Agent and Multi-Agent Systems: architectures and reasoning

Messaging communication in Mesa: Tuto

WO

June 14, 2022

CentraleSupélec

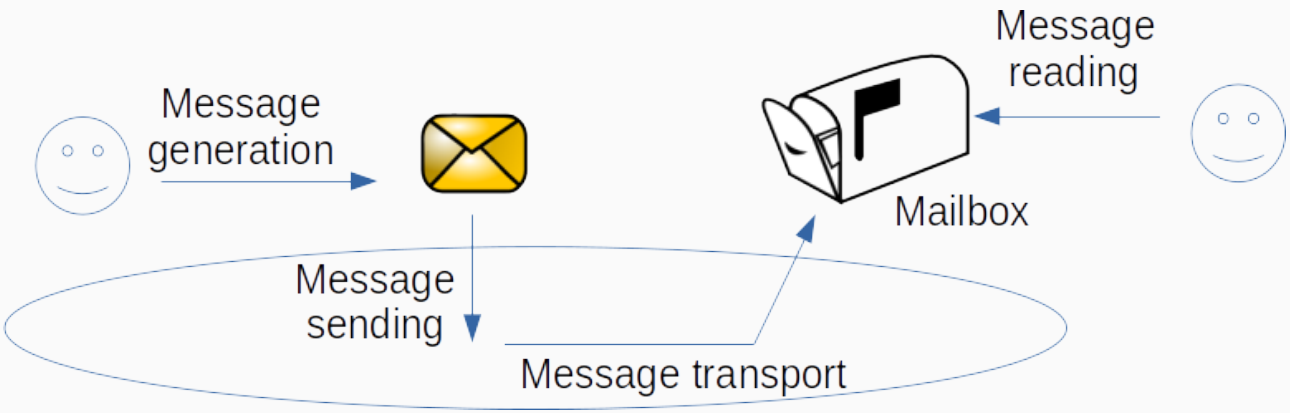
Table of contents

Introduction

Implementing messaging communication in Mesa

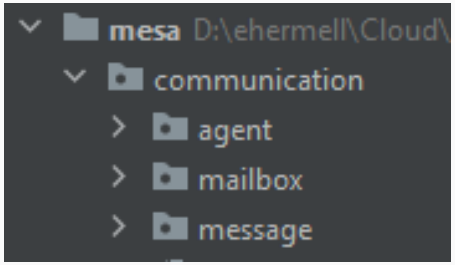
With the Mesa library, it is possible to set up indirect interaction mechanisms through the environment (like a blackboard or stigmergy mechanism, see today's session, section 1). However, it does not handle direct interactions such as message communication. So, first, we will implement our own communication layer in Mesa. This communication layer will be necessary for the realization of the final project.

Before implementing the communicating agents and according to the picture, we must create: (1) a `Message` object, (2) a `Mailbox` object and (3) a list of allowed performative for messages.



Implementing messaging communication in Mesa

To do so, open the folder mesa and you should have the afollowing:



Here is the description of what each folder will contains:

- **mesa**: root folder which will contain your python codes using the communication layer;
- **communication**: the root folder of the communication layer;
- **agent**: the folder which will contain the implementation of the communicating agent class;
- **mailbox**: the folder which will contain the implementation of the mailbox class;
- **message**: the folder which will contain the implementation of the message and performative class.

Question: create an *__init__.py* file in the communication, agent, mailbox and message folders to initiate python packages.

Messages

Messages

Enter the message folder. You have the file Message.py implementing the `Message` class. The purpose of this Message class is to create a python message object containing the receiver and sender identifiers but also the performative of the message sent as well as a content. Agents will exchange information using these items during their communication phases. The Message class is therefore composed of four attributes of the `Message` class:

- `from_agent`: the sender of the message identified by its id;
- `to_agent`: the receiver of the message identified by its id;
- message_performative: the performative of the message;
- content: the content of the message.

Messages

Enter the message folder. You have the file Message.py implementing the `Message` class. The purpose of this Message class is to create a python message object containing the receiver and sender identifiers but also the performative of the message sent as well as a content. Agents will exchange information using these items during their communication phases. The Message class is therefore composed of four accessor methods:

- `get_exp()`: return the sender of the message;
- `get_dest()`: return the receiver of the message;
- `get_performative()`: return the performative of the message;
- `get_content()`: return the content of the message.

Messages

Now that we have a usable `Message` object, we are going to create the set of allowed message performatives from a python enumeration. We will define seven message performatives for the moment. It will be very easy to add more later. The seven performatives are as follows:

- propose;
- accept;
- commit;
- ask why;
- argue;
- query;
- inform.

In the message folder, a file called MessagePerformative.py implements the `MessagePerformative enumeration` class.

Mailbox

Mailbox

To manage messages, each communicating agent will have his own mailbox. The purpose of this class is to provide to agents some mechanisms for handling sent and received messages. So, the `Mailbox` class is composed of two attributes :

- *unread_messages*: the list of unread messages;
- *read_messages*: the list of read messages.

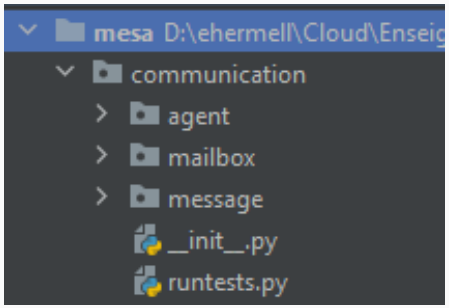
Mailbox

To manage messages, each communicating agent will have his own mailbox. The purpose of this class is to provide to agents some mechanisms for handling sent and received messages. So, the `Mailbox` class is composed of five methods.

- *receive_messages(message)*: receive a message and add it in the unread messages list;
- *get_new_messages()*: return all the messages from unread messages list;
- *get_messages()*: return all the messages from both unread and read messages list;
- *get_messages_from_performative(performative)*: return a list of messages which have the same performative;
- *get_messages_from_exp(exp)*: return a list of messages which have the same sender.

Practice Yourself

- The `Message` and `Mailbox` classes being created, we will test them.
- Go to the communication folder and create a new runtests.py file. In this file, we will incrementally add tests to verify that our implementations are working well.



Let's start testing the Mailbox class

Using the `assert()` function. Create three messages with various performatives, one mailbox and test the different methods of the `Mailbox` class (*receive_messages(message)*, *get_new_messages()*, *get_messages()*, *get_messages_from_exp()* and *get_messages_from_performative()*).

Message Service

Message Service

At this point, each agent will have their own mailbox instance and will be able to exchange messages. However, there are still no mechanisms to ensure that sent messages reach the right agents. As agents must not directly drop messages in the mailboxes of the other agents, we need to create a service (a message transport mechanism) which will be managed by the environment and which will take care of the management of message shipments and deliveries. This service is implemented in the `MessageService` class

Message Service

A *MessageService* object is a singleton: it can only be instantiated once. Thus, there can only be one postal service in the MAS.

This service has three attributes:

- *scheduler*: the scheduler of the SMA initialized in the Mesa model;
- *instant_delivery*: the instant delivery status of the MessageService. If True, the message will be delivered instantly in the mailbox of the agent;
- *messages_to_proceed*: the list of message to proceed.

one mutator method : *set_instant_delivery(instant_delivery)*: change the instant delivery status of the MessageService.

Four methods.

- *send_message(message)*: dispatch a given message if instant delivery is activated, otherwise add the message in the MessageService message list to be proceeded after;
- *dispatch_message(message)*: dispatch the given message to the right agent;
- *dispatch_messages()*: proceed and dispatch each message received by the message service;
- *find_agent_from_name(agent_name)*: return the agent according to the agent name given.

How to use MessageService Class?

To use `MessageService`, it is necessary to instantiate it in the constructor of the Mesa Model and give it as parameter the reference of the Scheduler:

```
def __init__(self):
self.schedule = RandomActivation(self)
self.__messages_service = MessageService(self.schedule)
```

Then, just call the method *dispatch_messages()* of the MessageService in the step() method of the Mesa Model.

```
def step(self):
            ...
            self.__messages_service.dispatch_messages()
            ...
```

The `MessageService` can dispatch the messages instantly or at each time step. You can change this behavior by calling the *set_instant_delivery(isntantly)* method and give as parameter a Boolean which represents by its value the activation or not of the instantaneous mode.

```
MessageService.get_instance().set_instant_delivery(False)
```

Communicating Agent

Communicating Agent

We now have all the tools implemented: (1) `Message`, (2) `MessagePerformative`, (3) `Mailbox` and (4) `MessageService`. We are going to create a communicating agent that inherits from **Mesa** `Agent` class. This `CommunicatingAgent` class is not intended to be used on its own, but must be inherit to create other agent classes.

If we refer to the four step mechanism presented on the previous section of the course, a communicating agent must be able to:

1. A communicating agent (sender) builds a message;
2. A communicating agent (sender) invokes a `send` method in the environment (through the `MessageService`) to send the message, as one of its actions;
3. The environment (through the `MessageService`) drops the message in the mailbox of the communicating agent (receiver). This can be done either instantly or not.
4. A communicating agent (receiver) reads its mailbox, either in a systematic manner as part of the perception mechanism in the procedural loop (*passive perception*) or on purpose, by calling a specific method (*active perception*).

The `CommunicatingAgent` must therefore have three attributes, one accessor method and seven methods. Go to the **agent** folder and create a new *CommunicatingAgent.py* file. Let's open this file and implement the `CommunicatingAgent` class

The three attributes of the `CommunicatingAgent` class are:

- *name*: the name of the communicating agent which replaces the unique id of **Mesa** `Agent` class;
- *mailbox*: the agent's unique and private mailbox;
- *message_service*: the reference to the message service instantiated in the environment.

The acceessor method of the `CommunicatingAgent` class is:

- *get_name()*: return the unique name of the communicating agent.

The seven methods of the `CommunicatingAgent` class are:

- *step()*: the `step()` methods of the communicating agent called by the **Mesa** `Scheduler` at each time tick;
- *receive_message(message)*: receive a message (called by the `MessageService`) and store it in the mailbox;
- *send_message(message)*: send message through the `MessageService` (`messages_service.send_message(message)`);
- *get_new_messages()*: return all the unread messages;
- *get_messages()*: return all the received messages;
- *get_messages_from_performative(performative)*: return a list of messages which have the same performative;
- *get_messages_from_exp(exp)*: return a list of messages which have the same sender.

As you can see, many methods of the `CommunicatingAgent` class are the same as the methods of the `Mailbox` class: we have chosen to have mailbox accessible only through dedicated methods which makes it private.

Practice Yourself !

All the features of the Mesa communication layer being implemented, we will complete the runtests.py file to test the `MessageService` and `CommunicatingAgent` classes. To do this, we will implement a Mesa `Model` and communicating agents:

1. Open the runtests.py file;
2. Implement a `TestAgent` class which inherits from `CommunicatingAgent` class;
3. Implement a `TestModel` class which inherits from Mesa `Model`. This model instantiate the `MessageService` and two communicating agents;
4. In the if *_name_* == "*_main_*":, instantiate the `TestModel` and make the two communicating agents communicate through messages. Use the `assert()` function to ensure that the behaviors (number of messages sent, received, etc.) are those expected.

Mesa: implement Alice-Bob-Charles example

The objective here is to reimplement the simple example Alice-Bob MAS by using the communication layer that we have just integrated in Mesa.

Create two communicating agents named Alice and Bob. Create a third agent named Charles whose role is hold a variable v and process messages from Alice and Bob:

- On their turn, Alice and Bob ask Charles for the value of v, using a message;
- If the value is different from their preferred value, they send a message to Charles to change the value of v;
- On its turn, Charles reads its mailbox and processes all messages:
  - Messages that request information about v produce an answer;
  - Messages that request a change to v are applied.