8INF844 - Système multi-agents

Rapport du TP1

Simulation de la bourse
semestre Hiver 2014

Florian THOMASSIN

Sommaire:

1. Choix techniques

Ce TP a été fait en C++11 sous Linux et utilise CMake comme toolchain de compilation.
Le projet nécéssite les libs Boost1.54:
-> boost::bind
-> boost::asio
-> boost::lexical_cast
-> boost::algorithm::string

La compilation se fait en C++11 car nous utilisions des std::mutex et d'autres fonctionalités du nouveau standard.

Le projet utilise aussi les bibliotèques standard C/C++. De plus, le projet utilise firefox pour afficher l'interface utilisateur. Elle a été développé en HTML5/JQUERY.

Le programme contient un server qui tourne en local et utilise le port 8080 pour communiquer. Il utilise aussi 4 threads au minimum, 3 pour les agents (clients), et 1 + le nombre de requettes en cours pour l'agent server.

La compilation se fait via la commande **cmake . && make.** Pour lancer le projet, il suffit de faire la commande ./start.sh (il faut avoir la version g++ 4.8 ou clang++ 3.7). Pour compiler avec clang++, la commande de compilation est **CXX=clang++ cmake . && make**.

Le projet est versionné avec GitHub.

2. Réalisation du projet

Cette simulation de bourse contient actuellement 3 agents qui font office de trader et l'environnement qui est assimilé à un agent. On peut le considérer comme un arbitre.

Le propre des systèmes multi-agents est la distributivité, c'est pour cela que le travail réalisé marche sous forme de client/serveur et les agents communiquent en envoyant des requètes sur un port détérminé. (Je n'ai pas eu le temps de tester le projet sur plusieurs machines mais théoriquement c'est possible).

Ici, c'est l'arbitre qui représente le serveur car c'est le point central. Toutes les transactions passent par lui. Il fonctionne de la manière suivante :

> Ouverture en lecture / écriture du port 8080 sur l'adresse IP:127.0.0.1
  > Ecoute du port 8080
  > Si une requète ce présente, l'agent lance un thread pour la traiter

> Mise à jour de l'interface utilisateur après traitement de la requète
> L'agent se stop uniquement s'il reçoit un SIGKILL de la part de l'utilisateur

Les requètes vont être parsée pour savoir de qui elles viennent et déterminer s'il s'agit d'une demande d'information où d'un ordre d'achat/vente. Dans le cas d'une demande d'information, l'agent va retourner l'information demandée (prix d'une action, stock disponible). Si la commande parsée est un ordre alors l'agent renvoie le statut de la transaction (bien passée ou non).

Afin de faire évoluer les cours et les stocks, tout ordre d'achat ou de vente va modifier à la hausse ou à la baisse les taux.

Les agents traders, quand à eux n'ont qu'un champs de vision très limité. Ils ne peuvent connaître le prix ou la quantité d'action disponible qu'en discutant avec l'agent arbitre (serveur).

Ils sont aussi caractérisés par un objectif à atteindre et un état mental. Ainsi un agent peut soit être contant, soit triste, soit fou. L'objectif de l'agent est lié à son état mental, ainsi s'il devient fou, il fera des transactions de façon aléatoire.

Il est toujours possible de sortir d'un état pour aller dans un autre grâce au facteur chance qui se modélise avec des modifications aléatoire de l'humeur.

2.1 Détail des agents traders

Les agents traders héritent de la classe agent qui possède des paramètres communs à tous.

Cette classe générique permet de stocker les informations suivantes:
> Le nom
> Les actions possédés
> L'argent disponible
> Le coefficient de bonheur et de folie
> Le statuts et l'objectif

L'agent générique a besoin d'un identifiant, de quantités par défauts pour les actions possédées et une quantité d'argent. Il possède 3 fonctions "getCotation" qui permet de récupérer la valeur d'une action et "getQuantity" qui récupère la quantité d'une action sur le marché.
De plus cette classe mère possède une fonction virtuel pur qui va varier pour chaque agents.

Les agents vont changer de comportement grâce au graphe de décisions suivant. Pseudo code:

```
if objectif == BUY
{
    happy += 7 && crazy++
    if buy action == ok
       happy++ && wallet-- && stock++
    else
      happy -= 5
}
else if objectif == SELL
{
    happy += 10 && crazy++
    sell action
}
else if objectif == STOCK
{
    happy -= 5 && crazy++
    buy best action to stock
}
else if objectif == RANDOM
```

```
{
    happy++ && crazy += 10
    buy or sell action (random choice)
}
else if objectif == STANDBY
{
    do nothing

}

if (old_wallet < wallet)
    happy += 5


if happy == 0
{
   status = SAD;
   objectif = STANDBY
}
else if crazy == 0 && happy !=0
{
   status = SAD
   objectif == STOCK
}
else if happy > crazy
{
   status = HAPPY
   objectif == BUY or SELL
}
else if crazy > happy
{
   status = CRAZY;
   objectif = RANDOM;
}
```

Grâce à cela le comportement de l'agent va être définit par la sa pseudo humeur.

Avoir le statut SAD va rendre l'agent moins efficace et donne accès aux objectifs STOCK, STANDBY
Avoir le statut HAPPY va rendre l'agent plus efficace et donne accès aux objectifs BUY et SELL
Avoir le statut CRAZY va rendre l'agent fou et automatiquement le mettre dans une situation de RANDOM

2.2 Détails de l'environnement (le serveur)

L'environnement est associé à un agent particulier car il ne dépends pas de la classe mère Agent.hpp mais va être entièrement autonome.

Sa composition et son fonctionnement est complexe car c'est un serveur qui parse les requettes des autres agents et les traites.

Il possède les paramètres suivant:
> La liste des cours des actions
> La liste des fichier pour intéragire avec l'UI
> Un mutex pour sécuriser les écritures
> Un io service boost::asio pour écouter le port de réception

Il utilise les fonctions suivante:
> Un main qui va initialiser et lancer le server
> "Serveur" qui va s'occuper d'écouter et de parser les requêtes

> “get_cotation” qui va permettre de répondre à une question sur le prix d’une action
> “get_quantity” qui va répondre à une question sur les stocks d’action
> “sellprocess” va effectuer un ordre de vente d’un agent
> “buyprocess” va effectuer un ordre d’achat d’un agent

De plus le serveur va en permanence mettre à jour les fichiers qui stockent les informations sur les actions et l’historique en permanence.

3. UI

L’interface graphique est là pour montrer l’évolution des 5 cours grâce à deux graphiques, l’un représentant l’évolution de prix sur 7 jours et l’autre l’évolution de la quantité de stocks d’actions.

Grâce aux logs générés l’UI va pouvoir afficher la liste des 30 dernières requètes effectués sur le server, c’est à dire la bourse, et les informations sur les 3 courtiers.

4. Difficultées rencontrées

Les difficultés auxquels j’ai dû faire face n’étaient pas liées à la modélisation théorique (choix des agents, type d'environnement, champs de vision des agents, etc) mais lors de la réalisation technique.

Ma volonté de faire un système client / serveur et donc distributable sur plusieurs machines m’a forcé à poser des limites sur les différentes types d’interaction.

Une autre difficulté majeure a été la gestion des mutex des clients et des serveurs. La calibration des valeurs initiales a été très compliqué aussi car les agents courtier se bloquent facilement dans des routines. Le déterminisme est très compliqué à casser.

5. Améliorations possibles

De nombreuses améliorations sont possibles mais les plus importante selon moi sont:

> Mettre de la documentation doxygen
> Passage du serveur en HTTPv3.
> Meilleure gestion d’erreur.
> Ajouter de la généricité avec des templates variadic pour les agents.
> Ajouter une base de données type SQLite pour enregistrer les données du marché et des agents.
> Refaire l’UI avec du PHP, RubyOnRails pour intéragir directement avec la BDD.
> Rendre ergonomique l’UI et engager un graphiste pour l’embellir.