IAD Mini Project Report

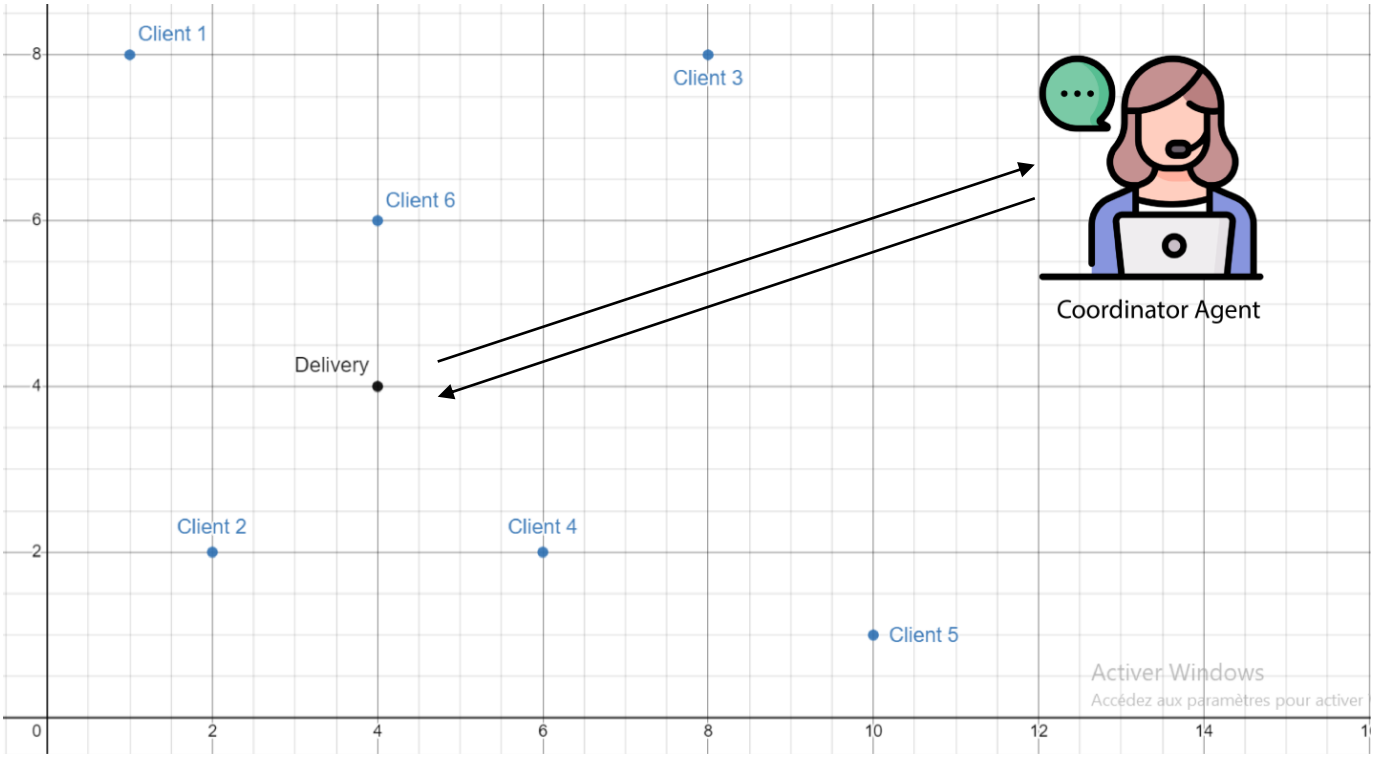
Introduction:

This work is carried out as part of an IAD Tp in the form of a Mini Project by Ben Jeddi Selim.

Description and objective of the application created:

The application produced is a simulation of a multi-agent system in which Customer agents and Delivery agents and Coordinator agents interact to deliver packages to different customers in an optimal order and in a reduced time.

This figure shows the dispersion of a set of clients in a benchmark as an example for our problem:



The objective of the application is to put the following course concepts into practice:

- Distributed resolution of a search and optimization problem: the Client agents with the delivery agent cooperate to find the point closest to the vehicle by involving the coordinator agent.
- Forms of interaction: communication, cooperation between agents in solving a problem, coordination of actions. : agents communicate with each other to exchange information and coordinate their actions.

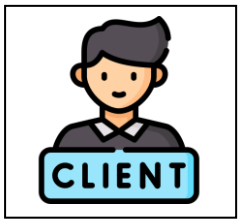
The agents (or classes) developed:

Agent	Kind	Structure	Behavior	Description
Customer Agent	Cognitive agent	• x and y coordinates	• Respond to contact requests	The purpose of the agent is to respond to requests for contact details from other agents in the system and to participate in a proposal and acceptance process with a "DeliveryAgent" agent.
Agent Delivery	Cognitive agent	• x and y coordinates	• Find the nearest customer • Travel to the nearest customer • Repeats procedure after each delivery (stops when customer count ends)	This agent plans its route by choosing the closest customer among the "ClientAgent" type agents and repeats this process each time it visits a new customer.
Coordinating Agent	Cognitive agent	• Activation by message	• Service registration: The CoordinatorAgent agent registers with the service type "Coordinator" to allow other agents to find it. • Receiving behavior: The agent has a cyclical behavior that listens to INFORM messages to receive information	The CoordinatorAgent acts as a central coordinator in a multi-agent system. Its main responsibility is to coordinate and supervise delivery agents (probably autonomous vehicles) by receiving updates on the points visited by these agents.

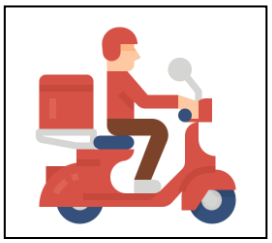
			about the points visited from the delivery agents and process end-of-mission messages.	

Project implementation:

The project is divided into 3 classes written in Java (Jade) as follows:



Customer



Delivery Agent

Coordinator Agent

```java
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;


public class ClientAgent extends Agent {
    int x;
    int y;
    protected void setup() {
        // Initialisation de l'agent Point
        Object[] args = getArguments();
        if (args != null && args.length == 2) {
            x = Integer.parseInt(args[0].toString());
            y = Integer.parseInt(args[1].toString());

        } else {
            System.err.println("L'agent Point nÃ©cessite des coordonnÃ©es x et y en tant
qu'arguments.");
            doDelete();
        }
        registerService("Point*");
        System.out.println(getLocalName() + ": Initial position : x: " + x + ", y: " + y);
        addBehaviour(new CoordinateRequestBehaviour());
    }
    private void registerService(String serviceType) {
        // Register the service with the Directory Facilitator (DF)
        try {
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName(getAID());
            ServiceDescription sd = new ServiceDescription();
            sd.setType(serviceType);
            sd.setName("PointService");
            dfd.addServices(sd);

            DFService.register(this, dfd);
        } catch (FIPAException fe) {
            fe.printStackTrace();
        }
    }
    private class CoordinateRequestBehaviour extends CyclicBehaviour {
        public void action() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            // Listen for incoming messages
            ACLMessage msg = receive();

            if (msg != null) {
                // Handle incoming request for coordinates
                if (msg.getPerformative() == ACLMessage.REQUEST && msg.getContent().equals("Position?"))
{
                    // Respond with the coordinates
                    System.out.println(getLocalName() + ": Received request for coordinates from " +
msg.getSender().getLocalName());
                    ACLMessage reply = msg.createReply();
                    reply.setPerformative(ACLMessage.INFORM);
                    reply.setContent(x + "," + y);
                    send(reply);
                    System.out.println(getLocalName() + ": Sent coordinates to " +
msg.getSender().getLocalName());
                    MessageTemplate proposeTemplate =
MessageTemplate.MatchPerformative(ACLMessage.PROPOSE);
                    ACLMessage propose = myAgent.blockingReceive(proposeTemplate);

                    if (propose != null) {
                        // Traiter la proposition du VehicleAgent
                        System.out.println(getLocalName() + ": Received proposal from " +
propose.getSender().getLocalName());

                        // Ã©valuer la proposition (simple ici, mais vous pouvez ajouter une logique
plus complexe)



                        // Accepter la proposition
                        ACLMessage accept = propose.createReply();
                        accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
                        send(accept);
                        System.out.println(getLocalName() + ": Accepted proposal from " +
propose.getSender().getLocalName());

                    } else {
                        System.out.println(getLocalName() + ": No proposal received.");
                    }
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                } else {
                    // Handle other types of messages if needed
                    //System.out.println(getLocalName() + ": Received unexpected message: " +
msg.getContent());
                }
            } else {
                // Block the behaviour until a message is received
                block();
            }
        }
    }
}
```

This code is an implementation of a JADE (Java Agent DEvelopment Framework) agent. Let's take a look at each part of the code to understand how it works:

1- ClientAgent class:

This class extends the JADE Agent class, making it a JADE agent.

The x and y coordinates represent the initial position of the agent and are initialized from the arguments passed during agent creation.

The setup method is called during agent initialization. It configures the agent by registering a service of type "Point*" with the Directory Facilitator (DF) and adding a behavior (CoordinateRequestBehaviour) to the agent.

2- registerService(String serviceType) method:

This method registers the agent with the DF by specifying the service type ("Point*"). This allows other agents in the system to discover this agent when searching for agents offering "Point*" type services.

3- CoordinateRequestBehaviour class:

This is a cyclical behavior that continually listens for messages.

The action method is executed every cycle of the behavior.

The behavior starts with a one second pause (Thread.sleep(1000)) to avoid execution too quickly.

Then the agent listens for incoming messages with the receive() method. If it receives a message, it processes it.

If the message is of type REQUEST with the content "Position?", the agent responds with its coordinates (x and y).

Then, the agent blocks to wait for a proposal (PROPOSE) from the VehicleAgent side. When it receives a proposal, it accepts it by sending an ACCEPT_PROPOSAL message.

The blockingReceive method is used to block behavior until a proposal is received or a deadline has passed.

```java
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class DeliveryAgent extends Agent {
    int x;
    int y;
    private List<AID> visitedPoints = new ArrayList<>();
    private Set<AID> allPoints = new HashSet<>();
    AID nearestPoint;
    protected void setup() {
        // Initialization of the VehicleAgent
        Object[] args = getArguments();
        if (args != null && args.length == 2) {
            x = Integer.parseInt(args[0].toString());
            y = Integer.parseInt(args[1].toString());
        } else {
            System.err.println("VehicleAgent requires x and y coordinates as arguments.");
            doDelete();
        }
        System.out.println(getLocalName() + ": Initial position: x: " + x + ", y: " + y);
        // Route planning behavior
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        addBehaviour(new VisitAllPointsBehaviour());
    }

    private class VisitAllPointsBehaviour extends CyclicBehaviour {
        private int step = 0;

        public void action() {
            switch (step) {
                case 0:
                    // Get all Point agents
                    AID[] pointAgents = searchAgents("Point*");
                    for (AID pointAgent : pointAgents) {
                        allPoints.add(pointAgent);
                    }
                    step++;
                    break;
                case 1:
                    // Find the nearest unvisited point
                    AID nearestPoint = findNearestUnvisitedPoint();
                    if (nearestPoint != null) {
                        visitedPoints.add(nearestPoint);
                        System.out.println(getLocalName() + ": I will deliver now: " +
nearestPoint.getLocalName());

                        // Log the names of visited points
                        List<String> visitedPointNames = new ArrayList<>();
                        for (AID visitedPoint : visitedPoints) {
                            visitedPointNames.add(visitedPoint.getLocalName());
                        }
                        //System.out.println(getLocalName() + ": Visited points so far: " +
visitedPointNames.toString());
                        informCoordinator(visitedPointNames);
                        // Propose to visit the point
                        ACLMessage propose = new ACLMessage(ACLMessage.PROPOSE);
                        propose.addReceiver(nearestPoint);

                        String proposeContent = "Propose:" + calculateDistance(nearestPoint) + ":" +
visitedPoints.toString();

                        propose.setContent(proposeContent);
                        send(propose);

                        // Wait for the response from the chosen point
                        MessageTemplate proposeTemplate =
MessageTemplate.MatchPerformative(ACLMessage.ACCEPT_PROPOSAL);
                        MessageTemplate refuseTemplate =
MessageTemplate.MatchPerformative(ACLMessage.REJECT_PROPOSAL);
                        ACLMessage response = receive(MessageTemplate.or(proposeTemplate,
refuseTemplate));

                        if (response != null && response.getPerformative() ==
ACLMessage.ACCEPT_PROPOSAL) {
                            System.out.println(getLocalName() + ": Proposal accepted by " +
response.getSender().getLocalName());
                            indicatePath(nearestPoint);

                            // Update the position after visiting the point
                            updatePosition(nearestPoint);
                        } else {
                            //System.out.println(getLocalName() + ": Proposal rejected or no response
received.");
                        }
                    } else {
                        // No unvisited points left, inform all Point agents
                        informAllPoints();
                        step++;
                    }
                    break;
            }
        }
```

```java
 private void indicatePath(AID nearestPoint) {
            String[] coordinates = getPointCoordinates(nearestPoint);

            if (coordinates.length == 2) {
                int destinationX = Integer.parseInt(coordinates[0]);
                int destinationY = Integer.parseInt(coordinates[1]);

                System.out.println(getLocalName() + ": Indicating how much time to arrive to " +
nearestPoint.getLocalName());

                // Coordinates of the vehicle
                int currentX = x;
                int currentY = y;

                // Move towards the destination
                while (currentX != destinationX || currentY != destinationY) {
                    // Update the vehicle's coordinates at each step
                    if (currentX < destinationX) {
                        currentX++;
                    } else if (currentX > destinationX) {
                        currentX--;
                    }

                    if (currentY < destinationY) {
                        currentY++;
                    } else if (currentY > destinationY) {
                        currentY--;
                    }

                    // Print the current position
                    System.out.println(getLocalName() + ": I will arrive to "+
nearestPoint.getLocalName() +" in "+ currentX + " minutes");
                }

                // Arrived at the destination
                System.out.println(getLocalName() + ": Arrived at " + nearestPoint.getLocalName());

                // Update the vehicle's final coordinates
                x = currentX;
                y = currentY;
            } else {
                System.out.println(getLocalName() + ": Invalid coordinates format received from " +
nearestPoint.getLocalName());
            }
        }

        private AID findNearestUnvisitedPoint() {
            AID nearestPoint = null;
            double minDistance = Double.MAX_VALUE;

            for (AID point : allPoints) {
                double distance = calculateDistance(point);
                if (!visitedPoints.contains(point) && distance < minDistance) {
                    nearestPoint = point;
                    minDistance = distance;
                }
            }

            return nearestPoint;
        }

        private double calculateDistance(AID point) {
            // Get the coordinates of the Point agent
            String[] coordinates = getPointCoordinates(point);
            int pointX = Integer.parseInt(coordinates[0]);
            int pointY = Integer.parseInt(coordinates[1]);

            // Calculate the distance
            return Math.sqrt(Math.pow(pointX - x, 2) + Math.pow(pointY - y, 2));
        }
```

```java
 private String[] getPointCoordinates(AID point) {
            // Send a request to the Point agent to get its coordinates
            ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
            request.setContent("Position?");
            request.addReceiver(point);
            send(request);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            // Wait for the response from the Point agent
            ACLMessage response = receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM));

            if (response != null) {
                return response.getContent().split(",");
            }

            return new String[]{"0", "0"}; // Default coordinates if no response
        }

        private void informAllPoints() {
            // Send an INFORM message to all Point agents to inform them that all points have been
visited
            ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
            inform.setContent("AllPointsVisited");
            AID[] allPointAgents = searchAgents("Point*");

            if (allPointAgents != null) {
                for (AID pointAgent : allPointAgents) {
                    inform.addReceiver(pointAgent);
                }
                send(inform);
                ACLMessage inform1 = new ACLMessage(ACLMessage.INFORM);
                inform1.setContent("finish");
                AID coordinatorAgent = getCoordinatorAgent();
                if (coordinatorAgent != null) {
                    inform1.addReceiver(coordinatorAgent);
                    send(inform1);
                }
                System.out.println(getLocalName() + ": Coordinator! All points have been visited.");
            }
        }
        private void updatePosition(AID point) {
            // Update the position of the vehicle agent after visiting the point
            String[] coordinates = getPointCoordinates(point);
            x = Integer.parseInt(coordinates[0]);
            y = Integer.parseInt(coordinates[1]);
        }
    }
    private AID getCoordinatorAgent() {
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Coordinator");
        template.addServices(sd);

        try {
            DFAgentDescription[] result = DFService.search(this, template);
            if (result.length > 0) {
                return result[0].getName();
            }
        } catch (FIPAException e) {
            e.printStackTrace();
        }

        return null;
    }
    private void informCoordinator(List<String> visitedPointNames) {
        // Send an INFORM message to the Coordinator agent
        ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
        inform.setContent("VisitedPoints:" + visitedPointNames.toString());
        AID coordinatorAgent = getCoordinatorAgent();
        if (coordinatorAgent != null) {
            inform.addReceiver(coordinatorAgent);
            send(inform);
        }
    }
    private AID[] searchAgents(String agentNamePattern) {
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType(agentNamePattern);
        template.addServices(sd);

        try {
            DFAgentDescription[] result = DFService.search(this, template);
            AID[] agents = new AID[result.length];
            for (int i = 0; i < result.length; ++i) {
                agents[i] = result[i].getName();
            }
            return agents;
        } catch (FIPAException fe) {
            fe.printStackTrace();
            return null;
        }
    }
}
```

1- DeliveryAgent class:

This class extends the JADE Agent class, making it a JADE agent.

The x and y coordinates represent the initial position of the delivery agent and are initialized from the arguments passed when creating the agent.

The visitedPoints list stores points visited by the delivery agent.

The setup method is called during agent initialization. It configures the agent by initializing its location, adding a behavior (VisitAllPointsBehaviour) for route planning, and waiting for two seconds.

2- VisitAllPointsBehaviour inner class:

This is a cyclical behavior that drives route and delivery planning logic.

The step variable is used to track the current step of the behavior.

In step 0, the agent searches for all agents of type "Point*" and adds them to the allPoints set.

In step 1, the agent searches for the nearest unvisited point using the findNearestUnvisitedPoint method. If such a point is found, the agent registers this point as visited, informs the coordinator of the visited points, offers to visit the point by means of a PROPOSE message, and waits for a response from the chosen point.

If the proposal is accepted (ACCEPT_PROPOSAL), the agent uses the indicatePath method to simulate moving towards the point and updates its position.

If all points have been visited, the agent informs all points and the coordinator that all deliveries have been made.

3- Utility methods:

calculateDistance: Calculates the distance between the delivery agent's current location and a given point.

getPointCoordinates: Sends a query to the specified point to get its coordinates.

informAllPoints: Informs all points that all deliveries have been made.

updatePosition: Updates the delivery agent's position after visiting a point.

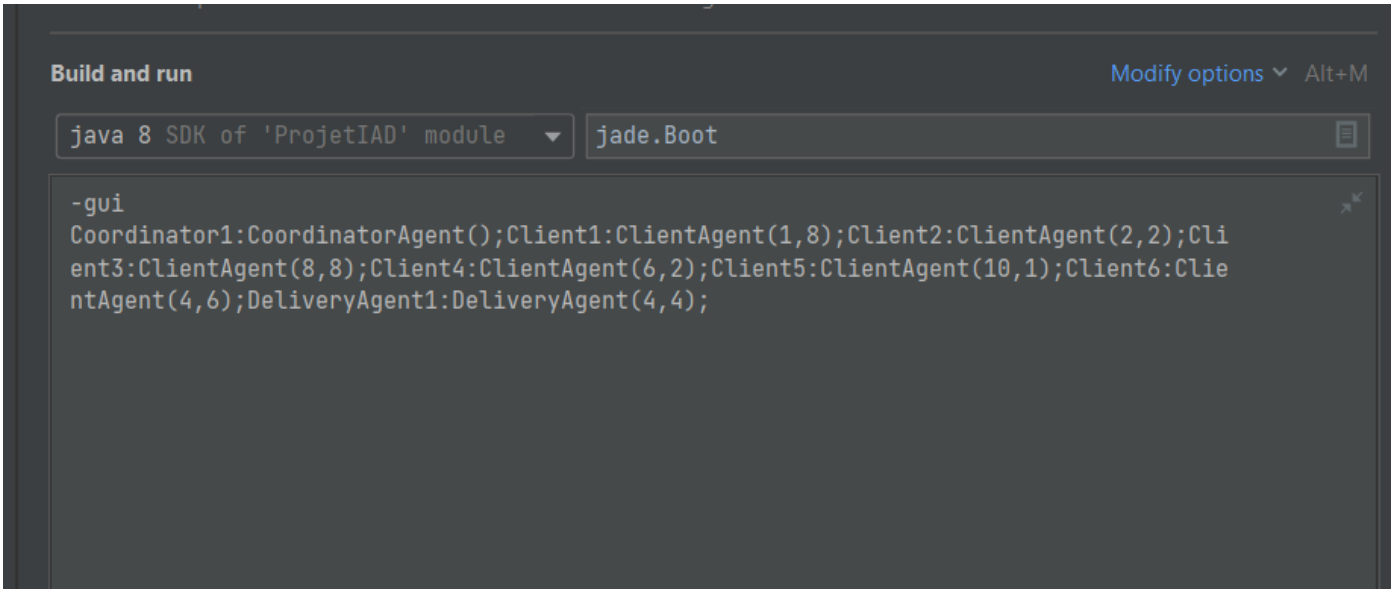
getCoordinatorAgent: Finds and returns the coordinator agent.

informCoordinator: Informs the coordinator of the points visited.

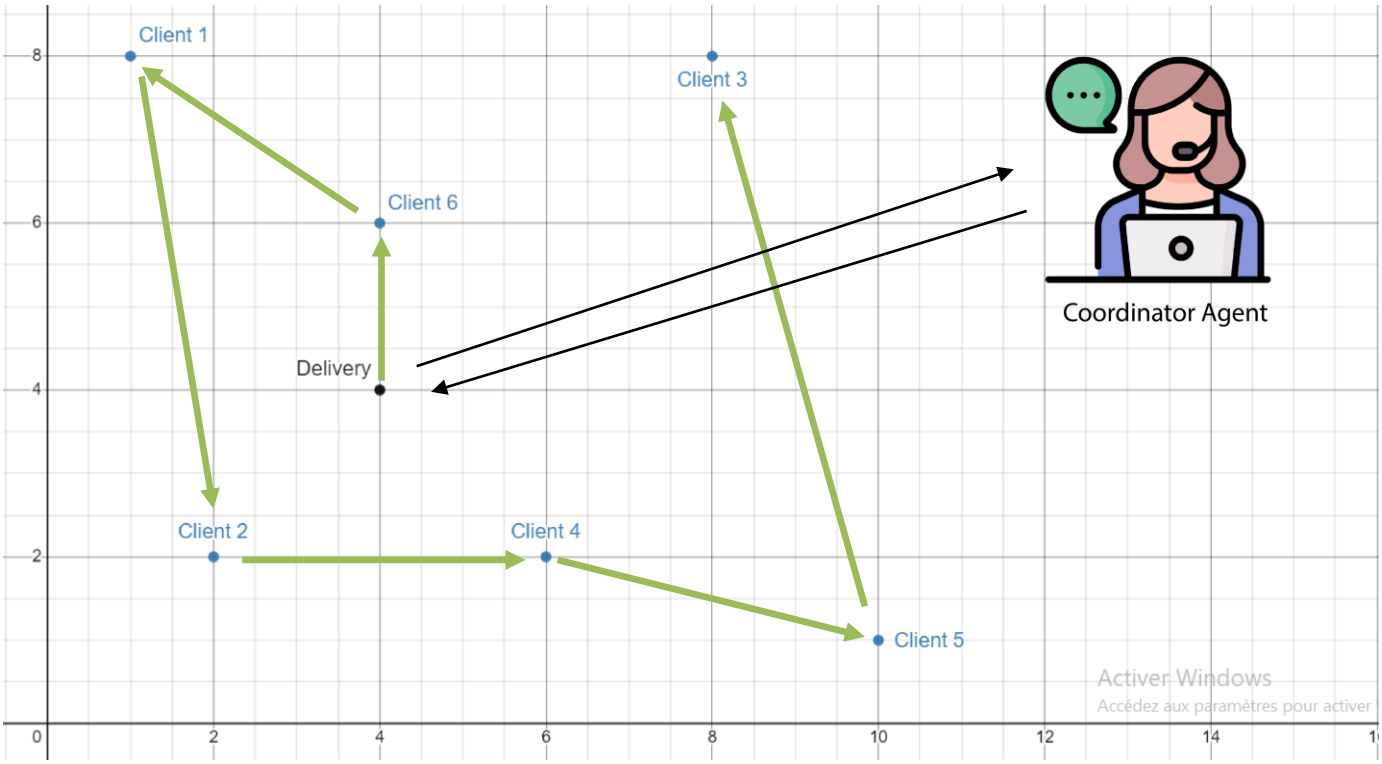
```java
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class CoordinatorAgent extends Agent {

    protected void setup() {

        // Add behavior to receive visited points information
        // Register the Coordinator agent with the "Coordinator" service type
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());

        ServiceDescription sd = new ServiceDescription();
        sd.setType("Coordinator");
        sd.setName("CoordinatorAgent");
        dfd.addServices(sd);

        try {
            DFService.register(this, dfd);
            System.out.println("Coordinator Agent " + getLocalName() + " is ready.");

            // Add behavior to receive visited points information
            addBehaviour(new ReceiveVisitedPointsBehaviour());
        } catch (FIPAException e) {
            e.printStackTrace();
        }
        addBehaviour(new ReceiveVisitedPointsBehaviour());
    }

    private class ReceiveVisitedPointsBehaviour extends CyclicBehaviour {
        public void action() {
            MessageTemplate informTemplate = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
            ACLMessage inform = receive(informTemplate);

            if (inform != null) {
                String content = inform.getContent();
                if (content.startsWith("VisitedPoints:")) {
                    String visitedPointsInfo = content.substring("VisitedPoints:".length());
                    System.out.println(getLocalName() + ": Client delivered so far " +
visitedPointsInfo);
                } else if (inform.getContent().startsWith("finish")) {
                    System.out.println(getLocalName() + ": Great Job Delivery Agent Have a Good Day!!
");
                } else {
                    block();
                }
            }
        }
    }
}
```

Example of application execution:

We tried to run our project with these parameters:



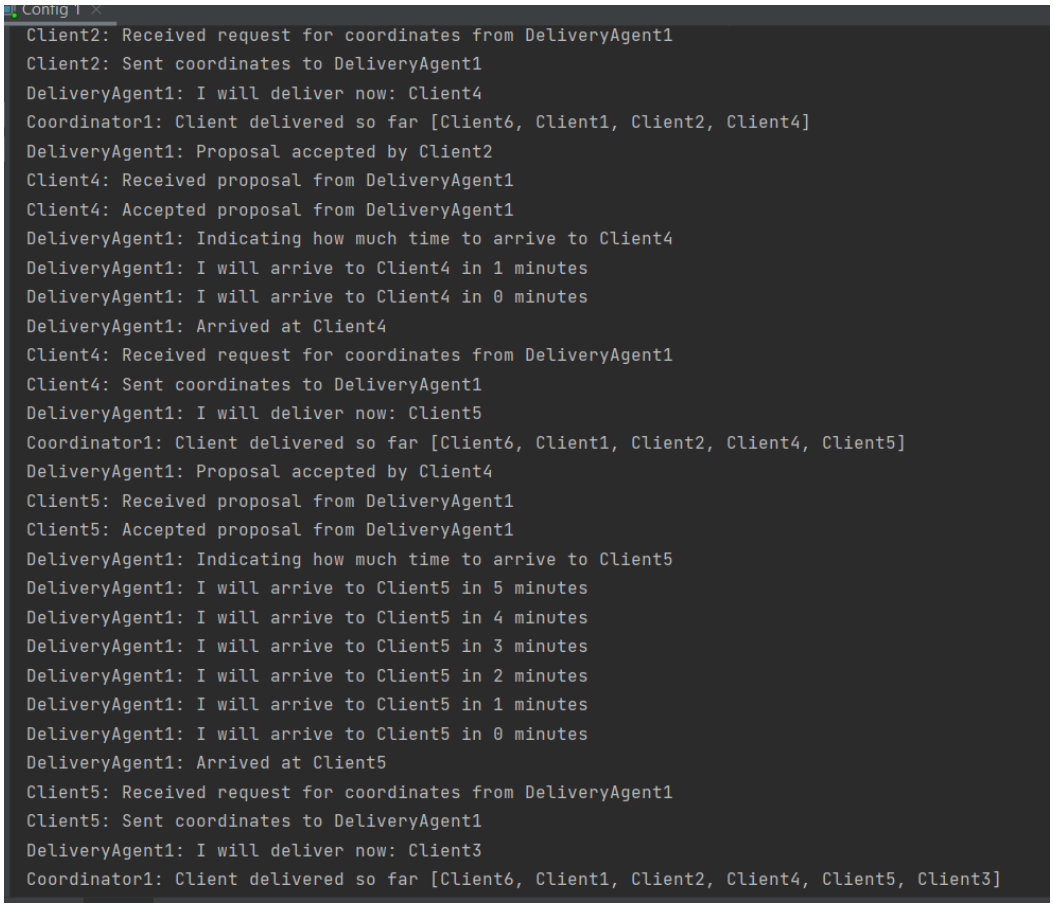
In this way we will have such a representation of our customers and deliverers:



In this example we notice that the deliveryman has moved towards the nearest customer 6 to deliver it then he continues his journey by delivering each time to the nearest customer while synchronizing his work with the coordinating agent by exchange of messages each time indicating the time per minute remaining to reach the next customer. and here is the display result:

```
http://DESKTOP-2VK6B4G:7778/acc
DeliveryAgent1: Initial position: x: 4, y: 4
déc. 11, 2023 2:26:12 AM jade.core.AgentContainerImpl joinPlatform
INFOS: ----------------------------------------
Agent container Main-Container@192.168.233.1 is ready.
------------------------------------------
Client4: Initial position : x: 6, y: 2
Client2: Initial position : x: 2, y: 2
Client6: Initial position : x: 4, y: 6
Coordinator Agent Coordinator1 is ready.
Client1: Initial position : x: 1, y: 8
Client3: Initial position : x: 8, y: 8
Client5: Initial position : x: 10, y: 1
Client2: Received request for coordinates from DeliveryAgent1
Client2: Sent coordinates to DeliveryAgent1
Client1: Received request for coordinates from DeliveryAgent1
Client1: Sent coordinates to DeliveryAgent1
Client4: Received request for coordinates from DeliveryAgent1
Client4: Sent coordinates to DeliveryAgent1
Client5: Received request for coordinates from DeliveryAgent1
Client5: Sent coordinates to DeliveryAgent1
Client6: Received request for coordinates from DeliveryAgent1
Client6: Sent coordinates to DeliveryAgent1
Client3: Received request for coordinates from DeliveryAgent1
Client3: Sent coordinates to DeliveryAgent1
DeliveryAgent1: I will deliver now: Client6
Coordinator1: Client delivered so far [Client6]
Client6: Received proposal from DeliveryAgent1
Client6: Accepted proposal from DeliveryAgent1
```

```
Client6: Received request for coordinates from DeliveryAgent1
Client6: Sent coordinates to DeliveryAgent1
DeliveryAgent1: I will deliver now: Client1
Coordinator1: Client delivered so far [Client6, Client1]
DeliveryAgent1: Proposal accepted by Client6
Client1: Received proposal from DeliveryAgent1
Client1: Accepted proposal from DeliveryAgent1
DeliveryAgent1: Indicating how much time to arrive to Client1
DeliveryAgent1: I will arrive to Client1 in 3 minutes
DeliveryAgent1: I will arrive to Client1 in 2 minutes
DeliveryAgent1: I will arrive to Client1 in 1 minutes
DeliveryAgent1: I will arrive to Client1 in 0 minutes
DeliveryAgent1: Arrived at Client1
Client1: Received request for coordinates from DeliveryAgent1
Client1: Sent coordinates to DeliveryAgent1
DeliveryAgent1: I will deliver now: Client2
Coordinator1: Client delivered so far [Client6, Client1, Client2]
DeliveryAgent1: Proposal accepted by Client1
Client2: Received proposal from DeliveryAgent1
Client2: Accepted proposal from DeliveryAgent1
DeliveryAgent1: Indicating how much time to arrive to Client2
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: I will arrive to Client2 in 0 minutes
DeliveryAgent1: Arrived at Client2
```




```
DeliveryAgent1: Proposal accepted by Client5
Client3: Received proposal from DeliveryAgent1
Client3: Accepted proposal from DeliveryAgent1
DeliveryAgent1: Indicating how much time to arrive to Client3
DeliveryAgent1: I will arrive to Client3 in 9 minutes
DeliveryAgent1: I will arrive to Client3 in 8 minutes
DeliveryAgent1: I will arrive to Client3 in 7 minutes
DeliveryAgent1: I will arrive to Client3 in 6 minutes
DeliveryAgent1: I will arrive to Client3 in 5 minutes
DeliveryAgent1: I will arrive to Client3 in 4 minutes
DeliveryAgent1: I will arrive to Client3 in 3 minutes
DeliveryAgent1: I will arrive to Client3 in 2 minutes
DeliveryAgent1: I will arrive to Client3 in 1 minutes
DeliveryAgent1: I will arrive to Client3 in 0 minutes
DeliveryAgent1: Arrived at Client3
Client3: Received request for coordinates from DeliveryAgent1
Client3: Sent coordinates to DeliveryAgent1
DeliveryAgent1: Coordinator! All points have been visited.
Coordinator1: Great Job Delivery Agent Have a Good Day!!
```

Possible interaction schemes implemented

The Client, Delivery and Coordinator agents interact with each other via ACL messages. The messages exchanged are as follows:

- “Position?” (Request) and (Reply) message to receive the “Inform” type response: sent by a Delivery agent to request the contact details of a Customer agent.
- “Propose” message: sent by a Delivery agent to a Customer agent to propose a destination point.
- "Accept_Proposal" message: sent by a Customer agent to accept the proposal from a Delivery agent.
- “Reject_Proposal” message: sent by a Client agent to reject the proposal of a Delivery agent.

Client agents and the Coordinator agent also interact with each other via ACL messages. The messages exchanged are as follows:

- "Inform" message: sent by a Delivery agent to the Coordinator agent to inform of the selection of a destination point, and then the latter informs the rest of the clients of the Delivery agent's destination and how much time remains to arrive.

Conclusion :

In this report, we presented a solution to a navigation problem in a dynamic environment. Our solution is based on the use of autonomous agents, capable of communicating with each other and negotiating routes.

Additionally, our solution is flexible. It can be adapted to different types of environments and constraints (examples: Taxi Drivers, etc.).