CS534 Lab

Antoine CHAPEL, Loup LOBET, Enzo MAZZOLENI

November 25, 2024

Contents

Chapter 1

The MaDKit framework

1.1 What's MaDKit

MaDKit (Multi-agent Development Kit) is a Java-based framework for developing simple multi-agent systems. It leverages the AGR (Agent-Group-Role) organizational model to structure agent interactions. In AGR, agents belong to groups and assume roles within those groups. Groups represent collaborative contexts, and roles define the specific functions or responsibilities agents fulfill within a group.

The framework is built around the `AbstractAgent/Agent` class, which provides the basic functionalities for agent behavior, including lifecycle management, messaging, and organization handling; AgentAddress, which represents the unique identity of an agent in a specific role within a group. Communication between agents occurs through asynchronous message passing, enabling agents to interact based on their roles without direct dependencies. MaDKit's design facilitates modular system construction, making it ideal for research, simulation, and the development of distributed and adaptive systems.

MaDKit agents behaviour can be programmed using three class methods: the `activate()` method, which is executed only once when initialising the agent, the `live()` method, which is executed at every agent life cycle, and the `end()` method, which is similar to the `activate()` method but is executed on the agent's death.

Chapter 2

Role-playing game

2.1 Turn around

We can represent our multi-agent system will consists of three different types of agent (with dynamic roles), the first one the Emitter agent which will send messages at random intervals to the Counter agents, then will die when they finished sending all there messages. The Counter agents will count the number of messages received from the Emitter agent. After a random time a counter agent dies and transforms itself in an emitter agent, and send its current counter value to the Controller agent. The Controller agent purpose is to keep the Counter agent number constant, to do this, whenever it receives a counter value, it creates a new Counter agent with the same counter value, who's read to count again. And so on. All these mechanics are represented in the following diagram (Figure 2.1).

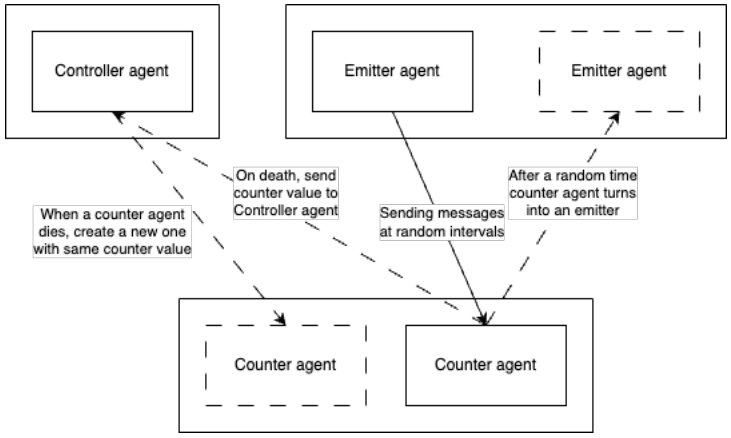

Figure 2.1: Agent diagram

The MaDKit framework is based on a "groups" for message sending and receiving, this means we cannot send messages directly to a specific agent, we first need to gather an agent by specifying the group it belongs to (with the `getAgentWithRole()` function). When initialising an agent we can specify the group it belongs to with the MaDKit java function: `getAgentWithRole()`. In our situation we can create 3 groups: one for the emitter agents, one for the controller agent, and one for the counter agents.

2.2 Emitter agent

To implement the behaviour of the emitter agent we need to randomly choose a fixed number of message to send before dying. Then we send the messages to the Counter agent group (`ROLECOUNTER`), and we wait a random time between each sendings. We can implement the emitter agent `live()` method as follow:

```
protected void live() {
    // we pick a random number between 1 and 10
    int nb_messages = new Random().nextInt(5, 15);

    AgentAddress other = null;
    while (other == null) {
        other = getAgentWithRole(COMMUNITY, GROUP, ROLECOUNTER);
        pause(50);
    } // loop until we found a random counter agent

    // we start sending the messages
    for (int i = 0; i < nb_messages; i++) {
        ReturnCode ret = sendMessage(other, new Message());
        // verify the counter agent didn't die.
        // If so, we find a new counter agent.
        if (ret != ReturnCode.SUCCESS) {
            while (other == null) {
                other = getAgentWithRole(
                    COMMUNITY,
                    GROUP,
                    ROLECOUNTER
                );
            }
            // the message wasn't sent successfully
            // do not increment the message counter.
            i--;
        }
        // we wait for a random time before sending a new message
        pause(random.nextInt(500, 1500));
    }
}
```

We can see in the above code that once we've selected a counter agent to send our messages to, we need to keep track of if this agent is still alive or not. We check if the message was sent successfully or not. If the counter agent is dead, we start looking for a new counter agent inside the message sending loop.

2.3 Counter agent

For the counter agent we simply use a loop to listen for the incoming messages (from the emitter agents), and we setup a timer with a random time for the agent to die after a random time. While receiving the messages the agent increments its message counter. When the agent will die it sends its current counter value to the controller agent, and transforms itself into a new emitter agent. We can implement the counter agent `live()` method as follow:

```
protected void live() {
    long start_time = System.currentTimeMillis();
    long stop_time = random.nextInt(1000, 10000);

    Message m;
    while (start_time + (System.currentTimeMillis()
               - start_time) < start_time + stop_time) {
        m = nextMessage();
        if (m != null) {
            getLogger().info("A new message ! : " + m.getSender());
            message_counter++;
            getLogger().info("message_counter: " + message_counter);
        }
        pause(1000);
    }
    getLogger().info("counter dying...");

    // send dying message to the controller
    AgentAddress controller = null;
    while (controller == null) {
        // select controller agent
        controller = getAgentWithRole(
            COMMUNITY,
            GROUP,
            ROLECONTROLLER
    }
    // send current message counter to the controller agent
    getLogger().info("Notify death to controller agent");
    sendMessage(controller, new IntegerMessage(message_counter));
    // launch new emitter agent
    launchAgent(new EmitterAgent(), true);
    pause(1000);
}
```

There is no particular difficulty for this agent the implementation is pretty much straightforward. We can also note that we could have used the agent `end()` method to handle the agent death (and send the counter value to the controller agent).

2.4 ControllerAgent

The controller agent goal is to spawn new counter agent whenever a existing counter agent dies. The newly created counter agents must have the just died counter agent message counter value. So we need to define a new constructor for the `CounterAgent` class takes an integer as an argument to specify the initial message counter value. To implement the controller agent behaviour we simply listen for incoming messages (containing a message counter value), and we spawn a new counter agent with the received counter value. We can implement the controller agent `live()` method as follow:

```
protected void live() {
    // listen for dying counters
    IntegerMessage m;
    while (true) {
        m = (IntegerMessage) waitNextMessage();
        if (m != null) {
            getLogger().info("A counter is dying: " + m.getSender());
            // spawn new counter with starting with the received integer
            getLogger().info("Spawn new counter");
            launchAgent(new CounterAgent(m.getContent()), true);
        }
    }
}
```

2.5 Simulate

When running the simulation, everything seems to work properly the emitter and counter agent dying and spawning as the simulation goes on, and the counter agents counter values is kept between counter agent deaths and spawns.

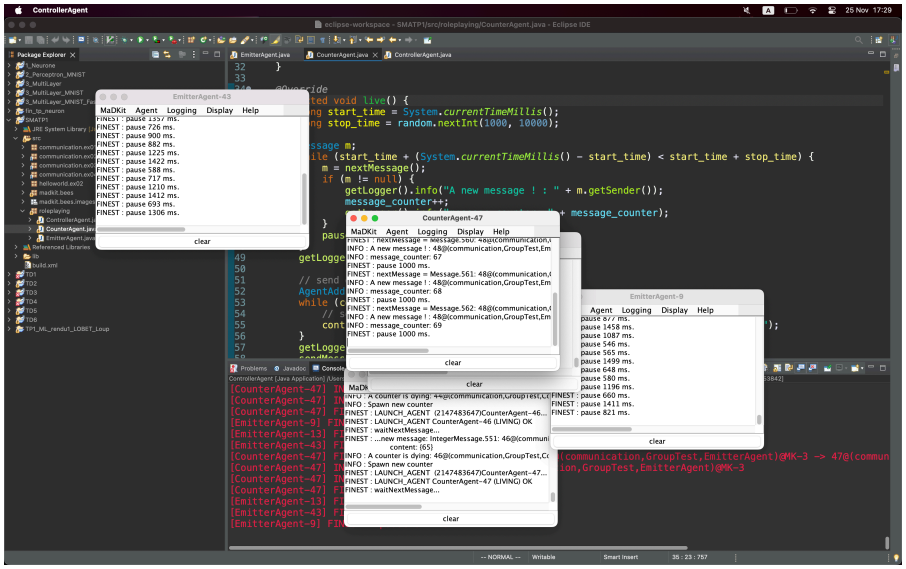

Figure 2.2: simulate the role play

Chapter 3

Adaptation of Bees

3.1 Source code modifications

To draw the queens as circles (and not lines, like other bees), we modify the method `computeFromInfoProbe()` in the `BeeViewer` file, using the `fillOval()` method. We need to modify the following code snippet:

```
if (trailMode) {
    final Point p1 = b.getPreviousPosition();
    g.drawLine(p1.x, p1.y, p.x, p.y);
} else {
    g.drawLine(p.x, p.y, p.x, p.y);
}
```

We need to check if the `AbstractBee` object we're drawing is an instance of `QueenBee`, if so we draw the queen with an oval. In the same fashion, we add modifications to also draw the hornets with a different shape (e.g. rectangle):

```
if (arg0 instanceof QueenBee) {
    if (trailMode) {
        final Point p1 = b.getPreviousPosition();
        g.fillOval(p1.x, p1.y, 15, 10);
    } else {
        g.fillOval(p.x, p.y, 15, 10);
    }
}
else if (arg0 instanceof Hornet) {
    if (trailMode) {
        final Point p1 = b.getPreviousPosition();
        g.fillRect(p1.x, p1.y, 20, 20);
    } else {
        g.fillRect(p.x, p.y, 20, 20);
    }
}
else {
    if (trailMode) {
```

```
            final Point p1 = b.getPreviousPosition();
            g.drawLine(p1.x, p1.y, p.x, p.y);
        } else {
            g.drawLine(p.x, p.y, p.x, p.y);
        }
    }
```

Chapter 4

Prey and Predators

The goal of this chapter is to explain the implementation of the hornet and the modification of the bees behaviour. The first step was to create a `Hornet` class from the `AbstractBee` with the desired behaviour.

4.1 Hornet behaviour

When the hornet spawns the first thing to do is to inform all the bees that a new hornet has appeared (see Figure 4.1). To do this we make the hornet broadcast a message to all the members of the `FOLLOWER_ROLE` as shown bellow:

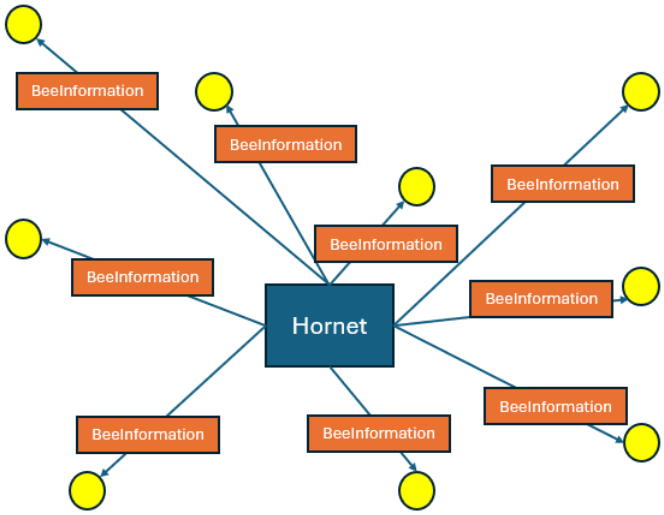

Figure 4.1: Hornet broadcast on spawning

```
protected void activate() {
    requestRole(COMMUNITY, SIMU_GROUP, HORNET_ROLE, null);
    // !!! we have to add this second role just for the viewer
    requestRole(COMMUNITY, SIMU_GROUP, BEE_ROLE, null);
    broadcastMessage(
        COMMUNITY,
```

```
            SIMU_GROUP,
            FOLLOWER_ROLE,
            new ObjectMessage<>(myInformation)
        );
}
```

Then in order for the bees to interact with an hornet, we need the bees to compute there own distance with the hornet in the `computeNewVelocity` method (see Figure 4.2). We define zone of influence for the hornet, which is circle around itt (with an arbitrary radius). When a bee enters in the hornet zone, that means it can be attacked by the hornet, but it also means the bee can contribute to the hornet death if there is a sufficient number of other bees in the range.

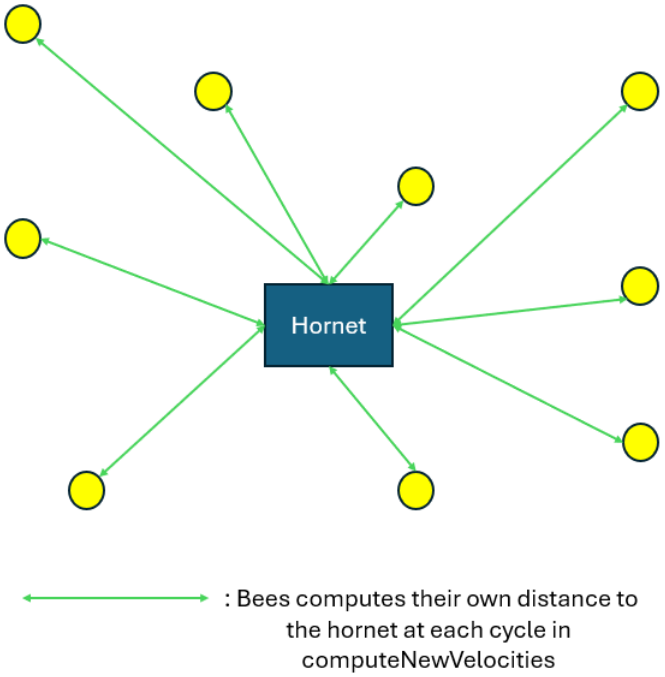

Figure 4.2: Bees compute their distance from the hornet at each lifecycle

When a bee enters the hornet range, it sends a `IntegerMessage` to the hornet with the id 1, and the hornet increments a intern counter that count the current number of bees being in its range (See Figure 4.3). In the same fashion a bee exiting the hornet range will send a message with the id 0 to the hornet, decreasing its counter by one. This mechanism allows the hornet to know the current number of bees it is surrounded by, and know if he needs to die or not (if the hornet is surrounded by sufficient number of bees, it will die).

Also a bee dying and (because of the hornet or because of the `BeeLauncher`, and being in a hornet range, need to inform that hornets that it is dead in order to avoid counter leaks. The implementation of this behaviour is demonstrated in the following code snippet:

```
 IntegerMessage m = (IntegerMessage) nextMessage();
if (m != null) {
```

```
    if (m.getContent().equals(1)) {
        beesInRange++;
        //handle the stop for the selected bee
        if(stopValue==0) {
            startTime=System.currentTimeMillis();
            sendMessage(m.getSender(), new ObjectMessage<>(null));
            stopValue=1;
        }

    } else if (m.getContent().equals(0)) {
        // a bee is quitting the range
        if (beesInRange > 0) {
            beesInRange--;
        } else {
            System.out.println("Erreur beesInRange negatif");
        }
    } else if (m.getContent().equals(3)) {
        //a bee has been killed by the simulation, we update the counter
        if (beesInRange > 0) {
            beesInRange--;
        }
    }
    //if too many bees are in the range, the hornet die
    if (beesInRange > 15) {
        killAgent(this);
    }
}
```

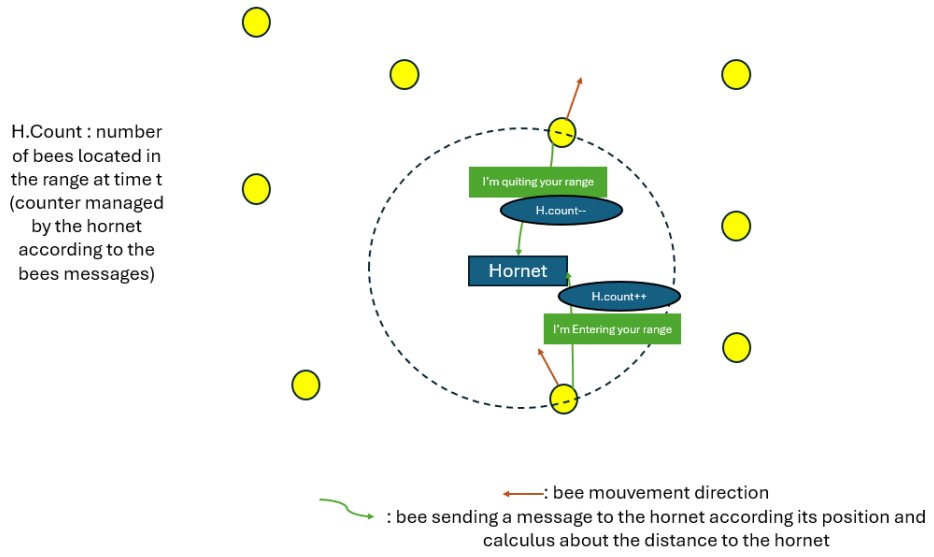

Figure 4.3: Bees compute their distance from the hornet at each lifecycle

Now that the hornet can keep track of the number of bees it is surrounded by, the next step for the hornet is to be able to choose an in-range bee to kill. When a bee enters the hornet range, if the hornet isn't currently attacking an other bee, in sends a message to that bee, so the bee knows it is going to be killed. The two of them will freeze (not move) during the killing process (see Figure 4.4), which takes around one second. The hornet is frozen when it kills a bee but still processes its counter to know if it is going to die because there are too many bees in its range (as shown in the above code).

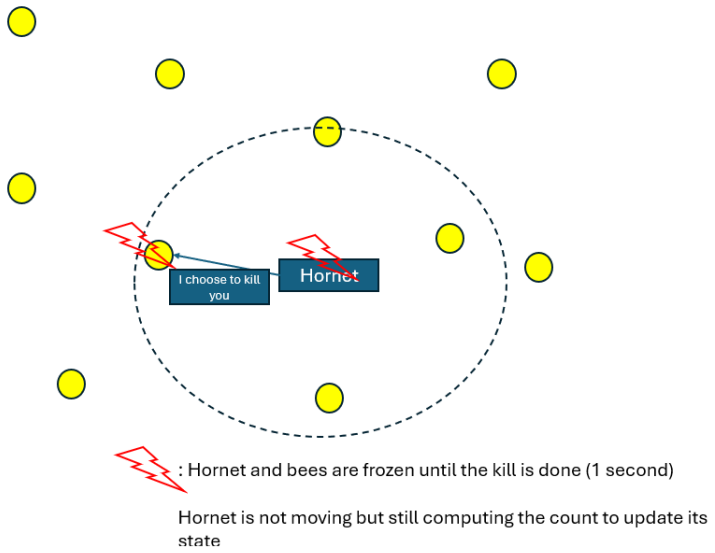

Figure 4.4: Bees compute their distance from the hornet at each lifecycle

We implement this behaviour in the hornet `computeNewVelocities()` method:

```
protected void computeNewVelocities() {
    if (beeWorld != null) {
        if(stopValue==1 && (System.currentTimeMillis()<startTime+500)) {
            //if the hornet is killing the bee, it doesn't move
            System.out.println("[H] Killing the bee, stop moving");
            dX=0;
            dY=0;
        }
        else if(stopValue==1 && System.currentTimeMillis()>=startTime+500) {
            stopValue=0;
            beesInRange--;
            System.out.println("[H] I killed the bee, begin to move again");
        }
        else {
            //normal moving for the hornet
            int acc = beeWorld.getHornetAcceleration().getValue();
            dX += randomFromRange(acc);
            dY += randomFromRange(acc);
        }
```

```
    }
}
```

NB. : please not that for easy reading not all the written code is inserted in this report. Please directly read the corresponding files to get full implementation.