# Rapport sur le projet « UgeGreed »

Par Axel BELIN et Thomas VELU

# Introduction

## Présentation du rapport

Le rapport, ci-présent, a pour but de présenter, expliquer et développer le projet « UgeGreed » ainsi que son processus de développement.
Dans ce rapport, plusieurs points seront traités :

- Une présentation de l'application et de son utilisation ;
- Une explication sur l'architecture du projet ;
- Les différences entre la version avant la soutenance et la version finale du projet ;
- La charge de travail ;
- L'état actuel du projet.

## Présentation du projet

UgeGreed est un projet pédagogique qui permet à plusieurs applications de communiquer entre elles et de distribuer des charges de travail.
Ce projet est fait dans un but pédagogique, il n'est pas sécurisé contre les applications externes et ne possède aucun système d'authentification.

## Utilisation du projet

Après avoir installé et compilé le projet GitLab, l'utilisateur peut le lancer sous deux modes différents :

- En mode ROOT, avec la commande

« java UgeGreed <PORT D'ÉCOUTE> » ;

- En mode NOEUD, avec la commande

« java UgeGreed <PORT D'ÉCOUTE> <ADRESSE DE LA MACHINE DISTANTE> <PORT D'ÉCOUTE DE LA MACHINE DISTANTE> ».

Si la connexion est réussie, alors l'utilisateur peut entrer des commandes dans l'application :

- « DISCONNECT » qui permet à l'utilisateur de se déconnecter de la machine distante en envoyant un message de déconnexion. Si la machine qui se déconnecte possède des machines connectés, alors ces machines seront reconnectées à la machine distante ;
- « ROUTE » qui permet d'obtenir la table de routage de l'application.
- « START » qui permet de distribuer un calcul sur le réseau à partir des informations permettant de récupérer le jar, ainsi que d'un intervalle de valeurs à calculer.

# L’architecture du projet

L’architecture est basé sur plusieurs points :

- La RFC qui décrit le fonctionnement général du protocole et de l’application ;
- L’application qui possède une architecture simplifiée afin d’avoir un développement plus aisé.

## La RFC

La RFC contient des exemples sur le fonctionnement du protocole. Il y a actuellement 3 versions de la RFC du projet :

- V1 correspond à la version avant la soutenance ;
- V2 correspond à la RFC de secours du professeur Arnaud CARAYOL ;
- V3 correspond à la V2 mais modifié pour être en raccord avec notre développement.

## UgeGreed

La base de l’application. Cette classe permet de lancer le client/serveur et de se connecter à d’autres applications.

## Application

La classe qui permet de créer l’application en instanciant la console de l’utilisateur, la connexion avec la machine distante (ou non). La classe Application permet le fonctionnement complet de l’application.

## Console

La classe qui permet de représenter l’interface utilisateur de notre application. L’utilisateur peut entrer des commandes qui seront envoyées.

## RouteTable

RouteTable définit une table de routage. Elle permet de définir un chemin vers une machine destinataire. Toutes les applications possèdent une table de routage qui est mise à jour à chaque opération.

## ApplicationContext

Cette classe permet de définir la communication entre deux machines. Chaque connexion possède une ApplicationContext qui permet d'enregistrer ou d'envoyer des données.

## Le package Reader

Ce package regroupe tous les « Reader », c'est-à-dire, les lecteurs. Ces lecteurs permettent de lire les données envoyées par une machine distante. Ces données peuvent prendre la forme de différentes données comme un message simple à une table de routage.

## Le package Records

Ce package définit les « Record » de notre projet. La plupart de ces records sont utilisés dans les Reader. Ces records permettent de facilement définir la composition d'un message.

## Le package Test

Ce package regroupe les tests effectués sur les Reader. Les tests nous permettent de suivre une norme sur chaque Reader du projet afin qu'ils fonctionnent dans tous les cas.

## Le package PacketProcessors

Ce package est dû à un refactor d'Axel BELIN. Ce package permet une meilleur lisibilité des traitements des paquets reçus.

# Les différentes versions

## Avant le retour sur la RFC

Sur notre RFC (v1), le fonctionnement de protocole se basait sur la machine ROOT. Les paquets étaient envoyés au ROOT puis envoyé à la machine cible.

Le gros problème de notre RFC était qu'il y avait une table de routage implicite, à chaque nœud, nous demandions si le nœud cible était un voisin ou non.

De plus, la logique de la RFC faisait en sorte que les paquets parcourent en hauteur (de ROOT aux fils) et ne pouvaient pas revenir au ROOT.

# Après le retour sur la RFC

En nous basant sur la RFC du professeur CARAYOL (v2), nous avons implémenté en premier la connexion et la déconnexion.

Avant la soutenance, la connexion entre nœuds était possible, la déconnexion simple entre deux nœuds se faisait mais la reconnexion des enfants n'était pas au point. Les enfants ne pouvaient pas se reconnecter à la machine mère.

Lorsqu'il y a une reconnexion, il est nécessaire que tout le monde soit prévenu ! C'est pour cela qu'il est plus judicieux d'envoyer la table de routage du nœud déconnectant.

Il est plus judicieux de faire une Map dans la classe ROUTE TABLE qui lie un Id et un ApplicationContext. Ce sera plus facile pour parcourir les éléments et choisir à qui parler.

Et un autre problème est le cas des buffers plus grand que 1024 octets.

# Version finale

Dans la version finale, la partie connexion et déconnexion sont fonctionnelles : une machine peut se connecter et se déconnecter du réseau sans poser de problème à ses enfants. De plus, il est également possible d'assigner des calculs à une autre machine du réseau et de recevoir les résultats de ce calcul (dans une intervalle donnée). Cependant, la distribution de calculs entre plus de 2 machines du réseau n'est pas encore fonctionnelle. Par exemple, pour 3 machines connectées au réseau, si une machine souhaite distribuer un calcul, ce dernier sera effectué par une autre machine mais pas par les deux autres machines.

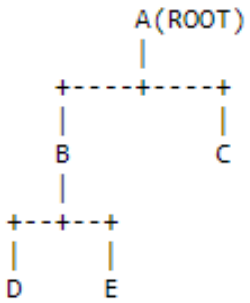
## Déconnexion

### 1. Ré-implémentation de la Table de routage

La table de routage est modélisée dans une table de hachage (HashMap) qui contient toutes les routes pour que la machine courante puisse communiquer avec n'importe quelle machine du réseau. La HashMap associe un identifiant de destination à l'identifiant intermédiaire par lequel la machine courante doit passer si elle veut envoyer un paquet.

De plus, nous avons utilisé une seconde table de hachage contenant toutes les instances de ApplicationContext permettant de communiquer avec les autres machines du réseau. Cette table associe un identifiant de destination à l'instance de ApplicationContext par laquelle la machine courante doit envoyer son paquet pour qu'il arrive à destination.

**Par exemple, soit le réseau suivant** :




La table de routage de D est la suivante :

D ➔ D

B ➔ B

A ➔ B

C ➔ B

E ➔ B

Si D veut envoyer un paquet à une autre machine, elle doit passer par la machine B. Elle devra donc récupérer dans sa table de routage l'objet ApplicationContext lui permettant de communiquer avec B via un socket TCP.

NB : L'identifiant de la mère de l'application courante est stocké dans un champ mutable. Cela permet de tester si un identifiant contenu dans la table est l'identifiant de la mère. Cet identifiant sera mis à jour si la mère de l'application se déconnecte du réseau.

En plus de stoker les routes, la table de routage offre les services suivants :

- Ajout d'une nouvelle route à partir d'un identifiant de destination, d'un identifiant intermédiaire et d'un objet ApplicationContext permettant que les 2 machines communiquent.
- Suppression d'une route vers une destination donnée
- Obtenir l'ensemble des identifiants des filles de l'application courante : on parcourt toutes les valeurs de la table de routage (il s'agit des machines que l'application peut atteindre directement). On élimine les doublons et on n'inclut pas l'identifiant de sa mère.
- Obtenir l'ensemble des identifiants des voisins de l'application courante : il s'agit de la même logique que pour les filles mais on inclut sa mère.
- Envoyer un paquet à une destination : on récupère l'objet ApplicationContext permettant de communiquer avec la destination donnée et on ajoute le paquet dans sa file d'envoi.
- Envoyer un paquet à tous les voisins : On récupère les objets ApplicationContext permettant de communiquer avec chacun des voisins. On ajoute le même paquet dans la file d'envoi de tous les objets. De plus, il est possible de définir un filtre de voisins à exclure, c'est-à-dire pour lesquels on ne veut pas envoyer le paquet.
- Envoyer un paquet à toutes les filles : il s'agit de la même logique que pour l'envoi aux voisins mais on n'inclut pas la mère de l'application.

- Remplacer toutes les occurrences d'un identifiant par un autre identifiant. Cela est utile pour mettre à jour la table de routage lorsqu'un paquet de déconnexion est reçu. L'inconvénient et qu'il est nécessaire de parcourir toutes les entrées de la table de hachage des routes, mais aussi de la table de hachage des contextes. En effet, il faut à la fois remplacer toutes les occurrences de l'identifiant de la machine qui se déconnecte ainsi que toutes les références de l'instance de l'objet ApplicationContext permettant de communiquer avec cette machine.

## 2. Implémentation du processus de déconnexion

Cette partie présente nos choix d'implémentation du processus de déconnexion d'une machine au sein d'un réseau.

NB : la déconnexion d'une application ROOT n'est pas gérée, elle n'a donc pas la possibilité de se déconnecter du réseau.

### 2.1 Émission d'une demande de déconnexion

Lorsqu'une application souhaite se déconnecter, elle envoie une demande de déconnexion à sa mère. Cette demande contient une liste d'identifiants de machines. Dans notre cas, il est nécessaire que l'identifiant de la machine souhaitant se déconnecter se trouve en première position de la liste d'identifiants. En effet, c'est une manière plus simple de savoir quelle machine souhaite se déconnecter. Ensuite, la liste contient les identifiants de toutes les filles de l'application souhaitant se déconnecter.

**Exemple** :

Soit la machine A qui possède 2 filles : B et C.

Si la machine A souhaite se déconnecter du réseau, alors elle enverra une demande à sa mère dans laquelle figurera la liste d'identifiants suivantes : [id_A, id_B, id_C]

### 2.2 Traitement d'une demande de déconnexion

Une application mère ne peut traiter qu'une seule demande de déconnexion à la fois. Si plusieurs applications filles souhaitent se déconnecter en même temps, une seule demande sera traitée par la mère. Pour les autres applications, c'est la responsabilité de l'utilisateur de relancer une nouvelle demande de déconnexion pour chacune de ces applications.

Lorsqu'une application mère reçoit une demande de déconnexion d'une de ses filles, elle stocke l'identifiant de la fille dans un champ mutable. Ainsi, l'identifiant sera conservé par la mère jusqu'à ce que la fille soit officiellement déconnectée du réseau. Il sera mis à jour lors de la prochaine déconnexion d'une machine. Dans le cas où la mère est déjà en train de se déconnecter, elle refusera toute demande de déconnexion de la part de ses filles. C'est à l'utilisateur de refaire une demande de déconnexion pour les machines dont la demande de déconnexion a été préalablement refusée car elles seront reconnectées à une nouvelle mère.

Dans tous les autres cas, la demande de déconnexion sera validée par la mère. Une fois cette demande validée, 2 cas sont possibles :

- Soit la fille souhaitant se déconnecter n'a pas de filles, dans ce cas :
    - La mère envoie un paquet de déconnexion à tous ces voisins pour signifier que la machine souhaitant se déconnecter doit être retirée des tables de routages. Ce paquet contient l'identifiant qui doit remplacer celui de la machine qui se déconnecte dans les

tables de routage des voisins. Ce nouvel identifiant est celui de la mère de la machine qui se déconnecte.

- La mère supprime la route vers la machine souhaitant se déconnecter de sa table de routage.

- Soit la fille souhaitant se déconnecter a au moins une fille, dans ce cas :
  - La mère stocke les identifiants de toutes les filles de l'application souhaitant se déconnecter afin de pouvoir attendre qu'elles se reconnectent. Ainsi, la mère sait combien de filles doivent se reconnecter à elle avant de confirmer la déconnexion de l'application souhaitant se déconnecter.

a. Demande de reconnexion des « orphelines »

Une fois que la demande de déconnexion a été acceptée par la mère, la machine envoie un paquet à ces filles contenant l'identifiant de leur nouvelle mère afin qu'elles puissent s'y reconnecter.

Pour se reconnecter à sa nouvelle mère, la machine fille doit :

- Ouvrir une nouvelle connexion TCP pour se connecter à l'adresse de socket de sa nouvelle mère et enregistrer le socket auprès de son selector.
- Créer un nouvel objet ApplicationContext qui sera rattaché à la Key permettant au selector de communiquer avec la nouvelle mère. L'ancien objet ApplicationContext permettant de communiqué avec l'ancienne mère est oublié.
- Ajouter une nouvelle route dans sa table de routage lui permettant de communiquer avec sa nouvelle mère : id_new_mother ➔ id_new_mother
- La table de routage associe cette route avec la nouvelle instance de AplicationContext afin de pouvoir faire communiquer les 2 machines.
- Avant de pouvoir envoyer un paquet de reconnexion à sa nouvelle mère, la fille doit s'assurer qu'elle y est bien connectée. Pour se faire, on attend que la connexion TCP soit établie avec la nouvelle mère et on passe la nouvelle Key en mode lecture et écriture afin d'envoyer le paquet de reconnexion et de pouvoir lire et traiter les futurs paquets envoyés par la nouvelle mère.

Le paquet de reconnexion contient : l'identifiant de la fille ainsi que la liste des identifiants de ses filles.

Une fois que la connexion a été établie et que le paquet de reconnexion a été envoyé à la nouvelle mère, la fille supprime la route permettant de communiquer avec son ancienne mère car elle a changé de mère.

### 2.3 Reconnexion des « orphelines »

Avant de confirmer la déconnexion de la machine souhaitant se déconnecter, la mère doit faire attention à ce qu'il n'y ai pas d'orphelines dans le réseau. Pour se faire, dès réception d'un paquet de reconnexion, la mère va commencer par vérifier si l'identifiant de la fille souhaitant se reconnecter est bien contenu dans l'ensemble des identifiants des machines devant se reconnecter. Si c'est le cas, la mère ajoute dans sa table de routage :

- Une route permettant de communiquer directement avec sa nouvelle fille : id_nouvelle_fille ➔ id_nouvelle_fille
- Pour toute les filles de sa nouvelle fille (petites filles), la mère ajoute une route permettant de communiquer avec chaque petite fille en passant par sa nouvelle fille :
- Pour toute les petites filles i : id_i ➔ id_nouvelle_fille
- La table de routage associe toutes ces nouvelles routes à l'objet ApplicationContext sur lequel la demande de reconnexion a été reçue. En effet, cet objet permet d'envoyer des paquets à la nouvelle fille.

Une fois la table de routage mise à jour, la mère retire l'identifiant de la fille des identifiants en attente de reconnexion car la nouvelle fille est maintenant proprement reconnectée.

Une fois que toutes les machines en attente de reconnexion se sont reconnectées à leur nouvelle mère, elle envoie un paquet de déconnexion à tous ces voisins pour signifier que la machine qui souhaitait se déconnecter doit être retirée des tables de routages. La mère supprime également la route vers la machine qui se déconnecte.

### 2.4 Gestion des paquets « Disconnected »

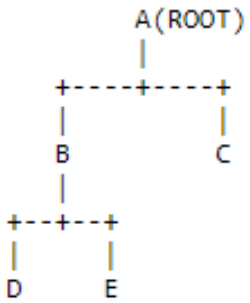
Lors de la version béta du projet, les paquets « Disconnected » n'étaient pas bien gérés, en particulier la table de routage n'était pas correctement mise à jour.

Dans cette version, les paquets « Disconnected » sont émis par la mère de la machine souhaitant se déconnecter pour signifier que la machine va se déconnecter du réseau et que toutes les tables de routage doivent être mises à jour pour remplacer toutes les références à la machine qui se déconnecte. Les paquets « Disconnected » sont de type Broadcast, c'est-à-dire qu'ils sont transmis à toutes les machines du réseau. Ce paquet contient 2 identifiants :

- L'identifiant qui doit remplacer celui de la machine qui se déconnecte dans les tables de routages : il s'agit de l'identifiant de la mère de la machine qui se déconnecte.
- L'identifiant de la machine qui se déconnecte. Cela permet d'indiquer quel est l'identifiant à remplacer dans les tables de routage.

Dès réception d'un paquet de déconnexion (« Disconnected ») la table de routage est mise à jour de la manière suivante :

- La route permettant de communiquer avec la machine qui se déconnecte est supprimée
- Toutes les routes pour lesquelles il faut passer par la machine qui se déconnecte sont modifiées : au lieu de passer par cette machine, il faut passer par sa mère.
    - Par exemple, soit le réseau suivant :




La machine B se déconnecte, avant réception d'un paquet de déconnexion, la table de routage de D est la suivante :

D ➔ D

B ➔ B

A ➔ B

C ➔ B

E ➔ B

A présent, D reçoit un paquet de déconnexion de B lui indiquant qu'il doit remplacer l'identifiant de B par l'identifiant de A dans sa table de routage :

- D supprime la route : B ➔ B
- D remplace toutes les occurrences de B par A dans sa table de routage :

  D ➔ D

  A ➔ A

  C ➔ A

  E ➔ A

Pour finir, une fois que l'application a mis à jour sa table de routage, elle retransmet le paquet de déconnexion à ses filles pour qu'elles fassent de même récursivement.

Si l'identifiant qu'il faut remplacer correspond à celui de la machine qui a reçu le paquet, alors cela signifie que la machine peut officiellement se déconnecter du réseau après avoir retransmis le paquet à ses filles. Elle va donc fermer sa connexion TCP et se suicider (elle interrompt son Thread).

## Répartition des Calculs

### 1. Algorithme de répartition des calculs sur le réseau

L'algorithme pour traiter les calculs est simple :

- La machine qui reçoit la demande de l'utilisateur va envoyer à un voisin une demande de disponibilité (combien il peut faire de calcul) ;
- Le voisin peut alors répondre positivement ou négativement à la demande, chaque machine a une charge de travail (1000 unités de calculs par défaut) ;
- La machine source va ensuite passer par un autre voisin (si le voisin précédent ne pouvait pas faire de calculs, la machine source va augmenter la charge de travail pour le prochain voisin).

## 2. Implémentation de la répartition des calculs

### 2.1 Réalisation des calculs

Les calculs assignés à une machine sont réalisés par une pool de Threads. Nous avons choisi d'utiliser l'API des ExecutorService permettant de soumettre des tâches à une pool de Threads de taille bornée. Nous avions déjà utilisé cette API dans le cadre des cours de concurrence, par conséquent, nous savions déjà l'utiliser. En principe, on soumet un ensemble de tâches à l'ExecutorService qui seront réparties entre plusieurs Threads. On attendra qu'elles soient terminées et on récupèrera leurs résultats dans le Thread courant.

Pour récupérer le résultat des calculs effectués par l'ExecutorService on appelle une méthode qui bloque jusqu'à l'arrivée du résultat. Afin d'éviter de bloquer le Thread principal de l'application avec l'appel de cette méthode nous avons décidé d'introduire un Thread supplémentaire dont le rôle est de soumettre tous les calculs assignés à l'ExecutorService, de récupérer leurs résultats et de les envoyer à la machine qui avait demandé le calcul dans des paquets Work Response. En effet, il pourrait être « dangereux » de bloquer le Thread principal qui fait tourner le selector car un calcul très long pourrait bloquer le selector pendant un long moment, ce qui n'est pas souhaitable. Par exemple, si le selector ne traite pas les paquets qu'on lui envoie, alors tout le réseau risque de se retrouver « paralysé » en attente de réponses de la machine courante.

### 2.2 Soumission des calculs au Thread qui réalise les calculs

Étant donné que les calculs sont réalisés et que leurs résultats sont renvoyés depuis un autre Thread, nous avons écrit une classe Thread-safe permettant de gérer :

- La soumission des demandes de calculs au Thread qui réalise les calculs
- L'envoi des résultats à la machine qui avait demandé le calcul

Le Thread qui réalise les calculs va se mettre en attente tant qu'il n'a pas reçu de demande de calcul. Dès lors qu'une demande a été reçue, la classe Thread-safe va l'enregistrer dans une structure de donnée partagée avant de réveiller le Thread des calculs afin qu'il effectue le calcul. Lorsque le Thread est réveillé, il récupère la demande, extrait le calcul à effectuer et le soumet à l'ExecutorService, puis bloque en attendant que le résultat du calcul arrive.

### 2.3 Envoi des résultats de calcul

Un calcul peut produire plusieurs résultats :

- Soit il termine correctement, dans ce cas un objet de type Result avec son résultat doit être envoyé à la machine qui avait demandé le calcul
- Soit il n'a pas pu être effectué en raison d'une erreur de téléchargement du jar, dans ce cas un objet de type Result indiquant l'erreur doit être envoyé à la machine qui avait demandé le calcul.
- Soit il n'a pas pu être terminé dans le temps imparti, dans ce cas un objet de type Result doit être renvoyé indiquant cette erreur.

Dans tous les cas, pour chaque Response calculée, le même type de paquet Work Response est renvoyé à la machine ayant demandé le calcul.

# La charge de travail

## Axel BELIN

Axel BELIN a été en charge d'implémenter ConnectReader, DisconnectReader, le package packetsProcessors, le refactoring de ApplicationContext, la classe Console, la réparation de la classe de RouteTable, et, avec l'aide de Thomas VELU, la classe Worker.

## Thomas VELU

Thomas VELU a été en charge de développer le restant des Reader, de la classe Application, un squelette de la classe Disconnector, JarHandler et, avec l'aide de Axel BELIN, la rédaction de la RFC et du rapport.

# L'état actuel du projet

## Fonctionnalités présentes

- La connexion entre machines ;
- La déconnexion entre machines ;
- La reconnexion des machines filles ;
- La mise à jour de la table de routage ;
- L'envoi d'une « work request » entre deux machines ;
- Le calcul d'une « work request » et obtenir sa réponse ;
- Recevoir une réponse : « work response » contenant la réponse d'un calcul effectué par une machine assignée.

## Fonctionnalités manquantes

- L'envoi d'un calcul à travers plusieurs machines ;
- Le partage équitable du calcul entre plusieurs machines;
- L'agrégation des résultats dans un fichier texte;
- La gestion des messages lourds (plus de 1024 octets).