# Rapport Projet Antroid
# Programmation Comparée

*Equipe MetaNoik*

# Introduction

Le problème posé dans notre projet était de pouvoir gérer l'évolution de fourmis sur une carte. Celles-ci peuvent se déplacer partout sur la carte, chercher de la nourriture et attaquer d'autres fourmis.

L'organisation du travail des fourmis est un modèle de relation entre agents d'une simplicité impressionnante. Chaque agent, representé par une fourmi, effectue une action simple et transmet son information aux autres agents afin de converger vers un but commun.

Description des contraintes:

Un des aspects les plus importants de ce projet était de respecter certaines contraintes. Celles que nous avons aléatoirement tirées sont :

- Utiliser un modèle agents-acteurs
- Minimiser les effets de bord

Ces deux contraintes étant assez contradictoires, nous avons dû faire preuve d'ingeniosité pour en venir à bout.

# Choix de langage

La contrainte agent-acteur nous imposait un langage dans un panel assez réduit de langages existants. Les deux principaux langages pour le modèle en question sont Erlang et Akka.

Nous avons choisi Akka. Cet outil très puissant est utilisable soit avec le langage Java soit avec Scala. Nous avons donc choisi d'utiliser le langage Scala que nous le connaissions déjà. De plus Scala est un langage qui permet facilement de minimiser les effets de bords contrairement à Java. Notons toutefois que Erlang semblait offrir des possibilités intéressantes. En tant que langage fonctionnel pur, il nous aurait peut-etre permis de respecter la seconde contrainte encore plus facilement.

# Choix d'implémentation

Le modèle agent répond à une problèmatique de mise à l'échelle. Chaque foncitonnalité du logiciel est representé par un acteur. Un acteur est une entité indépendante des autres.
La communication entre ces agents est assurée via des messages.
Il offre aussi des perspectives intéressantes pour la parallélisation des taches.
Ainsi, la division du travail est optimale et chaque agent effectue le minimum d'opérations nécessaires. On se rapproche d'un système décentralisé comme une réseau de fourmis, par exemple.

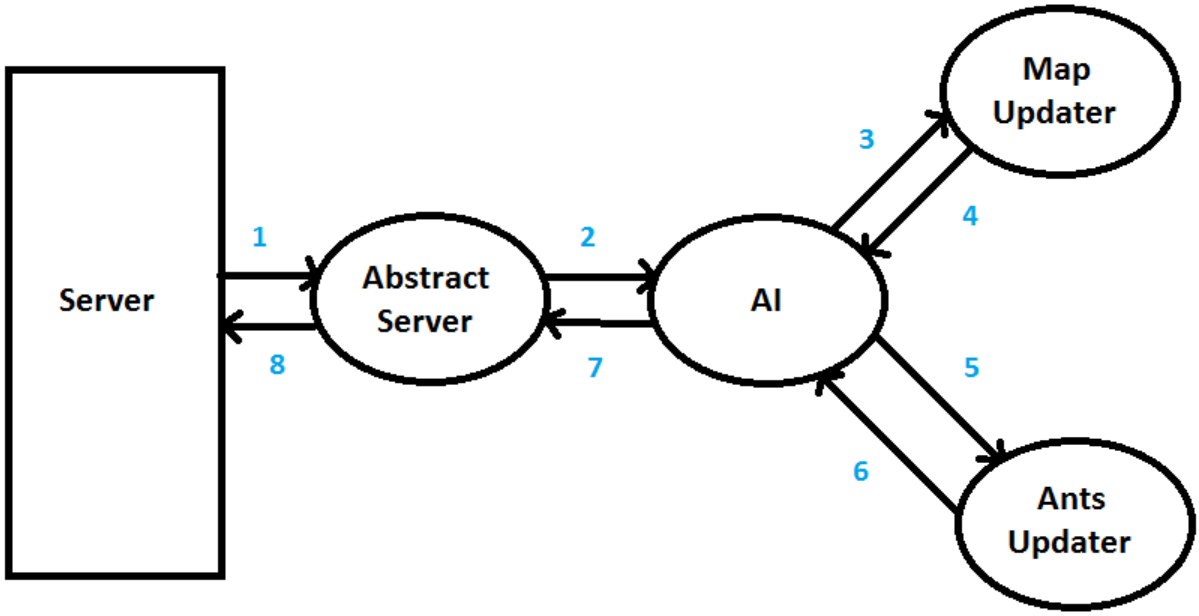
Dans une première approche, nous voulions abstraire le fait que la prise d'action était centralisée et nous avons pensé que notre code serait plus modulable si nous avions un agent par fourmi. Cette approche nous laissait le choix entre une IA globale et une intelligence pour chaque fourmi.

Malheureusement, nous nous sommes vite rendus compte que ce choix nous obligait à une recentralisation avant l'envoi au serveur et un split du message du serveur vers chaque fourmi. Cette optique se révelait peu efficace et impossible à mettre en place avec un système multi agents tel que nous le visions. Nous avons donc abandonné cette idée.
Nous avons opté pour une intelligence centrale communiquant directement avec notre abstraction du serveur.

<u>Description de l'implémentation:</u>

Tout d'abord nous avons implémenté une classe qui représente une abstraction du serveur. Nous lui envoyons des messages qui sont transformés en commandes compréhensibles par le serveur. Cela nous permet d'utiliser notre abstraction du langage en interne.
De plus, nous avons découpé notre programme en plusieurs agents:

- L'IA qui prend les décisions et envoie au serveur les commandes à éxécuter. Elle distribue les calculs à différents agents tels que :
- MapUpdater (met à jour la représentation de la carte dans les messages)
- AntUpdater (met à jour l'état des fourmis dans les messages)




# Absence d'états, vous dites ?

Nous avons contourné l'absence d'état interne en utilisant les messages de akka. Ceux-ci contiennent toutes les informations dont nous avons besoin et sont déconstruits par chacun des agents qui le lit. L'agent reconstruit ensuite un nouveau message qu'il retourne à un autre agent.
Avec cette approche par message, et l'absence d'état interne, nous avons été forcé de synchroniser les communications. Les messages sont envoyés par l'IA dans un sens défini. Elle envoie le message à l'agent A, attend le retour, puis renvoie ce message modifié à l'agent B, etc … Après la réponse du dernier agent, elle peut enfin prendre sa décision.
On voit dès lors la limite de la combinaison de nos contraintes. On ne peut en aucun cas envisager une véritable parallélisation.

Notons tout de même que notre contrainte de restriction des effets de bord a surtout été respectée pour tout ce qui concerne les fourmis et leurs algorithme. La partie communication avec le serveur n'a pas été sujet à ces contraintes.

# Les communications entre agents

Dans les différents fichiers scala, nous avons défini une API des messages qui circulent au sein de notre programme. Ceux-ci sont d'abord une abstraction des commandes du serveur et ensuite, les messages associant la communication entre les différents acteurs. Pour rappel, ces messages doivent etre séquentialisés, et c'est embetant pour délivrer toute la puissance du systeme multi agent.