POLYTECH
NICE-SOPHIA

Université Nice SOPHIA ANTIPOLIS

Rapport de projet SI4:

Framework de simulation de capteurs

Romain Alexandre, Cécile Camillieri, Fabien Foerster, Jérôme Rancati

Encadrant : Sébastien Mosser

Projet de fin de semestre SI4 2014

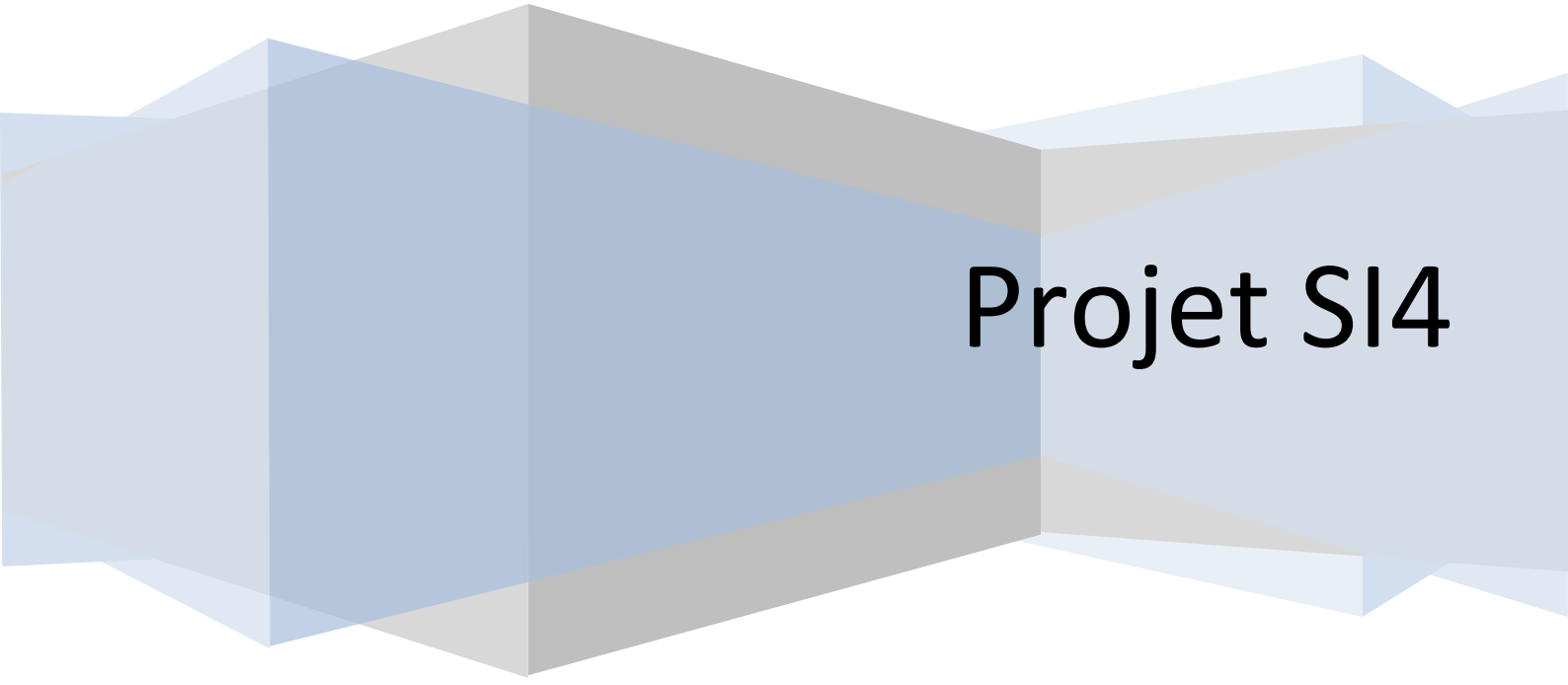

Table des matières :

I. Présentation générale

1. Smart Campus

Le projet Smart Campus étudie les possibilités d'équipement du campus SophiaTech en capteurs, ainsi que les différents services pouvant se construire sur ces capteurs. L'objectif est de fournir une plate-forme "*open data*", offrant des données accessibles à tous, chacun pouvant s'il le souhaite fournir de nouveaux services utilisant ces données.

L'équipe est composée de quatre SI5 en projet de fin d'étude (Cyril Cecchinel, Thomas Di'Meco, Matthieu Jimenez et Laura Martellotto), d'un étudiant de DUT informatique (Jean Oudot), et de deux enseignants chercheurs (Michel Riveill et Sébastien Mosser).

Les élèves de SI5 ont notamment travaillé sur l'architecture globale de SmartCampus, et défini plusieurs scénarios d'utilisation possibles :

- Capteurs de présence dans un parking. On peut alors imaginer la création d'une application mobile "Où me garer?" utilisable par les usagers du campus.
- Température et occupation des salles de l'école, ce qui peut permettre de trouver des salles libres pour travailler.
- File d'attente au restaurant universitaire. Cela permettrait de savoir à quelle heure y aller en minimisant le temps d'attendre

2. Notre projet au sein de Smart Campus

a. Problématique

Au sein d'un projet de cette envergure, le développement est très coûteux en temps et en argent. Ainsi dans le cadre de SmartCampus, il n'y a pour l'instant pas les budgets accordés permettant la mise en œuvre de capteurs en grand nombre.

C'est pour répondre au besoin de tester la viabilité et les fonctionnalités du projet, que nous intervenons. Nous nous proposons de fournir un *framework* permettant de mettre en place des capteurs simulés. Ceux-ci envoient des données simulées au *middleware* de collecte de données (intergiciel) créé par les SI5, via l'API qu'ils ont définie. L'architecture de SmartCampus est présentée dans la Figure 1 ci-dessous.

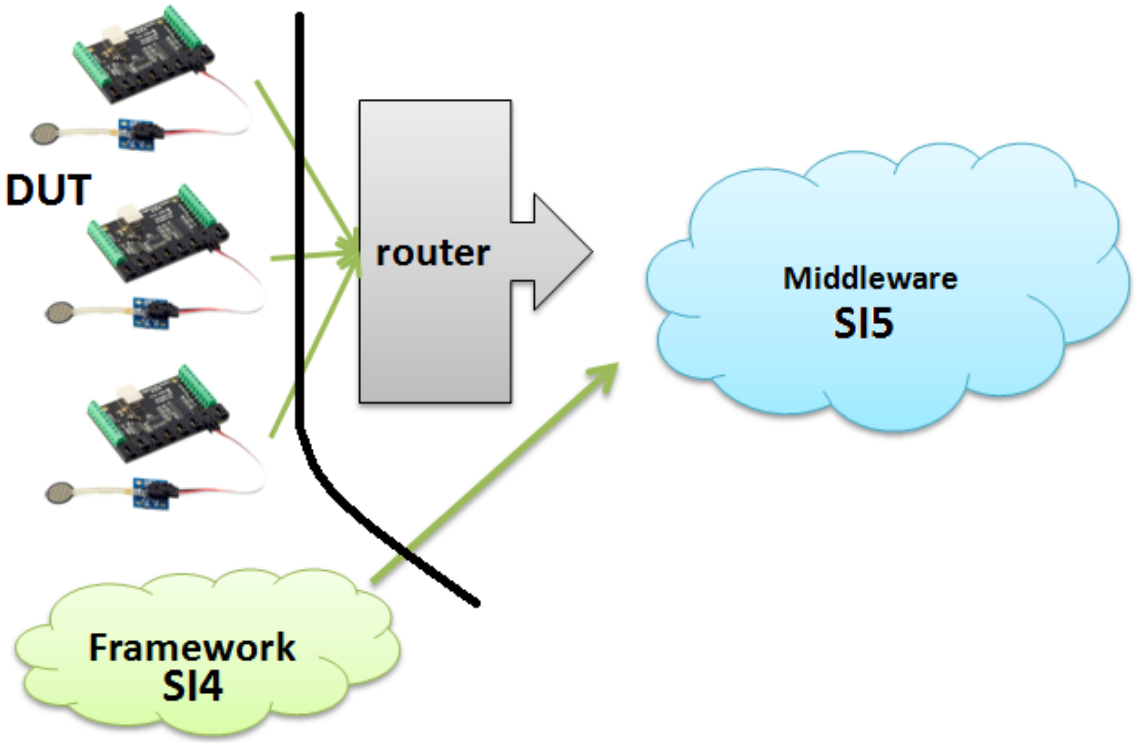

Figure 1 : Notre application doit pouvoir remplacer les capteurs physiques dans SmartCampus.

b. Solution proposée

Nous avons pu diviser la problématique en deux principaux défis :

- Permettre le test du *middleware* pour son passage à l'échelle. Pour cela il fallait notamment pouvoir simuler de nombreux capteurs agissant de façon concurrente.
- Fournir des données simulées cohérentes et donc utilisables par exemple dans des applications : "Où me garer?". Pour cela nous avons défini deux possibilités : mettre en œuvre des lois mathématiques de simulation, ou rejouer des données préexistantes.

Notre *framework* propose donc deux types de simulations :

- Le rejeu de données existantes obtenues depuis un fichier
- La simulation de données qui pourraient être fournies par des capteurs, à l'aide de la définition et de l'utilisation de loi mathématiques fournissant un modèle prédictif.

Ces simulations peuvent être effectuées d'une part en temps réel (afin notamment de tester le fonctionnement de la réception de données du middleware) ou en temps virtuel. Ce type de simulation permet par exemple de générer plusieurs mois de données, en un temps rapide. Ces données pourront ensuite être directement introduites dans la base de données du *middleware*.

Dans le but d'accroître l'utilisabilité de notre *framework*, nous avons décidé de ne pas nous limiter à la simulation de capteurs liée au projet SmatCampus. En conséquence, notre application se devait d'être la plus générique possible tout en gardant en vue l'objectif de réalisation des scénarios de SmartCampus.
Nous avons découpé notre projet en trois packages principaux :

- Conception du *framework* générique permettant de créer des simulateurs, et mise en place du moteur permettant d'effectuer des simulations via une api fluide (voir II.2.b).
- Création d'une bibliothèque standard qui met à disposition des utilisateurs du logiciel, des briques réutilisables facilitant la création de nouvelles simulations.
- Implémentation des scénarios SmartCampus proposé par les SI5

II. Contributions

1. Définition du simulateur

Notre principale problématique a été de permettre la génération de données cohérentes, et donc exploitables. Nous avons mis en place deux modes de simulation principaux (Figure 2) : l'un utilisant des modèles prédictifs définis par des lois mathématiques, l'autre permettant de rejouer les données présentes dans un fichier existant.

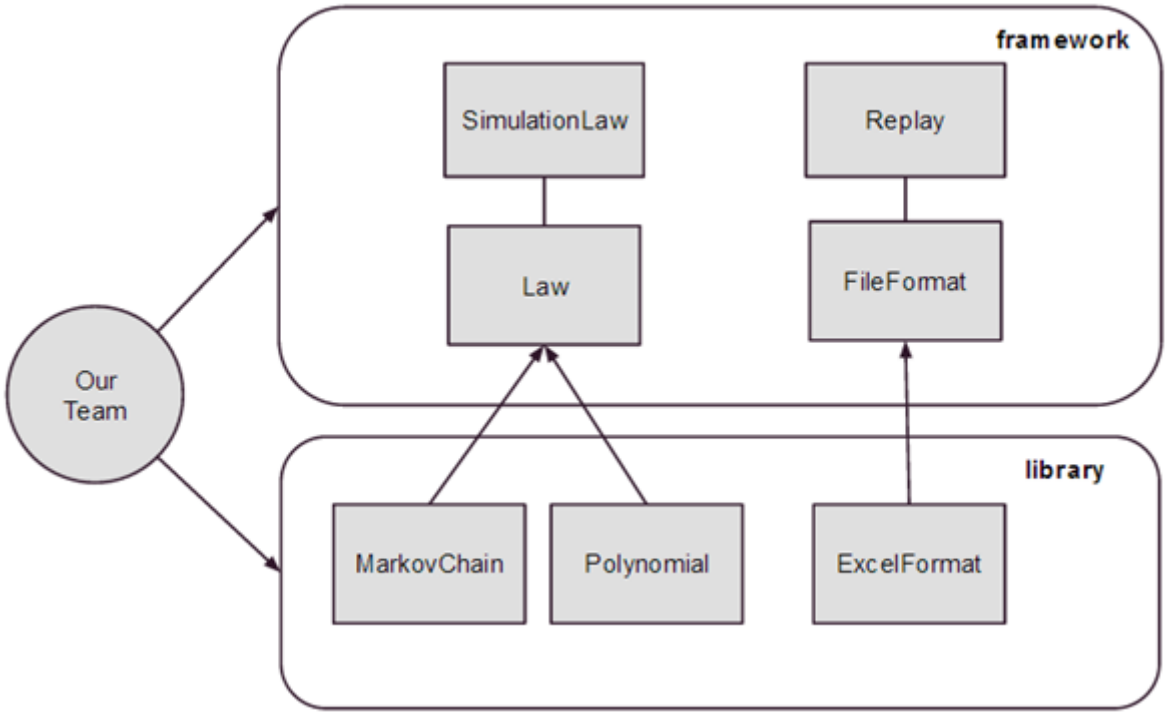

Figure 2 : Architecture des classes utiles à la définition d'une simulation

a. Utilisation d'un modèle prédictif

i. Présentation de l'architecture

Le mode de simulation de données permet de mettre en place des lois mathématiques afin de générer des données. Certaines de ces lois sont présentes dans la bibliothèque standard (chaîne de Markov et polynôme, comme indiqué dans la figure 2 ci-dessus), il est également possible d'en redéfinir (voir partie III).

Dans le cas du parking par exemple, nous devons pouvoir définir plusieurs parkings, contenant un certain nombre de places. Par conséquent, nous avons mis en place une classe pouvant représenter notre parking. Cette classe générique nommée *SimulationLaw*, contrôle un ensemble de capteurs et dirige les simulations.

ii. Gestion des lois mathématiques

Afin de générer des données, nous avons tout d'abord tracé des courbes représentatives de l'occupation du parking de l'école (Figure 3). Nous avons ensuite interpolé cette courbe afin d'obtenir un polynôme pouvant être mis en place dans notre programme.

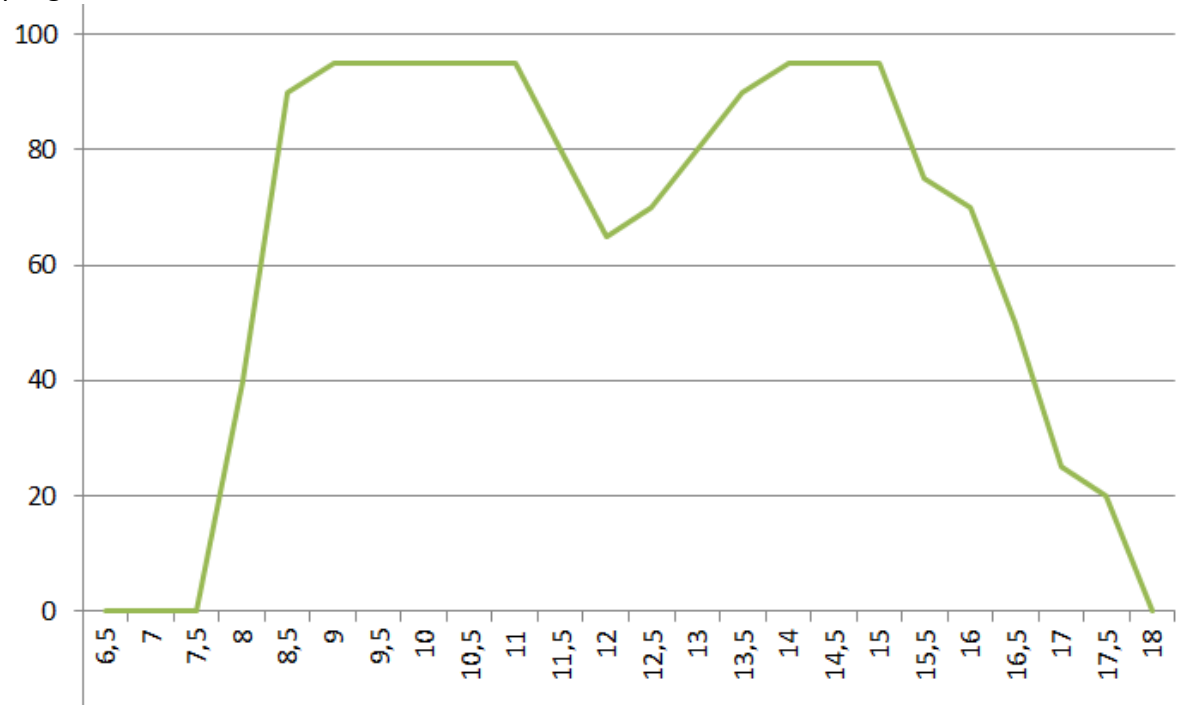

Figure 3 : Courbe représentant l'occupation du parking de polytech en fonction de l'heure de la journée.

Ce polynôme est représenté par une loi mathématique (classe Law), qui sera utilisée pour simuler les données voulues. Une loi est associée à chaque SimulationLaw. Celle-ci évalue la loi et fait ensuite passer son résultat à ses capteurs. Ensuite chaque capteur va pouvoir effectuer une transformation de cette valeur avant de l'envoyer sur le serveur. La classe PolynomialLaw est fournie dans la bibliothèque standard ce qui permet à un développeur de la réutiliser pour représenter son propre polynôme.

Bien que le polynôme permette de modéliser notre parking, cette loi ne semblait pas assez précise. Nous avons donc étudié des documents de recherche, et nous sommes arrêtés sur l'un d'eux, qui proposait de modéliser l'occupation d'un parking grâce à une chaîne de Markov (Figure 4). Nous avons donc implémenté une nouvelle *Law* pour la bibliothèque standard, décrivant une chaîne de Markov.

$$\begin{pmatrix} -\lambda & \lambda & & & & \\ \mu & -(\lambda+\mu) & \lambda & & & \\ & 2\mu & -(\lambda+2\mu) & \lambda & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ & & (m-1)\mu & -(\lambda+(m-1)\mu) & \lambda & \\ & & & m\mu & -m\mu & \end{pmatrix}$$

Figure 4 : Parking : matrice de transition [2].

Intrinsèquement, une chaîne de Markov nécessite de connaître son état courant afin de calculer l'état suivant. Nous avions donc besoin que la *SimulationLaw* connaisse les valeurs générées par ses capteurs. Nous avons donc mis en place une communication bidirectionnelle entre les capteurs et leur *SimulationLaw*. Les capteurs envoient à leur loi de simulation les données qu'ils génèrent. Ainsi la loi peut se servir des données précédentes pour effectuer son nouveau calcul.

Ce système nous a permis également de mettre en place un système d'agrégation de données. Comme la loi de simulation récupère les données de chaque capteur, elle peut ainsi calculer et envoyer au *middleware* des chiffres telles que valeur moyenne, minimale ou maximale.

iii. Gestion du temps de la simulation

C'est la *SimulationLaw* qui gère la vitesse d'exécution de la simulation à partir de la fréquence d'envoi définie au lancement de la simulation. Selon cette fréquence, elle lance les calculs de la loi et envoie les résultats à ses capteurs.

De plus nous avons mis en place deux modes de gestion du temps de simulation. Dans tous les cas, nous avons besoin d'une durée de simulation (par exemple deux minutes), et d'une période d'envoi des données (par exemple une seconde) :

- En temps réel, la simulation durera deux minutes effectives, et une donnée sera envoyée chaque seconde par tous les capteurs. De plus les données simulées auront des temps associés identiques au temps réels.
- En temps différé, seuls les temps associés au valeur générées correspondront à la durée et à la fréquence donnée. Le temps réel d'exécution sera bien inférieur puisque les données sont générées en flux continue. Dans ce mode, les données sont enregistrées dans un fichier afin de ne pas surcharger le serveur. Elles pourront ensuite éventuellement être introduites directement dans la base de données du middleware.

b. Jeu de données préenregistrées

Notre solution basée sur l'utilisation de lois mathématiques pose l'hypothèse que la personne souhaitant utiliser le *framework* connaisse une formule mathématique pouvant modéliser la situation qu'il souhaite simuler. C'est notamment pour donner une autre alternative aux utilisateurs que nous avons décidé de mettre en place un mode de rejeu de données.

Le mode replay permet de rejouer des données préenregistrées dans un fichier. Pour ce faire, nous utilisons une classe *Replay* similaire à la classe *SimulationLaw*, les deux héritent de la même classe mère. Le fonctionnement global est donc similaire. A l'instar de la Law utilisée dans *SimulationLaw*, la classe *Replay* utilise une classe *FileFormat*, correspondant à un format de fichier particulier.

En dérivant de la classe *FileFormat*, l'utilisateur pourra définir de nouveaux formats de fichiers qui pourront être utilisés afin de rejouer les données de n'importe quel fichier. Nous avons notamment livré dans la bibliothèque standard un format Excel.

Ainsi, l'utilisateur doit uniquement préciser quelles colonnes correspondent à l'heure de relevés des données, et celles qui correspondent aux données qu'il souhaite rejouer.

Le fait de pouvoir rejouer des données au format Excel nous semblait une fonctionnalité clé du mode *Replay*. En effet depuis le mouvement OpenData [5], il existe de nombreux fichiers disponibles au format Excel et de nombreux autres formats peuvent être également ramenés à ce format de fichier.

Afin de tester ce mode de fonctionnement dans des conditions réelles nous avons récupéré un fichier contenant des données de Suez Environnement via Fréderic Precioso. Le document faisant plusieurs centaines de milliers de lignes, nous avons donc dû prendre en compte le cas de fichiers Excel de taille importante.

Pour ce faire nous avons utilisé l'Event API de Apache Poi qui a l'avantage de ne pas charger l'intégralité du fichier dans la RAM, nous permettant ainsi de traiter les très gros fichiers Excel.

Comme dans le cas de la *SimulationLaw*, le mode *Replay* peut tourner en temps réel ou en temps différé.

2. Création de simulations

Nous avons vu précédemment comment a été conçue l'architecture du simulateur avec la génération de données via des lois mathématiques ou grâce à l'utilisation de données existant. Cependant, certains problèmes n'ont pas encore été résolus:

- Comment est gérée la concurrence entre les capteurs ?
- Comment peut-on lancer la simulation ?

a. Architecture de capteurs virtuels

i. Présentation d'Akka

L'un des principaux objectif de notre projet étant de tester la réception de données du *Middleware* SmartCampus, nous avons dû trouver une solution nous permettant de générer de nombreuses requêtes de manière concurrente.

Nous avions plusieurs possibilités. En premier lieu les threads. Cependant leur utilisation est lourde et une application les utilisant est difficile à maintenir. Dans un second temps, nous avons pensé à utiliser des processus qui sont plus léger. Toutefois, les processus sont limités à un système d'exploitation ce qui aurait été un frein à la portabilité de notre *framework*. Nous avons finalement choisi d'utiliser le *framework* Akka [1]. En effet, celui-ci à l'avantage d'être flexible et facile d'utilisation, une fois ses concepts assimilés.

Ce *framework* est basé autour d'un système d'acteurs hiérarchisés selon une structure arborescente (un acteur est supervisé par un père, et peut superviser d'autres acteurs qui seront ses fils). Ces acteurs communiquent entre eux par le biais de messages immuables. Ces derniers nous permettent notamment d'affecter les paramètres que nous utilisons au cours de l'exécution du programme. Ainsi Akka nous évite en grande partie d'avoir à gérer la concurrence, il nous "suffit" de penser à l'architecture et à l'organisation de nos acteurs et à la manière dont les messages vont passer entre les acteurs. Le déploiement et la synchronisation est déléguée à Akka.

Akka est capable selon ses développeurs de gérer 2,7 millions d'acteurs par Go de RAM (mémoire vive). Cela semblait donc plus que suffisant pour nous, en supposant que nous simulions chaque capteur par un acteur Akka.

ii. Utilisation d'Akka dans notre framework

Ainsi dans l'architecture décrite précédemment, les lois de simulation et les capteurs sont des acteurs qui communiquent entre eux en support aux simulations.

Puisqu'à Polytech par exemple, il y a 5 parkings séparés, il fallait pouvoir gérer plusieurs parkings indépendants dans une même simulation. Ainsi nous avons mis en place un acteur chargé de superviser nos *SimulationLaw* : le *SimulationController.* (figures 5 et 6)

De plus, dans le cas d'envoi données au serveur, afin que l'envoi (connexion, *etc.*) ne ralentisse pas la simulation, nous avons mis en place un nouvel acteur *DataSender*. Chaque capteur crée son propre *DataSender*, et lui donne les données à envoyer, afin que le temps que prend cet envoi à s'effectuer n'influe pas sur le déroulement de la simulation. C'est également via un *DataSender* que la loi de simulation envoie, le cas échéant, les valeurs supplémentaires qu'il calcule.

De la même manière dans le cas de l'écriture dans un fichier, c'est via un *DataWriter* que l'écriture se fait, mais cette fois celui-ci est commun à chaque loi de simulation et à ses capteurs. Ces deux acteurs *DataWriter* et *DataSender* héritent d'une même classe abstraite.

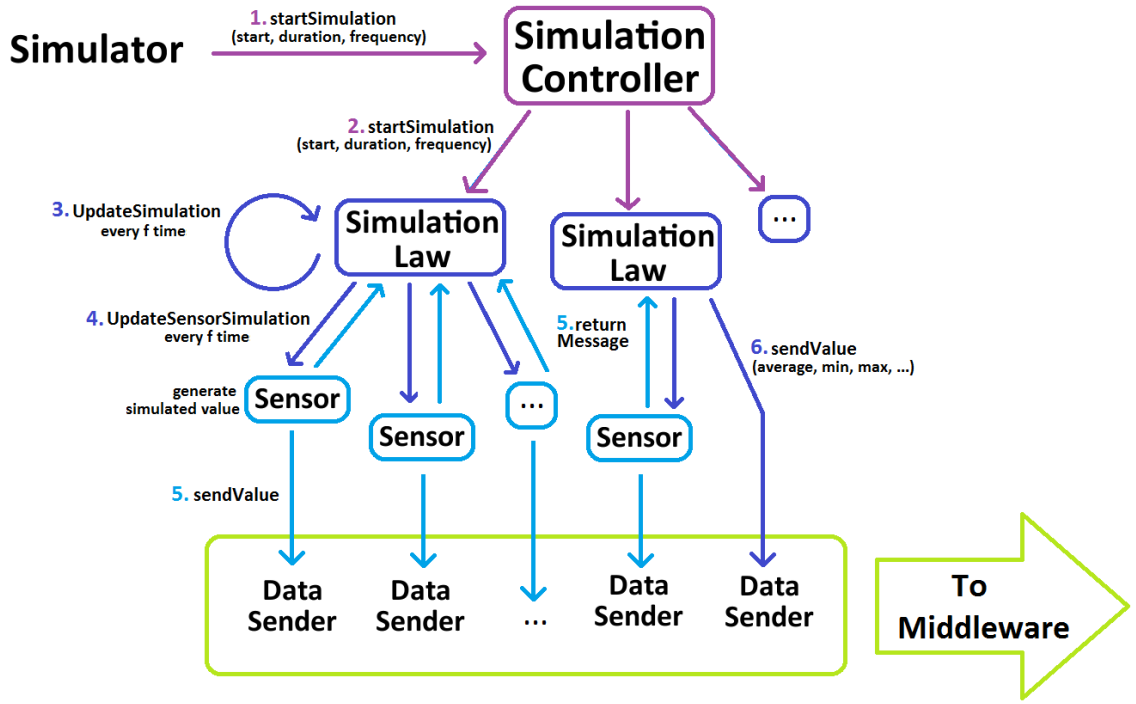

Figure 5 : Architecture des acteurs et des messages qu'ils s'échangent lors d'une simulation utilisant une loi mathématique.

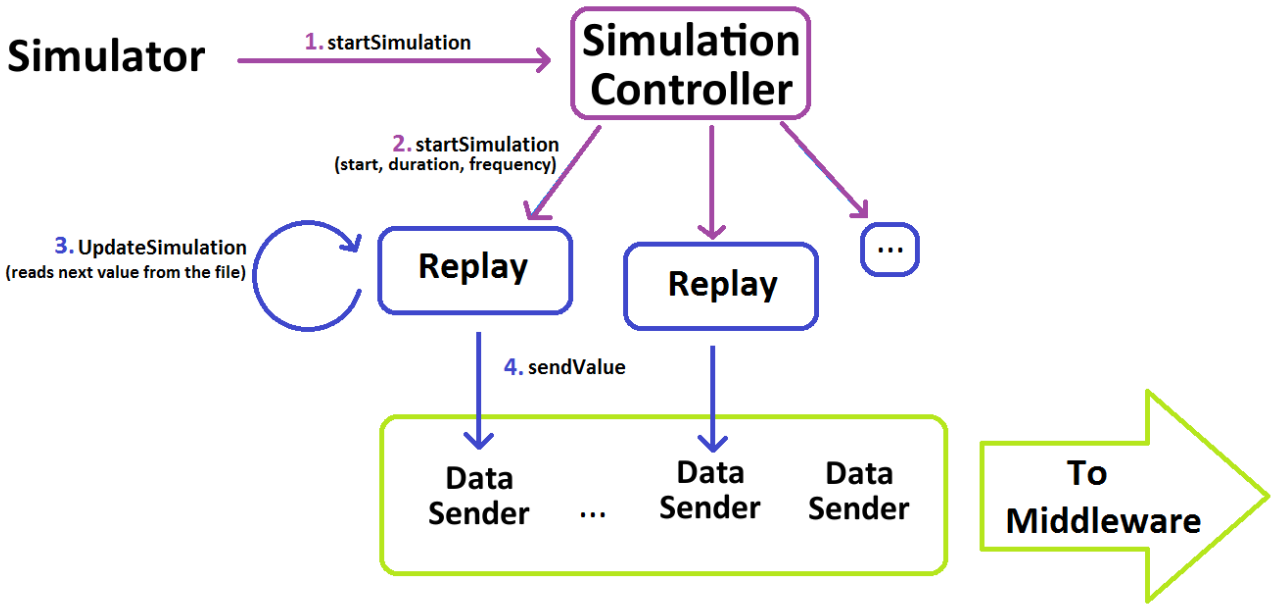

Figure 6 : Architecture des acteurs et des messages qu'ils s'échangent lors d'une simulation en mode rejeu de données.

b. Définition d'une simulation

Afin de faciliter l'utilisation de notre *framework*, nous avons décidé de mettre en place une "API fluide" [6]. Celle-ci se greffe en amont de notre *SimulationController*, et permet de masquer presque entièrement l'architecture du logiciel au développeur.

Le principe d'une API fluide est de proposer à l'utilisateur un mini-langage se rapprochant de la langue naturelle, afin de rendre son utilisation ainsi que sa lecture facile et intuitive.

Pour la mettre en place, nous avons créé une grammaire (Figure 7) associée au langage qu'elle représente.

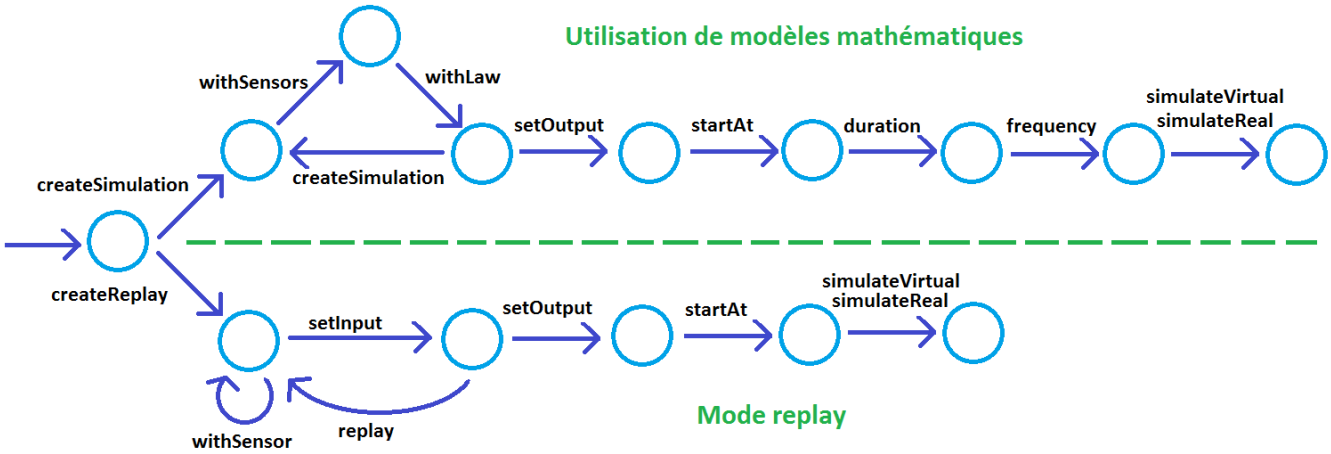

Figure 7 :Diagramme à états de notre API

La racine de notre grammaire nous permet de savoir si l'utilisateur va créer une simulation utilisant des modèles mathématiques ou un rejeu de données existantes. Passé cette état, nous arrivons dans les deux cas dans des états d'initialisation de paramètres. Après ces états, nous avons le choix de recréer le même mode de simulation ou d'initialiser les paramètres globaux de la simulation. Dans le dernier état, nous choisissons de lancer la simulation en temps réel ou en différé.

III. Application au scénario Smart Campus

1. Utilisation de notre projet

a. Installation du *framework*

Les trois parties de notre projet (framework, bibliothèque, SmartCampus) ont été créés sous Maven de qui permet de les installer très facilement.

Il faut exécuter les lignes de commandes suivantes pour l'installer :

```
~ git clone https://login@atlas.polytech.unice.fr/stash/scm/psiqje/private
~ cd private
~ mvn install
```

A la fin de l'exécution de cette commande, le programme compilé se trouvera dans le dossier target. La bibliothèque standard dépend du *framework*, et l'application à SmartCampus, des deux autres projets.

b. Rappel des fonctionnalités du programme

Il existe deux types de simulation :

- L'utilisation de modèles mathématiques
- Le mode replay

Ces deux fonctionnalités peuvent être jouées en temps réel ou en temps différé.

Pour pouvoir utiliser ces fonctionnalités, il y a deux actions possibles :

- L'utilisateur peut étendre des classes du *framework* ou de la bibliothèque standard afin de personnaliser ses simulations.
- Utiliser la fluent API pour créer et lancer des simulations en utilisant ses propres classes ou celles définies dans la bibliothèque standard.

2. Définition d'une simulation

Dans cette partie nous allons montrer concrètement deux exemples d'utilisation de notre *framework*. Le premier montrera comment on peut mettre en place un modèle mathématique, et s'en servir afin d'effectuer une simulation. Nous prendrons pour exemple le cas du scénario de capteurs de présence dans un parking. Le deuxième aura la même démarche, mais avec le mode de rejeu de données.

a. Simulation suivant une loi mathématique

i. Création du simulateur

Grâce à l'architecture que nous avons mise en place, une personne souhaitant créer une simulation suivant une loi mathématique doit uniquement redéfinir 3 classes :

- La loi de simulation. Dans notre cas, elle représente le parking.
- La loi mathématique utilisée. Ici nous implémenterons une chaîne de Markov (Figure 4).
- La transformation finale effectuée par les capteurs pour fournir la donnée à envoyer.

La chaîne de Markov nous renvoie par exemple : "20% de places du parking sont occupées" Chaque capteur va donc décider, à partir de ce pourcentage, si sa place est libre ou non.

Voici pour chacune de ces classes, la définition de notre classe et des méthodes nécessaires :

SimulationLaw :

```
public class ParkingMarkovSimulation extends SimulationLaw<Integer, Double, Boolean> {
	@Override
	protected Integer[] computeValue() {
		int i = 0;
		// compte le numbre de places de parking occupées à l'état précédent
		for (boolean b : this.values) {
			if (b) i++;
		}
		// t sera utilisé comme paramètre pour évaluer la loi
		Integer[] t = { i, i };
		return t;
	}
	/* Il est également possible de redéfinir la méthode onComplete, qui est appelée à chaque
	fois que les capteurs ont envoyé leur message de retour à la SimulationLaw */
	protected void onComplete() {
		int nbPlacesOccupied = 0;
		for (Boolean b : this.values) {
			if (b) nbPlacesOccupied++;
		}
		// envoie le pourcentage d'occupation du parking au middleware
		this.sendValue("occupation", ((nbPlacesOccupied * 100) / this.values.size()	+
		"%");
	}
}
```

Law :

```
public class ParkingMarkovLaw extends MarkovChain {
	public ParkingMarkovLaw(int nbPlaces, double arrivalFreq, double averageParkingTime) {
		super(nbPlaces + 1);
		for (int i = 0; i < this.size; i++) {
			/* creation de la matrice */
		}
		this.transition[nbPlaces][nbPlaces] = -(nbPlaces * averageParkingTime);
	}
```

```
	@Override
	protected Double evaluate( Integer... x) throws Exception {
		// la valeur de retour sera utilisée par les capteurs pour définir leur valeur
		return super.evaluate(x);
	}
}
```

SensorTransformation :

```
public class RateToBooleanSensorTransformation implements SensorTransformation<Double,
Boolean> {
	@Override
	public Boolean transform(Double rate, final Boolean last) {
		if (rate == 0) return false;
		Random r = new Random();
		return r.nextFloat() < rate;
	}
}
```

ii. Utilisation du simulateur

Voici un exemple de programme utilisant les classes définies plus haut :

```
Start sim = new StartImpl();
sim	.createSimulation("Parking", ParkingMarkovSimulation.class)
	.withSensors(500, new RateToBooleanSensorTransformation())
	.withLaw(new ParkingMarkovLaw(...))
	.setOutput("http://172.19.250.xxx:xxxx/xxx")
	.startAt("2014-01-22 08:25:00")
	.duration(Duration.create(2, TimeUnit.MINUTES))
	.frequency(Duration.create(1, TimeUnit.SECONDS))
	.simulateReal();
```

Ici on crée un parking suivant une chaîne de Markov et possédant 500 capteurs.
La simulation se déroule ici en temps réel, elle dure 2 minutes, et les capteurs génèrent une valeur chaque seconde.

b. Simulation en mode replay

	A	B	C	D	E	F	G	H
1	date	timestamp	conductivite	debit	niveau	NH4	O2	pH
2	01/17/2011	1	237,759995	351	1,778	4	11,88	7,9548
3	01/17/2011	2	267,679993	351,4	1,696	3	11,34	8,4126
4	01/17/2011	5	98,839996	351,4	4,872	2	11,718	7,308
5	01/17/2011	6					11,88	7,9548
6	01/17/2011	7					11,34	8,4126
7	01/17/2011	8						7,308
8	01/17/2011	11						7,9548
9	01/17/2011	12					11,34	8,4126
10	01/17/2011	13					11,718	
11	01/17/2011	14					11,88	
12	01/17/2011	17					11,34	8,4126
13	01/17/2011	23					11,718	7,308

Fig.6 Extrait d'un fichier Excel qui va servir au replay

i. Création du simulateur

Grâce à l'architecture que nous avons mise en place, une personne souhaitant créer une simulation qui consiste en un rejeu de données existantes (au format Excel) doit redéfinir qu'une classe qui hérite de *ExcelFormator*.

Dans cette classe on devra donner au *ExcelFormator* le numéro de la feuille de données qui nous intéresse, les colonnes qui correspondent au temps ainsi qu'un offset pour savoir à partir de quelle ligne commencent les données via le constructeur par défaut de la classe redéfinit.

Mais également la méthode *transform* qui transforme les colonnes qui représentent le temps en un timestamp en millisecondes.

```
public class SuezExcelFormator extends ExcelFormator{
	private final SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy");
	public SuezExcelFormator(){
		super(3,new String[]{"A","B"},2);
	}
	@Override
	protected long transform(String[] columns)  throws ParseException{
		long timestamp = 0 ;
		timestamp = sdf.parse(columns[0]).getTime();
		int hoursToMilli = Integer.valueOf(columns[1])*1000;
		timestamp += hoursToMilli;
		return timestamp;
	}
}
```

Il est également possible de rejouer des données provenant d'autres types de fichiers qui possèdent des colonnes nommées mais dans ce cas il faut hériter de *FileFormator* et redéfinir les méthodes adéquates.

ii. Utilisation du simulateur

```
Start sim = new StartImpl();
sim     .createReplay("Suez",SuezExcelFormator.class)
        .withSensor("o2sensor","G")
        .withSensor("pHsensor","H")
        .setInput("src/main/resources/test.xlsx")
        .setOutput("http://172.19.250.xxx:xxxx/xxx")
        .startAt(System.currentTimeMillis())
        .simulateReal();
```

Ci-dessus, nous créons une simulation permettant le rejeu des données du fichier test.xlsx, nous enverrons au middleware les valeurs contenues dans les colonnes G et H qui correspondent respectivement aux capteurs virtuels o2sensor et pHsensor.

IV. Bilan

1. Gestion du projet

a. Déroulement du projet

Durant ce projet, la découverte en profondeur du sujet et de nouvelles idées d'amélioration de notre *framework* nous ont poussé à effectuer certains choix de développement. Par conséquent, certaines idées initialement prévues n'ont pas été réalisées tandis que de nouvelles fonctionnalités ont été mises en place.

Nos objectifs initiaux étaient :
- *Framework* permettant d'effectuer des simulations
- Bibliothèque standard reprenant des éléments usuels
- Un environnement permettant d'utiliser ce *framework* dans le cadre du projet SmartCampus

Nous avons réaliser dans le cadre du projet :
- Dans le cadre du *framework* :
  - Permettre la création de simulateurs
  - Permettre de lancer des simulations à l'aide d'une *fluent* API
- Mise en place d'une loi polynomiale, d'une chaîne de Markov ainsi que de plusieurs *SensorTransformation* dans la bibliothèque standard.
- Mise en œuvre du scénario du parking de SmartCampus

Ayant initialement prévu de gérer tous les scénarios définis par SmartCampus, nous nous sommes finalement restreint uniquement à celui du parking. Ce changement d'orientation est dû à une volonté de mettre en place une application la plus générique et évolutive possible dans le but de faciliter un maximum la mise en place de nouveaux scénarios par la suite. Cela nous a permis également de nous concentrer sur l'implémentation du mode de rejeu de données, qui n'était pas prévu initialement.

b. Analyse de risque

Le plus gros risque de ce projet a été son étendue. En effet, de nombreuses possibilités se sont ouvertes à nous et il a fallu faire des choix, nous avons du dévier de nos objectifs initiaux au profit de la généricité et de l'évolutivité de notre application puisque ce sont les qualités principales que nous avons voulu fournir à notre programme.

Mais encore, nous avons eu un grand nombres d'interrogations tout au long de ce projet. En effet, tout d'abord, il nous fallait choisir le langage dans lequel nous allions développer notre programme. Scala semblait plus adéquat à notre problème mais nous n'avions que peut de connaissance de ce langage. Nous avons donc préféré écrire notre application en Java afin de nous concentrer sur la conception du *framework* (langage dans lequel nous avons des connaissances plus développées).

De plus, pour répondre au problème de la concurrence des capteurs virtuels, nous avons décidé d'utiliser un *framework* qui nous était totalement inconnu : Akka. Lorsque nous avions découvert ce *framework*, nous ne savions pas s'il pourrait répondre entièrement à nos besoins, ni si nous arriverions à le maîtriser.

Un problème de notre simulation que nous ne pouvions pas entièrement maîtriser, a été la possibilité que les temps de calculs dépassent les périodes d'envoi souhaités. Il est impossible de réagir à l'exécution pour répondre à ce cas de figure. Malgré cela, nous sommes parvenus à contourner le problème, en prolongeant la durée de la simulation tout en conservant la cohérence des données générées. De plus, nous signalons le problème à l'utilisateur, afin qu'il puisse prendre les mesures adéquates pour tenter d'y remédier.

Malgré ces obstacles nous pensons avoir réussi à mettre en place un *framework* solide et extensible, répondant de manière satisfaisante au problème posé, et pouvant être utilisé dans contextes très différents de celui de SmartCampus.

2. ... vers l'avenir

Certaines améliorations pourront être effectuées par la suite. En effet, la mise en place d'un stockage de données dans un fichier de type Excel serait un accroissement des capacités de notre application. Le développement d'une telle fonctionnalité permettrait de collecter les données dans des fichiers lorsque le serveurs est surchargé tandis que ces données pourrait être envoyées durant les heures creuses du serveurs.

De plus, SmartCampus est un projet fédérateur pour le pôle GLC du laboratoire I3S. Ce projet touche toutes les thématiques du pôle GLC du laboratoire [4]. Les domaines touchés par SmartCampus sont très larges, ce qui permet de susciter l'intérêt de nombreux chercheurs pour le projet, faisant de SmartCampus un pôle de rassemblement pour les chercheurs du laboratoire.

Mais plus important encore, notre *framework* a de l'avenir. En effet, une collaboration avec l'Université du Luxembourg est en cours. François Fouquet [3], un chercheur de l'Université du Luxembourg, est intéressé par nos résultats et souhaiterait réutiliser notre *framework* dans le cadre de son outil Kevoree (qui sert à gérer des grilles d'objets connectés), afin de simuler les différents capteurs qu'il n'a pas encore. Nous devrions le rencontrer au cours du second semestre.

V. Conclusion

Ce projet, nous a permis d'enrichir nos connaissance en Java, notamment au niveau de sa gestion de la généricité. De plus nous avons découvert un *framework* qui nous était inconnu : Akka.

L'étendue de Smart Campus et le travail que nous devions faire nous a fait rentrer dans un univers plus professionnel que scolaire : nous avons développé pour la première fois un *framework* complet accompagné d'une bibliothèque standard.

Le travail d'équipe que nous avons fourni a également permis de nous rapprocher du monde professionnel puisque nous avons travaillé en collaboration avec d'autres équipes telles que celles des SI5, des chercheurs du laboratoire I3S et dans le futur avec l'université du Luxembourg.

En nous appuyant sur le cas particulier de SmartCampus, nous avons réussi à fournir une solution complètement générique au problème de simulation de données en provenance de capteurs et qui sera probablement réutilisé.

VI. Références

Outils et *frameworks* utilisés :
[1] http://akka.io/

Documents de recherche :
[2] from : "Predicting Parking Lot Occupancy in Vehicular Ad Hoc Networks" by Murat Caliskan, Andreas Barthels, Bjorn Scheuermann and Martin Mauve
65th IEEE Vehicular Technology Conference, Dublin, Ireland, April 2007.

Autres :
[3] http://francois.fouquet.netcv.com/en
[4] http://glc.i3s.unice.fr/teams
[5] http://www.data.gouv.fr/
[6] http://www.martinfowler.com/bliki/FluentInterface.html