PROJET PACMAN

Système multi - agents

PRINTEMPS 2017

UV IA04

Magnin | Durocher | Seguin | Lamrous

<u>Objectif :</u> Dans le cadre de l'UV IA04, le projet du semestre consistait à développer entièrement un système multi-agent (SMA) pour mettre en application concrète la théorie et la pratique vue au cours du semestre. L'objectif étant que le système fonctionne de manière asynchrone et autonome afin que les agents construits communiquent, négocient et interagissent entre eux pour effectuer les tâches qui leurs sont délégués.

Introduction

Nous avons choisi de développer un jeu de plateforme similaire à un Pacman. Ce jeu de plateforme est complétement autonome et intelligent. Le joueur est une intelligence artificielle qui se déplace dans un environnement et cherche à survivre le plus longtemps possible. L'environnement est, quant à lui, défini par une grille à travers laquelle des monstres prédateurs se baladent de manière aléatoire.

Nous avons fait ce choix car nous cherchions à appliquer aux mieux la théorie vu en cours tout en cherchant une application concrète à cette dernière. Notre projet Pacman est l'implémentation de ce système multi-agents qui mixe les concepts théoriques de la plateforme jade avec intelligence artificielle et théorie du jeu.

Plan

Dans ce rapport, nous commencerons par une analyse informelle qui permettra au lecteur de se mettre en situation et de comprendre les questions que nous nous sommes posés lors de notre phase d'analyse. Cette première partie sera ainsi composée d'une définition de nos cas d'usage, de nos agents, de leurs fonctionnalités ainsi que de leurs accointances. Nous dédierons ensuite une seconde partie à la phase de conception, au travers de laquelle nous détaillerons plus en détail le protocole de communication le plus complexe que nous avons implémenté, pour répondre à notre problématique d'intelligence artificielle indépendante. Finalement la dernière partie présentera une mini démonstration de notre 'pacman' multi-agent.

Phase d'analyse

Rapidement, nous nous sommes rendus compte que la distinction des entités de notre SMA et la manière dont elles allaient interagir entre elles étaient l'enjeux cruciale de notre système.

Remarque : Nous avons choisi des noms anglais pour nos agents afin de rendre ce projet public. Nous garderons les mêmes nominations dans le rapport dans un soucis de cohérence.

Règle du jeu

La première étape était d'établir les règles du jeu de notre plateforme avant de réfléchir en termes d'agent ou de plateforme jade.

L'environnement est une grille dans laquelle évolue des monsters et un traveler. L'objectif du traveler est de survivre le plus longtemps possible. Si un monster et un traveler se rencontrent, le traveler meurt et la partie est finie. Nous avons trouvé amusant d'ajouter des obstacles dans la grille, infranchissables par toutes entités mobiles du système.

Réflexions

Voici les principales problématiques que nous avons été amené à palier durant la phase d'analyse.

a. Animation

Une des réflexions était la manière d'animer notre jeu. En effet, nous voulions que notre système soit entièrement autonome après son exécution. Cela implique une entité capable de 'faire tourner' la plateforme continuellement, jusqu'à l'arrêt de la partie.

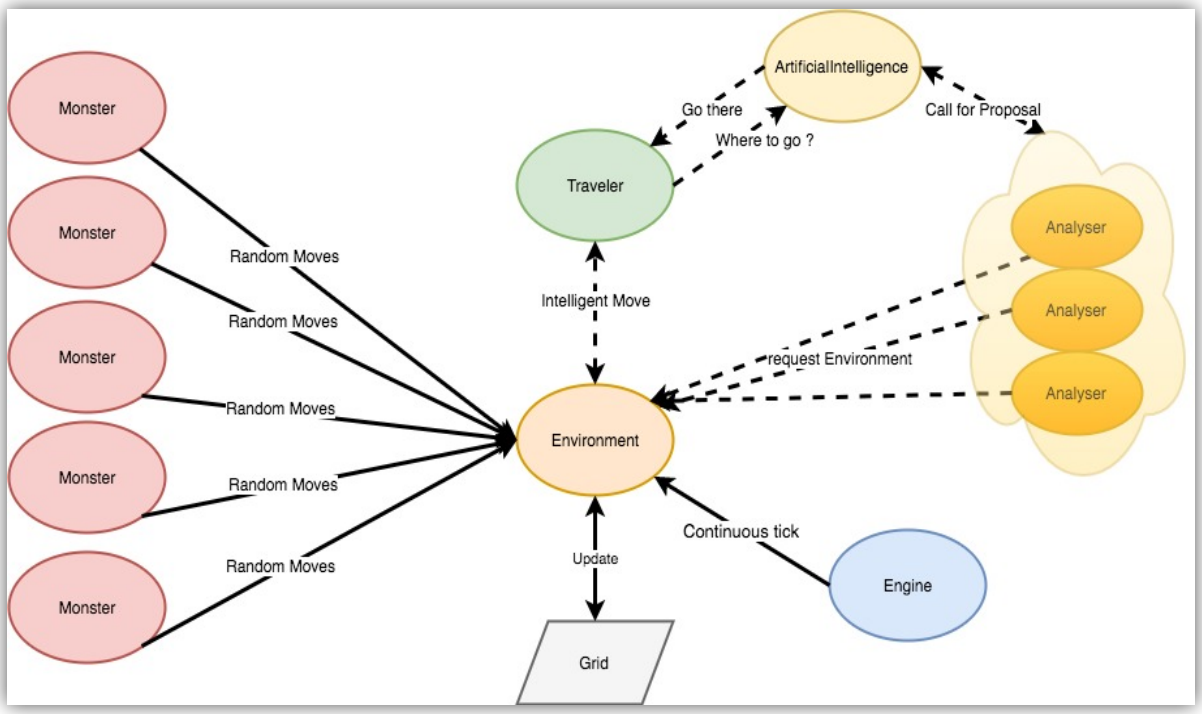
b. Relation traveler - plateforme

Une seconde réflexion s'est posé sur la relation IA(traveler)/plateforme. Notre traveler, guidé artificiellement dans notre système, devait correspondre à une entité indépendante du reste. En effet, nous voulions que notre système soit architecturé de telle sorte que notre IA puisse être aisément remplacé par un réel joueur humain. Nous nous sommes imposés cette contrainte car nous trouvions intéressant de réfléchir à une extension et une scalabilité de notre plateforme pour un développement post projet IA04.

Remarque : Nous détaillerons l'implémentation du moteur qui anime l'IA par la suite.

Cas d'usages

Les réflexions précédentes et nos observations d'un jeu de plateforme de type Pacman, entre autres, nous ont permis d'établir les cas d'usage de notre système afin que le jeu fonctionne correctement. Nous les résumerons dans ce diagramme de cas d'usages / interactions (en anglais à nouveau).



Justifications

Engine permet d'animer l'Environment à l'aide d'un tick continu. L'Environment est l'élement central du SMA. Ce dernier réagit au tick de Engine pour animer les Monsters. Les Monsters bougent alors aléatoirement en fonction de leur position.

De son côté le Traveler, indépendant de l'Environment et donc du tick de Engine se déplace à son propre rythme. Pour se faire, il consulte ArtificialIntelligence qui lui renvoit la meilleure position où aller. La décision de ArtificialIntelligence est le fruit d'une discussion avec les Analysers sur les positions présentes et éventuelles futures des Monsters dans la Grid.

Environment met alors à jour la Grid si le déplacement d'une entité mobile est valide, ou arrête la partie si le déplacement d'une entité mobile entraîne la rencontre du Traveler avec l'un des monstres.

Maintenant que nos cas d'usages relativement détaillés ont été fixés nous allons définir nos différents agents plus en détail.

Agents

Nous présenterons nos agents dans une table de Fonctionnalités.

Type d'agent	Fonctionnalités
Agent Monster	• S'enregistre auprès de Engine • Se déplace aléatoirement sous la demande de Environment
Agent Analyser	• S'enregistre auprès de Artificial Intelligence • Demande la position du monstre qu'il analyse à Environment • Evalue les positions possibles du monstre à 3 coups près • Retourne son analyse à Artificial Intelligence
Agent Artificial Intelligence	• Demande les positions analysées aux Analyser sous la demande de Traveler • Retourne la meilleure position à Traveler, après analyse.
Agent Traveler	• Possède son propre Tick d'animation • Demande à AI le meilleur déplacement à chaque tick • Envoie sa position à Environment
Agent Environment	• Initialise l'environnement de jeu (Grid) • Réagit au tick de Engine pour demander aux Monsters de bouger • Attend les réponses des entités mobiles continuellement pour mettre à jour sa Grid. • Affiche la Grid. • En cas de fin de partie, doit notifier les agents en cours.
Agent Engine	• Possède son propre tick • Enregistre les Monsters • Transmet à chaque tick les infos des Monsters à Environment

Maintenant que les fonctionnalités ont été posées nous allons les traduire en behaviour respectifs, et nous expliquerons plus en détails la tâche de l'agent à laquelle ils correspondent

Behaviours

Voici le détail des différents behaviour de chaque agent. Remarquons que lors de leur création, chaque agent de notre système s'enregistre auprès de l'agent DF afin de simplifier les communications. En effet, chaque agent peut ainsi accéder, via DF, à l'AID d'un autre agent du système. Nous avons aussi défini des constantes comme conversationID entre les agents. Elles permettront de filtrer les échanges entre les entités pour s'assurer que les messages soient captés et interprétés de la bonne manière (c.a.d par les bons behaviours)

Agent Monster

Chaque agent Monster possède les attributs :

- value (qui l'identifiera des autres)
- position (qui correspond à la Cellule de la Grille sur laquelle il est après s'être déplacé)
- oldPositon (qui correspond à la Cellule sur laquelle il était avant de s'être déplacé)

SubscribeToEngineBehaviour Lors de sa création, l'agent Monster envoie un message de performatif SUBSCRIBE à l'agent Engine. Cela permettra à Engine d'accéder à ses informations en appliquant *getReceiver*() sur le message reçu. **OneShotBehaviour**.

MoveBehaviour Capte les messages envoyés par Environment. Les messages doivent posséder le template adéquat (conversationID et performatif REQUEST). Lors de la reception, ce behaviour déclenche alors la méthode *move*() du Monster. La méthode va modifier les attributs position et oldPosition. Ce sont ces deux attributs qui seront alors envoyé en réponse à Environment. Notons que nous utilisons une structure de données CellsBag pour encapsuler les duos de position (ancienne / nouvelle). Cela permet, entre autres, de conserver un même protocole d'échange entre les entités mobiles et Environment. La réponse sera de performatif INFORM. **CyclicBehaviour**.

Agent Analyser

Un analyser sera créé pour chaque Monster créé. En effet, un Analyser ne se charge d'analyser qu'un seul Monster présent sur la plateforme. Chaque agent Analyser possède les attributs :

- value (qui l'identifie des autres et correspond au Monster qu'il analyse)

SubscribeToArtificialIntelligenceBehaviour Lors de sa création, l'agent Analyser envoie un message de performatif SUBSCRIBE à l'agent ArtificialIntelligence (AI). Cela permettra à AI d'accéder à ses informations en appliquant *getSender*() sur le message reçu. **OneShotBehaviour**.

WaitForRequestBehaviour Capte les messages envoyés par AI. Les messages doivent posséder le template adéquat (conversationID et perfomatif CFP). Lors de la réception, ce behaviour déclenche le sequential behaviour *SequentialLogicBehaviour.* **CyclicBehaviour.**

SequentialLogicBehaviour Déclenche de manière séquentielle les behaviours *RequestMonsterPositionBehaviour* et *WaitForMonsterPositionBehaviour*. **SequentialBehaviour.**

RequestMonsterPositionBehaviour Envoie un message (conversationID et performatif REQUEST) à l'agent Environment. **OneShotBehaviour**

WaitForMonsterPositionBehaviour Capte le message envoyé par Environment. Le message doit posséder le template adéquat (conversationID et performatif INFORM). Lors de sa réception, ce behaviour applique la fonction predictPositions() de l'Analyser. Cette méthode estime l'intégralité des positions que le Monster peut avoir sur les trois prochains coups et les retourne sous forme d'un tableau de cellules. C'est ce tableau de cellules avec les 'eventualMonsterPosition' qui sera envoyé en message (conversationID et performatif PROPOSE) à l'agent Environment. Lors de la réception du message, ce behaviour s'arrête. **Behaviour.**

Agent Artificial Intelligence (AI)

L'agent AI a pour objectif d'estimer la meilleure nouvelle position pour le Traveler. Il basera sa décision sur la position la plus éloignée (distance euclidienne) des 'predictedPosition' reçu par les Analyser. L'agent AI possède :

- analysersSubscriptionsList (la liste des AID des agents Analyser)
- predictedMonsterPositionsList (la liste des positions futures eventuelles des Monsters)
- readyForAnalyse (boolean qui permet de gérer le flux de messages)

MySequentialBehaviour Déclenche séquentiellement les behaviours *AcceptNewAnalyserSubscriptionBehaviour* puis *MyParallelLogicBehaviour.* **SequentialBehaviour.**

AcceptNewSubscriptionBehaviour Capte les messages envoyés par les Analyser. Les messages doivent posséder le template adéquat (conversationID et performatif SUBSCRIBE). L'AID du sender du message (Analyser) est alors ajouté à la liste analyserSubscriptions de l'agent AI. Lorsque la liste est pleine, ce behaviour s'arrête. **Behaviour.**

MyParallelLogicBehaviour Déclenche en parallèle les behaviours *GetRequestedFromTravelerBehaviour et GetProposalFromAnalyserBehaviour.* **ParallelBehaviour.**

GetRequestedFromTravelerBehaviour Capte les messages envoyés par Traveler. Les messages seront interceptés seulement si predictedMonsterPositionsList est vide. Les messages doivent posséder le template adéquat (conversationID et performatif REQUEST). Le message contient alors la position courante du Traveler. Ce behaviour lance alors un Call-For-Proposal (CFP) à tous les Analysers afin de récuperer les 'predictedMonsterPositions' de tous les Monsters de la plateforme. Il envoie donc un message (conversationID et performatif CFP) aux Analysers. Change ReadyForAnalyse à true. **CyclicBehaviour**.

GetProposalFromAnalyserBehaviour Capte les messages envoyés par les Analysers. Les messages seront interceptés seulement si readyForAnalyse est passé à true. Les messages doivent posséder le template adéquat (conversationID et performatif PROPOSE). Chaque message contient une liste des predictedMonsterPositions d'un Monster analysé. Ce behaviour stocke alors toutes les réponses dans predictedMonsterPositionsList. Une fois qu'il a reçu les n messages attendus. Ce behaviour passe readyForAnalyse à false et déclenche le behaviour *AnswerTravelerBehaviour*. **CyclicBehaviour**.

AnswerTravelerBehaviour Appelle la méthode mainLogic() de l'AI qui retourne la meilleur position (en terme de distance euclidienne avec les predictedMonsterPositions de la liste). Vide la predictedMonsterPositionsList. Envoie la position déterminée a Traveler. (conversationID et performatif INFORM). **OneShotBehaviour**.

Agent Traveler

L'agent Traveler est indépendant mais est considéré par Environment comme une entité mobile au même tître qu'un Monster. Il possède donc les attributs :

- value (qui l'identifie de manière unique sur la grille)
- position (qui correspond à la Cellule de la Grille sur laquelle il est après s'être déplacé)
- oldPositon (qui correspond à la Cellule sur laquelle il était avant de s'être déplacé)

RequestBestMoveBehaviour Envoie périodiquement une requête à agent AI. Message REQUEST avec conversationID. Le message contient la position de l'agent en JSON. **TickerBehaviour**

ForwardAIInfoToEnvironment Capte les messages envoyés par agent AI. Les messages doivent posséder le template adéquat (performatif INFORM et conversationID). Le message contient alors la meilleure position où aller, proposée par agent AI. A la reception du message, ce behaviour déclenche la méthode move() de l'agent Traveler. Cette dernière modifiera les attributs position de l'agent Traveler. Finalement, ce behaviour enverra à l'agent Environment ses positions afin que ce dernier effectue son déplacement dans la grille. De la même manière que agent Monster, agent Traveler enverra ses positions encapsulées dans un *cellsBag* afin de se conformer au protocole de l'agent Environment. **CyclicBehaviour**

Agent Environment

L'agent Environment définit l'environnement / la plateforme du jeu. Il est en relation avec les entités mobiles (Monsters et le Traveler), leur permettant ainsi de se déplacer dans l'environnement. Il possède donc les attributs :

- myGrid (qui est une grille de dimensions définies dans les constantes du logiciel)
- nshot (correspond au nombre d'itération du jeu : l'objectif du Traveler étant bien sûr de survivre pour un nshot maximal)

GetInformedFromEngineBehaviour Capte les messages envoyés par agent Engine. Le message doit posséder le template adéquat (performatif REQUEST). Le message reçu contient alors les informations (AID) d'un agent Monster. A la réception du message, ce behaviour commencera par afficher la grille en console puis il va donc envoyer un message au Monster, dont il possède maintenant les informations afin de lui demander de se déplacer (performatif REQUEST et conversationID). Remarquons que ce behaviour s'arretera (done = true) quand la grille.isOver sera true. Cet evenement correspondra à la fin du jeu (mort de l'agent Traveler). **Behaviour**.

GetInformedFromEntitiesBehaviour Capte les messages envoyés par toutes entités mobiles (agent Monster ou Traveler). Le message doit posséder le template adéquat (performatif REQUEST et conversationID correspondant à une discussion avec Monster OU Traveler). Le message reçu contient alors un cellsBag encapsulant l'ancienne et nouvelle position (cellule) de l'entité mobile. A sa réception, ce behaviour déclenche la méthode updateMyGrid() de l'agent Environment afin de mettre à jour la grille avec la nouvelle position de l'entité mobile. Remarquons que cette méthode testera la validité du déplacement. Si la nouvelle position correspond à une case infranchissable (mur), alors l'entité restera sur sa case (? la prevenir). Si la nouvelle position fait entrer en contact un Monster avec un Traveler, alors le Traveler meurt et la partie s'arrête. **Behaviour**

EndOfGameBehaviour Lorsque la partie s'arrête, (grille.isOver) ce behaviour notifiera d'un message (INFORM) l'agent Engine pour qu'il arrête son ticker. **OneShotBehaviour**

Agent Engine

L'agent Engine est le 'moteur' de la plateforme (indépendante du traveler). C'est lui qui enregistre les Monster au lancement du jeu et transmet périodiquement leurs informations (AID) à l'agent Environment afin qu'ils se deplacent de manière autonome. Lorsque la partie s'arrête (mort du Traveler), le 'moteur' de cet agent s'arrête. Il possède donc l'attribut suivants :

- mySubscriptions (liste de tous les AIDs des agents Monster de la plateforme)

MySequentialBehaviour Déclenche de manière séquentielle les behaviours *AcceptNewSubscriptionBehaviour, MyTickerBehaviour et WaitForEndOfGameBehaviour.*

AcceptNewSubscriptionBehaviour Capte les messages des agents Monsters. Les messages doivent posséder le template adéquat (performatif SUBSCRIBE). A leur réception, ce behaviour enregistre les informations du *sender* (AID d'un agent monster) dans sa liste mySubscription. Ce behaviour s'arretera lorsque la liste est pleine (taille fixée dans les constantes du logiciel). **Behaviour**.

MyTickerBehaviour A chaque *tick*, ce behaviour envoie autant de messsages qu'il y a d'AID stockés dans mySubscription, à l'agent Environment. A chaque *tick* donc, Engine transmet toutes les informations des Monsters à Environment. Chaque message contient l'AID et possèdent le performatif REQUEST. **TickerBehaviour.**

WaitForEndBehaviourOfGame Ce behaviour capte le message de fin de partie envoyé par Environment. Le message doit posséder le template adéquat (performatif INFORM). A sa réception, il détruit l'agent Engine (*myAgent.doDelete()*) pour arrêter le 'moteur'.

Maintenant que nos behaviours ont été formellement décris et détaillés, nous allons présenter, dans la seconde partie de ce rapport, la manière dont nous avons implémenté la communication entre nos agents. Et en particulier, quel protocole nous avons utilisé dans les interactions entre Traveler, AI et Analyser.

2. Phase de conception

Dans cette partie, nous nous concentrerons sur la partie la plus complexe de l'échange de messages entra agent que nous avons conçu. Il s'agit, dans notre cas, de l'implémentation d'un protocole CONTRACT-NET entre AI et Analyser.

Contexte

Rappelons avant tout que notre ligne rouge était de s'assurer que tous les agents liés au traveler (AI et Analyser) soit indépendant pour laisser à notre système la possibilité d'aisément les remplacer par une réelle intelligence humaine (un joueur réel).

Il était aussi important de considérer le caractère contractuel entre l'agent AI et les Analysers. En effet, l'agent AI effectue un appel général aux Analysers afin de récupérer des propositions de solutions (*predictedPosition*) dans notre cas. Ces différentes propositions sont ensuite analysées en interne par agent AI (en fonction de la position du Traveler connue) puis une décision est prise. Et ce de manière asynchrone et optimale.

Solution

Nous voyons bien ici que le protocole CONTRACT -NET propose une solution formelle intéressante à notre problématique. CONTRACT-NET est un protocole où un agent initiateur (agent AI ici) prend le rôle de manager qui désire qu'une tâche soit effectuée par un ou plusieurs agents (Analysers ici). La tâche demandée par agent AI aux Analysers sera de déterminer les positions futures des Monsters.

Remarque : Nous nous arrêterons à la partie 'proposition' de ce protocole. En effet CONTRACT-NET propose aussi une seconde partie 'de négociation' que nous ne prendrons pas en compte car inutile dans notre cas.

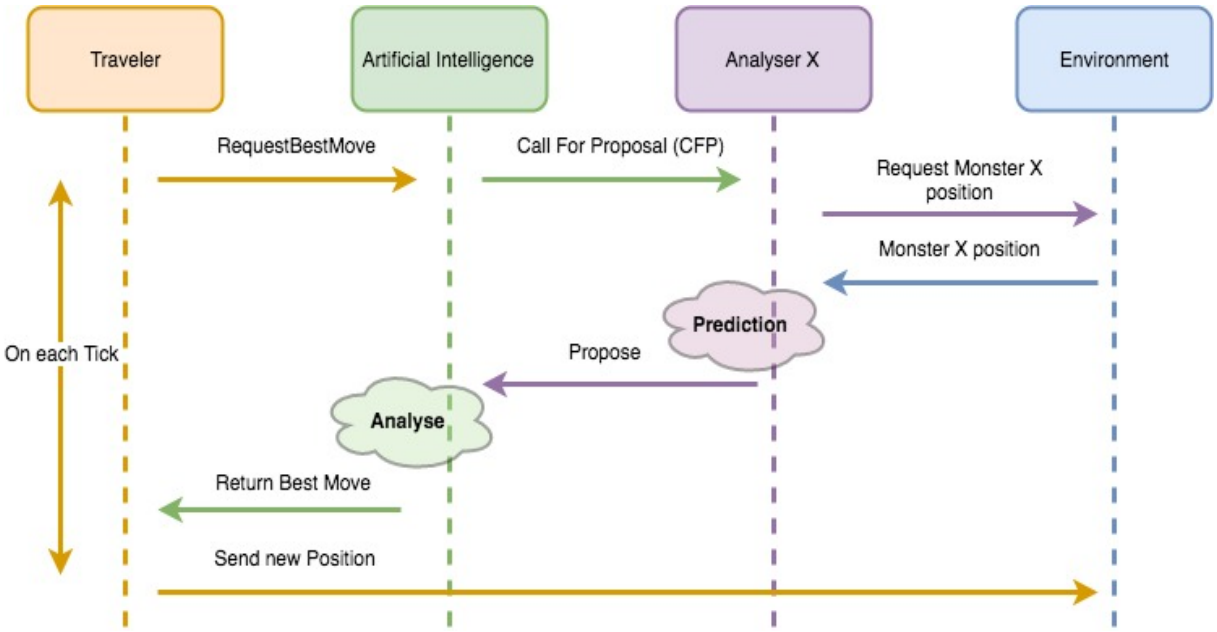


FIGURE 1. LOGIC AI & CONTRACT NET

Démonstration

Conclusion

Notre objectif était de concevoir et d'implémenter un système multi-agent concret. A travers notre plateforme de jeu, type Pacman, se mêle théorie des jeux, intelligence artificielle et programmation avec agents. Nous avons beaucoup apprécié réfléchir, concevoir et trouver des solutions pour pallier aux problématiques rencontrées dans le but de développer un jeu autonome, fonctionnel et amusant.

Ce projet aura été l'occasion de mettre en pratique une grande partie de la théorie des système multi-agents que nous avons vu en cours ce semestre. Nous avons cherché à conserver des bonnes pratiques d'architecture (indépendance et rôle des agents bien définis) mais aussi de programmation. Nous avons donc appliqué nos connaissances en Système Multi Agents, JADE, architecture logiciel, programmation asynchrone et langage JAVA à travers ce projet.