Système Multi-Agents avec JADE

-

Université de Caen Normandie

UFR SCIENCES

M1 Informatique - SMINF1F5

Auteurs : Hadjer CHEDJARI EL MEUR
Etienne BOSSU

Date : Janvier 2026

Table des matières

1 Introduction

1.1 Contexte

Ce projet s'inscrit dans l'UE "Systèmes Multi-Agents" et vise à implémenter une solution décentralisée à un problème d'ordonnancement dans un atelier de production en utilisant la plateforme JADE.

1.2 Objectifs

— Implémenter une solution **décentralisée** au problème d'ordonnancement
— Adapter le **Contract Net Protocol** pour l'allocation de tâches
— Coordonner des agents autonomes via négociation par enchères
— Optimiser l'utilisation des ressources (robots)

1.3 Problématique

Des robots aux compétences diverses doivent collaborer pour fabriquer des produits, en gérant :

— L'arrivée des produits
— La variabilité des compétences
— Les possibles échecs d'exécution
— Les files d'attente de chaque robot

2 Analyse du Problème

2.1 Contraintes du Système

#	**Contrainte**	**Statut**
1	1 agent Atelier et m agents Robots	✓
2	N compétences distinctes	✓
3	Arrivée produit $\in [\lambda_1, \lambda_2]$	✓
4	Chaque produit nécessite N_j compétences	✓
5	Robots avec S_i compétences, $p_{s_i,k} \in]0, 1[$	✓
6	Robots peuvent partager compétences	✓
7	Probabilité $p_{s_i,k}$ de succès	✓
8	Robot qui échoue recommence	✓
9	Un robot = un produit à la fois	✓
10	File d'attente possible	✓
11	Communication non bloquante	✓
12	Pas de traitement simultané	✓

TABLE 1 – État d'implémentation (✓= Implémenté, × = Non implémenté)

2.2 Paramètres

— m : Nombre de robots (défaut : 3)

— N : Compétences disponibles (5)
— N_j : Compétences par produit (3)
— S_i : Compétences par robot (3)
— λ_1, λ_2 : Intervalle génération (1000-2000ms)
— λ_3 : Temps exécution (500ms)
— $p_{s_i,k}$: Probabilité succès $\in [0.5, 1.0]$

2.3 Contract Protocol

1. **Génération** : Atelier → robot aléatoire
2. **Enchère** : Robot lance CFP pour compétence manquante
3. **Soumission** : Robots qualifiés proposent makespan
4. **Attribution** : Sélection minimum
5. **Exécution** : Robot gagnant traite
6. **Itération** : Jusqu'à complétion
7. **Retour** : Produit fini → atelier

3 Choix de Conception

3.1 Représentation des Produits

Classe Product :

```java
public class Product implements Serializable {
    private String id;
    private HashMap<String, Boolean> requiredSkills;
    private long startTime;
    private int nbFailures = 0;

    // Methodes: isFinished(), getNextMissingSkill(),
    //           setSkillDone(skill), addFailure()
}
```

Justification :

— **HashMap** : Permet de suivre dynamiquement l'état de chaque compétence requise (réalisée ou non) par des valeurs booléennes.
— **Sérialisation** : L'implémentation de l'interface `Serializable` est requise pour transmettre l'objet complet dans le contenu des messages ACL entre agents.
— **Statistiques** : Les attributs `startTime` et `nbFailures` assurent le suivi précis du temps de cycle et de la fiabilité du processus pour chaque produit.

3.2 Représentation des Compétences et Annuaire

Les compétences sont représentées par des identifiants textuels allant de `"0"` à `"4"`. Chaque robot stocke ses aptitudes dans une `HashMap<String, Double>` associant l'identifiant du skill à sa probabilité de succès $p_{s_i,k}$.

La découverte des services repose sur le **Directory Facilitator (DF)** de JADE :

— **Service général** : Chaque robot publie un service `"robot-service"` pour signaler sa présence dans l'atelier.
— **Services spécifiques** : Un service distinct est enregistré pour chaque compétence possédée (ex : `"skill-2"`), permettant au Manager de cibler uniquement les robots qualifiés lors d'un appel d'offres.

3.3 Justification des Behaviours

L'architecture logicielle utilise différents types de comportements JADE pour garantir l'asynchronisme et la simulation temporelle.

3.3.1 AtelierAgent

`GenerateProductBehaviour` **(WakerBehaviour)**
Introduit un délai aléatoire compris entre λ_1 et λ_2 avant la création de chaque produit, se reprogrammant lui-même pour assurer un flux continu.

`ReceiveFinishedProductBehaviour` **(CyclicBehaviour)**
Maintient une écoute constante de la boîte aux lettres pour traiter les retours de produits finis sans interrompre les autres tâches de l'agent.

3.3.2 RobotAgent

`ReceiveNewProductBehaviour` & `ResponderBehaviour`
Comportements cycliques dédiés à l'interception des messages INFORM, CFP et ACCEPT. Ils garantissent que le robot reste disponible pour la négociation même en période d'activité.

`ManagerBehaviour` **(OneShotBehaviour)**
Exécuté ponctuellement lorsqu'un produit nécessite une compétence externe pour initier le protocole d'appel d'offres via le DF.

`CollectProposalsBehaviour` **(WakerBehaviour)**
Définit une fenêtre temporelle fixe de 1000ms pour collecter les propositions des robots avant de désigner le vainqueur de l'enchère.

`WorkerBehaviour` **(TickerBehaviour)**
Scanne la file d'attente toutes les 200ms. Le travail effectif est simulé par un `WakerBehaviour` de durée λ_3 pour éviter de geler l'exécution des autres comportements.

3.4 Calcul du Makespan et Espérance

Lors des enchères, les robots estiment leur temps de traitement restant (makespan). Ce calcul intègre l'espérance mathématique d'une loi géométrique pour modéliser l'impact des échecs potentiels :

$$E(\text{essais}) = \frac{1}{p} \implies \text{Échecs moyens } (E) = \frac{1-p}{p} \tag{1}$$

Implémentation :

```
double E = (1.0 - prob) / prob;
// Estimation : temps nominal multiplie par le nombre d'essais
    esperes
totalTime += lambda3 * (1.0 + E);
```

4 Protocole de Communication

4.1 Messages FIPA-ACL

Le système s'appuie sur le standard FIPA-ACL pour assurer l'interopérabilité entre les agents. Le tableau suivant récapitule les échanges au sein de l'atelier :

Performative	**De**	**Vers**	**Contenu (Object/String)**
INFORM	Atelier	Robot	Objet `Product` initial
INFORM	Robot	Atelier	Objet `Product` terminé
CFP	Manager	Workers	Identifiant `"skill-X"`
PROPOSE	Worker	Manager	Valeur du `makespan` estimé
ACCEPT_PROPOSAL	Manager	Winner	Objet `Product` à traiter

TABLE 2 – Synthèse des interactions FIPA-ACL

4.2 ConversationId et Unicité

Afin de permettre aux agents de distinguer plusieurs enchères se déroulant simultanément, chaque protocole de négociation est identifié par une clé unique.

Format implémenté : `"nego-" + productId + "-" + timestamp`

L'ajout du `System.currentTimeMillis()` garantit qu'un même produit, s'il doit subir plusieurs phases de négociation successives pour différents skills, ne créera pas de confusion dans les boîtes aux lettres des robots ouvriers.

4.3 MessageTemplate et Filtrage

L'utilisation des `MessageTemplate` est systématique pour assurer que chaque comportement ne traite que les messages qui lui sont destinés, sans perturber les autres files d'attente de l'agent.

Listing 1 – Exemples de filtrage utilisés dans le projet

```
// Filtrage simple par performative (Reception de produit)
MessageTemplate.MatchPerformative(ACLMessage.INFORM)

// Filtrage combine pour la collecte des offres
MessageTemplate.and(
    MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
    MessageTemplate.MatchConversationId(conversationId)
)
```

4.4 Suivi de la charge réseau

Conformément aux exigences du sujet, chaque envoi de message déclenche l'incrémentation d'un compteur global situé dans la classe `Utils`. Ce mécanisme permet à l'Atelier d'afficher le coût total en communications de la simulation à chaque produit terminé.

5 Implémentation Détaillée

Cette section détaille les mécanismes internes et les algorithmes mis en œuvre pour assurer la coordination et la robustesse du système.

5.1 AtelierAgent : Flux de Production

L'Atelier gère le cycle de vie initial et final des produits. Son attribut principal est un objet `Random` permettant de simuler l'aléa de production.

Algorithme de Génération :

1. Instanciation de `Product` avec un ID incrémental.
2. Appel à `Utils.getRandomRobot()` : effectue un `DFService.search` pour trouver les robots disponibles et en sélectionne un par tirage uniforme.
3. Transmission du produit par un message `INFORM`.
4. Calcul du prochain délai via : $\lambda_1 + (rnd.nextDouble() \times (\lambda_2 - \lambda_1))$.

Listing 2 – Reprogrammation du comportement de génération

```
protected void onWake() {
    Product p = new Product("P-" + (++productCount), 5, 3);
    AID robot = Utils.getRandomRobot(myAgent);
    if (robot != null) {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(robot);
        msg.setContentObject(p);
        send(msg);
    }
    // Planification du prochain produit (Boucle infinie)
    myAgent.addBehaviour(new GenerateProductBehaviour(myAgent,
        getRandomTime()));
}
```

5.2 RobotAgent : Négociation et Exécution

5.2.1 Initialisation et Compétences

Lors du `setup()`, chaque robot s'attribue 3 compétences aléatoires. Pour chaque compétence, une probabilité de succès $p \in [0.5, 1.0]$ est définie. Cette valeur est critique car elle influence directement les offres faites lors des enchères.

5.2.2 Cycle de Négociation (Contract Net)

Lorsqu'un produit nécessite une compétence que le détenteur actuel ne possède pas (ou n'a plus besoin), l'agent bascule en rôle de **Manager** :

— **Appel d'offres** : Envoi d'un `CFP` à tous les agents proposant le service `"skill-X"` dans l'annuaire.

— **Évaluation** : Le `CollectProposalsBehaviour` reçoit les propositions (`makespan`) et sélectionne l'agent ayant le temps de traitement estimé le plus court.

— **Transfert** : Envoi d'un `ACCEPT_PROPOSAL` contenant l'objet `Product` sérialisé.

5.2.3 Exécution et Gestion des Échecs

Le travail effectif est géré par le `WorkerBehaviour`. C'est ici qu'est implémentée la logique de résilience demandée par le sujet.

Logique de reprise sur erreur :

```java
// Dans le onWake() du simulateur de travail
double prob = skills.get(skillToApply);
if (Math.random() < prob) {
    // SUCCES : Mise a jour de l'etat et redirection
    currentProduct.setSkillDone(skillToApply);
    myAgent.addBehaviour(new ManagerBehaviour(currentProduct));
} else {
    // ECHEC : Incrementation du compteur et reinsertion prioritaire
    currentProduct.addFailure();
    productQueue.add(0, currentProduct);
}
isWorking = false;
```

5.3 Suivi des Statistiques et Performance

Le système intègre un monitoring en temps réel pour évaluer l'efficacité de l'ordonnancement.

5.3.1 Analyse de la Fiabilité

L'échec est traité comme une donnée métier. Le compteur `nbFailures` est encapsulé dans le `Product` et voyage avec lui. À l'arrivée, l'Atelier calcule la moyenne globale, permettant d'identifier si la difficulté des tâches est en adéquation avec les capacités des robots.

5.3.2 Coût de Communication

Pour mesurer la charge réseau sans surcharger le système, nous utilisons un `AtomicInteger` dans une classe utilitaire.

— **Avantage** : Garantit l'intégrité du compteur malgré les accès concurrents des agents.

— **Calcul** : Chaque message (`send`) incrémente le total. Cela permet de corréler le nombre de messages avec le nombre de robots.

6 Diagrammes

6.1 Diagramme de Séquence

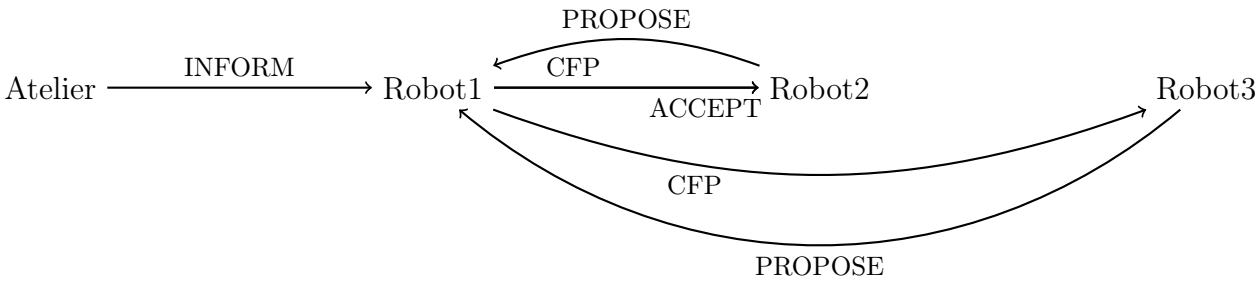

FIGURE 1 – Flux de communication simplifié

6.2 Diagramme d'États RobotAgent

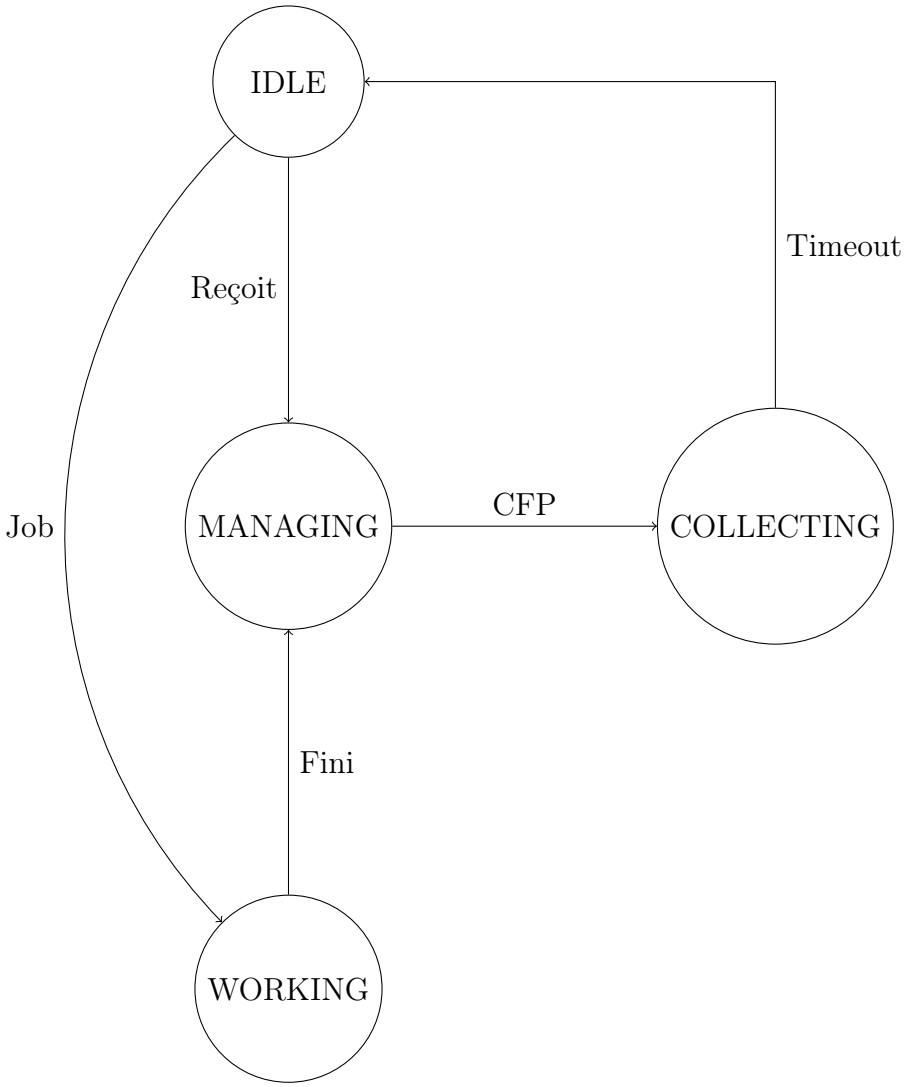

FIGURE 2 – États principaux d'un RobotAgent

7 Guide d'Utilisation

Le projet est fourni avec une structure prête à l'emploi. Les dépendances JADE sont situées dans le dossier `lib/` et les sources dans le dossier `src/`.

7.1 Scripts de Lancement

Des scripts automatisés sont fournis pour simplifier l'exécution et la génération de la documentation technique.

7.1.1 Exécution (run.bat / run.sh)

Ces scripts permettent de lancer la simulation avec des paramètres par défaut ou personnalisés.

Usage :

```
# Sous Windows
run.bat [nbRobots] [lambda1] [lambda2] [lambda3]

# Sous Linux/Mac
./run.sh [nbRobots] [lambda1] [lambda2] [lambda3]
```

7.1.2 Documentation (doc.bat / doc.sh)

Ces scripts génèrent la documentation Javadoc.

```
# Generation de la Javadoc
./doc.sh
```

7.2 Configuration de la Simulation

Le programme `Main` accepte les paramètres suivants en ligne de commande :

1. **nbRobots** : Nombre d'agents Robots à instancier (défaut : 3).
2. **lambda1** / **lambda2** : Intervalle de génération des produits pour l'Atelier.
3. **lambda3** : Temps de traitement nominal pour les robots.

7.3 Sorties Console et Traces

Les traces d'exécution permettent de suivre en temps réel les négociations et l'état des stocks de chaque robot.

Listing 3 – Traces d'execution et statistiques de production

```
Atelier Atelier pret.
Robot Robot1 pret. Competences: {1=0.80, 4=0.54}
Robot Robot2 pret. Competences: {2=0.67, 4=0.96}
Atelier : Produit P-1 envoye a Robot2

V [Worker Robot2] Produit recu : P-1 (File: 1)
*** [Worker Robot2] SUCCES sur P-1
!!! [Manager Robot2] Aucun robot trouve pour le skill 3

--------------------------------------------------
--- PRODUIT FINI : P-1 ---
   > Termine par       : Robot2
   > Temps de cycle    : 4768 ms
   > Echecs sur ce P   : 0
   > Moyenne echecs    : 0.00
   > Total Messages    : 37
--------------------------------------------------
```

8 Analyse des Résultats et Conclusion

8.1 Bilan Technique et Points Forts

Le système multi-agents développé remplit l'ensemble des objectifs fixés par le cahier des charges. L'approche décentralisée, portée par la plateforme JADE, a permis de créer un environnement de production autonome.

Parmi les points forts de cette implémentation, nous pouvons noter :

— **Robustesse et réalisme** : L'intégration d'une logique d'échec et la réinsertion prioritaire des produits en file d'attente simule fidèlement les aléas d'un atelier réel.

— **Optimisation par le Makespan** : Le calcul du temps de traitement incluant l'espérance mathématique des échecs permet une allocation intelligente des tâches, favorisant les robots les plus performants.

— **Monitoring précis** : Grâce à l'instrumentation de la classe `Utils` et l'usage de compteurs `AtomicInteger`, nous obtenons un suivi rigoureux de la charge réseau et du coût de communication du protocole *Contract Net*.

8.2 Limites et Perspectives d'Évolution

Malgré l'efficacité globale du système, l'analyse des logs a mis en évidence une vulnérabilité critique liée à la distribution aléatoire des compétences. Si une compétence n'est possédée par aucun robot au démarrage, le système entre dans une impasse où certains produits ne peuvent jamais être terminés.

Pour résoudre ce problème ou améliorer ce projet, plusieurs améliorations sont envisageables :

— **Contrôle de couverture** : Implémenter un agent superviseur vérifiant la présence de tous les services nécessaires dans l'annuaire DF avant d'autoriser le lancement de la production par l'Atelier.

— **Interface Graphique** : Bien que les traces console soient complètes, une interface de visualisation en temps réel permettrait de mieux observer les flux de produits entre les robots.