D3.1
Electronic institutions for community building (v1)

<table>
<thead>
<tr>
<th>Core authors</th>
<th>Dave de Jonge, Nardine Osman, Bruno Rosell i Gui, Carles Sierra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainers/Editors</td>
<td>Dave de Jonge</td>
</tr>
<tr>
<td>PRAISE research group leaders</td>
<td>Mark d’Inverno, François Pachet, Carles Sierra, Luc Steels</td>
</tr>
<tr>
<td>PRAISE project leader</td>
<td>Carles Sierra</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grant agreement no.</th>
<th>318770</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>PRAISE - Practice and peRformance Analysis Inspiring Social Education</td>
</tr>
<tr>
<td>Date</td>
<td>August 30, 2013</td>
</tr>
<tr>
<td>Distribution</td>
<td>Public</td>
</tr>
</tbody>
</table>
Abstract

This deliverable describes the work realised in the first year of the PRAISE project regarding the development of Electronic Institutions as basic framework for building social e-learning communities. We explain the modifications we have made to the existing Electronic Institutions framework and the new tools we have had to add to it in order to make it suitable for the development of learning communities. These modifications include a peer to peer communication layer, a type of agent called a Device Manager that is responsible for the division of work load among the devices in the network, and a type of agent called a Rest Governor that forms a layer between the user devices and the peer to peer network. It converts http requests from these users into messages in the electronic institution. Furthermore we have added a javascript library that enables human users to connect to an electronic institution through a web browser and another javascript library that automatically generates a user interface so that human users can perform actions in the institution. We describe a model that enables us to integrate existing web services with any instance of an electronic institution and we describe the specification of an electronic institution that we have designed as an initial test case that will form the basis for further development of the PRAISE framework.

Keyword list: Electronic institutions, community building, human interface, web services, community rules
## Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>04.12.12</td>
<td>Bruno Rosell i Gui</td>
<td>Creation</td>
</tr>
<tr>
<td>0.2</td>
<td>30.07.13</td>
<td>Dave de Jonge, Nardine Osman, Bruno Rosell i Gui, Carles Sierra</td>
<td>Fill in for release candidate</td>
</tr>
<tr>
<td>0.3</td>
<td>04.09.13</td>
<td>Dave de Jonge, Nardine Osman, Carles Sierra</td>
<td>Release check in</td>
</tr>
<tr>
<td>0.4</td>
<td>12.09.13</td>
<td>Dave de Jonge</td>
<td>Added a longer executive summary, a longer abstract, added a discussion session at the ends of chapters 4 and 5, and added a small introduction to both these chapter to relate them more clearly to the project.</td>
</tr>
<tr>
<td>1.0</td>
<td>13.09.13</td>
<td>Dave de Jonge, Carles Sierra</td>
<td>Public release.</td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable describes the work done by the PRAISE staff during the first year to provide the tools that allow the creation of social learning communities.

An Electronic Institution is a tool that allows software agents to come together and interact according to pre-defined protocols. It makes sure that the norms of the institution are enforced upon the agents and thus prevents them from misbehaving. A framework called EIDE that provides the necessary tools for the development and execution of electronic institutions has been under development for more than 15 years. Our goal is to build user communities for music learning on top of this framework.

Users of the PRAISE platform will be able to design their own communities with their own norms in the form of electronic institutions. The users can publish the specifications of these institutions online so that other users may choose to start a new instance of such an institution, or join already running instances.

In this deliverable we describe a new version of EIDE that allows us to run electronic institutions over a peer-to-peer network rather than on a server. This has several advantages, such as better scalability when users stream large audio files over the network. Furthermore, it removes the burden of maintaining a server and provides more privacy as users do not need to upload their files to a central server.

We have implemented the p2p version of EIDE by adding a new type of agent to the framework called Device Manager. Each device in the network will have its own Device Manager, and the collection of all Device Managers are together responsible for making sure the components of the electronic institution are divided evenly over the network. Moreover, each Device Manager is responsible for maintaining a local database of electronic institution specifications. Users can search for these specifications by entering a search query that looks for specific key words. The Device Managers make sure that the query is broadcast over the network and return those specifications from their local databases that match the search query.

In order to make communication between laptops, smartphones, tablets and other devices possible over a peer to peer electronic institution we have created another type of agent called the Rest Governor. This agent forms a layer between the user and the peer to peer electronic institution, and makes sure that http requests from the device of the user are translated into messages that have meaning within the electronic institution.

Another tool we have added to the EIDE framework, is a pair of javascript libraries that allow human users to interact in an electronic institution through a web browser. The first of these libraries makes the connection between the web browser and an
agent that represents the user in the electronic institution. It provides all state information of the institution to the web browser and communicates all the user actions to the agent, which will then translate those user-actions into actions executed inside the electronic institution.

The other library uses the information from the first library to generate a graphic user interface. This automatically generated user interface is entirely general in the sense that it is independent of any institution. It displays all relevant information of the institution and allows the user to do anything that he or she is allowed to do by the institution. The advantage of this is that one does not have to design a new user interface for every newly designed institution, which is especially useful for testing purposes when the institution specification is still under development. This library can however be replaced easily by a new library if one wants to have a custom designed user interface.

Furthermore, we describe a general method for integrating web services with electronic institutions. This allows us to use the electronic institution to control user access to the webservice. Especially when a web service involves uploading large files, for example in the case of an audio analysis tool, it is important that users do not use it too often, as this would overload the network. Since the EIDE framework does not support sending large files we needed to design a new model that on one hand allows the user to send large files, outside the electronic institution, while on the other hand the electronic institution still remains in control of this file transfer.

Finally, we describe an example of an electronic institution that we have designed as the initial test case to test the new tools described in this deliverable. It is meant as a starting point for further development of the Music Circle website.

This electronic institution defines a protocol for a student to share and discuss his play with a tutor and other students. Furthermore the institution contains two scenes for uploading an audio recording to the Music Circle website and for requesting an analysis of the music by an automatic audio analysis tool respectively.
Contents

1 Introduction .................................................. 1
   1.1 Electronic institutions .................................. 1
   1.2 Electronic Institution Development Environment ............. 5
   1.3 AMELI .................................................. 8

2 Peer to Peer Electronic Institutions ............................ 9
   2.1 Electronic Institutions Repository .......................... 11
      2.1.1 Publishing an Electronic Institution Specification .......... 11
      2.1.2 Removing an Electronic Institution Specification .......... 13
      2.1.3 Searching for an Electronic Institution Specification .......... 13
   2.2 Peer to Peer Network Management ........................... 14
   2.3 Dynamic AMELI .......................................... 15
   2.4 Future work ............................................. 16

3 REST Governor ................................................ 16
   3.1 Data Models ............................................. 18
      3.1.1 Agent Attribute ....................................... 18
      3.1.2 Conversation Attribute .................................. 18
      3.1.3 Movement ............................................. 19
      3.1.4 Create Node Object ..................................... 19
      3.1.5 Way attribute ......................................... 21
      3.1.6 Data Type Instance ..................................... 22
      3.1.7 Function instance ....................................... 23
      3.1.8 Variable Instance ....................................... 23
      3.1.9 Scene Message ......................................... 23
      3.1.10Reason ................................................ 27
   3.2 Rest Actions .............................................. 27
      3.2.1 Registering a Governor .................................... 27
      3.2.2 Logging Out ............................................ 28
      3.2.3 Searching for an Electronic Institution ..................... 29
      3.2.4 Registering an Electronic Institution Specification .......... 32
      3.2.5 Getting Information of a Registered Electronic Institution Specification .............................................. 32
      3.2.6 Searching for all Running Electronic Institutions .......... 33
      3.2.7 Starting a New Electronic Institution ...................... 33
      3.2.8 Enter to Participate in an Electronic Institution .......... 34
      3.2.9 Obtaining Information about All Electronic Institutions the Agent is Participating in ............................. 35
      3.2.10 Obtaining Information of an Electronic Institution Conversation .................................................. 35
      3.2.11 Getting information about the Possible Movements ............ 37
      3.2.12 Moving to Another Conversation ........................... 38
      3.2.13 Getting Information of the Possible Messages to Say .......... 39
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Auction house performative structure example</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Institutions in the sense proposed by North “… set of artificial constraints</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>EIDE framework scheme</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>A screen shot of the Islander editor</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>Scheme of the social layer</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Diagram of peer to peer Electronic Institutions components</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>UML diagram of the ExternalAgent class</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>Diagram of the interaction of the REST Governor with the peer to peer</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Search electronic institution specifications form</td>
<td>43</td>
</tr>
<tr>
<td>11</td>
<td>Electronic institution information view</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>Join to electronic institution form</td>
<td>44</td>
</tr>
<tr>
<td>13</td>
<td>Form to move to another scene</td>
<td>45</td>
</tr>
<tr>
<td>14</td>
<td>Scene protocol view</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Interaction with an electronic institution</td>
<td>47</td>
</tr>
<tr>
<td>16</td>
<td>The components necessary to generate the GUI. Solid arrows indicate exchange of information. The dashed arrow indicates that the GUI is created by the DefaultGuiGenerator.</td>
<td>50</td>
</tr>
<tr>
<td>17</td>
<td>The Default Gui, applied to the Music Circle institution</td>
<td>51</td>
</tr>
<tr>
<td>18</td>
<td>Interaction between an user and a web service, mediated by an EI. Solid arrows represent exchange of messages, dashed arrows represent the creation of a resource. The arrows are numbered in chronological order.</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>The performative structure of the Guided Practice Session Institution</td>
<td>64</td>
</tr>
<tr>
<td>20</td>
<td>The PeerDiscussion protocol</td>
<td>66</td>
</tr>
</tbody>
</table>

**Listings**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example of an agent attribute in JSON format</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Example of a conversation attribute in JSON format</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Example of a movement model in JSON format</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Example of a create node movement model in JSON format</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Example of a way attribute movement model in JSON format</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Example of a data type model in JSON format</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Example of a function instance in JSON format</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>Example of a variable instance in JSON format</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Example of a scene message in JSON format</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>Example of a reason in JSON format</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>Example of a search result in JSON format</td>
<td>31</td>
</tr>
</tbody>
</table>
1 Introduction

In this deliverable we explain the work realised during the first year of the PRAISE project regarding the creation of tools for the creation of social learning communities. This work is focused on satisfying the requirements described in deliverable D1.4 [13], which are mainly the following two: to adapt the current Electronic Institution software to support learning communities, and to facilitate the specification of community norms using Electronic Institutions (EI). To satisfy the first requirement we have adapted and created the following tools:

- Peer-to-Peer electronic institutions, that allows to create dynamic electronic institutions (see section 2).
- Rest governor that provides a HTTP REST layer to interact in an electronic institution (see section 3).
- Automatically generated user interface to helps any human user to interact into an electronic institution (see Section 4).
- EI web services, thus the integration of software or web services to be a part of an electronic institution (see section 5).

For the second requirement we have developed specifications that describe the rules of different scenarios that we have been working on during this first year (see section 6).

Before we explain more about the realized work we will, in the next section, introduce the concept of electronic institutions and the related software that has already been developed previously. Unfortunately we do not have the space here to go into a detailed description of electronic institutions, so we will only give a short overview and introduce the main concept that are necessary to understand for the rest of this deliverable. For a more in-depth discussion of electronic institutions we refer to, for example, [1] [3] [10] [14].

1.1 Electronic institutions

As the complexity of real-world applications increases, particularly with the advent of the Internet, there is a need to incorporate organisational abstractions into computing systems that ease their design, development, and maintenance. Electronic Institutions are at the heart of this approach [7]. Electronic Institutions provide a computational analogue of human organisations in which intelligent agents play different organisational roles and interact to accomplish individual and organisational goals. In this scenario, agent technology helps enterprises reduce their operational costs and speed-up time to market by helping distributed business parties, represented by agents, run smoother and in a better coordinated fashion. Electronic Institutions
appear as the glue that puts together self-interested business parties, coordinating, regulating, and auditing their collaborations (Figure 3).

Just like any human institution, an Electronic Institution is a place where participants come together and interact according to some pre-defined protocol. It makes sure that the norms of the institution are enforced upon its participants and thus prevents them from misbehaving. An Electronic Institution therefore provides the infrastructure in which agents can interact in an autonomous way within the norms of the institution.

A commonly cited example is that of a fish market, with buyers and sellers engaging in interactions aimed at buying and selling fish. They have strict conventions by which fish is traded under strict negotiation protocols. More specifically, the fish market is an auction house that enforces certain conditions on the eligibility of traders, the availability and delivery of goods and the behaviour of participants. While the actual trading makes up the critical part of the fish market, there are other interactions that are also governed by rules. For example, before any trading can be undertaken, sellers must deliver fish to the market, and buyers must register. Furthermore, once a deal has been agreed, the sellers must pay for and collect the fish, and the buyers must collect payment. Beyond this example, many other institutions have similar sets of distinct activities that can be identified, like hotels and universities.

Electronic Institutions have been under development for more than 15 years [1, 3, 10, 14] which has resulted in a large framework consisting of tools for implementing, testing, running and visualizing them. Therefore, before we continue, we need to introduce the basic concepts behind the Electronic Institutions framework.

**Scenes** Just as there are meetings in human institutions in which different people interact, Electronic Institutions have similar structures, known as scenes, to facilitate interactions between agents. Scenes are essentially group meetings, with well-defined communication protocols (see Figure 1) that specify the possible dialogues that agents are allowed to have within these scenes. For example, an electronic fish market may include an auction scene in which buyers compete to purchase fish, with a protocol that involves making bids. There may be many simultaneous instances of such auctions within a fish market, each referred to as a Scene instance.

Scenes within an institution are connected in a network that determines how agents can legally move from one scene to another. This network is called the Performative Structure. In the fish market example a buyer can only enter the auction scene after passing the registration scene.

**Actions** Activities are dialogical as they are achieved via agent interactions composed of non-divisible utterances, that occur at discrete instants of time. These utterances can be modelled as messages that conform to a certain pattern, and physical actions are represented by appropriate messages of this form.
In a fish auction, for example, a buyer commits to buy a box of fish at a certain price by making a bid, while the actual physical action of transferring money from the buyer to the auction house is triggered when the auctioneer declares that the box is sold. In the rest of the document we will therefore use the words ‘action’ and ‘message’ interchangeably.

For each message that can be sent, a number of parameters may be specified by the protocol. When making a bid in an auction for example, the maker of the bid should include the amount of money he bids in the message. The Electronic Institutions framework supports several basic parameter types, such as ‘Integer’, ‘String’ and ‘Boolean’. Apart from these basic types the designer of an institution can define custom types, which are composed of one or more parameters of a basic type.

**Scene Protocols** The interactions between the agents in a scene in an Electronic Institution have to follow a certain protocol. The protocol defines which agent can say what and when within the scene. At each moment during the execution of a protocol, the protocol is in a certain state, depending on the messages that have been said so far. The current state of the protocol determines what kinds of messages each agent can send.

In an auction for example, the protocol may start in a state in which the auctioneer introduces the next item under auction. Participants are not allowed to make any bid yet in this state. Once the auctioneer announces the start of the auction, the state changes to a bidding state, in which the participants are allowed to make their bids.

A protocol is therefore represented as a directed graph in which the nodes are the states of the protocol. Each edge of the graph is labelled with one or more message patterns (see for example Figure 1). A message can only be sent if it
satisfies one of the patterns labelling one of the outgoing arcs from the current state.

**Roles** Scene protocols are not specified in terms of agents, but rather in terms of roles. Every agent plays a specific role that determines which actions it can take at which moment. Roles can be understood as standardized patterns of behaviour that agents, when instantiating a role, must respect. The identification and regulation of roles is considered as part of the formalisation process of any organisation [11]. Any agent within an Electronic Institution is required to adopt some role(s). A major advantage of using roles is that they can be updated without having to update the actions for every agent on an individual basis. Recently, the concept of role is becoming increasingly studied by software engineering researchers and even more recently by researchers in the agents community.

**Constraints** As explained above, the state of the protocol restricts the set of possible actions that can be taken by the agents. Moreover, this set of possible actions can be restricted even further by including constraints in the protocol. Constraints are given as sentences in a first-order logic attached to a message pattern. A message can only be sent if its corresponding constraints are satisfied.

**Governors** Each agent participating in an Electronic Institution has a special agent assigned to it, called its Governor. The Governor of an agent $\alpha$ has control over each message that is being sent by $\alpha$. Whenever $\alpha$ tries to send a message, this message first passes $\alpha$’s Governor, which checks whether the protocol is in the correct state and whether the corresponding constraints are satisfied. If so, the Governor forwards the message to its recipient. If not (for example, because the agent made a bid that is higher than what he can afford), the Governor blocks the message.

**Ontology** As explained above messages can have parameters, which can be of a basic type or of a user-defined type. Each Electronic Institution has an Ontology associated to it that stores the definitions of these user-defined types. Also it stores for each message how many parameters it has and which types those parameters have.

**Dialogical framework** Some aspects of an institution such as the objects of the world and the language employed for communicating are fixed, constituting the context or framework of interaction amongst agents. In a dialogic institution, agents interact through speech acts. Institutions establish the acceptable speech acts by defining the ontology (vocabulary) —the common language to represent the "world"— and the common language for communication and knowledge representation which are bundled in what we call dialogic framework. By sharing a dialogic framework, we enable heterogeneous agents to exchange knowledge with other agents.
Figure 2: Auction house performative structure example

**Transition** When moving from one scene to another an agent always passes a so-called transition. When the agent is at a transition it can choose which role it will play in the scene it is going to, and it can wait for other agents so that they can move to the next scene simultaneously.

**Performative structure** Scenes and transitions are connected, composing a network of scenes which is called the performative structure, and which captures the existing relationships among scenes (see figure 2). The specification of a performative structure contains a description of how the different roles can legally move from scene to scene. Agents within a performative structure may be participating in different scenes, with different roles, at the same time.

**Normative rules** Agent actions in the context of an institution may have consequences that either limit or enlarge its subsequent acting possibilities. Such consequences will impose obligations to the agents and affect their possible paths within the performative structure.

### 1.2 Electronic Institution Development Environment

The design and execution of an Electronic Institution is realised through a framework called The Electronic Institutions Development Environment (EIDE), which is implemented in Java. It consists of a set of tools aimed at supporting the engineering of intelligent distributed applications as Electronic Institutions [8]. EIDE allows for engineering both Electronic Institutions and their participating software agents. Notably, EIDE moves away from machine-oriented views of programming toward organizational-inspired concepts that more closely reflect the way in which we may understand distributed applications. It supports a top-down engineering approach: firstly the organization, secondly the individuals.

318770 August 30, 2013
Figure 3: Institutions in the sense proposed by North “... set of artificial constraints that articulate agent interactions”.

Figure 4 depicts the role of the EIDE tools in the electronic institutions development cycle. This cycle is regarded as an iterative, refining process fully supported by the EIDE tools.

EIDE is composed of:

**ISLANDER** A graphical tool that supports the specification of the rules and protocols in an electronic institution [5] (see Figure 5). It allows you to visually define the scenes, roles, protocols, message patterns, constraints, ontology and other components of the institution. It then converts the visual representation into xml format (the *EI-specification*) that can be read by AMELI.

**AMELI** Software platform to run electronic institutions [8]. Once an electronic institution is specified with ISLANDER it can be executed by AMELI. AMELI comprises of a set of agents that control the execution of scene instances, and is responsible for assigning a Governor to each participating agent. An Electronic Institution can be executed by starting AMELI with an EI-specification. When it is running, agents can join it by requesting entrance to the institution, and, once entered, they can start communicating according to the protocols of the institution.

**aBUILDER** Agent development tool. This is a set of libraries that provide a generic agent layer to participate in the electronic institution and a set of tools that can be used to generate agent skeletons from a specific Electronic Institution specification. These agents could be implemented by third parties, interested in participating in the Electronic Institution for their own reasons. For example, one may design an online fish auction as an Electronic Institution, so that potential buyers can develop their own agents to make bids according to their...
Figure 4: EIDE framework scheme

Figure 5: A screen shot of the Islander editor
personal bidding strategies.

**SIMDEI** Simulation tool to animate and analyse specifications created with ISLANDER prior to the deployment stage.

### 1.3 AMELI

The software that provides the execution of electronic institutions, is called AMELI[8]. It enables agents to act in an electronic institution and controls their behaviour. The main features of AMELI are:

- To provide a way for different agents with different architectures to communicate with one another without any assumption about their respective internal architectures.

- To enforce a protocol of behaviour as specified in an institution specification upon the agents. This means that AMELI makes sure that the agents can only do those actions that the protocol allows them to do.

Figure 6 describes the different parts of AMELI. It consists of three layers: a communication layer, which enables the agents to exchange messages, a layer consisting of the agents that are acting in the institution, and in between a social layer, that controls the behaviour of the participating agents. The social layer consists of the following agents:

**Governor** For each participating agent in an institution there is one Governor. This Governor forms the link between a participating agent and the Electronic Institution. Every time the participating agent wants to make an action in the institution, this is communicated to the governor in the form of a message. The governor determines whether the participating agent is indeed allowed to do that action at that moment and, if so, forwards the message to other agents in the social layer in order for the action to have effect. If the participating however is not allowed to do that action, the governor simply blocks the message and hence prevents the action from being executed. Furthermore, the Governor keeps the status of the agent in the Electronic Institution and its properties.

**SceneManagers** There is a scene manager for each instance of a scene protocol that is active in the electronic institution. This agent controls and keeps the status of a scene protocol of an Electronic Institution. So, it controls when an agent can enter or exit the scene protocol, the scene state and of when an agent can say a message to another.

**TransitionManagers** There is one transition manager for each transition in each active performative structure. This agents controls and synchronizes the movements of the agents between the nodes in a performative structure.
**EInstitutionManager** This agent is unique to each running instance of an Electronic Institution and it keeps the status of the Electronic Institution. It also allows or denies an agent to enter into the Electronic Institution.

The communication between the agents of the social layer is taken care of by a generic communication layer that can be implemented using various technologies. This allows us to distribute the AMELI in different hosts. The communication between the agent and its Governor on the other hand does not take place in this communication layer. The communication channel between them is created when the agent tries to enter into the electronic institution. They could communicate through a socket connection or directly at the code-level if the agent is created as a service in the Electronic Institution.

## 2 Peer to Peer Electronic Institutions

This section describes the conversion of the current AMELI tool (see Section 1.3) to run over a peer to peer network. Furthermore, we have modified the current infrastructure to run dynamic Electronic Institutions that satisfy the requirements described in deliverable D1.4 [13].

Running AMELI over a peer to peer network has several advantages over a centralised system that apply to the PRAISE platform. One of these advantages is that a peer to peer network is able to maintain ‘itself’. That is: we do not need to maintain a server. This is a big advantage as it implies we do not need anybody to take care of
the server maintenance. Moreover, it means that even after the PRAISE project has finished, the platform can continue to exist, regulated by the user community itself.

Another advantage is that a peer to peer network automatically scales with its number of users. Especially when the users are sending large data files and streams such as audio and video over the network, it is very beneficial if we do not have to rely on a server that may not be able to handle large amounts of data.

Furthermore, the use of a peer to peer network allows users to store their files locally, rather than on a server. This can be an advantage for privacy reasons as people may want to stay in control of their files.

Finally, the dynamic structure of the peer to peer networks could help us to create dynamic communities without the necessity to have a central server to coordinate all the users. Currently, the AMELI distribution platform is started using a configuration file in XML format that contains the information necessary for the Electronic Institution to run and the addresses of the nodes that form the peer to peer network. So, if any of this information needs to be modified one has to stop the platform, edit the configuration file with the required changes, and start the platform again. The existing framework therefore does not allow for the dynamic infrastructure that is required by the project.

In order to solve this problem we have created a new infrastructure that is managed by an agent called then Device Manager. The main tasks of this agent are:

- To provide a repository of Electronic Institution specifications (see Section 2.1).
- To create the peer to peer network (see Section 2.2).
- To manage the execution of the Electronic Institution (see Section 2.3).

As can be seen in Figure 7 the agents that participate in the distributed system exchange messages with a Device Manager or a Governor. These messages are serialized as Java objects and sent through a socket connection between the sending agent and the receiving agent.

This kind of communication can quickly become very complex as there are different protocols to take into account. This makes it difficult for programmers to implement agents that will participate in Electronic Institutions. For this reason we have implemented a Java class called ExternalAgent that serves as a base class for any new EI-agent that you may want to implement (see Figure 8). It provides methods to do all the actions that an agent can do in the infrastructure and hides all the complexity of the communication protocols. When at runtime a Governor is created for an agent, this agent receives a GovernorContext object that can be used to do the actions following the rules defined by the Electronic Institution specification.

If, on the other hand, the programmer wants to use another computer language or a more generic mechanism to participate in the infrastructure he or she can make use of the Rest Governor tool, as explained in Section 3.
2.1 Electronic Institutions Repository

One of the necessary components to run an Electronic Institution is the specification that describes the rules that the agents have to follow. The Device Manager provides a repository of Electronic Institution specifications that can be used to start a new instance of an Electronic Institution. This repository is a combination of files stored on a hard drive plus the indexer/search library Apache Lucene. Agents can add a new specification to the repository (see Section 2.1.1), remove a specification from the repository (see Section 2.1.2), or perform a search through all the specifications that are in the repository (see Section 2.1.3).

2.1.1 Publishing an Electronic Institution Specification

When an agent wants to add a new specification to the repository it sends a message to the Device Manager that includes the specification to add. The Device Manager then checks if the specification is valid. If not, it replies to the agent with a message telling that the specification cannot be added. Otherwise, it stores the specification as a file into the working directory with the name of the Electronic Institution plus its hash. Next, it indexes the specification using Apache Lucene to help the search process (see Section 2.1.3). Finally, the Device Manager informs the agent that the specification is added and returns it a HTTP address that the agent

Figure 8: UML diagram of the ExternalAgent class
can use to retrieve the published specification, for example [http://10.10.200.199:46251/AuctionHouse_207674017.xml](http://10.10.200.199:46251/AuctionHouse_207674017.xml). This address is public and can be shared with any other agent that has access to the host where the Device Manager is running.

The directory where the specification files are stored is by default the sub directory `.ACE/ace.ei.EInstitutionDB` in your home directory, but it can be changed when the Device Manager is launched. This directory and its content are not removed when the Device Manager is started or stopped, so the specifications that are added by the agents are maintained in the repository between executions. Also you can add new specification files to this directory but they do not become public until the Device Manager is restarted, because the specifications need to be indexed again.

### 2.1.2 Removing an Electronic Institution Specification

An agent can remove an Electronic Institution from the repository by sending a message to the Device Manager with the HTTP address of the published specification to remove. An agent can only remove those specifications that it has published itself and the Device Manager is the same running agent.

On the other hand if you remove an Electronic Institution specification file from the directory where the Device Manager has stored it, it can not be used to launch a new instance of this institution and it is no longer public, but it will continue to appear in the results of search actions because the file is not removed from the Apache Lucene indices. Under those circumstances one should not remove a specification from the directory unless the Device Manager is stopped.

### 2.1.3 Searching for an Electronic Institution Specification

Every time that the Device Manager is started or an Electronic Institution specification is added or removed from the repository, the available specifications are indexed using the Apache Lucene library. This software creates a database of documents that contain text fields that are used to index the documents and to retrieve them in the search process. In our case each specification in the repository is created as a document with the following fields:

- **name**: The Electronic Institution name.
- **description**: The Electronic Institution description.
- **uri**: The HTTP address in which the specification is published.
- **keyword**: This field contains list of keywords defined for the Electronic Institution. The keywords are defined in the description as words separated by commas, started by the text `{@keywords` and finished by `}`. For example: `{@keywords auction,market,electricity}`.
**input_role** This field contains the name of a role that an agent can have when trying to enter the Electronic Institution. The document contains one field for every role that can possibly be assumed by an agent when entering the institution.

When an agent searches for an Electronic Institution specification it sends a message to the *Device Manager* with the *Apache Lucene* query that describes the specification it is looking for. The *Device Manager* then searches in its internal data base for the specifications that match the query and informs the agent if it found any. Next, it asks the other *Device Managers* that are in the peer to peer network to search in their internal repositories if there exists any specification that satisfies the query.

The *Apache Lucene* query is a string with the name of the field of interest plus and the pattern that has to be matched. If, for example, one wants to search for Electronic Institutions that have a name that starts with ‘Auction’ and that contain a role named ‘guest’, the query will be `name:Auction* AND initial_role:guest`. *Apache Lucene* provides many features that can be used in the search engine [9], but the main ones are:

- **Wildcards:** you can use ‘?’ to match any character or ‘*’ to match multiple characters. For example the word ‘te?t’ will match text or test, and the word ‘test*’ will match test, tests or tester.

- **Fuzzy Searches:** you can use ‘ ’ at the end of a word to specify that you are searching for anything similar. For example the term ‘roam ’ will match foam or roam.

- **Range Searches:** this allows matching all the values that are in a certain range. This range can include or exclude the limits using ‘[’, ‘]’ or ‘{’, ‘}’ respectively. For example ‘{Aida TO Carmen}’ matches all the words between Aida and Carmen excluding these two values themselves. To include them one needs to specify the pattern ‘[Aida TO Carmen]’.

- **Boolean operators:** this allows grouping of patterns by using the boolean operators as **AND**, **OR** or **NOT**. For example ‘roam AND -roam?’ will match foam but not roam.

### 2.2 Peer to Peer Network Management

To simplify the management of the peer to peer network that the *Device Manager* uses for communication with other *Device Managers*, or to launch an instance of an Electronic Institution (see Figure 7), we have decided to use the the *Freepastry* library. This library creates a *Pastry* peer to peer network [4, 2, 12], and provides some useful features such as the routing of messages, or the possibility to create broadcast messages.

---

2 http://www.freepastry.org/FreePastry/
On the other hand, Freepastry does not have any feature for the discovery of new nodes that can be added to the peer to peer network. Therefore, we have created a simple discovery mechanism using UDP packet broadcasting. This process begins when a new Device Manager is started. This agent then opens a socket to listen for UDP requests and sends a broadcast message over the local network with the P2P node information. When other Device Managers receive the broadcast they use the received information to add the node information to their P2P table and reply with their information. After that the original Device Manager uses the received information to update its internal P2P table.

2.3 Dynamic AMELI

Another feature of the Device Manager is the creation of Electronic Institution instances dynamically. That is: it is capable of creating a running AMELI infrastructure (see Section 1.3) where the agents can interact following the protocols defined in a given Electronic Institution specification without the necessity of creating a configuration file. The creation of this Electronic Institution is divided in two stages: in the first stage the agents that form the social layer are created, while in the second stage the Governors of the participating agents are created.

The launch of a new Electronic Institution instance starts with an agent requesting a Device Manager to create the infrastructure with a given EI-specification from the repository (see Section 2.1). Next, the Device Manager and the other Device Managers that are in the P2P network (see Figure 7) cooperatively start all the components necessary for a running Electronic Institution instance. This allows the distribution of workload between the host that form the P2P network. Thus, the agents that form part of the social layer (EIManager, SceneManager, . . .) are created in the different nodes following a round robin algorithm and they use the same P2P network created by the Device Managers to communicate between one another. As soon as all the necessary components have been started the agent is informed that the institution is ready. The next step for the agent is to request the Device Manager to create a Governor. Once this Governor is created it opens a socket server where it waits for the requests from the agent to send messages in the Electronic Institution. It also uses the same socket to send the agent information on the Electronic Institution state or messages that it receives from other agents that are participating in the same scenes.

Instead of launching a new instance of an Electronic Institution, one can also join one that is already running. In this case the agent would ask the Device Manager for a list of all running instances that execute a given EI-specification. The Device Manager then uses the P2P network to send a broadcast message and receive information about the active instances in any part of the distributed system that are executing the given specification. The Device Manager will then send a list of those running instances back to the agent. After that the agent can request that a Governor be created so that it can participate in one of them.
When a Governor has been created for a participating agent, this Governor can only be killed after the agent has left the Electronic Institution.

### 2.4 Future work

Now we have the basic infrastructure for a peer to peer infrastructure, but we have to improve the creation of the P2P network and also we have to add new features as for example the capacity to recovery if one node fails. On the first case we have to provide new mechanisms that allow to distribute the peer to peer network on Internet and not only into a local network. May be we can improve the discovery process for use the Apache ZooKepper technology. This technology allows to publish information on a server that can be used by the peer nodes to coordinate themselves.

On the other hand the main features that we have to add are:

- Provide some mechanism that can restore the infrastructure if one node fails.
- Allow to the nodes to maintain their state between executions.
- Create a better way to create the software agent and link them to their Governors. May be we can use the Java annotations and POJO objects to simplify the way to develop a software agent.

In the next deliverable, D3.2 Electronic Institutions for community building (v2), we will explain more about the taken solutions and these implementations.

### 3 REST Governor

One of the project’s requirements is to allow PRAISE users to interact using various devices such as tablets, computers or mobile phones.

This interaction would be difficult if a user has to use the interaction described in the previous section. For this reason we have added a tool named REST Governor to the peer to peer infrastructure. This tool is the combination of a Jetty web server and an ExternalAgent (see Section 2), that converts web actions (HTTP GET or POST requests) to actions in the Electronic Institution. As can be seen in Figure 9 a device (tablet, laptop,...) sends a HTTP request to the Jetty server demanding to do some action. On the server there is one ExternalAgent running for each user that is logged into the system. The server passes the request on to the ExternalAgent of the corresponding user, which then converts it to the corresponding EI-message. This message is then sent to a device manager or a Governor depending on the type of action to do. The ExternalAgent captures the result of the action and informs it

---

to the user by replying to the HTTP request. Some of the actions require more time to be processed than available to maintain the HTTP connection. So, to prevent the connection from being closed by a time-out, the server replies quickly with a JSON object that contains an identifier that you can use to do polling. Thus, one should use the received identifier in order to repeatedly consult the state of the requested action. As a rule the result of the finished actions are available at least during five minutes, after that, you receive an empty state response.

Every ExternalAgent maintains the state of its corresponding user in the peer to peer infrastructure and captures any change of the state of the Electronic Institution that the user is participating in. Unfortunately, the HTTP connections are bi-directional so the user device needs to actively request this information from the ExternalAgent in order to know what changes have taken place in the institution. For example, in order to be notified of any messages that have been sent to you by other users, you need to continuously poll the ExternalAgent.

In the rest of this section we describe the JSON objects that represent the data that the user interchanges with the system, and the REST actions that the user can do in the infrastructure.
3.1 Data Models

In this section we describe the main JSON objects that a user and the REST Governor can interchange. These objects represent the main types of data that agents usually exchange when participating in an Electronic Institution.

3.1.1 Agent Attribute

This data model is used to identify agents within the institution. It consists of the name of the agent and the role it is playing. An example of this model is displayed in Listing 1. This JSON object has the following attributes:

- **name** *String* value with the name of the agent in the Electronic Institution.
- **role** *String* value with the role that the agent is currently playing.

3.1.2 Conversation Attribute

This data model identifies a scene or transition instance in an Electronic Institution. Thus it identifies a scene instance, transition or performative structure. An example of this model is displayed in Listing 2. This JSON object has the following attributes:

- **identifier** *Integer* value that represents the instance identifier of the conversation.
- **name** *String* value with the name of the performative structure, scene or transition that the conversation represents.
- **type** *String* value with the type of the conversation. Thus the name of the scene type, the name of the performative structure, or the type of transition.
- **performativeStructure** *JSON* object with the description of the represented performative structure, scene or transition. If the conversation is the main performative structure this field is not defined or it is empty.
3.1.3 Movement

This data model describes the movement of an agent that has just exited from a scene, to some other scene, passing a transition. A Govenor can send an object of this type to its corresponding agent to inform it of the possible moves it can make. Also, a participating agent can send an object of this type to its Govenor as a request to make this movement. An example of this model is displayed in Listing 3. This JSON object has the following attributes:

- **transition** JSON object with the conversation attribute that identifies the transition that the movement passes (see paragraph 3.1.2).
- **agents** Array of JSON objects with the attributes that identify the agents that should move to the transition (see paragraph 3.1.1).
- **destinations** Array of JSON objects with way attributes or create node JSON objects. These objects describe the possible conversations to go to from the transition (see paragraphs 3.1.3 and 3.1.4).

3.1.4 Create Node Object

This data model describes how an agent can move from a transition to a scene, with the effect that the agent will enter into a new instance of that scene. An example of this model is displayed in Listing 4. This JSON object has the following attributes:

- **role** String value with the name of the role for the agent that creates the new scene instance.

- **sceneName** String value with the name of the node in the performative structure where the agent is going to create a new scene instance.
Listing 3: Example of a movement model in JSON format

```json
{
  "transition": {
    "type": "And",
    "identifier": -1,
    "performativeStructure": {
      "type": "LessonPS",
      "identifier": -1,
      "name": "LessonPS"
    },
    "name": "toAdmission"
  },
  "agents": [],
  "destinations": [
    {
      "destination": "ONE",
      "role": "guest",
      "sceneName": "Admission",
      "agents": [],
      "types": {
        "0": "AdmissionProtocol"
      }
    }
  ]
}
```
Listing 4: Example of a create node movement model in JSON format

```json
{
    "nodeType": "ClassroomProtocol",
    "destination": "NEW",
    "role": "teacher",
    "sceneName": "Classroom",
    "agents": [],
    "nodeProperties": {
        "allowComments": false,
        "minimumTrust": 0.23,
        "student": "John",
        "matter": "Violin",
        "classTime": 3600000
    }
}
```

**nodeName**  *String* value with the name for the new scene instance.

**destination** *String* value equals to ‘NEW’, because it is used to distinguish from the *Way object* (see paragraph 3.1.5).

**agents** *Array* of *JSON* object with the attributes of the agents that are to move together to the new scene instance (see paragraph 3.1.1).

**nodeProperties** *JSON* object that contains the values with which the properties of the scene protocol should be initialized. The attribute is the property name and the value is the *JSON* representation of the value for the property. These values can be a number, a boolean, a string, a data type (see paragraph 3.1.6), or an array of any of the previous types.

### 3.1.5 Way attribute

This data model describes how an agent can move from a transition to a scene. An example of this model is displayed in Listing 5. This *JSON* object has the following attributes:

**destination** *String* that can take one of the following values: ‘ONE’, ‘SOME’ or ‘ALL’. These values respectively indicate that the agent will enter one instance of the destination scene, some instances, or all currently running instances. The destination type ‘NEW’ is not used by this model because is used by the create node model (see paragraph 3.1.4).
Listing 5: Example of a way attribute movement model in JSON format

```
{
   "destination": "ONE",
   "role": "student",
   "sceneName": "Admission",
   "agents": [],
   "types": {
      "0": "AdmissionProtocol"
   }
}
```

**role** *String* value with name of the role that the agent will play when it enters the scene.

**sceneName** *String* value with the name of the node in the performative structure that the agent will enter.

**agents** *Array of JSON* object with the attributes of the agents that are to move together to the scene instance(s) (see paragraph 3.1.1).

**types** *JSON* object that contains the instances of the scene to enter. The attribute identifier is the instance identifier and the value is the name of the scene type protocol or sub performative structure to enter.

### 3.1.6 Data Type Instance

When an agent sends a message to another agent this message may consist of a number of variables and the values assigned to them. The variables should be of a certain type, indicated at design time. The type of such a variable can be a basic type such as Integer, Float, Boolean and String, but can also be a user defined type, composed of multiple variables. Such a composed type is called a Data Type. The structure of such a data type is described in the ontology of the Electronic Institution. An example of an instantiated Data Type variable is displayed in Listing 6. This JSON object has the following attributes:

**type** *String* value that has to be equal to ‘DATA_TYPE’. It is used to differentiate this object from other objects as the function instance that can be defined in a scene message (see paragraph 3.1.9).

**name** *String* value with the name of this data type as defined in the ontology.

**elements** *Array of JSON* objects with the data type elements. This object contains the attributes name and value. The first one is a string value with the name of
the element in the data type. The second one contains a JSON element with the associated value for the element. It could be a string, number, boolean, a data type or an array of any of the previous types.

3.1.7 Function instance

As explained above a message may contain variables with their associated values. These variables given as parameters of a certain function, which defines the semantic interpretation of the variables. The structure of the function instance is described in the ontology of the Electronic Institution. An example of this model is displayed in Listing 7. This JSON object has the following attributes:

- **type** String value equals to ‘FUNCTION’. It is used to differentiate this object from other objects as the data type values that can be defined in a scene message (see paragraph 3.1.9).
- **name** String value with the name of the function as defined in the ontology.
- **arguments** Array of JSON elements that represent the function arguments. An element could be of type string, number, boolean, a data type, a variable instance, another function instance or an array with any of the previous values.

3.1.8 Variable Instance

When an agent wants to send a message in a scene, this message may contain several variables for which a value must be chosen. A Variable Instance object contains information about the type and the name of such a variable. The agent can receive this information from its Governor. An example of this model is displayed in Listing 8. This JSON object has the following attributes:

- **type** String value equals to ‘VARIABLE’. It is used to differentiate this object from other objects as the data type values that can be defined in a scene message (see paragraph 3.1.9).
- **name** String value with the name of the variable in the scene.
- **varType** String value with the name of the type associated to the variable.

3.1.9 Scene Message

This model describes the message that an agent can exchange with other agents. An example of this model is displayed in Listing 9. It has the following attributes:
Listing 6: Example of a data type model in JSON format

```
{
    "type": "DATA_TYPE",
    "name": "Evaluation",
    "elements": [
        { "name": "student", "value": "John" },
        { "name": "teacher", "value": "Moriarty" },
        { "name": "Lessons", "value": [
            { "type": "DATA_TYPE",
              "name": "LessonEvaluation",
              "elements": [
                { "name": "lessonId", "value": 1 },
                { "name": "position", "value": "Excellent" },
                { "name": "performance", "value": "Average" }
              ]
            },
            { "type": "DATA_TYPE",
              "name": "LessonEvaluation",
              "elements": [
                { "name": "lessonId", "value": 1 },
                { "name": "position", "value": "VeryGood" },
                { "name": "performance", "value": "BelowAverage" }
              ]
            },
            { "type": "DATA_TYPE",
              "name": "LessonEvaluation",
              "elements": [
                { "name": "lessonId", "value": 3 },
                { "name": "position", "value": "Failing" },
                { "name": "performance", "value": "BelowAverage" }
              ]
            }
        ]
    ]
}
```
Listing 7: Example of a function instance in JSON format

```json
{
  "type": "FUNCTION",
  "name": "login",
  "arguments": [
    {
      "type": "VARIABLE",
      "name": "studentName",
      "varType": "String"
    },
    {
      "type": "VARIABLE",
      "name": "email",
      "varType": "String"
    }
  ]
}
```

Listing 8: Example of a variable instance in JSON format

```json
{
  "type": "VARIABLE",
  "name": "studentName",
  "varType": "String"
}
```
Listing 9: Example of a scene message in JSON format

```json
{
  "particle": "request",
  "sender": {
    "role": "student",
    "name": "John"
  },
  "receiver": {
    "role": "teacher",
    "name": "Moriarty"
  },
  "content": {
    "type": "FUNCTION",
    "name": "login",
    "arguments": [
      {
        "type": "VARIABLE",
        "name": "studentName",
        "varType": "String"
      },
      {
        "type": "VARIABLE",
        "name": "email",
        "varType": "String"
      }
    ]
  }
}
```

**particle**  *String* value with the illocutionary particle of the message.

**sender**  *JSON* object with the attributes of the agent that send the message (see 3.1.1).

**receiver**  *JSON* object with the attributes of the agent that has to receive the message (see 3.1.1).

**content**  *JSON* element with the information that is being sent from the sender to the receiver. It could be a string, number, boolean, a data type, a variable instance, a function instance or an array with any of the previous values (see paragraphs 3.1.6, 3.1.7 and 3.1.8).
3.1.10 Reason

If an agent tries to send a message, but the Governor prevents it from doing so, the Governor replies with an object of this type to inform the agent why the message could not be sent. An example of this model is displayed in Listing 10. It has the following attributes:

- **code** Integer value with the error identifier.
- **msg** String value with a human readable message that explains why the message could not be sent.

3.2 Rest Actions

In the previous section we have described the different data models that an agent and its REST Governor can interchange. In this section we will focus on the actions that an agent can request its governor to do into the institution. These actions are controlled by web services following a REST methodology. For this reason the description of each available action will start with a box with the HTTP request method (GET, POST, ...), followed by the base URI that identifies the action. After that follow a short description of the action and the required or optional arguments, and finally the expected JSON object returned as a result of the action.

3.2.1 Registering a Governor

```
GET /login.json
```

This web service is used to create a new link between the agent and its Governor. At this moment you only have to provide a user name and password that has at least eight characters.

- **agentName** String value with the name of the user to identify it into the infrastructure.

Listing 10: Example of a reason in JSON format

```json
{
  "code": 1,
  "msg": "There is no way from/to the conversation in the specification.[1]."
}
```
**agentPassword**  *String* value with the password used to connect the agent with its *REST Governor*.

If the agent can not be logged in into the infrastructure it will receive a *JSON* object with the following attributes:

**agentName**  *String* value with the name of the agent that has tried to log in.

**agentNameError**  *String* value with the error message that explains why the name is not valid to log in.

**agentPassword**  *String* value with the password that the agent has tried.

**agentPasswordError**  *String* value with the error message that explains why the password is not valid to log in.

Otherwise, if the agent did get access to the institution, it receives a *JSON* object with the following attributes:

**agentName**  *String* value with the name of the agent that has logged in.

**agentHash**  *String* value with the logged session identifier.

The rest of the web services require these attributes to identify the session where the agent is logged in. Thus, in the following web actions the agent has to add the **agentName** and **agentHash** values obtained as parameters, otherwise it receives the **401 HTTP** error (Authentication is required).

### 3.2.2 Logging Out

**GET**  `/logout.json`

This action is used to close the agent connection with is Governor. This is only possible if the agent is no longer participating in any Electronic Institution, otherwise the Governor will remain active until the agent has left all the Electronic Institutions where it has entered.

One needs to add the following parameters to the log out action:

**agentName**  *String* value with the name of the agent.

**agentHash**  *String* value with the session identifier obtained from the log in process.

If all goes well you obtain an empty *JSON* object, otherwise you will obtain an object with the following attributes:
agentName  String value with the name of the agent.

agentHash  String value with the session identifier obtained from the log in process.

actionError  String value with the error message that explains why it was not possible to log out.

3.2.3 Searching for an Electronic Institution

```
POST /search/einstitution.json
```

This action is used to search for Electronic Institution specifications that have been uploaded to the platform (see paragraph 2.1.3). You can search the published Electronic Institutions with a specific name, description, keywords or role names. The search process may require more time than a request to a standard web service would need, so for this reason the first call will return a search identifier which can later be used to request the state of the search. The maximum time available for a search is five minutes. After that, if you ask for the state again, you will be informed that the search action has finished without any result. If you are familiar with the Apache Lucene syntax you can search by creating a query and using the following parameters:

agentName  String value with the name of the agent.

agentHash  String value with the session identifier obtained from the log in process.

eInstitutionQuery  String value with the Apache Lucene query that should be used to search for Electronic Institution specifications.

Otherwise, you can use the following parameters to perform a search over some of the fields that describe the Electronic Institutions, using a pattern with wildcards, and search... (more about can be found in Section 2.1.3).

agentName  String value with the name of the agent.

agentHash  String value with the session identifier obtained during the log in process.

eInstitutionName  String value with the Apache Lucene pattern that the names of the returned EI-specifications should match.

eInstitutionDescription  String value with the Apache Lucene pattern that the descriptions of the returned EI-specifications should match.

eInstitutionKeywords  String value with the Apache Lucene pattern that at least one of the keywords of each returned EI-specification should match.
eInstitutionInputRoles  *String* value with the *Apache Lucene* pattern that at least one of the roles of each returned EI-specifications should match.

When you have done any of the previous queries you obtain a search identifier that you can use to request the search state using the following parameters, but using the *GET* method.

**agentName**  *String* value with the name of the agent.

**agentHash**  *String* value with the session identifier obtained during the log in process.

**eInstitutionSearchId**  *String* value with the search identifier obtained after the first *POST*.

The possible reply for any of the the previous *HTTP* queries is a *JSON* object with the following parameters (As an illustration see Listing 11).

**eInstitutionSearchId**  *String* value with the search session identifier.

**eInstitutionQuery**  *String* value with the *Apache Lucene* query. This value is only returned after the search process has been started.

**actionFinished**  *Boolean* value that is *true* when the search action has finished.

**foundEInstitutions**  *Array of JSON* object with the found Electronic Institution specifications that match the specified query. Each of the elements of the array contains the following attributes:

- **eInstitutionId**  *String* value with the URL that identifies the Electronic Institution specification. This address is public and can be used to obtain the Electronic Institution rules.

- **eInstitutionName**  *String* value with the name of the Electronic Institution specification.

- **eInstitutionDescription**  *String* value with the description of the Electronic Institution specification.

- **eInstitutionInputRoles**  *Array of String* values with the names of the roles that an agent can adopt when entering the Electronic Institution.

- **eInstitutionKeywords**  *Array of String* values with the keywords that describe the the Electronic Institution.
Listing 11: Example of a search result in JSON format

```json
{
  "eInstitutionSearchId":1,
  "eInstitutionQuery":"(input_role:"student")",
  "foundEInstitutions":[
    {
      "eInstitutionId":"http://10.10.200.199:39572/MusicCircle_1000451154.xml",
      "eInstitutionName":"MusicCircle",
      "eInstitutionDescription":"",
      "eInstitutionInputRoles":["student","teacher","staff"],
      "eInstitutionKeywords":[]
    },
    {
      "eInstitutionId":"http://10.10.200.199:39572/ViolinLesson_207674017.xml",
      "eInstitutionName":"ViolinLesson",
      "eInstitutionDescription":"",
      "eInstitutionInputRoles":["student","teacher","staff"],
      "eInstitutionKeywords":[]
    }
  ]
}
```
3.2.4 Registering an Electronic Institution Specification

**POST** /publish/einstitution.json

This action is used to upload an Electronic Institution specification to the system. This specification is then registered in a database so that any other agent can consult or access it. In this case it is necessary that you configure the `enctype` as `multipart/form-data`, otherwise the file will not be uploaded. The action parameters are:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the log in process.
- **eInstitutionFile** *File* that contains the Electronic Institution specification to register.

If the Electronic Institution cannot be registered it will return an empty JSON object, otherwise it will return a JSON object with the attribute `eInstitutionId` with the Electronic Institution identifier. This identifier is a URL that you or other people in the system can use to obtain the uploaded Electronic Institution file.

3.2.5 Getting Information of a Registered Electronic Institution Specification

**GET** /show/einstitution.json

This action is used to obtain information of a registered electronic institutions. The action parameters are:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the log in process.
- **eInstitutionId** *String* value with the URL that identifies the Electronic Institution specification. This address is public and can be used to obtain the Electronic Institution rules.

The result will be a JSON object equal to the one that one retrieves after a search for Electronic Institutions as described in Section 3.2.3. Thus, this object will contain the Electronic Institution name, description, roles and keywords. If the id does not refer to a registered or valid Electronic Institution, this request returns an empty JSON object.
3.2.6 Searching for all Running Electronic Institutions

GET /search/eimanager.json

With this command you can search the system for all the running Electronic Institutions for a specific specification. As the previous search operation (see Section 3.2.3), this action requires more time than the default for a normal web service reply, so for this reason the first time you make this request, you will receive a search identifier that you can use later to request the search state. The action parameters are:

- **agentName**  *String* value with the name of the agent.
- **agentHash**  *String* value with the session identifier obtained during the log in process.
- **eInstitutionId**  *String* value with the URL that identifies the Electronic Institution specification that has to run the searching active managers.

The reply of the server will be a JSON object with the following attributes:

- **eIManagerSearchId**  *String* value with the search session identifier.
- **foundEIManagers**  *Array of String* values with the identifiers of the running Electronic Institutions that are executing the specified Electronic Institution specification.

After making the request for the search, you can request the state of the search using the parameter **eIManagerSearchId** instead of the **eInstitutionId**. The search operation finishes however after five minutes; after this time you would receive an empty JSON object.

3.2.7 Starting a New Electronic Institution

POST /start/eimanager.json

This action is used to start a new Electronic Institution instance. This starts a new AMELI environment (see Sections 1.3 and 2.3). This request needs the following parameters:

- **agentName**  *String* value with the name of the agent.
- **agentHash**  *String* value with the session identifier obtained during the log in process.
**eInstitutionId** *String* value with the URL that identifies the Electronic Institution specification that should be executed.

If the institution cannot be created the agent receives an empty *JSON* object as a response. Otherwise, it receives a *JSON* object with the following attributes:

- **eInstitutionId** *String* value with the URL that identifies the Electronic Institution specification that is being executed.
- **eiManagerId** *String* value with the identifier of the new Electronic Institution instance.

### 3.2.8 Enter to Participate in an Electronic Institution

**POST** `/enter/einstitution.json`

This action is used to enter a specific Electronic Institution instance. The action requires the following parameters:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the login process.
- **eInstitutionId** *String* value with the URL that identifies the Electronic Institution specification that is executed by the Electronic Institution.
- **eiManagerId** *String* value with the identifier of the Electronic Institution instance to enter.
- **eiAgentName** *String* value with the name with which the agent will enter the institution.
- **eiAgentRole** *String* value with the name of the role that the agent will adopt when entering the institution.

If the agent is not allowed to enter the Electronic Institution it will receive an empty *JSON* object. Otherwise it will return a *JSON* object with the attribute `gov-ernorId` that is the governor session identifier in the entered institution. For every action that the agent wants to do in the institution it will need to pass this identifier in the request.
3.2.9 Obtaining Information about All Electronic Institutions the Agent is Participating in

**GET /playing_einstitutions.json**

This action is used to get information of Electronic Institutions in which the agent is participating. The parameters are:

- **agentName**  String value with the name of the agent.
- **agentHash**  String value with the session identifier obtained during the log in process.

It returns an array of JSON objects, one for each Electronic Institution the agent is participating in, that have the following attributes:

- **governorId**  String value with the identifier of the governor session in the Electronic Institution instance.
- **conversations**  Array of Integer values with the identifiers of the conversations (scene instances and transitions) in which the agent is participating.

3.2.10 Obtaining Information of an Electronic Institution Conversation

**GET /conversation.json**

This action can be used to obtain information of the state of a conversation in which the agent is participating. The action parameters are:

- **agentName**  String value with the name of the agent.
- **agentHash**  String value with the session identifier obtained during the log in process.
- **governorId**  String value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId**  Integer value with the identifier of the conversation in the Electronic Institution instance, of which the agent wants information.

The returned object depends on the state of the conversation: playing in a scene, exited from a scene, or moving to some other conversation. The possible attributes that the returned JSON can have are displayed in Table 1.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>governorId</td>
<td>string</td>
<td>Any</td>
<td>Identifier of the governor in the Electronic Institution.</td>
</tr>
<tr>
<td>conversationId</td>
<td>integer</td>
<td>Any</td>
<td>Identifier of the scene instance.</td>
</tr>
<tr>
<td>protocol</td>
<td>object</td>
<td>Playing</td>
<td>Object with the attributes that identify the scene in the Electronic Institution (see paragraph 3.1.2).</td>
</tr>
<tr>
<td>protocolState</td>
<td>string</td>
<td>Playing</td>
<td>The name of the state that the scene is in.</td>
</tr>
<tr>
<td>protocolPlayers</td>
<td>array</td>
<td>Playing</td>
<td>Array of JSON objects with the attributes of the participating agents (see paragraph 3.1.1).</td>
</tr>
<tr>
<td>accessMovements</td>
<td>array</td>
<td>Exited</td>
<td>Array of JSON object with the possible movements to other conversations (see paragraph 3.1.3).</td>
</tr>
<tr>
<td>transition</td>
<td>object</td>
<td>Moving</td>
<td>Object with the attributes that identify the transition node where the agent has moved to (see paragraph 3.1.2).</td>
</tr>
<tr>
<td>conversations</td>
<td>array</td>
<td>Moving</td>
<td>Integer array with the identifiers of the conversations where the agent has moved to.</td>
</tr>
<tr>
<td>conversationEvents</td>
<td>array</td>
<td>Any</td>
<td>Array of JSON objects that describes the events and actions that have happened in the conversation.</td>
</tr>
</tbody>
</table>

Table 1: Return attributes for the JSON object of the GET conversation action
3.2.11 Getting information about the Possible Movements

GET /conversation/access_movements.json

This action can be used by the agent to obtain the possible conversation where the agent can go to from its current conversation. As previous actions this execution can require more time than available for a rest connection, so when you make this request for the first time you obtain an identifier that you can use to get information about the state of the request. The action parameters are:

agentName  String value with the name of the agent.

agentHash  String value with the session identifier obtained during the log in process.

governorId  String value with the identifier of the governor session in the Electronic Institution instance.

conversationId  Integer value with the identifier of the conversation of which the agent wants to know the possible movements to other conversations.

The server will reply with a JSON object with the following attributes:

governorId  String value with the identifier of the governor session in the Electronic Institution instance.

conversationId  Integer value with the identifier of the conversation in the Electronic Institution instance from which the agent can move to other conversations.

governorActionId  Integer value with the identifier of the request that can be used to retrieve the current state of the request.

governorActionFinished  Boolean value that is true if all required information has been found.

accessMovements  Array of JSON objects with the movements that the agent can do from the specified conversation (see paragraph 3.1.3).

Remember that until the governorActionFinished is not equal to true, the Governor is still collecting information from the institution, so, you need to request the action again, adding the parameter governorActionId in order to obtain the query state.
3.2.12 Moving to Another Conversation

**POST /conversation/move_to.json**

This action can be used by the agent to move from one conversation to another. Note that in order to move to another conversation the agent first needs to exit from its current scene (see paragraph 3.2.15). This action requires more time than is available for a rest connection, so when making this request for the first time, the agent obtains an identifier that can be used later to obtain the state of the request. The action parameters are:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the log in process.
- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation from which the agent wants to move.
- **movement** *JSON* object with the movement that the user wants to do (see paragraph 3.1.3).

The server will reply with a *JSON* object with the following attributes:

- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation from which the agent could move.
- **governorActionId** *Integer* value with the identifier of the action that can be used to ask again about the movement state.
- **governorActionFinished** *Boolean* value that is *true* if the movement has completed.
- **reason** empty if the movement completed successfully, otherwise a *JSON* object with the explanation why the movement failed (see paragraph 3.1.10).
- **conversations** *Array of Integers* with the identifiers of the conversations that the agent has moved to.

Remember that until the **governorActionFinished** is not equal to *true*, the movement action is still processing. So, the agent needs to call the action again, adding the parameter **governorActionId** with the obtained value, in order to obtain the movement state.
3.2.13 Getting Information of the Possible Messages to Say

```
GET /conversation/possible_messages.json
```

This action can be used by the agent to get the patterns of the messages that it can say in a conversation. This action requires more time than available for a rest connection, so the first time you will obtain an identifier that you have to use to obtain the information of the action result. The action parameters are:

- **agentName** *String* value with the name of the user when it has logged in.
- **agentHash** *String* value with the session identifier obtained during the login process.
- **governorId** *String* value with the identifier of the governor session in the Electronic Institution environment.
- **conversationId** *Integer* value with the identifier of the conversation where the messages can be said.

The server will reply with a *JSON* object with the next attributes:

- **governorId** *String* value with the identifier of the governor session in the Electronic Institution environment.
- **conversationId** *Integer* value with the identifier of the conversation where the messages can be said.
- **governorActionId** *Integer* value with the identifier of the action that can be used to ask again about the possible messages.
- **governorActionFinished** *Boolean* value that is *true* if it has obtained the required information.
- **possibleMessages** *Array of JSON* objects with the patterns of the messages that can be said (see paragraph 3.1.9).

Remember that until the **governorActionFinished** is not equal to *true*, the action is still processing. So, the agent needs to call the action again, adding the parameter **governorActionId** with the obtained value, in order to obtain the possible messages.

3.2.14 Saying a Message in a Scene

```
POST /conversation/say_message.json
```

This action is used by the agent to communicate a message to another agent. The action parameters are:

- **agentName** *String* value with the name of the user who will send the message.
- **agentHash** *String* value with the session identifier obtained during the login process.
- **governorId** *String* value with the identifier of the governor session in the Electronic Institution environment.
- **conversationId** *Integer* value with the identifier of the conversation where the message will be said.
- **message** *String* value with the content of the message to be sent.

The server will reply with a *JSON* object with the next attributes:

- **governorId** *String* value with the identifier of the governor session in the Electronic Institution environment.
- **conversationId** *Integer* value with the identifier of the conversation where the message was said.
- **messageId** *Integer* value with the identifier of the message that has been said.
- **messageSent** *Boolean* value that is *true* if the message has been sent.

Remember that if the agent has not yet obtained all the possible messages, it is necessary to call the action again using the **governorActionId** parameter to get the complete list of messages.
This action can be used by the agent to say a message to another agent in a conversation. Remember that the message is only sent to the receiver if it does not break any of the rules of the Electronic Institution.

This action requires more time than is available for a rest connection, so when making this request for the first time, the agent obtains an identifier that can be used later to obtain the state of the request. The action parameters are:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the log in process.
- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation where agent is trying to say the message.
- **message** *JSON* object with the message that the agent is trying to say (see paragraph 3.1.9).

The server will reply with a *JSON* object with the following attributes:

- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation where the agent is trying to say the message.
- **governorActionId** *Integer* value with the identifier of the action that can be used to ask again about the state of the say action.
- **governorActionFinished** *Boolean* value that is *true* if it has obtained the required information.
- **reason** This is present if the message could not be said because some rule of the institution was not satisfied. It is a *JSON* object with a text that explains why the message could not be said (see paragraph 3.1.10).
- **protocolState** This is present if the message has been said. It is a *String* value with the name of the new scene state after the message has been said.

Remember that until the **governorActionFinished** is not equal to *true*, the action is still processing. So, the agent needs to call the action again, adding the parameter **governorActionId** with the obtained value, in order to obtain the state of the say action.
3.2.15 Exiting from a Scene

**POST /conversation/exit_scene.json**

This action is used by the agent to exit from a scene before moving to a new conversation. Once the agent has requested this action the governor waits until it is possible to do so, so it can require a lot of time, which can produce a timeout for the rest action. For this reason, when the agent calls it, it obtains an identifier that it can use to obtain the information of the action result, calling again with this information. The action parameters are:

- **agentName** `String` value with the name of the agent.
- **agentHash** `String` value with the session identifier obtained during the log in process.
- **governorId** `String` value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** `Integer` value with the identifier of the conversation from where the agent is requesting to leave.

The server will reply with a `JSON` object with the following attributes:

- **governorId** `String` value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** `Integer` value with the identifier of the conversation where the agent is trying to leave.
- **governorActionId** `Integer` value with the identifier of the action that can be used to ask again about the exit state.
- **governorActionFinished** `Boolean` value that is `true` if it the user has successfully exited from the scene.
- **reason** This is present if the user could not leave the scene. It is a `JSON` object with a text that explains why the user could not leave the scene (see paragraph 3.1.10).
- **exitedScene** `Boolean` value that is `true` if the agent has left the scene.

Remember that until the `governorActionFinished` is not equal to `true`, the system is still waiting to know if the agent can leave the scene or not. So, the agent needs to request the action again, adding the parameter `governorActionId` with the obtained value, in order to know whether the agent has left the scene or not.
3.2.16 Do a ‘Stay And Go’ in a Scene

**POST** /conversation/stay_and_go.json

This action is used by the agent to do an ‘stay and go’ into a scene. That is: to go to another scene, while staying in the current scene at the same time. Once made this request the governor waits until it is possible to do a ‘stay and go’ action. This can require a lot of time, so this can produce a timeout. For this reason, when you it is called, the agent obtains an identifier that can be used to obtain information about the action result. The action parameters are:

- **agentName** *String* value with the name of the agent.
- **agentHash** *String* value with the session identifier obtained during the log in process.
- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation where the used would do the ‘stay and go’.

The server will reply with a *JSON* object with the following attributes:

- **governorId** *String* value with the identifier of the governor session in the Electronic Institution instance.
- **conversationId** *Integer* value with the identifier of the conversation from where the agent wants to do the ‘stay and go’.
- **governorActionId** *Integer* value with the identifier of the action that can be used to ask again about the action state.
- **governorActionFinished** *Boolean* value that is **true** if the action has finished.
- **reason** This is present if the agent was not allowed to do the ‘stay and go’ in the conversation. It is a *JSON* object with a text that explains why it was not possible to do the action (see paragraph 3.1.10).
- **goneConversationId** This is present if the ‘stay and go’ action was successful. It is an *Integer* value with the identifier of the conversation where the agent is going to.

Remember that until the **governorActionFinished** is not equals to **true**, the system is waiting to known if the agent can or not do the ‘stay and go’ action in the scene. So, you have to call again the action adding the parameter **governorActionId** with the obtained value to known if the agent has cloned or not.
Remember that until the governorActionFinished is not equal to true, the system is still waiting to know if the agent can do the ‘stay and go’ or not. So, the agent needs to request the action again, adding the parameter governorActionId with the obtained value, in order to know whether the agent has successfully performed the ‘stay and go’.

3.3 Peer to Peer user interface

The rest services are useful if you are programming an agent, but it could be difficult to use for a human user. To simplify the user interaction with these services we have created some HTML pages served by the embedded Jetty server. So, a human user can interact with the services described in Section 3.2 using a web browser.

When a new peer to peer infrastructure is started with the REST Governor, it starts the default browser and opens the log in web page. This page is used to create a Governor to participate into the P2P infrastructure. After the user has been logged in, it can go to the main page (see Figure 10) where it can a) search for the available electronic institutions (see Section 3.2.3); b) publish an electronic institution specification (see Section 3.2.4); or c) show which electronic institutions it is participating in (see Section 3.2.9).

In Figure 10 you can see the result of searching for all the available specifications in which a user can participate with the Student role. If you click over the ‘?’ image of one of these specifications, a new web page is loaded with the information of the specification (see Figure 11). In this page it is possible to search for the running electronic institutions that follow the rules of a particular specification, or to start a new one. In both cases when you have a running electronic institution to join you see a web page similar to Figure 12. In this form you only have to choose the role that
you wish to play in the electronic institution and it calls the Governor to enter it (see Section 3.2.8).

When you have been accepted in the electronic institution, you will view a page with the information of the possible movements to the scenes where the agent can go (see Figure 13). You can edit the path to the desired scene and do the movement. As we explain before the movement action may require some time (see Section 3.2.12), for this reason the page uses some AJAX timeouts to reload the page with the movement state.

Finally, in Figure 14 you can see the page that you obtain when you have been admitted into a scene protocol. In this page you can see a) the name and role of the agents that participates in the scene; b) the history of events that have happened in the scene; c) the possible messages that the agent can say in the current state; d) a button to quit the scene; and e) a button to do an ‘Stay And Go’.

### 3.4 Future work

The future work for this tool will be to focus on using different data codifications not only JSON, and to provide a more user friendly user interface. For the first one we will modify the embedded server to allow adding plugins to return the data in different formats as XML or prolog terms. These changes can help the agent developers to create an easier way to integrate their agents.
Figure 13: Form to move to another scene

Figure 14: Scene protocol view
On the other hand we will improve the user interface to be more user friendly, because some pages as those that are used to participate in an electronic institution (see Figures 14 and 13) are difficult to understand if you do not have some knowledge about electronic institutions. For this we will merge the current work with the tool that generate automatic user interfaces for electronic institutions (see Section 4).

4 User Interface for Electronic Institutions

The goal of PRAISE is to have human users engaging in social music learning, through a community that is regulated by an electronic institution. This means that we needed to provide a user interface so that users can indeed connect to and interact within an electronic institution.

In this section we describe how we enable human users to participate in electronic institutions, interacting with one another and with software agents, through a web browser. One can for example imagine that human users share recordings of their playing an instrument, and comment on each other’s recordings. This may be aided by software agents analysing some objective musical aspects of the recordings. All this could be regulated by an electronic institution that forces the users to follow certain lesson plans.

4.1 Our Framework

A human user would interact in an EI by clicking buttons in a browser window. To allow these actions to have effect in the EI, we have implemented a software agent that can represent the user inside the EI and that executes the actions requested by the user. This agent is called the GuiAgent. Although the current implementation does not do anything autonomously, if necessary it can be extended to give it more sophisticated capabilities, such as giving intelligent strategic advice to the user.

Apart from the GuiAgent we need to have a GUI in the form of a website that allows the user to interact. On one hand, one may want to have a good-looking GUI that is specifically designed for a given institution. But on the other hand, one may not want to develop an entirely new GUI for every new institution, or one may want to have a generic GUI available to test a new EI during its development, so that one can postpone the design of its final GUI until the EI is finished and tested. Therefore, we have built a framework that allows for both. It generates a GUI automatically from the EI-specification, but at the same time provides an API so that web designers can easily create a custom GUI for each new EI. It is used on top of the existing EIDE framework and consists of the following components:

- A Java agent called GuiAgent that acts in the Institution on behalf of the human user.
D3.1 Electronic institutions for community building (v1) FP7-ICT-2011- 8 Collaborative project 318770 PRAISE

(a) ‘Classic’ EI with only software agents

(b) An EI with one software agent and two human users participating

Figure 15: Interaction with an electronic institution

- A Java component that encodes all relevant information the agent has about the current state of the institution into an xml file.

- A Javascript library called *EiGuiInterface* that translates the xml file into a Javascript object called *EiStateInfo* that represents the information about the Electronic Institution.

- A Javascript library called *DefaultGuiGenerator* that generates a default Graphic User Interface (as html) based on the EiStateInfo object.
4.2 How it Works

In order for a user to participate in an institution, there must be some server running an instance of this institution and user needs to know the url of that server. When the user wants to join the institution, all he or she needs to do is to open a web browser on any device, and navigate to institution’s url. The process then continues as follows:

1. A web page including both Javascript libraries is loaded into the browser of the user.
2. The page sends a login request to the server.
3. Upon receiving this request the server starts a GuiAgent to represent the user and, depending on the specific institution, other agents necessary to run the institution.
4. When the GuiAgent is instantiated it analyzes the EI-specification to retrieve all static information about the institution (see below).
5. The page starts a polling service that periodically (typically several times per second) requests a status update from the GuiAgent.
6. When the GuiAgent receives a status update request it asks its Governor for the dynamic information about the current status of the institution.
7. The GuiAgent converts both the static and the dynamic information into an xml file which is sent back to the browser.
8. The DefaultGuiGenerator Javascript library then uses this information to update the user interface (more information about this below).
9. The user can now execute actions in the institution or move between its scenes by clicking buttons on the web page.
10. For each action the user makes, a http-request is sent to the GuiAgent.
11. The GuiAgent uses the information from the http-request to create an EI-message which is sent like any other message in a standard EI.

As explained, the GuiAgent uses two sources of information: static information from the EI-specification stored on the hard disk of the server and dynamic information from the Governor. The static information consists of:

- The names and protocols of the scenes defined in the institution.
- The roles defined in the institution.
- The ontology of the institution.
Table 2: The http-requests sent from the browser to the GuiAgent

<table>
<thead>
<tr>
<th>Http-Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/login?name=alice&amp;role=guest</td>
<td>Enter the EI with given name and role.</td>
</tr>
<tr>
<td>/sendMessage?name=alice&amp;receiver=bob&amp;msg=bid&amp;amount=1000</td>
<td>Send a message with given parameters.</td>
</tr>
<tr>
<td>/gotoScene?name=alice&amp;role=guest &amp;sceneName=Admission</td>
<td>Enter the given scene with the given role.</td>
</tr>
<tr>
<td>/exitScene?name=alice</td>
<td>Exit the given scene.</td>
</tr>
<tr>
<td>/gotoTransition?name=alice &amp;transitionName=transition1</td>
<td>Go to the given transition.</td>
</tr>
<tr>
<td>/request_update?name=alice</td>
<td>Request an update of the status of the EI.</td>
</tr>
</tbody>
</table>

While the dynamic information consists of:

- The actions the user can do in the current scene.
- For each of these actions: the parameters to be filled out by the user.
- Which agents are present in the current scene
- Whether it is allowed to leave the scene and, if yes, to which other scenes the user can move.

### 4.3 Generating the GUI

Every time the browser receives information from the GuiAgent, it updates the GUI, which takes place in two steps, handled by the two respective Javascript libraries. In the first step the EIGuiInterface converts the received xml into a Javascript object called EiStateInfo, which is composed of smaller objects that represent the static and dynamic information as explained above.

In the second step the EiStateInfo-object is used by the DefaultGuiGenerator library to draw the GUI. This GUI is completely generic, so it looks the same for every institution. If one wants to design a more fancy user interface tailored to one specific institution, one can write a new library that replaces the DefaultGuiGenerator.
Figure 16: The components necessary to generate the GUI. Solid arrows indicate exchange of information. The dashed arrow indicates that the GUI is created by the DefaultGuiGenerator.
Electronic Institution: MusicCircle_0.5

Scene: OtherQuestions
Current State: -
Your Role: user

Your Actions
Please choose your next action: 

Specify the details of your action:
Choose Action:
- questionEdit
- responseEdit
- submit

Incoming Messages
- score Granted
- text
- data
- data
- data

Figure 17: The Default Gui, applied to the Music Circle institution.

The fact that these two steps are handled by two different libraries enables a designer to reuse the EIGuiInterface when designing a new GUI, so he does not have to worry about how to retrieve the relevant information from the EI. All information will be readily available in the EiStateInfo-object, so you only need to worry about how to display it on the screen.

4.4 The Default User Interface

The default user interface is displayed in Figure 17. It is divided in four sections:

- A menu bar in the top that allows for navigating from scene to scene.
- A panel showing general information about the status of the user: the scene in which it currently is, the state of the scene, and the role the user is playing.
- A panel to display the messages coming in from other users and agents.
- A panel where the user can choose which action to take (i.e. which message to send), and fill out the parameters.

The figure shows the GUI for an agent participating in an Electronic Institution named as MusicCircle. This institution has six scenes, hence the menu bar shows six menu options. Two of those menu items are grayed out meaning that the user
currently cannot move to those scenes. We have intentionally chosen to make the navigation between scenes resemble as much as possible the way a user navigates between menu-items on a regular website. If an user is present in two or more scenes at the same time, the browser will have a separate tab opened for each scene.

The panel on the left is where the user interacts with the other users and agents in the EI. The user can choose a which action to take from a drop down list. This list only shows those actions that the user currently can do, hence preventing the user from sending illegal messages (note that even if the user would be able to send illegal messages, they would still be blocked by the Governor. However, for the sake of user-friendliness we only want to display messages that the user can indeed send).

Once the user has selected an action to take, the browser displays a form that enables the user to fill out the necessary parameters. Since each action in the institution is modeled as a message, the user needs to choose an agent that is going to receive the message. The EI-specification defines which agents can receive which kind of message. A bid in an auction house for example should always be sent to the agent playing the role of auctioneer.

The form shows one input control for each parameter of the message. The type of control depends on the type of the parameter. For example, if the the parameter is of type integer, a numeric input control appears, while if the parameter is of type string, a text box appears. In case the parameter is of a user-defined type, a sub-form appears with several controls, one for each of the variables of the user-defined type.

### 4.5 Customizing the GUI

In order to draw a GUI, a GUI-generator can use the information from the EiStateInfo-object. For example: if the user chooses to make a bid in an auction. The GUI-generator would read from the EiStateInfo-object that an Integer parameter must be set to represent the amount of money the user wants to bid. A GUI-designer should make sure that whenever a parameter of type Integer is required the GUI displays an input-control that allows the user to introduce an integer value.

The fact that one can also define user-defined types in an EI allows for a lot of flexibility. Suppose for example that one would like an user to record an audio file and send this in a message to an other agent. Electronic Institutions do not support Audio files by default. However, the institution designer may introduce a user-defined type with the name ‘Audio’. Once the user chooses to send a message that includes audio, the EiStateInfo-object will indicate that a parameter of type Audio is needed. A customized GUI-generator could then be programmed such that a microphone is activated whenever this type of parameter is required.
4.6 The API

4.6.1 Requirements

In order to run an Electronic Institution in which a user can participate you need the following components:

- The EiServer.
- The xml-specification of an Electronic Institution.
- A website that is going to act as the GUI through which the user can interact with the institution. This website should include the EiGuiInterface.js javascript file.

The website and the xml-spec need to be uploaded at the server.

When the website calls the login() function with a username and a role as parameters, a request is sent to the server to start a GuiAgent, which will represent the user in the institution. This GuiAgent will then enter the institution with the given name and role. The user controls the GuiAgent through the website.

4.6.2 Generating and updating the GUI

When the user is logged in, a polling service is automatically started that retrieves an EiStateInfo object at regular intervals. This object contains all the relevant information for the GUI about the Electronic Institution. Each time such an object is retrieved it is passed to a method called store(). This method should be implemented by the GUI designer so that he can fetch the EiStateInfo object and use it to generate the GUI.

4.6.3 Sending Messages

When the user wants to send a message, he or she may need to set a number of parameters in the message. These parameters can be retrieved by calling the getMessageVariables() method, which returns an array of Variable objects. The GUI-designer should make sure that the user can input the values he wishes for the parameters and that they are set in their corresponding Variable objects. These variable objects can then be passed to the sendMessage() function, in order to send the message.

4.6.4 Moving to a new Scene

When the user wants to move to another scene the moveToTransition() function should be called to move the GuiAgent from the current scene to a transition. Arrived at the
transition, the gotoScene() function should be called to move from the transition to the scene.

4.6.5 Functions

This section describes the relevant functions implemented in the EiGuiInterface.js file.

4.6.5.1 login(name, role)

Call this function to request to login with the given name and role.

- name: a string specifying the name with which the user wants to log in.
- role: a string specifying the role with which the user wants to log in.

4.6.5.2 sendMessage(messageName, receiverName, actionVariables)

Call this method to send a message.

- messageName: a string specifying the name of the message.
- receiverName: a string specifying the name of the receiver of the message.
- actionVariables: an array of Variable objects that hold the values the user has chosen.

Note that a message can only be sent if for all its required parameters the itsValue field of the corresponding Variable object is set.

4.6.5.3 getMessageVariables(actionName)

Given the name of an action, one can retrieve the set of variables that need to be set by calling this function. Returns an array of Variable objects (see Section 4.6.9).

For each of the returned objects the itsValue property will be undefined. The GUI-designer should make sure that the value of this property is set when the user has entered the value he wants to assign to the variable.
4.6.5.4 moveToTransition(targetSceneName)

Call this function to exit the current Scene and go to the transition that leads to the given target scene.

To be precise: it sends a request to exit the scene, and when a response to this request is received, it sends another request to move to the transition that lies in the path between the current scene and the target scene.

One can check whether the GuiAgent has arrived at the transition by verifying that the EiStateInfo.currentScene variable is undefined, while the EiStateInfo.currentTransitionName variable is defined.

4.6.5.5 gotoScene(targetSceneName, role, enterMode)

Once the GuiAgent has moved to a transition, this function can be called to request to move to the given scene, with the given role. The enterMode parameter is a string that can only have either the value “ONE” or the value “NEW”. It indicates whether you want to join an existing scene instance or start a new scene instance.

4.6.6 The EiStateInfo Object

The EiGuiInterface.js file defines a set of objects to store the information about the Electronic Institution. The most important one being the EiStateInfo object. It contains the following fields:

- **institutionName**
  - Type: String
  - Description: Indicates the name of the electronic institution.

- **scenes**
  - Type: Array of Scene objects (See Section 4.6.7)
  - Description: Contains one scene object for each scene in the institution.

- **userName**
  - Type: String
  - Description: Indicates the name of the user.

- **currentScene**
  - Type: Scene (See Section 4.6.7)
- Description: An object that represents the current scene in which the agent is (for now we assume an agent can only be in one scene at a time). If the agent is not in a scene, it is undefined.

- **currentTransitionName**
  - Type: String
  - Description: The name of the current transition. If the agent is not in a transition, it is undefined.

- **conversationId**
  - Type: Integer
  - Description: Identifies the current conversation. This value is zero in the initial scene and increases each time the agent moves to a new transition or a new scene.

- **role**
  - Type: String
  - Description: The role that the user is playing in the current scene instance. If the agent is not in a scene, this value is undefined.

- **agents**
  - Type: Array of Agent objects (see also Section 4.6.8)
  - Description: One for each agent that is currently present in the current scene. If the agent is not in a scene, this value is undefined.

- **stateName**
  - Type: String
  - Description: Indicating the name of the current state of the current scene. If the agent is not in a scene, this value is undefined.

- **inFinalState**
  - Type: Boolean
  - Description: Is ‘true’ if the current state is a final state of the current scene, and ‘false’ otherwise. If the agent is not in a scene, this value is undefined.

- **messageActions**
  - Type: Array of strings
  - Description: Lists the names of all the actions the user can do.

- **destinationScenes**
- Type: Array of Strings
- Description: Lists the names of all scenes that the user can currently move to.

• incomingMessages
  - Type: Array of Strings
  - Description: Lists new incoming messages sent to the user.

• userWarnings
  - Type: Array of Strings
  - Description: Warnings to the user. For example, if the user has sent a message but the Institution didn’t allow it, the user will receive a warning that the message failed including the reason why.

• debugWarnings
  - Type: Array of Strings
  - Description: debug info.

4.6.7 The Scene Object

A Scene object contains the following fields:

• name
  - Type: String
  - Description: The name of the scene.

• description
  - Type: Strings
  - Description: A description of the scene.

• isInitial
  - Type: Boolean
  - Description: Is ‘true’ if the scene is the initial scene, and ‘false’ otherwise.

• isFinal
  - Type: Boolean
  - Description: Is ‘true’ if the scene is a final scene, and ‘false’ otherwise.

4.6.8 The Agent Object

An Agent object contains the following fields:
**name**
- Type: String
- Description: The name of an agent.

**role**
- Type: String
- Description: The role of an agent.

### 4.6.9 The Variable Object

This object represents a variable for which the user needs to assign a value in order to send a message.

- **type**
  - Type: String
  - Description: The type of the variable. It can be any of the following: “Integer” “String”, “Float”, “Boolean”, “DataType”, “Enumeration”, or “Role”.

- **varName**
  - Type: String
  - Description: The name of the variable.

- **isRequired**
  - Type: Boolean
  - Description: Is ‘true’ if this variable is required (i.e. if this value is ‘false’ the user can send the message without specifying a value for this variable).

- **itsValue**
  - Type: the type specified by the ‘type’ field.
  - Description: The value currently assigned to the variable.

### 4.6.9.1 DataType Variables

DataTypes are types that are defined by the designer of the institution. They add a bit of object orientation to the Electronic Institution framework and are similar to classes in OOP languages such as java.

For example, an institution-designer could define a class named Person, that consists of three fields: age, of type Integer, height, of type Float and name, of type String.

DataType Variables are special cases of Variable objects, so they have the same properties, plus the following two:
D3.1 Electronic institutions for community building (v1)FP7-ICT-2011- 8 Collaborative project 318770 PRAISE

• **dataTypeName**
  
  - Type: Boolean
  - Description: The name the user gave to this class (e.g. “Person”).

• **fields**
  
  - Type: Array of Variable objects
  - Description: The variables that make up the DataType object (e.g. three Variable objects corresponding to the age, height and name of a Person-object).

Note that for a DataType Variable the `type` property is always “DataType”, and the `itsValue` property is always undefined. Also note that the fields of a DataType Variable can themselves also be DataType Variables again.

4.7 Discussion

To summarize this section: we have developed a set of components on top of the existing EIDE framework that allows human users to participate in an electronic institution through a web browser. One of these components is a javascript library that establishes the connection between a web browser and an agent that represents the user in the electronic institution. Another javascript library automatically generates a user interface, based on the institution specification and real-time information received from the Governor. This automatically generated user interface is especially useful for quick testing purposes, because it allows us to participate in the institution without having to design a user interface. On the other hand it can easily be replaced by a new javascript library if one wants to design a custom user interface.

Currently, this new framework does not allow users to play in more than one scene at the same time. This is a feature that needs to be added in the future, because in many cases it is necessary that an user can be in more than one scene at the same time. Furthermore, it currently does not support users changing roles during the execution of the institution. Also, when a user creates a new scene instance, the framework does not allow the user to select which protocol should be executed.

5 Integration of Web Services

The PRAISE framework will provide several services to help the user. One can think for example of automatic agents that analyse certain aspects of the music, such as timing, or similarity with an example recording. However, one can imagine that if many users want to use this tool at the same time, the server may get overloaded.
Therefore, it is essential that the electronic institution can prevent users from overusing the tool. It could for example restrict access to the tool to one time a day for normal users, but allow extra access to premium users.

We here describe how Electronic Institutions can be applied to give a user access to a web service, or prevent the user from accessing it. As an example we take a web service in which a user sends an audio file, containing music played by the user with a musical instrument, together with a file that represents the score of the music, to a web service which then analyses the ‘onset’ of the musical notes. The results of this analysis are then sent back to the user in HTML, JSON or XML format.

The difficulty of integrating a web service with the existing electronic institution infrastructure lies in the fact that the current implementation only allows agents to exchange short text messages and does not allow the exchange of bulk data such as audio. Therefore, on one hand, the bulk data needs to be sent to the web service in a separate channel, but on the other hand the Electronic Institution still needs to be able to interrupt the exchange of information between the user and the web service, when necessary. We have solved this problem with the following protocol (see also Figure 18):

1. The user clicks ‘open webservice’, which causes a http request to be sent to the EI-server, which passes the request on to the GuiAgent.
2. The GuiAgent sends an EI-message within the institution to a WebServiceAgent.
3. The webservice agent sends an ‘access granted’ or ‘access denied’ message, with a url to the webservice, back to the GuiAgent.
4. The GuiAgent passes this message back to the browser in a http response.
5. The browser then opens the url in a separate window.
6. The new window enables the user to send data to the web service. For example, it allows the user to choose an audio file to upload. The user clicks submit to send the data. This will cause the data to indeed be sent to the web service, but also sends a notification to the EI-server.
7. When the web service has analyzed the audio, the results are stored on the server.
8. The WebService notifies the WebServiceAgent that the data is ready.
9. The WebServiceAgent sends an EI-Message to the GuiAgent including a url to the file with the results.
10. The GuiAgent sends the url to the browser, which can then use this url to load the data.
Figure 18: Interaction between an user and a web service, mediated by an EI. Solid arrows represent exchange of messages, dashed arrows represent the creation of a resource. The arrows are numbered in chronological order.
Note that indeed the EI is in control of the interactions between the user and the web service. The users cannot access the service without permission from the EI, since the user needs the url of the web service. The EI could send, together with the url, a password that can be used only once, to make sure that the user needs to request permission from the EI every time.

We have assumed in the above that the web service provides its own GUI, that needs to be opened in a separate browser window. Of course, this is not necessary, and the GUI to the web service may as well be fully integrated with the EI GUI. In that case step 5 can be skipped.

5.1 Discussion

One challenge we need to face here is that on one hand we want to be able to integrate any third party web service into our framework without having to make any changes to the web service and making as few changes to the PRAISE framework as possible. But, on the other hand, we want the user to have the experience that all web services are fully integrated with the PRAISE platform. These two objectives may sometimes be conflicting.

For example, a user may have uploaded an audio recording to the Music Circle database, and want an audio analysis tool to analyse it. This audio analysis web server may however require that the user upload the recording again. For the user this is inconvenient since he or she has already uploaded the recording and it would make more sense if the web service would simply take the recording from the database.

To solve this problem we would have to write a layer between the web service and the user that allows the user to select one of its previously uploaded recordings and then automatically uploads this recording from the database.

In other words, although we aim to have a general connection between the PRAISE framework and any web service, we still may need to write some extra code in order to integrate third-parties web services seamlessly into our framework.

6 Electronic Institution Specification

In this section we describe the Electronic Institution specification that we have developed as a first use case for the Music Circle website. This specification will be the starting point for the further development of the website.

The idea behind the specification is the following: a student learns to play a musical instrument and is guided by a tutor, as well as other students (so called ‘peers’). After every practice session the student will upload a recording of his or her play, together with a file that represent the musical score played by the student. A web service will give automatic feedback on the onset of the notes played by the student,
in comparison to the score. The tutor and peers can then download and listen to the track, and have the opportunity to give feedback to the student.

The Electronic Institution we have designed makes sure that the discussion follows a standardized pattern, in order to enforce the active participation of all participants. For now it only serves as an initial test case. The goal is to later implement institutions that follow a specific work flow that satisfies certain pedagogical principles.

### 6.1 Roles

The institution specifies five roles. the first three of them are to be played by human users, they are the above mentioned roles of student, tutor and peer. Moreover, there are two more roles that are to be played by software agents that represent the two web services. There is database role that is responsible for managing the uploaded tracks, and there is the role of onsetAnalyzer which is to be played by a software agent responsible for the onset analysis.

### 6.2 Scenes

We have chosen a design in which every activity that consists of interacting with a specific web service is modelled as a single scene. This allows a modular architecture of the institution. Every time a user needs to make use of a specific web service, the user leaves his current scene and moves to the scene dedicated to that web service.

In our current example there are two activities that are provided by web services: the first activity is the uploading of a recording, the second such activity is the analysis of the onset. Therefore we have designed two scenes in the institution that facilitate track upload and onset analysis respectively. Furthermore, there is a third scene in which the discussion between the student, the tutor and the peers takes place.

When a student enters the institution, he or she first moves to the Upload scene. Here he or she will upload his or her first recording. The student then moves on to the AnalyzeOnset scene where the student uploads a digital version of the score he or she has been playing. The onset analysis web service will then compare the uploaded track with the score file and reply with the results.

Finally, the student will enter the PeerDiscussion scene where a discussion with the tutor and the peers take place. During the discussion the student may need to make a new recording of his or her play and upload it. In order to do so, the student can leave the PeerDiscussion scene, an move back to the Upload scene and the AnalyzeOnset scene to finally come back to the PeerDiscussion scene and resume the discussion.
6.3 The UploadRecording Protocol

The UploadRecording allows the student to upload his or her performance to the database of Music Circle, so that it can be shared with other users. This protocol works in a very straightforward way:

1. The student uploads the audio file and simultaneously sends a message to the database agent that it is uploading.

2. When the upload is finished the database agent replies with a message to the student with an URL that links to the uploaded file.

6.4 The AnalyzeOnset Protocol

The AnalyzeOnset protocol allows the student to send his recording plus a file with the score to a web service that analyses the onset of the notes.

1. When the student enters the AnalyzeOnset scene he or she requests the analysis of a certain recording. This request is made by sending a message to the OnsetAnalyzer.

2. The OnsetAnalyzer will reply with a URL that links to an input form where the student can upload the score file and the recording.

3. When the student submits the input form, the files are uploaded and the OnsetAnalyzer is notified of the fact that the files have been uploaded.

4. The OnsetAnalyzer sends a message to the student to inform that it has received the files.
5. Finally, the web service will analyse the audio and the score file, and when this is finished the OnsetAnalyzer sends a message to the student containing another URL. This URL links to a page that displays the results of the analysis.

Note that in this scene the student uploads his recording again, even though he or she has already done that in the UploadRecording scene. The reason for this is that it simply allowed us to quickly integrate the several web services. In the future we will make sure that it is no longer necessary to upload the same recording twice.

6.5 The PeerDiscussion Protocol

The peerDiscussion protocol defines the way the student, the tutor and the peers discuss the recording of the student. It works as follows (see also Figure 20):

1. The student sends a link to the uploaded audio file to all others.

2. The tutor asks all peers for feedback.

3. When all peers have given their feedback, the tutor comments on this feedback.

4. The tutor explains the issues with the performance of the student.

5. The tutor asks all peers to suggest a solution to the student.

6. When all peers have given their suggestion, the tutor comments on the suggestions.

7. The tutor suggests a solution to the student.

8. The tutor and the student together discuss the issues and the solution.

9. After the discussion, the student leaves the peerDiscussion scene, to practice and make a new recording.

10. After the student has uploaded his or her new recording, he or she comes back to the PeerDiscussion scene.

11. The student sends a link to the new uploaded audio file to all others.

12. The tutor gives feedback on the new recording. If this the teacher is entirely satisfied the protocol ends. Otherwise, if the feedback is not entirely satisfied but the feedback is positive, the protocol goes back to step 4. If the tutor provides negative feedback the protocol goes back to step 7.
Conclusions

In this deliverable we have explained the work done during the first year of the PRAISE project for the tool created to build social learning communities.

We have introduced the basic concepts behind electronic institutions, that allows to describe the coordination rules for an agent community, and their implementation in a framework called EIDE.

We have described some of the necessary adaptations that we have made to the current EIDE framework in order to enable PRAISE users to design their own communities with their own norms in the form of electronic institutions, and in order to make electronic institutions work over a peer to peer network. This will allow a better scalability of the PRAISE platform increases privacy and makes the platform easier to maintain. Furthermore we have described a general method for integrating web services with electronic institutions, so that existing audio analysis tools can be integrated without much effort into the PRAISE system, while remaining under control of the electronic institution.

In order for human users to participate in the electronic institutions, we have developed a javascript tool that allows humans to connect to and take actions in electronic institutions. Moreover, it generates the user interface automatically. Finally, we have described a specification of an electronic institution that we have designed as the initial test case to test the new tools described in this deliverable and which is meant as a starting point for further development of the Music Circle website.

For the future, we need to add a mechanism that ensures the institution keeps on running if or more nodes in the peer to peer network fail, the capability of storing information between executions of an electronic institution. Also we need to improve the user interfaces that allow users to search for electronic institutions and to par-
participate in electronic institutions, because at this point they are not as user friendly as they could be. Furthermore, the user interface for EI participation should allow users to participate in more than one scene at the same time, to allow users to select a role when entering a new scene instance, and allow users to select a protocol when instantiating a new scene instance.

References


