A Framework for Integrating Heterogeneous Agent Communication Platforms

Andrei Olaru
and Adina Magda Florea
Department of Computer Science
University Politehnica of Bucharest
313 Splaiul Independentei
060042 Bucharest, Romania
Email: cs@andreiolaru.ro

***Abstract*—[1]When developing multi-agent systems, the initial choice of deployment platform has a long-term impact on the project, as it many times restricts agent architecture, communication protocols, and available services. The goal of this paper is to present the architecture of tATAmI-2.5, a framework that is able to integrate agents deployed using different environments, and communicating using different communication platforms. This framework is based on the tATAmI-2 agent development and deployment framework, which allows agents to be deployed on various communication platforms without modifying the agent code. The details of the proposed architecture are presented, including insights into the bootstrap process and message routing.**

I. Introduction

In the world of multi-agent system applications, the initial choice of development and / or deployment framework for agents is crucial. Several popular agent frameworks exist, such as JADE [1], JIAC [2], Jason [3] (and JaCaMo [4]), Agent Factory [5], and more. However, the choice of one of these frameworks for development (we will discuss each in Section II) has two effects: first, it influences the internal structure of agents, to match with the specific agent architecture used by the framework, potentially imposing some agent-oriented programming (AOP) language; second, it makes it very difficult to come back on the decision. Moreover, it is of significant difficulty to interoperate (i.e. allow the communication between) agents deployed using different frameworks. The goal of the presented framework is to reduce the impact of this initial choice.

In recent years, we have developed, in collaboration with LIP6[2] and under the AmIciTy initiative[3], the tATAmI agent framework for developing agent-based Ambient Intelligence (AmI) applications. In time, tATAmI has progressed from a PC/Android Jade-based platform for deploying AmI applications [6], [7], to the tATAmI-2 version [8] that decouples the agent implementation from the platform offering communication and mobility services, using a component-based design that offers a great deal of flexibility in how a developer is allowed to design the agent.

More precisely, tATAmI-2[4] works with abstractions for the implementations of platforms, agents, and agent components: the *platform* is what offers services such as communication, directory and mobility; agents are the persistent, autonomous entities that communicate and move using the services offered by the platform; agent components (in the case of Composite Agents – see Section III) isolate various functionalities of the agent, such as messaging, visual interface or input/output. The result is that the developer may implement the agent regardless of how communication will be done (e.g. TCP/IP, WebSockets, JMS) and the same code can be used for agents while using different means of communication or directory services. The component-based design means that existing components can be added or removed from the agent without modifying any source code, only configuration files. Of course, a component offering a particular service can be implemented in various ways, without changing any other component as a result; for instance, an agent may switch from a triple-based knowledge base to a graph-based knowledge base by only changing the configuration files.

Another goal of tATAmI-2 was ease of deployment. Once the tATAmI-2 application is started on all machines used in a system execution, it takes only one click to deploy agents and initiate the execution, according to the existing XML configuration file. Monitoring and control of all agents can be done from a single machine.

One obvious evolution of tATAmI-2, since it can deploy agents using various communication platforms, but in different executions, is to be able to deploy agents on various communication platforms simultaneously, but as part of the same *system*. That is, in tATAmI-2.5 we need to be able to have a multi-agent system in which some agents communicate by means of TCP/IP in a LAN, some agents (of which one is also in the LAN) communicate by means of WebSocket connections to a server, and some agents are deployed on a wireless sensor network and use OLSR (Optimized Link State Routing) to connect to a gateway in the LAN.

This framework is motivated by two situations: a developer may desire to integrate in the same system, with minimal changes, agents previously designed and deployed using different deployment environments; or a developer may find it

[1]The final version of this paper is available on IEEE Xplore at http://ieeexplore.ieee.org/document/7426110/.

[2]Laboratoire d'Informatique de Paris 6, University Pierre et Marie Curie.

[3]http://aimas.cs.pub.ro/amicity

[4]The code is open source at https://github.com/tATAmI-Project/tATAmI-PC/tree/tATAmI-2/master .

appropriate to use different types of inter-agent communication in different parts of the system, e.g. a routing method for a Wireless Sensor Network and a different method of routing for agents in a Local Area Network, but in the same larger system.

This paper details the requirements for an architecture that integrates heterogeneous agent communication platforms, presents the design of such an architecture, and presents the changes that are needed in tATAmI-2 in order to reach the tATAmI-2.5 version.

A. Framework Requirements

The requirements for an agent framework that supports agents using different platforms for communication (and other services, such as directory and mobility) are listed below:

- the user should be able to deploy the system with ease just by starting the applications on the various machines (with a minimal number of command line arguments), and agents should be created and deployed automatically in the required configuration;
    - a trade-off is allowed for network connection parameters, that can be either set from the command line at every execution (quick, but repetitive) or in a configuration file (slower, but persistent);
- except for arguments set in the command line, executions should be easily repeated with the same settings across the system;
- the user should be able to monitor the state of the whole system and the logs of all agents in the system from a single machine; this would be the *System Central* machine;
- the implementation of the agents should be ignorant of the network setup and the communication platforms in the system. At most, the agent may need to know, beside the agents it wants to communicate with, the name (identifier) of the platform on which its correspondent is loaded. If the name of the target agent is unique across the system, the identifier of its host platform should be redundant.
- in the particular case of our framework, the architecture and implementation have to be as backwards-compatible with tATAmI-2 as possible, such that a node running tATAmI-2 is able to connect in the same manner to a platform, regardless of the fact that there are more platforms running in the same system.

B. Framework Summary

In order to fulfill the requirements above, we propose the tATAmI-2.5 architecture, which draws upon the architecture of tATAmI-2 [8]. In this architecture, the actual communication within the multi-agent *system* is done by *platforms*, to which agents have uniform access by means of platform-specific *messaging components*. Each agent has a unique *name* in the platform and is *loaded* on at least one platform – for the purpose of simplicity we consider any normal agent is loaded on exactly one platform and has a unique identifier across the platform. Each platform has a unique *identifier* across the system, spans across multiple machines, or *nodes*, and may have a node which is its *center*. There may exist, of course, non-centralized platforms. Any node that is part of at least two platforms is a *frontier* machine, on which special *Frontier Agents* will execute to help message routing. The system has a single *System Management* machine from which the entire system can be controlled and monitored.

When the execution starts, before agents are created, special agents are started on frontier and central machines, and they communicate in order to establish the topology of the network and routing and directory data. Subsequently, when agents from different platforms wish to communicate, the messages are relayed through frontier agents and are carried by the platforms that link them.

The next section discusses several popular agent frameworks and Section III summarizes the features and functionality already offered by tATAmI-2. The proposed solution is detailed in Sections IV to VI. The last section draws the conclusions.

II. Related Agent Frameworks

In this section we will present some other popular agent frameworks [9]. We do not desire to present these platforms in opposition to our proposed architecture, rather to present how we could integrate these platforms with tATAmIso that agents deployed on one platform are able to communicate with agents deployed on another.

JADE [1] was the base of tATAmI-1 and it was an inspiration while creating tATAmI-2. It is a powerful and easy to use agent development and deployment framework that offers agent management, mobility, and communication for agents implemented in Java. Jade has already been integrated with tATAmI-2 by creating an agent wrapper in Jade that contains a Composite Agent.

JIAC [2] is a production-grade framework for developing complex agent systems, on both workstations (JIAC-V) and mobile/embedded devices (microJIAC). It has a focus on industrial applications, by offering features for security, management and scalability. Like Jade, it is written in Java and agents are created and managed programmatically. While we have not attempted this yet, we expect to use techniques similar to integrated Jade agents in order to integrate JIAC agents.

Jason [3] is based on AgentSpeak and is an AOP language with its own special syntax to define goals, conditions, and plans, combining Prolog syntax with its own. Jason does support multiple deployment environments, one of them being Jade. It is probably possible to obtain a certain level of integration without modifying the Jason syntax.

Agent Factory [5], together with its implementation for mobile devices – Agent Factory Micro Edition [10], includes flexibility for implementing agents in various programming languages, but currently contains development kits only for Jason-based implementations. As with Jason, we expect a certain level of integration with tATAmIto be possible.

There are not yet many initiatives that deal with interconnecting different agent platforms. One very recent project

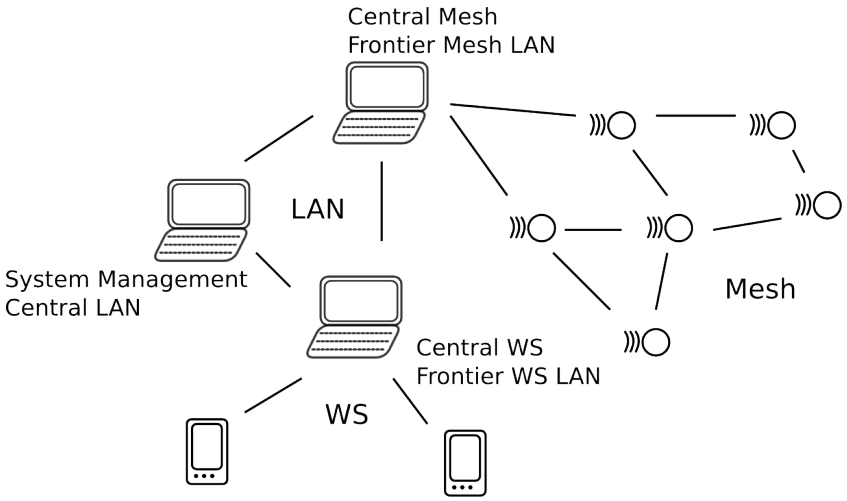

Figure 1. Representation of the scenario in Section I: three nodes are connected by a LAN communication platform; one of the nodes is also the coordinator for a mesh wireless sensor network; another node is also the central server for a WebSocket platform.

is Siebog [11] that attempts to connect previous approaches by the same authors, namely XJAF and Radigost, with a Jason interpreter and a non-axiomatic logic reasoning system (DNARS). XJAF (Extensible Java EE-based Agent Framework) [12] is a framework that harnesses the functionality offered by Java EE enterprise servers for load balancing and robustness. Radigost [13] is a web-based, HTML5 reliant solution. While the approach is very interesting the target of the tATAmI platform is to have as little dependence on systems that are complex to manage (such as enterprise or application servers). It is the user's choice of what method to use to connect agents.

While this paper does not discuss agent programming languages, it is worth mentioning that tATAmI also integrates a parser and execution engine for the S-CLAIM language [6], a descendant of CLAIM (Computational Language for Autonomous, Intelligent and Mobile agents) [14].

III. TATAMI-2: FEATURES AND FUNCTIONALITY

There are a few elements that are essential to the tATAmI-2 architecture[5]. These are the system, the platform, the node, the agent, and the agent component. It is important to mention these elements, as the tATAmI-2.5 architecture, which we present in this paper, uses the same basic elements.

The *system* is the entire system of platforms which host agents. In tATAmI-2.5, two agents are part of the same system if they can communicate between them. The system is responsible for routing messages, potentially across platforms, from one agent to another. An *execution*, or *run*, of the system is the process that starts with the boostrap, contains the entire lifecycle of the agents in the system, and ends after all agents have ended their individual execution.

A *platform* is an infrastructure that enables various features for agents, such as communication or mobility. One such platform can be Jade. Another may be underpinned by WebSocket communication. And so on. The system manages the local instance of a platform through an instance of the `PlatformLoader` interface. The instance has a *name* (unique across the system, and the same for all instances which are part of the same platform) and has methods to *start* and *stop* the platform, *create a node* and *load an agent*. It also can be queried what implementation it recommends for a particular component type. For instance, if an agent must contain a "messaging" component, but does not specify the exact implementation (via class path), the system loads the component recommended by, and therefore specific to, the platform. Abstractly, we can see the platform-specific code as having two parts – one part that is tied to the node (the `PlatformLoader` instance) and one part that is tied to the agent (the `MessagingComponent` instance).

A *node* is a machine that is part of the system. We use the term both for a machine that is part of the system, and for the instance of tATAmI that runs on that machine. On one node there may be one or more platforms running. A node is characterized by its *name*. The name of the node must be unique throughout the system, otherwise platforms containing it may not be able to join the system.

An *agent* is characterized by its *name*. The name of the agent is unique across the platform on which it is loaded. An agent resides on a node and on one or multiple platforms. An agent may move from one node to another executing the same platform, if the platform supports mobility. Any agent that is loaded on a platform must implement the `AgentManager` interface, which has methods for inquiring the agent's name, for *starting* and *stopping* the agent, and for passing a link to the platform to the agent, as an implementation of the marker interface `PlatformLink`. The platform link may be used by platform-specific components to access functionality offered by the platform.

An agent of a particular type (e.g. composite) is loaded by an implementation of the `AgentLoader` interface. An agent loader is able to *pre-load* the agent creation data for an agent, and later, using this creation data, to *load* the agent and return an `AgentManager` instance.

[5]Some of these elements have been renamed in the architecture presented in this paper, so in the description of tATAmI-2 we will use these new names, for the sake of clarity.

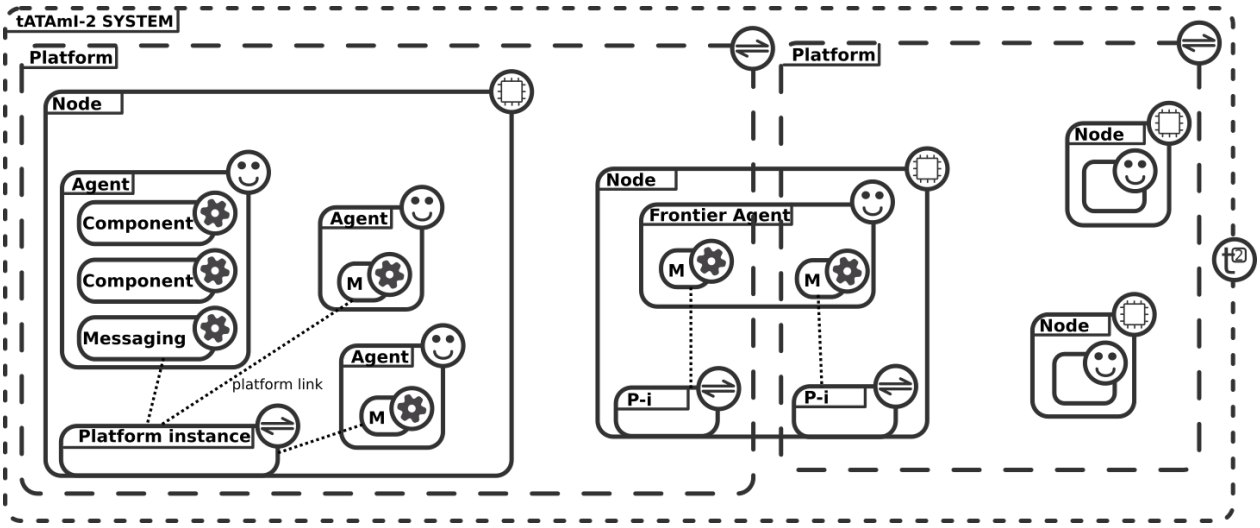

Figure 2. The connection of the various conceptual elements in the tATAmI-2.5 architecture. A tATAmI-2.5 *system* contains several *platforms* that span across multiple *nodes* (machines). An instance of the `PlatformLoader` of each platform runs on each node that is part of the platform. *Agents* run on the nodes and use *messaging components* to communicate, via the platform, to which agent has a *platform link* that the platform-specific messaging component can use. A *Frontier Agent* has one messaging component for each of the platforms that contain its node.

A *composite agent* contains various *components* that offer specific functionality. Components have access to other components and to the platform, identifying them by type or name. For instance, a visualization component is able to contact the messaging component of the agent, if any, to send and receive messages. The messaging component, which is specific to the platform (e.g. agents load a specific class for using messaging on Jade, according to the recommendations of the Jade platform loader), is able to contact the platform in a specific way so as to actually send a message to another agent. Components are also able to post agent events, which the agent disseminates to all other components.

IV. System Architecture

In building an architecture that integrates multiple agent communication platforms into a single system, we have identified some essential challenges, for which we will discuss solutions throughout the paper. These challenges reflect the ones discussed in previous work discussing tATAmI-2 [8]. The challenges are as follows:

- a bootstrap process must be designed such that it is simple to the user to start the system, but all elements of the system are able to access and gather the information they need in order to function normally. This challenge covers accessing servers (if any), contacting the other platforms, being able to contact System Management, and being able to correctly compute routing data for cross-platform messages.
- routing and addressing methods must be designed such that messages sent from one agent can reach any other agent in the system, without the first agent knowing anything else than the second agent's identifier (potentially, the host platform of the second agent may be needed as well).
- means must be designed for the same agent to be able to send and receive messages using two or more communication platforms; more precisely, for the same agent to integrate more than one messaging component.

A. Elements in the Architecture

The proposed architecture is built upon the tATAmI-2 architecture and therefore includes the elements presented in Section III, namely *system*, *platform*, *node*, *agent* and *component* (see also Figure 2). While the framework can work equally well with agents that are not composite, we will generally consider them to be so. Other implementations may use only the platform-recommended messaging components, or may use their own implementation altogether.

As opposed to tATAmI-2, however, the management of the execution is slightly different. The system is controlled and monitored by an entity called *System Management*. It executes on the *System Central* node. Although starting up before any platforms are started, it then loads itself as an agent on one of the platforms running on the node, as specified by the execution configuration. System Management displays a User Interface that allows the user to monitor the system, to initiate the creation of agents, and to start the system run (or execution).

Any node on which a single platform is started is a normal node. Any node on which more platforms are started is a *frontier* node and can be used to route messages between any two of the platforms that execute on the node. A frontier node will start a *Frontier Agent*, which has the particular feature of having one messaging component for each of the platforms that run on the node. It also features a `FrontierComponent` that manages the routing of messages between platforms. The name of a frontier agent is generated as `Frontier<Platform><ID>`, that is, it will contain the identifier of one of the platforms running on the node (any of them) and an identifier that is generated by that platform in such a way that it is unique to the platform, and, as the

identifier of the platform is unique to the system, the result will be a unique name across the system as well.

Some communication platforms (such as WebSocket, Jade and many others) use a client-server architecture. In such platforms, the server must have a public IP address that is known by all the clients. For such platforms, we will consider a *Central** agent, which runs on the server machine of the platform. The name of the agent depends on the identifier of the platform, e.g. for platform LAN-2 the name of the agent will be `CentralLAN-2`.

B. The System Graph

The *System Graph* is a structure that represents how the nodes (especially frontier and central) of different platforms are connected. It is used to compute routing information. For instance, an agent that communicates through WebSockets must send its message to a Frontier Agent that will use the LAN to carry the message to another Frontier Agent that is part of the wireless sensor network.

While routing information (System Graph) could be computed in a completely distributed manner [15], we choose to do it semi-centralized: the routing information is computed by System Management, which disseminates it to Central* agents, which in turn disseminate it to Frontier Agents and smart nodes in their platforms. We choose the centralized approach for two reasons: the intention of the tATAmI approach is to benefit from a centralized manner of controlling and visualizing the system, therefor there will always exist a central machine for that purpose; second, a feature of tATAmI is that it boots quickly – a distributed approach for computing routing information would converge in a longer time. Only the computation of the System Graph is centralized, and happens at bootstrap. After the routing data is disseminated to the platforms, routing is done in a distributed manner (see also Section VI-A).

We introduce here the notion of "silly" and "smart" nodes. A "silly" node sends cross-platform messages to its Central agent, which will route it appropriately. A "smart" node receives routing information from its platform and uses it to send cross-platform messages directly to the appropriate Frontier Agent. The distinction between the two types of nodes is done by the implementation of the platform, and also by the implementation running on the node itself. In fact, "silly" nodes exist for the purpose of compatibility with tATAmI-2 instances.

V. The Bootstrap Process

The first challenge listed in Section IV relates to the bootstrap process. The target is that, after the user has started tATAmI on all nodes, nodes will be ready to route any cross-platform messages appropriately.

While designing this process (and also the set of command-line arguments presented in the next section), we have taken into consideration the following scenarios that may occur:

- a local execution where a communication platform and all agents run on the same machine;
- a central machine and several nodes run an application in which all agents use the same communication platform (the situation in tATAmI-2);
- the central node is the center of two (or more) communication platforms, and nodes in the system use one of those platforms for communication;
- two (or more) platforms exist in the system, and the center of each platform is in the network of the other;
- two platforms exist, and there is at least one node that is part of both platforms;
- several platforms exist, and there are at least two platforms for which there is no node that is part of both of them.

The bootstrap process we have designed proceeds as follows:

1) central machines for all platforms are started, and Central* agents start as well. The order is not important;
2) other nodes are started; on nodes that are part of multiple platforms, Frontier Agents are started; each Frontier Agent has a list of the platforms it is part of;
3) Frontier Agents report the elements of the frontier to the Central* agents for each of the platforms they are part of; if it is made available by the platform, some indication of distance to the network center is also sent, to help compute the System Graph;
4) System Management disseminates the name of the platform where System Management can be found to all Frontier Agents in the platform;
5) Frontier Agents disseminate this information to their other Central* agents, which disseminate the information, in turn, to other frontiers, so that all Central* agents know on which platform runs System Management;
6) Central* agents send back local network information (frontier data) to System Management;
7) System Management computes the complete system graph and sends it to Central* agents;
8) Central* agents send routing data to Frontier Agents and individual "smart" nodes;
9) as more frontier nodes are started, Central* agents send updated network information to System Management, which updates the complete system graph; updates are only sent at larger intervals (e.g. 500ms, 1s, 2s);
10) the connected platforms are shown in the System Management UI so that the user can evaluate if enough of the system is visible.

A special case arises when a Central* agent is in the network of another platform. In this case, it will present itself as having the capabilities of Frontier Agents as well, and it can be contacted directly to route messages to its platform.

In the protocol above an error can arise if two platforms with the same identifier join the system. The second platform to do so (and any subsequent one) will receive a message that it can't be integrated in the system and all frontier agents that link it to the system will be informed to drop any connection to it. This will lead to a notification to the user, but the platform may continue to run on its own.

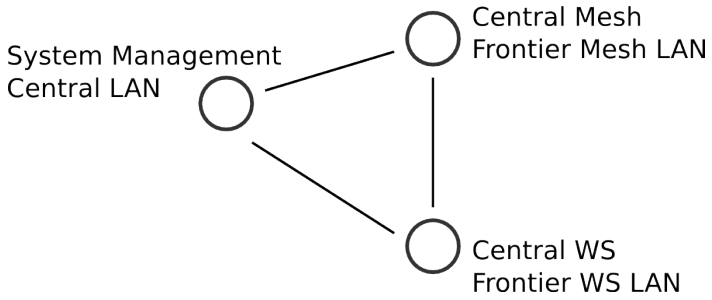

Figure 3. The system graph corresponding to the scenario presented in Section I and in Figure 1.

The initial process of computing routing data can be summarized in 4 phases:

1) on each platform the Central* agent gathers information from the Frontier Agents in the platform;
2) System Management disseminates its platform to everybody else;
3) all Central* agents send information about their platform to System Management;
4) System Management sends the System Graph to all the platforms, which disseminate it internally.

Phase 1 can happen simultaneously with phases 2-4. In the entire process, messages must go from the center of the system to the platforms, then back, and then back again.

A. Booting Quickly

In order to ensure an effortless experience for the user, starting up the system should be as easy as possible. Since the application must be started on each of the nodes in the system (except when some other remote solution exists), the settings for each instance should be as simple as possible, using simple commands for simple applications. It is important to note that, given the variety of possible scenarios (some of which have been enumerated at the beginning of this section), designing a minimal set of arguments may be challenging.

To start the application from the command line, space-separated arguments are used.

The scenario file (see the next Section) must be specified as the first argument. Beside specifying node and agent data, it may also specify network connection data.

The rest of the argument list can be split into sub-lists, such that each list starts with an argument beginning with a dash. Each sub-list is considered as being a specific setting. The following settings are supported:

- `-iscentral [main-platform-id]`
  Specifies that the current node is the node where System Management will run; if followed by an argument in the same list, the argument specifies the identifier of the platform that will load the System Management agent; otherwise, an arbitrary platform will be used, if more are started on this node;
- `-center IP port other...`
  If the default communication platform is used (or it is specified in the scenario file), this setting gives the connection information for the Central machine of the platform; it is not mandatory that an IP is used, as the settings will be passed directly to the `PlatformLoader` instance;
- `-here node IP port other...`
  If the default communication platform is used (or it is specified in the scenario file), this setting indicates connection information related to the current node. It must start with the name of the current node, optionally followed by local IP address and port, and potentially other settings;
- `-platformID type settings...`
  This setting specifies a platform to be started on the current node. The settings of the platform may contain the address and port of its central node (all settings are passed to the `PlatformLoader` instance).
- `-platformType settings...`
  This form is an *alternative* to the setting above, where the user only specifies the type of the platform, and its identifier will be the same as the type.
- `-wh width height`
  This setting specifies the width and the height of the surface on the screen to be used for the UI of the system and the agents. Other local settings may be created in a similar fashion.

Setting the platform using the arguments described above, it is very easy for a user to start tATAmI. For example, starting a (non-central) node by specifying only the local and remote IP (as may be needed when using Jade) is done with the command

```
tATAmI scenario.xml -center <IP1> -here <IP2>
```

Similarly, when working without a local scenario file and using a platform that does not use the local IP (such as WebSockets), one may run tATAmI like this:

```
tATAmI -websockets -center <IP1> -here Node2
```

B. Booting in a Repeatable Manner

While it is very good to have command line arguments that enable the user to specify settings for an execution at a particular moment of time, repeatability of experiments can only be ensured through persistent files describing the execution.

We have already discussed and presented the concepts behind scenario files in the past [8]. A scenario file is an XML

file that completely describes the execution of the local node and potentially the agents to deploy on other nodes.

In order to adapt scenario files to working with multiple platforms, it is necessary to be able to insert in the scenario multiple nodes describing the platforms to start locally. Each platform-central node will have several parameters which are key-value pairs. In these pairs the user is able to configure the platform completely: specify its name, type, central connection settings, local connection settings, and so on.

VI. Platform Services across a Multi-Platform System

While creating the tATAmI-2.5 architecture so as to support multiple communication platforms deployed simultaneously, messaging was one of the central issues. More precisely, two challenges were identified:

- how to be able to deliver a message sent from an agent using one communication platform to an agent using a different one?
- how to handle messaging in such a way that agents can be ignorant of what platform(s) is/are used in the system, including of the platform that the agent itself is loaded on.

For the second challenge, part of the problem has already been solved in tATAmI-2. A component (or some other part, if the agent is not a Composite Agent) can send a message by calling a method in the agent's `MessagingComponent` (which may be platform-specific, but always has the same API), and specifying the source endpoint, the target endpoint, and the content of the message. Endpoints are formed of the address of the agent (which is used only inside the messaging component, and may therefore be platform-specific) and optionally an 'internal path' that indicates the component or the functionality of the agent which is addressed. The address of the agent is ideally the agent's name.

If the target agent is in a different platform than the sender agent, there are two possibilities: either there is (or may be) another agent with the same address somewhere else in the system, or there is only one agent in the system with that name. In the first case, the `MessagingComponent` API must be modified in order to support specifying the identifier of the target platform.

In the second case, it is the system that must determine the platform to send the message to. This can be done by keeping a list of agents in the System Management agent. This also ensures that it is clear if an agent name is used just once or more times across the system. For increased performance, platform-central nodes my also keep (partial) lists of agent names (e.g. caches).

A. Message Routing

As presented in Section V, routing information is created as the system starts, before any user-created agents are started. Routing data consists of the System Graph (see Section IV-B), which is the graph of direct links between central nodes and frontier nodes. If a frontier node is simultaneously a central node, a single node will be used in the graph that will contain both these features. System Management is able to process the graph in order to remove frontier nodes that are not necessary.

Routing data is contained in central, frontier, and "smart" nodes. According to the routing data, agents are able to pre-compute which is the frontier agent that is best to contact for each other platform in the system. When a message must be routed for that platform, the next hop is already determined.

For backwards compatibility, we will consider three cases for platforms that join the system:

- "silent" platforms are platforms that are deployed using tATAmI-2, but contain Frontier Agents, but whose architecture remains unchanged. Thanks to Frontier Agents, they will receive messages from the rest of the system, but they are not adapted to send messages to other platforms (they don't understand the concept);
- "silly" platforms use a tATAmI-2-based implementation, which don't use the System Graph. All messages (or just messages to unknown destinations) go to the Central* agent, which sends them in turn to the appropriate Frontier Agent. For a minimal number of modifications, all messages to one platform will go to the same Frontier Agent (no smart routing);
- "smart" platforms are able to route messages to various Frontier Agents, using the System Graph received from System Management. These platforms may contain "smart" nodes, which are also able to understand the System Graph and use it to route messages without the help of the Central* agent.

There is one more issue that must be addressed in modifying tATAmI-2 to support multiple messaging platforms. Namely, each frontier agent must be able to communicate using any of two or more platforms, at any time. While this is possible to implement in tATAmI-2 using multiple agents for each frontier node (one agent per platform) and having the agents communicate between them locally "out of platform", this is cumbersome and unnecessary. tATAmI-2 can be modified to support multiple messaging components in the same agent. In this case, other components will either need to specify which messaging component they want to use, or use one of the available components by default.

VII. Conclusion and Future Work

Being able to deploy a multi-agent system in which different sets of agents communicate using different communication platforms can be a decisive advantage an agent development and deployment. With such a framework, the user is able to choose the most appropriate platform for every part of the system, or is able to integrate, within the same system, agents that are deployed using different communication platforms.

This paper presents the architecture and design details for the tATAmI-2.5 framework, which allows the integration of different agent communication platforms. Insights are given into how the system should be started, into the bootstrap process, and into the routing mechanism.

As future work it is required that this architecture is realized, integrated into the existing implementation, and improved

in order to be deployable for research applications, at the beginning, and later for production applications.

ACKNOWLEDGMENT

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

REFERENCES

[1] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with JADE," *Intelligent Agents VII Agent Theories Architectures and Languages*, pp. 42–47, 2001.

[2] M. Lützenberger, T. Küster, T. Konnerth, A. Thiele, N. Masuch, A. Heßler, J. Keiser, M. Burkhardt, S. Kaiser, and S. Albayrak, "JIAC V: A MAS framework for industrial applications," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1189–1190.

[3] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, vol. 8.

[4] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013.

[5] S. Russell, H. Jordan, G. M. O'Hare, and R. W. Collier, "Agent factory: a framework for prototyping logic-based AOP languages," in *Multiagent System Technologies*. Springer, 2011, pp. 125–136.

[6] V. Baljak, M. T. Benea, A. El Fallah Seghrouchni, C. Herpson, S. Honiden, T. T. N. Nguyen, A. Olaru, R. Shimizu, K. Tei, and S. Toriumi, "S-CLAIM: An agent-based programming language for AmI, a smart-room case study," in *Proceedings of ANT 2012, The 3rd International Conference on Ambient Systems, Networks and Technologies, August 27-29, Niagara Falls, Ontario, Canada*, ser. Procedia Computer Science, vol. 10. Elsevier, 2012, pp. 30–37. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050912003651

[7] A. Olaru, M.-T. Benea, A. El Fallah Seghrouchni, and A. M. Florea, "tATAmI: A platform for the development and deployment of agent-based ami applications," in *Proceedings of ANT-2015, the 6th International Conference on Ambient Systems, Networks and Technologies, June 2-5, London, United Kingdom*, ser. Procedia Computer Science, E. Shakshuki, Ed., vol. 52. Elsevier, June 2015, pp. 476–483. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915008182

[8] A. Olaru, "tATAmI-2 – a flexible framework for modular agents," in *Proceedings of AgTAmI 2015, the International Workshop on Agent Technology for Ambient Intelligence, the 20th International Conference on Control Systems and Computer Science, May 27-29, Bucharest, Romania*, I. Dumitrache, A. M. Florea, F. Pop, and A. Dumitrascu, Eds., vol. 2. IEEE Computer Society, May 2015, pp. 703–710. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7168503

[9] C. Bădică, Z. Budimac, H.-D. Burkhard, and M. Ivanović, "Software agents: Languages, tools, platforms," *Computer Science and Information Systems*, vol. 8, no. 2, pp. 255–298, 2011.

[10] C. Muldoon, G. M. P. O'Hare, R. W. Collier, and M. J. O'Grady, "Agent factory micro edition: A framework for ambient applications," in *Proceedings of ICCS 2006, 6th International Conference on Computational Science, Reading, UK, May 28-31*, ser. Lecture Notes in Computer Science, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3993. Springer, 2006, pp. 727–734.

[11] "Siebog: An enterprise-scale multiagent middleware," visited august 2015. [Online]. Available: https://github.com/gcvt/siebog

[12] D. Mitrović, M. Ivanović, M. Vidaković, and Z. Budimac, "Extensible java ee-based agent framework in clustered environments," in *Multiagent System Technologies*. Springer, 2014, pp. 202–215.

[13] D. Mitrović, M. Ivanović, Z. Budimac, and M. Vidaković, "Radigost: Interoperable web-based multi-agent platform," *Journal of Systems and Software*, vol. 90, pp. 167–178, 2014.

[14] A. Suna and A. El Fallah-Seghrouchni, "A mobile agents platform: architecture, mobility and security elements," in *Programming Multi-Agent Systems*. Springer, 2005, pp. 126–146.

[15] G. R. Andrews, *Concurrent programming: principles and practice*. Benjamin/Cummings Publishing Company, 1991.