Système multi-agents

TP SMA

Année 2022 – 2023

Réalisé par :

FERCHIOU Iskander
KANAAN Nour Eddine
SENER Emre

Responsable de l'UE :

M. AKNINE Samir

POLYTECH Lyon - UNIVERSITE CLAUDE BERNARD LYON 1
Domaine Scientifique de La Doua – 15, Boulevard Latarjet
69622 VILLEURBANNE CEDEX
Tél. (33) 04.72.43.12.24 - Fax. (33) 04.72.43.12.25
http://polytech.univ-lyon1.fr/

Sommaire

I. Introduction

Dans le cadre de notre module d'ouverture à la recherche, nous avons réalisé la conception et le développement d'un système de négociation automatique multi-agents. L'objectif était de mettre en place un cadre permettant de faire échanger des offres entre des agents fournisseurs et acheteurs afin de simuler des ventes de ticket de déplacement.

Chaque agent possède des contraintes et des préférences qui doivent être prises en compte lors d'une négociation. Par exemple, le budget maximum de l'acheteur est un élément majeur qui permet d'influencer le prix d'une offre. De même, les fournisseurs ne doivent pas vendre un ticket en dessous de son prix minimum, ce qui laisse une certaine marge de négociation avant de céder le produit.

Que ce soit pour un fournisseur ou un acheteur, la maximisation du gain était également un enjeu majeur du TP. En ce sens, nous nous sommes attardés sur l'impact de différentes stratégies de négociation.

Lors de ce compte-rendu, nous reviendrons à plusieurs reprises sur le programme réalisé à travers diverses explications et illustrations (diagramme de classe, diagramme de séquence, graphiques, ...). Au fur et à mesure, nous justifierons les choix de conception et la logique derrière l'intégration de concepts importants comme le multithreading.

Notre code est accessible via le lien suivant : https://github.com/IskanderFerchiou/TP_SMA

II. Diagramme de classes

Pendant ces derniers mois, nous avons régulièrement généré des diagrammes de classes à l'aide de notre IDE (IntelliJ IDEA) pour vérifier la cohérence des liens entre nos classes et éviter des problèmes de conception comme des dépendances circulaires. A l'issue du TP, nous avons fini par coder un total de 14 classes visibles sur le schéma suivant :

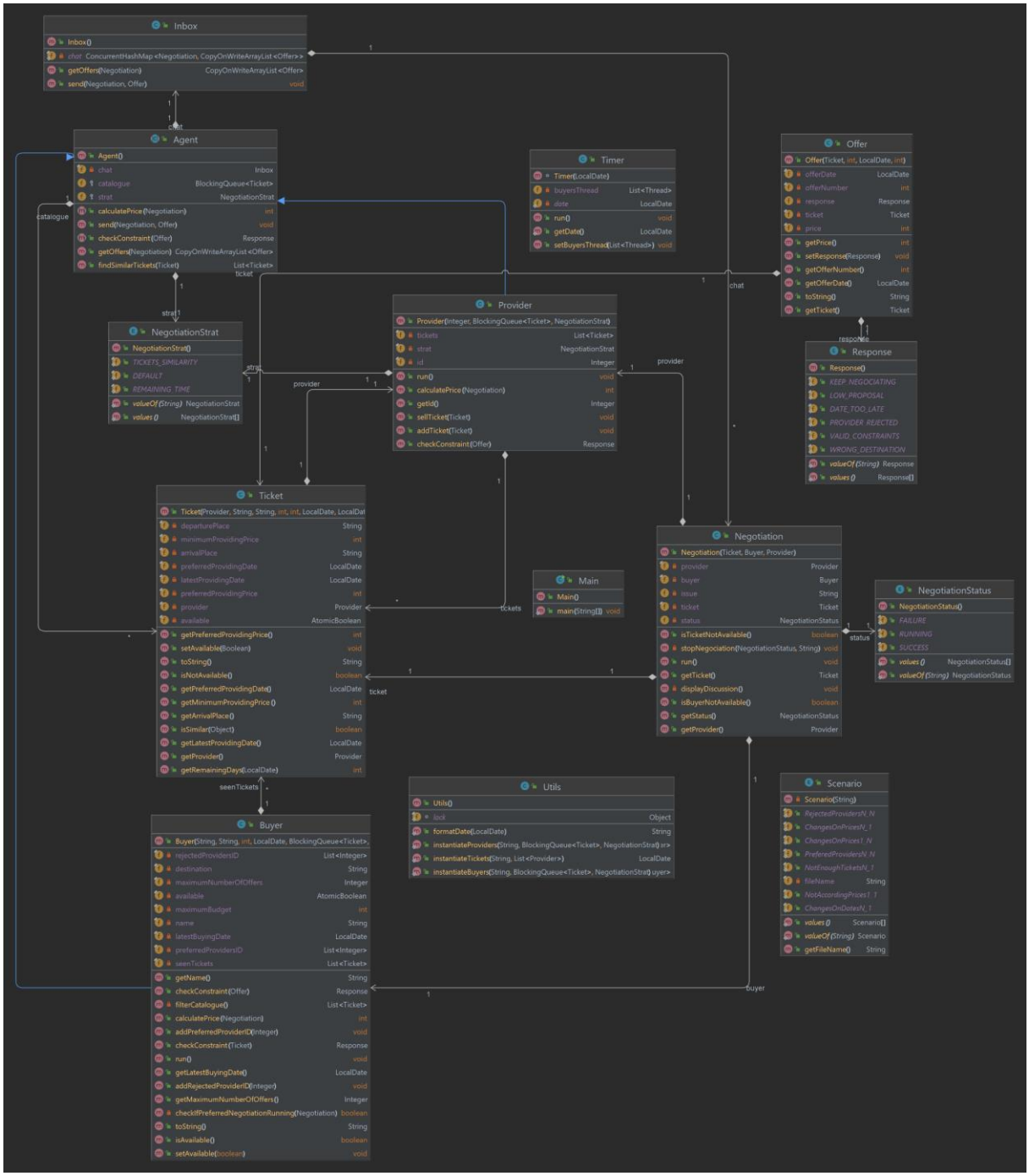

Figure 1 - Diagramme de classes

➢ **Agent** est une classe abstraite dont **Provider** et **Buyer** héritent. Son rôle principal est de factoriser le code en commun entre un acheteur et un fournisseur.

- **Ticket** représente l'objet principal d'une négociation. Cette classe répertorie toutes les informations pertinentes aux yeux de l'acheteur comme l'identité du fournisseur, le lieu de départ, etc. Elle répertorie également les contraintes et les préférences d'un vendeur pour un ticket (prix de vente minimum, date limite de vente, …).

- **Inbox** est dédiée à l'enregistrement des échanges entre un acheteur et un fournisseur dans un dictionnaire dont la clé est une négociation (espace de discussion) et la valeur est une liste d'offres (messages envoyés).

- **Offer** symbolise la proposition (message) d'un acheteur ou d'un fournisseur pendant une négociation. Pour chaque offre, on spécifie plusieurs attributs comme la date, le prix, le ticket ou encore la réponse de l'interlocuteur.

- **Negotiation** est une discussion (thread) initiée par un **Buyer** avec un **Provider** à propos d'un **Ticket**. Plus précisément, il s'agit d'une succession de **Offer** envoyées entre chaque agent avec une étude du prix, des contraintes et des disponibilités tout au long de l'échange. Cette classe est notamment importante dans le cadre de la communication N-N car il est possible de générer N*N threads de négociation avec un suivi régulier des variables partagées (exemple : disponibilité du ticket).

- **Timer** permet de gérer le temps, et plus précisément, la date des négociations en incrémentant cette donnée d'une journée toutes les 200 millisecondes. Cette classe est un thread dont la durée de vie est liée à celle des acheteurs afin d'éviter que le programme continue son exécution indéfiniment.

- **Utils** contient les méthodes permettant d'initialiser des acheteurs, fournisseurs et tickets à partir de fichiers CSV ainsi qu'un objet partagé (verrou) entre des threads dans une optique de synchronisation.

- **Response** est une énumération qui renferme les différentes réponses possibles à une offre d'un agent après une étude des contraintes (budget, date, blacklist, etc.).

- **Scénario** est également une énumération qui associe chacun des noms des fichiers CSV de notre jeu de données à un scénario de test. Chaque possibilité permet de tester un aspect du code (concurrence N-N, dépassement de la date limite, budget insuffisant, …).

- **NegotiationStrat** et **NegotiationStatus** sont des énumérations qui contiennent, respectivement, les différentes stratégies de négociation et les états possibles d'une négociation.

- **Main** permet de définir un scénario de test et une stratégie de négociation avant d'instancier les différents acteurs, démarrer les threads et exécuter notre application.

III. Multithreading et structures partagées

Durant l'élaboration de ce TP, nous nous sommes appuyés sur le concept de multithreading afin d'exécuter plusieurs threads simultanément et de faire communiquer N acheteurs avec N fournisseurs. En ce sens, nous avons défini une classe abstraite Agent qui implémente l'interface Runnable au lieu d'hériter de la classe Thread afin de faciliter le partage des objets entre les threads et optimiser l'utilisation de la mémoire. Un acheteur et un fournisseur héritent tous les deux de la classe Agent, ce qui permet d'accéder à des structures partagées comme le catalogue ou le chat par exemple, mais aussi de surcharger des méthodes communes telles que calculatePrice, findSimilarTickets ou encore checkConstraint.

Au fur et à mesure, notre groupe s'est rendu compte que les threads sont susceptibles de modifier des variables en même temps. Mais encore, la valeur de ces variables doit constamment être la même entre tous les threads afin de prendre la bonne décision pendant une négociation. Par exemple, il n'est pas possible de vendre un ticket dont la présence au sein du catalogue n'est plus d'actualité. Pour répondre à cette problématique, nous avons fait appel à plusieurs structures « thread-safe » c'est-à-dire capables de fonctionner correctement lorsqu'elles sont lues ou modifiées simultanément par plusieurs threads.

- **BlockingQueue** représente le type de notre catalogue. Par définition, il s'agit d'une file d'attente synchronisée qui gère les opérations de lecture et d'écriture de manière simultanée. Nous avons découvert cette structure en étudiant la programmation concurrente et plus précisément, le problème des producteurs et des consommateurs. Cette file d'attente assure que tous les agents ont accès aux mêmes instances de tickets au même moment.

- **AtomicBoolean** définit le type des variables associées à la disponibilité d'un ticket et d'un acheteur. Étant donné que l'utilisation d'un booléen classique n'est pas « thread-safe », nous avons trouvé une autre alternative qui provient directement du package « Concurrent » de Java. Contrairement à un booléen volatile, un AtomicBoolean est modifiable par plusieurs threads et ne nécessite aucune synchronisation avant d'effectuer une opération sur une variable.

- **LocalDate** représente le type de la date du jour (et de toutes les dates présentes dans le code pour une question d'homogénéité). Contrairement à la classe Date, LocalDate est une classe immuable et donc « thread-safe » car toutes les méthodes retournent une copie au lieu de modifier l'état de l'objet originel. En passant à la date suivante, l'utilisation de ce type garantit que tous les agents observent la même date à l'instant t.

- **ConcurrentHashMap** définit le type de notre espace de discussion (chat). Si un thread itère sur une HashMap classique et qu'un autre met à jour cette structure de données au même moment, il est possible de voir apparaître une erreur de type « ConcurrentModificationException ». Dès lors, il est préférable de s'orienter vers une

structure telle que ConcurrentHashMap qui se trouve être adapté aux situations de compétition (race condition). Notre groupe a fixé la négociation en clé et une liste d'offres envoyées par un agent en valeur. De cette façon, il est possible d'accéder à l'historique de discussion entre un acheteur et un fournisseur pendant une négociation.

- **CopyOnWriteArrayList** représente le type de la liste des offres envoyées par un agent. Cette structure de données est utilisée au lieu d'une ArrayList pour les mêmes raisons évoquées précédemment au niveau de la classe ConcurrentHashMap. Chaque fois qu'un élément est modifiée, une nouvelle ArrayList est renvoyée. En termes de performance, il est toujours préférable de faire appel à des ArrayList. Cependant, ces tableaux dynamiques ne sont pas synchronisés et ne sont pas adaptés à un contexte multithreading, ce qui justifie l'emploi d'une CopyOnWriteArrayList.

Pendant la phase de développement, nous avons également eu l'occasion d'expérimenter d'autres structures comme CountDownLatch et Phaser qui permettent de bloquer des threads afin de les démarrer en même temps par la suite. Cependant, nous avons fini par mettre en place un système à base de wait() du côté des acheteurs et notifyAll() du côté du timer en s'appuyant sur un objet partagé (verrou) entre les threads. Le principe est simple : bloquer les acheteurs à la fin de chaque journée de négociation et les débloquer toutes les 200 millisecondes afin de laisser le temps aux négociations d'avancer en même temps. Cette logique permet de contourner l'utilisation des structures évoquées au début de ce paragraphe tout en apportant une synchronisation basée sur la date du jour.

IV. Protocole de négociation

Lors de ce TP, nous nous sommes inspirés de la société actuelle et plus précisément, du comportement des acheteurs sur des marchés en ligne communautaire comme Vinted ou Leboncoin. Pour mieux comprendre le fonctionnement de notre code, nous avons réalisé **un diagramme de séquence** qui met en lumière les différents éléments intervenant lors d'une négociation entre un fournisseur et un acheteur. En raison de sa taille imposante, ce dernier est disponible au sein du répertoire « UML ».

Au lancement du programme, on instancie les acheteurs, les fournisseurs et les tickets à partir des fichiers CSV présents dans le répertoire « datasets ». Préalablement, on veillera à définir un scénario de test ainsi qu'une stratégie de négociation en fonction des observations souhaitées. De plus, un catalogue et un timer sont également utilisés pour gérer, respectivement, la mise en vente des tickets et l'incrémentation de la date. Bien entendu, les threads associés aux acheteurs et fournisseurs sont démarrés afin de mettre en route le code principal de chacun des acteurs.

A l'instar de la vie réelle, le fournisseur commence par ajouter ses tickets dans le catalogue afin que les potentiels acheteurs puissent y avoir accès. Parallèlement, les acheteurs vont suivre un procédé spécifique en boucle. Dans l'ordre, ils vérifient la validité de la date limite

d'achat par rapport à la date actuelle, consultent et filtrent le catalogue selon leurs contraintes et leurs préférences, font une offre au vendeur en démarrant un thread symbolisant la négociation, puis effectuent un suivi de chaque négociation. Si la présence d'un fournisseur préféré est avérée, l'acheteur négocie uniquement avec lui au départ. En cas d'échec, il initiera de nouvelles négociations avec d'autres fournisseurs. Par rapport aux fournisseurs "blacklistés", aucune négociation n'est entamée avec eux.

Au sein du thread négociation, l'acheteur et le vendeur s'échangent (à tour de rôle) des offres tout en prenant en compte la disponibilité de l'acheteur et du ticket sur le marché. En effet, plusieurs négociations peuvent avoir lieu en même temps avec différents fournisseurs, acheteurs et tickets. En ce sens, il est nécessaire de mettre fin à certaines négociations lorsque le ticket a été vendu/acheté ailleurs par exemple.

Durant toute la durée d'une négociation, chaque acteur vérifie si les contraintes (date limite, budget, prix de vente minimum, etc.) liées à une offre sont bien respectées avant de répondre à son interlocuteur ou de procéder à la vente du ticket. Par ailleurs, nous avons limité le nombre d'offres possibles par un acheteur et nous mettons fin à une négociation lorsque ce seuil a été atteint. Selon la stratégie de négociation définie initialement, le montant des offres est susceptible d'être différent lors du calcul des prix par chaque agent.

Dès qu'une offre a été envoyée par chaque acteur, on bloque la négociation et le timer débloque la situation en passant au jour suivant. Le changement de date s'effectue toutes les 200 millisecondes pour laisser le temps aux négociations d'avancer en même temps.

Pour finir, on récupère l'historique des échanges entre l'acheteur et le vendeur à partir de la boîte de réception de chacun et on affiche les offres proposées tout au long de la négociation pour mieux comprendre la tournure des évènements.

Remarque : les instances des classes Ticket et Offre n'ont pas été représentées sur le diagramme afin de faciliter la compréhension de celui-ci.

V. Jeu de données

a. Structure des datasets

Afin de tester la robustesse de notre code ainsi que la qualité des stratégies de négociations, nous avons établi plusieurs jeux de données (dans des fichiers CSV) utilisés dans différents scénarios de test. En ce sens, notre équipe a mis en place une convention de nommage pour chaque fichier :

- Buyers - X.csv contient tous les acheteurs à la recherche d'un billet. Ces fichiers contiennent notamment les contraintes et les préférences d'un acheteur qui varient selon le nom du scénario. Un nom a été attribué à chaque client pour mieux les distinguer lors des négociations.

- ➢ Providers - X.csv renferme tous les vendeurs ayant des tickets en vente. Sachant que les contraintes et les préférences de ce type d'agent dépendent du ticket, nous avons simplement renseigné les identifiants des fournisseurs dans le fichier. De cette manière, il est possible d'attribuer des tickets à un vendeur et de placer son ID dans la liste des fournisseurs préférés ou détestés par un acheteur.

- ➢ Tickets - X.csv englobe tous les tickets présents dans le catalogue. En plus des informations de base comme le lieu et la date de départ, on retrouve également les contraintes et préférences liées au ticket. Comme évoqué précédemment, les données telles que le montant de départ, le prix minimum et la date d'échéance varient en fonction du ticket et non du vendeur.

Pour choisir un scénario, il existe une variable *scenarioName* dans le code de la classe *Main* qui prend en valeur une des options présentes dans l'énumération *Scenario*. Par exemple, l'affectation du scénario *NotEnoughTickets_N-1* dans *scenarioName* conduit au chargement des fichiers suivant :

- Buyers - NotEnoughTickets_N-1.csv
- Providers - NotEnoughTickets_N-1.csv
- Tickets - NotEnoughTickets_N-1.csv.

b. Scénarios de test

A présent, nous allons expliquer en détail le contenu des jeux de données associés à un scénario en mettant en lumière l'objectif de chaque test. En ce qui concerne l'interprétation des résultats, notre groupe a réalisé une démonstration (accessible via le lien suivant : https://www.youtube.com/watch?v=fB4MjS0qHX8) qui s'attarde sur le sujet.

- ➢ Scénario 1 (*NotEnoughTickets_N-1*) s'appuie sur 4 acheteurs, dont 2 voulant aller à Tokyo et 2 souhaitant voyager à Paris. En revanche, il n'y a qu'un vendeur disposant d'un seul ticket pour chacune de ces destinations. Au niveau des prix, le budget maximum de chaque acheteur permet d'acheter un ticket disponible dans le catalogue. Les clients n'ont aucune restriction ou préférence vis-à-vis des vendeurs. Ici, le but est de tester le comportement des acheteurs en situation de compétition lorsqu'il n'y a pas assez de tickets pour tout le monde.

- ➢ Scénario 2 (*ChangesOnPrices_1-N*) repose sur un acheteur et 4 vendeurs qui disposent chacun d'un ticket. Ces billets respectent les contraintes de l'acheteur par rapport à la destination et la date. Les différences entre les tickets se font uniquement au niveau du prix. Encore une fois, l'acheteur n'a aucune restriction ou préférence en ce qui concerne les vendeurs. L'idée est de tester la concurrence entre les vendeurs et donc de voir le prix auquel l'acheteur va acheter son ticket.

- Scénario 2 bis (*ChangesOnPrices_N-1*) est l'inverse du scénario décrit précédemment. Cette fois-ci, 3 acheteurs sont à la recherche d'un ticket strictement identique au niveau des dates mais aussi de la destination. Seul le budget maximum de chaque acheteur varie. Un seul ticket correspondant aux critères des clients est mis en vente par un seul vendeur. Aucune restriction ou préférence des acheteurs par rapport aux vendeurs. L'objectif est de tester la concurrence entre les acheteurs pour un même ticket et donc de voir le prix auquel le fournisseur va vendre son produit.

- Scénario 3 (*PreferedProviders_N-N*) fait intervenir 2 acheteurs et 2 vendeurs qui proposent un ticket chacun. Ces tickets respectent les contraintes des acheteurs que ce soit en termes de destination, de date ou de prix. Contrairement aux précédents scénarios, les acheteurs préfèrent tous les deux le fournisseur dont l'ID est 0. De ce fait, le but est de tester l'importance accordée par les acheteurs à leurs préférences. Plus précisément, on s'attend à ce que les deux clients démarrent une seule négociation avec le vendeur ayant l'identifiant 0 et que le « perdant » consulte l'autre fournisseur après-coup.

- Scénario 3 bis (*RejectedProviders_N-N*) est l'inverse du scénario décrit précédemment en matière de préférences : la liste des fournisseurs préférés devient la liste des fournisseurs détestés. On garde les mêmes contraintes et agents mais on ajoute un nouveau fournisseur avec deux tickets susceptibles d'attirer les clients. Ici, les acheteurs ont une mauvaise relation avec un vendeur différent donc ils négocieront uniquement avec ceux qui ne sont pas « blacklistés ». Ce scénario permet de vérifier qu'un client ne rentre jamais en négociation avec un fournisseur mal aimé.

- Scénario 4 (*ChangesOnDates_N-1*) s'appuie sur 4 acheteurs et 1 vendeur qui dispose de plusieurs tickets avec des dates limite de vente différentes pour une unique destination. En ce qui concerne les acheteurs, ils recherchent la même destination et possèdent le même budget maximum : seule la date limite d'achat varie. De plus, les clients n'ont aucune restriction ou préférence vis-à-vis des vendeurs. Ainsi, le but est de tester l'effet des dates limite d'achat des clients et des dates limite de vente des tickets sur les négociations (des interruptions sont attendues).

- Scénario 5 (*NotAccordingPrices_1-1*) repose sur un seul acheteur et un seul fournisseur ayant un ticket. Aucune restriction ou préférence des acheteurs par rapport aux vendeurs. Ici, le billet concorde au niveau de la destination et de la date de départ recherchée. Néanmoins, le budget maximum de l'acheteur est inférieur au prix minimum du ticket. L'idée est donc de vérifier qu'une vente n'a pas lieu et que les deux agents négocient bien jusqu'au nombre maximum d'offres fixé préalablement sans jamais dépasser le budget maximum et le prix de vente minimum.

VI. Calcul du prix et stratégies de négociations

Une fois le système de négociation fonctionnel, nous avons mis en place 3 stratégies différentes susceptibles d'influencer le calcul du prix avant l'envoi d'une offre à un agent. En effet, la prise en compte de la stratégie de négociation intervient dans la méthode calculatePrice() présente chez les acheteurs et les fournisseurs. Comme son nom l'indique, cette fonction envoie le prix de la nouvelle offre envoyée par un agent pendant une négociation. Elle prend en paramètre une négociation afin de retrouver le ticket (sujet de la négociation) et de récupérer les offres envoyées préalablement.

a. Coefficient de négociation

Avant d'aborder les stratégies, il est important d'évoquer le coefficient de négociation pour mieux saisir le calcul des prix. Au sein de la fonction calculatePrice(), nous utilisons une variable « coefNegotiation » qui représente la variation entre chaque offre. Ce taux s'obtient en effectuant la différence entre la première et la dernière offre d'un acheteur divisé par le maximum d'offres possibles (fixé préalablement) moins un. L'utilité principale de cette donnée est d'augmenter ou de diminuer le prix d'une offre progressivement en proposant le budget maximum de l'acheteur ou le prix minimum du ticket à la fin. Cependant, la formule varie légèrement selon le type de l'agent :

➢ Fournisseur : $\frac{prixTicket - minPrixTicket}{prixTicket * (maxOffres - 1)}$

Ici, nous avons fait en sorte que le coefficient de négociation se base sur le prix initial et le prix minimum du ticket. Pour mieux comprendre, admettons qu'un acheteur a le droit de faire 6 offres maximum pour un ticket coûtant 800 € au départ avec un prix minimum de 600 €. Dans ce cas, coefNegotiation est égale à 25 % divisé par 5, ce qui donne 5 %. Ainsi, le fournisseur réduira le prix de ses offres de 5 % à chaque fois au cours de la négociation.

➢ Acheteur : $\frac{m - premiereOffre}{(maxOffres - 2) * m)}$ où $m = min(prixTicket, budgetMax)$

Malgré des similitudes entre les deux formules, un peu de complexité s'ajoute du côté de l'acheteur. Tout d'abord, notre groupe a décidé, arbitrairement, que ce type d'agent propose 80% du prix initial du ticket (ou le budget maximum en cas de dépassement) en guise de première offre au fournisseur. A partir de la deuxième offre, il devient alors possible d'utiliser notre coefficient de négociation en faisant la soustraction du minimum entre le budget maximum et le prix du ticket (un acheteur ne proposera jamais un montant au-dessus du prix du ticket) avec la première offre envoyée. Bien entendu, on veillera à prendre en compte la proposition transmise au départ au niveau du dénominateur en retirant 1 au nombre maximum d'offres, ce qui explique la présence de la constante 2. Reprenons l'exemple cité plus haut : la première offre de l'acheteur est égale à 800 - 800 * 0.2 = 640 € et coefNegotiation à 20 % divisée par 4 donc une variation de 4 % à partir de la deuxième offre.

Remarque 1 : une proposition de l'acheteur au-dessus du prix minimum du ticket n'engendre pas forcément un achat immédiat car le fournisseur procédera à la vente uniquement après 2 offres reçues pour faire monter les enchères et éviter de conclure la négociation dès la première offre.

Remarque 2 : une offre se base sur le budget maximum (acheteur) ou le prix minimum du ticket (fournisseur) lorsque le prix calculé est supérieur à ces contraintes.

b. Stratégies implémentées

A partir de la deuxième offre d'un agent, les stratégies de négociation entrent en jeu :

- **DEFAULT** consiste simplement à augmenter ou réduire (en fonction du type de l'agent) le dernier montant envoyé en utilisant le coefficient de négociation. Cette façon de procéder fait abstraction des circonstances (date, marché, etc.) et garantit un minimum d'équité entre un acheteur et un fournisseur car ils calculent les prix de la même manière.

- **TICKETS_SIMILARITY** se base sur le principe de "l'offre et la demande". Chaque agent inspecte le marché des tickets et note le nombre de tickets similaires (à l'instant t) par rapport au lieu de départ et d'arrivée. Si ce nombre n'est pas nul, le coefficient de négociation varie en fonction du profil de l'agent : il augmente de 3% pour un fournisseur afin de vendre rapidement le ticket et il diminue de 3% pour un acheteur entre afin de réduire le prix de ses offres et maximiser son gain.

- **REMAINING_TIME** permet de diminuer le prix du ticket et d'augmenter l'offre de l'acheteur si la date d'expiration est proche. Cette stratégie est souvent utilisée pour les produits périssables tels que les aliments ou les produits cosmétiques ayant une durée de vie limitée. En réduisant les prix de ces produits, le vendeur peut les écouler rapidement avant qu'ils n'expirent et ainsi minimiser les pertes. Du point de vue des acheteurs, nous avons décidé d'augmenter leur coefficient de négociation de 3% dans l'optique d'atteindre le budget maximal plus rapidement et acheter le ticket avant la date d'expiration (un voyage est un événement important qu'il ne faut pas rater).

c. Comparaison des stratégies

Dans l'optique de mettre en valeur l'impact de chaque stratégie de négociation, nous avons exécuté notre programme avec le scénario *ChangesOnDatesN_1* en spécifiant une stratégie différente à chaque fois. Il s'agit du seul scénario permettant d'appliquer la stratégie « REMAINING_TIME » car les dates limite de vente sont très proches.

Parmi toutes les négociations effectuées, nous avons porté notre attention sur l'échange ayant permis la vente du ticket dont la date limite est le 04/01/2023. Étant donné que les pourparlers démarrent le 01/01/2023, cela laisse le temps d'observer les conséquences de la proximité avec la date d'expiration. A l'instar des autres tickets, le prix de départ est de 800 €

et le prix minimum est de 600 €. Concernant l'acheteur, il possède un budget maximum de 650 €. Tout d'abord, commençons par la stratégie « DEFAULT » :

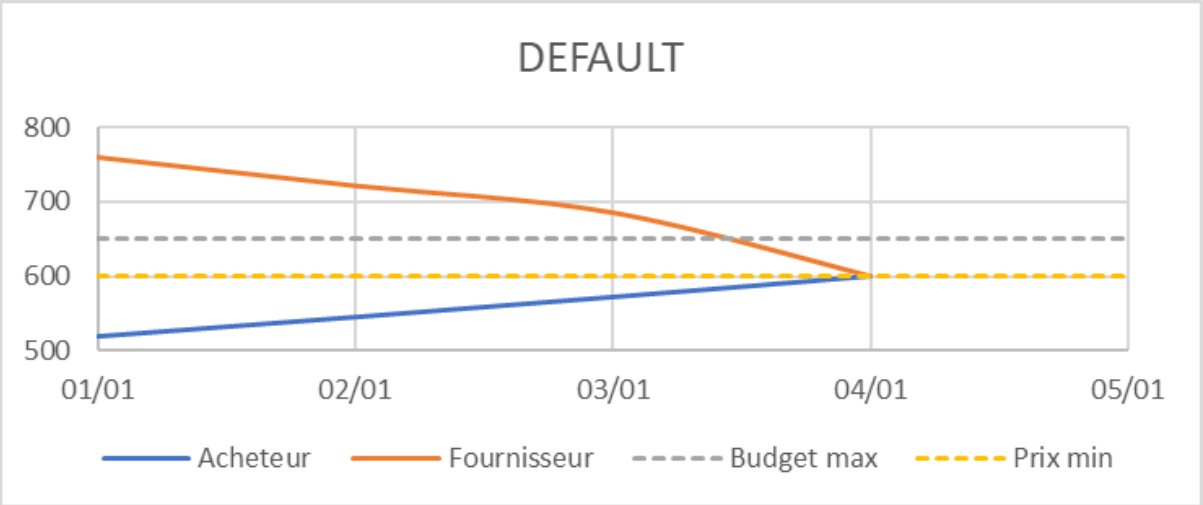

Figure 2 - Stratégie par défaut

Ce premier graphique montre que les offres de l'acheteur augmentent progressivement tandis que les propositions du fournisseur diminuent au fur et à mesure. Au bout d'un moment, les courbes finissent par se croiser : c'est le moment de l'achat. Ici, le montant de la transaction est de 601 € au 04/01.

Attention : aucune vente ne peut être effectué si le montant d'une offre ne se trouve pas dans l'intervalle $[prixMin;\ budgetMax]$ et que le nombre de propositions est inférieur à 2.

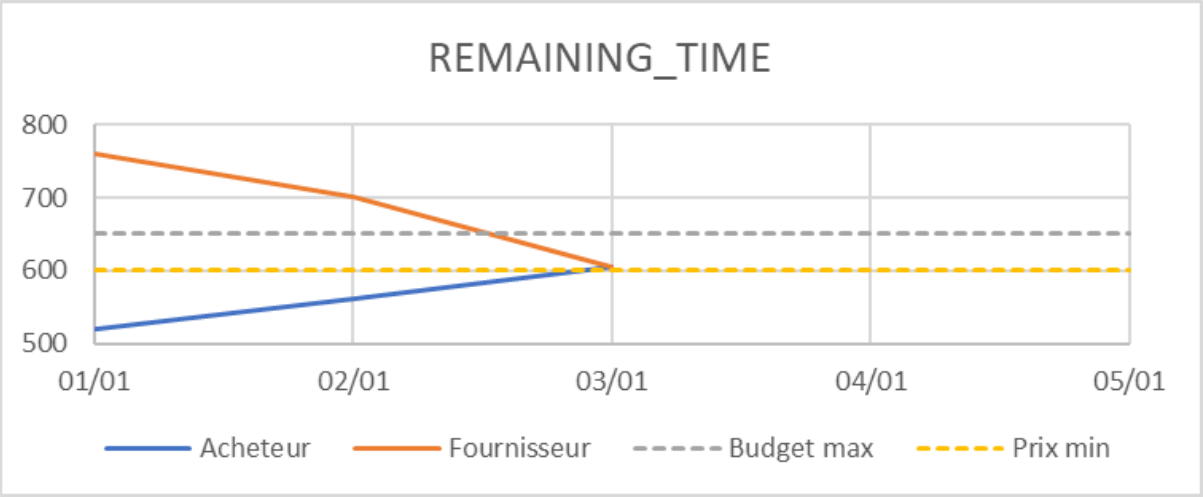

Figure 3 - Stratégie basée sur la date d'expiration d'un ticket

Par rapport au schéma précédent, on constate que la stratégie « REMAINING_TIME » aide l'acheteur à se procurer le ticket plus rapidement avec un jour d'avance (03/01). En revanche, ce dernier a payé 4 € de plus, soit 605 € au total, alors qu'il avait proposé 573 € au même moment avec la stratégie par défaut. Cela se justifie notamment par le nombre de jours restants avant l'expiration du ticket. A partir de 2 jours d'écart, l'acheteur est motivé à l'idée

de se procurer le ticket pour ne pas rater son voyage. Du côté du fournisseur, il a également fourni des efforts pour vendre rapidement son ticket étant donné qu'il propose 700 € contre 722 € avec la stratégie par défaut le deuxième jour.

Que ce soit la proposition de l'acheteur qui dépasse le prix minimum du ticket ou la contre-offre du fournisseur qui devient inférieur au budget maximum, ces deux événements ont les mêmes chances de se produire à l'approche de la date limite. On en déduit que cette stratégie n'est pas forcément avantageuse pour l'un ou l'autre. Financièrement parlant, elle est même désavantageuse pour les deux agents sur le long terme.

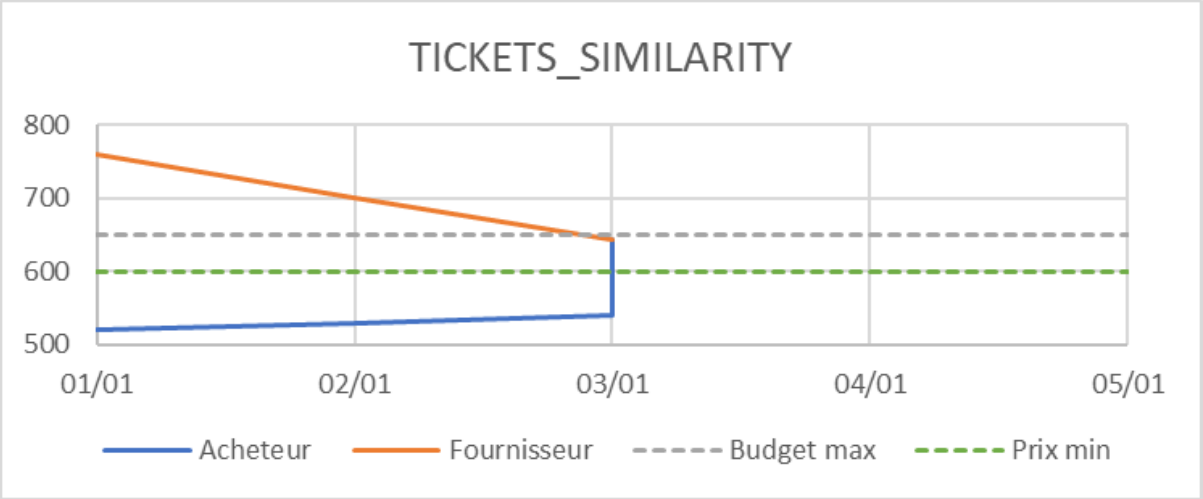

Figure 4 - Stratégie basée sur le nombre de tickets similaires

Dans l'illustration ci-dessus, on constate que le vendeur fait de fortes concessions en proposant 700 € dès le deuxième jour puis 644 € au troisième jour car le catalogue comporte 3 autres tickets similaires. Réciproquement, les offres de l'acheteur augmentent lentement puisqu'il se retrouve en position de force. Entre chaque proposition, le montant croît de 10 € seulement jusqu'au 03/01 avant d'accepter la proposition du vendeur.

Initialement, notre groupe pensait que cette stratégie de négociation serait clairement en faveur de l'acheteur. Cependant, les conditions nécessaires à une vente (respect du budget, prix minimum et nombre d'offres minimum) sont susceptibles d'être réunies plus rapidement grâce à l'effort du fournisseur sans avoir besoin de transmettre un montant dérisoire.

Dans la grande majorité des cas, l'acheteur acceptera toujours l'offre du vendeur et non l'inverse. Dès lors, on en déduit que le fournisseur mène la danse au niveau du prix et que la stratégie « TICKETS_SIMILARITY » n'est pas forcément avantageuse pour le client. Elle est même favorable au vendeur lorsque sa proposition est très proche du budget maximum de l'interlocuteur (en l'occurrence 650 € ici).

Après une comparaison de ces 3 stratégies de négociation, il est possible d'affirmer que :

- DEFAULT est la meilleure approche en termes d'équité et de gain. Il y a autant de chances qu'une offre de l'acheteur soit acceptée (vente) qu'il y a de chances qu'une proposition du vendeur soit approuvée (achat).

- REMAINING_TIME favorise l'écoulement des stocks et l'obtention du voyage au détriment du gain.

- TICKETS_SIMILARITY aide le fournisseur à vendre ses tickets rapidement malgré la concurrence et la position de force de l'acheteur.

VII. Fonctionnalités globales de l'application

Comme demandé à l'issue de la dernière séance de TP, nous avons fait l'inventaire de ce qui fonctionne et ce qui ne fonctionne pas :

Fonctionnalité attendue	Fonctionnelle	Non fonctionnelle
Établir différents types de communication entre les agents : 1-N, N-1 et N-N.	✓	
Gérer les contraintes et les préférences d'un agent.	✓	
Fixer un nombre maximum d'offres possibles.	✓	
Créer une fonction de décision prenant en compte des stratégies de négociation.	✓	
Garder un historique des offres échangées entre les agents pendant une négociation.	✓	
Mettre en place un protocole de négociation.	✓	
Prendre en compte la disponibilité d'un ticket et d'un acheteur lors d'une négociation.	✓	
Sécuriser parfaitement l'accès aux méthodes et attributs d'une classe.		✗

Par rapport au dernier point, notre solution présente quelques imperfections mineures au niveau de la visibilité des propriétés d'une classe par d'autres classes. A titre d'illustration, un client est en mesure d'accéder au prix minimum du ticket. Bien entendu, il ne prend jamais en compte ce montant durant l'élaboration d'une offre mais cette donnée reste accessible, ce qui ne devrait pas être le cas. Pour pallier ce problème, il pourrait être judicieux de stocker le prix minimum du ticket dans une map définie dans la classe Provider en définissant le ticket en clé et le prix minimum en valeur.

VIII. Conclusion

a. Bilan

Pour conclure, il est possible de dire que nous avons pris beaucoup de plaisir à travailler sur la conception de ce système de négociation automatique multi-agents. C'était très formateur sur plusieurs aspects et chacun d'entre nous est monté en compétences tout au long de ces dernières semaines. Plusieurs séances en mode « pair programming » ont eu lieu pour faire avancer le TP tous ensemble. Notre bagage de connaissances s'est notamment élargi à travers la mise en pratique du multithreading et des structures permettant de gérer les modifications concurrentes d'un objet en situation de compétition (race condition).

b. Améliorations possibles

Pour aller plus loin, il serait intéressant d'intégrer quelques points d'amélioration afin de bonifier la qualité du code. En premier lieu, il faudrait permettre à un vendeur (ou à l'utilisateur) d'ajouter d'autres billets au fil du temps et d'ajouter de nouveaux acheteurs dynamiquement durant l'exécution afin de rendre le programme plus « vivant ». Ensuite, il aurait été pertinent de dissocier les stratégies de négociations entre les agents pour ajouter une part de stochasticité et dynamiser davantage le code.

Actuellement, un thread associé à un acheteur s'arrête lorsqu'un ticket a été acheté. Ce comportement manque légèrement de réalisme car un client pourrait très bien avoir envie d'acheter un aller-retour ou plusieurs tickets pour des proches. De ce fait, ajouter une nouvelle contrainte avec les variables associés (nombre de tickets souhaités, liste de billets recherchés, …) dans la classe Buyer pourrait être une bonne idée.

En raison de la charge de travail de ce semestre, le professeur a retiré quelques contraintes du sujet du TP. Parmi les tâches initialement prévues, une classe d'évaluation permettant de comparer les stratégies de négociation aurait dû être introduite. Instaurer une telle classe pourrait renforcer le niveau de l'étude réalisée au sein de ce rapport. Mais encore, la mise en place de coalition afin de faire passer un cap aux agents était également envisagé au départ. A titre d'illustration, les clients auraient été amenés à initier des négociations groupées (regroupement en fonction des objectifs et des contraintes) à travers un représentant. Cette idée permettrait d'accentuer l'importance accordée par le vendeur aux contraintes de son interlocuteur. Dans le même esprit, il serait également concevable d'unir les fournisseurs en vue de réaliser des actions communes comme une harmonisation des prix selon les critères d'un ticket pour atténuer l'aspect concurrentiel et être sur le même piédestal en termes d'équité.

Par ailleurs, une interface graphique pourrait également faciliter le paramétrage de l'application (stratégie de négociation, scénarios de test, ...) et simplifier la visualisation des négociations entre les agents. Enfin, il serait pertinent de mettre en place des tests unitaires dans l'optique de garantir le bon fonctionnement des méthodes implémentées.

Bibliographie

KOUIS, Amine. « Différence entre l'interface Runnable et la classe Thread en java ». WayToLearnX, 13 septembre 2018. https://waytolearnx.com/2018/09/difference-entre-linterface-runnable-et-la-classe-thread-en-java.html

baeldung. « Guide to Java.Util.Concurrent.BlockingQueue ». Baeldung, 22 décembre 2021. https://www.baeldung.com/java-blocking-queue

GEMMAIL, Raf. « Implement a Producer-Consumer Pattern Using a BlockingQueue ». OpenClassrooms. 8 novembre 2022. https://openclassrooms.com/fr/courses/5684021-scale-up-your-code-with-java-concurrency/6667996-implement-a-producer-consumer-pattern-using-a-blockingqueue

teto. « Answer to "Volatile boolean vs AtomicBoolean" ». Stack Overflow, 24 septembre 2010. https://stackoverflow.com/a/3787435

baeldung. « Migrating to the New Java 8 Date Time API ». Baeldung, 19 avril 2020. https://www.baeldung.com/migrating-to-java-8-date-time-api

KANJILAL, Joydip. « Introduction to ConcurrentHashMap in Java ». Developer.com, 24 juillet 2022. https://www.developer.com/java/concurrenthashmap-java/

RAM, Vikyath. « Difference between ArrayList and CopyOnWriteArrayList in Java ». TutorialsPoint, 21 juin 2020. https://www.tutorialspoint.com/Difference-between-ArrayList-and-CopyOnWriteArrayList-in-Java

YUAN, Kai. « Start Two Threads at the Exact Same Time in Java ». Baeldung, 22 décembre 2021. https://www.baeldung.com/java-start-two-threads-at-same-time

DOUDOUX, Jean-Michel. « La gestion des accès concurrents ». Développons en Java. https://www.jmdoudoux.fr/java/dej/chap-acces_concurrents.htm

PEREZ, Emilie. « Les stratégies de négociation ». Cours BTS Communication (blog), 21 août 2020. https://cours-bts-communication.fr/strategies-negociation/

COLAS, Timothé. « Comment choisir sa stratégie de négociation ? ». Walter Learning, 10 janvier 2023. https://walter-learning.com/blog/soft-skills/comment-choisir-sa-strategie-de-negociation

« Offre et demande — Analyse de la tendance » [en ligne]. Trading View. https://fr.tradingview.com/ideas/supplyanddemand/