

PRE-RAPPORT DU PSAR

API générique pour le développement d'applications réparties

Tarik Atlaoui

Nicolas Peugnet

Kimmeng Ly

Max Eliet

09 Mars 2020



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 développées 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)

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.

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 reçu 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;
        }
    }

    @Override
    public void processMessage(Message message) {
        int host = infra.getId();
        System.out.println("" + host + " Received hello from " + message.getIdsrc());
        if (host != 0) {
            int dest = (host + 1) % infra.size();
            infra.send(new ExampleMessage(infra.getId(), dest, "hello"));
        }
        infra.exit();
    }

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

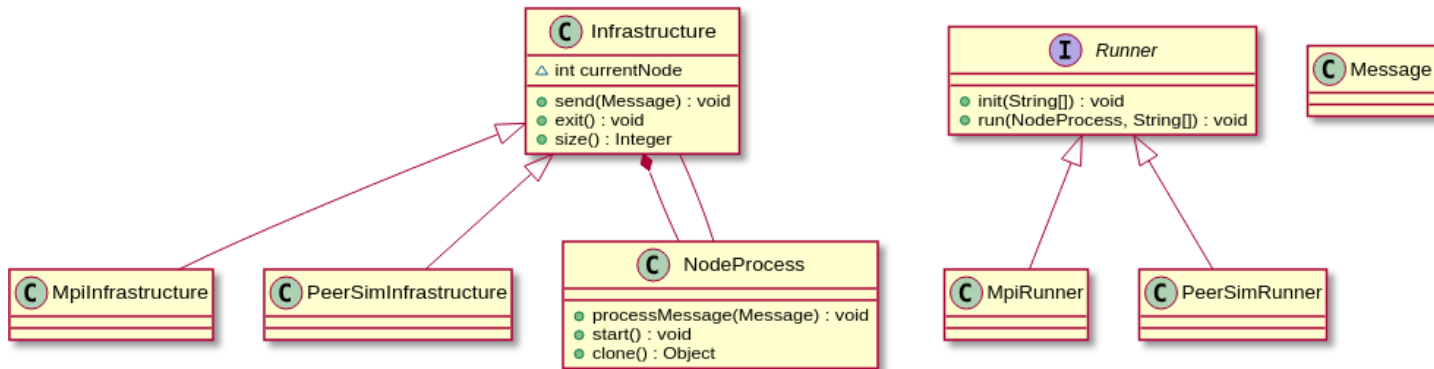
public class BasicTest {

    @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 notre réalisation par définir une class Infrastructure abstraite qui contient tous les éléments qui doivent être demandé pour les deux Api puis nous avons créé deux sous-classes (MpiInfrastructure et PeerSimInfrastructure) qui ont adapté les interactions avec les deux API.

Sachant que le lancement des deux applications est radicalement différent, nous avons réalisé qu'il était primordial que les deux lancements soient unifiés pour permettre un développement plus rapide et sûr, pour y remédier nous avons créé l'interface Runner qui nous permet d'instancier des exécutions des deux Api avec les mêmes arguments, ainsi il suffit d'instancier MpiRunner ou PeerSimRunner avec en argument le nom de la class qui implante NodeProcess, le nom du runner et le nombre de processus.

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);
```

processMessage elle aussi doit être redéfinie par l'utilisateur et permet de définir le comportement du processus lors de la réception d'un message

```
abstract class Message implements Serializable
```

Message est la class définie afin de permettre une abstraction du traitement des messages et doit être redéfinie et doit être étendue par l'utilisateur

```
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 PeerSim.

4.3 Implantation vers MPI

L'Api MPI (Message Passing Interface) est avant toute une norme pour le passage de message entre différent ordinateur ou sur le meme ordinateur. elle énormement utiliser pour la communication sur des architecture distribuer. Dans notre cas elle c'est révéler extremement interessante car MPI a été implémentée sur presque toutes les architectures de mémoires ainsi elle a était addapter pour chaque une de la façon la plus optimal possible cependant les teste et le debug sur celle-ci reste difficile du a l'imposibiliter d'avoir un controle sur les événements qui apparaisse contrairement a PeerSim.

4.4 Implantation vers PeerSim

PeerSim est utiliser pour créer des noeud et simuler une architecture pair-à-pair ont générant des protocol et des événements qui sont définis par l'utilisateur , l'exécution et la suite d'événements est toujours la meme ce qui permet d'acceler le debug d'une application répartie avant sont déploiement ou de reproduire une séquences d'événements qui a provoquer un bug sur une application déjà existante, cependant écrire du code ont PeerSim est exhaustif et peux difficilement etre exploiter pour une utilisation autre que le debug.

4.5 API pour un scenario

4.6 Rédaction du rapport

5 Plan de validation

5.1 Tests de validation

Pour chaque fonctionnaliter ajouter nous ajoutant un teste et nous nous assurons que les teste précédent fonctionne toujours.

La class BasicTest est la toute 1er classe de teste que nous avons implémenter elle s'assure que les fonctionnaliter de base de notre Api send recive et le format de description de scenarion est fonctionnel

5.2 Comment montrer que cela fonctionne bien, tests unitaires, respect du cahier des charges

6 Planning des tâches

6.1 Une deadline pour chaque tâche dans un ordre chronologique