UQAC
Université du Québec
à Chicoutimi

Projet

Hiver 2021

Simulation systèmes

(8INF802)

Maîtrise en informatique

Nicolas

Le Roux

Bastien

Anfray

Table des matières

Boucle de simulation principale :

La boucle de simulation principale est modélisée dans le fichier main.java. Dans ce fichier on se contente juste d'initialiser les différents threads représentant l'immeuble et les ascenseurs, ainsi que l'affichage en console. L'immeuble va venir lui à son tour initialiser ses différents composants sous forme de threads (personnes entrant dans l'immeuble, ses étages). Chaque thread est lancé via sa fonction run(). Une fois tous les threads lancés, ils vont pouvoir communiquer entre eux en temps réel le tout en parallèle et en autonomie. L'avantage d'utiliser des threads pour modéliser cette boucle principale comparé à plusieurs boucles for imbriqués est que chaque entité (ascenseur, personne, immeuble) peut tourner en parallèle des autres, alors que dans une boucle for il aurait fallu agir un par un sur les différentes entités. L'inconvénient est que cela est plus gourmand en ressources.

```java
public class Main {
    //fonction main du programme
    public static void main(String[] args) throws InterruptedException {
        Immeuble immeuble = new Immeuble(); //on crée un immeuble
        ArrayList<Ascenseur> ascenseurs = immeuble.getAscenseurs(); //on récupère les ascenseurs
        //pour chaque ascenseur, on crée un thread, on le lance et on le renomme
        for (Ascenseur ascenseur : ascenseurs) {
            Thread ascensceurThread = new Thread(ascenseur);
            ascensceurThread.start();
            ascensceurThread.setName("Ascenseur "+ascenseur);
        }

        //on crée un Thread pour l'immeuble et on le lance et le renomme
        Thread immeubleThread = new Thread(immeuble);
        immeubleThread.start();
        immeubleThread.setName("Immeuble");

        //on crée un thread pour l'affichage dans la console, on le lance et on le renomme
        DisplayConsole displayConsole = new DisplayConsole();
        displayConsole.init();
        displayConsole.start();
        displayConsole.setName("DisplayConsole");

        //on lance la simulation pendant 10 minutes
        LocalTime time = LocalTime.now();
        while(Duration.between(time, LocalTime.now()).toMinutes() != 10){ } //durée d'éxécution en minutes
        System.out.println("\n\n----- Fin du programme -----\n\n");
        System.exit(0);
    }
}
```

Structure de données :

Afin de modéliser les différentes entités de la simulation nous avons créé plusieurs classes :

- Une classe Immeuble qui a pour attribut le nombre d'ascenseurs, le nombre d'étages, un coefficient d'accélération, une liste d'ascenseurs, une liste de personnes, une liste pour calculer la performance de la simulation, et une instance de cet immeuble qui sera passé dans d'autres classes. Sa méthode run vient générer une valeur avec la loi de poisson et suivant cette valeur on vient ajouter un certain nombre de personnes dans l'immeuble.

```
public void run() {
    //toutes les minutes on :
    while(true){
        // TODO Auto-generated method stub
        LoiDePoisson ldp = new LoiDePoisson(0.5);//génère une valeur avec la loi d
        System.out.println("Nombre de nouvelles personnes : " + ldp.getNombre());
        for(int i = 0; i<ldp.getNombre(); i++){ //pour chaque i de 0 à la valeur g
            Personne personne = new Personne(); //on crée une personne
            Thread personneThread = new Thread(personne); //on crée thread de cett
            personneThread.start(); //on lance le thread de la personne
            personneThread.setName(personne.toString()); //on change le nom du thr
            lPersonnes.add(personne); //on ajoute la personne à la liste des perso

        }
        try {
            Thread.sleep(60000/acceleration); //on attend une minute
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

- Une classe Personne avec comme attributs son étage actuel, étage d'arrivée, son prénom, un fichier servant à définir son prénom, le temps local lors de l'appel de l'ascenseur et le temps local lors de l'arrivée à destination, une variable servant à indiquer si la personne est dans un ascenseur ou non. Sa méthode run vient choisir un ascenseur, appeler cet ascenseur, monter dans l'ascenseur, définir son temps de travail grâce à la loi exponentielle et rappeler l'ascenseur quand elle a fini de travailler.

```java
public void run(){
    //la personne va se comporter de cette façon :
    try {
        System.out.println(prenom+" arrive : Etage de départ : " + etage_actuel + "  Etage d'arrivée :
        Ascenseur ascenseur = choixAscenseur();//choisi un ascenseur
        demandeAscenseur(ascenseur); //demande l'ascenseur choisi
        dansAscenseur(ascenseur); //monte dans l'ascenseur choisi

        //la personne va attendre un temps aléatoire généré via une loi exponentielle
        LoiExp loiexp = new LoiExp(0.016666666666666);
        System.out.println(this + " va travailler pendant " + (int)loiexp.getNombre() + " minutes");
        int acceleration = Immeuble.getInstance().getAcceleration();
        Thread.sleep((long) (loiexp.getNombre()*60000/acceleration));

        //quand la personne a fini de travailler, elle veut revenir à l'étage 0
        etage_arrivee = 0;
        ascenseur = choixAscenseur(); //choisi un ascenseur
        demandeAscenseur(ascenseur); //demande l'ascenseur choisi
        dansAscenseur(ascenseur); //monte dans l'ascenseur choisi
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

- Une classe Ascenseur avec comme attributs un identifiant, une vitesse, son algorithme d'ordonnancement, une variable indiquant si les portes sont ouvertes, une variable indiquant si l'ascenseur est en mouvement, un attribut Idle venant définir le comportement de l'ascenseur au repos. le numéro de l'étage actuel, une liste d'étages qui définit le trajet de l'ascenseur, une liste de personnes, et une liste de personnes qui attendent cet ascenseur. Sa fonction run() vient faire déplacer l'ascenseur suivant les étages dans la liste d'étages suivant et son ordonnancement.

```java
public void run(){
    try {
        System.out.println("Ascenseur run()");
        while(true){
            if(etages_suivant.size() != 0){ //si la liste des étages suivants n'est pas vide
                move(getEtageSuivant(ordonnancement));//on se déplace à l'étage suivant en fo
                if(etages_suivant.size() == 0){ //si la liste redevient vide ensuite,
                    idle();//l'ascenseur est au ralenti
                }
            }
        }
    } catch (InterruptedException e) { //gestion des exceptions
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

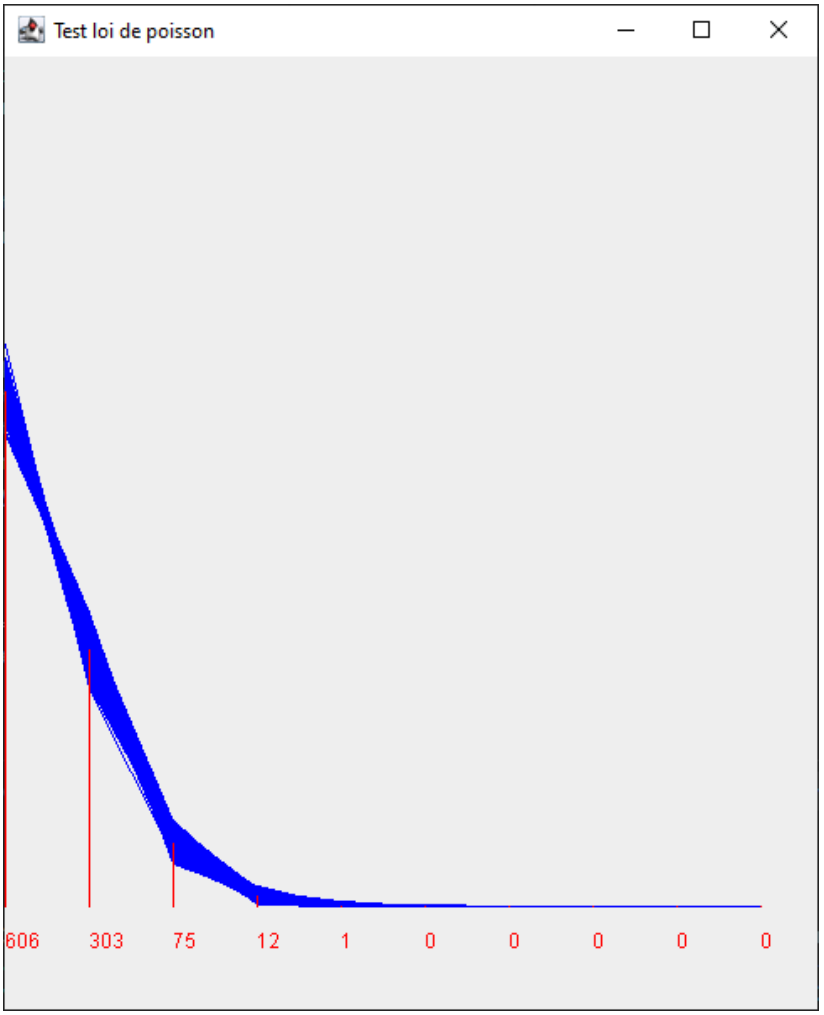
Loi de poisson et exponentielle :

Afin de vérifier que ces lois soient correctes, nous avons réalisé un test pour chacune d'entre elles.

1. Le test de loi de poisson (fichiers TestLoiPoisson.java et TestLoiPoissonDisplay.java) va faire 1000 fois l'expérience de générer 1000 nombres avec la loi de poisson. Ici on a pris un lambda égal à 0.5.

   Ce test affiche un graphique contenant plusieurs éléments :

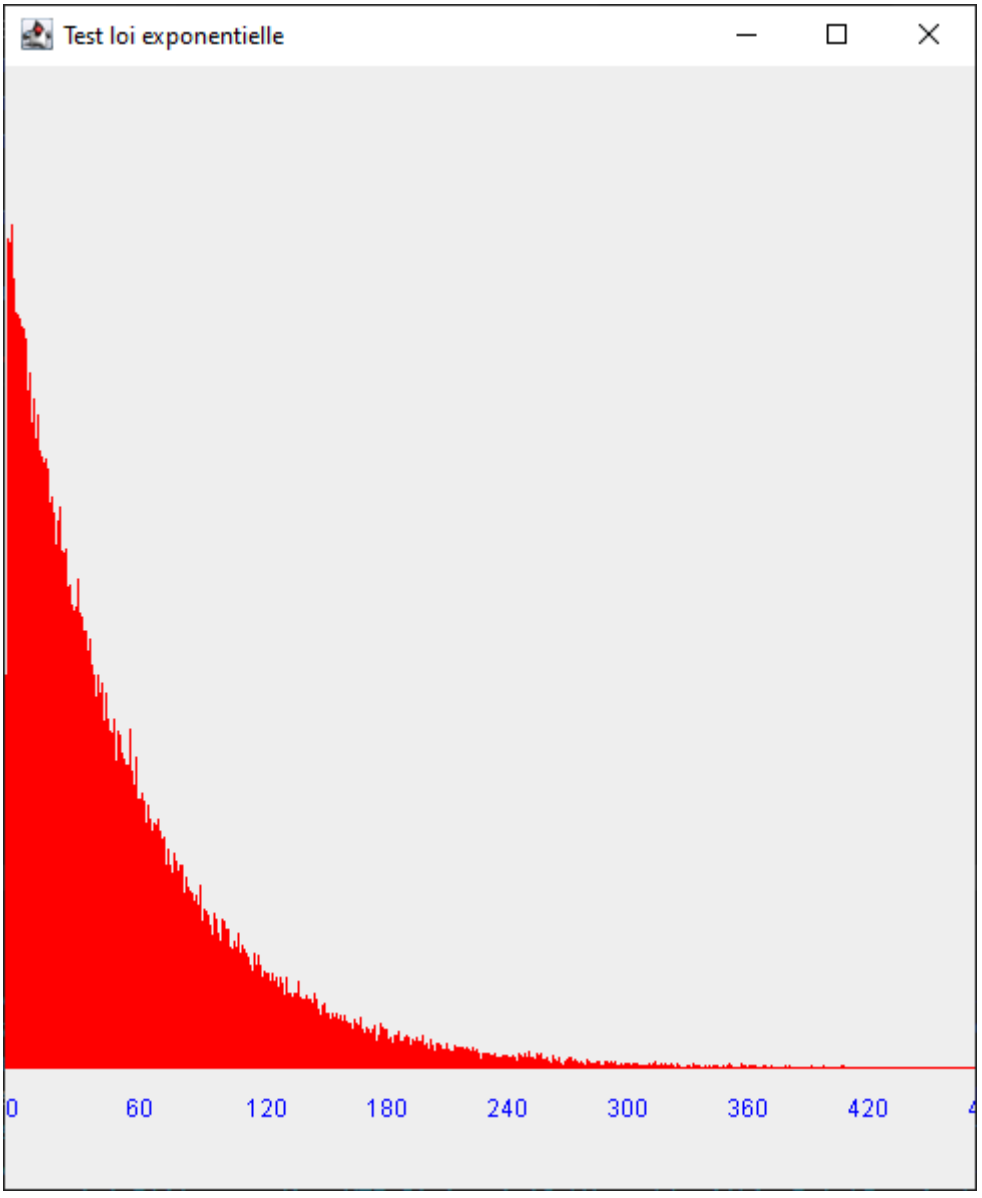
   - Les lignes bleus représente le nombre de 0, 1, 2, 3, …, 9 générés à chaque experiences
   - Les lignes verticales rouges représentent la moyenne du nombre de 0, 1, 2, 3,..., 9 générés



Test de la loi de poisson

2. Le test de loi exponentielle (fichiers TestLoiExp.java et TestLoiExpDisplay.java) va générer 100 fois 1000 nombres avec la loi exponentielle. On a prit ici un lambda égal à 1/60

   Ce test affiche un graphique affichant les proportions du nombre de fois que les nombres ont été générés. On remarque ici que le graphique a bien une forme exponentielle avec une moyenne de 60.



Test de la loi exponentielle

Mesures de performances :

Afin de mesurer la performance de chacun des algorithmes nous utilisons le procédé suivant : Dans la classe Immeuble nous avons une méthode calculPerformance(), cette fonction se contente de faire la moyenne de toutes les performances obtenues pour chaque personne de la simulation. Cette performance est calculée en faisant la différence entre l'attribut t_demandeAscenseur et t_arriveeEtage de la classe Personne. Ainsi on récupère le temps local de la machine en secondes entre le moment où une personne appuie sur le bouton de l'ascenseur et le temps local de la machine au moment où elle arrive à sa destination. Au début de la simulation la valeur ne va pas tout de suite être en état stable, elle sera en "warm-up", après une certaine durée on pourra constater qu'elle va se stabiliser.

Ici toutes nos simulations ont été réalisées pendant 10 minutes à une vitesse accélérée par 50. L'avantage est de pouvoir réaliser beaucoup de tests en peu de temps, et est assez fiable pour comparer différentes situations entre elles. Mais l'inconvénient est que la mesure de performance n'est pas exacte à cette vitesse, elle est plus faible que si on faisait une simulation à vitesse réelle. Ici on s'en sert uniquement afin de comparer toutes les différences entre les situations.

Algorithmes d'ordonnancement

Dans notre scénario à un seul ascenseur nous avons implémenté le First Come First Serve et le Shortest Seek Time First. Dans la classe Ascenseur l'algorithme est déterminé depuis la variable ordonnancement. Sa valeur est un enum de soit FCFS ou SSTF.

FCFS:

FCFS consiste à faire sortir en premier la personne qui a appuyé en premier sur le bouton de l'ascenseur. Les portes ne s'ouvrent pas tant que cette personne n'est pas arrivée à sa destination. Elles ne s'ouvrent pas non plus pour faire descendre des gens sur le chemin. Sur cet algorithme nous avons trouvé une mesure de performance de 84

SSTF:

SSTF consiste à choisir la direction la plus proche de la position actuelle de l'ascenseur. Au moment de choisir sa direction, l'ascenseur va ainsi parcourir sa liste etages_suivant et choisir l'étage le plus proche de la position actuelle. Pour cet algorithme nous avons trouvé une mesure de performance de 56.

Il faut préciser que les tests sur ces algorithmes ont été effectués avec l'idle STAY.

Suite à ces tests nous avons retenu l'algorithme SSTF pour la suite de projet de part ses performances supérieures à FCFS.

Politiques de ralenti :

Nous avons développé 2 politiques de ralentis afin de chercher laquelle est la meilleure : STAY et INF. Afin de tester ces différentes politiques, nous avons réalisé notre simulation pendant 10 minutes en vitesse accélérée par 50, et en utilisant l'ordonnancement SSTF.

STAY :

La politique STAY consiste à laisser l'ascenseur à son étage actuel lorsqu'il n'a plus d'étage auquel il doit se déplacer. Nous avons obtenu une mesure de performance égale à 56

INF :

La politique INF consiste à faire descendre l'ascenseur d'un étage lorsqu'il n'a plus d'étage auquel il doit se déplacer. Nous avons obtenu une mesure de performance égale à 61

Nous pouvons remarquer que la politique d'idle la plus efficace est celle du STAY devant la politique INF. C'est donc la STAY que nous avons retenue pour la suite.

Implémentation de plusieurs ascenseurs

Chaque ascenseur étant géré par un thread à part, il n'a pas été très difficile d'implémenter un système composé de plusieurs ascenseurs.
Le nombre d'ascenseurs peut être changé dans la classe Immeuble, l'affichage et le choix des ascenseurs par les personnes vont s'adapter.

Nous avons donc fait une simulation dans les mêmes conditions que précédemment, en utilisant l'ordonnancement SSTF et la politique de ralenti STAY.

- Pour N=1 ascenseur, nous avons obtenu une mesure de performance de 56
- Pour N=2 ascenseurs, nous avons obtenu une mesure de performance de 40
- Pour N=3 ascenseurs, nous avons obtenu une mesure de performance de 31

Nous remarquons ici qu'avec 3 ascenseurs, les personnes mettent moins de temps pour arriver à leur étage d'arrivée que lorsqu'il n'y a qu'un seul ou deux ascenseurs.
La différence entre 1 et 2 ascenseurs est assez flagrante, de même entre 2 et 3. Dans le cadre d'une prise de décision sur l'implémentation d'un ascenseur au sein de l'UQAC, cette simulation nous donne un aperçu de l'utilité de ce 3ème ascenseur. A priori ce 3ème ascenseur pourrait apporter un

gain de performance notable comparé aux deux ascenseurs. Cependant étant donné que nous jouons avec seulement un sous ensemble assez restreint des paramètres possibles pour un ascenseur (il manque ici par exemple la capacité maximale des étages et des ascenseurs, la vitesse des ascenseurs etc…) nous ne pouvons pas assurer que le résultat obtenu est proche de celui envisageable dans la vie réelle mais il en donne déjà une bonne indication .

Vidéo :

Dans cette vidéo vous verrez l'exécution de la simulation en console avec les paramètres suivants : Algorithme SSTF, Idle : STAY, 3 ascenseurs, 7 étages, accélération = 20, Loi de poisson avec lambda= 0.5, Loi exponentielle avec lambda = 1/60.

La première colonne sert d'indicatif sur le numéro des étages, la deuxième sert de compteur pour le nombre de personnes dans ces étages, ensuite les ascenseurs sont représentés par un chiffre, ce chiffre indique le nombre de personnes que l'ascenseur contient.

https://drive.google.com/file/d/1fUGaF7JQCbaaOzSCn8wWyBnD1ulkKlA3/view?usp=sharing

Manuel d'utilisation :

Pour lancer le programme, il suffit d'ouvrir un IDE qui supporte le langage Java et de run le main présent dans la classe Main.java.
Plusieurs paramètres sont modifiables :

Paramètre	Variable	Fichier
Nombre d'ascenseurs	nbAscenseurs	Immeuble.java
Nombre d'étages	nbEtages	Immeuble.java
coefficient d'accélération de la simulation	acceleration	Immeuble.java
vitesse des ascenseurs	vitesseAscenseur	Ascenseur.java
ordonnancement	ordonnancement	Ascenseur.java
idle	idle	Ascenseur.java

Si vous voulez exécuter les tests de loi de poisson et de loi exponentielle, vous pouvez lancer les main de TestLoiPoissonDisplay et TestLoiExpDisplay.