Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of Science and Technology Houari Boumédiène

Faculty of Computer Science

Department of Computer Science

Master's in Intelligent Information Systems

Course : Agents Technology

Multi-agent systems Project

SI SABER Karim Mounir, 202031067146
OULD ROUIS Zakaria, 202032035272

Academic Year : 2023 / 2024

Table des matières

Table des figures

1 General Introduction

Multi-Agent Systems (MAS) are an important area of research in artificial intelligence and distributed computing. A MAS is composed of multiple interacting agents, each capable of autonomous decision-making and acting based on their environment and objectives. These systems are particularly useful for solving complex problems that involve multiple stakeholders with different goals, as they can model and simulate interactions in various scenarios such as auctions, negotiations, and resource allocations.

In this project, we explore two distinct scenarios using MAS. The first part involves a negotiation scenario in the form of an auction, where one seller interacts with multiple buyers. The second part involves a mobile buyer agent who interacts with multiple sellers to make a decision based on multiple criteria. These scenarios illustrate the flexibility and capability of MAS to handle diverse and dynamic environments.

2 Part 1 : Multi-Agent Negotiation (1 Seller - Several Buyers)

2.1 Introduction

The first part of the project focuses on modeling an auction scenario using a Multi-Agent System. This involves a single seller and multiple buyers participating in a dynamic bidding process. The objective is to simulate the interaction and negotiation between these agents to reach a final sale.

2.2 MAS architecture

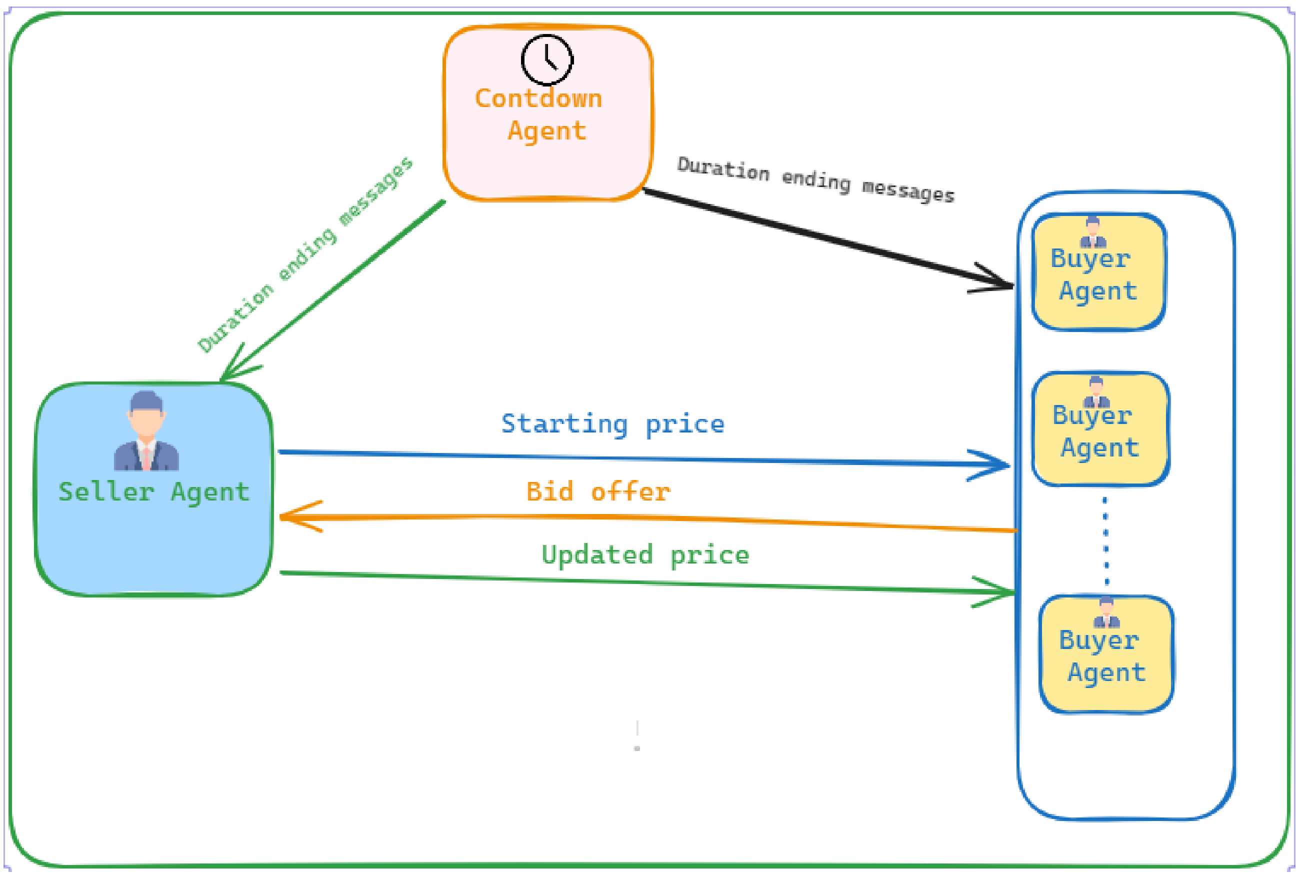

FIGURE 1 – Part 1 MAS architcture

2.3 Auction Scenario

1. **Item Listing** : The seller lists an item for sale.
2. **Opening Price** : The seller sets an initial asking price for the item.
3. **Bidding Process** : Buyers place bids higher than the current asking price.
4. **Price Update** : The seller communicates the highest bid to all buyers.
5. **Iteration** : Steps 3 and 4 are repeated until either all buyers stop bidding or the auction time ends.
6. **Final Decision** : If the highest bid exceeds the seller's reserve price (unknown to buyers), the item is sold to the highest bidder.

2.4 Implementation

2.4.1 Main class

2.4.2 Creating the main container

```java
    public static void main(String[] args) {
        try {
            jade.core.Runtime rt = jade.core.Runtime.instance();
            jade.core.ProfileImpl p = new jade.core.ProfileImpl(false);
            p.setParameter(Profile.GUI, "true");
            p.setParameter(Profile.LOCAL_HOST, "localhost");
            jade.wrapper.AgentContainer container1 = rt.createMainContainer(p);
```

FIGURE 2 – Main container

2.4.3 Creating buyer agent

```java
for (int i = 0; i < nb_buyers; i++) {
                jade.wrapper.AgentController buyer = container1.createNewAgent("Buyer0".concat(String.valueOf(i)), "org.example.Buyer", null);
                buyer.start(); // Start buyers
            }
```

FIGURE 3 – Buyer Agent

2.4.4 Creating seller agent

Argument : StartingPrice

```java
            // Define arguments for Seller
            Object[] startPrice = {"1500"};
            jade.wrapper.AgentController seller = container1.createNewAgent("Seller", "org.example.Seller", startPrice);
            seller.start(); // Start seller
```

FIGURE 4 – Seller Agent

2.4.5 Creating contdown agent

Argument : BidDuration

```
            //duration in seconds
            Object[] bidDuration = {"1"};
            jade.wrapper.AgentController countdownAgent = container1.
createNewAgent("Countdown", "org.example.Duration", bidDuration);
            countdownAgent.start(); // Start CountDown Agent
```

FIGURE 5 – Contdown Agent

2.5 Contdown Class(Duration)

In this section, we extract the auction duration argument passed into the main class and introduce a CountdownBehaviour.

```
            //duration in seconds
            Object[] bidDuration = {"1"};
            jade.wrapper.AgentController countdownAgent = container1.
createNewAgent("Countdown", "org.example.Duration", bidDuration);
            countdownAgent.start(); // Start CountDown Agent
```

FIGURE 6 – Duration

Behaviour : We decrements the duration and we stop wthe cyclic behaviour when duration = 0.

```java
class CountDown extends Behaviour {
 int duration;
 public CountDown(int duration) {
     this.duration = duration;
 }

 @Override
 public void action() {
     // Decrease the duration by 1
     duration--;
```

FIGURE 7 – Contdown Behaviour

```java
public boolean done() {

    // send ending alert before exiting
    if(duration == 0){
        ACLMessage msg = new ACLMessage(ACLMessage.DISCONFIRM);
        msg.setContent("done");
        for (int i = 0; i < nb_buyers; i++) {
            msg.addReceiver(new AID("Buyer0".concat(String.valueOf(i)),AID.ISLOCALNAME));
        }
        msg.addReceiver(new AID("Seller",AID.ISLOCALNAME));
        myAgent.send(msg);

    }
```

FIGURE 8 – Behaviour ending

2.6 Seller Class

2.6.1 Sending the starting price

```java
    public boolean done() {

        // send ending alert before exiting
        if(duration == 0){
            ACLMessage msg = new ACLMessage(ACLMessage.DISCONFIRM);
            msg.setContent("done");
            for (int i = 0; i < nb_buyers; i++) {
                msg.addReceiver(new AID("Buyer0".concat(String.valueOf(i)),AID.ISLOCALNAME));
            }
            msg.addReceiver(new AID("Seller",AID.ISLOCALNAME));
            myAgent.send(msg);

        }
```

FIGURE 9 – Behaviour ending

Cyclic behaviour

— Here, the seller agent will continuously receive new bid offers from buyers by checking for performative type "PROPOSE" and ontology "new offer". If an offer exceeds the current ‘updated price‘ value, it will become the new highest bid. The seller will then inform all buyers of the updated highest price using the "updated price" performative.

```java
        // Add cyclic behavior to handle incoming messages
        this.addBehaviour(new CyclicBehaviour() {
            @Override
            public void action() {
                // Define message template for receiving price offers
                MessageTemplate priceOfferTemplate = MessageTemplate.
MatchPerformative(ACLMessage.PROPOSE);
                ACLMessage priceOfferMessage = receive(priceOfferTemplate);
// Receive the price offer


// Check if a new price offer has been received and if it is higher than the curr
ent highest price
                if (priceOfferMessage != null && Integer.parseInt(
priceOfferMessage.getContent()) > currentHighestPrice) {


// Update the highest price and the buyer with the highest offer
                    System.out.println("Highest price " + priceOfferMessage.
getContent() + " from " + priceOfferMessage.getSender().getName());
                    currentHighestPrice = Integer.parseInt(priceOfferMessage.
getContent());
                    highestOfferBuyer = String.valueOf(priceOfferMessage.
getSender().getName());

                    // Send the updated highest price to all buyers
                    ACLMessage updatedPriceMessage = new ACLMessage(ACLMessage.
INFORM);
                    updatedPriceMessage.setContent(String.valueOf(
currentHighestPrice));
                    for (int i = 0; i < nb_buyers; i++) {
                        updatedPriceMessage.addReceiver(new AID("Buyer0".concat(
String.valueOf(i)), AID.ISLOCALNAME));
                    }
                    myAgent.send(updatedPriceMessage);
                }
```

FIGURE 10 – Seller behaviour

Behaviour ending

```java
                  // Define message template for receiving the end of bid countdown
                  MessageTemplate countdownTemplate = MessageTemplate.MatchPerformative(ACLMessage.DISCONFIRM);
                  ACLMessage countdownMessage = receive(countdownTemplate);


// Check if the countdown message indicates the end of the auction
                  if (countdownMessage != null && countdownMessage.getContent().equals("done")) {
                      System.out.printf("Highest price is " + currentHighestPrice + "$. Product has been sold to " + highestOfferBuyer);
                      System.out.println(myAgent.getLocalName() + " is deleted");
                      myAgent.doDelete(); // Terminate the agent
                  }
```

FIGURE 11 – Seller ending behaviour

2.7 Buyer Class

To send a new offer, each buyer agent will begin by receiving the updated auction price from the seller agent and then update the ‘highestPrice‘ variable accordingly.

```
// Initialize a random generator and an array of bid increment values
    Random randomGenerator = new Random();
    int[] bidIncrements = {50, 100, 200, 300, 500, 1000};
```

Here, we use an array of bid increment values, each buyer when he will send a new offer will use this fonction :

Offer = highestPrice(current) + a random value taken from array .

```java
// Send a new proposition to the seller with an incremented bid
                    int newBid = highestPrice + bidIncrements[randomGenerator.
nextInt(bidIncrements.length)];
                    ACLMessage bidMessage = new ACLMessage(ACLMessage.PROPOSE);
                    bidMessage.setContent(String.valueOf(newBid));
                    bidMessage.addReceiver(new AID("Seller", AID.ISLOCALNAME));
                    myAgent.send(bidMessage);
                }
```

FIGURE 12 – Buyer Bid

This code snippet defines a message template to receive notifications about the end of the auction. When a message with the content "done" is received, it signals the agent to terminate itself and print a message indicating its deletion.

```java

// Define a message template for receiving the end of bid countdown
                MessageTemplate countdownTemplate = MessageTemplate.
MatchPerformative(ACLMessage.DISCONFIRM);
                ACLMessage countdownMessage = receive(countdownTemplate);

                // Check if the countdown message indicates the end of the auction
                if (countdownMessage != null && countdownMessage.getContent().
equals("done")) {
                    System.out.println(myAgent.getLocalName() + " is deleted");
                    myAgent.doDelete(); // Terminate the agent
```

FIGURE 13 – Buyer deletion

2.8 Execution

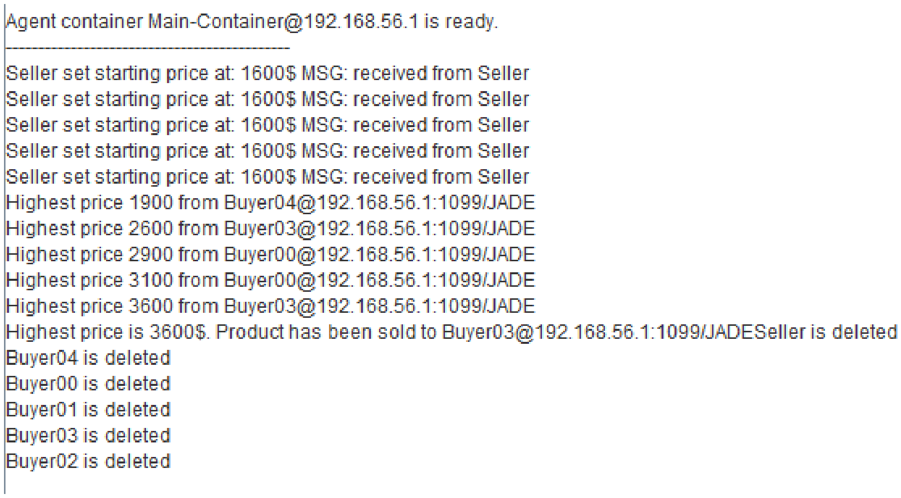

FIGURE 14 – Execution

2.9 Graphic interface

The graphical interface includes input fields for the **number** of buyers and the **starting price**.

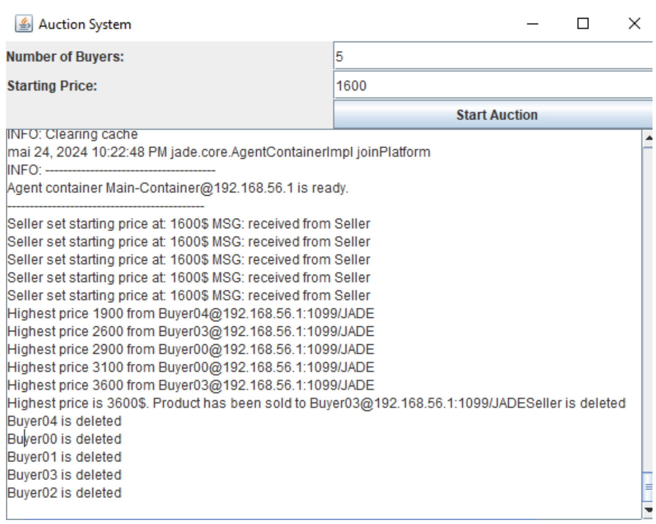

FIGURE 15 – Graphic interface

3 Multi-Criteria Decision and Mobile Agents

3.1 Introduction

The second part of the project involves implementing a scenario where a mobile buyer agent interacts with multiple seller agents across different platforms. The buyer agent evaluates offers based on multiple criteria and migrates between containers or platforms to find the best deal. This scenario demonstrates the use of mobile agents in dynamic environments where decisions are based on multiple factors.

3.2 MAS Architecture

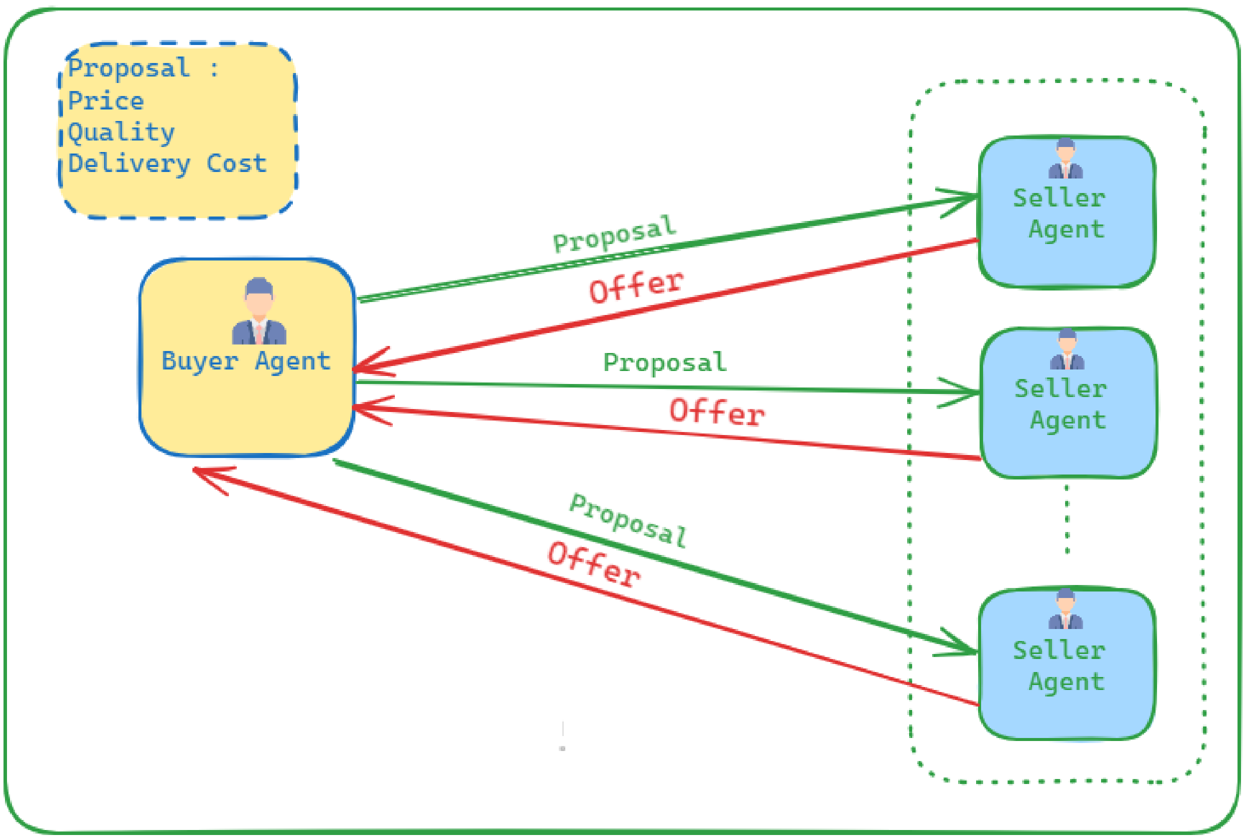

FIGURE 16 – Part 2 MAS Architecture

3.2.1 Roles

1. **Buyer Agent**
   Sends purchase proposals to all seller agents and receives offers from them.
2. **Sender Agent**
   Receives purchase proposals from buyer agents and sends selling offers in response to the buyer agent.

3.3 Class explanation (Inter Container)

3.3.1 Main class

1. **Create the first container and assign it to a profile(for sellers**

```
            try {
                // Create the first container
                jade.core.Runtime rt = jade.core.Runtime.instance();
                jade.core.ProfileImpl p1 = new jade.core.ProfileImpl(false);
                p1.setParameter(Profile.GUI, "true");
                p1.setParameter(Profile.LOCAL_HOST, "localhost");
                p1.setParameter(Profile.CONTAINER_NAME, "container1");
                if (!INTER_CONTAINER) {
                    p1.setParameter(Profile.PLATFORM_ID, "platform1");
                }
```

FIGURE 17 – First container

2. **Instanciate the sellers and start them**

```
for (int i = 0; i < nb_sellers; i++) {
                jade.wrapper.AgentController seller = container1.
createNewAgent("Seller" + i, "org.example.Seller", null);
                seller.start(); // Start sellers
            }
            // Start the first container
            container1.start();
```

FIGURE 18 – Instanciate the sellers

3. **Create the second container for the buyer**

```
// Create the second container
                jade.core.Runtime rt2 = jade.core.Runtime.instance();
                jade.core.ProfileImpl p2 = new jade.core.ProfileImpl(false
);
                p2.setParameter(Profile.GUI, "true");
                p2.setParameter(Profile.LOCAL_HOST, "localhost");
                p2.setParameter(Profile.CONTAINER_NAME, "container2");
                if (!INTER_CONTAINER) {
                    p2.setParameter(Profile.PLATFORM_ID, "platform2");
                }
                jade.wrapper.AgentContainer container2 = rt2.
createAgentContainer(p2);
```

FIGURE 19 – Second container

4. **Instanciate the buyer**
   Argument : Proposal

```java
// Define arguments for Buyer
                Object[] proposal = {"4", "7", "4"};

                // Create a buyer in the second container
                jade.wrapper.AgentController buyer = container2.
createNewAgent("Buyer", "org.example.Buyer", proposal);
                buyer.start(); // Start buyer
```

FIGURE 20 – Buyer instanciation

3.3.2 Proposall Class

This class implements the Serializable interface, which is mandatory for using the getContentObject() messaging method in JADE, as it deserializes the content into a Proposal class. Here we define two methods to convert the type to a List and a String.

```java
public List<Integer> getAsArray(){
   List<Integer> result = new ArrayList<>();
   result.add(price);
   result.add(quality);
   result.add(deliveryCost);
   return result;
}

@Override
public String toString() {
   return "Proposal{" +
           "price=" + price +
           ", quality=" + quality +
           ", deliveryCost=" + deliveryCost +
           '}';
}
```

FIGURE 21 – Proposall class

3.3.3 Seller Agent

1. **Receive buyer proposal**

```java
addBehaviour(new CyclicBehaviour() {

            @Override
            public void action() {

// Define message template to match incoming INFORM messages
                MessageTemplate mProp = MessageTemplate.MatchPerformative(
ACLMessage.INFORM);

                // Receive the proposal message
                ACLMessage proposal = receive(mProp);


// Check if the proposal is not null and has the correct ontology
                if (proposal != null && proposal.getPerformative() ==
ACLMessage.INFORM
                        && proposal.getOntology().equals("proposal sending"
)) {
                    Proposal props;
                    try {
                        // Extract the content of the proposal message
                        props = (Proposal) proposal.getContentObject();
```

FIGURE 22 – Receive proposal from Buyer

seller will first receive the buyer proposal of INFORM performative and proposal sending

2. **Create the selling offer**

```java
// Create a new offer by modifying the received proposal
                    Proposal sellerOffer = new Proposal();
                    sellerOffer.price = props.price + random.nextInt(50);
                    sellerOffer.quality = props.quality + random.nextInt(50);
                    sellerOffer.deliveryCost = props.deliveryCost + random.nextInt(50);

```

FIGURE 23 – Create selling offer

3. **.Send the offer to buyer**

```java
ACLMessage msg = new ACLMessage(ACLMessage.PROPOSE);
                    msg.setOntology("offre seller");
                    try {
                        msg.setContentObject(sellerOffer);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                    msg.addReceiver(new AID("Buyer", AID.ISLOCALNAME));
                    myAgent.send(msg);
```

FIGURE 24 – Sending the offer to buyer

3.3.4 Buyer Class

1. **Buyer migration to Seller container**

```java
        String containerName = "container1";
        ContainerID destination = new ContainerID();
        destination.setName(containerName);
        doMove(destination);
```

FIGURE 25 – Buyer migration to Seller container

2. **Retreive the proposal and send a new one**

```java
proposition = getArguments(); // Get the proposition arguments
        p.price = Integer.parseInt(proposition[0].toString());
// Set the price from arguments
        p.quality = Integer.parseInt(proposition[1].toString());
// Set the quality from arguments
        p.deliveryCost = Integer.parseInt(proposition[2].toString
()); // Set the delivery cost from arguments

        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
// Create a new ACLMessage
        msg.setOntology("proposition sending");
// Set the message ontology
        msg.setLanguage("JavaSerialization");
// Set the message language
        try {
            msg.setContentObject(p);
// Set the content of the message to the proposition object
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        // Add receivers for the message
        for (int i = 0; i < nb_sellers; i++) {
            msg.addReceiver(new AID("Seller".concat(String.valueOf
(i)),AID.ISLOCALNAME));
        }
        send(msg);
```

FIGURE 26 – Retreive the proposal

3. **Receiving offer from sellers**

```java
MessageTemplate MsgT = MessageTemplate.MatchPerformative(ACLMessage.PROPOSE);
ACLMessage offer = receive(MsgT); // Receive the offer message


// Check if the offer message is not null and has the correct ontology
if(offer != null && offer.getPerformative() == ACLMessage.PROPOSE &&
        offer.getOntology().equals("offre seller")){

    // Extract the content of the offer message
    Proposal sellerOffer = (Proposal) offer.getContentObject();

    // Log the received offer and the sender
    System.out.println("received buying "+ sellerOffer.toString() + " from " +  offer.getSender().getLocalName());
    sellerCpt++; // Increment the seller counter
```

FIGURE 27 – Receiving offer from sellers

3.3.5 Proposal Evaluation by Buyer

Once the Buyer receives the Proposal object, it will evaluate the proposal using the following rule :

$$\text{Score} = \sum (\text{BuyerProposal(attribute)} \times \text{SellerProposal(attribute)})$$

— **If it is a minimization criterion** :

$$\text{Score}+ = \text{NormalizedBuyerProposal(attribute)} \times \text{NormalizedSellerProposal(attribute)}$$

— **If it is a maximization criterion** :

$$\text{Score}+ = \text{NormalizedBuyerProposal(attribute)} \times (-1) \times \text{NormalizedSellerProposal(attribute)}$$

Additionally, each value must be normalized :

$$\text{NormalizedBuyerProposal(attribute)} = \frac{\text{BuyerProposal(attribute)}}{\max(\text{BuyerProposal(of all attributes)})}$$

$$\text{NormalizedSellerProposal(attribute)} = \frac{\text{SellerProposal(attribute)}}{\max(\text{SellerProposal(of all attributes)})}$$

```java
double value = 0; // Initialize the value variable


// Calculate the value based on preferences and normalize the attributes
                        int i = 0;
                        while (i < p.getAsArray().size()) {
                            if (prefs.get(i) == 1) {
                                // Maximize
                                value -= normalise(p.getAsArray().get(i), p
) * normalise(sellerOffer.getAsArray().get(i), sellerOffer);
                            } else {
                                // Minimize
                                value += normalise(p.getAsArray().get(i), p
) * normalise(sellerOffer.getAsArray().get(i), sellerOffer);
                            }
                            i++;
                        }

                        System.out.println(value);
// Print the calculated value
                        offers.put(offer.getSender().getName(), value);
// Add the offer to the map
                    }
```

FIGURE 28 – Proposal evaluation

4. **Behaviour ending**
   Once SellerCounter equals to Number of Sellers, the buyer agent will select the offer with the minimum score from the offers map. It will then display this offer along with the corresponding seller.

```java
public boolean done() {
                    // Check if offers have been received from all sellers
                    if (offers.size() == nb_sellers) {
                        // Get the minimum value from the map
                        OptionalDouble minValueOptional = offers.values().
stream().mapToDouble(Double::doubleValue).min();

                        // Check if the minimum value exists
                        if (minValueOptional.isPresent()) {
                            double minValue = minValueOptional.getAsDouble();

                            // Find the key corresponding to the minimum value
                            Optional<String> minKeyOptional = offers.entrySet
().stream()
                                    .filter(entry -> entry.getValue().
doubleValue() == minValue)
                                    .map(Map.Entry::getKey)
                                    .findFirst();


// Check if the key corresponding to the minimum value exists
                            if (minKeyOptional.isPresent()) {
                                String minKey = minKeyOptional.get();

// Print the winner with the minimum offer value
                                System.out.println("The seller " + minKey +
" won with the best value : " + minValue);
                            } else {
                                System.out.println(
"No seller found with the minimum offer value");
                            }
                        } else {
                            System.out.println("0 : No offers available");
                        }

                        return true; // Indicate that the behavior is done
                    }
                    return false;
// Indicate that the behavior is not done yet

                }
```

FIGURE 29 – Behaviour Ending

3.4 Execution

```
INFO: ----------------------------------------
Agent container container2@192.168.56.1 is ready.
--------------------------------------------
received buying Proposal{price=23, quality=55, deliveryCost=48} from Seller8
-0.2623376623376624
received buying Proposal{price=32, quality=7, deliveryCost=18} from Seller0
0.6741071428571428
received buying Proposal{price=45, quality=24, deliveryCost=19} from Seller5
0.2793650793650793
received buying Proposal{price=12, quality=38, deliveryCost=8} from Seller4
-0.6992481203007519
received buying Proposal{price=43, quality=16, deliveryCost=8} from Seller1
0.3056478405315614
received buying Proposal{price=49, quality=53, deliveryCost=53} from Seller3
0.09973045822102422
received buying Proposal{price=10, quality=56, deliveryCost=35} from Seller2
-0.5408163265306123
received buying Proposal{price=9, quality=37, deliveryCost=42} from Seller9
-0.18707482993197277
received buying Proposal{price=50, quality=55, deliveryCost=15} from Seller7
-0.3246753246753248
received buying Proposal{price=42, quality=28, deliveryCost=13} from Seller6
0.0816326530612245
The seller Seller4@192.168.56.1:1099/JADE won with the best value : -0.6992481203007519
PS C:\Users\ksisa\OneDrive\Bureau\TechAgentPt2-master>
```

FIGURE 30 – Part 2 execution

3.5 Graphic Interface

The graphical interface includes input field for the **number** of sellers.

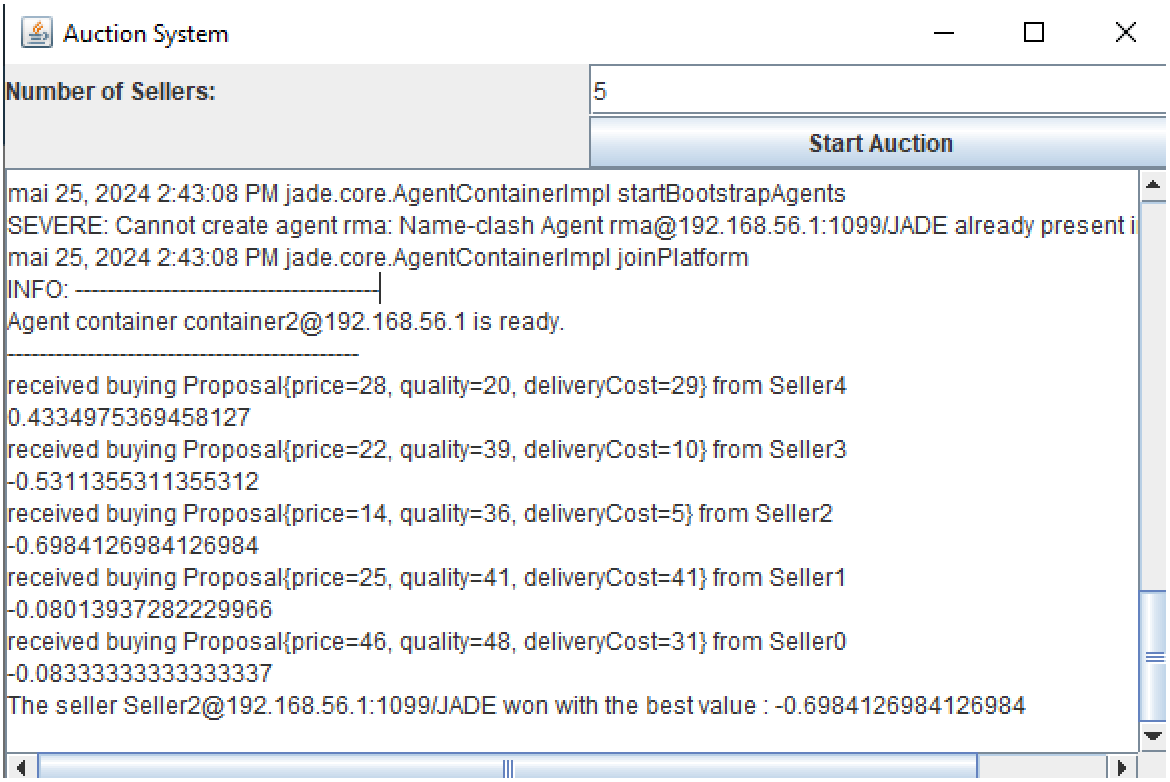

FIGURE 31 – Part 2 Graphic Interface

3.6 Inter Platform Migration

```java
                AID remoteAMS = new AID("ams@remotePlatform:1099/JADE"
  , AID.ISGUID);
                  remoteAMS.addAddresses("http://SRV_URB:7778/acc");
                  PlatformID destination = new PlatformID(remoteAMS);
                  doMove(destination);
          }

```

FIGURE 32 – Inter Platform Migration

3.7 Inter Container Migration Illustation

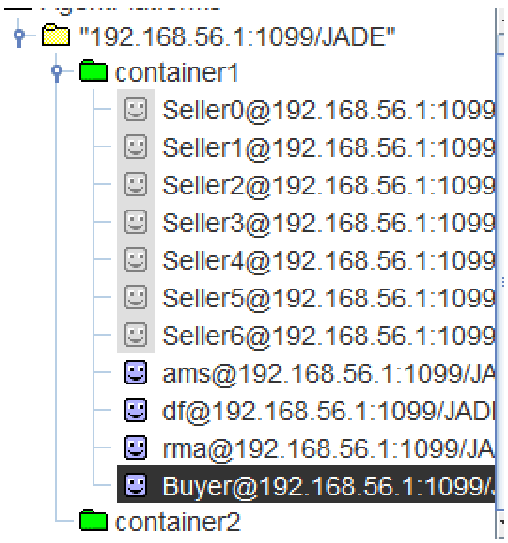

FIGURE 33 – Before Migration

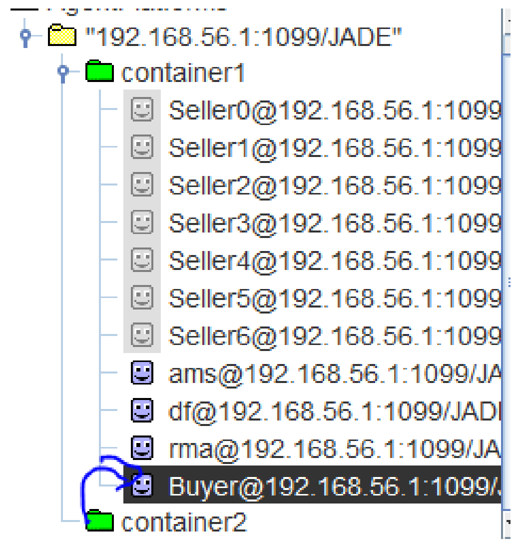

FIGURE 34 – After Migration

FIGURE 35 – Inter-Container Migration

4 Overall Conclusion

This project focused on the implementation and analysis of multi-agent systems (MAS) through two primary scenarios : multi-agent negotiation and multi-criteria decision making. The project showcased the potential and challenges of using MAS in auction environments, highlighting how agents can interact, negotiate, and make decisions based on predefined criteria.

4.1 Multi-Agent Negotiation

In the first part, we implemented an auction system with one seller and multiple buyers. This scenario demonstrated the dynamic interactions between agents in a competitive environment. The seller agent initiated the auction with an opening price, and the buyer agents engaged in iterative bidding, aiming to outbid each other. The seller continuously updated all buyers with the highest bid, and the process repeated until the auction concluded. The successful implementation of this scenario highlighted the efficiency of MAS in handling real-time negotiations and decision-making processes.

4.2 Multi-Criteria Decision Making and Mobile Agents

The second part extended the project to a scenario where a buyer agent, acting as a mobile agent, evaluated offers from multiple sellers based on multiple criteria. The buyer agent migrated across different containers and platforms, collecting offers and calculating scores based on normalization and specific evaluation rules. This part of the project emphasized the versatility and mobility of agents in distributed environments, as well as the complexity of decision-making when multiple factors are considered.

4.3 Interface and User Interaction

An intuitive graphical user interface (GUI) was developed to facilitate user interaction with the auction system. Users could input the number of sellers and observe the auction process and results directly within the GUI, enhancing the usability and accessibility of the system.

4.4 Overall Impact

The project successfully demonstrated the practical applications of MAS in auctions and decision-making processes. The agents' ability to negotiate, adapt, and make informed decisions in real-time showcased the robustness of MAS in dynamic environments. This project lays the foundation for further exploration and optimization of MAS in various domains, such as e-commerce, supply chain management, and automated trading systems. The insights gained from this project underscore the potential of MAS to revolutionize how autonomous systems can collaborate and compete in complex scenarios.