Octochat
Le chat décentralisé

Alexis Giraudet Benjamin Sientzoff

27 novembre 2014

Table des matières

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 (*STL* et *Boost*, plus précisément :

Boost.Build le système de compilation (équivalant 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 `./octochat`

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 découper elle aussi en deux parties s'occupe des 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 application 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 reçoive cette *octoquery*.

Dans notre cas, nous avons voulu que "décentralisé" rime 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 communications, 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

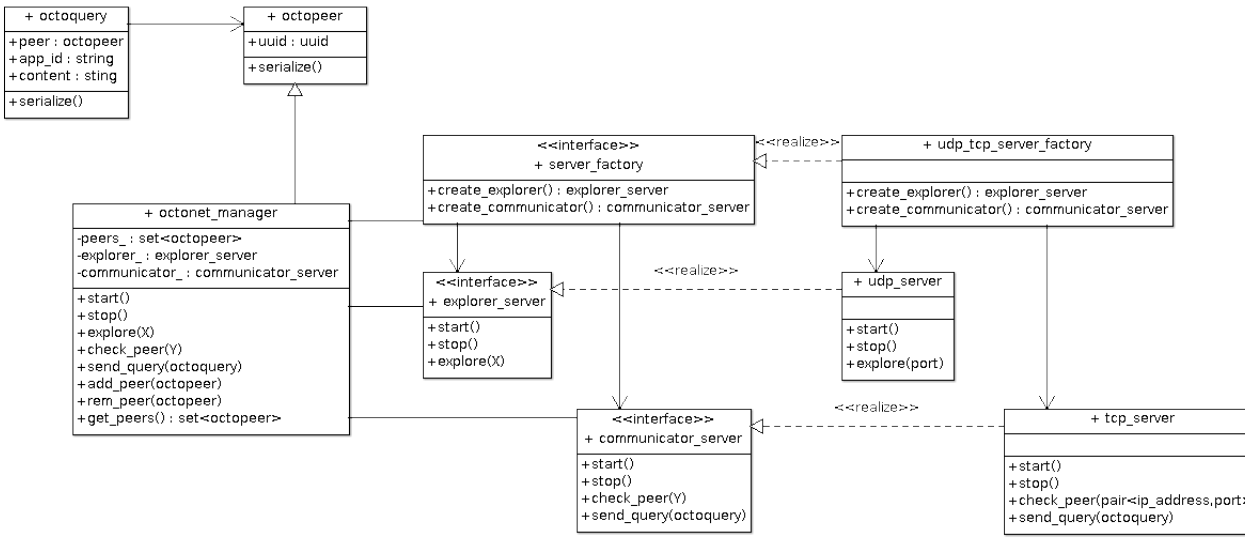

FIGURE 1 – Diagramme UML du *pattern abstract factory*

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 pourrons être notifiées de la connexion/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 rajouté un système de filtre très simple pour que les applications puissent communiquer avec leurs homologues distant voir même avec d'autres applications.
Pour cela, nous avons ajouté un membre *app_id* aux *octoquery* et les applications réalisant 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

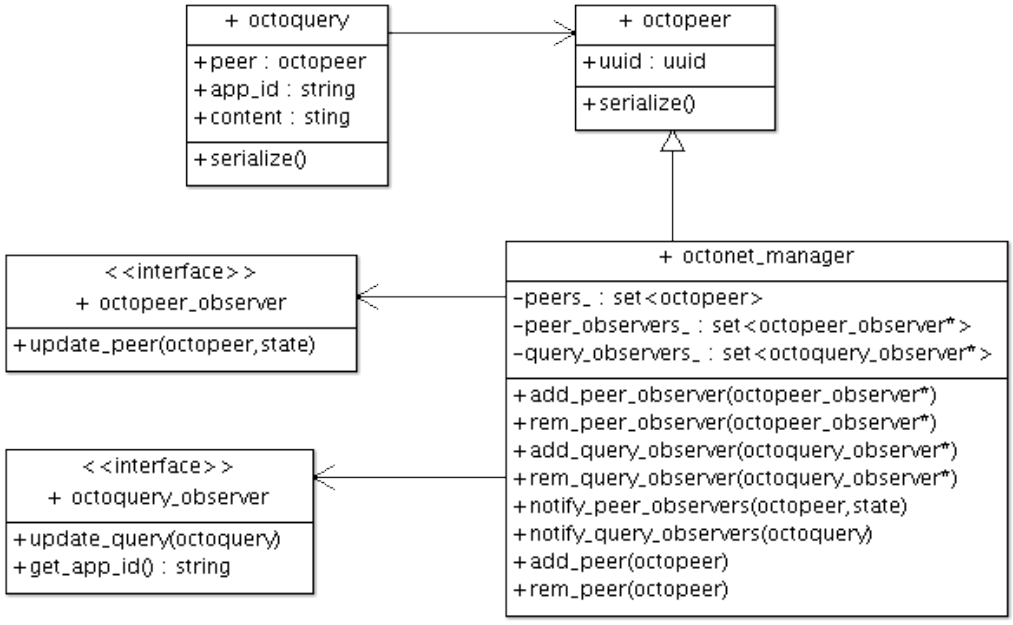

FIGURE 2 – Diagramme UML du *pattern Observer*

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 du *pattern 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 veut rejoindre un utilisateur se trouve sur plusieurs postes. Dans quel poste alors se connecter en premier ? Et dans le cas où le nom de l'utilisateur est pris, qui doit l'avertir ? Ces questions se règlent facilement en y réfléchissant un peu plus. L'utilisateur se connectera simultanément
Le *pattern state* nous a permis de gérer la connexion des utilisateurs. Le diagramme UML est visible à la figure 4.

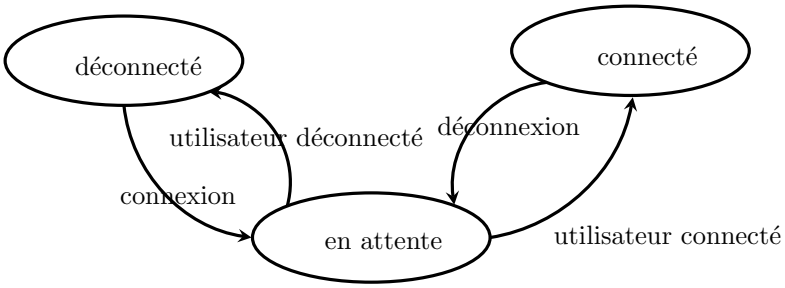

FIGURE 3 – Automate des transitions des états de *octosession*

Donc, une *octosession*, au démarrage de l'application, est dans l'état **déconnecté**. Ensuite, l'utilisateur donne un pseudonyme. La session passe alors dans l'état d'**attente**. Si il y a des pairs, qui font tourner *octochat* sur le réseau, une requête est envoyée à ces pairs avec le nom de nouvel utilisateur. Si le nom est pris, la pair qui est désignée comme *maître* envoie le message d'erreur correspondant. Sinon, une approbation est retourné à l'utilisateur qui est alors **connecté**.
Si il n'y a pas d'autre pair sur le réseau, la session passe directement dans l'état **connecté**.

On obtient alors le diagramme à la figure 4. On a donc trois états concrets, *connected_octostate*, *waiting_state* et *deconnected_octostate*. Ces états sont dans le programme transcrit en classes, c'est le principe même du *pattern state*. Ces trois classes héritent de la classe abstraite *octostate*. Dans la classe abstraite on y définit les fonctions membres qui correspondent aux transitions de l'automate. Le comportement par défaut de ces fonctions consiste à ne rien faire. Les classes qui utilisent ces transitions redéfinissent alors le comportement des fonctions.

Dans la classe utilisatrice des états, *octosession*, on construit une instance de chaque états. Ensuite les sélecteurs adéquats nous permettent de récupérer ces instances lorsqu'un état veut franchir une transitions.
Enfin, un modificateur dans la classe *octosession* permet de changer l'état courant de l'objet (stocké dans un attribut).

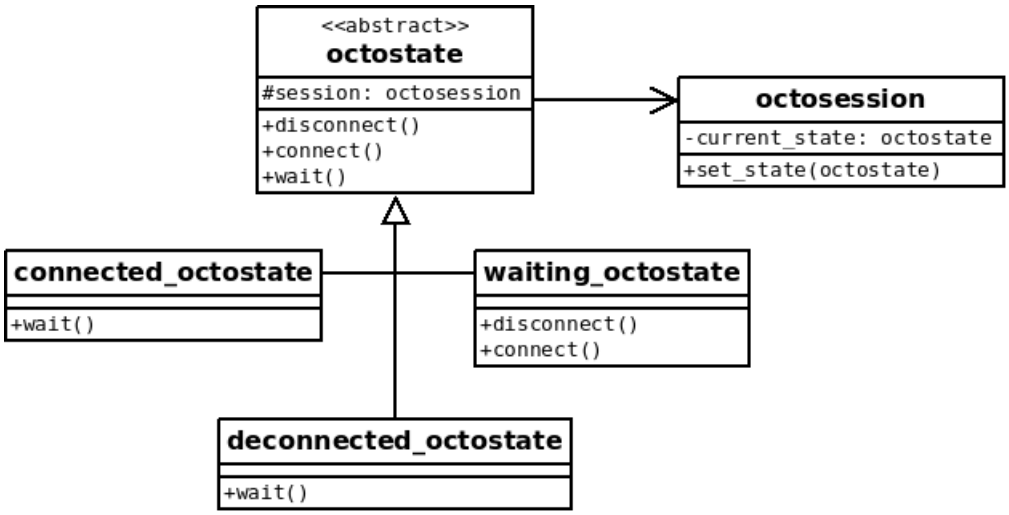

FIGURE 4 – Diagramme UML du *pattern state*

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 fonctionnement 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 modulable. 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.