

Octochat  
*Le chat décentralisé*

Alexis GIRAUDET      Benjamin SIENTZOFF

1<sup>er</sup> décembre 2014

## Table des matières

<b>1</b>	<b>Utilisation et fonctionnement global</b>	<b>3</b>
1.1	Compilation du programme . . . . .	3
1.2	Utilisation . . . . .	3
1.3	Sous le capot . . . . .	3
<b>2</b>	<b>Patrons de conception</b>	<b>5</b>
2.1	Abstract factory . . . . .	5
2.2	Observer . . . . .	6
2.3	State . . . . .	7

## Introduction

Ce projet a été réalisé dans le cadre du cours *Objet et développement d'applications* dans lequel M. RICHOUX nous a enseigné l'utilisation des *Design Patterns*. L'ambition de ce projet ne s'arrête pas là car nous souhaitons poursuivre le développement de notre programme. Le sujet de notre projet est la création d'un client de chat décentralisé dans l'esprit du *peer to peer*.

Octochat, notre programme, est donc un client de chat qui n'a pas besoin de serveur central pour fonctionner. Lancer le programme, choisissez un nom d'utilisateur est c'est parti. Cependant, la mise en place d'une telle application n'est pas aisée. Ce rapport retrace la conception d'Octochat jusqu'au 26 novembre 2014. Il met en évidence les points qui ont posés problèmes et les différents *patterns* utilisés.

Dans une première partie, nous verrons comment compiler et lancer notre projet. Dans cette même partie nous présentons les principes et idées à la base du projet. Puis nous présenterons les patrons de conception utilisés.

# 1 Utilisation et fonctionnement global

## 1.1 Compilation du programme

**Compilation** Notre programme utilise deux bibliothèques à savoir la librairie standard du langage C++ ( *STL* et *Boost*, plus précisément :

- Boost.Build le système de compilation (équivalent de *make* et des *Makefile* mais plus portable)
- Boost.Thread pour les *threads* et les *mutex*
- Boost.Log pour les *logs* et le débogage
- Boost.Asio pour les entrées/sorties asynchrones sur le réseau
- Boost.Serialization pour (dé)sérialiser les données envoyées sur le réseau
- Boost.System pour les *Smart Pointers* et le *Lexical Cast*
- Boost.Uuid pour l'identification d'objets de manière unique

Pour compiler notre programme nous avons donc besoin d'un compilateur (incluant la *STL*) et d'installer *Boost*, c'est pourquoi nous avons réalisé un *Makefile* qui s'occupe d'installer *Boost* localement et de compiler le programme automatiquement. Une fois la compilation terminée, les exécutables sont placés dans le dossier *build* :

- octowatch* écoute le réseau
- octoglobalchat* propose de chatter avec toutes les paires connectées
- octochat* permet de chatter avec des utilisateurs

**Remarque** Il est possible d'installer *Boost* avec un gestionnaire de paquets à condition d'avoir les privilèges suffisants.

Pour compiler le projet, on commence par cloner le dépôt, puis on lance la commande *make* à la racine du projet.

```
$ git clone https://github.com/blasterbug/Octochat.git
$ cd Octochat
$ make
```

## 1.2 Utilisation

Si toutes les étapes précédentes se sont bien passées, vous êtes maintenant en mesure d'utiliser Octochat. Pour lancer l'application taper simplement *./build/octochat*

En réalité, trois exécutables ont été compilées, c'est expliqué précédemment. pour lancer tel ou tel exécutable, remplacer *octochat* par le nom de l'exécutable souhaité.

## 1.3 Sous le capot

Dès le début, nous avons décidé de diviser notre application en couches. Une première couche s'occupe de la gestion du réseau à proprement parler. Une seconde couche, qui pourrait être découpée elle aussi en deux parties s'occupe de l'aspect applicatif du programme.

**Couche Réseau** Lors du développement de la couche réseau, nous avons cherché à découplé le plus possible les applications clientes de l'application réseau c'est pourquoi nous avons utilisé le *pattern abstract factory*. En ce qui concerne la communication des données entre la couche réseau et les applications clientes, l'utilisation du *pattern observer* s'est révélé naturel.

**Couche applicatif** La couche applicatif permet de définir un protocole sur lequel OctoChat peut reposer permettant ainsi de gérer les utilisateurs d'un salon (une *octoroom*), leur connexion et leur déconnexion au sein des salons.

## 2 Patrons de conception

### 2.1 Abstract factory

Les pairs c'est à dire les instances de l'application réseau sont appelées *octopeer*, une *octopeer* est identifiée de manière unique grâce à un UUID généré à sa création. Une *octopeer* peut transmettre des données via une *octoquery*, une *octoquery* est composée d'une string contenant les données (données sérialisées) et d'une *octopeer* correspondant à l'émetteur ou bien au destinataire suivant que l'on envoie ou bien que l'on reçoit cette *octoquery*.

Dans notre cas, nous avons voulu faire rimer “décentralisé” avec “autonomie”, c'est pourquoi nous avons identifié deux services distincts capables de gérer les entrées/sorties des pairs.

- Le premier service, est le service d'exploration qui consiste à identifier des pairs potentielles en vue de communiquer avec elles. Nous avons appelé ce service *exploration\_server* (le mot serveur est un abus de langage mais dans ce cas le concept associé est plus intuitif). Un service d'exploration très simple pourrait tout simplement consister à lire un fichier contenant des informations de connexions sur les pairs (ip/port, nom d'hôte) ou bien de manière plus évoluée, envoyer un *broadcast* sur le réseau local et observer les réponses puis tester ces pairs potentielles grâce au second service.
- Le second service propose de tester si une pair se cache derrière les informations fournis par l'explorateur, puis de transmettre des *octoquery* aux pairs vérifiées, il s'agit du *communicator\_server*.

L'avantage de cette approche est l'indépendance envers les protocoles de communication, en effet une *octoquery* peut très bien être encapsulée dans un paquet TCP ou bien être transférée via HTTP, voire même fonctionner uniquement sur la boucle locale.

Pour définir précisément ses services et orienter leur implémentation nous avons utilisé le *pattern abstract factory*, comme on peut le voir à la figure 1.

**Remarque** Sur le schéma UML à la figure 1, X et Y sont des types génériques.

Nous avons donc implémenté *server\_factory* avec le protocole UDP pour explorer le réseau et le protocole TCP pour la communication entre les pairs.

***udp\_server*** La fonction *explore* se charge d'envoyer un *broadcast* contenant les informations de connexion (à savoir l'adresse IP et le port du serveur de communication), la fonction *run* se chargera donc de lancer le serveur qui réceptionnera ces informations pour les transmettre au *communicator\_server*.

***tcp\_server*** La fonction *check\_peer* se charge d'établir une connexion avec une potentielle *octopeer* (comme nous le verrons dans la partie sur le *pattern observer*). Si la connexion réussie alors les *octopeer\_observer* seront notifiés). La

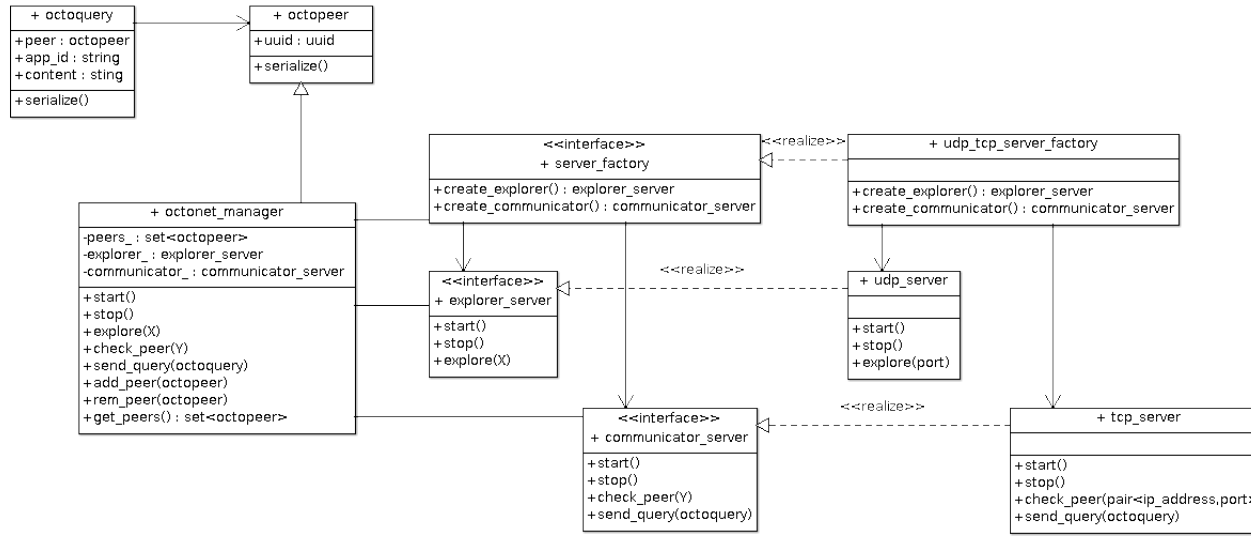


FIGURE 1 – Diagramme UML du *pattern abstract factory*

fonction *send\_query* permet d'envoyer une *octoquery* et *run* lance le serveur TCP qui recevra les *octoquery* (et notifiera les *octoquery\_observer*).

Nous avons donc implémenté *server\_factory* avec le protocole UDP/TCP, mais il est tout à fait possible d'utiliser d'autres protocoles pouvant utiliser les noms de domaine par exemple.

## 2.2 Observer

Ensuite pour interagir avec les applications nous avons naturellement utilisé le *design pattern observer*, visible à la figure 2.

En effet, les applications pourront être notifiées de la connexion ou déconnexion d'une *octopeer* en implémentant l'interface *octopeer\_observer*.

Ensuite les applications devront être notifiées de l'arrivée d'une nouvelle *octoquery*. Pour cela, nous avons ajouté un système de filtre très simple pour que les applications puissent communiquer avec leurs homologues distant, voire même avec d'autres applications.

Pour cela, nous avons ajouté un membre *app\_id* aux *octoquery* et les applications implémentant l'interface *octoquery\_observer* retournent leur propre *app\_id*. Donc, lorsqu'une *octoquery* est reçue, seules les applications ayant le même *app\_id* que l'*octoquery* sont notifiées (un *octoquery\_observer* retournant un *app\_id* vide sera toujours notifié).

Ce système de notifications permet donc à divers applications d'interagir ensemble, prenons par exemple une application de partage de fichiers et une application de chat : on pourrait très bien imaginer la possibilité d'utiliser ces

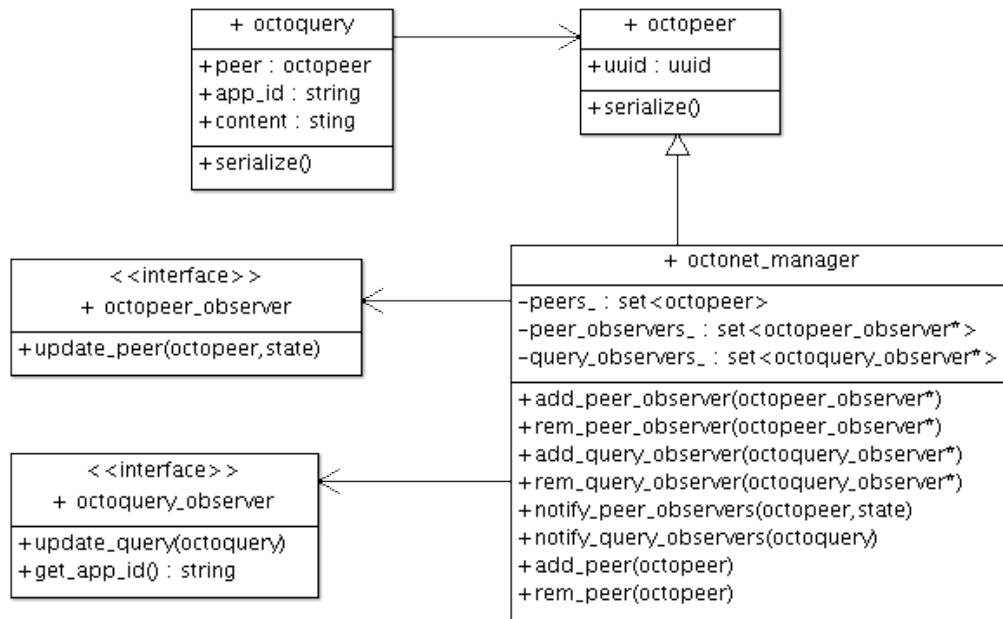


FIGURE 2 – Diagramme UML du *pattern Observer*

deux applications de manière complémentaire, c'est à dire pouvoir partager des fichiers dans le chat.

La classe *octonet\_manager* est donc à la fois la classe cliente du *pattern abstract factory* mais aussi la classe *observable* de l'*observer*. Nous avons donc rajouté la façade *octonet* pour masquer certaines fonctionnalités aux applications (notamment les fonctions de notification).

### 2.3 State

L'une des difficultés rencontrés lors de ce projet était la gestion des utilisateurs. Ces derniers doivent utiliser des pseudonymes uniques dans chaque salon. Le problème est que le salon que peut vouloir rejoindre un utilisateur se trouve sur plusieurs postes. Dans quel poste alors se connecter en premier ? Et si l'utilisateur a choisi un pseudonyme déjà utilisé, qui doit le prévenir ? Ces questions se règlent facilement en y réfléchissant un peu plus. L'utilisateur se connectera simultanément aux divers salons distants. Mais s'agissant d'une application réseau, il faut gérer la latence du réseau. La connexion et la déconnexion d'un utilisateur au sein d'un salon n'est pas instantané. On a alors introduit le *pattern state*. Ainsi, l'application a trois états.

Le premier état, **déconnecté**, est l'état dans lequel l'application démarre. Une fois que l'utilisateur a choisi un pseudonyme et qu'il se connecte, l'application passe dans l'état **attente**. Lorsque l'application rentre dans cet état, elle va chercher à se connecter aux pairs locales (si il y en a).

Deux cas de figure sont envisageables.

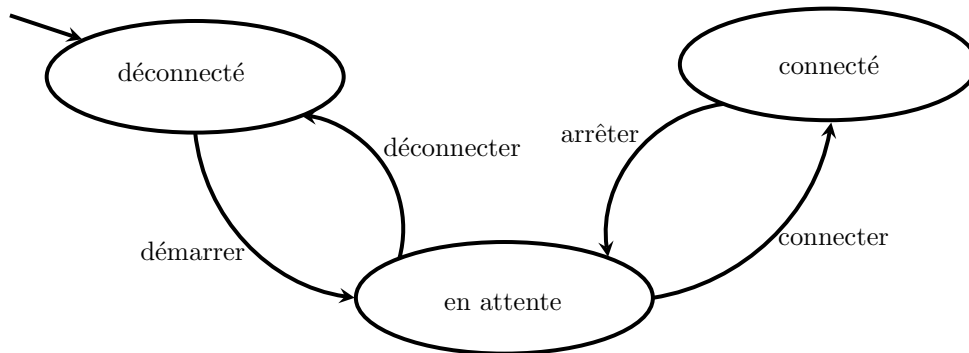


FIGURE 3 – Automate du comportement de *octosession*

- Le premier cas à considérer, le plus simple à traiter, est lorsque le utilisateur est effectivement connecté à un salon avec le pseudonyme choisi. Le programme est alors dans l'état **connecté**.
- Le second cas qui peut survenir est lorsque le nom de l'utilisateur est déjà utilisé. L'application retourne alors dans l'état **déconnecté** et l'utilisateur est invité à choisir un autre pseudonyme.

Le *pattern state* nous permet à présent de gérer la connexion des utilisateurs. D'autre part, on peut maintenant influencer sur le comportement de l'application. En effet, suivant les états du programme, des actions sont autorisées ou interdites à l'utilisateur. Le diagramme UML présentant le *pattern* est visible à la figure 4.

Le diagramme à la figure 4 représente l'automate traduit en terme de classes dans l'application. On a donc trois classes concrètes pour les trois états possibles.

*connected\_octostate* qui correspond à l'état **connecté**

*waiting\_state* qui correspond à l'état **attente**

*disconnected\_state* qui correspond à l'état **déconnecté**

Ces trois classes héritent de la classe abstraite *octostate*. Dans cette classe abstraite on y définit les fonctions membres qui correspondent aux transitions de l'automate, mais pas seulement. On y met également les fonctions de *octosession* qu'on souhaite activer ou désactiver selon l'état du programme. Le comportement par défaut de toutes ses méthodes consiste à ne rien faire (corps des fonctions vide). Pour activer telle ou telle fonction, on la redéfinit dans la classe de l'état approprié.

Dans la classe utilisatrice des états, *octosession*, on construit une instance de chaque état. Ensuite les sélecteurs adéquats nous permettent de récupérer ces instances lorsqu'un état veut franchir une transition.

Pour finir, un modificateur dans la classe *octosession* permet de changer l'état courant de l'objet (stocké dans un attribut). Seulement les états sont sensés utiliser cette fonction.



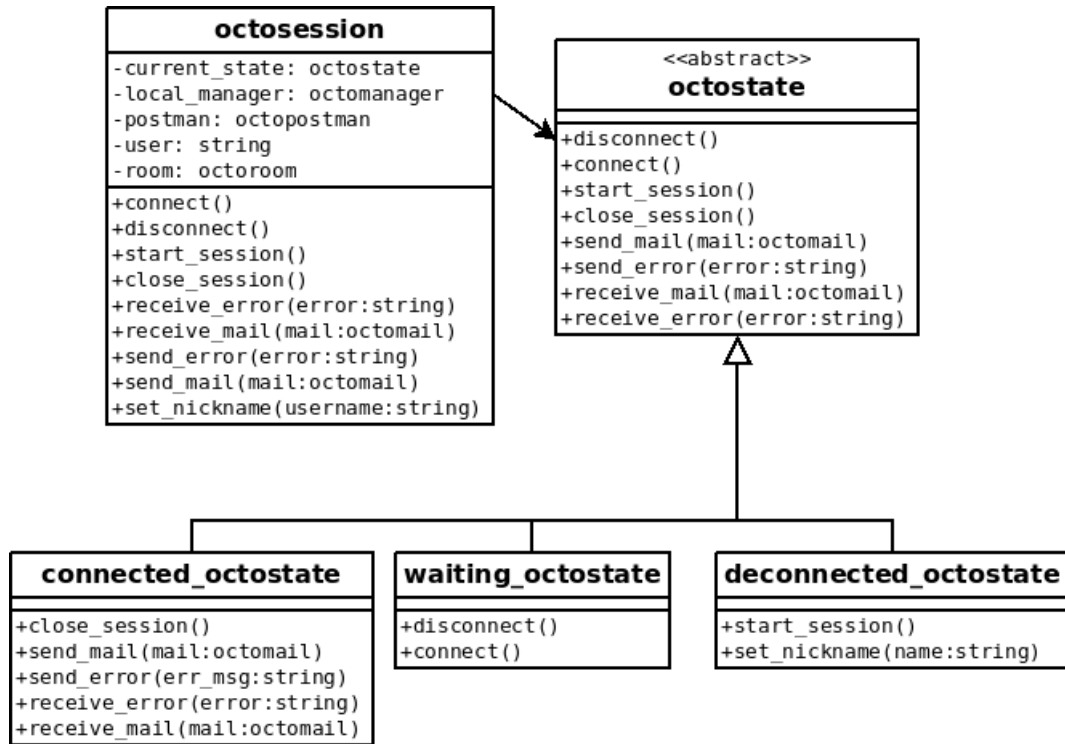


FIGURE 4 – Diagramme UML du *pattern state*

## Conclusion

La réalisation de ce projet nous a permis d'appliquer les notions vu en cours. L'utilisation de la librairie *Boost*, nous a permis de nous familiariser avec l'une des librairies les plus populaires existantes pour le C++.

Malgré les difficultés rencontrés, le projet ne fonctionnait pas comme nous l'espérions. En effet le développement d'une application réseau et l'assimilation de la librairie *Boost* ont été des obstacles significatifs. Cependant les briques du projet restent viables. En effet, l'exploration et la connexion aux pairs locales fonctionnent.

L'utilisation de patrons de conception a permis de rendre le projet modulaire. En particulier la conception par niveau d'abstraction rend possible l'utilisation du serveur avec d'autre type d'applications. Notre code peut donc être réutilisé. Des évolutions possibles sont déjà envisagées, comme un système de partage de fichiers.