Octochat
Le chat décentralisé

Alexis Giraudet Benjamin Sientzoff

26 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 poursuite le développement de notre programme. Le sujet de notre projet est la création d'un client de chat qui n'utilise pas de serveur principal comme c'est le cas pour ce genre d'application réseau. Le fonctionnement est détaillé plus loin.

Octochat, notre programme, est donc un client de chat qui n'a pas besoin de serveur 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.

1 Utilisation et fonctionnement global

Boost Pour des questions de dépendances et pour faciliter le développement du programme, notamment pour ce qui est du réseau, nous avons choisi d'utiliser la librairie *Boost*. L'utilisation de cette librairie nous permet également d'utiliser le système de compilation associé.

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 *thread]* et les *mutex*
Boost.Log pour les *logs*
Boost.Asio pour les entrées/sorties asynchrones sur le réseau
Boost.Serialization pour sérialiser les données envoyées sur le réseau
Boost.System pour les *Smart Pointers* et le *Lexical Cast*

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 précédentes étapes 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 C'est la partie du programme la plus critique et la plus difficile à mettre en œuvre. Le serveur, *octonet* qui tourne en local sur chaque client, utilise deux *threads*. Le premier permet d'écouter sur un premier port. L'écoute de ce port assure le maillage des pairs local. En effet, à chaque fois qu'un pair arrive, une tram réseau est envoyé sur ce port en *broadcast*. On peut ainsi maintenir une liste des pairs connectées. Le second *thread* écoute un second port qui permet de recevoir des requêtes réseau permettant de transmettre des données une fois qu'on connaît les paires locales. Ces requêtes permettent notamment à la couche de plus au niveau de mettre en place un protocole, *octochat protocol*. Un protocole spécialement conçu pour notre programme.

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.

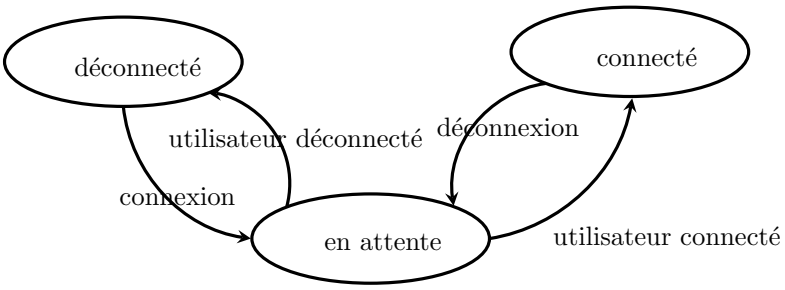

FIGURE 1 – Automate des transitions des états de *octosession*

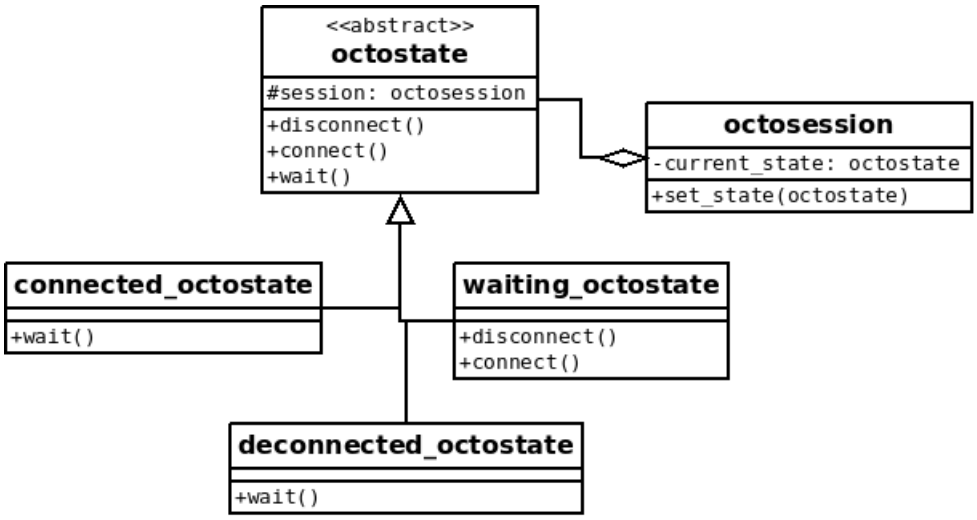

FIGURE 2 – Diagramme UML du *pattern state*

2 Patrons de conception

2.1 Observer

2.2 Factory method

2.3 State

L'une des difficultés rencontrés lors de ce projet était la gestion des utilisateurs. Ces derniers doivent utiliser des speudonymes unique dans chaque salon. Le problème est que le salon que veut rejoindre un utilisateur peut se trouver 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 ? Le *pattern state* nous a permis de gérer la connexion des utilisateurs. Le diagramme UML est visible à la figure 2.

Conclusion

je conclu