Profiling and optimization for Android applications on the tATAmI platform

Understanding the tATAmI Platform and the S-CLAIM Language

Jaber Hassoun

Advisor:

S. l. dr. ing. Andrei Olaru

University "Politehnica" of Bucharest

2013

CONTENTS

1 Introduction ... 3

2 The tatami Platform ... 4

2.1 Structure and Components ... 4

2.2 Scenarios ... 6

2.3 JADE ... 7

3 S-CLAIM ... 8

3.1 Semantics: ... 9

- Communication ... 9
- Mobility ... 9
- Agent ... 9
- Knowledge management ... 9
- Control primitives ... 9

3.2 Syntax ... 9

3.3 JAVA Functions ... 10

4 An Example PC/Android Scenario (The ProCon Debate Application) ... 11

4.1 The Concept ... 11

4.2 The Implemented Scenario ... 11

4.2.1 Agent Structure ... 11

4.2.2 Agent Implementation ... 11

4.2.3 Sequence Diagrams: ... 13

5 Conclusions: ... 18

6 Future Work: ... 19

References ... 20

1 INTRODUCTION

tATAmI is a multi-agent systems based platform for ambient intelligence applications. The realization of this platform was a collaborative effort of Andrei Olaru, Thi Thuy Nga Nguyen and Marius-Tudor Benea (Olaru, 2011) seeking a platform for the deployment and testing of AmI applications.

The ProCon application was a diploma project that implements a simple scenario, using the tATAmI platform, in which the users are able to introduce pro and con opinions on a subject, using their Android devices.

The goal of this research is to optimize the tATAmI platform and to profile and solve performance issues in the Android application and to optimize it in order to offer an enjoyable experience to the user and an improved work-flow and to make the deployment of the application easier.

Chapter 2 “The tATAmI Platform” introduces the platform discussing its structure and components, the ways scenarios and discussed and run and the features and benefits of Jade as the platform is underpinned by it.

Chapter 3 “S-CLAIM” goes into details about the S-CLAIM programming language; its syntax, its semantics and how it makes us of Java Functions.

Chapter 4 “The ProCon Scenario” describes the idea of this android application and how the implementation of a scenario of a debate over a subject would go about.

2 THE TATAMI PLATFORM

tATAmI (towards Agent Technology for Ambient Intelligence) is a platform designed and built having the following requirements in mind:

- the use a programming language for the high-level implementation of agents
- a modular and extendable structure
- an improved representation of knowledge that is easily translatable from and to XML and other serializable formats
- deployablity on mobile devices
- traceability and visualization
- the use of scenario-based simulation
- the possibility of integration with other platforms and protocols

Also, tATAmI is underpinned by JADE which is a popular and easy-to-use environment for the deployment of multi-agent systems, and which will be discussed later in this chapter.

2.1 STRUCTURE AND COMPONENTS

tATAmI has several components and they are:

- the ***Core*** component, containing the classes for agents, organized on several layers:
  - Agent communication, mobility, and management - JADE agents are used.
  - Agent logging and visualization - agents must send data about their activity, as well as their neighbors, to a centralized entity. Each agent also displays a window on the screen of its execution host.
  - Hierarchical mobility for agents - protocols and behaviors that allow agents to automatically move together with their parents.
  - Web service access - functionality to expose agents as web services and allow agents to access web services.

- S-CLAIM interpretation and execution - a parser for S-CLAIM agent description files, and the components that transform the definition into actual behavior.
- Knowledge Base - an interchangeable component that allows access to knowledge through a standard set of functions.
- Context-awareness - use of context matching for problem solving and exchange of relevant context information.

* the ***Simulation*** component, serving for the repeatable execution of scenarios:

- Uses as input XML files that completely define the execution scenario.
- Deploys the agents according to the scenarios, on the specified containers and machines.
- Sends "external" event messages, equivalent to perceptions of agents in a real environment.

* the ***Visualization*** component, that assures the centralized visualization of agent activity:

- Receives log reports and mobility events from agents.
- Displays all agent logs in a centralized, chronological manner.
- Displays the system structure (topology), showing relations between agents.

- Provides components for the automatic layout of agent windows on the screen of the machine they execute on.

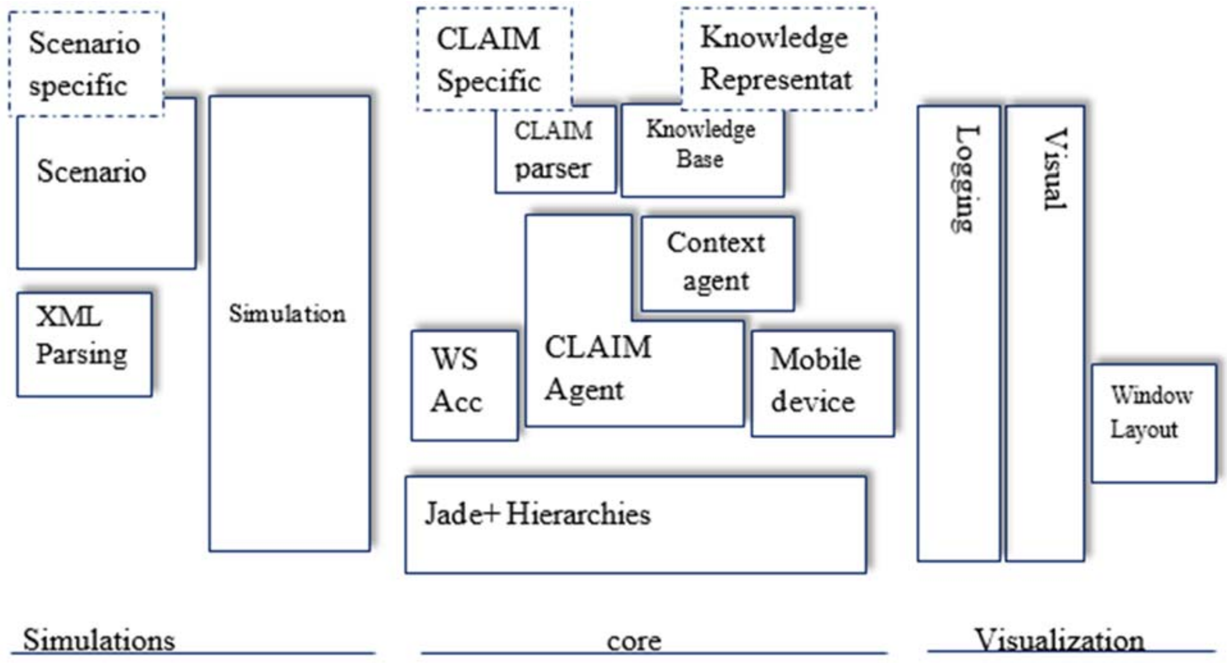


Fig. 2.1 An informal visual representation of the components of the tATAmI platform. With solid border, actual components of the implementation. In dotted line, specifications and formats that characterize the inputs of the components. (Olaru, 2011)

2.2 SCENARIOS

The scenario is specified by means of an XML file that contains information on the containers and agents to be created, on the initial knowledge of agents, and on the events to generate.

The purpose of a scenario is to reproduce an execution, and the mentioned information is all that is needed for this execution.

The features that are specified in the scenario file are the following:

- JADE configuration: sets details like the name of the main JADE container, the IP address and port of the platform and of the local machine, and the id of the platform.

- Path for the agent description files and packages with Java code attachments.

- Containers: names of the containers to be created on the local machine, and, in the case of the machine with the main container, the names of remote containers where agents will be created.

- Agents: inside the description of each container, the agent creation information is specified: agent name, agent type (name of the description file), initial knowledge of the agent, code attachment, agent parameters, gui information.

- Event information: timeline specifying what events to trigger at what moments of time in moments of time (in simulation time). Events are messages that will be sent to the specified agents.

2.3 JADE

JADE is an enabling technology, a middleware for the development and run-time execution of peer-to-peer applications which are based on the agents paradigm and which can seamlessly work and interoperate both in wired and wireless environment.

The agents paradigm applies concepts from artificial intelligence and speech act theory to the distributed object technology. The paradigm is based on the agent abstraction, a software component that is autonomous, proactive and social:

- *Autonomous*: agents have a degree of control on their own actions, they own their thread of control and, under some circumstances, and they are also able to take decisions;
- *Proactive*: agents do not only react in response to external events (i.e. a remote method call) but they also exhibit a goal-directed behavior and, where appropriate, are able to take initiative;
- *Social*: agents are able to, and need to, interact with other agents in order to accomplish their task and achieve the complete goal of the system.

Agent-based systems are intrinsically peer-to-peer: each agent is a peer that potentially needs to initiate a communication with any other agent as well as it is capable of providing capabilities to the rest of the agents.

JADE is fully developed in Java and offers the following features:

- Distributed applications composed of autonomous entities
- Negotiation and Coordination
- Pro-activity
- Multi-party applications
- Interoperability
- Openness
- Versatility
- Ease of use and mobile applications

3 S-CLAIM

S-Claim, whose name is an acronym for (Smart Computational Language for Autonomous Intelligent Mobile agents) (ProcediaComputerScience.), is an easy to use high-level declarative Agent-oriented programming language that was created to allow the representation of cognitive skills such as beliefs, goals and knowledge, while meeting the requirements of mobile computation and execution in smart environments.

S-CLAIM allows programmers to use the agent-oriented paradigm during the whole process of designing and implementing an AmI application, as it specifies only agent-related components and operations, leaving algorithmic processes aside. S-CLAIM also offers lightweight agents, cross-platform deployment and mobile device compatibility.

3.1 SEMANTICS:

The semantics of S-CLAIM are a simplification of the semantics of the CLAIM language (Suna & Seghrouchni, 2004), which reduces the list of primitives to only the ones which are characteristic to agent management and interaction. S-CLAIM specifies the following primitives (classified by their destination):

- **Communication**: *send, receive*
- **Mobility**: *in, out*
- **Agent management**: *open, acid, new*
- **Knowledge management**: addK, removeK, readK, forAllK
- **Control primitives**: condition, if, wait

3.2 SYNTAX

• S-CLAIM uses the notion of Blocks, where a Block is defined using the delimiters '(' and ')' and a keyword as follows:

(<keyword>)

• Variables in S-CLAIM are preceded by two question marks if they are meant to be re-assignable. Otherwise, a variable is preceded only by one question mark.

• A usual agent definition contains the name of the defined type of agent, the parameters for the agent, and its list of behaviors.

Three types of behavior are specified:

- *initial* : executed at the creation of the agent.

- *reactive* : executed as consequence of receiving a message and, optionally, fulfilling some conditions.

- *proactive* : goal-oriented behavior that executes without the need external events.

An example of an agent that only has one initial behavior :

```
(agent SimpleAgent ?destination
  (behavior
    (initial sender
       (send ?destination (struct message hello)))))
```

3.3 JAVA FUNCTIONS

S-CLAIM is a programming language that only proposes primitives that are directly related to the agent, its components, and the other agents in the system. These primitives can be used inside agent descriptions that make it easy for the agent designer to focus on the agent-related features.

There are processes that cannot be easily performed with the default primitives, this is why the developer can attach one or more Java class files that contain the needed functionality, and call the functions in exactly the same way as S-CLAIM primitives.

All S-CLAIM-attached Java functions have the same prototype: they take a Vector of values, some of which may be unbound. When the function ends, some of the unbound values may be bound, and in this way the function is able to fill missing pieces in patterns. And the function returns a boolean value, that can be used in the S-CLAIM code in if or condition statements.

These java functions are called from the agent's S-CLAIM code in the following manner, where "equalString" is the called function:

(condition (equalString ?name A1))

This simple example is a function that compares whether two strings are equal or not. Its java code is this:

```
public static boolean equalString(Vector<ClaimValue> arguments) {
            String s1 = (String) arguments.get(0).getValue();
            String s2 = (String) arguments.get(1).getValue();
            return s1.equals(s2);
}
```

4 An Example PC/Android Scenario (The ProCon Debate Application)

4.1 The Concept

The ProCon application is one on android which allows users to debate over a subject using their mobile java-based devices by typing their opinions and sending them after classifying each opinion as either positive (Pro) or negative (Con) regarding the subject in question.

4.2 The Implemented Scenario

(The stucture and relations of the three adf2 files and what not)

4.2.1 Agent Structure

There are three types of agents in this application that do all the work: *PDAAgent*, *EmissaryAgent* and *GroupCoordinatorAgent*.

- *PDAAgent* is the agent which interacts directly with the GUI and receives the orders from the user.
- *EmissaryAgent* is the one delegated by PDAAent to handle the communication instructions with the Group through the GroupCoordinatorAgent
- *GroupCoordinatorAgent* receives outer instructions only from the EmissaryAnget and as its name implies it handles the coordination among the agents letting them know each others' opinions and refreshes this knowledge with each update to any agent's opinion.

By Default, the initial taxonomy of agents is a simple one where the PDAAgent is the parent of the EmissaryAgent.

However, when an agent is added to a group, i.e. when the PDAAgent instructs the EmissaryAgent to register to the GroupCoordinatorAgent, this hierarchy slightly changes since the EmissaryAgent becomes a child of the GroupCoordinatorAgent.

4.2.2 Agent Implementation

In this section, parts of the S-CLAIM code will be presented showing the main functionalities of each agent class.

The PDAAgent's reacts to the instructions received from the GUI or from the EmissaryAgent and its main functions are joinGroup, addOpinion, deleteOpinion and displayOpinoin

```
(reactive joinGroup
      (input join tip ?groupCoordonatorAgent)
      (readK (struct knowledge emissary ?emissary))
      (send ?emissary (struct message leaveGroup))
      (send ?emissary (struct message join ?groupCoordonatorAgent)))

(reactive addOpinion
      (input add tip ?tag ?opinion)
      (readK (struct knowledge emissary ?emissary))
      (send ?emissary (struct message add ?tag ?opinion)))

(reactive deleteOpinion
      (input delete tip ?id)
      (readK (struct knowledge emissary ?emissary))
      (send ?emissary (struct message delete ?id)))

(reactive displayOpinion
      (receive displayOpinion ?opinion ?opinionTag)
      (if (equalString ?opinionTag pro)
            then (output proOpinion ?opinion))
      (if (equalString ?opinionTag con)
            then (output conOpinion ?opinion)))
```

In this section, parts of the S-CLAIM code will be presented showing the main functionalities of each agent class.

EmissaryAgent as mentioned before plays the mid-role between the PDAAgent and the GourpCoordiatorAgent, and its main tasks are handling joining and leaving groups, adding and removing opinions, and sending all the Agent’s opinions to the GroupCoordinator as the following piece of code shows:

```
(reactive joinGroup
      (receive join ?groupCoordonatorAgent)
      (in ?groupCoordonatorAgent)
      (send ?parent (struct message register ?name)))

(reactive leaveGroup
      (receive leaveGroup)
      (send ?parent (struct message unregister ?name)))

(reactive addOpinion
      (receive add ?opinionTag ?newOpinion)
      (removeK (struct knowledge sequence ?opinionId))
      (assembleOutput ?name ?opinionId ?newOpinion ?newCompleteOpinion)
      (increment ?opinionId ?newOpinionId)
      (addK (struct knowledge sequence ?newOpinionId))
      (send ?parent (struct message opinion ?opinionTag ?newCompleteOpinion))
      )
```

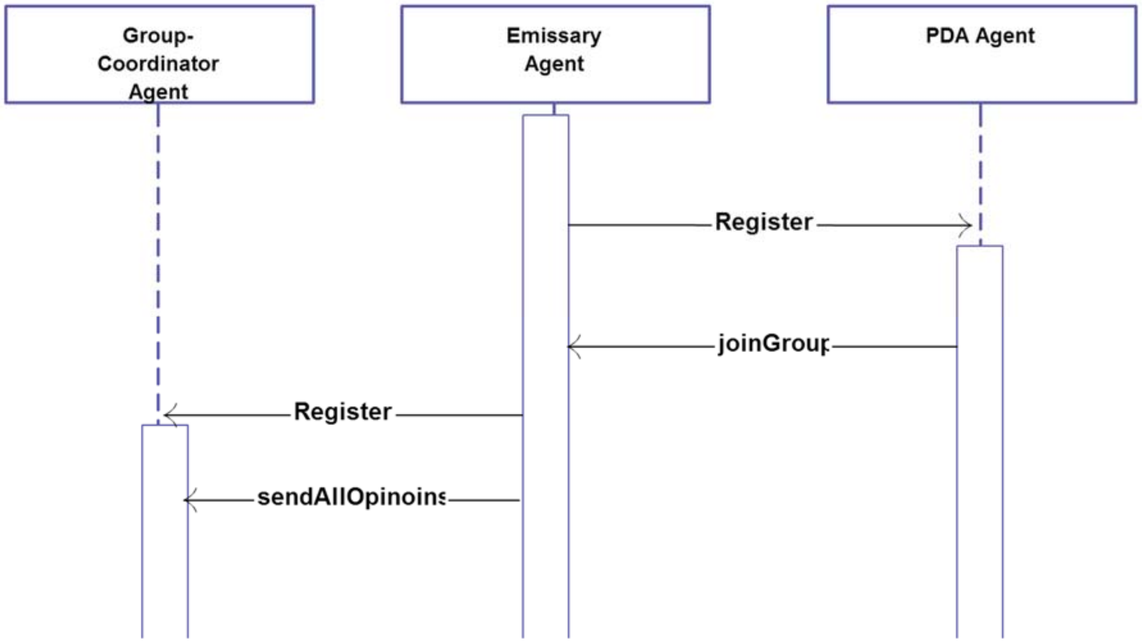
```
(reactive deleteOpinion
      (receive delete ?opinionId)
      (removeK (struct knowledge allOpinionsPro ?allOpPro))
      (removeK (struct knowledge allOpinionsCon ?allOpCon))
      (deleteOpinionFunction ?allOpPro ?opinionId)
      (deleteOpinionFunction ?allOpCon ?opinionId)
      (addK (struct knowledge allOpinionsPro ?allOpPro))
      (addK (struct knowledge allOpinionsCon ?allOpCon))
      (send ?parent (struct message refresh)))

(reactive sendAllOpinions
      (receive sendAllOpinions)
      (readK (struct knowledge allOpinionsPro ?allOpPro))
      (readK (struct knowledge allOpinionsCon ?allOpCon))
      (send ?parent (struct message opinion con ?allOpCon))
      (send ?parent (struct message opinion pro ?allOpPro)))
      )
```

And the GroupCoordinatiorAgent takes care for keeping all agents up to date with each others' opinions by reacting properly to the messages received from the EmissaryAgent.

4.2.3 **Sequence Diagrams:**

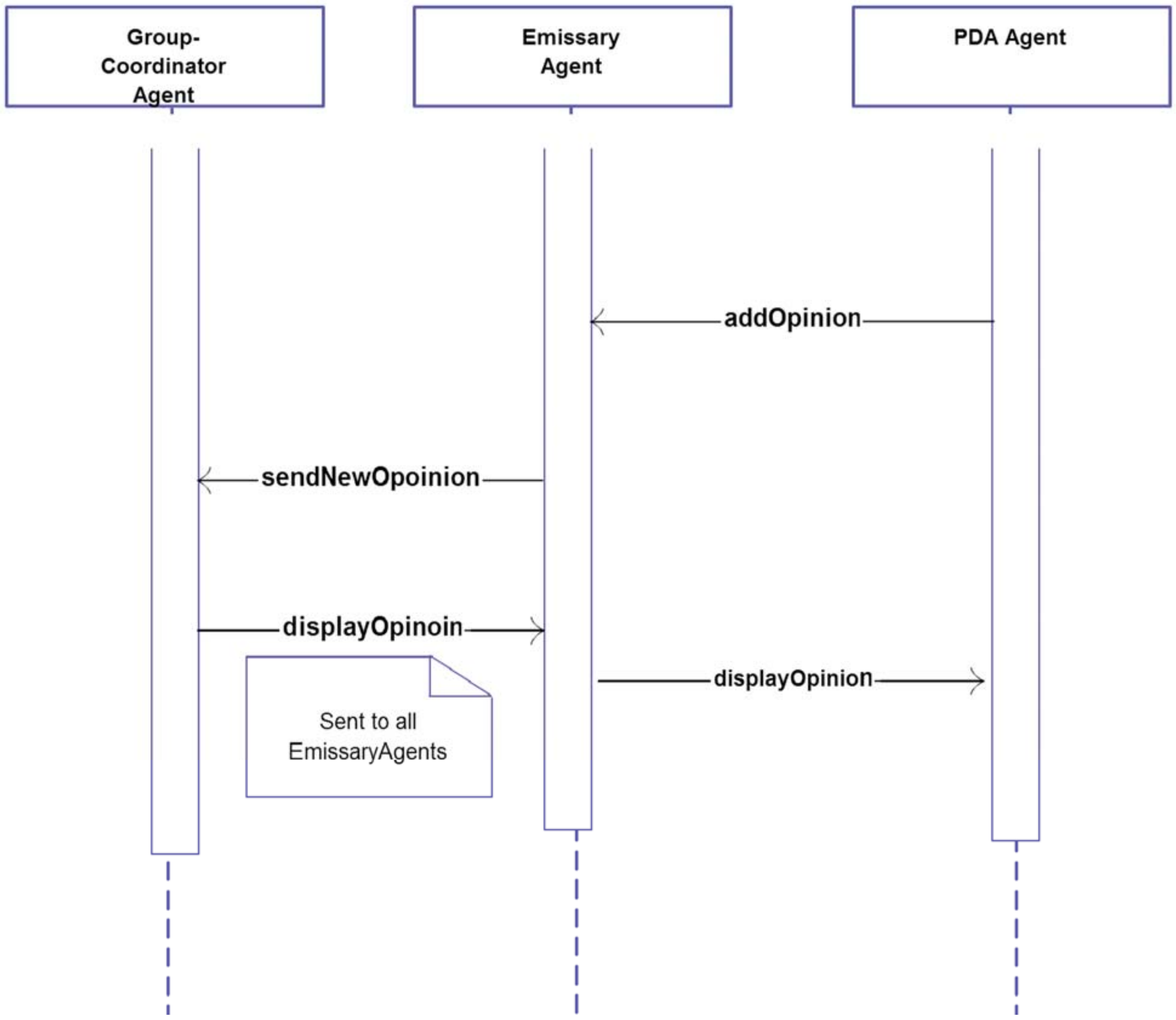
The following diagrams will show how exactly the mentioned tasks are achieved and will clarify the communication processes

➢ Joining a Group:



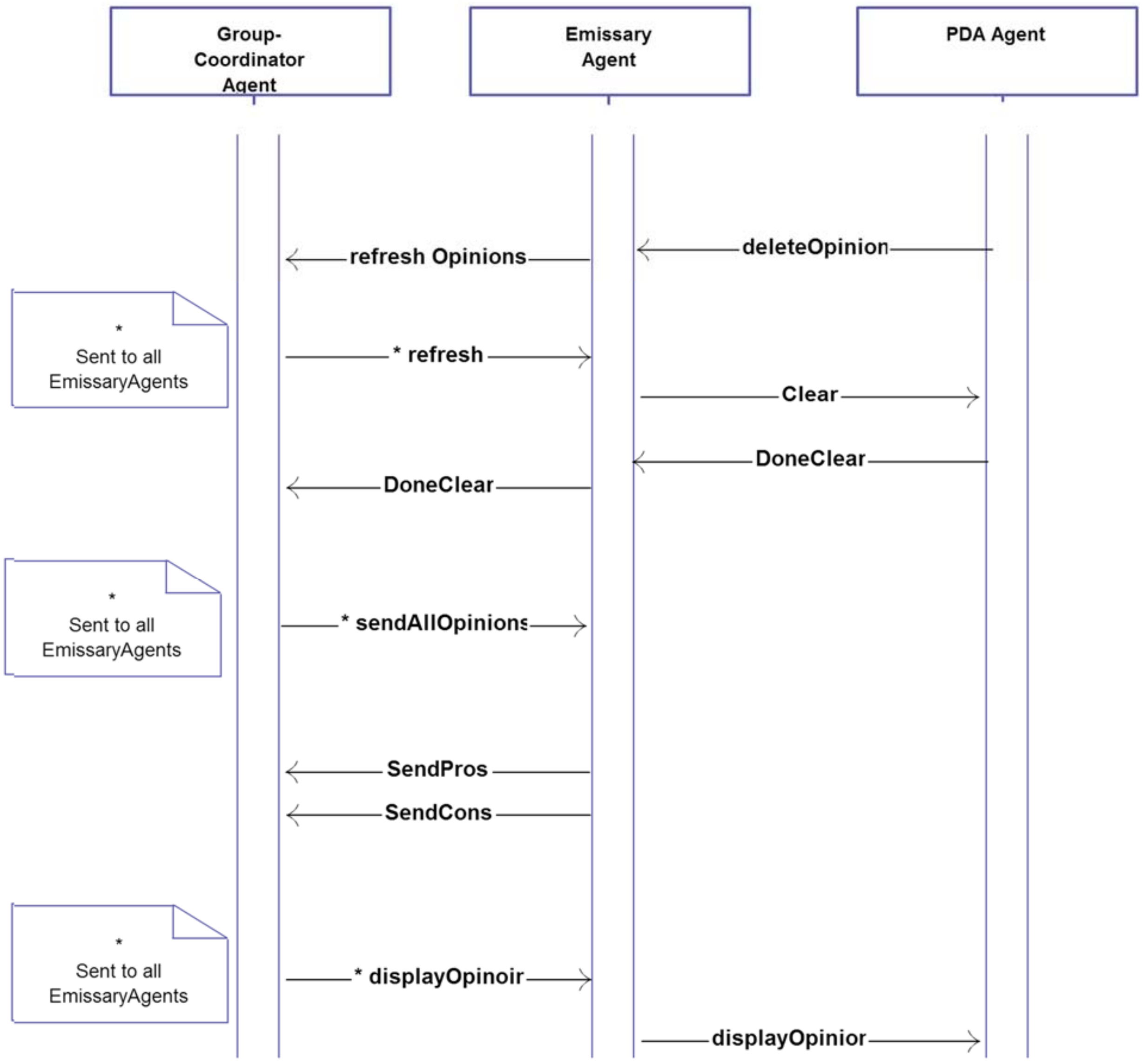
4.1 Sequence Diagram showing how an agent joins a group

➢ Adding an opinion:



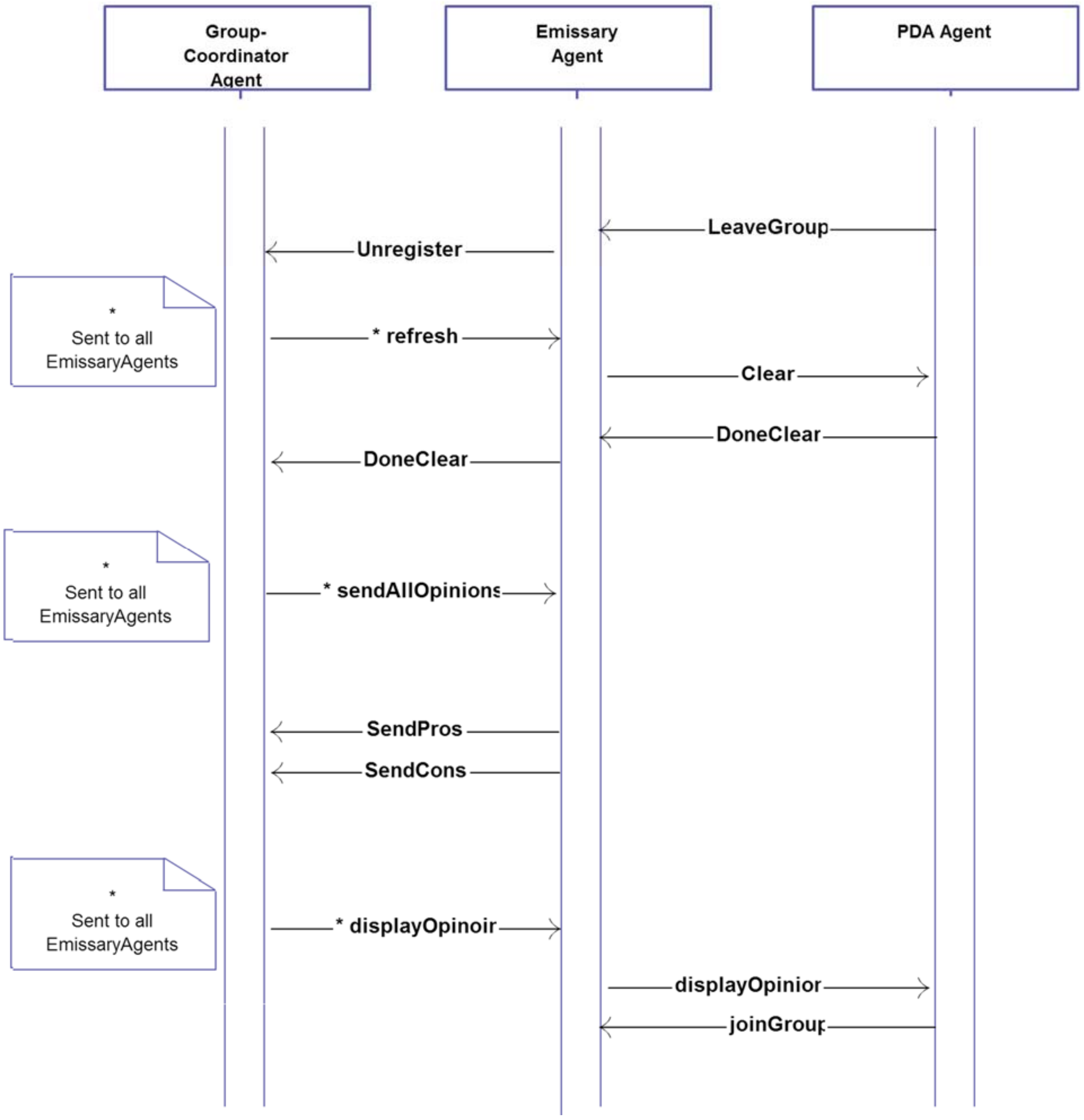
4.2 Sequence Diagram showing the process in which an agent adds an opinion (Pro or Con)

➢ Deleting an opinion:



4.3 Sequence Diagram showing the process in which an agent removes an opinion (Pro or Con)

➢ Leaving a Group (and joining another one)



4.4 Sequence Diagram showing how an agent leaves a group.

As seen in these diagrams, with each modification, all the agents will be updated with the most recent state of the system.

5 CONCLUSIONS:

This document presented the tATAmI platform detailing its structure and components, how scenarios are created and JADE, which underpins this platform. Also, it went through the semantics and syntax of S-CLAIM and its use of Java functions. Then, an Example Scenario of five agents playing the “BOLTZ!” game was described, along with its implementation showing how an agent class written in S-CLAIM would be like, and how agents would be described in an XML file.

6 FUTURE WORK:

Since the tATAmi platform had recently been re-structured, the application must be ported fully to it making the necessary changes. Also, more actions could be added to the Agents making the application richer.

Also, an editor that allows developers to write S-CLAIM code easily and elegantly would be a very nice addition to the project. It could be developed as an Eclipse plug-in and offer some of the following features:

- Open the specific type of Agent file (*.adf2).
- Color and suggest auto-completion for S-CLAIM keywords.
- Find the existing variables and method in the *.java/xml files in the same project and also color and suggest auto-completion for them while typing S-CLAIM code.
- A kind of a "run" command/visual-button for the file (of the specific type) to check if it follows some specific syntactic rules or not.
- Showing errors and the line numbers in which they occurred.

REFERENCES

Valentina Baljak,Marius Tudor Benea,Amal El Fallah Seghrouchni,Cédric Herpson,Shinichi Honiden,Thi Thuy Nga Nguyen,Andrei Olaru,Ryo Shimizu,Kenji Tei,Susumu Toriumi. *Procedia Computer Science*. Retrieved from Science Direct: http://www.sciencedirect.com/science/journal/18770509/10/supp/C

Olaru, D. I. (2011). A Context-Aware Multi-Agent System For AmiEnvironments.

Suna, A., & Seghrouchni, E. F. (2004). Programming mobile intelligent agents: An operational semantics. *Web Intelligence and Agent Systems*, pp. 47-67.