Module CSC5002 — ASR6:

Middleware for Internet distributed applications

Rapport du micro-projet

VLib Tour Application

Réalisé par :

Sana Bouchahoua et Amani Graja

TABLE OF CONTENTS

Introduction 3

1. Structure du projet 4

2. Technologies utilisées 6

3. Implémentation du module : VLibTour-Emulation-Visit 7

4. Implémntation du module : VLibTour-Tour-Management 8

5. Implémentation du module : VLibTour-Group-Communication-System 9

6. Implémentation du module : VLibTour-Lobby-Room 10

7. Intégration des différents modules dans la classe VLibTourVisitTouristApplication 11

Conclusion 12

Annexes 13

INTRODUCTION

Dans le cadre de ce projet, nous allons développer le backend d'une application mobile VLibTour, proposée pour encourager l'utilisation du système de cyclotourisme.

A travers cette application, des groupes de touristes peuvent faire des tours à Paris à vélo et visiter des lieux d'intérêts POIs.

Les tours sont sous la forme d'une séquence de points d'intérêt POIs. Chaque touriste utilise l'application VLib pour se déplacer d'un POI à un autre afin d'arriver à la fin à la même destination de son groupe en suivant un chemin particulier. Tous les participants peuvent voir la carte et la localisation des autres membres de leur groupe.

Le projet initial est constitué de plusieurs modules représentant les différents composants qui doivent être développés et qui peuvent être déployés indépendamment.

La figure suivante détaille l'architecture du système, il s'agit bien d'une architecture de système reparti :

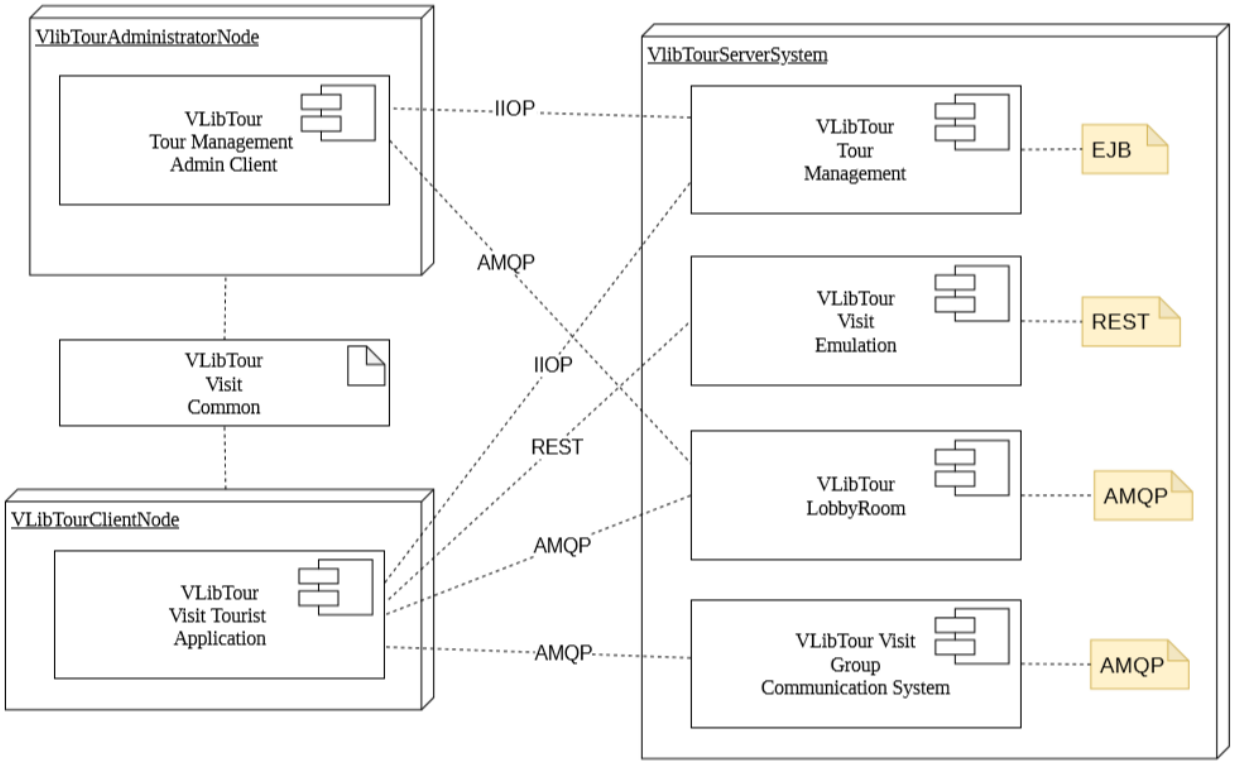

Figure 1: Architecture générale du système

1. STRUCTURE DU PROJET

Nous avons récupéré une version initiale des modules Maven de notre projet.

Ces modules sont dans l'arborescence suivante :

```
├── pom.xml
├── README.txt
├── run_scenario_w_mapviewer.sh
├── Scripts
├── vlibtour-libraries
   ├── geocalc
   └── vlibtour-common
├── vlibtour-tour-management
   ├── vlibtour-tour-management-api
   ├── vlibtour-tour-management-bean
   └── vlibtour-tour-management-entity
├── vlibtour-lobby-room-system
   ├── vlibtour-lobby-room-api
   └── vlibtour-lobby-room-server
├── vlibtour-emulation-visit
├── vlibtour-client
   ├── vlibtour-admin-client-tour-management
   ├── vlibtour-client-emulation-visit
   ├── vlibtour-client-group-communication-system
   └── vlibtour-client-lobby-room
└── vlibtour-scenario
```

Nous avons établi des changements sur les modules suivants en utilisant à chaque fois une nouvelle technologie afin de réaliser les objectifs du projet.

- **vlibtour-emulation-visit**: ce module repose sur une API REST et assure l'émulation des positions des différents utilisateurs notamment Joe et Avrel.
- **vlibtour-client**: contient les modules des différents clients :

- vlibtour-admin-client-tour-management : le client de VLibTour Tour Management (EJB entities)

- vlibtour-client-lobby-room : le client de VLibTour lobby room (protocole AMQP)

- vlibtour-client-group-communication-system : le client de VLibTour group communication system (protocole AMQP)

- vlibtour-client-emulation-visit : le client de VlibTour emulation visit (technologie REST)

- **vlibtour-scenario**: ce module contient le client de l'application : VLibTourVisitTouristApplication et permet de dessiner la carte et les positions des utilisateurs en utilisant le service OpenStreetMap.
- **vlibtour-tour-management**: ce module assure la gestion des tours par la technologie EJB
  - vlibtour-tour-management-api: le module de l'API du VLibTour Tour Management
  - vlibtour-tour-management-entity: le module qui contient les classes des deux entités POI et Tour et définit les relations entre eux.
  - vlibtour-tour-management-bean: le module qui définit le serveur de VLibTour Tour Management bean et contient les méthodes listées dans le module de l'API.

- **vlibtour-lobby-room-system:** ce module parent assure la création des visites et les rejoindre (RabbitMQ)
  - vlibtour-lobby-room-api: l'API du module VLibTour lobby room
  - vlibtour-lobby-room-server: le serveur du module VLibTour lobby room

- **vlibtour-bikestation:** permet aux utilisateurs de récupérer des informations sur les vélos disponibles (Cette partie n'a pas été développée dans ce projet).

2. TECHNOLOGIES UTILISÉES

Configuration de la machine :

Le paramétrage de notre Shell bash se passe au niveau du fichier de configuration : « .bashrc ». On ajoute donc les lignes suivantes pour préparer notre environnement de travail :
~/MWInstallation/java8
~/MWInstallation/env_glassfish.sh
~/MWInstallation/env_rabbitmq.sh
Ensuite, on lance le script Shell de notre application « install_glassfish.sh » qui utilise les scripts du répertoire 'Scripts'.

Parmi les technologies et les exigences de l'environnement que requière ce projet, on cite les suivantes :

Java8, Maven, Erlang, RabbitMQ Broker, EJB, REST

On s'intéresse dans ce rapport principalement de :

- Enterprise JavaBeans (EJB) qui est une architecture de composants logiciels côté serveur pour la plateforme de développement Java EE.
- Java EE qui offre des bibliothèques logicielles additionnelles dédiées à des applications professionnelles, facilitant le développement d'applications pour une architecture distribuée.
- REST (REpresentational State Transfer) qui est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web RESTful, établissant une interopérabilité entre les ordinateurs sur Internet.
- RabbitMQ qui est un message broker et a pour rôle de transporter et de router les messages depuis les « publishers » vers les « consumers » en implémentant le protocole AMQP. Ce broker utilise les exchanges et bindings pour savoir s'il doit délivrer, ou non, le message dans la queue.

3. IMPLEMENTATION DU MODULE : VLIBTOUR-EMULATION-VISIT

Ce module repose sur une API **REST**. Il est ainsi composé d'un client et d'un serveur REST.

De plus, il assure l'émulation des positions des différents utilisateurs notamment Joe et Avrel qui se déplacent d'un POI à un autre pour un tour donné.

L'implémentation de ce module a pour objectif d'écrire les méthodes manquantes dans la classe du serveur qui assurent le déplacement des utilisateurs et leur visualisation sur la carte et d'écrire le proxy du côté du client qui implémente ces méthodes.

1/ Dans le serveur, on écrit les méthodes suivantes :

stepsInVisit : qui permet à chaque utilisateur se trouvant sur un POI particulier, de se déplacer vers le POI suivant.

stepsInCurrentPath : retourne la nouvelle position d'un utilisateur s'il va passer à la position suivante ou bien sa position actuelle s'il se trouve sur le dernier POI du tour.

getCurrentPosition : retourne la position actuelle de l'utilisateur.

getNextPOIPosition : retourne la positon du prochain POI du tour.

On définit pour chaque méthodes les annotations suivantes afin de séparer la couche d'implémentation de la couche web :

@GET : c'est le verbe HTTP qui provoque l'appel de la méthode.

@Path("/getNextPOIPosition/{user}") : c'est le lien vers la ressource.

@Produces(MediaType.APPLICATION_JSON) : pour que la méthode renvoie son contenu JSON à la classe du serveur.

2/ On écrit ensuite dans la classe client/proxy le code qui implémente les méthodes du serveur REST. Le proxy envoie des requêtes REST au serveur pour appeler les différentes méthodes. Chacune retourne une position particulière.

3/ On modifie ensuite la classe VlibTourVisitTouristApplication pour pouvoir lancer notre client.

On crée une instance VlibTourVisitTouristApplication dont les paramètres sont :

- Le tourId qu'on lui affecte : "The unusual Paris"
- Le groupId qu'on lui affecte : Optional.empty()
- Le userId, qui peut être soit Joe ou bien Avrel selon le choix qu'on affecte en exécutant la commande lors du lancement du client.

Ensuite, on écrit la boucle « While » pour pouvoir déplacer l'utilisateur sur la carte à travers les appels du proxy.

4/ Pour pouvoir lancer ce service, on exécute les commandes suivantes :

On lance le serveur à partir du répertoire vlibtour-emulation-visit : *Mvn exec :java@server*

On lance le client à partir du répertoire the vlibtour-scenario *: Mvn exec :java@touristappliavrel*

Avantages de l'utilisation de l'architecture REST dans ce module :

Scalabilité : Ce système étant sans état « *stateless* » libère les ressources de mémoire, permettant au service de s'adapter au nombre de requêtes simultanées.

Performance : L'objectif en performance est atteint en utilisant des caches pour garder les données disponibles près de l'endroit où elles sont traitées.

4. IMPLEMNTATION DU MODULE : VLIBTOUR-TOUR-MANAGEMENT

Dans ce module, nous implémentons un client-serveur en utilisant la technologie EJB. On utilise pour cela 3 sous-modules (*entity*, *bean* et *api*) qu'on ajoute dans le fichier pom.xml du module parent pour qu'ils deviennent exécutables lors du lancement de l'application.

L'implémentation des composants EJB :

Tout d'abord, on écrit la classe de l'API qui présente une interface contenant les différentes opérations de gestion des POIs et des tours.

Ensuite, on s'intéresse au *bean* (serveur). On écrit donc les méthodes présentant les services et les traitements effectués sur les différentes entités.

Cette classe est *stateless* donc elle ne conserve pas son état entre les différents appels ce qui garantit la scalabilité de notre système.

Enfin, on s'intéresse au module *entity* qui contient les classes des deux entités POI et Tour et définit les relations entre eux.

La relation entre la collection des tours et la collection des POIs est une relation ManyToMany. En effet, chaque POI peut appartenir à plusieurs tours à la fois et chaque tour contient une collection de POIs.

```
@ManyToMany(mappedBy="pois")
private Collection<Tour> tours;
```

```
@ManyToMany
private Collection<POI> pois ;
```

Le déploiement de EJB :

Un EJB doit être déployé sous forme d'une archive jar contenant un fichier qui est le descripteur de déploiement et toutes les classes qui composent chaque EJB.

L'archive doit contenir un répertoire META-INF qui contiendra lui-même le descripteur de déploiement.

Le reste de l'archive doit contenir les fichiers .class avec toute l'arborescence des répertoires des packages. Ainsi que le jar des EJB peut être inclus dans un fichier de type EAR.

Les avantages de l'architecture EJB dans ce module :

Réutilisabilité : l'EJB assure la réutilisation des composants.

De plus, l'utilisation d'attributs et des annotations est possible avec l'architecture EJB.

5. IMPLEMENTATION DU MODULE : VLIBTOUR-GROUP-COMMUNICATION-SYSTEM

Le module « group communication system » est basé sur la programmation évènementielle. Chaque touriste envoie sa position dans un message d'une manière périodique qui sera affichée chez les autres utilisateurs du même groupe.

Cette classe permettra de gérer des files de messages afin de permettre à différents touristes de communiquer très simplement.

Pour ce faire, nous avons utilisé la technologie RabbitMQ qui implémente le protocole AMQP dans la classe *VLibTourGroupCommunicationSystemClient* du module *vlibtour-client-group-communication-system* du répertoire *vlibtour-client.*

La création de ce proxy a pour but d'alléger le code de la classe *VLibTourVisitTouristApplication*.

Nous allons donc pouvoir faire en sorte que notre application envoie des messages vers RabbitMQ, qui va ensuite les transmettre à d'autres clients qui vont pouvoir agir en conséquence.

On commence alors par l'établissement de la connexion à travers laquelle, il y'a ouverture d'un ou plusieurs « channels ». Ces channels sont des canaux de communication indépendants permettant de faire passer différentes communications en parallèle au sein de la même connexion TCP.

Ensuite, on s'intéresse à la production d'un message. En effet, les touristes qui produisent un message sont appelés des « producers ». Une fois leur message produit, ils déposent ce message dans un « exchange » qui va servir à trouver dans quelle(s) "queue(s)" le message doit être entreposé.

On se sert par la suite du binding qui constitue les règles qui vont permettre à l'exchange de déterminer dans quelle(s) queue(s) il doit déposer le message.

Les queues sont les endroits finaux où sont entreposés les messages. Une fois dans une queue, un message est prêt à être consommé.

Les touristes de notre application sont ceux aussi qui consomment les messages et sont appelés des « consumers ». Ils s'abonnent maintenant à des files de messages, et RabbitMQ leur transmettra un message dès que celui-ci arrivera dans la queue. On peut avoir plusieurs consumers pour la même file de message. Pour cela nous nous servirons des *Exchange_name, Queue_name* et *bindingKey* pour éviter les collisions avec les autres groupes de communication appartenant à d'autres tours.

Avantages de RabbitMQ dans ce module :

Disponibilité : Si un consommateur n'est pas disponible pour une raison quelconque, les messages envoyés par le producteur seront simplement mis en attente jusqu'à ce que le consommateur soit de nouveau disponible.

Cela signifie que le producteur peut poursuivre son propre travail jusqu'à ce que le consommateur soit de retour et prêt à traiter les messages reçus. Sans la file d'attente des messages, le producteur devrait attendre que le consommateur soit prêt. Au lieu de surcharger le consommateur avec des charges de travail qu'il ne peut pas gérer, RabbitMQ permet au consommateur de récupérer le travail quand il est prêt d'être traité.

Évolutivité : De même, s'il y a trop de messages pour qu'un consommateur puisse les traiter, les messages peuvent simplement rester dans la file d'attente pour être consommés plus tard. Alternativement, plus de consommateurs peuvent être ajoutés pour traiter les messages provenant de la file d'attente. Cela améliorera les délais de traitement.

6. IMPLEMENTATION DU MODULE : VLIBTOUR-LOBBY-ROOM

Dans ce module on s'intéresse à la classe *VLibTourLobbyServer* qui implémente le « VLibTour lobby room service » qui fournit l'infrastructure pour les « group communication systems ». Quand un touriste veut commencer un tour, il contacte le lobby room pour créer ou rejoindre un groupe. Créer un groupe et le rejoindre donne l'accès à l'utilisateur au groupe de communication correspondant. Chaque groupe possède un Vhost (groupId) créé et géré par le lobby room server.

Interface du lobby room service :

Présente l'API qui définit les méthodes implémentées dans le serveur du lobby room.

Implémentation du lobby room server :

Dans cette classe, on implemente les méthodes suivantes :

- *createGroupAndJoinIt* qui permet au touriste de créer un groupe et le rejoindre, et elle retourne l'URL qui contient l'identifiant du *groupe (grouId)*, le login *(userId)* et le mot de passe *(password) et* permet donc aux autres membres du groupe de communication de se connecter.

- *joinAGroup* qui permet au touriste de rejoindre un groupe existant.

Implémentation du lobby room client/proxy :

Cette classe contient deux constructeurs, le premier pour un touriste qui souhaite rejoindre un groupe existant et le deuxième pour celui qui souhaite créer un nouveau groupe.

Ensuite, chaque touriste peut utiliser les méthodes implémentées du côté du serveur.

Après création de la « connection » et du « channel », un JsonRpcClient assure l'appel des méthodes mentionnées.

Avantages de RabbitMQ dans ce module :

Sécurité :

- Elle est assurée par le concept du Vhost avec le contrôle d'accès : Seuls les utilisateurs d'un même Vhost peuvent recevoir les positions des autres touristes appartenant au même groupe.
- Pour se connecter, chaque utilisateur doit posséder un userId et un password.
- De plus, on peut attribuer à chaque utilisateur des permissions bien déterminées.

7. INTEGRATION DES DIFFERENTS MODULES DANS LA CLASSE VLibTourVisitTouristApplication

Après avoir implémenté les différents modules chacun à part, nous les intégrons tous ensemble grâce à la classe du client le VLibTourVisitTouristApplication.

Chaque nouveau client, peut accéder à l'application. Il peut créer un nouveau groupe en spécifiant le tourId et le rejoindre. Sinon, il rejoint un groupe qui existe déjà.

En se déplaçant d'une position à une autre il envoie sa position actuelle aux autres touristes du même groupe et reçoit les siennes.

Finalement, sur la même carte, un touriste peut visualiser sa propre position et les positions des membres de son groupe.

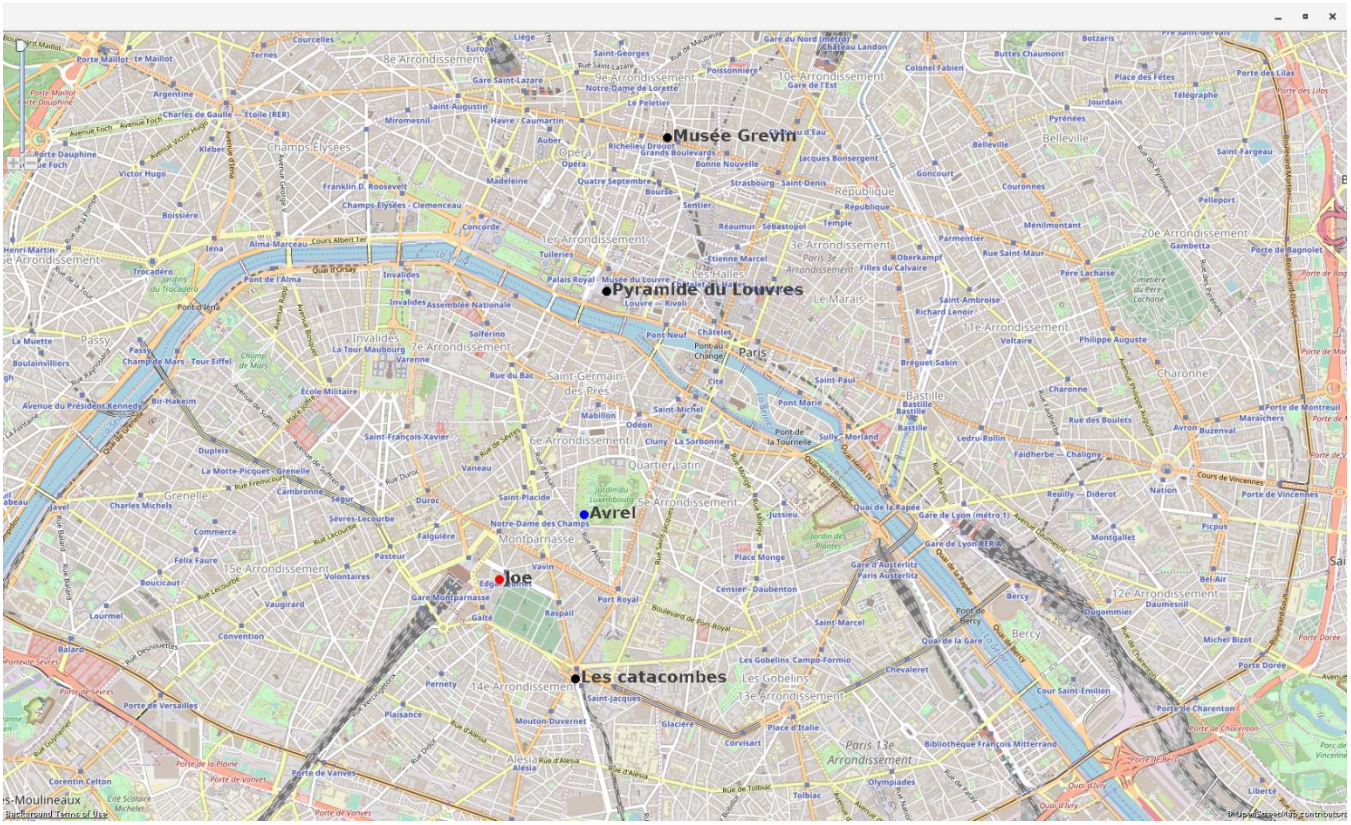

Figure 2: Joe et Avrel sur la même carte

CONCLUSION

Ce projet étudie le cas d'un système reparti. Il consiste à développer le backend d'une application mobile en utilisant plusieurs technologies assurant les meilleures performances et respectant les propriétés extra fonctionnelles.

A travers cette application, des groupes de touristes peuvent faire des tours à Paris à vélo et visiter des lieux d'intérêts POIs.

Nous réussissons à la fin de ce projet à développer les différents modules du système, les tester indépendamment et les intégrer ensemble.

Grâce aux différentes architectures implémentées, nous assurons un degré de fiabilité, sécurité et extensibilité par la réutilisation de composants pouvant être gérés et mis à jour sans affecter le système global, même pendant leur fonctionnement.

Lors de la réalisation, nous avons rencontré quelques difficultés liées principalement à la mise en place de l'environnement du travail. En plus, parfois nous passons beaucoup de temps sur une erreur qu'on n'arrive pas à comprendre facilement ce qui nous pose une charge horaire un peu élevée. Cependant, grâce à la collaboration avec nos enseignants et nos collègues et aux recherches faites, nous arrivons toujours à trouver des solutions et finaliser toutes les étapes.

Cette application est intéressante et nous pouvons ajouter plusieurs fonctionnalités et améliorations pour la rendre plus attirante. Pour cela, on peut proposer l'enregistrement des données relatives aux distances parcourues et leur comparaison aux sorties précédentes.

ANNEXES

Figure 3: cas d'utilisation du module Tour Management

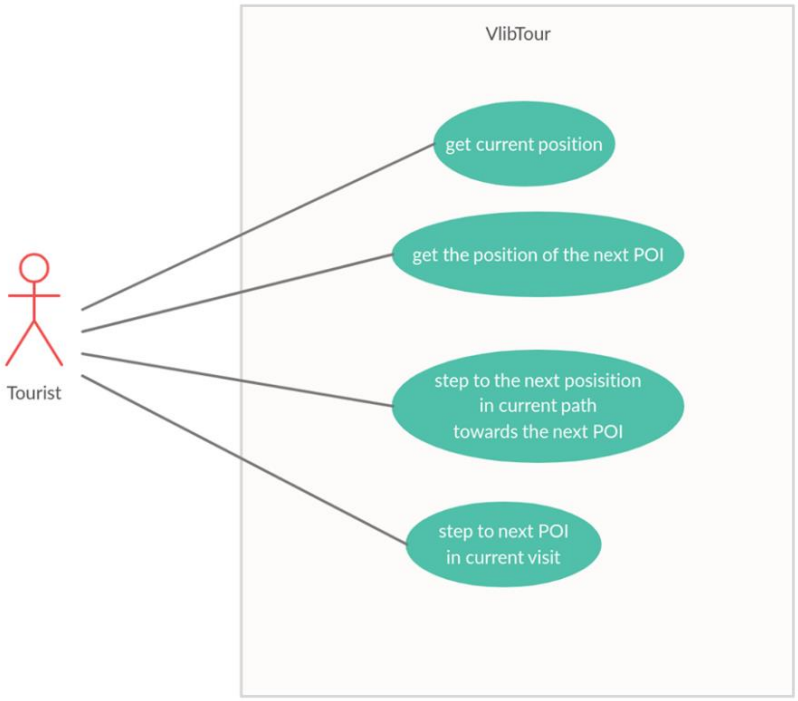

Figure 4: Cas d'utilisation du module Visit Tour Emulation

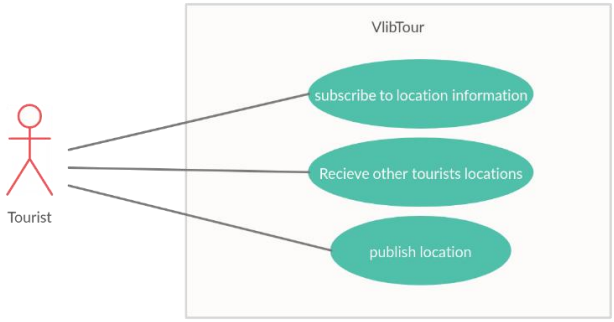

Figure 5: Cas d'utilisation du module Group Communication System

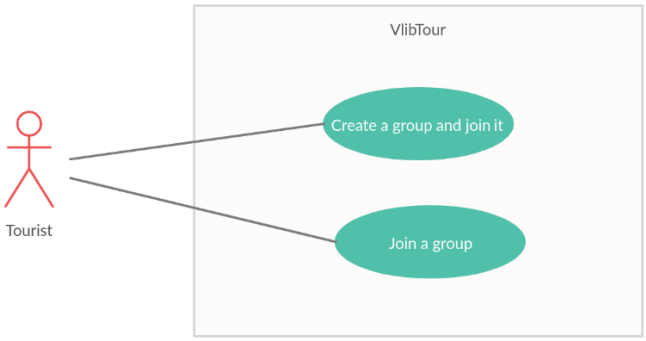

Figure 6: Cas d'utilisation du module Lobby Room