CentraleSupélec

Multiagent systems
The Lost Island (part 1)

N. Sabouret and W. Ouerdane

January the 23rd, 2019

Preamble

Finish the previous practical session (introduction to JADE) before you start this one.

1 On the way to the Lost Island

The practical sessions in this Multi-Agent System Course will be devoted to the programming of a negotiation & argumentation simulation. Agents representing human participants will have preferences over several options and will need to negotiate with each other to make a common decision.

The application context that we will consider is a very classical "toy-example" in argumentation. Imagine a group of people that need to select a limited number of items for a survival trial on a lost island. Each possible item has different constraints (*weight* or *size*) and different values depending on different criteria (*e.g.* a tent is good for *comfort*, a fishing pole is useful for *food*, etc) and the agent's view on the item. The negotiation comes when the participants have different preferences on the criteria and the argumentation will be used to help them decide which item to select.

The argumentation framework will be discovered and implemented later in the course. The goal of today's session is only to implement communication protocols to support the negotiation.

2 Preferences

Each object that can be carried onto the Lost Island is characterised by *values* on a given set of *criteria*. Each participant can attribute different values for the same item on the same criteria (for example, one agent can rate the item "milk" as "very useful" on the criterion "food" while another considers that it is "useless" because it has milk allergy). In addition, each agent has a partial or total order on the criteria themselves. This defines the participants' *preferences*.

It is also possible, if you want, to define orders on the criteria values themselves. While most criteria will have a "natural" preference order that apply to all participants (everyone prefers a cheap, comfortable, useful and solid item), you can also imagine some criteria that have no natural order of preference. For example, consider the criterion "food", you can have the values "fish", "meat", "fruit" and "vegetables". Each participant can have a different order of preferences for such a criterion.

However, in the first version of the project, we will focus on naturally ordered criteria. The agents will be able to attribute different values to the same items on each criterion and they will have to negotiate based on these local values and their preference order on the criteria.

Exercise 1

1. Implements the required classes to encode the items, the criteria and the values. The items should have:

- A name represented by a `String` value;
- A value for each possible criterion;

One of the criteria must be the *cost* for taking the item (*e.g.* representing its weight). Examples of criteria (other than cost) include: food, sleep, hygiene, health, comfort, calling for help, solidity... Feel free to invent whatever criterion you think is relevant!

2. Build a sufficient number of instances of objects. You do not need to cover all possible combinations of criteria values, but a good coverage is important if you want to have an interesting argumentation.

3. Implement the `Preference` class whose instances represent the preferences of participants. The preferences consists of:

   - Value of each item on each criterion;
   - An order on the criteria;
   - Possibly, some (partial) orders for criteria that have no natural order.

   One participant will be associated with a single instance of this class.

4. Write different constructors for your model of preferences: you will need a random constructor but also a constructor that receives specific values for testing (either from reading a file or from an internal data structure).

5. Write in the `Preference` class an aggregation function that computes the score of a given object based on its values and the corresponding preferences. The aggregation function will be a simple Ordered Weight Average with arbitrary weights. You can you the same weights for all instances of the `Preference` class.

3 Agents

3.1 Participants

The agents will represent the participants of the expedition to the Lost Island. For the moment, you will only consider two such agents. Multiparty negotiation will only be discussed in future sessions if time allows.

Exercise 2

1. Implement a class `Adventurer` as a JADE agent. These agents must:

   (a) Have an instance of `Preference`, initialised through setup parameters;
   (b) Print a debug message to indicate they are ready to negotiate;
   (c) Begin with a simple listener behaviour that awaits for messages.

2. Implement a `Launcher` class that loads N adventurers, $N = 2$ being defined as a public static final integer variable in the `Adventurer` class.

3.2 Supervisor

In addition to these agents, we need one agent to supervise the outcome of the negotiation. While this can be done by each agent separately, it might be convenient to have one thing at a single spot.

Exercise 3

1. Implement a `Storage` JADE agent that can store a list of *selected* items.
2. For convenient reasons, the list of all *possible* items (that you built in Exercice 1) shall be defined as a public static final variable in the `Storage` class. This information will be visible to all agents.

   Don't get confused: you have two different lists. The static variable of possible items, and the (local to `Storage`) list of selected items.
3. Implement a private method that computes the remaining available cost, based on `MAXIMUM` (defined as a public static final integer variable) and the currently *selected* items.

3.2.1 Remaning cost

This agent must be able to answer information requests from other agents about the remaining available cost. To this goal, we will use a classical QUERY_REF/INFORM_REF protocol. The initiator agent builds a message of performative QUERY_REF with the string "remaining cost" as content. The receiver answers with a performative INFORM_REF and the string "remaining cost=X" with X the actual remaining cost.

Exercise 4

1. Draw (using the AUML standard) the protocol that corresponds to such a communication.
2. Implement a listening behaviour in the `Storage` agent that allows the agent to answer to remaining cost requests from other agents.
3. Implement a querying behaviour in the `Adventurer` agent that simply displays the result and test your protocol.

3.2.2 Item selection

Another feature that is required for the Supervisor agent is to change the list of selected items as soon as all N agents agreed to given option. In the next section, we will use an ad-hoc message performative to accept a proposal: the TAKE performative. The idea is that once the two `Adventurer` agents agree on taking an item, the initiator of the proposal will send a message to the `Storage` agent with the performative TAKE and the name of the item as the content.

Exercise 5

1. Define the TAKE performative as a public static final int variable of value 100 (so that it does not interfere with one of the 22 FIPA-ACL performatives). The content of a TAKE message will always be the *name* of the selected item.
2. Implement a listening behaviour that deals with TAKE messages.
3. Test this behaviour using a `OneShotBehaviour` in your Adventurer class that sends a TAKE message.

4 Messages

During the negotiations, participants will exchange messages about the items and their preferences. The communication model we are going to implement in this practical course relies on the several performatives. We will use QUERY_REF, INFORM_REF and TAKE for the interactions between the adventurers and the storage agent (this has already been implemented). We will use 5 performatives for the negotiation between two adventurers: PROPOSE, ACCEPT, COMMIT, ASK_WHY and ARGUE. The next subsection explains the role of each performative.

4.1 List of performatives

4.1.1 PROPOSE

This performative is used to propose to take an item. Its content is the name of the item. For example:

```
Agent1 to Agent2: PROPOSE(bread)
```

4.1.2 ACCEPT

This performative is used to accept to take an item. Its content is the item's name. If should always appear after a PROPOSE of the same item, but several other message can exist between the proposal and the acceptance. For example:

```
Agent1 to Agent2: PROPOSE(knife)
...
Agent2 to Agent1: ACCEPT(knife)
```

4.1.3 COMMIT

This performative is used to confirm that an object has to be taken. Its content is an item's name. Both agents must send (and receive) this message before the protocol's initiator can refer about it to the storage agent. Agents must only commit to taking an object once they are sure to take it, which means:

1. They have enough remaining *cost* in the storage;
2. One agent proposed to take it;
3. The other agent (or all other agents in the case of a multiparty negotiation) accepted to take it.

For example:

```
Agent1 to Agent2:  PROPOSE(bread)
Agent2 to Agent1:  ACCEPT(bread)
Agent1 to Storage: QUERY_REF(remaining cost)
Storage to Agent1: INFORM_REF(remaining cost = 10)
Agent1 to Agent2:  COMMIT(bread)
Agent2 to Agent1:  COMMIT(bread)
```

Agents should not commit to take an item unless they ensure that there is enough space left on the storage. Once they have send a COMMIT message, agents consider that the item is no longer in the list of available items.

4.1.4 ASK_WHY

This performative is used to ask another agent to propose arguments for taking an object. Its content is an item's name. It should be used after a propose. For example:

```
Agent1 to Agent2: PROPOSE(bread)
Agent2 to Agent1: ASK_WHY(bread)
```

This message expects an ARGUE message in answer.

4.1.5 ARGUE

This performative is at the core of the negociation process. It is used to explain another agent the arguments in favor or against a given item. As will be explained in the theoretical course on negotiation and argumentation, the idea is to communicate a logical inference that defends the agent's position. As a consequence, the content of such a message is of the form:

```
A => B
```

With A a logical formula whose terms are called the premises, and B a positive or negative literal, which is the conclusion. For example, an agent might want to explain another agent that it favors the item "bread" because it is easy to transport:

```
bread <= transport=easy
```

Similarly, an agent might explain another agent that "bread" should not be taken because because it is useless for comfort:

```
not bread <= comfort=useless
```

Content

In the context of this practical course, we will limit our **premises** to two types of propositions:

- The value of criterion C is X;

  This is a know fact for both agents, but using it in an ARGUE message simply asserts that this information can be used by the sender agent to infer a position in favor or against the item.

  In our communication model, we shall write:

  ```
  C = X
  ```

- Criterion C_1 is more important than criterion C_2;

  This information is local to the sender agent: it does not necessary hold for the receiver. However, using it in an ARGUE message informs the receiver about the sender's preferences. In other words, it gives an argument in favor or against the item.

  In our communication model, we shall write:

  ```
  C1 > C2
  ```

Those two possible types of propositions can be combined in a message. The following paragraphs give some concrete examples.

As for the **conclusion**, it must be the name of an item, preceded by the operator "not" when the inference is against the item.

Example 1

Here is a very simple dialogue in which Agent 2 gives an argument **in favor of** taking the item "knife" based on its value for the criterion "transport".

```
Agent1 to Agent2: ASK_WHY(knife)
Agent2 to Agent1: ARGUE(knife <= transport=easy)
```

Example 2

Here is a very simple dialogue in which Agent 1 gives an argument **against** taking the item "knife" based on its value for the criterion "food" and its **local** preference for criterion "food" over "transport".

```
Agent1 to Agent2: ASK_WHY(knife)
Agent2 to Agent1: ARGUE(knife <= transport=easy)
Agent1 to Agent2: ARGUE(not knife <= food=useless, food>transport)
```

When several premises are used, they are separated by a comma. This must be interpreted as the conjunction operator (AND).

4.2 Complete example

Here is a sample dialogue with all performatives. Note that this does not describe a single communication protocol, but rather the achievement of 4 or 5 different protocols that interleave.

4.2.1 Preferences

Let us consider two agents (Agent1 and Agent2). They have to choose only one item between the bread and the knife: there is only room left for one of them. The system considers four different criteria: transport, protect, food and comfort. For simplicity, in this example, both agent have exactly the same value for each item.

	transport	protect	food	comfort
bread	easy (**)	useless (**)	very useful (***)	useless (*)
knife	very easy(***)	very useful (***)	useless (**)	useless (*)

The criteria have a natural order (represented by stars in the table). However, agents have different order of preferences on the criteria themselves (total orders in our example):

```
Agent1: food > protect > comfort > transport
Agent2: protect > food > transport > comfort
```

4.2.2 Dialogue

The following example dialogue could be produced by our agents. Each line is composed of the number of the dialogue turn, the name of the sender agent (the recipient agent is the other one), and the message itself.

```
01: Agent1 - PROPOSE(bread)
02: Agent2 - ASK_WHY(bread)
03: Agent1 - ARGUE(bread <= food=very useful)
04: Agent2 - PROPOSE(knife)
05: Agent1 - ASK_WHY(knife)
06: Agent2 - ARGUE(knife <= transport=very easy)
07: Agent1 - ARGUE(not knife <= food=useless, food>transport)
08: Agent2 - ARGUE(not bread <= protect=useless, protect>food)
09: Agent1 - ACCEPT(knife)
10: Agent1 - COMMIT(knife)
11: Agent2 - COMMIT(knife)
```

At this point of the negotiation, the agent that **made the accepted proposal** can store the knife in the storage:

```
12: Agent2 to Storage - TAKE(knife)
```

4.2.3 Concluding remarks

That the inference mechanism is not described in this example. In a future practical session, you will have to implement the reasoning process that computes the necessary arguments in favor or against items and to select which one to use in the negotiation.

It is also possible that two order of preferences conflict in such a way that it is not possible to conclude on a given item. For example, imagine the following exchange of information:

```
Agent1 to Agent2: ARGUE(bread <= food>protect)
Agent2 to Agent1: ARGUE(not bread <= protect>food)
```

In such a situation, the agents must infer that they cannot conclude on the proposed item. The initiator should withdraw its proposal and the negotiation should no longer consider it as a candidate for the trip to the lost Island[1]. You can add a new performative to handle such situations.

4.3 Implementation

Exercise 6

1. Implement the above performatives as public static final integer variables in the `Adventurer` class. You shall use specific integer values that do not conflict for each of your newly created performative.

2. We recommend that you implement a data structure for the content of ARGUE. Write an `Argument` class that consists of a tuple with:

   - The item name;
   - A boolean value (for positive or negative arguments);
   - A list of couples that can be either $(criterion, value)$ and/or $(criterion, criterion)$.

   Using the required getter, setters and `toString` methods, you will be able to use this data structure both for the decision model and for the ACL communication.

3. It is possible to use either raw content for ACL messages (*i.e.* Strings) and to write a simple parser to process the content of ARGUE messages. This is not very difficult but you will have to trim your substrings to remove the whitespaces. You can also use the ContentObject methods to transport objects in ACL messages, as long as you have a serializable object. Choose your preferred method!

5 Protocols

Agents will follow specific protocols when interacting. We shall use the AUML standard to draw each protocol.

Exercise 7

1. Draw the protocol that correspond to a proposal with an ACCEPT or a ASK_WHY.

2. Implement this protocol in your agent, with a random selection between ACCEPT and ASK_WHY.

3. Test this protocol with random values. No decision is made by the agent at this point.

[1] In a more advanced model, agents could be able to argue about their order of preferences so as to solve such conflicts, instead of dropping the object.

Exercise 8

1. Draw the protocols that correspond to a request for explanation (ASK_WHY followed by one single ARGUE).
2. Implement this protocol in your agent, with a random selection for the explanation's content. Again, no decision is made at this point.
3. Test this protocol.

6 Decision and argumentation

In order to make decisions about the items to propose, to accept, to attack or to defend, the agents will required several features:

- Find their favourite item in the current list of opened item;
- Decide whether or not to accept a proposal;
- Build an argumentation graph.

In this first session, we will only consider two negotiator agents, and one Storage agent.

6.1 Finding the favourite item

Agents will always propose their favourite item.

Exercise 9

1. Generate two `Adventurer` agents and one storage agent with a list of items, with different criteria and values. The list of items is the same for both agents. Each agent however has its own set of preferences, as seen in exercise 1.
2. Using the methods implemented in exercise 1 and the OWA that computes the value of each item, write a method that allows an agent to select its most preferred item in a list. In case of equality, select one randomly.
3. Write and implement a protocol for an `Adventurer` agent to ask the `Storage` about the remaining cost, filter the list of items with this cost and selecting the most preferred one and fits within the remaining cost.
4. Write and implement a protocol in which both `Adventurer` agents register to the `Storage` agent as soon as they are ready to make their first proposal. The `Storage` agent waits for both agents to be ready and calls for the first one that registered. This agent will then contact the other one with its proposal, following the protocol defined in exercise 7.

6.2 Deciding to accept for a proposal

Agents will accept any proposal that corresponds to an item that belongs to the top 10% of its preferred items list.

Exercise 10

1. Write a method that checks whether an item belongs to the 10% most preferred one of the agent.
2. Use this method to decide between ACCEPT and ASK_WHY in the protocol.
3. In case of acceptance, write and implement the protocol that corresponds to the double-COMMIT interaction. Both agents simply send a COMMIT message to their interlocutor as soon as they have the three following information:

- One agent made a proposal on the item;
- The other agent accept the proposal on the item;
- The item fits in the remaining cost.

4. Implements the TAKE protocol, that is triggered by the agent that proposed the item that was accepted. This agent sends a message to the `Storage` agent.

5. The `Storage` agent and both `Adventurer` agents shall update the list of available items. We then move back to exercise 9: both agents will selected a new item for proposal, declare this to the `Storage` agent and a new PROPOSE protocol can be initiated.

6.3 Building the argumentation graph

In order to answer to an ASK_WHY performative, agents must be able to write the content of the ARGUE message, following the data structure from Exercise 6. Several approaches are possible. In this project, we will extract the list of arguments **during the construction of the initial proposal**.

For example, in the dialogue from section 4.2.2, Agent 1 proposes to take the item "bread". The argument, used at turn #3 of the dialogue, is based on the value of the criterion "food" for this item.

The algorithm extracts the list of criteria on which the item has a "good enough" value. In our example, we can assume that "***" and "****" are the "good enough" values. This threshold must be a parameter for each criteria.

The agent then selects the criterion, in this list, with the highest position in its own preference order.

Exercise 11

1. Write a method that computes the list of possible arguments for defending a proposal.

2. Implement and test a decision procedure that answer with an ARGUE message when a proposal is questioned.

When one agent receives an ARGUE message, three situations are possible:

- If the agent cannot contradict the argument, it shall accept the proposal;
- If it can contradict the argument, it can answer with another ARGUE message;
- It can also answer with a PROPOSE message, which is another manner to attack the argument. In the example from section 4.2.2, turn #6 attacks turn #3.

The agent can contradict the argument if:

- The criterion is not important for him.

  This means that some criterion with a "low value" ("*" or "**") is more important in its own preference order. In the example from section 4.2.2, the agent considers that protection is more important than food. This is used in turn #8 to attack turn #3.

- The local value for the agent is not the same as the one given as arguments.

  For example, if one agent is allergic to Milk, it will not accept the argument `food=very useful`.

- It prefers another item and has arguments to defend this item.

Exercise 12

1. Write a method that decides whether an argument can be contradicted or not;
2. Write the Jade behaviour that answers to an ARGUE message based on the possibilities of contradiction;
3. Test your behaviour on a concrete example.
4. You can deal with blocking situations (*e.g.* when two agents have opposite preferences) by terminating the protocol. The item is no longer considered as a possible candidate.

7 Report and following steps

7.1 Report

We expect that you write a 5 to 10 pages report about your negotiation model. Give some printouts of negotiations. Compute some statistics on the negotiation outputs depending on the simulation parameters.

You should clearly identify the parameters and the observed values.

7.2 Extensions

If time allows, you should consider the possible extensions to your work:

- Multiple party negotiation.

  When more than two agents negotiate, you must define a policy for the conversation turn and decide how to make the final decision on a particular item.

- Consider criteria with non-natural orders;
- Use a different aggregation rule for finding local the order on the items;
- Change the order of preferences when one agent accepts a proposal, based on the values that were exchanged during the negotiation;
- Consider additional information on the agents that could serve as explanations for the criteria values (such as "being allergic") and that could be used during the negotiation.