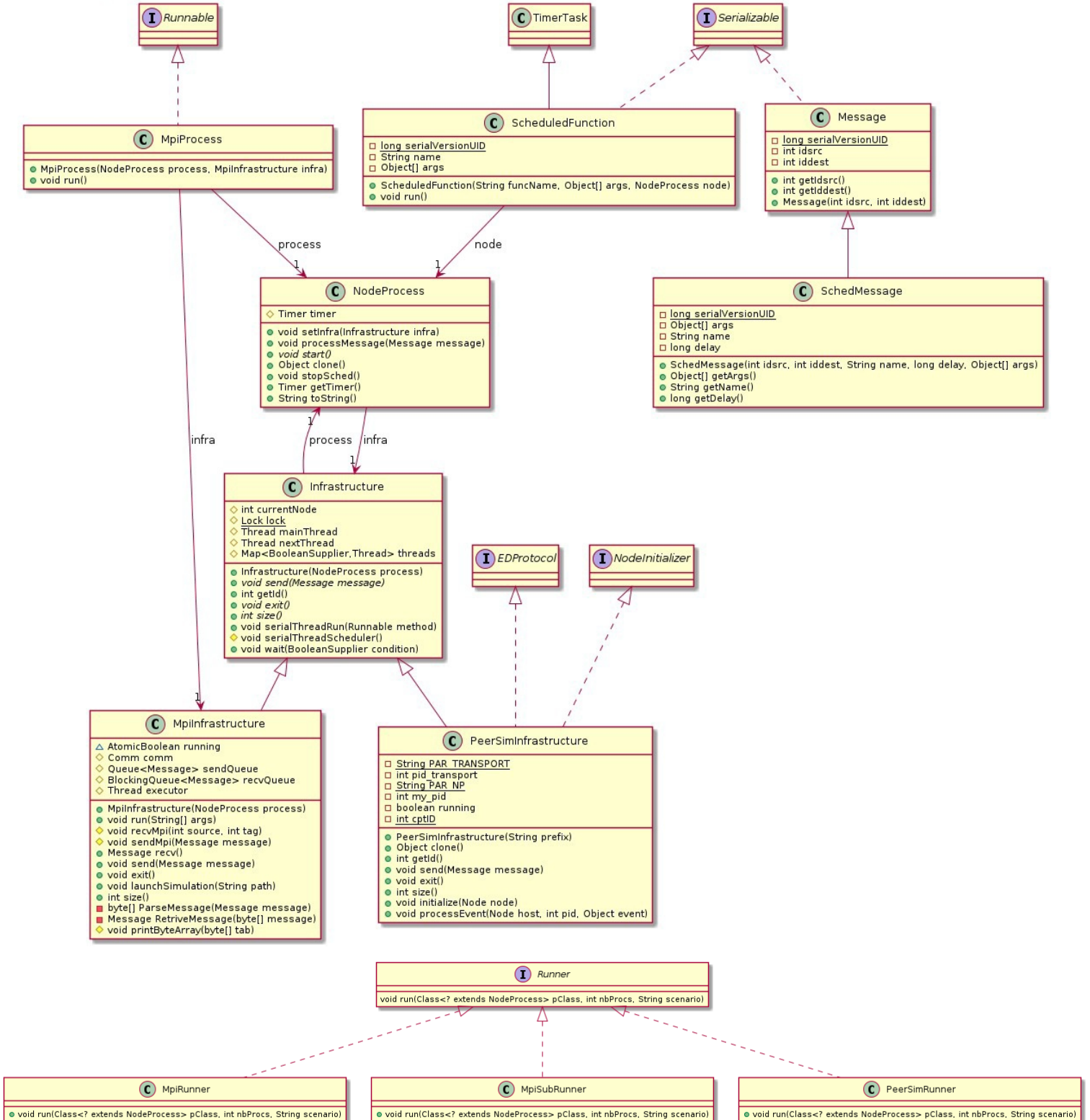
public class BasicTest {

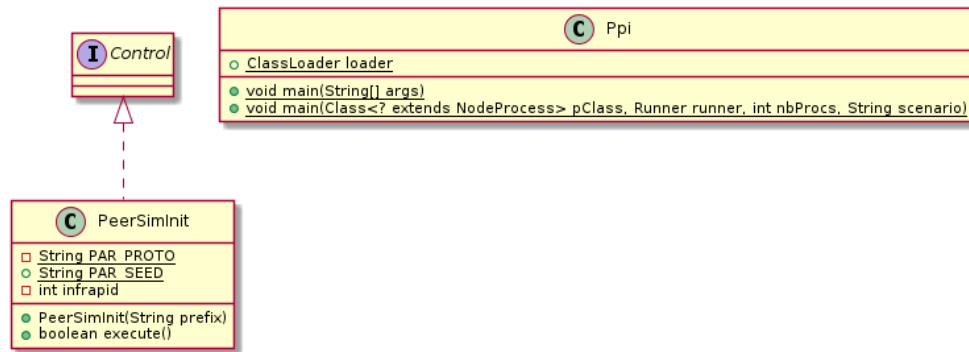
    @Test
    public void MpiExample() {
        Ppi.main(new String[] { ExampleNodeProcess.class.getName(), MpiRunner.class.getName() });
        assertTrue(true);
    }

    @Test
    public void PeersimExample() {
        Ppi.main(new String[] { ExampleNodeProcess.class.getName(), PeerSimRunner.class.getName() });
        assertTrue(true);
    }
}
```

4 Les étapes de réalisation

4.1 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.

4.2 Définitions des primitives offertes et explication

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

La primitive start doit être redéfinie par l'utilisateur et permet de définir le code à exécuter 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 être étendue par l'utilisateur.

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

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

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

Ordinnanceur 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 être 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'exécuter le code utilisateur en MPI ou en PeerSim.

4.3 Implantation vers MPI

Pour l'implantation de l'adaptateur 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 adaptateurs implantés, pour MPI la classes MpiInfranstructure 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.

4.4 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.

4.5 API pour un scenario

L'utilisateur aura aussi accès à une API permettant de décrire un scenario où il devra spécifier en premier lieu le nombre de noeuds demandé, 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'exécution exacte du scénario demandé.

5 Déploiement

5.1 Déploiement en MPI

5.2 Déploiement en Peersim

6 Comment écrire un adaptateur sur un autre API

7 Plan de validation

7.1 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.

7.2 Démonstration

Lors de la soutenance, une démonstration sera lancée pour montrer de façon évidente que notre interface fonctionne. Le même code applicatif sera exécuté sur plusieurs machines (potentiellement virtuelles) une première fois via l'infrastructure MPI puis sur celle de Peersim. pour prouver le bon fonctionnement de notre application nous avons décidé de faire jouer une note musicale aux ordinateurs où chacun devra attendre un certain délai avant de jouer sa note tout en s'assurant qu'il est le seul à jouer à ce moments pour mpi et pour PeerSim nous allons nous assurer que l'exécution est déterministe

8 Conclusion

8.1 Les principales tâches de chacun

8.2 Améliorations possibles