Projet de Fin d'Études
ASR - 2013/2014

Sondes géographiques pour systèmes répartis multi-échelles

Rapport d'activité

Étudiants : Luc Clément, Arnaud Saunier
Encadrants : Claire Lecocq, Sam Rottenberg, Chantal Taconet

Mardi 28 janvier 2014

Tables des matières

Introduction

La gestion des données de contexte est synonyme de préoccupation majeure dans le développement des applications de demain. En effet, les applications sensibles au contexte représentent un marché important pour le développement futur des applications mobiles. Celles-ci utilisent des informations de contexte de haut niveau d'abstraction, elles-mêmes traitées et obtenues à partir d'informations de contexte brutes issues de l'environnement de l'utilisateur.

Ce rapport est organisé de la manière suivante : tout d'abord sera présenté le contexte du projet, à savoir le projet INCOME, puis seront exposées les étapes de développement du projet, justifiant les choix d'architecture effectués et détaillant les différentes fonctions mises en place, ensuite seront mis en avant les différents commentaires sur le travail rendu, entre autres les améliorations possibles pour le projet, et les difficultés rencontrées lors de la réalisation, et enfin un manuel d'utilisation pour garantir la maintenabilité du code.

I. Contexte du projet

1.1. INCOME

Le projet INCOME (INfrastructure de gestion de COntexte Multi-Échellle pour l'Internet des Objets) est un projet financé par l'Agence Nationale de la Recherche (ANR) dans le cadre du programme Infrastructures matérielles et logicielles pour la société numérique (édition 2011). Soutenu par plusieurs organismes, tels que les pôles de compétitivité Aerospace Valley et Systematic, c'est cependant l'Institut de Recherche en Informatique de Toulouse qui en est le responsable. À ceci s'ajoutent les partenaires que sont Artal et Telecom SudParis.

Le projet a pour objectif de fournir infrastructure intergicielle pour la gestion de contexte multi-échelle pour l'Internet des objets offrant les services suivants :

- Acquisition des données de contexte
- Production des informations de plus haut niveau d'abstraction
- Acheminement des informations aux applications ambiantes

"INCOME a pour ambition de fournir des méthodes et des outils pour la gestion de contexte multi-échelle et de répondre aux verrous associés. INCOME cible le niveau infrastructure pour des applications grand public sensibles au contexte, à déployer à grande échelle en termes de nombre de sites de déploiement et de nombre d'utilisateurs.

Pour ce type d'application, les informations de contexte disponibles à l'utilisation varient en fonction des dimensions géographique et temporelle. Dans ces conditions, des stratégies de déploiement autonomiques des entités de gestion de contexte sont essentielles. Ces stratégies permettront de résoudre automatiquement les problèmes liés à l'instabilité et à l'ouverture de l'environnement tout en respectant un ensemble de contraintes de qualité de service ou de sécurité." (http://www.irit.fr/income/)

1.2. Présentation du projet

Dans le cadre du projet INCOME, notre projet s'inscrit dans une approche géographique du système, avec des mesures de localisation, et dans le développement de sondes géographiques multi-échelles.

La problématique est la suivante: comment prendre en compte les échelles géographiques associées à un consommateur dans l'échange de données avec des producteur(s)/consommateur(s)? Il faut donc déterminer ce que l'on entend par "échelle géographique associée à un consommateur".

En ayant pris en compte cette problématique, on comprend l'intérêt d'utiliser un système dit "multi-échelle". En effet, chaque consommateur sera défini par ses informations géographiques, qui vont correspondre à plusieurs "couches" géographiques définissant de manière unique un point de la Terre.

Le projet s'articule de la manière suivante:

- Obtention des informations de géolocalisation d'un terminal (fixe ou mobile)
- Transformation de ces informations en informations de plus haut niveau (adresse, ville…) c'est-à-dire en "couches" géographiques
- Tests d'appartenance d'un ensemble d'éléments distribués géographiquement à une même échelle géographique, c'est-à-dire déterminer la plus petite échelle géographique commune d'un ensemble d'éléments
- Affichage des informations sur une carte

Notre projet s'appuie sur la récupération et l'utilisation des coordonnées GPS, à savoir latitude et longitude (en format décimal). Le schéma suivant propose une vision simpliste de la gestion d'informations de localisation dans notre API.

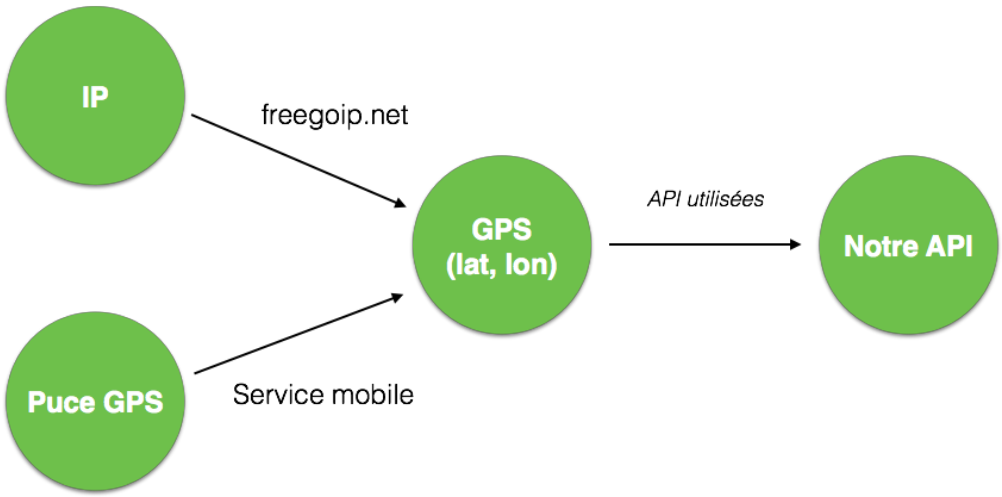


Figure 1. Schéma de l'acheminement des données

Une fois récupérées, les coordonnées GPS sont utilisées dans les différentes API que nous utilisons afin d'obtenir dans un premier temps les informations de géolocalisation et dans un second temps, après traitement par notre API, les informations de plus haut niveau.

Ensuite, les algorithmes implantés dans notre API vont permettre de déterminer les niveaux d'échelle communs entre plusieurs clients (jusqu'à n clients).

Enfin ces informations pourront être utilisées pour la gestion de données de contexte, ou tout simplement pour être affichées sur une carte.

II. Développement

2.1. Environnement de développement

Le projet a été développé sous **Eclipse**, en langage Java, et a utilisé des API Java dont les détails seront donnés dans la suite. Nous avons principalement développé sous les systèmes d'exploitation Mac OS X et Linux, et un peu sous Windows.

Les librairies nécessaires à l'exécution du projet sont les librairies **JMapViewer** (pour l'affichage de cartes, cf 2.2.3.) et **Jersey** (pour le service REST, cf 2.3.1.).

Voir la bibliographie pour les sources.

2.2. Description et cas d'utilisation des API

2.2.1. Relatives à la gestion de l'adresse IP

La gestion de l'adresse IP d'un terminal pour en déterminer ses coordonnées GPS a été étudié mais n'a pas été implanté dans la version finale du projet. En effet, dans la version finale, les coordonnées GPS sont entrées en "dur", à la fois pour un gain de temps mais également pour un gain de précision. Cependant, une possible solution est l'API **REST** publique *freegeoip.net*. *freegeoip.net* est financé par la communauté et est libre d'utilisation (jusqu'à 10000 requêtes par jour).

L'utilisation de ce genre d'API est très simple, il suffit d'implanter une requête HTTP GET dans le code, de la manière suivante :
Appel en REST à l'adresse : freegeoip.net/{format}/{ip ou hostname}

Exemple de requête REST et de réponse :
adresse : http://freegeoip.net/xml/
réponse :

```
-<Response>
  <Ip>157.159.110.48</Ip>
  <CountryCode>FR</CountryCode>
  <CountryName>France</CountryName>
  <RegionCode>A8</RegionCode>
  <RegionName>Île-de-France</RegionName>
  <City>Évry</City>
  <ZipCode/>
  <Latitude>48.6328</Latitude>
  <Longitude>2.4405</Longitude>
  <MetroCode/>
  <AreaCode/>
</Response>
```

Un inconvénient important de ce genre de service est la précision de celui-ci. En effet, l'adresse IP enregistrée est celle du noeud de raccordement d'abonnés du fournisseur d'accès internet correspondant au réseau internet auquel le terminal est connecté. Ainsi, les coordonnées renvoyées correspondront au noeud de raccordement d'abonnés et non à la position précise du terminal.

Il en va de même pour récupérer la position d'un terminal mobile. C'est pourquoi, il devrait être plus efficace d'utiliser la puce GPS du terminal mobile directement. Cette solution n'a pas été testée et est laissée inexplorée.

2.2.2. Relatives à la géolocalisation

Après avoir récupéré les coordonnées GPS d'un terminal, il faut être capable d'en déduire les informations qui nous intéressent, c'est-à-dire les échelles géographiques, qui sont des informations de plus haut niveau. Pour ce faire, plusieurs API ont été étudiées, en particulier leur service de *reverse geocoding*.

Le *reverse geocoding* ou géocodage inversé, consiste tout simplement à effectuer l'inverse du géocodage: trouver, à partir de coordonnées GPS plusieurs informations :

1. un point d'intérêt, accompagné de son adresse.
2. une adresse complète, avec numéro.
3. une adresse partielle, sans numéro.
4. un arrondissement ou une commune
5. un département, une région
6. un pays
7. tout autre zonage métier considéré, c'est-à-dire des zones plus spécifiques (zone d'activité, etc.)

On distingue alors parmi ces éléments, la topologie qui suit:

1. les adresses issues d'un noeud
2. les adresses issues d'un chemin
3. les adresses issues d'une relation (ou zone)

Cette fonction de *reverse geocoding* étant centrale dans notre projet, nous avons dû chercher des API efficaces répondant à ce besoin, et nous avons retenu deux API de géolocalisation, à savoir **Google Geocoding API** et **Nominatim API** (du projet **OpenStreetMap**).

Google Geocoding API

L'API Google Geocoding fait partie des services proposés par Google Maps. Google Maps constitue une base de données très fournie et semble le choix idéal tant au niveau de la fiabilité qu'au niveau du contenu. L'API Google Geocoding fournit un moyen direct d'accéder au services de géocodage et de géocodage inversé via des requêtes HTTP. Nous nous intéressons ici au service de géocodage inversé.

Ce service est supporté en utilisant en paramètres les coordonnées GPS, i.e. latitude et longitude. Par exemple, la requête suivante contient les valeurs de latitude et longitude de la rue Charles Fourier à Évry:

http://maps.googleapis.com/maps/api/geocode/xml?latlng=48.624422,2.44438&sensor=true

Ici le format de retour est le format XML mais peut être changé en JSON. Le résultat obtenu est comme suit:


```
▼<GeocodeResponse>
  <status>OK</status>
 ▼<result>
    <type>street_address</type>
    <formatted_address>5 Rue Charles Fourier, 91000 Évry, France</formatted_address>
   ▼<address_component>
      <long_name>5</long_name>
      <short_name>5</short_name>
      <type>street_number</type>
    </address_component>
   ▼<address_component>
      <long_name>Rue Charles Fourier</long_name>
      <short_name>Rue Charles Fourier</short_name>
      <type>route</type>
    </address_component>
   ▼<address_component>
      <long_name>Évry</long_name>
      <short_name>Évry</short_name>
      <type>locality</type>
      <type>political</type>
    </address_component>
   ▼<address_component>
      <long_name>Essonne</long_name>
      <short_name>91</short_name>
      <type>administrative_area_level_2</type>
      <type>political</type>
    </address_component>
   ▼<address_component>
      <long_name>Île-de-France</long_name>
      <short_name>IDF</short_name>
      <type>administrative_area_level_1</type>
      <type>political</type>
    </address_component>
   ▼<address_component>
      <long_name>France</long_name>
      <short_name>FR</short_name>
      <type>country</type>
      <type>political</type>
    </address_component>
   ▼<address_component>
      <long_name>91000</long_name>
```


Comme on peut le constater, le schéma XML qui semble opérer ici, parait relativement compliqué. C’est une des raisons qui nous a poussés à ne pas choisir cette API.

De plus, l'utilisation de cette API n'est pas libre et est soumise à une réglementation qui ne convient pas au projet que nous avons mené. Nous avons envoyé une requête à Google France en espérant un retour favorable afin d'utiliser l'API sans restriction dans un cadre purement éducatif, mais aucune réponse ne nous est parvenue.

Projet OpenStreetMap

Le projet OpenStreetMap a pour but de constituer une base de données géographiques libre du monde en utilisant le système GPS et d'autres données libres et ainsi permettre aux utilisateurs d'exploiter des carte du monde librement. Le projet a été lancé en 2004 par Steve Coast à l'**University College de Londres**. Début 2013, le projet a atteint son millionième contributeur. En effet, ce projet est basé sur l'intervention et la collaboration de tout utilisateur volontaire.

De la même manière que Google Maps, OpenStreetMap propose de nombreux services et API, certains pour contribuer au projet (API d'édition, …) et d'autres pour utiliser les données du projet. Dans le cadre du projet, plusieurs éléments vont nous intéresser: les API Nominatim et Overpass ainsi que la librairie JMapViewer (qui seront développés dans la partie suivante, cf. 2.2.3).

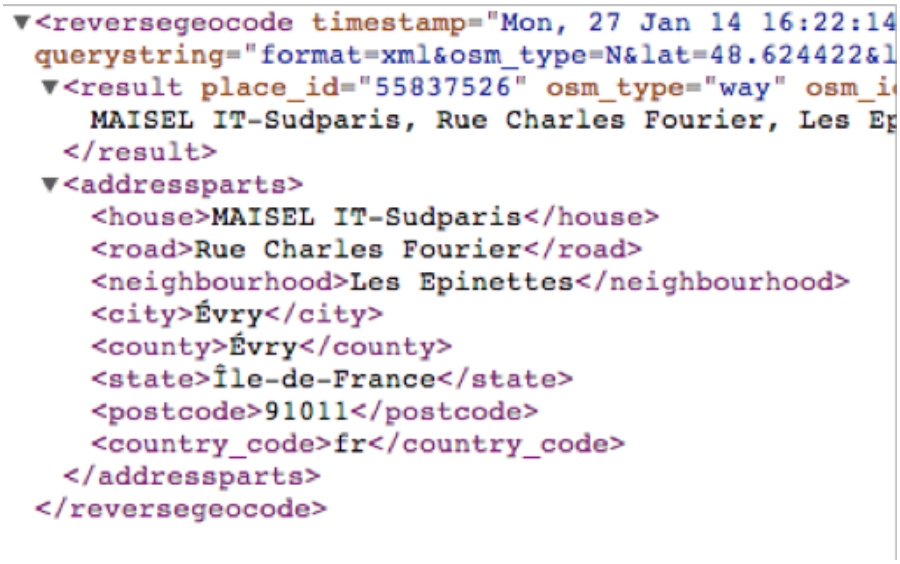
Nominatim API

À l'instar de Google Geocoding API, Nominatim fournit un moyen direct d'accéder au services de géocodage et de géocodage inversé via des requêtes HTTP.

La requête suivante effectue la même opération que l'on a vue précédemment, à savoir interroger le service (ici OpenStreetMap) pour avoir un retour d'adresse :

http://nominatim.openstreetmap.org/reverse?format=xml&lat=48.624422&lon=2.44438&zoom=18&addressdetails=1

Ainsi le résultat obtenu en XML :



On constate que le résultat obtenu semble plus simple que celui obtenu avec l'API Google Geocoding.

Cette API n'étant pas plus compliquée, voire plus simple à utiliser que les API de Google Maps, et possédant l'avantage de faire partie d'un projet libre (OpenStreetMap), elle comprend d'autres API et outils intéressants, alors nous avons décidé de l'utiliser dans notre projet.

L'utilisation de cette API est en effet assez simple, nous l'appelons de la façon suivante :

```
String REST_URI = "http://nominatim.openstreetmap.org/reverse";
ClientConfig config = new DefaultClientConfig();
Client client = Client.create(config);
WebResource service = client.resource(REST_URI);
reversegeocode geocodeResponse = service.path("xml").queryParam("lat", lat)
        .queryParam("lon", lon).queryParam("zoom", "18").queryParam("addressdetails", "1")
        .get(reversegeocode.class);
```

Ainsi à l'aide de **JaxB**, nous récupérons directement une instance de la classe **reversegeocode** que nous avons développée sur le schéma de réponse de l'API.

2.2.3. Relatives à la gestion de données distantes et à l'affichage

Dans l'optique d'implanter une carte lors de l'exécution du projet, il a fallu étudier les solutions qui s'offraient à nous, et surtout répondre au questions suivantes:

- Comment tracer une carte avec les données d'OpenStreetMap ?
- Comment récupérer ces données afin de tracer ce qui nous intéresse ?

JMapViewer

JMapViewer est une librairie Java permettant d'intégrer une carte OSM dans une application Java. Les données utilisées pour afficher des informations sur une carte sont les données directement issues d'OpenStreetMap.

Prenons un exemple: nous voulons tracer les frontières d'un pays comme la France. En regardant dans la documentation de la librairie :

http://josm.openstreetmap.de/doc/index.html?org/openstreetmap/gui/jmapviewer/JMapViewer.html

Nous constatons qu'une solution s'offre à nous: nous pouvons effectivement utiliser la classe MapPolygonImpl afin de dessiner le polygone qui représente les frontières de la France. Or en regardant le constructeur, **MapPolygonImpl**(ICoordinate... points) , nous remarquons que nous avons besoin de l'ensemble des points composant la frontière de la France.

C'est ici qu'intervient l'API Overpass.

Overpass API

Overpass est une API “lecture-seule” qui renvoie les données demandées par l’utilisateur à l’aide de requêtes spéciales. L’utilisateur effectue ces requêtes et l’API renvoie les données correspondant à ces requêtes. A contrario de l’API principale, Overpass est essentiellement destinée à la consommation de données.

Les requêtes sont au format XML ou au format Overpass QL (format propre à l’API) (cf. Bibliographie). Dans le cadre de notre projet, nous testons les requêtes dont nous voulons retrouver les données via l’interface d’Overpass Turbo (cf. Bibliographie, Figure 2), afin de vérifier la véracité de celles-ci. Puis nous utilisons le système de liens permanents pour les requêtes implantées dans le code. Exemple de lien permanent : http://overpass-api.de/api/interpreter?data={requête}

```
rel
  ["name"="Toulouse"]
  ["admin_level"="7"];
(>;);
out;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass API »>

<node id="ref1" lat="x1" lon="y1"/>
<node id="ref2" lat="x2" lon="y2"/>
        ...
<way id="id2">
   <nd ref="ref2" />
   <nd ref="ref1" />
   <nd ref="ref3" />
        ...
 </way>
<way id="id1">
        ...
</way>
<relation id="idRelation">
  <way ref="id1"/>
  <way ref=« id2"/>
        ...
</relation>

</osm>
```

Figure 2. À gauche, un exemple de requête. À droite le schéma de retour de l’API, au format XML.

Le traitement de ces données est abordé dans la partie 2.3.4. de ce rapport.

2.3. Architecture utilisée

2.3.1. Architecture centralisée

Nous avons choisi une architecture centralisée pour notre projet. Ainsi, l'architecture est composée d'un serveur REST auquel peuvent se connecter des clients pour accéder aux différents services du serveur (voir partie 2.3.3).

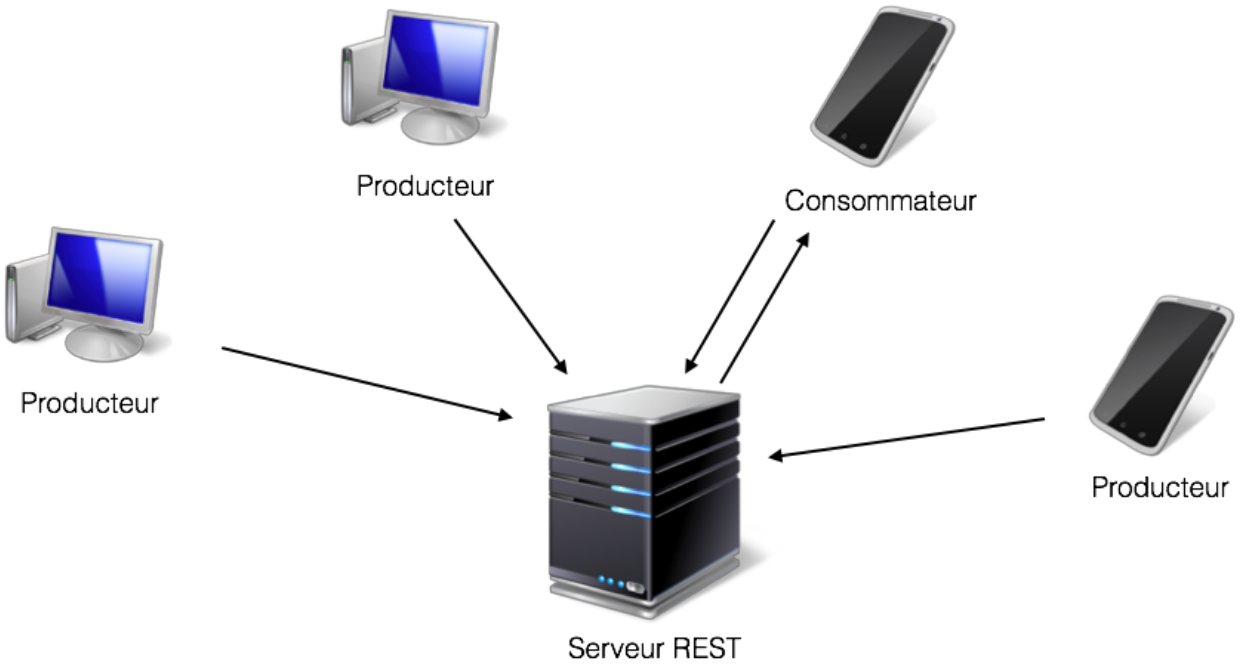


Figure 3. Schéma de l'architecture centralisée

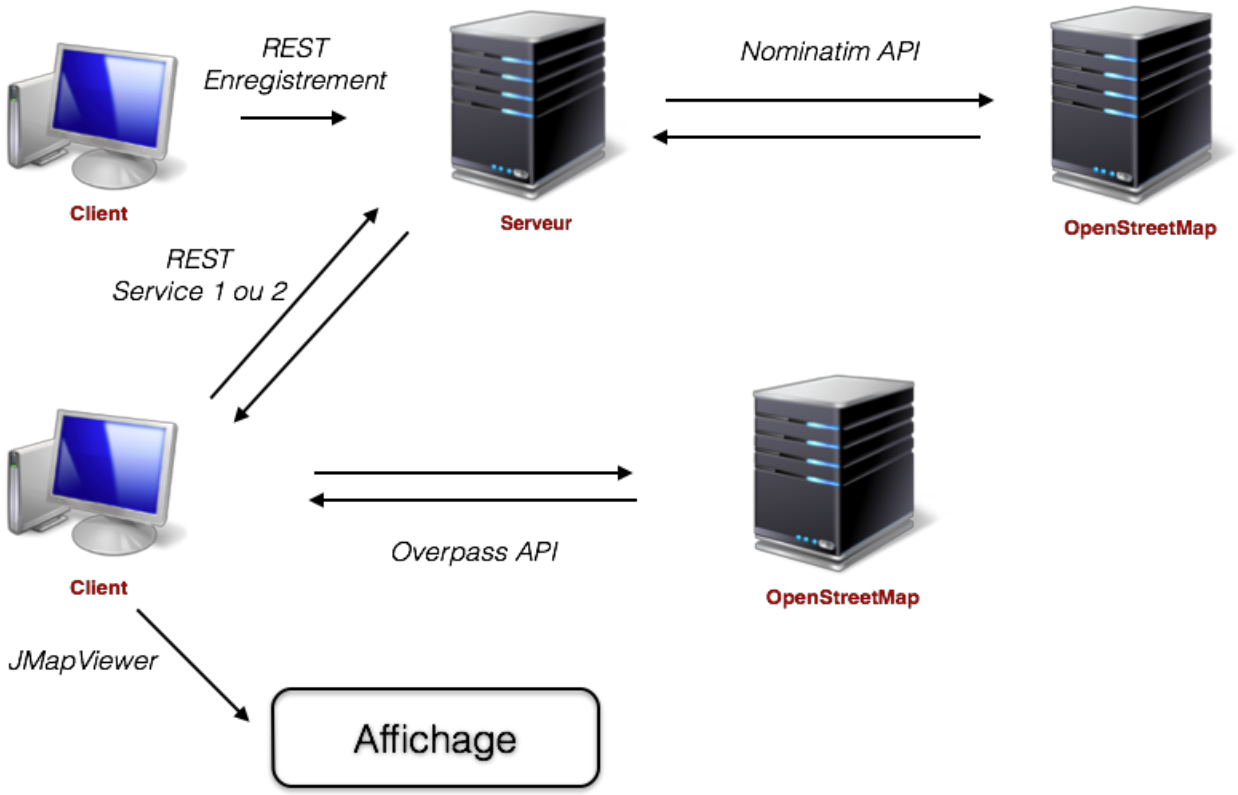
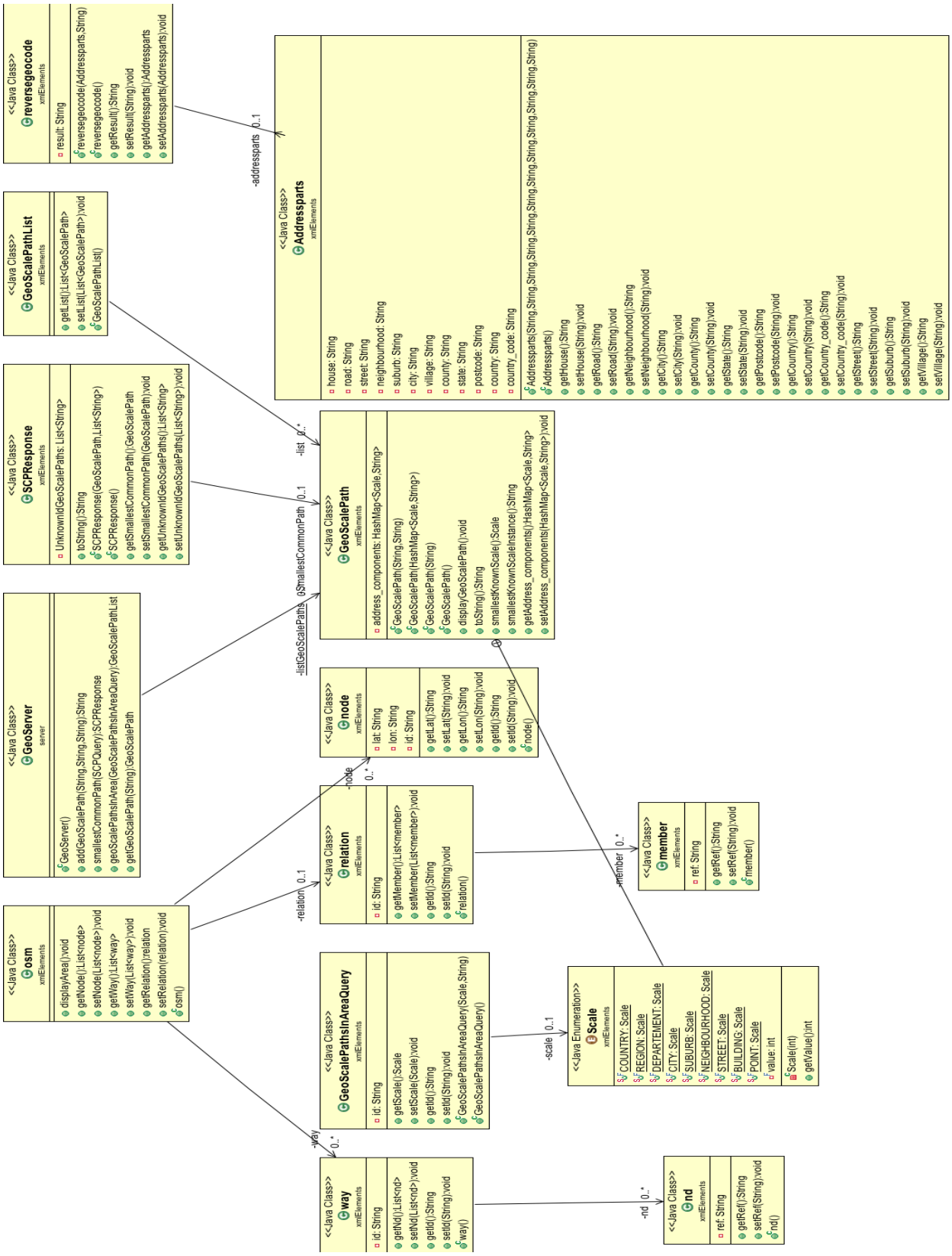


Figure 4. Schéma du déroulement d'un appel

La figure 4 montre les processus d'appel aux différentes API et les services qui entrent en jeu lors de l'exécution de notre projet.

2.3.2. Diagramme UML - Figure 5



2.3.3. Services fournis par le serveur

2.3.3.1. Enregistrement d'un client

Le premier service proposé par le serveur est tout simplement l'enregistrement sur le serveur d'un client. Le prototype du service est le suivant :

```
@GET
@Produces("text/plain")
@Path("addGeoScalePath")
public String addGeoScalePath(@QueryParam("name") String name, @QueryParam("lat") String lat, @QueryParam("lon") String lon)
```

Les trois arguments sont passés comme chaînes de caractères : l'identifiant du client à enregistrer, sa latitude et sa longitude. Ce service renvoie "success" si le client a bien été enregistré sur le serveur.

2.3.3.2. Détermination de la plus petite échelle commune

Ce service calcule, à partir d'une liste d'identifiants de clients, le plus petit chemin d'échelles commun entre tous les clients enregistrés parmi ceux de la liste passée en argument, ainsi qu'une liste des identifiants passés en argument qui n'étaient pas enregistrés sur le serveur.

Par plus petit chemin d'échelles commun, il faut comprendre un **GeoScalePath** comprenant comme **address_components** l'ensemble des instances d'échelles communes à tous les clients incriminés, à condition qu'ils n'existent pas pour ces clients des niveaux d'échelles plus larges différents (par exemple, ce service ne renvoie pas l'échelle ville si les deux clients à comparer se trouvent tous les deux dans une ville de même nom mais dans des régions ou pays différents).

Nous utilisons deux classes sérialisables, **SCPQuery** et **SCResponse**, comme classes d'arguments et de retour pour que **JaxB** puisse les marshaller automatiquement en XML. Le prototype du service est donc le suivant :

```
@POST
@Produces("application/xml")
@Consumes("application/xml")
@Path("smallestCommonPath")
public SCPResponse smallestCommonPath(SCPQuery idGeoScalePaths) throws JAXBException
```

2.3.3.3. Découverte des terminaux à proximité

Ce service permet à un client de récupérer l'ensemble des noeuds se trouvant dans la même instance d'échelle passée en argument d'un client, dont l'identifiant est également passé en argument.

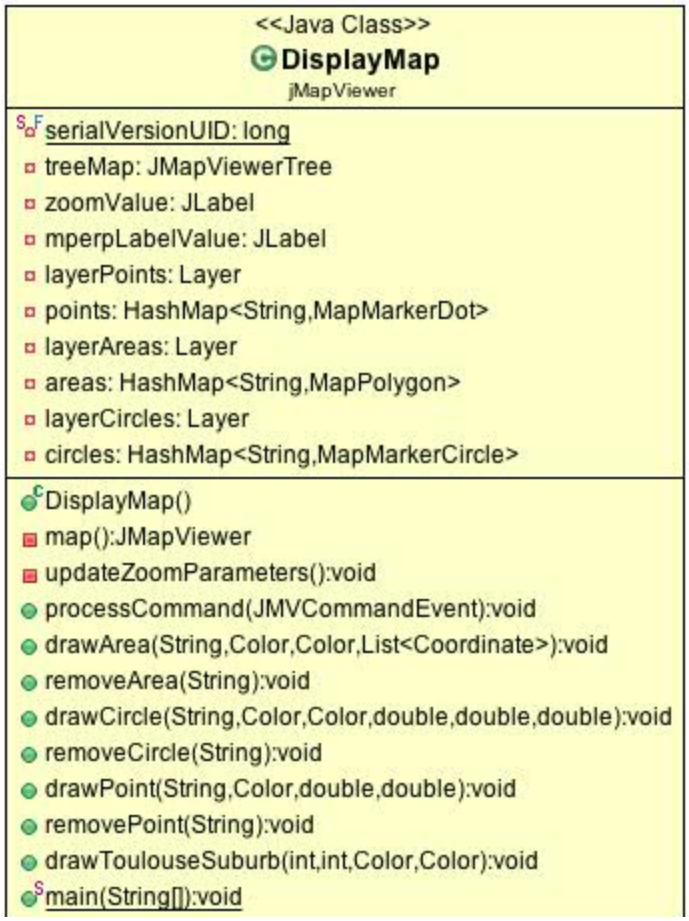
L'algorithme de ce service est relativement simple : le serveur parcourt la **HashMap** et ajoute un client à la liste des clients "à proximité" si pour chaque échelle du client de référence plus large (ou égale) à l'échelle de référence, le client étudié possède les mêmes échelles avec des instances d'échelles identiques à celles du client de référence.

Nous avons également créé deux classes sérialisables, comme pour le service de détermination de la plus petite échelle commune, **GeoScalePathsInAreaQuery** et **GeoScalePathList**, comme classes d'arguments et de retour pour que **JaxB** puisse les marshaller automatiquement en XML. Le prototype du service est le suivant :

```
@POST
@Produces("application/xml")
@Consumes("application/xml")
@Path("geoScalePathsInArea")
public GeoScalePathList geoScalePathsInArea(GeoScalePathsInAreaQuery query) throws JAXBException
```

2.3.4. Affichage des résultats

L'affichage des résultats sur une carte visible pour l'utilisateur se fait à l'aide de la librairie JMapViewer. Les outils et méthodes mis à disposition nous permettent de créer des objets simples (point, ligne, zone) sur une carte, toujours en prenant en argument les coordonnées GPS. La classe java **DisplayMap** instancie une carte et met en place les éléments nécessaire à la navigation dans celle-ci par l'utilisateur. Nous vous invitons à regarder la documentation de la librairie ainsi que la classe **DisplayMap** de notre projet.



Une des fonctionnalités qui nous intéresse particulièrement est la possibilité de tracer des polygones. En effet, c'est avec ce genre de méthode que nous pourrons représenter des zones géographiques et ainsi être capable d'afficher les échelles géographiques que nous avons créées ainsi que les informations obtenues grâce aux deux services décrits plus haut dans ce rapport.

À ce titre nous avons créé des méthodes de dessin de zone (**drawArea**), de cercle (**drawCircle**) et de point (**drawPoint**) qui permettent de dessiner chaque objets facilement (notamment définir un style graphique particulier).

Le code suivant représente les méthodes **drawArea** et **removeArea**.

```
 private Layer layerAreas = new Layer("LayerAreas");
 private HashMap<String, MapPolygon> areas = new HashMap<String, MapPolygon>();

public void drawArea(String name,Color borderColor , Color innerColor, List<Coordinate> nodes) {

    Style style = new Style();
    Color myColor = new Color(innerColor.getRed(), innerColor.getGreen(), innerColor.getBlue(), 50);
    Font font = new Font("Calibri", Font.BOLD, 20);

    style.setColor(borderColor);
    style.setBackColor(myColor);
    style.setStroke(new BasicStroke(2));
    style.setFont(font);

    MapPolygon frontieres = new MapPolygonImpl(layerAreas, name, nodes, style);

    //Add the area to the map
    removeArea(name);
    areas.put(name, frontieres);
    map().addMapPolygon(frontieres);
}


 public void removeArea(String name) {
     if (areas.containsKey(name)) {
         map().removeMapPolygon(areas.get(name));
         areas.remove(name);
     }
 }
```

Le système de couche (ou "layer") permet de regrouper plusieurs objets à dessiner (points, zones, cercles…) dans un même calque. Afin de pouvoir supprimer l'un de ces objets facilement par la suite pour l'effacer de la carte, à chaque fois que nous en ajoutons un, nous l'enregistrons dans une **HashMap** afin de pouvoir le récupérer facilement grâce à son identifiant.

Avant de pouvoir dessiner la zone à proprement parler, il nous faut la liste des coordonnées qui en composent la frontière. Ces points sont récupérés grâce à l'API Overpass. En reprenant le schéma vu plus haut:

```
rel
  ["name"="Toulouse"]
  ["admin_level"="7"];
(>;);
out;
```

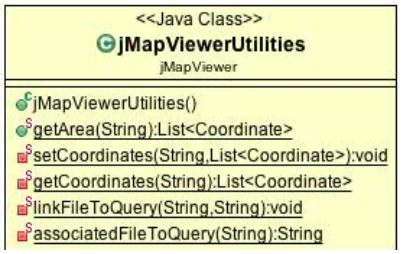
```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass API »>

<node id="ref1" lat="x1" lon="y1"/>
<node id="ref2" lat="x2" lon="y2"/>
        ...
<way id="id2">
   <nd ref="ref2" />
   <nd ref="ref1" />
   <nd ref="ref3" />
        ...
 </way>
<way id="id1">
        ...
</way>
<relation id="idRelation">
  <way ref="id1"/>
  <way ref=« id2"/>
        ...
</relation>

</osm>
```

Il est important de noter une chose: les balises <node .../> au début du fichier indiquent les points qui composent la frontière voulue, les balises <way.../> les chemins, et enfin la balise <relation .../> représente la frontière en tant qu'objet OSM.

Or si nous prenons simplement la liste des points à partir des balises <node … />, comme il semble légitime de faire, alors le tracé obtenu ne sera pas viable car les balises <node .../> sont organisées en fonctions de leur identifiant et non en fonction de leur succession dans la frontière. Il en va de même pour les balises <way .../>. Or dans la balise <relation ../> contenant elle-même des balises <way .../> faisant référence aux balises <way .../> vue précédemment, l'ordre de ces balises est le bon (en termes de succession dans la frontière). Il faut donc procéder en partant du bas du fichier :

(1) avec la balise <relation .../> on récupère l'ordre des way
(2) avec chaque balise <way .../> "indépendante" (qui n'est pas dans la balise <relation .../>) on récupère le bon ordre des nodes
(3) à partir des balises <node .../> restantes indépendantes on récupère les coordonnées de chaque point dans le bon ordre pour l'affichage (que l'on va enregistrer dans une **java.util.List**).

C'est pour cette raison que la classe **JMapViewerUtilities** a été implantée. En effet, elle contient la méthode **getArea** qui va précisément faire ce qui a été décrit plus haut.



Récupérer une zone est donc fastidieux car il faut systématiquement agencer les noeuds récupérés pour les remettre dans le bon ordre. Nous avons donc implémenté un système de sauvegarde de réponses aux requêtes de l'API Overpass dans le bon ordre.

Pour cette sauvegarde, nous avons décidé de procéder de la façon suivante : Un fichier "txt/overpassQueries/listQueries" contient l'ensemble des requêtes déjà enregistrées ainsi que le nom du fichier associé contenant la réponse à la requête en question (c'est-à-dire l'ensemble des points représentant la zone, dans le bon ordre), chaque ligne de ce fichier listQueries étant de la forme "queries<>nomDuFichierAssocié"..

Lors de l'appel à la fonction **getArea**, si la réponse à la requête a déjà été enregistrée localement (ce qu'on vérifie avec la méthode **associatedFileToQuery**), on ne fait pas appel à l'API Overpass mais on récupère les données en local (complexité linéaire donc bien plus rapide !) grâce à la méthode **getCoordinates**. Si ce n'est pas le cas, on fait appel à l'API Overpass pour récupérer les coordonnées souhaitées dans le bon ordre, et si le temps de traitement est trop long (plus de 500 ms), avant de renvoyer le résultat, on fait appel aux méthodes **setCoordinates** et **linkFileToQuery** pour enregistrer le résultat de la requête en local avec comme nom de fichier un UUID généré aléatoirement, afin de pouvoir y accéder plus rapidement les fois suivantes.

Les méthodes setCoordinates, getCoordinates, linkFileToQuery et associatedFileToQuery ont été déclarées *private* afin de ne pas toucher manuellement au rangement des données en local. Seule la méthode **getArea** accède à ces méthodes.

premières lignes du fichier listQueries :

(rel[name="Midi-Pyrénées"];._;>;);out;<>000001
43-d4b6-9542-0000-0143d4b754c4
(rel[name="Toulouse"][admin_level=8];._;>;);out
;<>00000143-d4b8-d56a-0000-0143d4b8e112
….

III. Difficultés rencontrées / Pour aller plus loin

Difficultés rencontrées

La première difficulté que nous avons rencontrée était la question des licences et des droits des API. Nous avions en effet prévu de nous orienter vers les API et outils de Google Maps au début de notre projet, mais leur utilisation n'est pas libre et soumise à certaines restrictions qui nous gênaient dans notre projet, mais surtout qui auraient pu devenir un vrai problème par la suite si le projet INCOME décidait de réutiliser notre code dans un programme plus large.

Nous avons tenté de joindre Google, en vain, et nous avons donc "résolu" ce problème en nous orientant vers le projet libre OpenStreetMap, qui nous permettait d'utiliser librement des outils similaires à ceux de Google Maps.

Une deuxième difficulté fut celle de la documentation du projet et des API issues d'OpenStreetMap. En particulier, aucun schéma XML n'était disponible pour les réponses des différentes API utilisées (Nominatim et Overpass), et nous avons donc du créer nos propres schémas, qui sont assez larges mais sans doute pas tout à fait complets. De plus, le projet étant communautaire, certaines zones géographiques sont moins bien référencées que d'autres, avec peu de données. Cela ne nous a posé aucun problème majeur dans le cadre de notre démonstration car nous sommes dans la zone urbaine de Toulouse qui est relativement bien décrite, mais cela reste un problème en général pour l'utilisation de cette API.

Enfin, l'affichage des données a parfois pu nous poser problème, encore une fois à cause du manque de documentation et d'exemples en ligne, davantage pour l'API Overpass et l'affichage de zones que pour l'utilisation de la librairie JMapViewer. Notamment, nous avons trouvé que les zones ou régions étaient très mal décrites par l'API Overpass et donc peu facile à dessiner sur la carte.

Pour aller plus loin

Notre projet part du constat selon lequel l'utilisateur dispose des coordonnées GPS des terminaux qui l'intéressent. Nous avons mentionné plus haut dans le rapport la possibilité d'intégrer la gestion de l'adresse IP des terminaux afin de récupérer les coordonnées GPS requise pour notre API. Cette voie n'a pas été approfondie, étant donné le temps limité accordé à ce projet. Nous voulions nous concentrer sur la partie la plus importante c'est-à-dire le traitement des données et la construction de l'API à proprement parler.

Les identifiants utilisés pour caractériser chaque terminal se connectant au serveur sont les noms que l'utilisateur donne à ceux-ci. Or, on peut imaginer donner des identifiants automatiquement, de manière unique à chaque terminal sans avoir à internvenir. Pour cela, l'adresse MAC peut être utilisée, seule, ou associée à un **UUID**. La classe **TestMAC** implante la récupération de l'adresse MAC de la machine.

Nous pouvons également imaginer étendre ce projet avec la gestion des terminaux mobiles, ce qui nécessiterait de développer des sondes en langage mobile tel qu'Android. Cette idée est à l'origine un des résultat du projet demandé si le temps nous le permettait.

Nous avons développé notre projet via l'IDE Eclipse, cependant, pour une utilisation plus universelle, il serait préférable d'utiliser l'outil Maven et ainsi créer des scripts de compilation.

IV. Manuel d'utilisation

Voir le paquetage **tests** du projet, ainsi que la Javadoc.

Comme dit précédemment, **Maven** n'a pas été utilisé dans le cadre de ce projet, donc les seuls besoins pour exécuter notre code sont un environnement **Java**, les librairies **Jersey** et **JMapViewer**.

Pour lancer en local, il suffit de lancer la classe **Publisher** qui lancera le serveur local, puis lancer une classe qui implante une connexion à celui-ci. Exemple :

```
static final String REST_URI = "http://localhost:9999/MyServer/";

public static void main(String[] args) {
    Client client = Client.create(new DefaultClientConfig());
    URI uri=UriBuilder.fromUri(REST_URI).build();
    WebResource service = client.resource(uri);
```

Certaines classes n'ont pas besoin de serveur pour être testées, notamment en ce qui concerne l'affichage d'éléments sur une carte (cf. paquetage **tests**).

Pour les démonstrations réalisées lors de la présentation :

Démonstration 1 :

- Lancer Publisher.java
- Lancer Clients.java
- Lancer Graphic.java et Graphic2.java
- Passer d'une étape à la suivante en appuyant sur entrée dans la console de Clients.java
    - Graphic affiche un point de vue externe, avec la position de chaque client, le suburb dans lequel se trouve Léon et un cercle autour de Chantal.
    - Graphics2 affiche le point de vue de Léon, qui ne souhaite partager ses données de contexte qu'avec les clients situés dans son suburb, on ne voit donc sur la carte que Léon, son suburb et les clients qui se trouvent dans son suburb.

Démonstration 2 :

- Lancer Publisher.java
- Lancer Clients.java
- Lancer Graphic3.java
- Passer d'une étape à la suivante en appuyant sur entrée dans la console de Clients.java
    - Graphic3 affiche l'ensemble des protagonistes, ainsi que la plus petite échelle commune entre Sophie et Amel. Dans la démonstration, selon l'étape, il peut s'agir de la région Midi-Pyrénées, de la ville de Toulouse, ou bien des suburb 1 ou 5 de Toulouse.

Conclusion

Ce projet était une opportunité pour nous de développer nos compétences en Java, notamment dans la gestion d'API et la gestion de données qui en sont issues, mais surtout, cela nous a donné la chance de participer à un projet d'envergure plus importante qu'est INCOME. L'usage permanent de Git nous a également permis de nous familiariser avec cet outil indispensable.

Les objectifs atteints

Nous avons pu atteindre les objectifs qui nous avaient été donnés et même approfondir certains points, et nous avons rendu un code fonctionnel, accompagné d'une démonstration et d'une javadoc complète.

Le travail restant

- Portage du code sous Androïd ou autres systèmes d'exploitation mobile.
- Intégration du code ou adaptation au projet INCOME.

Remerciements

Nous tenons à remercier particulièrement nos encadrants :

- Claire LECOCQ
- Sam ROTTENBERG
- Chantal TACONET

Bibliographie

JMapViewer : http://svn.openstreetmap.org/applications/viewer/jmapviewer/

Jersey : https://jersey.java.net/

Google Geocoding : https://developers.google.com/maps/documentation/geocoding/?hl=fr

Nominatim : http://wiki.openstreetmap.org/wiki/Nominatim
http://wiki.openstreetmap.org/wiki/Nominatim/FAQ
http://wiki.openstreetmap.org/wiki/Nominatim/Development_overview

Overpass Turbo : http://overpass-turbo.eu/

Overpass API : http://overpass-api.de/
http://wiki.openstreetmap.org/wiki/Overpass_API

Dépôt Git du projet :
git+ssh://fusionforge.int-evry.fr//var/lib/gforge/chroot/scmrepos/git/asr-geo-probes/asr-geo-probes.git

INCOME : http://www.irit.fr/income/