Contract N° IST-034824

Workpackage 10

Sustainable Research Community Building in the Open Knowledge Space

Deliverable 10.7

Visualisation Service for P2P Infrastructure and EvESim based on Google Maps

Project funded by the European Community under the "Information Society Technology" Programme

Contract number: IST-034824
Project acronym: OPAALS
Title: Open Philosophies for Associative Autopoietic Digital Ecosystems

Deliverable N°: D10.7
Due date: August 2008
Delivery date: November 20, 2008

Short description:
This report gives a short overview of the code developed for visualising the OPAALS network topology and service distribution based on Google Maps. The visualisation component can visualise both real P2P system networks and simulated network topologies. Together with the visualisation, a network crawler component for the Soapod infrastructure was implemented as proof-of-concept prototype.

Author: SUAS (Thomas Kurz, Thomas J. Heistracher, Raimund Eder)
Partners contributed: SUAS, TUT
Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Author, Organisation**
0.1	22/10/2008 (first submission)	SUAS
0.2	06/11/2008 (submission for final check)	SUAS
1.0	20/11/2008 (final submission)	SUAS

Quality check
1st internal reviewer: Runa Sarkar (IITK)
2nd internal reviewer: Saulo Barretto (IPTI)
3rd internal reviewer: Paulo Siqueira (IPTI)

Dependencies:

Work Packages	• Beside the visualisation of a P2P infrastructure, the work documented in this deliverable has a close connection to the efforts in WP3. For presenting networks of whatever kind (social networks, SME networks, computational networks, etc.) to the public, the visualisation of the network is required and commenced in this work. The continuation of the efforts will be documented in WP3 in connection to the EvESim framework. • As the network crawler component documented in this report visualises real P2P network nodes as well, it has also relevance for the future development of any future kind of P2P infrastructure. For example it could work as a core-service for future P2P nodes. • Additionally to the connections to other workpackages, this work is strongly linked to T10.5, Visualisation of the OKS, and has also potential to be utilised by partners from the regional development. It is already planned to integrate the visualisation into the *Guigoh* platform as well.

Dependencies:

Partners	The whole consortium, especially partners IITK, IPTI, LSE, TI, and UniS in the above described Workpackages and Tasks could benefit from reading this deliverable and the online documentation, respectively.
Domains	Although this is a computer science deliverable in the form of code, the idea behind the visualisation of the network topology and the service distribution is especially important to the social science domain. Moreover, the concepts and models of other disciplines benefit from a visualisation of the network behaviour. Unfortunately, the limited resources at the moment just enable a first step for crawling and visualising the network topology. The chosen technologies and schemes are nevertheless chosen to be highly extendible for richer applications in future. Additionally, the definition of interfaces of the software was synchronised on a regular basis with partners TUT and IPTI. These online synchronisation meetings were documented and are summarised in the OPAALS wiki.
Targets	This deliverable mainly targets at everyone who is interested in the visualisation of networks based on Google Maps.

Contents

Executive Summary

The document at hand reports the activities in T10.10, P2P infrastructure visualisation. This task included the design and implementation of a service for 'crawling' and visualising network topologies. As the visualisation is important for simulations as well as for understanding the infrastructure itself, a universal visualisation component was developed which is able to visualise 1) real network topologies in the existing community in the example of one P2P infrastructure, and 2) the simulated network topologies coming from the EvESim framework (T3.8, T10.11). The following document and the related source code provide following functionalities:

- A *network crawler component* for crawling Soapod (`http://www.soapod.org`) nodes and persisting the topology in an XML file.
- Structure and schema for *persisting network descriptions* extensible in a XML file as well as considering asynchronous update functionality. All this was developed in frequent synchronisation with partners TUT and IPTI and is aligned with the visualisation pipeline principle developed by TUT.
- Visualisation and dynamic update of a topology visualisation in Google Maps with Pop-up windows for detailed node information.

The up-to-date documentation is available at `http://www.evesim.org`. The Javadoc and JavaScript in-code documentation can be found within the code base and will be adapted in synchronisation with the following activities.

Because this is a code deliverable and the documentation is available online as well as within the code base, this document acts just as the official document to report the dependencies and project specific tasks for D10.7.

1 Introduction

The EvESimulator provides a simulation framework for biologically inspired P2P systems – the so called Evolutionary Environment (EvE) as a part of a Digital Ecosystem. It focuses clearly on providing a framework for various simulations depending on a number of variables and not a distinct simulation itself. Although its focus is on the EvE, the EvESimulator simulates a DBE or – depending on the use case – Business-, Computing- or Social-Networks. Besides, the EvESimulator can be summarized as an collaborative platform for interdisciplinary research acting as a framework for understanding, visualising and presenting the DBE concepts to contributors.

For the analysis of non-linear and adaptive interactions that are usually too complex to be captured analytically, we use computer-aided simulations to specify the rules of behaviour for individual entities as well as their respective interactions. The simulation itself aims at simulating a multitude of individual entities and at exploration of the consequences of the given rules. As the individual entities are usually implemented in form of agents, the simulation of their behaviour and interactions is known as an agent-based simulation.

The particular innovation in the EvESim is the interdisciplinarity of contributors as well as the extendibility. For example findings from social science regarding the critical variables for SME behaviour are implemented in the simulation in order to provide a more realistic interaction behaviour. These settings can then be included in a computational simulation of transactions in a P2P network for example. As the simulation framework is very generic and open for extensions it offers also many options for cross-domain simulations. Additionally, these can exemplarily show differences of network behaviour for external stakeholders in an understandable way.

2 Visualisation Service for P2P Infrastructure and EvESim based on Google Maps

Before a network can be visualised, the data which will be visualised need to be collected, persisted and made available for the visualisation component. Therefore, this short report about the visualisation service for the OPAALS P2P infrastructure starts with a section about the crawler component used for searching the network. It continues with a section on the XML based file format which was defined as a joint effort of partner TUT and SUAS and concludes with a description of the visualisation component itself. As there is no definite decision on the standard infrastructure, the implementation of the visualisation service that is documented in this deliverable was kept as much as possible platform independent. In synchronisation with the OPAALS computer science coordinator, SUAS decided to build a proof-of-concept prototype of the visualisation service on top of the P2P system Soapod. A further reason for that decision was the work and evaluation of the P2P systems: 1) ServENT, and 2) Soapod along the EvESim activities documented in D3.5. The present code base is currently not implemented as a P2P service itself, but can be adapted to any kind of P2P environment.

2.1 Network Crawler

The crawling-algorithm's function is to discover – based on one node – all other nodes that are connected to the Peer-To-Peer network. Each node carries a list of neighbours, which is made up of URLs referring to peers directly connected to that node. In other words, a node looks up its directly connected neighbors' neighbor list, stores those in a network description list, and does look-ups to all the nodes in

the list again.

As a result, the list grows with every neighbour look-up to a node. This procedure is repeated as long as no new nodes disappear in a look-up. The implementation uses a recursive algorithm for reading the neighbours list. Each Soapod node comes with a so called *Administration Service* which allows to read the neighbours by an public method. In order to avoid loops and redundant entries, all new nodes are stored in a hash table, which holds all available nodes in the network at the end of the crawl procedure.

2.2 Network Description XML

The following network description in XML format was developed at SUAS with continuous feedback from partner TUT. For that reason, online meetings were held on a regular basis and documented in the OPAALS wiki. For reaching a maximum extent of reusability and flexibility for future extensions, SUAS decided to use the Eclipse Modelling Framework (EMF) for parsing and creating the XML files.

In the following, a simple *networkdescription.xml* file can be seen. For the visualisation, these XML files have to be in an XML folder for the visualisation service. The code – available in the download area at `htt://www.evesim.org` – includes a sample *networkDescription.xml* file as well as the appropriate folder structure.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/css" href="../xml.css" ?>

<networkDescription>
 <nodes>
  <node URI="office@fh-salzburg.ac.at" type="SME" name="SUAS">
   <name>SUAS</name>
   <!-- name is an extra tag - better human readable format -->
   <accessMethods>
    <accessmethod type="web" uri="http://thomaskurz.example.com" />
    <!--URI entpsr ID!-->
    <accessmethod type="servent" uri="http://123.1.21.3" />
   </accessMethods>
   <location lat="47.4759" lng="13.00"> Salzburg </location>
   <servicePool type="onOffer">
    <service id="032" uri="DBE:EvE:EvESim">
     <!-- Services do not need a URI necessarily,
          therefore the ID is the unique identifyer -->
     <name>EvESim</name>
     <description>EvESim, agent-simulation, Genetic algorithms</description>
     <accessmethods>
      <accessmethod type="SOAP" uri="http://dbe.fh-sbg.ac.at/~reder/evesim.jnlp" />
     </accessmethods>
```

```
    </service>
   </servicePool>
  </node>
 </nodes>
</networkDescription>
```

After a network visualisation is loaded by the JavaScript of the visualisation service, it is displayed in a Google Map. The visualisation will be described in the following section. In order to avoid reloading of the whole topology every time when changes are recognised by the crawler, an additional update file-structure was developed. Every X seconds, the JavaScript in the visualisation service reads the changes in the so called *update.xml* file. The changes are only applied, when the timestamp-tag of the *update.xml* changes. In the following, a short example of a *update.xml* file can be found, which is also available in the download section of EvESim:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/css" href="../xml.css" ?>

<networkDescription>
 <timestamp>1010</timestamp>
  <nodes>
   <node action="delete" URI="office@fh-salzburg.ac.at" type="SME" name="SUAS"></node>
  </nodes>
</networkDescription>
```

For using the EMF framework for writing and parsing XML data, first the structure of the XML data via an XML Schema (XSD) must be defined. The schema was modelled on basis of an example of such a network-description XML-File. After defining the structure, the XML Schema Definition must be mapped to an Ecore Model used by EMF, which represents the basis for the Generation Model (GenModel). Finally, the automatic code-generation tool of the eclipse Modelling Framework is able to generate the classes for instantiating elements and attributes of the Schema to serialize XMI- or XML-Files. Modifications in the parser can now be done by changing the Ecore Model in eclipse and the code generation for the parser classes is done automatically.

2.3 Visualisation with Google Maps

The visualisation component for a network with AJAX update routine contains six JavaScript files, two XML-Files, and one HTML file. These files and methods will be documented technically in the following.

2.3.1 JavaScript Object Array

The object array can be described as the main object of the application. In this array, each node will be saved with all data. This is necessary for a better Google Map handling. By saving all events as own objects, removing and adding of single lines, markers and descriptions will be much easier. Without individually addressed objects, the Map would have to redraw each object in each refresh. This could lead to serious performance problems.

2.3.2 XML-Files

As mentioned before, the whole data transfer with the crawler is done via two different XML-Files: 1) The *networkDescription.xml* file stores the whole information of all network nodes and their connections, access methods and service pools of a defined time. 2) The *update.xml* file stores the update data of any network node. Furthermore, it includes a tag called *timestamp*. This node contains the timestamp, repsectively time, on which the update XML-File was built.

2.3.3 JavaScript

The JavaScript code is divided into the following files:

xmlRead.js: This file contains the AJAX request routine and the `catchResponse` and the `catchUpdate` functions. These functions will be called after the AJAX request. Both of them start to read the XML content and call different functions for the respective content.

xmlGET.js: This file contains all XML functions for initializing the object array. The first function is called `getMain`, it sets up a new node-object in the object array and saves the main data like name, latitude, longitude etc. After finishing it calls three other functions for the three different subnodes of each node: 1) `getAM` (get access methods), 2) `getC` (get connections) and 3) `getSP` (get service pools).

xmlUPD.js: This file contains all XML functions for the update routine. All functions work similar to the `xmlGET` functions. The main difference is the additional reading of a new attribute called `action\verb`.

gm.js: This file contains all functions for the Google Map objects. Every method gets the data from the object array, creates the defined object (like a polyline, e.g. or a marker) and saves it in the object array.

2.3.4 The update routine

The application does everything without user interaction. The crawler just has to write the *update.xml* file in the valid format in the respective folder. The most important value in the update.xml file is the *timestamp*. Without incrementing it, the update routine will not start. To enable a node for updating, the user has to write the node in the update.xml file. The URI or any other identifier of each node is mandatory. After writing the node with its URI, the user has to fill in the *action* attribute to the respective node/subnode. The attribute can be set to three different values:

- delete: deletes the whole node and any subnodes
- update: changes all content from the node and the subnodes.
- add: adds a new node including the available subnodes. The action attribute itself can be set on the following four different nodes/subnodes

***node* tag**

```
<node action="delete" URI="R"></node>
```

***connection* node**

```
<node URI="office@muenchen.de">
 <connections>
  <connection action="add" to="R">R
 </connections>
</node>
```

***accessMethods* node**

```
<node URI="office@muenchen.de">
 <accessMethods>
  <accessmethod action="update" uri="R">R
 </accessMethods>
</node>
```

***service* node**

```
<node URI="office@tampere.fi">
 <servicePool type="onOffer">
  <service action="update" id="R" uri="R">R</service>
 </servicePool>
</node>
```

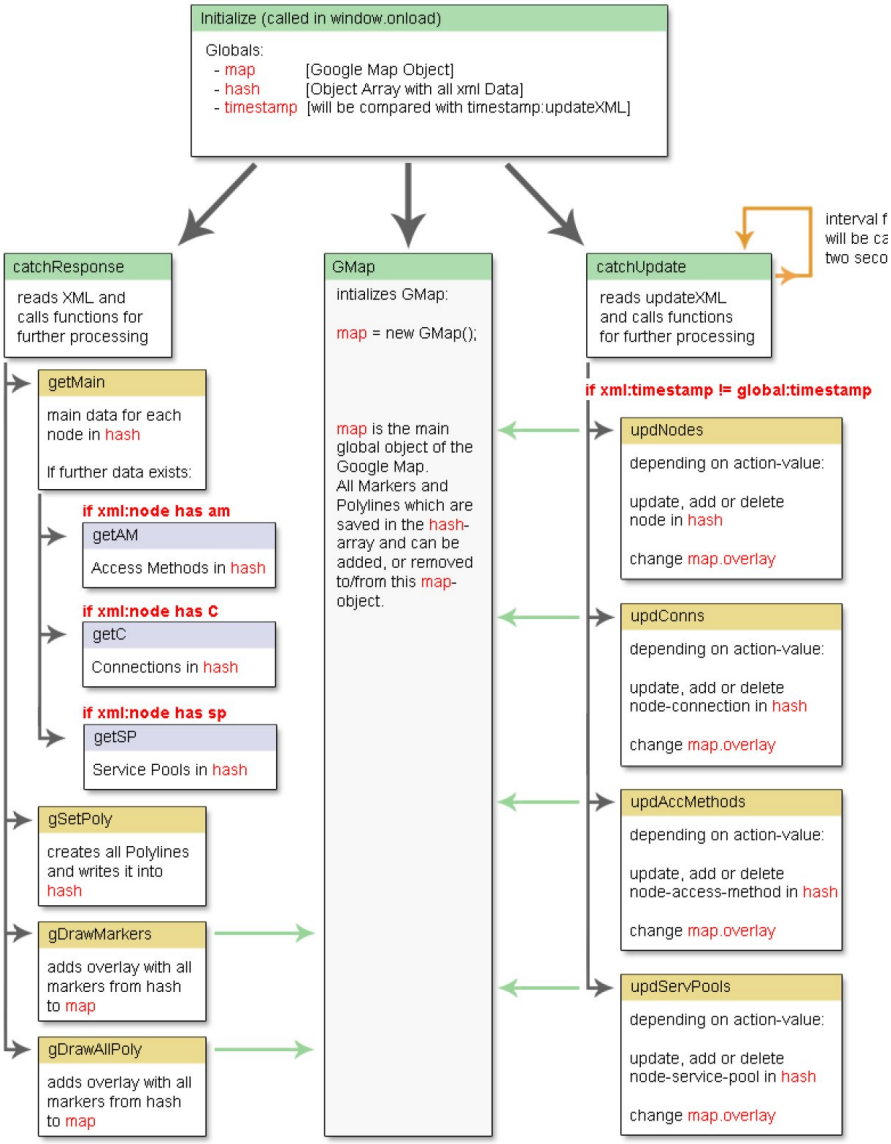

Figure 2.1: Application structure and activity diagram

2.3.5 Process

Initially the web application is opened in a compatible web browser. The web browser loads the *index.html* file in which all necessary JavaScript files and functions will be loaded or called, respectively.

The JavaScript body onload event calls an initialization function. This function needs three different globals to call any other function: `map`, `hash` and `timestamp`. As shown in Figure 2.1, `initialize()` itself calls three other functions.

The first function `GMap()` sets the global variable `map` to a GMap object.

The second function `catchResponse` reads the *networkDescription.xml* file. This happens only the first time, when the application is loaded. `CatchResponse` fills the hash array with all data from the *networkDescription.xml* file. In addition to that, it creates the `GLatLng` and `GPoint` object for every node and saves it in the hash, too. After reading and saving all data from the *networkDescription.xml* file, `catchResponse` calls three different subfunctions which all begin with letter *g*. The *g* stands for Google which should show that all these functions are related to the GMap object. 1) `gSetPoly` saves all available polylines to any node which has a connection. This data has to be saved after reading the other content, because a polyline needs two points (start- and endpoint). 2) `gDrawMarkers` steps in a for-loop through the hash and adds every node as a marker on the GMap. After that, 3) `gDrawAllPoly` does exactly the same. The only difference is that `gDrawAllPoly` adds the polylines in the hash to the map. This function has to be an own function because `GMap` only can add a polyline to two existing markers.

The third function `catchUpdate` is an interval function. This means that this function will be called by the JavaScript interpreter every two seconds. The function reads the *update.xml* file. The value `networkDescription:timestamp` is the most important one in this file. As shown on Figure 2.1, the function compares this XML value with the global `timestamp` value. If it does not differ from the global value, the function does nothing till the next time it will be called. If it differs, the function sets the global `timestamp` to the new value and calls all available update functions for all the nodes written in *update.xml*. The four functions jump to the parts of the update routine of the node. Depending on the update value, the function checks the hash array and deletes or changes the given values in it. If the update value is set to add, the functions add new variables/arrays in the hash.

3 Conclusion and Outlook

The current version of the visualisation component (see Figure 3.1) will be extended in WP3 and WP10 mainly. Summarising, the next steps are the following:

- Integration of visualisation components into *EvESim*
- Integration of visualisation components into *Guigoh*
- Propagation and search for additional application areas in other workpackages and tasks, e.g. use cases of partner IITK
- Extension of visualisation capabilities within WP3 and WP10, mainly connected to *EvESim*

Although there is no explicit funding for the extension of the current visualisation component, we try to extend and integrate the outcomes of this task in other workpackages and tasks.

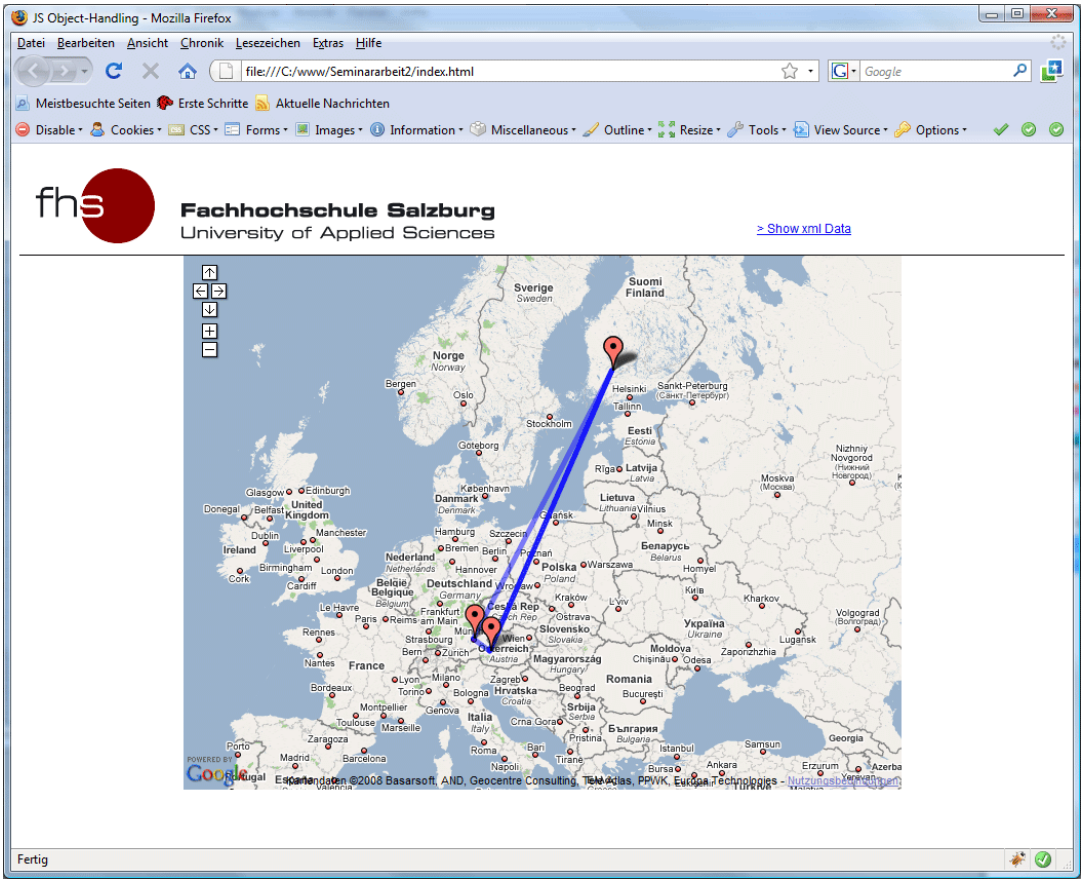

Figure 3.1: Current status of the Google Maps visualisation.