Problème de dinner des philosophers avec le système multi-agents

Cong-Minh Dinh

***Abstract*— L'objectif principal de cet article est de mettre en place un système capable d'étudier le comportement des agents pour résoudre le problème du Dîner de philosophie.**

I. INTRODUCTION

A. Problème de dinner des philosophers

Le problème Dining Philosophers est une simple abstraction des problèmes de synchronisation permettant d'allouer plusieurs ressources réutilisables. Elle est posé et résolu par E. Dijkstra en 1965.

La description classique du problème suppose qu'il existe cinq philosophes qui travaillent et ont faim de temps en temps. Il y a cinq assiettes et un seul fourchette entre assiettes voisines (Fig 1).

Un philosophe a besoin de deux fourchettes à manger. Si la fourchette est prise, elle ne sera pas libérée avant qu'un philosophe ait fini de manger. Chaque philosophe fait un cycle : prenant, mangant, déposant, et pensant.

L'un des états possibles (les pires) d'un tel système est atteint lorsque cinq philosophes ont faim en même temps et que chacun prend un fourchette sur la table. Enfin personne ne peut manger.

De nombreuses solutions sont proposées pour résoudre ce problème. Le défi consiste à proposer une solution optimale pour que les philosophes ne restent pas dans l'impasse. La solution doit être équitable et efficace pour que les philosophes n'aient pas à attendre plus longtemps pour manger. Ainsi, notre objectif est de proposer une solution qui maximise le parallélisme pour que tous les philosophes soient nourris et satisfaits.

Dans ce rapport nous allons développer un algorithme pour résoudre le problème par le système multi-agent. Cette méthode pourrait surmonter toutes les autres solutions (ex sémaphore, etc.) et fournira une solution optimale du problème.

Fig. 1. Le problème de philosophies

B. Système Multi-agent

Un système multi-agents (SMA) est un système informatisé composé de plusieurs agents intelligents en interaction. Les systèmes multi-agents peuvent résoudre des problèmes difficiles ou impossibles à résoudre pour un agent individuel.

Le but des systèmes multi-agents est de comprendre comment des processus indépendants peuvent être coordonnés.

Un agent peut être décrit comme autonome car il a la capacité de s'adapter lorsque son environnement change. Un système multi-agents est constitué que plusieurs agents existent en même temps, partagent des ressources communes et communiquent entre eux (Fig 2). La question clé dans les systèmes multi-agents est de formaliser la coordination entre les agents.

Dans ce rapport, on construit un système multi-agents pour résoudre le problème du philosophe. On commence par la modélisation des activités de philosophes, puis en l'implémentant sur Java et en évaluant les résultats obtenus.

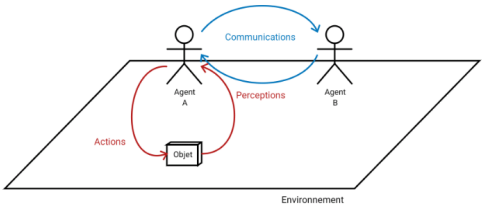

Fig. 2. Schéma théorique d'un système multi-agents

II. Modélisation de processus et Conditions

A. Modélisation

Le modèle de système est montré sur la figure ci-dessous.

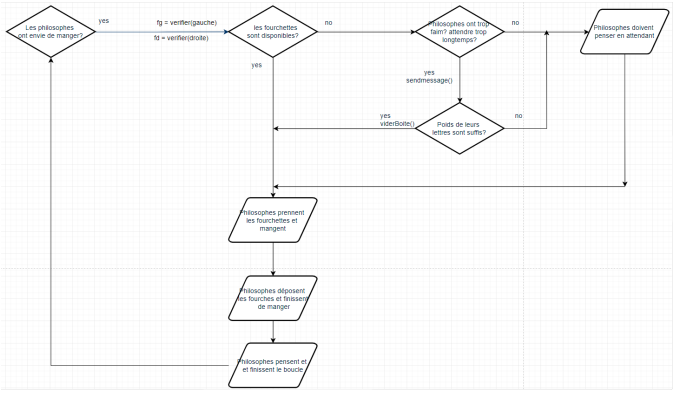

Fig. 3. Schéma de notre programme

B. Condition

Pour que les agents fonctionnent correctement, nous supposons une symétrie parfaite et des conditions égales décrites ci-dessous :

— tout philosophe affamé mange tôt ou tard.
— tous les philosophes suivront la même procédure.
— un philosophe échangera des messages entre eux.
— les voisins ne mangent pas en même temps.

III. Implémentation

On a créé deux packages sur Java, l'un appelé Philosophe et l'autre nommé Platforme. Le Philosophe est un composé de l'Agent, comme la Table étant un composé de l'Environment. Dans la plateforme, l'Agent travaille sur l'Environnement et l'Environnement lui répond également. Ces deux classes attaquent donc simultanément (figure 4).

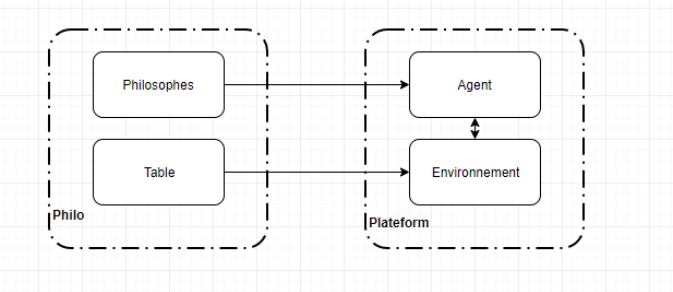

Fig. 4. Les packages utilisées dans notre programme

Ainsi, on défini sur Java *Table extend Environment*, *Philo extends Agent* et *Agent extends Thread*.

A. Agent

1) Philosophes: Le philosophe est un élément actif du système.

Logique théorique

Le philosophe vit selon son temps interne, pense et se sent parfois affamé.

Lorsque le philosophe aura faim, il vérifiera le statut de ses fourches. Si des fourches à droite et à gauche sont disponibles, l'autorisation sera donnée d'acquérir les ressources (ie des fourches). Si le philosophe est incapable de manger, il est affecté à l'état d'attente. En ce moment, il pense.

Un philosophe n'est autorisé à manger lorsque ses philosophes de gauche et de droite ne sont pas en état de manger. Si cette condition est vraie, le philosophe est informé et un signal lui est envoyé pour qu'il commence à prendre des fourches et manger.

Quand un philosophe cesse de manger, son état change en pensant. En ce temps, il est informé que ses voisins ont une chance de manger. Cela changera le statut du philosophe de manger à penser, libèra les états des fourches et vérifiera le nouveau philosophe qui a faim.

Implémentation

Afin de coder sur Java, on utilise un $compteurFaim$ et un $compteurPenser$. Chaque fois que le philosophe travaillera, son niveau de faim augmentera et son niveau de pensé diminuera. La même logique s'applique au philosophe quand il attend ou mange.

On aura également besoin d'une variable $fourchettes\_acquises$ comme un signal lorsque les deux fourches sont déjà tenus ou toujours libres.

Si les fourchettes ne sont pas disponibles, le philosophe devait réfléchir en même temps et un $Thread.sleep()$ est donc utilisé pour suspendre l'exécution en cours.

Un compteur de temps $total\_attente$ défini par $System.currentTimeMillis()$ est également configuré pour contrôler le temps de chaque philosophe et évaluer le résultat avec précision.

Dans notre program, un philosophe est défini à partir d'un numéro id, de son niveau de faim, de son niveau de réflexion

et de la liste des fourchettes disponibles. Un cycle ci-dessous est toujours garanti (Fig 3) :

— La première consiste à confirmer si les philosophes ont faim ou non. Si les fourchettes ne sont pas disponibles, les philosophes doivent attendre un moment. Pendant ce temps, ils devront réfléchir.
— Ensuite, ils prendront les collations et commenceront à manger.
— Après mangé, ils n'ont plus de faim et ont envie de penser.
— En fin, ils pensent et finissent le cycle.

Tout ce cyle est implémenté dans $boucle\_procedurale()$. Sachant que le boucle répète et ne s'arrête jamais, dans le program principale *main* on a également implémenté un $Thread.sleep()$ afin d'arrêter nos agents après un temps spécifié.

2) Table: Cette partie n'est pas faite en tant qu'agent, mais il s'agit de contrôler l'utilisation des ressources et attribuer les états de fourchettes aux philosophes. En cela, on crée d'abord une liste booléenne représentant les états de fourchette. Des méthodes telles que *prendre*, *poser* et *vérifier* sont également appliquées dans cette section pour définir et modifier l'état disponible de fourchettes.

— *prendre* : Le philo prend les fourches à sa gauche et à sa droite. Ces fourches disponibles deviennent indisponibles.
— *poser* : Le philo déposent les fourches à sa gauche et à sa droite. Ces fourches indisponibles deviennent disponibles.
— *vérifier* : Cela nous permet de savoir si une fourche est disponible ou non.

B. Message

En termes réels, on veut créer également une communication entre les agents. Un bloc de message serait mis en place. Il esst défini comme une classe capable d'envoyer et de recevoir des messages.

Imaginons le cas où les philosophes ont faim ou s'ils attendent depuis longtemps. Ainsi, ils peuvent envoyer un message à la boîte de lettres et demander à être la prochaine personne à être mangée.

Pour ce faire, le système de message est défini comme un processus entre les étapes (Fig 5) :

1) Création (entrée), destruction (sortie)
2) Exécution, Envoi et réception (consultation) de messages

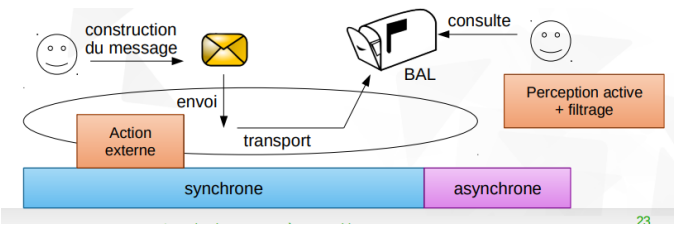

Fig. 5. Envoi de message synchrone

En raison de contraintes de temps et de conditions plus simples, supposons l'apparition d'un responsable qui contrôle le nombre de lettres envoyées et répond au philosophe si sa demande est favorable (Fig 6).

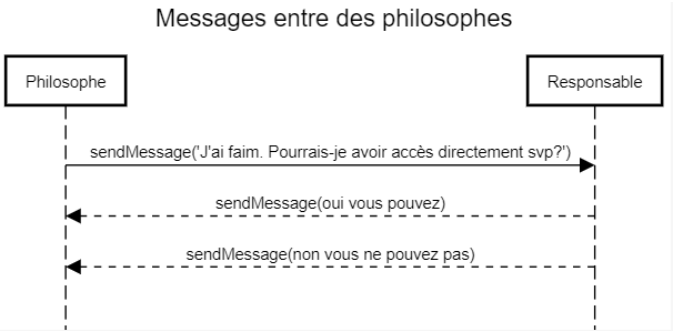

Fig. 6. Diagramme d'échange de messages

Au début, on a pensé de créer une boîte de lettres commune à tous les philosophes. Cependant les lettres de philosophes seraient mélangées entre eux et il serait difficile de distinguer les messages de chaque philosophe.

On souhaite un système égal et unifié pour tous les philosophes, basé sur le nombre et les permatifs des lettres qu'ils ont envoyées. On a donc créé une boîte aux lettres dans laquelle son numéro de compartiment était égal au nombre de philosophes. Lorsqu'un philosophe a faim, il envoie une lettre à son propre compartiment (méthode $sendMessages$). Cette lettre est composée d'un permatif et d'un contenu. Mais seulement le permatif qui nous intéresse. Le responsable compte les permatives de toutes les lettres dans le compartiment du philosophe et décide s'il peut lui donner un accès direct aux repas (méthode $getBoite$). Après avoir obtenu la priorité, toutes ses lettres seront annulées et sa nouvelle demande sera recalculée dès le début (méthode $viderBoite$).

C. Diagramme de l'UML

Pour visualiser la conception d'un système, on a créé un diagramme du langage UML.

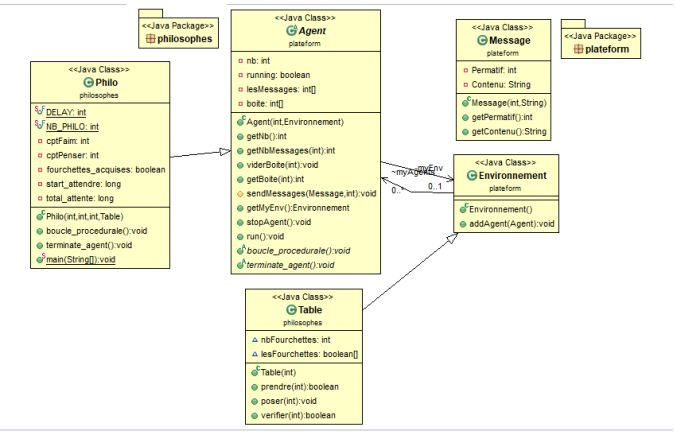

Fig. 7. Diagramme de l'UML pour notre système

On constate que les relations entre les classes de notre système sont bien respectées par la règle annoncée à la Fig 4.

IV. Résultat obtenu

```
// Result, n = 5
L'agent Thread-3 a attendu en tout 8
Thread-0 a envoy une lettre
Thread-2 doivent penser et encore attendre
Thread-4 a attendu pendant 3
L'agent Thread-2 a attendu en tout 6
Thread-0 doivent penser et encore attendre
L'agent Thread-0 a attendu en tout 6
L'agent Thread-1 a attendu en tout 7
L'agent Thread-4 a attendu en tout 6
Le repas est fini en 12.0
fourchette 0 = false
fourchette 1 = false
fourchette 2 = false
fourchette 3 = true
fourchette 4 = false
```

Premièrement, nous voyons que tous les cas sont exécutés et non arrêtés à tout moment. Tous les agens ont été pu exécuter un cycle : avoir faim, envoyer une messager, recevoir une réponse, attendre, poser les fourchettes, manger, déposer les fourchettes et penser.

Lorsqu'on augmente le nombre de philosophes, le temps du repas est également beaucoup plus long et varie beaucoup. Par exemple, si le nombre de philosophe est égal à 5, l'heure du repas est toujours comprise entre *11-13 (ms)*. Mais si le nombre de philosophe est à 100 par exemple, ce temps serait de l'ordre de *30-100 (ms)* variant d'autant plus.

V. Conclusion

Bien qu'il existe encore des théories et des solutions pouvant améliorer les résultats et que nous n'avons pas encore appliquées ni analysées dans un temps limité, nous avons terminé le projet et compris comment utiliser un système multi-agents.

Il était également difficile car on n'a pas de compétences expertes en Java. Par exemple, on a passé plusieurs heures à cause d'un problème bloqué lors de l'utilisation d'une *liste de liste* pour stocker un message dans chaque comportement. Pour la contrainte de temps, on a décidé d'utiliser une array *int* pour compter les permatives de toutes les lettres du philosophe. Une annexe de l'implémentation de la *liste de liste de messages* est également conservée en annexe de notre fichier Agent.

En résumé, ce petit projet nous aide à comprendre l'implémentation d'un système multi-agents et à améliorer nos compétences en Java et en programmation. L'algorithme SMA fourni ci-dessus constituerait une solution optimisée au problème du philosophe, en évitant le problème de blocage, de famine et de minimiser le temps au blocage des ressources.

References

[1] Cours de Système Multi Agent (CentraleSupélec)
[2] Wikipedia