Université de Caen Normandie

Département d'Informatique

M1 Informatique - SMINF1F5

Système Multi-Agents de Production Robotisée

avec JADE

Contract Net Protocol
Ordonnancement Décentralisé

Auteurs : Hadjer CHEDJARI EL MEUR
Etienne BOSSU
Encadrant : Grégory Bonnet
Date : Janvier 2026

Systèmes Multi-Agents

Table des matières

1 Introduction

1.1 Contexte

Ce projet s'inscrit dans l'unité SMINF1F5 "Systèmes Multi-Agents" et vise à implémenter une solution décentralisée à un problème d'ordonnancement dans un atelier de production automatisé 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 :

```
public class Product implements Serializable {
    private String id;
    private HashMap<String, Boolean> requiredSkills;

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

Justification :

— HashMap flexible pour suivi d'état
— Serializable pour transmission ACL
— Génération aléatoire de 3/5 compétences

3.2 Représentation des Compétences

Format : Chaînes `"0"` à `"4"` pour $N = 5$.
Stockage robot :

```
HashMap<String, Double> skills; // skill -> p_{s_i,k}
```

Enregistrement DF :

— Service général : `"robot-service"`
— Services spécifiques : `"skill-0"` à `"skill-4"`

3.3 Behaviours et Justification des Choix

Le choix des types de comportements est crucial pour assurer la réactivité des agents et la simulation du temps.

3.4 Justification des Behaviours

3.4.1 AtelierAgent

`GenerateProductBehaviour` **(WakerBehaviour)**
Utilisé pour introduire un délai aléatoire $[\lambda_1, \lambda_2]$ avant chaque création de produit. L'agent se reprogramme lui-même à la fin pour créer une boucle de génération infinie.

`ReceiveFinishedProductBehaviour` **(CyclicBehaviour)**
Permet une écoute permanente et non-bloquante de la boîte aux lettres. Indispensable pour récupérer les produits finis dès qu'ils arrivent, quel que soit le moment.

3.4.2 RobotAgent

`ReceiveNewProductBehaviour` & `ResponderBehaviour` **(CyclicBehaviour)**
Assurent la réactivité du robot. Ils tournent en boucle pour intercepter immédiatement les nouveaux produits (INFORM) ou les appels d'offres (CFP) sans jamais s'arrêter.

`ManagerBehaviour` **(OneShotBehaviour)**
Lancé uniquement "à la demande" quand une compétence manque. Il effectue une action ponctuelle (lancer l'enchère et contacter le DF) puis se détruit immédiatement pour libérer des ressources.

`CollectProposalsBehaviour` **(WakerBehaviour)**
Agit comme un "Timer" pour l'enchère. Il attend exactement 1500ms (le temps de laisser les autres répondre), puis se déclenche une seule fois pour dépouiller les votes et choisir le vainqueur.

`WorkerBehaviour` **(TickerBehaviour)**
S'exécute périodiquement (toutes les 200ms) pour vérifier la file d'attente et faire avancer le travail. Contrairement à une boucle `while` qui bloquerait l'agent, ce "battement de cœur" permet au robot de continuer à recevoir des messages pendant qu'il travaille.

3.5 Calcul du Makespan

Formule théorique :

$$\text{makespan} = \sum_{\text{produits}} \lambda_3 \times (1 + E), \quad E = \frac{1 - p_{s_i,k}}{p_{s_i,k}} \tag{1}$$

Implémentation : Nous utilisons la formule de l'espérance d'une loi géométrique pour estimer le temps réel incluant les échecs potentiels :

```
double E = (1.0 - prob) / prob;
// Temps espere = temps nominal * (1 + nb_echecs_moyens)
totalTime += lambda3 * (1.0 + E);
```

4 Protocole de Communication

4.1 Messages FIPA-ACL

Performative	De	Vers	Contenu
INFORM	Atelier	Robot	Produit
INFORM	Robot	Atelier	Produit fini
CFP	Manager	Workers	"skill-X"
PROPOSE	Worker	Manager	makespan
ACCEPT_PROPOSAL	Manager	Winner	Product

TABLE 2 – Messages utilisés

4.2 ConversationId

Format : `"nego-" + productId`
Exemple : `"nego-P-1"`, `"nego-P-2"`

4.3 MessageTemplate

```
// Filtrage par performative
MessageTemplate.MatchPerformative(ACLMessage.CFP)

// Filtrage combine
MessageTemplate.and(
    MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
    MessageTemplate.MatchConversationId("nego-P-5")
)
```

5 Implémentation Détaillée

5.1 AtelierAgent

5.1.1 Attributs

```
private long lambda1 = 1000, lambda2 = 2000;
private int productCount = 0;
private Random rnd = new Random();
```

5.1.2 GenerateProductBehaviour

Algorithme :

1. Créer Product("P-{++count}", 5, 3)
2. Trouver robot aléatoire via DF
3. Envoyer INFORM avec product
4. Réactiver après délai $\in [\lambda_1, \lambda_2]$

```
protected void onWake() {
    Product p = new Product("P-" + (++productCount), 5, 3);
    AID robot = Utils.getRandomRobot(myAgent);

    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
    msg.addReceiver(robot);
    msg.setContentObject(p);
    send(msg);

    myAgent.addBehaviour(new GenerateProductBehaviour(
        myAgent, getRandomTime()));
}
```

5.2 RobotAgent

5.2.1 Initialisation

```
protected void setup() {
    // 1. Recuperer lambda3
    lambda3 = Long.parseLong((String)args[0]);

    // 2. Generer 3 competences aleatoires
    for(int i=0; i<3; i++) {
        int skillId = (int)(Math.random() * 5);
        skills.put(String.valueOf(skillId),
                   0.5 + Math.random() * 0.5);
    }

    // 3. Enregistrer au DF
    DFService.register(this, createDFD());

    // 4. Lancer behaviours
    addBehaviour(new ReceiveNewProductBehaviour());
    addBehaviour(new ResponderBehaviour());
    addBehaviour(new WorkerBehaviour());
}
```

5.2.2 ManagerBehaviour

Algorithme :

1. Obtenir skill = product.getNextMissingSkill()
2. Si skill == null : INFORM → Atelier, RETURN
3. Rechercher robots avec "skill-X" dans DF
4. Envoyer CFP à tous
5. Lancer CollectProposalsBehaviour

```
public void action() {
    String skill = product.getNextMissingSkill();
    if (skill == null) {
        // Renvoyer a l'atelier
        sendINFORM(atelier, product);
        return;
```

```
    }

    DFAgentDescription[] robots = searchDF("skill-" + skill);
    sendCFP(robots, "skill-" + skill, "nego-" + product.getId());
    addBehaviour(new CollectProposalsBehaviour(...));
}
```

5.2.3 CollectProposalsBehaviour

```
protected void onWake() {
    MessageTemplate mt = and(
        MatchPerformative(PROPOSE),
        MatchConversationId(conversationId)
    );

    ACLMessage best = null;
    double bestTime = Double.MAX_VALUE;

    while ((msg = receive(mt)) != null) {
        double time = Double.parseDouble(msg.getContent());
        if (time < bestTime) {
            bestTime = time;
            best = msg;
        }
    }

    if (best != null) {
        sendACCEPT_PROPOSAL(best.getSender(), product);
    }
}
```

5.2.4 ResponderBehaviour

```
public void action() {
    ACLMessage msg = receive(cfpOrAccept);

    if (msg.getPerformative() == CFP) {
        double mk = calculateMakespan();
        sendPROPOSE(msg.getSender(), mk);
    }
    else if (msg.getPerformative() == ACCEPT_PROPOSAL) {
        Product p = (Product) msg.getContentObject();
        productQueue.add(p);
    }
}
```

5.2.5 WorkerBehaviour

```
protected void onTick() {
    if (!isWorking && !productQueue.isEmpty()) {
        // Recuperation du produit en tete de file
        Product currentProduct = productQueue.remove(0);
        String skillToApply = currentProduct.getNextMissingSkill();

```

```
        if (skills.containsKey(skillToApply)) {
            isWorking = true;

            // Simulation du temps de travail
            myAgent.addBehaviour(new WakerBehaviour(myAgent, lambda3)
                {
                protected void onWake() {
                    double prob = skills.get(skillToApply);

                    // GESTION DE L'ECHEC (Loi de Bernoulli)
                    if (Math.random() < prob) {
                        // SUCCES : On valide et on passe a la suite
                        currentProduct.setSkillDone(skillToApply);
                        myAgent.addBehaviour(new ManagerBehaviour(
                            currentProduct));
                    } else {
                        // ECHEC : Reinsertion prioritaire (index 0)
                        productQueue.add(0, currentProduct);
                    }
                    isWorking = false;
                }
            });
        }
    }
}
```

5.3 Suivi des Statistiques

Pour répondre aux exigences de suivi des performances (nombre moyen d'échecs et charge réseau), nous avons instrumenté le code de la manière suivante :

5.3.1 Métriques d'Échec

Le nombre d'échecs est une propriété intrinsèque à la complexité de réalisation d'un produit. Nous avons donc choisi d'encapsuler cette donnée directement dans la classe `Product`.

- Chaque produit possède un compteur `nbFailures` initialisé à 0.
- Lors d'un échec dans le `WorkerBehaviour` du robot, la méthode `product.addFailure()` est appelée avant la réinsertion dans la file.
- À la réception du produit fini, l'`AtelierAgent` agrège ces données pour calculer une moyenne glissante :

$$\text{Moyenne} = \frac{\sum \text{Échecs}}{\text{Nombre Produits Finis}}$$

5.3.2 Charge Réseau (Nombre de Messages)

Pour quantifier les échanges du protocole Contract Net, nous utilisons un compteur statique thread-safe (`AtomicInteger`) partagé via la classe `Utils`. Bien que moins "purement" distribuée, cette approche est optimale pour une simulation s'exécutant au sein d'une même JVM, évitant la lourdeur d'un agent "Sniffer" dédié. Chaque appel à `send()` dans les agents incrémente ce compteur global, offrant une vue temps-réel de la charge de communication.

Listing 1 – Affichage des statistiques dans l'Atelier

```
// Exemple de sortie console implementee
System.out.println("> Moyenne Echecs : " + String.format("%.2f", avg)
    );
System.out.println("> Total Messages : " + Utils.totalMessages.get())
    ;
```

6 Diagrammes

6.1 Diagramme de Séquence

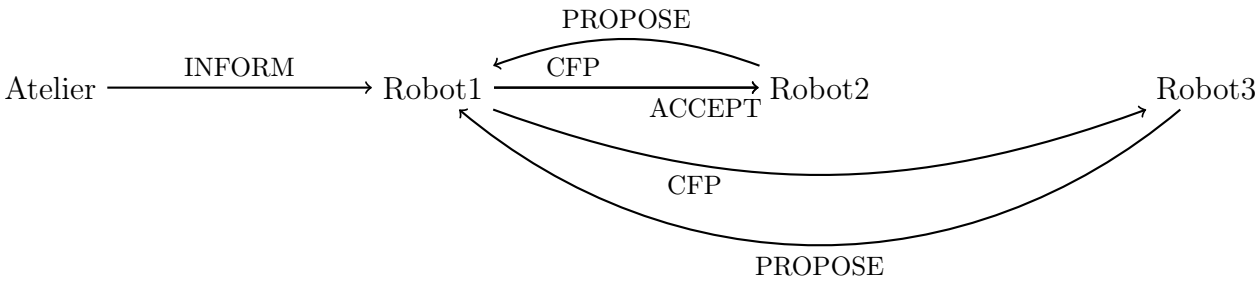

FIGURE 1 – Flux de communication simplifié

6.2 Diagramme d'États RobotAgent

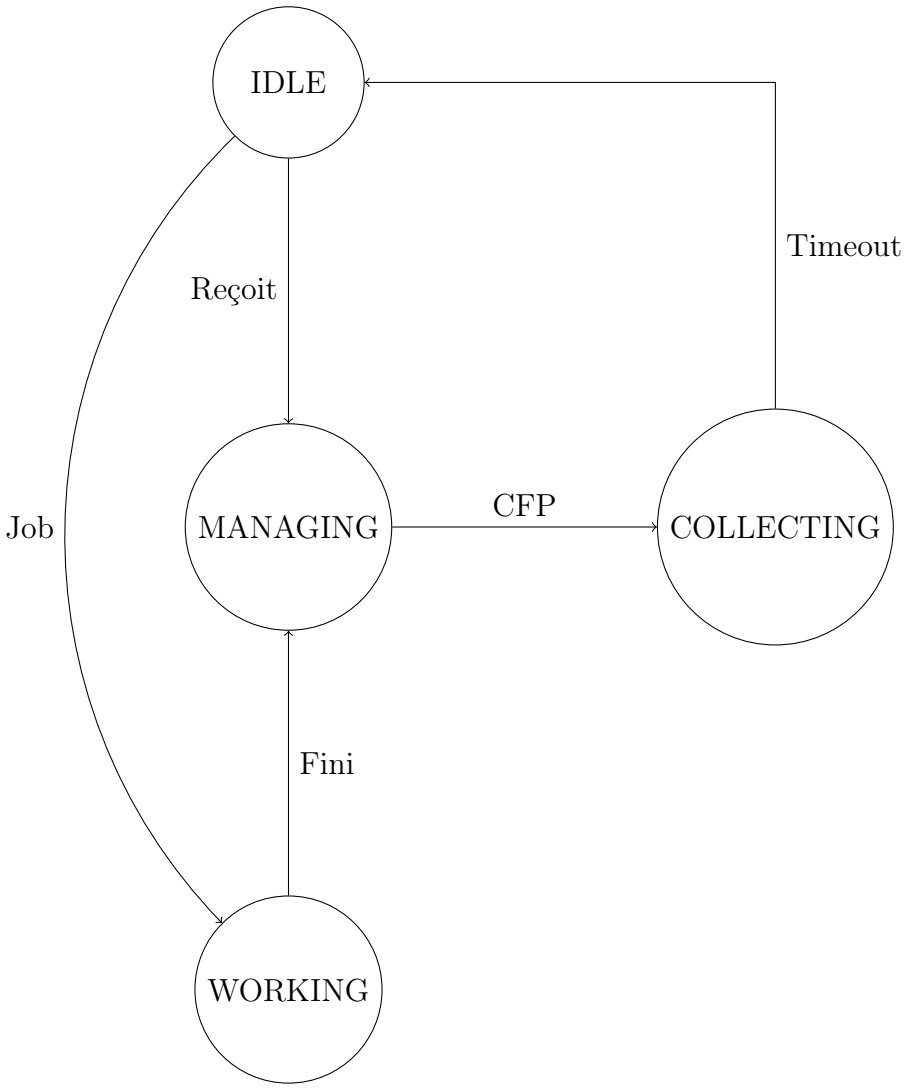

FIGURE 2 – États principaux d'un RobotAgent

7 Guide d'Utilisation

7.1 Compilation

```
javac -cp "jade.jar:." Main.java agents/*.java \
      model/*.java utils/*.java
```

7.2 Exécution

Défaut (3 robots) :

```
java -cp "jade.jar:." Main
```

Personnalisé :

```
java -cp "jade.jar:." Main [nbRobots] [lambda1] [lambda2] [lambda3]
```

Exemples :

```
# 5 robots
java -cp "jade.jar:." Main 5

# 4 robots, parametres temporels
java -cp "jade.jar:." Main 4 2000 5000 1000
```

7.3 Script de Lancement

launch.sh (Linux/Mac) :

```
#!/bin/bash
ROBOTS=${1:-3}
LAMBDA1=${2:-1000}
LAMBDA2=${3:-2000}
LAMBDA3=${4:-500}

java -cp "jade.jar:." Main $ROBOTS $LAMBDA1 $LAMBDA2 $LAMBDA3
```

Utilisation :

```
chmod +x launch.sh
./launch.sh 5 2000 4000 1000
```

7.4 Sorties Console

```
Robot Robot1 pret. Skills: [0, 2, 4]
Robot Robot2 pret. Skills: [1, 3, 4]
Atelier Atelier pret.
Atelier : Produit P-1 envoye a Robot2
>>> [Manager Robot2] Enchere pour Produit P-1 (Skill 1)...
+++ [Manager Robot2] Vainqueur pour P-1 est Robot2
VVV [Worker Robot2] J ai gagne le job P-1. File: 1
... [Worker Robot2] Travail en cours sur P-1 (Skill 1)
*** [Worker Robot2] FINI skill 1 sur P-1
--- PRODUIT FINI : P-1 recu de Robot3 ---
```

8 Écarts et Améliorations

8.1 Points Forts

- Décentralisation complète
- Scalabilité (ajout robots facile)
- Utilisation correcte des behaviours JADE
- Protocole Contract Net adapté
- Gestion files d'attente

8.2 Améliorations Possibles

— Implémenter probabilité d'échec
— Calcul makespan précis
— Timeout adaptatif (CollectProposals)
— Statistiques temps réel
— Interface graphique
— Métriques performance

9 Conclusion

Ce projet a permis de mettre en œuvre un système multi-agents décentralisé pour l'ordonnancement de production. L'utilisation de JADE et du Contract Net Protocol a démontré l'efficacité de la coordination par enchères.

9.1 Apprentissages

— Maîtrise de JADE et ses behaviours
— Protocoles de négociation FIPA-ACL
— Coordination distribuée
— Directory Facilitator
— Communication asynchrone

9.2 Perspectives

— Optimisation multi-critères
— Tolérance aux pannes
— Apprentissage adaptatif
— Benchmarking performance

Références

[1] Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Wiley.

[2] FIPA (2002). *FIPA Contract Net Interaction Protocol Specification*. Foundation for Intelligent Physical Agents.

[3] Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley.