OPAALS PROJECT

Contract n° IST-034824

WP10: Sustainable Research Community Building in the Open Knowledge Space

D10.20 – Emulation and testing of OPAALS P2P infrastructure by utilisation of EvESim

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

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D10.20

Due date: May 2010

Delivery Date: July 2010

Short Description: In this deliverable the new version of EvESim and its potential to serve the OPAALS community as a multi-domain tool are described. Mainly it focuses on the new model driven simulation architecture, extended visualisation infrastructure and the frameworks new capabilities as SNA tool.

Author: SUAS (Raimund Eder, Thomas J. Heistracher, Thomas Kurz, Christoph Rücker), IPTI (Fernando Colugnati)

Partners contributed: SUAS, IPTI, IITK

Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Name, organization**
0.1	15/07/2010	Initial version
0.2	04/08/2010	Submission for internal review

Quality check

Internal Reviewers: Paul Krause (UNIS), Jason Finnegan (WIT)

Dependencies:

Achievements*	Complete EvESim architecture and simulation model redesign, Introduction of various Geo- and Social network visualization options, Simulation and Emulation use case, Data acquisition from various social networks, SNA capabilities introduced into EvESim
Work Packages	WP3, WP10
Partners	SUAS, IPTI, IITK
Domains	Computer Science, Social Science
Targets	
Publications*	C. Rücker. Use Case and Feature Extension for a Multi-Agent Simulation Framework (Diploma Thesis). Salzburg University of Applied Sciences, Austria, June 2010.
PhD Students*	None
Outstanding features*	Redesign of EvESim Architecture, integration of NASA WorldWind and Google Earth for geovizualisation, Using EvESim for SNA (Biotech / Ireland, Agropedia / India, Using EvESim for monitoring of infrastructure
Disciplinary domains of authors*	R. Eder (Information Technologies, Software and Systems Engineering) T. Heistracher (Information Technologies, Software and Systems Engineering, Biophysical Modelling) T. Kurz (Information Technologies, Software and Systems Engineering, Interpersonal Communication) C. Ruecker (Information Technologies, Software and Systems Engineering) F.A.B. Colugnati (Statistics and Social Network Analyst)

The information marked with an asterisk (*) is provided in order to address Recommendation n. 4 from the Year 2 review report

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit : http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

creative commons

COMMONS DEED

Attribution-Noncommercial-Share Alike 3.0 Unported

You are free:

to Share to copy, distribute and transmit the work.

to Remix to adapt the work.

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights..

Contents

Table of Contents 1

Executive Summary 2

1 Introduction 3

2 EvESim Architecture 6
- 2.1 Simulation life cycle . . . 8
- 2.2 EveSimulator Application . . . 8
  - 2.2.1 EveSimulator User Interface . . . 9
  - 2.2.2 EveSimulator Node Instances . . . 9
- 2.3 EveSimulator Workspace . . . 11
  - 2.3.1 EveSimViz Project . . . 12
  - 2.3.2 EveSimViz.Edit and EveSimViz.Editor Projects . . . 13
  - 2.3.3 EveSimViz.WorldWind Project . . . 13
- 2.4 Eve Component Model . . . 13
  - 2.4.1 EveComponent . . . 15
  - 2.4.2 EveSimulation . . . 17
  - 2.4.3 EveMonitor . . . 20
- 2.5 EveSimulator Support Classes . . . 22
  - 2.5.1 EveComponentRegistry . . . 22
  - 2.5.2 EveSimThreadGroup . . . 22
- 2.6 EveSimulator Distributed Session Management . . . 23
- 2.7 EveSimulator Reporting . . . 25

3 EvESimulator Visualisation 26
- 3.1 Google Earth Visualisation . . . 26
  - 3.1.1 EveSimulator Internal Web Container . . . 27

3.1.2 Static Content Servlet 28
3.1.3 Velocity Servlet 29
3.1.4 KML Servlet 29
3.2 NASA World Wind Visualisation 30
3.2.1 Notifications within the World Wind Prototype 31
3.2.2 Select-Mechanism 33

4 Simulation, Emulation and Testing 36
4.1 Simulation Conception 36
4.1.1 Simulation Model 38
4.1.2 Simulation Parameters 41
4.2 Document Editing Agent Conception 42
4.2.1 Agent Properties 44
4.2.2 Agent Behavior 45
4.3 Document Editing Service Conception 47
4.3.1 Service Setting 49
4.3.2 Service Behavior 50
4.4 Infrastructure Implementation 50
4.5 Document Agent Implementation 51
4.6 Document Editing Service Implementation 53
4.7 Simulation Data Analysis 54
4.7.1 Simulation Scenarios 55
4.7.2 Simulation Analysis 56

5 Acquisition of Social Network Data 65
5.1 Acquisition of Social Network Data 66
5.1.1 Guigoh and Facebook Data 67
5.1.2 Metric Calculation and Visualization 69

6 Evesim for Social Network Analysis Practitioners 71
6.1 SNA Practicing 72
6.2 A brief review on SNA Computational Resources 73
6.3 Evesim perspectives from a SNAnalyst 74

7 Related Activities 77
7.1 BioTec Visualization Ireland 77
7.2 Agropedia Social Network Visualization India 78

7.3 Monitoring 79
7.4 Flypeer Search Service 80
7.5 Gnutella vs Freenet Search by EvESim 81
7.5.1 Network Exploration 81
7.5.2 Semantic Search Simulation Model 82
7.5.3 Example Application 83

8 Conclusion **85**

Bibliography **90**

Executive Summary

This report and code deliverable gives a description of the final EvESimulator (EvESim) architecture and covers a variety of its technical implementation aspects in detail together with a representative emulation test case yielding performance measurement results. Some of the central features of the EvESim framework are map-based visualization, ease of use of configuration and deployment (EvESim bundle), data import from large real-world social networks (Guigoh and Facebook), concurrent instantiations, and overlay-concept for emulation capabilities. Social network practitioners are specifically addressed by a dedicated chapter describing EvESim from a high-level viewpoint in the context of three role models, which are *analyst*, *catalyst*, and *instance creator*; EvESim provides a framework where all these main stakeholders can collaborate effectively. In addition, a distributed document editing service is described as a use case for testing the underlying P2P infrastructure. Different search algorithms are utilised and several stress tests are performed in different topological setups. Finally some related application examples of EvESim are given.

The source code associated with this deliverable can be obtained from the EvESim homepage www.evesim.org under "EvESim Source Code" – the EvESim bundle for different platforms is available from the “Current Version” section ibidem.

1 Introduction

The popularity of social network websites is reflected in the activities of the multiagent research community during the last years. Foci have been for example the study of phenomena such as general user behavior, negotiation processes and recommendation systems. In parallel to the increase in connectivity and availability of web-based networking tools, the discussion on governance, underlying trust challenges and the misusage potential of personal data is ongoing; recent research on distributed and peer-to-peer based systems pointed out the benefits of avoiding central authorities in social networking.

The EvESimulator (EvESim) presents a framework enabling simulation, emulation and testing of social networks and distributed architectures in general by taking advantage of the benefits of a distributed multiagent system. We consider the research on distributed systems and architectures in conjunction with the avoidance of central authority as interesting fields of research and want to provide a simulation, emulation and testing framework, where stakeholders from different disciplines can work together in the conception, modeling and execution of simulations, emulations and tests.

The architecture of the present framework enables the simulation of distributed systems on top of an underlying infrastructure that is kept interchangeable. Thus a simulation is always carried out atop of one of the available infrastructures and thus enables emulating the behavior exhibited in such an infrastructure as well. When speaking about simulation cases or emulation cases in the following, the simulation of a concrete real-live scenario on a given infrastructure is meant.

EvESim does not intend to replace existing simulation frameworks. Although developed for the simulation and emulation of Digital Ecosystems, it evolved in the last years to a multidisciplinary framework for simulation, emulation and test of distributed systems. Here we want to stress the uniqueness of the approach for Social Network Analysts, e.g. IPTIs or NUIM social network groups (role model

analysts), for computer scientists and software engineers, e.g. IPTIs Flypeer P2P implementation group (role model *instance creators*), or SUAS as the provider of the EvESim framework for bridging these two groups of stakeholders for the benefit of SMEs (role model *catalysts*). Easy handling for non-technically experienced users, map-based visualization, combination of analysis parameters and customization of the framework are some of the main benefits.

Central goal of the catalysts is especially modeling and creation of interdisciplinary simulation cases. The catalyst is typically working closely with the analyst and the instance creator. He/she is bridging and also decoupling these two stakeholders in working on interdisciplinary emulation cases, which have a common interest for both. On one hand, by implementing an easy-to-use interface for adding variables and setting up simulation environments for the analyst, the catalyst is hiding the underlying infrastructure and technical implementation for the analyst and provides an easy-to-use front end also suitable for non-technically experienced people. Additionally, the catalyst needs to understand the overlapping interests of analysts and instance creators and needs knowledge about the framework implementation. On the other hand, the catalyst offers a technical interface to the instance creator, which allows plugging-in of various distributed infrastructures decoupled from a concrete data set or from a specific emulation and test scenario.

An instance creator is concerned with a test of a distributed infrastructure. Such typically service-based infrastructures can be classic web services, already available service containers which are connected by new P2P approaches or innovative P2P infrastructures. In order to test such systems, there specific test frameworks are available. If it comes to the comparison of such service approaches and the real-life testing, it is rather hard to set up proper testing environments. Here the instance creator can adapt to the interfaces of the framework and run already available test and emulation scenarios set up by the catalyst, or program new scenarios by extending the available agent model. The big advantages herein are: 1) the possibility of comparison between distributed infrastructures, 2) the ability to re-use simulation cases, and 3) the availability of topology and behavior data from SNA without having detailed knowledge in this field.

In the following, the architecture of EvESim is laid out with special focus on its distributed nature and on related session management, then its visualisation features are described before the more technical aspects of emulation and testing are dealt with. A separate chapter covers the technical aspects of SNA data import

from Guigoh and Facebook, before the high-level discussion of EvESim for the SNA practitioners is given. Finally, several world-wide dissemination and usage activities engaging EvESim are compiled for the benefit of the readers.

2 EvESim Architecture

The EveSimulator is separated into two different logical main components. The *EveSimulator User Interface* (EveUI) on the one hand is responsible for the management and visualization of simulations. The *EveSimulator Node Instances* (EveNI), on the other hand, are hosting the simulations in a distributed manner. This section covers the logical architecture of these two components. Technical details will be discussed in Sections 2.2.1 and 2.2.2.

Figure 2.1 provides an overview of the components and how they are interconnected. As can be seen from the figure, the EveUI is the central part of the simulation and connected to all actors in the simulation. It acts as the management console and *window* to the simulation cluster, as the current state of the simulation model is replicated to the user interface periodically.

Each EveUI is connected to a number of EveNIs. The links are configured within the simulation setup. The communication between the components is achieved by HTTP calls and XML Web Services [1]. The HTTP calls are used to send commands to the individual EveNIs. The XML Web Services are used to send the simulation setup to the individual EveNIs and to gather the information from the EveNI's message queues. In order to establish this communication the EveNI's have to be reachable via an IP address. Each EveUI is connected to a number of EveNIs. The links are configured within the simulation setup. The communication between the components is achieved by HTTP calls and XML Web Services [1]. The HTTP calls are used to send commands to the individual EveNIs. The XML Web Services are used to send the simulation setup to the individual EveNIs and to gather the information from the EveNI's message queues. In order to establish this communication the EveNI's have to be reachable via an IP address. Within the EveUI a synchronization service is responsible for collecting the EveNI's messages and for dispatching them locally to the simulation components. More details about the communication can be found in Section 2.2.2 below.

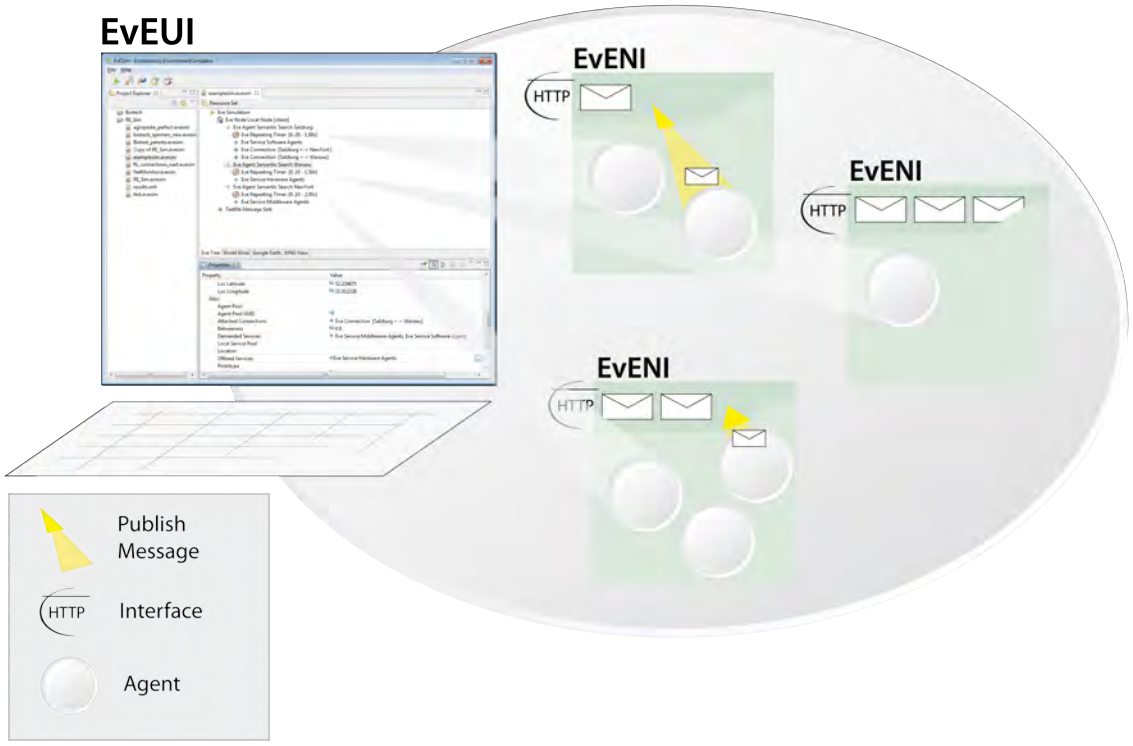

Figure 2.1: EveSimulator logical architecture.

Not all users can access an EveNI cluster. Nevertheless, users can run simulations using two additional options: The first option is to configure nodes as *local*. In this case the actors are running directly within the model of the EveUI. The advantage of this approach is that simulation setups can be easily tried out. The disadvantage is, that the simulation behavior can vary, as the distributed aspect of the simulation is skipped in this case.

To tackle the disadvantage, the second approach can be used. It is possible to start an EveNI within the EveUI. The nodes in the simulation setup have to be configured to run remote, but with the local EveNI as a network endpoint. Although this local EveNI is running within the same JVM as the EveUI, the full network and messaging stack is used, providing a more accurate test of a simulation setup.

2.1 Simulation life cycle

The first step of the simulation life cycle is to create a simulation setup within the EveUI. Each simulation gets a unique session ID so that different simulations can be run on the EveNI instances simultaneously. More information on how to setup a simulation can be found on the EvESimulator homepage [2].

When the simulation setup is complete, the simulation workflow is started by invoking the *Start Simulation* command. This command starts the synchronization services in the EveUI for the simulation. This services retrieve the messages from the EveNI message queues and dispatch them to the local message receivers.

After that, the simulation setup is sent to each EveNI individually which concludes the simulation setup phase.

After the simulation setup is complete, the EveUI triggers the simulation start by sending the appropriate command to the EveNI. The EveNI starts all timer instances which are responsible for executing the agents. (This is described in more detail in Section 2.4.2).

After a simulation run is complete, the stop simulation action triggers the shutdown of a simulation. First, all agents' timers are stopped. After that, the synchronization services are stopped as well. Finally, all remote sessions are deleted on the EveNIs.

In the following, an overview of the plugins developed within the EveSimulator efforts is given. After that the component model is described in order to provide an overview of the distributed EveSimulator framework. Thereafter, support classes are described and, finally, a section about distributed session management concludes this chapter.

2.2 EveSimulator Application

As stated above, the EveSimulator described in this paper consists of the EveSimulator User Interface (EveUI) and the EveSimulator Node Instances (EveNI).

The EveUI is used for the setup and management of a simulation. If the simulation is performed locally, only the EveUI has to be started and the simulations are performed within the application instance.

If the simulation has to be distributed, individual EveNI have to be installed and started within the network. These nodes are then configured within a simulation in the EveUI. The user interface is responsible for the distribution of the simulation

setup data, the management of the simulation and for the message distribution between the nodes and the user interface.

The following sections further describe these two components.

2.2.1 EveSimulator User Interface

Figure 2.2 shows the EveSimulator User Interface (EveUI).

The left part (1) is the project explorer. This view is used to manage projects containing individual simulations. A simulation must be part of a project, as this is enforced by the Eclipse framework. If a new simulation is created or an existing simulation is double-clicked, it is opened in the EveSimulator Model Editor.

The EveSimulator Model Editor, situated in the upper right part of the figure (2), shows the components of a simulation in a hierarchical form. This editor can be used to add, remove or re-arrange EveSimulator components in order to create a simulation setup. The possible components are described in Section 2.4 below. Each of these components in the editor has individual attributes. These attributes can be monitored and changed using the property editor.

This property editor, which can be seen in the lower right part of Figure 2.2, shows all attributes for a selected component. With the selected component, the attributes shown in the property editor can change, as e.g. a *EveSimulation* component does not have the same attributes as an *EveAgent* component. Additionally the attributes are categorized in order to provide a leaner user experience. More information on the component's attributes can also be found in Section 2.4.

Using the View Tab Selector, marked as (2a) in the figure, the user can switch between different perspectives. The Eve Tree shows the simulation data in a hierarchical form. The World Wind and the Google Earth tabs switch to the prototypical geovisualisation views described further in Chapter 3. The JUNG view visualizes the data as node graph using the Java Universal Network/Graph Framework (JUNG) [3] framework and is further described in [4].

2.2.2 EveSimulator Node Instances

An EveSimulator Node Instance (EveNI) is started by running Java with the *EveServerNode*[1] class as starting point. Optionally the listening port can be set using the *port* parameter (e.g. port=8081). If the port is not manually set, the default port 8080

[1] *org.evesim.EveServerNode*

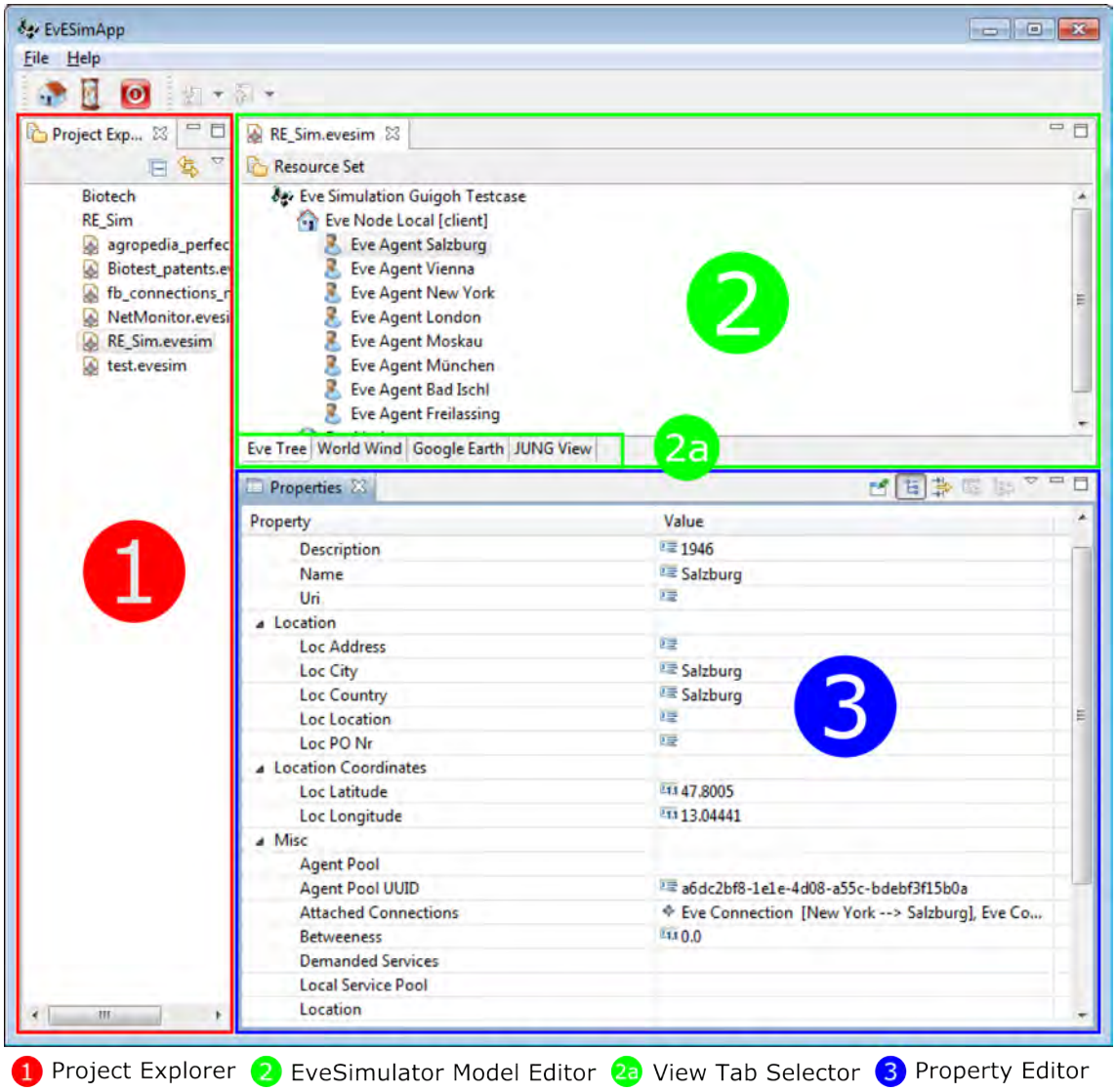

Figure 2.2: EveSimulator GUI.

is used. As soon the EveNI is started, it is available using the HTTP protocol and ready to be used by the EveUI.

There are five possible commands which can be invoked using the URL:

```
http://{node address\}:{port}/eve/{session id}/{command name}
```

The parameter *command name* must correspond to one of the commands listed in Table 2.1.

Init is used to initialize a simulation session, start the session's message queue and send the simulation setup to the node. This is the only HTTP-POST request, as the simulation setup is sent to the node as XMI data.

Start starts all agents deployed to the EveNI and *Stop* shuts down all agents.

Delete is used to delete all data belonging to a session and required for a clean start of a simulation.

Using the *msg* command, the messages waiting in the message queue are returned as XMI serialized content. This command is used by the EveUI's synchronization service to gather all data waiting to be pulled from the EveNI cluster by the EveUI.

Command	Type	URL
init	POST	`http://{node address\}:{port}/eve/{session id}/init`
start	GET	`http://{node address\}:{port}/eve/{session id}/start`
stop	GET	`http://{node address\}:{port}/eve/{session id}/stop`
delete	GET	`http://{node address\}:{port}/eve/{session id}/delete`
msg	GET	`http://{node address\}:{port}/eve/{session id}/msg`

Table 2.1: List of EveNI commands

More implementation details of the messaging subsystem are provided in Section 2.6.

The previous sections provided an overview of the user perspective. The following sections explain the structure of the software projects of the EveSimulation framework.

2.3 EveSimulator Workspace

The EveSimulator workspace contains all EveSimulator plugin projects, what is depicted in the project explorer screenshot in Figure 2.3. As can be seen, the EveSimulator is separated into four different projects: (i) the EveSimViz project, (ii) the EveSimViz.Edit project, (iii) the EveSimViz.Editor project and (vi) the

EveSimViz.WorldWind project. These projects are discussed in the following in more detail.

2.3.1 EveSimViz Project

This first project contains the basic simulation model and source code.

The *src_gen* folder contains the source code based on the EveSimulator model. The class stubs, including all attributes and the method signatures are automatically created. Developers then can add the relevant source code into the generated method bodies. To prevent the source code generation utility to overwrite customized code, these methods have to be marked using Javadoc tags [5], [6].

Besides the automatically generated stubs, there is a separate *src* folder containing manually maintained source code files to separate generated source from developed source code. References to classes maintained in the other folder can be created as it was only one folder containing all files.

The following three nodes in the project explorer contain the individual dependencies of the project: (i) referenced Java libraries configured for the project, (ii) a reference to the Java Runtime Environment (JRE) foundation classes, and (iii) references needed to compile this project as an Eclipse plugin.

The *icons* folder is automatically created to contain the icons representing the model instances in the editor.

The *lib* folder contains all referenced libraries as jar files which are bundled together with the plugin.

The *META-INF* directory contains meta-information for the Eclipse plugin in form of a manifest file.

The *model* folder contains all model related files. The most important files are *evesim.ecore* and *evesim.genmodel*. The *evesim.ecore* file contains the simulation model as explained in Section 2.4. Additionally the *evesim.genmodel* file provides information needed for the EMF source code generation facility, such as descriptions and meta-data for the individual model attributes. These two files are used as input to automatically generate the source code stub for the model classes contained in the *src_gen* folder.

The *network_description.ecore* and *network_description.genmodel* files represent the model used to import legacy EveSimulator data and data from other partners. The *network_description* model was created for the previous EveSimulator version and is described in [7] and [8].

The *visualisation.** files are deprecated and will be deleted from the project repository.

The *resources* folder contains resources used for the Google Earth visualization which is further explained in Section 3.1.

Again, the *schema* folder contains RCP relevant information which is not further explained in this document.

The files *build.properties*, *plugin.properties* and *plugin.xml* are plugin related files as well, and are contained in all projects.

It is important to mention the *EveSimViz.product* file, which is used to create, export and start such an application bundle. The exported bundle then can be given to end-users in order to start the simulator independent of the development environment.

2.3.2 EveSimViz.Edit and EveSimViz.Editor Projects

These two projects contain the source code for the model editor depicted in Figure 2.5 and Figure 2.6. This editor is separated into two individual plugins. The first plugin, *EveSimViz.Edit*, contains the general source code for creating the property editor (which can be seen in Figure 2.2 in the lower right part) of the individual EveComponent instances. Whereas the second plugin, *EveSimViz.Editor*, uses all other plugins to build a complete user interface for modeling simulation setups and controlling simulations. Within this plugin, the *EvesimEditor* class integrates the visualization views as individual tabs into the editor. This, for example, can be seen in Figure 3.3, between the WorldWind view and the property editor.

2.3.3 EveSimViz.WorldWind Project

The WorldWind visualization is, contrary to the Google Earth view, separated from the editor as an own plugin. This project is explained in more detail in Section 3.2 below.

2.4 Eve Component Model

The data model within the EveSimulator is defined using EMF tools within Eclipse. This section outlines the individual EveSimulator model classes, on which the plug-ins – described in Section 2.3.1 – rely on. The simulator's source code is separated

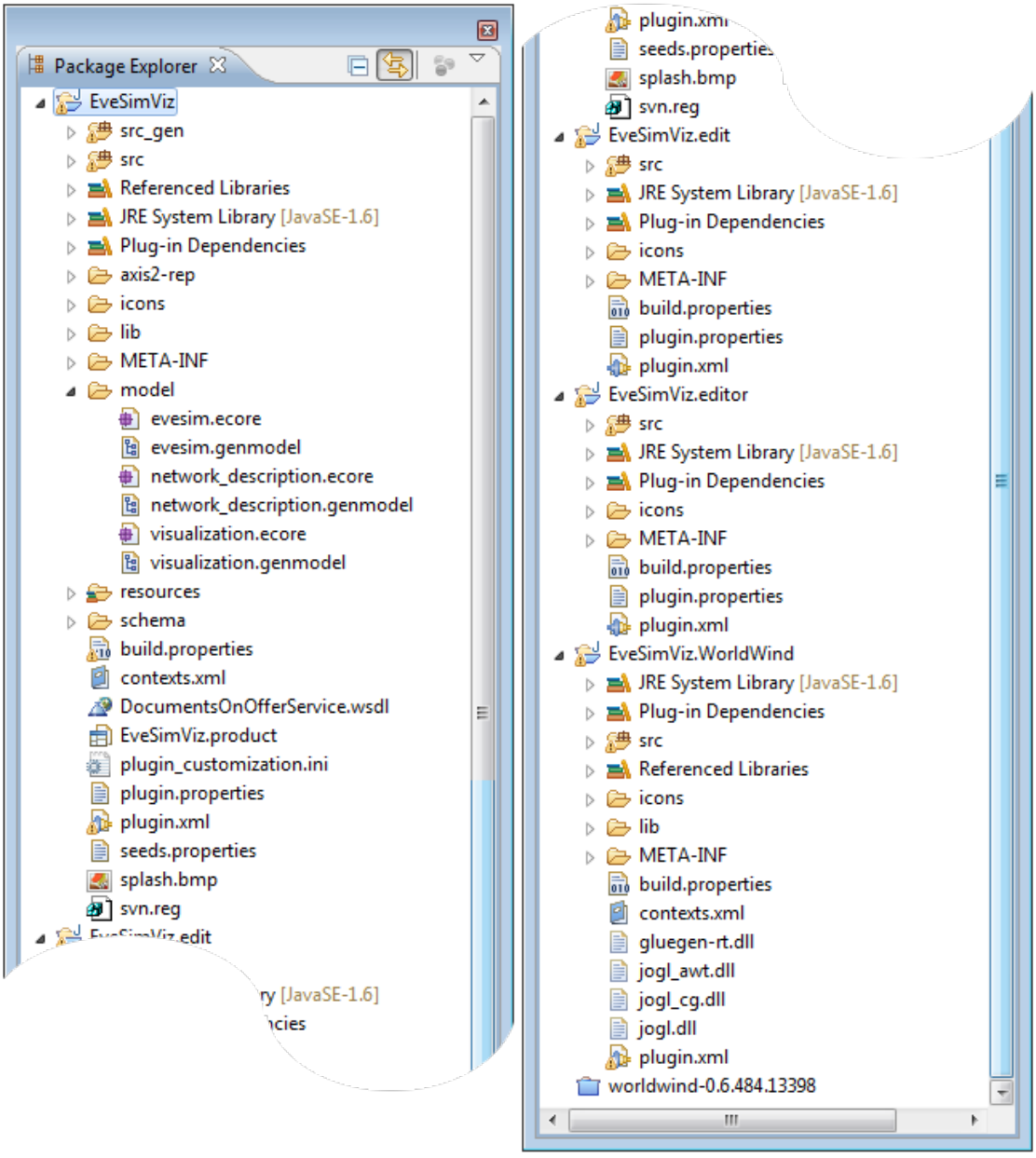

Figure 2.3: EveSimulator Workspace.

into (i) automatically generated and (ii) manually generated source code. Automatically generated code, or more precisely code implemented within generated stub classes, is organized in a dedicated *src_gen* folder. All other code is stored in a separated *src* folder. The following sections describe the functionality of the classes generated from the EveSimulation model. From Section 2.5 on, selected support classes, not existing within the modeled packages, are explained. The overall class hierarchy of the component model is depicted in Figure 2.4.

2.4.1 EveComponent

The class *EveComponent* is the common denominator of all model classes within the simulation framework. It encapsulates all basic information relevant for all inherited classes such as ID, name, description, session information and geospatial information. In addition to that, the fundamental attributes and methods for hierarchical clustering of EveComponents are implemented. Furthermore, the relevant interfaces for serialization and property change management are included at this point as well. Finally, the simulation relevant abstract method *execute* is defined. Inherited classes can override this *execute* method, which is used to implement the agent behavior. The relevance if this method will be explained in Section 2.4.2.

As mentioned before, EveComponent classes are designed to support hierarchical structures of all other classes inherited from EveComponent. However, for a structured simulation, this flexibility is limited by the user interface. An example of such a hierarchy, corresponding to a valid EveSimulator model, is depicted in Figure 2.5, which is a screen shot of a simulation setup. All *Eve Model* classes, described in the following, have EveComponent as their base class.

Attributes and methods needed for connecting the components are integrated and explained in Section 2.4.2.

At the root of a simulation or monitoring structure, two different child classes of EveComponent are allowed: (i) EveSimulation and (ii) EveMonitor. The valid session ID is stored in the root component. If the getter-method of the session ID is invoked on an EveComponent, the lookup is delegated to the root component.

The following sections explain EveSimulation and all relevant sub-classes of EveComponent and how they can be structured to form a simulation case. EveMonitor and all related sub-classes of EveComponent used to structure monitor cases are described from Section 2.4.3 onwards.

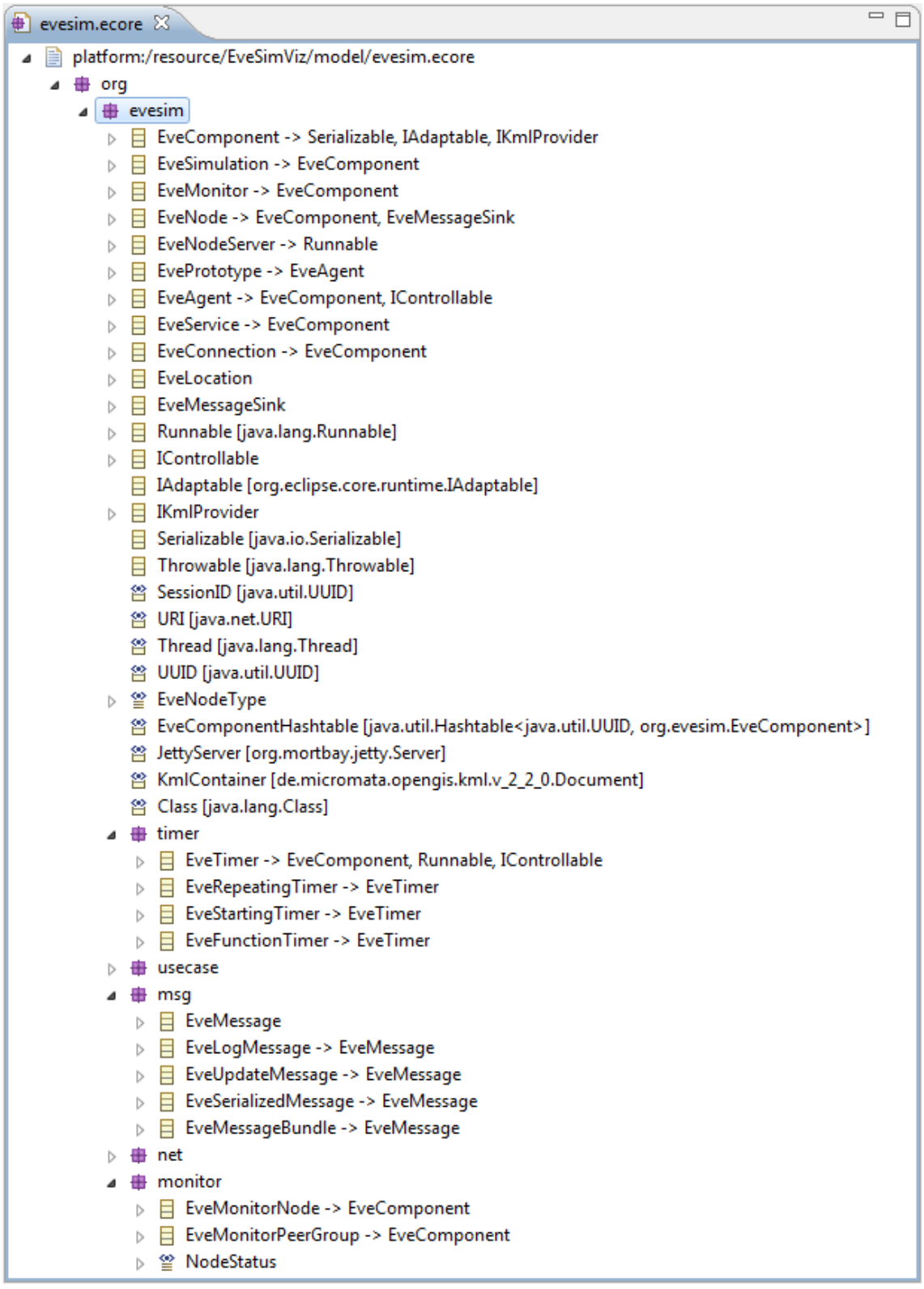

Figure 2.4: EveSimulator Component Model.

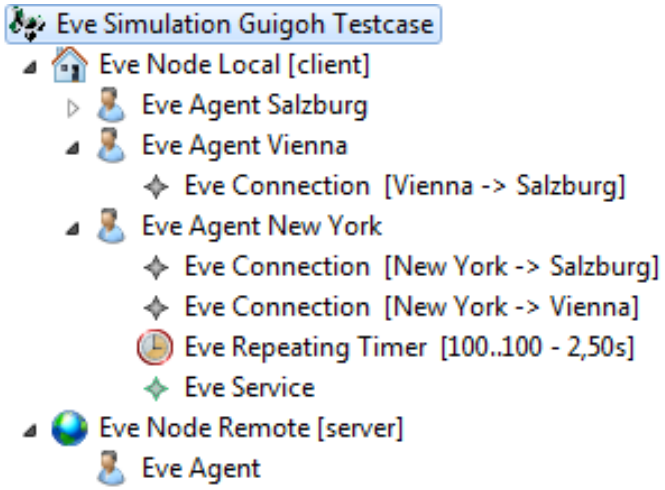

Figure 2.5: EvESimulator Class Hierarchy.

2.4.2 EveSimulation

EveSimulation is the first of two possible root elements within the EveSimulator and serves as the basis of a simulation setup. The second possibility is to use EveMonitor as root, which is described in Section 2.4.3. *EveSimulation* contains the methods to control a simulation as well as all necessary parameters to run the contained simulation. Instances of EveNode or EvePrototype are the only allowed children of an EveSimulation instance:

EveNode

Directly below the EveSimulation root instance, an unlimited number of EveNode instances form the physical network of nodes within the simulation network can exist. Each EveNode can contain an unlimited number of EveAgents. An EveNode can be either *local* or *remote*. Agents – attached to local EveNodes – will be executed within the local virtual machine, whereas agents – linked to remote EveNodes – will be sent to independent network nodes connected using physical network connections. For this to work, EveNode contains all network related attributes, to configure such a network node. For details on local versus remote nodes, see Section 2.

EvePrototype

Parallel to EveNode entries, the EvePrototypes act as templates for EveAgents. Each EveAgent can be linked to an EvePrototype in order to centrally configure individual properties of these agents. This enables mass configuration of agents of different types without having to change the properties individually. EvePrototype contains the same attributes and possible sub-elements, which are explained further in Section 2.4.2.

EveAgent

EveAgent itself contains the basic attributes of agents within a simulation. The actual behavior is encapsulated in subclasses of EveAgent. For this mechanism to work, the abstract *execute* method of EveComponent has to be overridden. Additionally an EveTimer instance has to be added as an EveAgent's child. The purpose of such a Timer is to trigger the execute method using an individual algorithm, which is explained in Section 2.4.2.

As mentioned above, EveAgent is intended to be an abstract class as well. The real agent behavior is implemented in subclasses of EveAgent. To do that in an orderly way, these agents are implemented in individual packages within the *use case* package. Currently two different use cases are prototypically implemented: (i) a Semantic Search use case and (ii) a Social Network Analysis use case.

EveAgent allows three different types of children: (i) EveConnections (see Section 2.4.2), (ii) EveServices (see Section 2.4.2) and (iii) subclasses of EveTimer (see Section 2.4.2).

EveConnection

Instances of EveConnection represent connections between different agents. Each EveConnection is added as a child to an agent (or prototype) which itself serves as the origin of a connection. The destination of such a connection instance is a reference attribute within the EveConnection object. By setting a *directional* flag, the connection can be switched between unidirectional and bidirectional. As connections should be accessible from both ends of the connections and connections can be attached to all EveComponents, additional fields and methods in the EveComponent class support the connection management. The attribute *attachedConnections* contains references to all EveConnections referencing a component. These so called

bidirectional opposite references are automatically managed by the EMF.

By invoking the *getConnections* method of an EveComponent, all valid connections for this component are extrapolated. This means that all connections originating from this component *and* all bidirectional connections pointing to this component are returned in a list. As this list is generated on each call, this method should be treated as a performance bottle neck and therefore should be used with care.

The *isValid* method is used to check the validity of the connection and performs sanity checks on the parameters and references. This method is mainly used by the visualization components and the previously mentioned *getConnections* method.

Another important method of the EveConnection class takes one EveComponent as an argument and returns the opposite component of this connection. This method is named *getOpposite* and is used to write generalized code without taking care if this is a connection from one component to another or a bidirectional component pointing in the reverse direction. If the provided component is not part of this connection a *RuntimeException* is thrown.

Additionally to the destination reference, all connections also contain information about strength and type. Strength is a floating point number, whereas type can be assigned custom string. Agents can (but may not) adapt their behavior according to these two attributes. Connections can be enabled or disabled using an additional attribute, named *connected*.

EveService

EveService, as well as EveAgent, serves as an abstract base class to be overridden for each individual use case. The corresponding, use case dependent, EveAgent is responsible for using the service instances. This can be achieved using two different approaches.

First, the execute method (inherited from EveComponent) can be overloaded. In that case, the built-in mechanism for service execution can be used. Services implemented this way, can easily be used in custom use cases. As a disadvantage of this approach, no parameters can be provided to the execute method and results cannot be returned to the agent.

Second, a custom method can be implemented in the service class as well. By following this approach, parameters can be provided to the method and a result can be returned to the agent, executing this custom method. Using such a custom

method, agents have to be adapted accordingly, in order to leverage the service.

EveTimer

For executing the agents, sub classes of the abstract EveTimer class are used. This abstract EveTimer contains nothing more than basic functionality to manage the timer's Threads, as well as the start and stop mechanisms. The purpose of timer instances is to invoke the *execute* method of the timer's parent object. To enable different scheduling behaviors on the one hand, and to get the possibility to enhance the simulation with custom behaviors on the other hand, this modular approach is used.

At the time of writing, the following EveTimers are implemented:

EveRepeatingTimer This timer is mainly for testing purposes. Besides the number of iterations, the sleep time between the executions can be configured.

EveFunctionTimer This timer has been created for the Document Editing Simulation Case. It enables execution interval configuration with mathematical functions and is describes in greater detail in section 4.4.

Simulations based on the EveSimulator framework are built on top of the model classes just described. To support network monitoring services as well, the following classes are used.

2.4.3 EveMonitor

Analog to EveSimulator, EveMonitor serves as root component for a network monitoring tree, consisting of peergroups and nodes. An example for such a monitor configuration is depicted in Figure 2.6. As with the simulation classes, the monitoring classes are generalized and can be adapted to the user's needs with specific implementations. Monitoring functionality is currently implemented only for the Flypeer [9] peer-to-peer service infrastructure, which was developed within the OPAALS research project [10].

For each EveMonitor model, a dedicated daemon thread is started in the background, which periodically invokes the EveMonitor's execute method. This execution is propagated via the EveMonitorPeerGroup to all EveMonitorNodes. The period of invocation can be set via the sleepTime property of the EveMonitor instance.

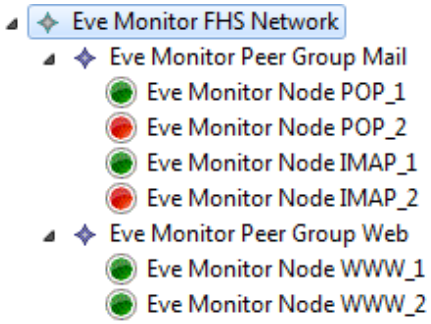

Figure 2.6: EvEMonitor Class Hierarchy.

EveMonitorPeerGroup

An EveMonitorPeerGroup is used to contextually combine nodes that logically belong together. For example, nodes hosting technically related services can be viewed as existing in the same peergroup. Individual nodes can belong to more than one peergroup, as these nodes can host unrelated services as well. Besides for structuring purposes, the peergroups do not encapsulate any functionality. On the invocation of the execute method, the execute method is invoked on all child nodes. The following section explains the implementation of the monitoring within the EveMonitorNode classes.

EveMonitorNode

Using EveMonitorNode instances, administrative users can configure nodes to be monitored within the network. This is achieved by adding an instance into the monitoring tree (such as in Figure 2.6) and configuring a network endpoint address. At the current state, only basic status information is managed within the EveMonitorNode class: whether the node is of status *OK*, *Fail* or *Recovering*. The node status *OK* indicates that the node is working correctly. If the monitor service was not able to contact the node to monitor, the status is changed to *Fail*. The node status *Recovering* is used if a node can be contacted that was in a *Fail* state before. Therefore, *Recovering* indicates that a node – although working – has not proved it's reliability yet. After a configurable amount of time in the *Recovering* state, the node status is updated to *OK*. More status possibilities can be added by manipulating the NodeStatus enumeration within the Eve Model. This approach is not restricted to Flypeer networks, as the EveMonitorNode can be used as a parent class for other

monitoring purposes as well.

In the default implementation, the *execute* method tests the node configured via the *netURL* attribute and sets the *nodeStatus* attribute accordingly.

The previous sections provided an overview of the model used within the EveSimulator, which is only a part of the application. A selected set of classes is described in the following sections.

2.5 EveSimulator Support Classes

Besides the previously described model classes, there are many additional support classes. In this section, a selected number of these classes are explained in order to provide an implementation overview.

2.5.1 EveComponentRegistry

As explained in Section 2.6, model updates are sent from the individual agents to the user interface using a message queue. If an attribute in the model changes, a notification is sent to the *EveSimServerSession* object which generates an *UpdateMessage*. When the *EveSimClientSession* receives such an *UpdateMessage*, the receiver address is used to find the destination component in the *EveComponentRegistry*, and the corresponding attribute is updated on the client side.

The *EveComponentRegistry* tracks all *EveComponents* within one simulation. This implies that each session has it's own *EveComponentRegistry* instance to separate these sessions. To keep an properly indexed registry, the session's root *EveComponent* has to be provided. If an component is not found in the registry, a new indexing cycle is triggered by the registry itself. If after that re-indexing the component is still not found, an Exception is thrown and the underlying operation is aborted.

2.5.2 EveSimThreadGroup

The class *EveSimThreadGroup* is the key to the remote Exception management within the simulation framework. During the initialization of a new *EveSimServerSession*, the session's own *EveSimThreadGroup* is created. All actors in the simulation are then part of this thread group. As this class extends the *ThreadGroup* class, the *uncaughtException* method can be overloaded. This method is invoked

by the VM, when an Exception is thrown that is not caught within the thread's *run*-method. This so-called uncaught Exception is then passed on to the *EveSimServerSession*, which uses the simulator's messaging mechanism, to generate a log message which is passed to the simulation user interface.

An additional benefit of collecting related threads into one thread group is that it is easier to separate the threads of different simulations during debugging sessions as well as to have better notification of all threads with one method call.

2.6 EveSimulator Distributed Session Management

One of the key features of the EveSimulator is the possibility to create distributed simulations in order to test infrastructure software in real-world scenarios. The EveSimulator framework provides an infrastructure-independent component model, that can be adapted to support different platforms. To support such a new platform, only the agent classes have to be adapted accordingly, as the simulation's control mechanism is independent of the individual infrastructure code.

The distribution architecture is depicted in Figure 2.1. As can be seen from the figure and the text above, the EveSimulator user interface is used to control the distributed simulation. Additionally, the user interface is capable of hosting nodes to support simulations without access to node clusters.

In case simulations are running in a distributed manner, session classes on both ends are responsible for controlling the simulation. Additionally to that, these session objects take care about distributing messages and synchronization of the model data.

These dedicated session objects are responsible for managing distributed simulations. Due to this approach, different simulations can be hosted simultaneously on a cluster of nodes. Each session is identified by a unique session ID following the IETF standard for Universally Unique Identifiers (UUID) [11].

On the client (user-interface) side of such a simulation, the *EveSimClientSession* object is responsible for the management of one simulation session. This includes dispatching messages to the remote nodes as well as deploying the simulation model within the node cluster. Additionally, individual *EveSimSyncServices* are started in order to collect all simulation control messages for the dedicated session within the network.

To setup a remote node, an instance of *EveNodeServer* has to be started. The

EveNodeServer acts as a web service offering the management calls of a distributed simulation. For each call, the session ID is required. The first call must be the initialization call for a session, including the node's model data. If control messages for an uninitialized session are invoked, an error is returned to the caller. With the initialization, a *EveSimServerSession* is created. From then on, all control messages are sent to the session's *EveSimServerSession* object, which itself acts accordingly. Additionally to that, a message queue is instantiated there, which is polled by the client's *EveSimSyncServices*.

The message queue and the corresponding sync services are developed to support subclasses of the abstract *EveMessage* class. The abstract EveMessage forms the base model including data for sender, receiver, session and timestamps.

The following message types are supported at this point:

EveLogMessage. A plain-text message which includes a message and a detailed description in two separate fields. This message, for example, is used when a Exceptions occurs within a simulation on the remote node.

EveUpdateMessage. Changes in the simulation model on the remote node are replicated to the controlling client using this type of message. The changes are tracked and then replicated using the EMF internal methods. Therefore it is possible to extend all EveComponents and rely on this replication mechanism. This message includes fields for the identification of the changed attribute as well as old- and new-values.

EveSerializedMessage. This message type can be used to deliver all serializable Java objects, and can be used by custom agents in order to report back to the interface.

EveDataMessage. For reporting purposes EveDataMessages are sent to dedicated message sinks. Further details can be found in Section 2.7.

EveMessageBundle. As it is not efficient to individually deliver each single message, messages are collected and sent as message bundles.

New message-types can easily be added by extending the EveMessage class (or alternatively one of the classes described above) within the *org.evesim.msg* package.

2.7 EveSimulator Reporting

To gather simulation results, a very flexible message subscription model is used. This approach needs senders, messages and receivers. Sender can be any component within the simulation. The sender has to create an instance of *EveDataMessage*, which contains a Hashtable containing Key-Value pairs and therefore can be used to transport arbitrary data. The message is serialized to XMI and then submitted to a message queue. The EveUI collects the messages and then dispatches them to message sinks.

These message sinks have to be configured in the simulation setup and implement the *EveMessageSink* interface. All sent messages are dispatched to the message sinks. These sinks are responsible for the processing of the messages. This enables developers of use-cases to implement their individual message sinks for e.g. different result calculations.

The EveSimulator offers the *EveFileMessageSink*, in order to generate comma separated value (CSV) text files, which for example are used to save the results of the Semantic Search use case. This message sink takes the first received data message as a reference message. The fields in this reference message determines the structure of the produced output file. All fields existing in the reference message are written as header field descriptors into the file. From all subsequent data messages exactly the same fields are gathered and written to the file in the same order to produce consistent results. If, however, the structure of the data message changes, additional values are omitted and missing values are skipped (leading to two or more field separators next to each other, indicating missing values).

Due to this approach, this message sink is very generic and can be used with any message producer. If this sink is not specialized enough, individual message producers can easily be developed and integrated by creating a new class implementing the *EveMessageSink* interface.

3 EvESimulator Visualisation

Based on the Eve Component Model described in the previous chapter, geospatial data can be used to generate visualizations in order to provide a better overview to the user. In this Chapter, two different approaches based on widely spread geovisualisation platforms are described and technical aspects of the related prototypical implementation are discussed. After giving an overview of the components and how they are related together and used in the context of EvESim, some limitations are dealt with as well.

3.1 Google Earth Visualisation

Google does not offer an Eclipse compatible widget for the integration of Google Earth [12] within an RCP application. However, for the integration of Google Earth, the official Google Earth Plugin [13] (Version 5.2.0.5932) can be used. Using this plugin, an Earth instance can be embedded in a web page, which itself can be loaded using the system browser, integrated in Eclipse.

Although this is a feasible approach, there is an architectural caveat: a technical barrier between the simulator model within the application, and Google Earth running in a sandbox within the application's browser window. A local HTTP connection can act as a bridge between the rich client application and the visualization running in the browser sandbox.

The sequence diagram in Figure 3.2 outlines the communication flow of the used approach: Initially the browser has a configured staring point, which is an web page delivered by an application internal web server. This starting point includes an randomly chosen port number and a session identification. A random port number allows the user to run more instances of the EveSimulator simultaneously. A session identification is needed, as there can be more visualization frames at the same time as well. Within this web page, an Google Earth Plugin instance is loaded and

Figure 3.1: EveSimulator Google Earth Visualization.

linked to another local web service delivering the session's data as KML [14], [15]. Additionally different resources have to be delivered as well, such as stylesheet files and images referenced within the stylesheets.

In order for this approach to work, the internal web container runs three different servlets, which will be explained in more detail in Section 3.1.1: (i) The static content servlet delivers plain content such as images or stylesheet files. (ii) The velocity servlet is to enrich static HTML pages with dynamic content. (iii) Finally, the KML servlet publishes the data, stored within the Eve Model, to the Google Earth Plugin instance.

The output of the just described visualization component can be seen in Figure 3.1.

3.1.1 EveSimulator Internal Web Container

Using web plugins for geospatial visualization within rich client applications has one important drawback: the information exchange between the application internal data model and the visualization component, running within a browser sandbox, can only happen indirectly.

One EveSimulator instance is able to manage and open a number of simulations simultaneously. Therefore, the web container has to be aware of the different loaded models. This is achieved by using the session IDs included in the EveCom-

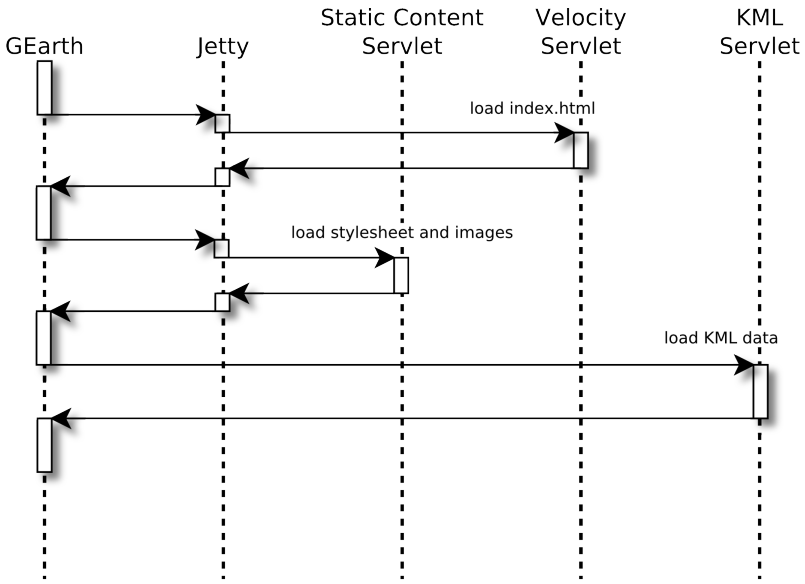

Figure 3.2: Google Earth Visualisation.

ponent model. For each request to the servlet container, this session ID has to be included. Using this session ID, the requests are routed, using EveSessions, to the corresponding EveComponent tree.

The following different servlets are used to deliver the web content needed for the geospatial visualizations:

3.1.2 Static Content Servlet

The Static Content Servlet is responsible for content delivery, such as images or HTML files, directly to the internal browser. The content itself is bundled with the application itself and has to be loaded using the Java Classloader. The servlet container is configured to redirect requests within the */default/** context to this servlet. The remainder of the URL is used to lookup the corresponding resource within the available application's classpath. Additional *HTTP-GET* parameters can be used to identify the corresponding session. However, if session-specific content has to be delivered, the Velocity Servlet, explained in the following Section, has to be used.

3.1.3 Velocity Servlet

The Velocity Servlet is one approach to deliver dynamic content within the EveSimulator, such as HTML or stylesheet files. It is not intended to deliver binary content such as images.

Apache Velocity [16] is a Java based template engine enabling the separation [17] of model and view. The servlet container is configured to redirect requests to the */vml/** context to this servlet. The remainder of the URL is used to lookup the corresponding template resource within the available application's classpath. The template resource serves as a source file which contains references and markup statements. These references are then substituted by values passed to the Velocity Engine via context objects.

Obviously the session ID is one of the accessible values. But also the root component of the EveSimulation or EveMonitor tree can be accessed. As Velocity Engine also supports dynamic loading and recursive execution of templates, the while data tree can be used to generate content.

Although this approach could also produce KML, the following section explains the generation of KML using a dedicated servlet.

3.1.4 KML Servlet

The KML Servlet is designed to deliver KML content to the Google Earth Plugin. The servlet container is configured to redirect requests to the */kml/** to this servlet. Contrary to the other servlets, the remainder of the URL is omitted.

The generation of KML is tightly integrated into the Eve component model. This servlet is using this integrated mechanism to retrieve the KML code corresponding the current Eve model and delivers it to the requesting source: (i) the Google Earth Plugin or (ii) the KML markup viewer.

Key to generating KML within the simulator is the *IKMLGenerator* interface. This interfaces serves two purposes. On one hand it identifies components that can produce KML code. On the other hand, it defines the *generateVisualisation* method. This method requires a reference to a currently generated KML document and attaches a KML placemark. To "'KML-enable"' an EveComponent, the IKMLGenerator interface has to be added and the method mentioned needs to be implemented.

Here the above-mentioned EveSession mechanism is relevant, as these instances

generate the root KML-documents and walk through the EveComponent tree recursively triggering the generateVisualisation methods of the components implementing the IKMLGenerator interface. After all placemarks are added to the KML-document, the KML result is serialized and sent to the component requesting the KML code for the specified session ID.

3.2 NASA World Wind Visualisation

World Wind [18] is a free virtual globe or geobrowser developed by NASA and released under the NASA Open Source Agreement (NOSA) license [19], [20]. While initially based on .NET libraries and DirectX, World Wind now is fully based on Java incorporating Java™Binding for the OpenGL®API (JOGL) [21] named World Wind Java SDK (WWJ). As WWJ is intended as a SDK and a basic building block for virtual globe applications, no official virtual globe world wind application is available.

WWJ provides a Java Abstract Window Toolkit (AWT) component for easy integration into custom developed Java programs. AWT is a windowing, graphics and UI toolkit for Java providing an API for developing GUIs. But as Eclipse Plugins are based on the SWT toolkit a bridge has to be used to integrate WWJ into the EveSimulator. The Eclipse Foundation provides a SWT/AWT bridge incorporated at this point which is explained in [22].

The prototypical integration of WWJ incorporates the visualization if EveAgents and their respective EveConnections building a network. For this reason two individual layers are attached to the virtual globe: (i) an AnnotationLayer used for placing annotations containing the agents data, and (ii) an RenderableLayer for connecting the annotations by SurfacePolylines. The previously mentioned SWT/AWT bridge is used within class WorldWindViewComposite, which is the base class of the plugin and is responsible for the visualization. This composite is attached as a tab to the model editor and provides a reference to the root element (the EveSimulation or EveMonitoring node) of the currently loaded model.

As the model is completely independent of the visualization, a dedicated class named *WorldWindProvider* is responsible for translating between the visualization and the data. This class manages the two previously mentioned layers and creates annotations or SurfacePolylines for the components within the model data. For convenience, subclasses dedicated to EveComponents are provided: (i) EveS-

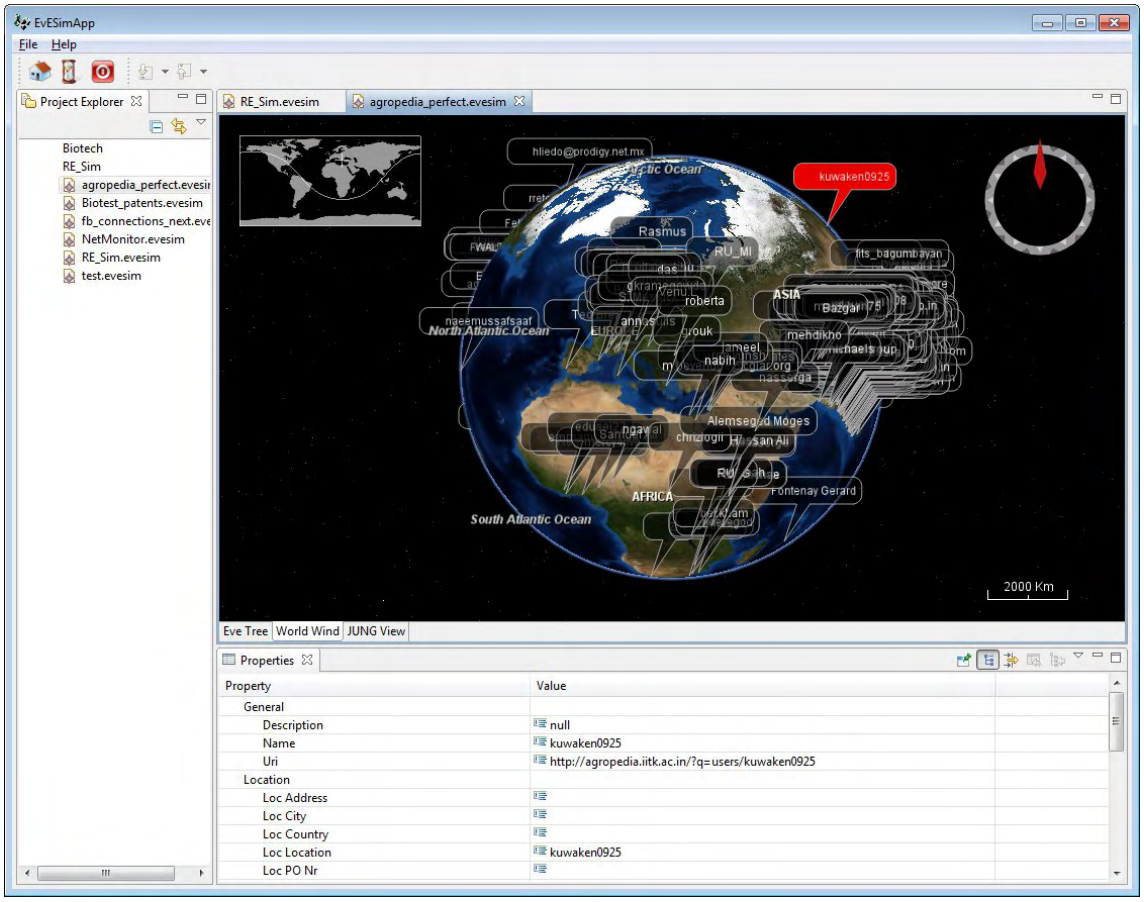

Figure 3.3: World Wind Visualisation.

imGlobeAnnotation for agents, (ii) EveSimMonitorAnnotation for monitoring instances, and (iii) EveSimConnectionPolyline for connections.

Figure 3.3 depicts the simulator using the WorldWind visualization component. As can be seen, one of the annotations is selected and therefore highlighted with red background. As it is important to propagate the modification and selection of components to and from the World Wind Plugin, the following section describes the notification mechanism implemented for the prototype to seamlessly integrate it into the EveSimulator.

3.2.1 Notifications within the World Wind Prototype

In order to create a visualization component that seamlessly integrates within the simulator, a few considerations have to be made: (i) changes within the data model

have to update the visualized components, (ii) the selection of components on the virtual globe have to be propagated to the simulator so that the correct property dialog is shown, and (iii) the selection of a component within the tree editor has to propagate to the WorldWind plugin so that the correct component is highlighted there as well.

For handling the first case, the EMF notification mechanism is used. As all components are inherited from the EMF EObject class, this notification mechanism is automatically available to all model classes. Each EveSimGlobeAnnotation, EveSimMonitorAnnotation or EveSimConnectionPolyline instance subscribes to changes of the related model instance by creating an *Adapter*. The method invoked on a property change for a EveSimConnectionPolyline instance can be seen in Listing 3.1. This code in this example tests whether the *connectedComponent* was changed. If that is true and the location is still valid, the coordinates are updated. As can be seen there is no behavior handling such as the addition or removal of components. This behavior is implemented in the superior *WorldWindProvider* class. Analog to the just explained approach, the *WorldWindProvider* subscribes to notifications. But as it is responsible for all structural changes to the data tree, an *EContentAdapter* is used, which is attached to the root component and listens to all modifications to that component and its children. As can be seen in Listing 3.2, it is checked if a component was added and removed. According to the object's class, the corresponding *add*-method is then invoked to properly update the visualisation accoring to the changes performed.

```
protected void addChange(Notification notification) {
  if (notification.getFeature() instanceof EReference
     || notification.getEventType() == notification.SET) {
     EReference attr = (EReference) notification.getFeature();
     if ("connectedComponent".equals(attr.getName())) {
       if (connection.isValid())
         updateLocation();
     }
  }
}
```

Listing 3.1: Modifications of component attributes

```
protected void addChange(Notification notification) {
  if (notification.getNotifier() instanceof EveComponent) {
     EveComponent cmp = (EveComponent) notification.getNotifier();
     if (notification.getEventType() == Notification.ADD) {
       // An add notification is sent by the parent of the deleted
       // component
       EveComponent objectToAdd = (EveComponent) notification.getNewValue();
```

```
8         addComponent(objectToAdd);
9       } else if (notification.getEventType() == Notification.REMOVE) {
10        // A remove notification is sent by the parent of the deleted
11        // component
12        EveComponent objectToRemove = (EveComponent) notification.getOldValue();
13        removeComponent(objectToRemove);
14      }
15    }
16  }
```

Listing 3.2: Structural modificaton of the component tree

Although the modification of the model from within the visualization component is not implemented at this point, no further adoptions would be needed in order to propagate these changes back, as the automatically generated tree editor for the data model is based upon the EMF-own notification mechanism. Besides the just explained propagation of modifications to the model data, the selection of components has to be notified to and by the implemented plugin in order to provide an integrated look-and-feel.

3.2.2 Select-Mechanism

The mechanism to select components in the visualization is managed by the *WorldWindProvider*. In order to receive notifications if an item is selected within the virtual globe, the *gov.nasa.worldwind.event.SelectListener* interface has to be implemented. This is done by the *WorldWindProvider*, which then can directly subscribe to *SelectEvents*. Now on a selection change, the *selected* method is invoked which can be seen in Listing 3.3. The *selectionAllowed* attribute is a synchronization flag used to avoid recursive reactions on selections (propagating changed selections in a loop until a StackOverflowException aborts the loop). If the selected component implements the *ISelectable* interface, the selection is changed within the visualisation.

The *ISelectable* interface is used to implement the selection mechanism generalized for all subclasses of EveComponent. Each visual component (EveSimGlobeAnnotation, EveSimMonitorAnnotation and EveSimConnectionPolyline in the protoype) implements this method. On creation of the visual component, the WorldWindProvider adds it to a registry using the EveComponent as a key and the ISelectable as a value. Classes implementing the *ISelectable* interface have to provide methods to set the selection status and for the retrieval of the corresponding EveComponent. In the method to change the selection status, the visual representation

of the selected component can be changed.

```
1 public void selected(SelectEvent e) {
2   if (!selectionAllowed)
3     return;
4   if (e.getEventAction().equals(SelectEvent.LEFT_CLICK)) {
5     Object o = e.getTopObject();
6     if (e.getTopObject() instanceof ISelectable) {
7       selectComponent((ISelectable) e.getTopObject());
8     }
9   }
10 }
```

Listing 3.3: Selection method

The following two sections explain how the selection of components are propagated to and from the World Wind Prototype.

Propagating selections to the World Wind Prototype

For handling the selection of components outside of the implemented visualization plugin, the Eclipse SelectionService has to be used. In order to do that, the WorldWindProvider implements the ISelectionListener interface. By implementing this interface, the WorldWindProvider can subscribe to selection changes within the application.

The implementation of the ISelectionListener interface only requires the *selectionChanged* method to be implemented. This method is automatically invoked by the SelectionService, as soon as the current selection changes. The WorldWindProvider's selectionChanged method checks if a instance of EveComponent is selected. In the case that a component was selected, the registry is used to lookup the ISelectable instance corresponding to the EveComponent. If such a ISelectable instance is found, the selection is changed to that instance and the selected state is set to *true*. If an old selection is available, the selection state is set to *false*.

Propagating selections from the World Wind Prototype

The submission of a notification to the selection from the World Wind Prototype has to be externalized in an own thread as can be seen in Listing 3.4. This is a requirement of the Eclipse SWT UI, as otherwise the user interface could freeze until the notification is dispatched to all subscribes. Therefore, an exception is thrown if circumventing this limitation is tried. Additionally, it can be seen that the synchronized flag *selectionAllowed* is set to false during the dispatching of the

notification in order to avoid any deadlocks.

```
1 private void notify(EveComponent newSelection) {
2   final ISelection sel = new StructuredSelection(newSelection);
3   getDisplay().syncExec(new Runnable() {
4
5     public void run() {
6       synchronized (selectionAllowed) {
7         selectionAllowed = false;
8       }
9       composite.getSelectionProvider().setSelection(sel);
10      synchronized (selectionAllowed) {
11        selectionAllowed = true;
12      }
13    }
14  });
15 }
```

Listing 3.4: Notification of selection changes to the Eclipse UI

This chapter provided an overview of the implementation of the geovisualization component of the EvESimulator based on the architecture described in Chapter 2. Two different prototypical implementations are described using KML as an interface to integrate Google Earth (see Section 3.1) on the one hand, and a AWT-SWT bridge to integrate NASA WorldWind (see Section 3.2)on the other hand.

The following chapter focuses on an exemplary real-world use-case of the EvESimulator.

4 Simulation, Emulation and Testing

To demonstrate the capabilities of the EvESim framework, this chapter outlines a complete example simulation case, explaining the simulation logic, architecture, implementation and execution as well as emulation and test capabilities. It illustrates the integration of the *Flypeer* infrastructure into a simulation case. The complete description of the simulation case can be found in [4].

4.1 Simulation Conception

In general, the simulation is intended to test a service that provides the functionality to create, edit and delete text documents and additionally, to test the underlying communication infrastructure. The service does not implement a real document editing service just performs file dumps of the created documents on the hard disk. It is designed as a *Flypeer* service and therefore, to be deployable in a *Flypeer* network. The first step is the creation of a set of agents that execute the simulation. The acquisition of the network data that is used to create the simulating agents is laid down in detail in Section 5.1. Utilizing this data given in XML format as a starting point, a list of agents with the specified connections and other properties is created and can be read and displayed by the *EvESim* framework. To achieve compliance with the framework the data has to be converted using the EMF model that is used by the framework. After this conversion the framework is able to read the data and create the visualizations accordingly. The actual parser implementation is described in Section 4.4. After the conversion the agents can be operated to construct the actual simulation.

A simulation consists of the service to be tested, a number of gateway agents which are deployed on distributed physical *EvENode* instances and a larger number of testing agents. Basically, subsets of the testing agents are placed into the administration container of a gateway. This gateway is used as a communication

proxy because the *Flypeer* infrastructure is only supporting one node per running software instance. The gateway provides *Flypeer* communication functionalities to its underlying agents giving them the opportunity to call the service in the *Flypeer* network. As the infrastructure is based on transactions, the gateway agent exposes its own transaction initiation object. The underlying agents can use it, in order to create transactions to the service to be tested. The transaction results are transferred directly to the initiating agent without the usage of the gateway agent. To enable more agents on one running *EvENode*, the introduction of this gateway agent is inevitable. The gateway is started once and runs during the whole simulation (running time) only maintaining the connection between its agents and the network.

When using existing data sets, the simulation can be created randomly which means the user specifies the number of *EvENodes* and the agents are distributed randomly among them. As the agent's behavior is composed of document creation, editing and deletion these documents have to be specified. In existing data sets, agents have specified services that they provide. For the purpose of this simulation these services are representing the documents that the agent creates, edit or delete. The architecture of these agents is described in Section 4.2.

The timer component is in charge of triggering the time when an agent executes its behavior. The *EvESim* framework is a timer-based simulation framework which means every component's behavior has its controlling timer. For the simulation case basically two timer components are introduced and designed. The first one is a timer that triggers an agent's behavior only once. It is therefore called starting timer and used to launch the initialization process of the gateway agents. The simulation agents are supposed to have a different behavior. The framework provides a repeating timer which executes the behavior periodically. Giving the agents the opportunity to change their behavior trigger period, the timer designed has to provide the capabilities of changing time parameters. The timer is designed to calculate the periods for executing after a time function. The parameters needed for the time function can be specified before starting the simulation and changed during runtime. The timer supports four time function patterns which are **linear**, **potential**, **exponential** and **logarithmic** ones.

This section gives an overview to the basic simulation architecture and the additional components developed to acquire network data, create graph view visualizations and metric calculations, create a simulation setting and control behavior

of the simulating agents. The next section continues with the simulation model which describes the actual simulation procedure and the interaction of the simulating components. It provides a description of the schema that has been created for the simulation, leading to the section in which simulation parameters are described.

4.1.1 Simulation Model

First of all, the simulation model is designed to be realizable with the *Flypeer* infrastructure. This results in an adaption to bypass the problem of only one running *Flypeer* node entity per running instance and the communication is adapted to be realizable within the *Flypeer* transactional concept. Consequently, the simulation consists of agents, gateways and the service to be tested. Probably the model can be extended in future by adding multiple instances of tested services. This depends on the usability of the search mechanisms provided by the final release of the *Flypeer* infrastructure. Every single part of the simulation model is described in the following.

The *EvEGUI* represents the central controlling entity of every simulation. It can create and store simulations in order to reuse them. Additionally, it shows the configuration of the simulation in a tree view, enabling the user to configure the parameters of the simulation. These parameters are explained in Section 4.1.2. The *EvEGUI* is also capable of illustrating a geo-localized visualization of the simulation network as well as a graph representation. Besides, it runs a sync service that collects the information, which is generated from the agents during the simulation. When creating a simulation from scratch, the *EvEGUI* provides a tree view enabling the user to add all the components needed in a hierarchical way. The structure that the simulation case has, is illustrated in Figure 4.1. The following description gives an overview of the used components and its tasks. The complete simulation process is explained afterwards including all the communication that takes place between the components.

Every simulation setting starts with a root *EvESimulation* object, representing the root component of the simulation. It is mainly used to set a simulation name and a session ID which every component of the simulation has in common in order to identify which component belongs to which simulation. Next in the simulation hierarchy are the **EvENode** objects. They represent the physical instances of server nodes that are installed on machines and connected via a network for example the Internet. Typically the user configures the server URL within the *EvENode* object,

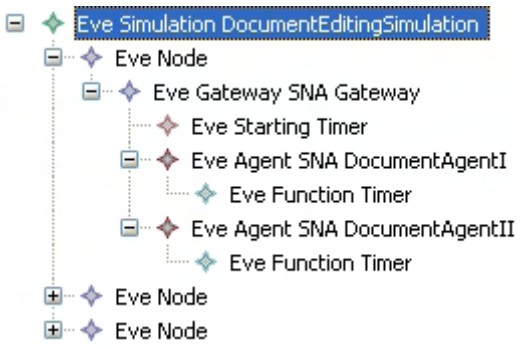

Figure 4.1: Simulation Structure

to enable the *EvEGUI* to distribute the simulation to the server correctly. The server URL is also needed later on to collect the messages generated by this node. By assigning coordinates to the *EvENode* object the framework is able to show not only the simulated entities but also the simulating entities within the geographical visualizations. The *EvENode* basically can hold and execute every type of agent that has been implemented by the user.

The objects explained in the following are developed for the presented simulation case. As in the simulation, agents need a gateway in order to use the infrastructure correctly. Therefore, the next object in the tree hierarchy has to be an *EvEGateway* object. In this specific simulation case this object only has the task to provide the infrastructure access to its underlying agents. To trigger the initialization of the infrastructure, the gateway object has a starting timer that executes the behavior of its assigned component once. Next, the actual *EvEAgent* objects are put into the simulation setting. Every agent instance has to have an *EvETimer* object assigned to it. This timer is responsible for its parent agent's specific behavior. The parameters that can be configured within the agent object are described in Section 4.2. For the simulation the agents are using function timers that are capable of constructing complex time functions to control execution period. The specific time functions are described in the section before. Finally, every agents has to have specified demanded services which are equivalent to demanded documents needed for the simulation. These documents are created, edited and maybe deleted during simulation runtime.

This leaves only one missing object, which is the service that has to be tested. As shown in Figure 4.1 this object does not exist in the simulation model. The reason for that is that the service should be independent from the whole simulation in order to be replaceable and flexible. For the simulation an example service is

created that is capable of serving the agent's document creating, editing and deleting functionalities. This service is described in Section 4.3. Maybe the service will be replaced with a real-world service like Google Docs[1] in future.

After introducing all the specific components and their relationships, the actual simulation process is explained in detail the following section. Moreover, the description covers the processes that happens on one *EvENode* instance as they are exemplary for all the other ones. First, the **pre-simulation phase** is illustrated. After the model has been designed correctly in the above described structure and the distribution to the executing *EvENodes* has been successful, the simulation is ready to be put into execution. All the components are instantiated on their execution places and the configuration parameters are set. The document editing service has been started and its initialization process has finished successfully. In this pre-simulation state all the timers involved are waiting for the starting signal to begin with the component execution. The gateway is waiting for its single execution in order to initialize its *Flypeer* node representing the gateway to the infrastructure. The agents are waiting for being executed by their assigned timer's components.

Finally, as the user starts the simulation all the timer components begin to execute their parent agent components. This is the starting point for the **simulation phase**. The gateway is starting its initialization process. As long as it has not finished, the agent timers execute the agents every 500 milliseconds to determine whether the responsible gateway is started or not. When the gateway starting process is finished, the agent is informed and additionally notifies its assigned timer which executes the agent and subsequent calculates its first waiting time period after elapsing the execution is repeated. When the agent is executed, it calculates a random number and initiates a **creating**, **editing** or **deleting** transaction according to the defined probabilities.

In a **creating** transaction the agent randomly chooses a document from its demanded services. It initiates a transaction from the type create with the document name, its own name as creator, the content of the document, the session ID of the current simulation and a current system timestamp as parameters. These parameters are sent to the document editing service via the *Flypeer* infrastructure. The document editing service receives these transaction parameters, creates the described document if it does not already exist. Creation of a document in this context means, instantiating a programmatic representation of the document in the service

[1] `http://docs.google.com/`

which stores a creator, a list of editors, content, and a list of timestamps indicating when exactly actions were taken on the document.

This creating procedure is similar to the **editing** and **deleting** transactions. With the change of the type in the transaction parameters the agent can invoke either an edit action or a delete action. When editing a document, the agent sends the content that has to be added into the file and the service edits the document if it exists. In case it does not exist, the service creates it and adds the content as well. Lastly, if the service receives a delete request for a file it is not deleted but marked so, to ensure that no actions can be taken on it. The recreation of a file after erasure it is not possible.

During the simulation process every agent is performing several transactions. The infrastructure has to guarantee that the service is found and called in every single transaction correctly. The infrastructure relies on the precise name of the service, which has to be known before starting the transaction. The service answers every incoming transaction with a reply message. It contains the result of the request which states whether an action was taken correctly or not due to occurred problems. Moreover, it includes the timestamp to enable the agent to calculate the time that was needed to execute the transaction. The agent stores these transaction times and the results received from the service. When the timers have reached their maximum execution count the simulation is finished. Results obtained from the simulation are the transaction times and action results from every single agent on the simulation side and the documents created, which record all the actions that were taken on the service side.

The complete architectures of the simulation agent and the document editing service are described in Sections 4.2 and 4.3. Also the communication that is conducted between the components is explained in further detail. However, the next section deals with parameters that can be adjusted within and around a simulation and influence the simulation results. The parameters which can be configured within the agents and the service are discussed in Sections 4.2.1 and 4.3.1.

4.1.2 Simulation Parameters

Before dealing with the main components, the general simulation parameters are topic of this section. To be more precise, simulation parameters that are not directly connected to either the simulating agent or the service involved and therefore, can be defined as external or environmental factors. The first environmental factor is

the network hardware infrastructure used. The condition and the performance of the infrastructure influence the simulation as the communication of the distributed nodes relies on it. Besides, the simulation depends on how far the distances between the nodes are and on the bandwidth every node has available for the simulation communication.

There are two possible ways of running a simulation which are either local or distributed. Running a simulation locally means all the components are initialized and executed on the physical hardware. Resulting in a simulated network traffic that is realized with the usage of the callback interface on the network card. This centralization on one machine is on the one hand, responsible for lower transmission times between the components and on the other hand, for complete avoidance of any network barrier. Even if the local simulation results are not equal to the distributed ones, the user has the opportunity to run simulations without a distributed network of nodes. But as this is not a real-world scenario and in order to test the *Flypeer* infrastructure, the simulation components are distributed. This leads to higher transmission times and increased probability of transmission fails caused by the network failures or barriers.

The number of components is a key parameter in every simulation that can be influenced. As each agent relies on an underlying gateway, there can be variations in how many gateways a simulation includes and how many agents are using on gateway to execute their communication. Especially when a large number of agents uses one gateway, chances of interferences are much higher and failed transactions are emerging. As the final version of the infrastructure is supposed to support search mechanism, the number of services probably will also be variable in future.

To recap, this section covers external simulation parameters and factors which influence the simulation. Knowing that there are many more factors that could be mentioned, the section should only highlight the most important ones. The next section introduces the specific architecture of the document editing agent and its behavior within the simulation.

4.2 Document Editing Agent Conception

The document editing agent is the working component in the simulation presented. In general, a software agent is an independent working part of software that is capable of acting independently and communicating with its environment and other

agents in order to fulfill a given task [23]. Software agents are used in a wide range of application areas as they are working autonomously and self-governing. They are perfectly usable for simulating actions or tasks that are performed "by a composite of individuals in the real world". As these individuals are interconnected and communicate, also the agents have to have exactly these capabilities. The composite of these agents can always be observed as a P2P network. The agent presented in this section is designed within the *EvESim* framework for the purpose of a document editing service test. This service is described in Section 4.3 [23].

The section above deals with the location of the agents within the simulation structure. The agent itself does not 'know' much about its environment, despite its own parent entity which represents the gateway to the infrastructure and its behavior control entity. The behavior that the agent is performing, is outlined in Section 4.2.2. The gateway gives the agent the ability to communicate with the service it intends to test. Normally it would have the ability to access the infrastructure without the usage of a gateway, but as explained in Section 4.1.1, this is not possible due to infrastructure limitations. In which time periods the agent is using the infrastructure to execute its tests, is controlled by an assigned timer entity. The agent as such, does not act without the trigger of this particular timer entity. Therefore, its existence and configuration is compulsory for every agent.

In the following, the testing procedure taken by the agents is laid down in detail. By default every agent in the *EvESim* framework has a list of services on demand, which in this case are used as the documents needed to test the document editing service. The services on demand can be configured before the simulation is executed. As the agent receives his first action trigger from its timer, it randomly chooses one of its demanded services. Randomness is used to guarantee that the agent does not always take the same service and to simulate a more realistic behavior. Once the service has chosen its name, additional parameters, specified in Section 4.1.1, are packed into a transaction prepared for the *Flypeer* infrastructure. Finally, the agent has to determine which specific action it wants to execute. There are three possibilities which are creating, editing or deleting the chosen document. To give the user more parameters to influence the choice of action is probability-based and described in Section 4.2.1.

When the action is selected the transaction is started and, as the communication is an asynchronous process within the *Flypeer* infrastructure, the agent has to wait for results. There are seven possible answers that the agent can receive from the

service. They are **existed**, **created**, **created edit**, **edited**, **edit failed deleted**, **deleted** and **deleted failed**. The first two possibilities follow a create action of the agent, the next three, an edit action and the last two a delete action. Receiving **existed** indicates that the services has already created a document with the stated name in the create transaction within the given simulation session. When the creation of a document was successfully accomplished by the service, the agent receives a **created** reply. When the determination process has resulted into an edit operation, the reply message is **created edit**. This implies that the document which the agent intended to edit had not existed but was created. In this context the edit request is equivalent to a create request. When **edited** is returned, a normal edit of a document has been accomplished by the service. The last possible result for an edit action is **edit failed deleted**. This follows a request where the agent aimed at editing a document that has already been deleted by another agent. As the recreation of documents is not possible, the agent receives this result to be notified of the failure of the request. Finally, when the agent has dropped a **delete** request, the first possible result is deleted which indicates the effective execution of the request. Lastly, the result **deleted failed** returned by the service covers the case that the document the agent intends to delete has either been previously deleted or has not been created yet. In addition to the returned status, all the result messages from the service include the original timestamp that was handed over by the agent. This enables the agent to calculate the exact time the transaction needed including the transmission time to the service and backwards. The transaction time is used in Chapter 4.7 to determine the performance of the infrastructure.

Summing up, this section gives an overview about the document editing agent, its acting and the actual testing procedure explaining all the possible results shortly. The next section outlines the properties that are influencing the test procedure and are editable by the user.

4.2.1 Agent Properties

The following paragraphs describe the properties of the agent that can be edited by the user in order to influence the results of the simulation. They are shown in Figure 4.2 which is a screenshot of the *EvEGUI*. First, to give the user more influence on the actions executed by the agents, those actions are not chosen randomly but follow a possibility schema. This means specifying the probability for the creating, editing and deleting action. In the Figure, the probability for creating the documents that

are specified within demanded services is twelve percent. Editing the probabilities results in a big change of the simulation outcome. For example, if the deleting probability is high, documents are vulnerable of being deleted before other agents had the chance to execute an edit operation. Also influencing the results is the number of documents the agent has in its demanded services as a broader range distributes the operations better leaving less operations on one document.

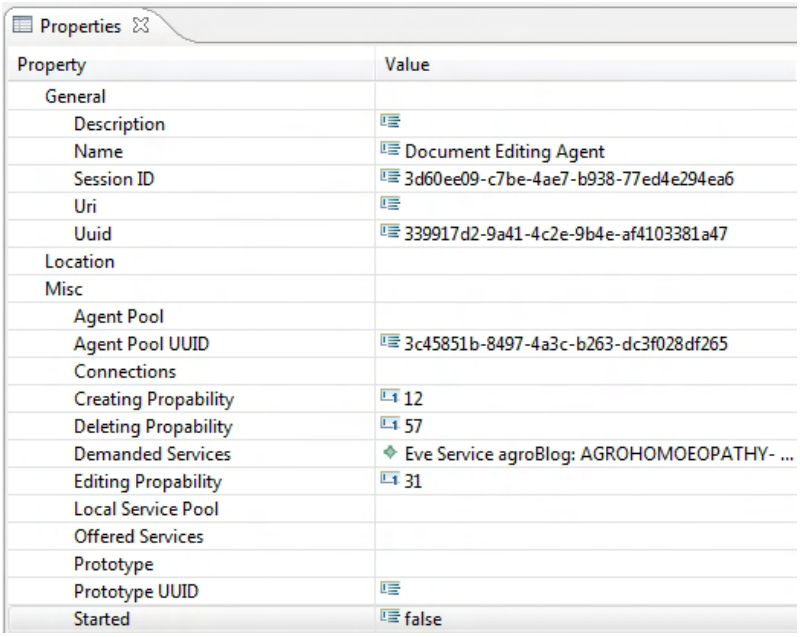

Figure 4.2: Document Editing Agent Properties

This section has shortly explained the editable properties of the document editing agent as they play a very important role in Chapter 4.7 influencing the simulation results. The next section elaborates the behavior that every agent performs.

4.2.2 Agent Behavior

A central point of every simulation is the actual behavior that every agent follows and which influences the way the agent performs actions. Goals that are set for the agent behavior are that the periods of execution are based on time functions and that the agent can dynamically change its own behavior function during simulation runtime. When the user has assigned a timer to the simulation agent, the time function can be influenced by setting property variables. These properties are illustrated in Figure 4.3. The time functions, that can be constructed by the timer, are described in

Section 4.1. This one illustrates the way that these functions can be influenced.

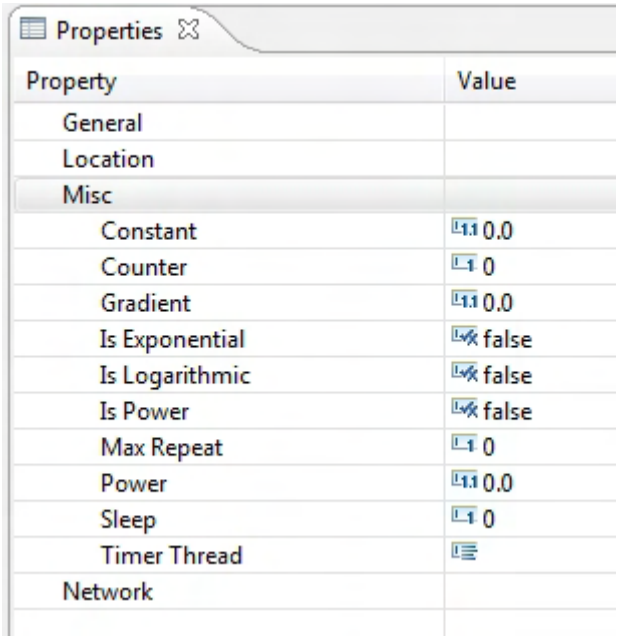

Figure 4.3: Function Timer Properties

First, the two most important parameters settable are the gradient and the constant. They represent the basic parameters for every time function. By default the timer is working with a linear time function composed by the basic parameters. As displayed in Figure 4.3, there are three additional function flags that can be placed. Setting the exponential flag indicates that the timer changes the linear behavior into an exponential one. As the exponential functions are growing tremendously they are only restrictedly usable because the time periods are getting very long. Logarithmic functions are better usable for the time periods. In order to work with them, the logarithmic flag has to be placed which implies the unset of all other flags as only one time function can be used at the same time. Last, the time function can follow a potential progression. To use this type for the timer the user has to specify one additional parameter which is the power of the function. These four functions in total give the user a broad range of possibilities to construct a time function that describes the simulated behavior best.

Moreover, the agent is supposed to be able to change its behavior according to its assigned timer entity's time function. Presently this change of behavior has to be implemented by the agent itself. A behavior change event can be triggered for example by specifying a definite period count and change the behavior when this

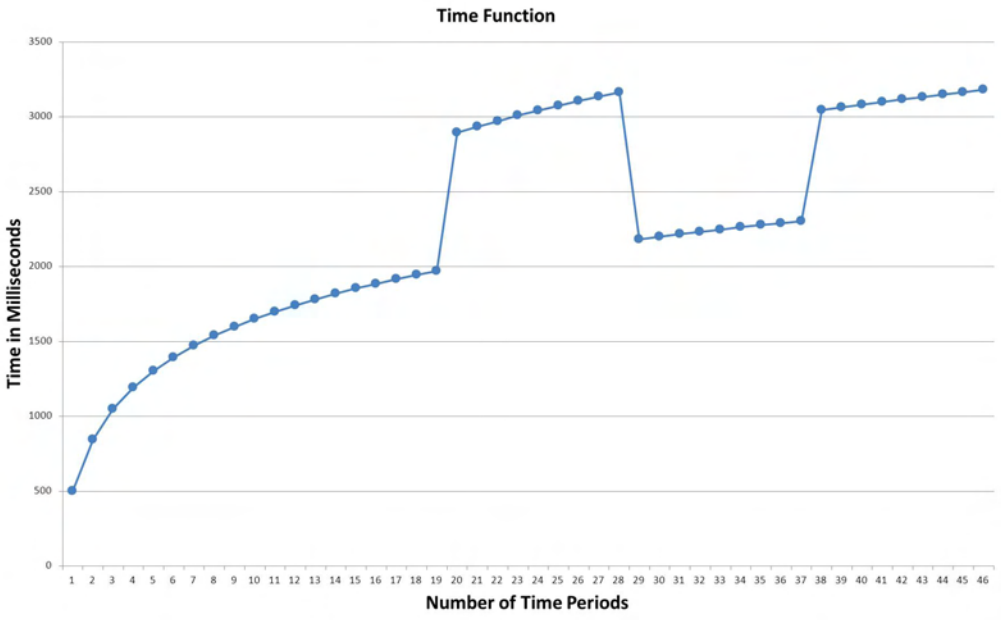

Figure 4.4: Agent Behavior Time Function

count is reached. To illustrate this behavior change operations, an example is given in Figure 4.4. It displays a logarithmic time function which changes three times. The first change happens after twenty timer periods which gives the time function a higher gradient resulting in a jump. This higher level is held for ten time periods and then changes back to the original time function. Another jump is triggered ten periods later. This example represents one time function, but as the agent can change every parameter presented above, there are a lot of other possibilities.

This section finishes by presenting the behavior that is performed by every specific agent in the simulation and the parameters that can be influenced by the user before and by the agent itself during simulation runtime. One possible behavior time function is presented and the parameters to manipulate it are stated. To finalize the architecture chapter, the document editing service is introduced in the next chapter.

4.3 Document Editing Service Conception

This section describes the document editing service which embodies the target service for the testing processed by the simulation. This service is designed as a *Flypeer* service running within the infrastructure and therefore, reachable in a transactional way. It is proposed to handle tasks which are common for document editing services

like Google Docs[2]. These tasks provide functionalities to the users which are the creation of documents within the service, the handling and deletion of existing ones. The service is responsible for the correct administration of all the documents and assures that the users can correctly accomplish modifications. Furthermore, the service sorts the documents concurrent to the simulation sessions that are executed to enable multiple simulations communicating with the same service simultaneously.

To make the document information created by the using agents persistent, the service stores the documents in two different ways. On the one hand, they are programmatically stored as *EvEDocument* objects, which include the creator of the document, its content, its editors and a flag that indicates whether the document has been deleted or not. On the other hand, the service performs a file dump after each modification transaction and writes the stored information into real files. This enables the user to conduct an easier analysis of the created documents. Depositing the information also implies an additional workload to the service and therefore, is more realistic then just storing the information within the software execution. The content of such a dumped document can be found in Listing 4.1. It shows the events that were taken to edit this document which were one create event, many edit events and one delete event. The create event is illustrated in line one and contains the name of the creating agent, the event type which is 'created edited', the transaction time from the agent to the service, the human readable date of the creation event at the agent and the same date in milliseconds. Explaining the event type 'created edited' implies that the document was not generated with a regular create event but the agent tried to execute an edit event instead. This edit event is converted into a create event at the service.

```
1 DocumentAgentI,Created Edited,78,Thu Apr 01 14:02:17 CEST 2010,1270123337468
2 DocumentAgentII,Edited,-121875,Thu Apr 01 14:02:20 CEST 2010,1270123340359
3 DocumentAgentI,Edited,110,Thu Apr 01 14:02:20 CEST 2010,1270123340750
4 DocumentAgentII,Edited,94,Thu Apr 01 14:02:24 CEST 2010,1270123344500
5 DocumentAgentII,Edited,156,Thu Apr 01 14:02:26 CEST 2010,1270123346562
6 DocumentAgentI,Edited,-121735,Thu Apr 01 14:02:27 CEST 2010,1270123347890
7 DocumentAgentII,Edited,-121718,Thu Apr 01 14:02:27 CEST 2010,1270123347953
8 DocumentAgentI,DELETED,-121890,Thu Apr 01 14:02:31 CEST 2010,1270123351906
```

Listing 4.1: Generated Service Document

Furthermore, the 'edit' events are marked with the event type 'edited' and give information about what the agent has edited, at which time and how long the transaction took. There is a problem illustrated in line two: There has obviously

[2] http://docs.google.com/

been a negative transaction time between the agent and the service. This results from the asynchronicity of the agent's and the service system's clock. Therefore, these transaction times cannot be taken for analysis as they are mainly influenced by the system's clock and not by the transaction itself. More details on how this problem was solved are given in Chapter 4.7. After some 'edit' events, the last event shown in line eight is a 'deleted' event marked with the type entry 'deleted', which implies that the modification and the recreation is not possible. The document data is not deleted in order to show the document history. As the service dumps all the documents that are created, the complete history of the events can be analyzed. If there are simultaneous sessions working with the same service, the documents are placed into a folder that is named after the session ID of the simulation to guarantee a clear division of the documents.

After explaining the general design of the service, the next section elaborates on the setting of the service and its connection to the simulation.

4.3.1 Service Setting

Talking about the setting of the service, it has to be stated that it represents an independent part of the simulation. Meaning it is not controlled by a timer function and also not deployed with the *EvESim* framework. Consequently, it runs as a totally independent entity, only connected to the simulation as it offers its functionalities within the *Flypeer* infrastructure. Originally, the integration of the service into the simulation context was planned, but in order to make the service easily exchangeable it was outsourced. The drawback is, the service is not able to communicate with the *EvEGUI* as it has no ability to exchange messages. But, as the intention of the simulation headed more into the direction of creating the ability to simulate testing of a real-world service, this independent architecture has been chosen.

Dealing with the service as an independent part goes along with several advantages. It can be pre-initialized avoiding waiting times during the simulation and it can be reused for many simulations without restarting it. As the introduction of the final version of the *Flypeer* infrastructure is supposed to support service search, there can be further development in the direction of multiple service entities. Deploying many service entities on different physical hardware distributed over the Internet, will give the agents the probability to choose which service they want to use.

Finalizing the architecture chapter, the last section presents the service behavior which outlines the actions taken by the service when agents request actions.

4.3.2 Service Behavior

The service does not implement a real behavior, which means it waits for requests of the agents and handles them if possible. These requests come along with a transaction over the *Flypeer* infrastructure. After getting all the parameters that are included in the transaction, the service checks the type of request and reacts accordingly.

The first possibility is that the requesting agent wants to create a document. The next step taken by the service is to check whether this document exists. If it does, the service sets the return result to 'document already exists' and sends this back via the infrastructure. If not, the service creates the document according to the parameters and returns a 'creation successful'.

The second possibility is that the request includes a document edit inquiry forcing the service to determine whether the document exists or exists but has been deleted. The third possibility in this case is that the document does not exist. This is handled by letting the service create this document instead of editing it. An existing document leads to a successful editing and to a respective result sent back. If it turns out that the document has been deleted, the agent receives a 'failure' as the deleted document cannot be edited or recreated.

The fourth possibility is that the agent wants to delete a document. After the check if the document exists and has not been deleted, the service reacts by either deleting it or sending back a 'failure' as the deletion operation is not possible. After every operation which has been described, the service dumps the documents into physical files as explained in the Section 4.3.

Summing up, this section outlines the operations taken by the document editing service as the final section of the architecture chapter. This gives a glance at the whole conception of the simulation case, data acquisition, data visualization and metric calculation. The next chapter briefly describes the implemented code structures that are newly-created to put the architecture into a running program.

4.4 Infrastructure Implementation

To create the infrastructure, the implementation of some classes in order to assist the simulation agent, has been necessary. The first one covered is the implementation of the *EvEGateway*. This class implements the *EvEAgent* interface as the normal simulating agents, but does not interact with the simulating agents at all.

Its purpose is to initialize, administer and in turn guarantee its underlying agents the possibility to use the infrastructure. Programmatically the *EvEGateway* is designed as an agent that is started once, initializing the *Flypeer* node and providing the *TransactionInitiator* class to its child agents. The gateway has a status that can be requested by the agents in order to retrieve information whether the gateways initialization has finished or not. Other functionalities are not provided by the gateway, as not needed. To start the gateway class, a special *EvEStartingTimer* has been implemented that executes the gateway once, to activate the infrastructure support for the agents. The timer implements the *EvETimer* interface which provides the properties that all timers in the framework share.

The timer that has been created to supply the simulation agents with a behavior control is described next. It is named *EveFunctionTimer* and designed to utilize time functions that control the time periods for the agent's behavior execution. These functions are implemented by using the 'Java Math' functions for calculation. To enable the control of the functions, the timer offers parameters. These parameters and the time functions supported are described in Section 4.1. Additionally, they can be influenced by the assigned agent during the simulation runtime. Furthermore, these timers are reusable for all simulations.

The *EvEDocument* class is one of the essential classes for the simulation, as the document editing service stores all the events that happen to a document. It contains a lot of information parameters that are relevant for the simulation. These are first of all, the name of the document and its author, second a list of editors which are the agents that have interacted with the document, third the content of the document which contains the current text data, fourth a list of transaction times that illustrates the time periods which every document manipulation needed, fifth the session ID to assure that the documents belong to a specific simulation and last a flag that indicates whether the document is marked deleted or not. These parameters represent the most important data that is created during the simulation.

4.5 Document Agent Implementation

The *DocumentEditingAgent* represents the executing entity in the simulation and is derived from the *EvEAgent* interface, the framework is offering. This interface contains a number of methods and parameters which the agents have in common. For example, the list of demanded services, which are used for the document rep-

resentations at the beginning of every simulation, is derived from that. The most important one is the 'execute' method that triggers the agent to perform an action which is used by the timer to control its assigned agent entity. The actions performed by the *DocumentEditingAgent* are described in the following. As the agent is not capable of communicating with the infrastructure, the initial call of the execute method invokes a check procedure in the agent that retrieves the status of the gateway instance in charge. As long as the gateway is not initialized successfully the agent is idle only checking the status periodically. As soon as the gateway has been started, the agent gains the *TransactionInitiator* instance and begins to perform actions.

In order to manage the three different types of actions which are 'create', 'edit' and 'delete', the agent is equipped with probability parameters. Before an action is taken, the agent generates a random number which is used for two purposes. On the one hand, it is utilized to gain one random document that is processed during next action and on the other hand, to define the type of the action itself. The result of the action type is weighted with the purpose of fulfilling the probability specifications. After this pre-action process is finished, the agent starts the defined action with the specified document. The probability parameters of the action type choice are set in advance and fixed during the simulation execution.

As the process of creating, editing and deleting documents is described in Section 4.1 the agent's ability of changing its behavior is outlined in the following. Changing its behavior is a profitable ability for an agent as it enables the adaptation of its behavior to certain circumstances meaning that the agent is able to react and not only to act. The changing of the agent's behavior is the programmatically altering of parameters that define the agents assigned timers execution periods. At the moment, this change has to be implemented manually in the agent class, as there is no functionality, such as a change handler class, provided by the framework. Further development may lead to the provision of such a handler class. Within the *DocumentEditingAgent* a change behavior method is called after every performed execution in order to alter the behavior.

The event that triggers a behavior is characterized within this method and if it occurs, the method changes the timer parameters. There are several indicators that can be used to define when that happens. The agent, used in the simulation, changes its behavior action count based, which means after a defined number of execution events. The time function that results from such a changing behavior is

shown in Figure 4.4.

However, the events for behavior change can be chosen far more sophisticatedly and can be directly interlinked with outcomes from the actual action the agent is performing. A time based changing behavior can be used to control the agent as well. The final goal is that the agent dynamically changes its behavior in an ongoing process of integrating values of the environment, its own parameter values and the target it is supposed to achieve.

4.6 Document Editing Service Implementation

This is a short discription of the implementation of the *EveDocumentEditingService*. The class implements three interfaces which are the *FlypeerBusinessService*, the *FlypeerServiceListener* and the *FlypeerRollbackListener* interface.

The *FlypeerBusinessService* interface is the main class from which every service deployed within the *Flypeer* infrastructure has to be derived. There are several methods the interface provides. The 'start' method is called when the service is deployed, which happens when the *Flypeer* node, which it is assigned to, is started. An object is passed and can be used by the service to get some simple information about the host peer. This method is used to initialize the complete list of documents that is stored at the service as well as a list of sessions the service is administrating.

Next, the 'stop' method, which is called when the service is going to be undeployed, can be used to clean up resources that the service might have opened. As the *DocumentEditingService* does not have any particular resource to clean, this method is not used. The 'clear' method is called after each transaction finishes successfully. This method is crucial for the service as the dumping of the managed documents described in Section 4.3 is operated within this method.

This leaves two 'get' methods open. On the one hand, there is the 'getServiceListener', which returns an implementation of the *FlypeerServiceListener* interface where the actual logic of the service is executed. The service implements the *FlypeerServiceListener* interface itself and is described later on. On the other hand, the 'getRollbackListener' method, which is called when a transaction fails, gives the user the opportunity to implement some actions to respond to the failure of a service invocation. As the service does not support the rollback of a document manipulation this is not implemented.

The *FlypeerServiceListener* interface provides the 'serviceEvent' method, which

is responsible for dealing with the service calls that reach the service. It receives a single parameter of the type 'ServiceContext', which contains information regarding the call, including the service parameters. These parameters are described in Section 4.3.

Last, the *FlypeerRollbackListener* interface provides the method 'rollback'. This method also receives a parameter of the type 'ServiceContext' that is equivalent to the one passed to the 'serviceEvent' method of the 'FlypeerServiceListener' interface. The 'rollback' method is called when a transaction fails and allows the service developer to take the appropriate actions.

The service administrates two list objects that contain the documents currently stored in the service and the according sessions to which these documents belong to. When a transaction is finished and the service has executed a change to one of the documents, the file dumping process is triggered. As the first step, the service iterates through the list of sessions and creates folders with the session names if they do not already exist. The second step is the iteration over the documents to be dumped. Every document is handled by retrieving its session ID and storing its content in a file situated in the according session folder. This process is executed after every transaction that the service handles.

In a nutshell, this chapter is about the implementation characteristics which complete the picture of the simulation design, architecture and implementation. Now all the relevant information to understand how the simulation works, concerning the implementation, are given. The next chapter deals with the analysis of the simulation results that are retrieved from performing simulations with the described software implementation.

4.7 Simulation Data Analysis

This chapter presents the simulations that are performed with the described simulation framework and the developed use case. The analyzed data is generated by the simulating agents and the according test service during simulation runtime. The accuracy of the gained data is limited by the following issue. As the computer hardware uses internal clocks that are millisecond (ms) accurate, all the time periods are measured and displayed in ms. Every simulation that involves time measurements is facing problems with the underlying clock asynchronism between the executing computers. This results, for instance, in negative transaction times for the service

execution. See also Section 4.3 for details. To avoid this problem, the time periods are calculated directly by the clock owner by sending its own timestamp and getting it back with the response message. By subtraction of the received and current timestamp of the clock owner, an exact time period can be calculated. The following simulation scenarios outline the way in which the infrastructure is tested and introduce the key parameters that are considered important for the analysis of the gained data and the evaluation of the infrastructures performance.

4.7.1 Simulation Scenarios

The simulation scenarios presented are examples for simulations that can be performed with the *EvESim* framework, but there are many more possibilities. In general, the simulation scenarios can be categorized in two classes. The first one is the network infrastructure class where the distribution of the simulating nodes within the framework varies, aiming at the testing of the service infrastructure itself. The second one targets at the testing of the service and its capabilities resulting in a variation of service using agents and documents. One actual simulation contains both classes and therefore, shows results for them. There are configurations that belong to the first class and are categorized in three different network distribution scenarios which are local area network (**LAN**), inter local area network (**iLAN**) and wide area network (**WAN**). These scenario denotation refers to the distribution of the simulating components and are described in the following.

In the **LAN** scenario the nodes and the services performing the simulations are situated physically in the same network, meaning directly connected via cable. Naturally this configuration is the easiest one to be established but not the one that is most likely to happen in the real-world, where nodes are normally connected via the Internet. The precise node distribution is as follows. The document editing service and one of the simulation nodes is situated at the same physical hardware. Another simulation node is situated in the same network, connected with the first one via cable and a network switch. This configuration should demonstrate how fast the infrastructure is working when the network delay is minimized. Nevertheless, this configuration is not useful for gaining data about real-world usage as this infrastructure is intended to be used via the Internet. To approach this real-world usage, the next used scenario distributes the simulating entities over different LANs.

In this **iLAN** scenario the service is deployed on a server situated in another LAN as the two simulating nodes. As the communication is performed over net-

work bounds and firewalls, it gives a more realistic picture of the infrastructure performance. The communication is still performed within a local network area as the different LANs are physically situated in the same building. This scenario has been chosen to give an impression of how the infrastructure is working under normal network conditions but still without using the Internet. It illustrates how heavily network barriers influence the performance and the reliability of the infrastructure.

Finally, the biggest network scenario, in terms of physical distribution, is the **WAN** scenario. In this case the service is placed on a standard personal computer that is connected to the Internet via an Internet Service Provider (ISP). The simulating nodes are, as in the previous scenario, situated in the same LAN and connected to the Internet via a company infrastructure. This scenario shows the infrastructure's performance within a real world test over the Internet and through various network boundaries and firewalls. The obtained data gives an impression on how stable and efficient the infrastructure is working within normal service consumption over the Internet.

The second class deals with the capabilities and performance of the service itself. The configuration of the testing agents influences the result. The agent can be configured by editing its execution period time function and additionally, the probabilities for the different service events. These properties are described in more detail in Section 4.2. Moreover, the number of agents that are involved in a simulation is influencing the results as there are more actions taken on the documents and the service has to handle higher request loads. The actions performed by the agents are also explained precisely in Section 4.2 and the service request handling accordingly in Section 4.3. As there are a lot of possibilities to design the simulation on the agent side, it is obvious that not all ways are described. The next section focuses on the analysis of the data gained from one specific simulation in all three network configurations. The results achieved are exemplary, but give an impression on the overall behavior and capabilities.

4.7.2 Simulation Analysis

The results presented in this section are gained from the simulation case which is shortly described in the following. There are three simulating entities which are the service deployed within a peer and two independent *EvENodes*. The three distribution configurations for these entities used to generate the presented results are described in Section 4.7.1. Every simulating node contains two simulating agents

that have the same execution period time function. This function is illustrated in Figure 4.4. As shown, the agents are configured to change their behavior two times during the simulation. Furthermore, every agent includes three demanded services which serve as documents that are going to be created with the service requests. Lastly, the agents are set up with event probabilities of twenty five percent for a creating event, seventy percent for an editing and five percent for a deleting one. The deleting probability is set very low because if all the documents are deleted soon after the creation, no actions can be taken and all the following requests would fail. In the following the simulation results are for the three distribution scenarios are presented and analyzed.

The first set of results covers the performance and reliability of the tested service. To define a parameter that illustrates the performance and reliability the transaction time is chosen. This execution process includes the time which the request needs from the simulating agent to the service and after request handle back to the agent. The Figures 4.5, 4.6 and 4.7 show the times that the transactions of every agent consumed. The results are split into three diagrams and can be analyzed.

First, the LAN scenario which is displayed in Figure 4.5 generated results that underline how the service is working under local circumstances. The first three transactions show significantly high transaction times beginning with twenty seconds, but dropping fast to five seconds at the fifth transaction. This phenomenon is infrastructure-specific and appears within every scenario. It results most likely from JXTA internal processes for the mapping of the communicating partner and the service retrieval that has to take place. As explained later, the transaction times are fluctuating in this period between three and five seconds. Stating that the transmission delay generated by the network is extremely low within the same LAN and even close to zero on the same machine, these transaction times are not remarkable. As the service does not need seconds to fulfill its requested task, the time is lost in the infrastructure. Because the mapping and coordination of the underlying JXTA infrastructure is quite overloaded, the reason is likely situated there. Despite, the infrastructure has worked stable and all the started transactions were executed successfully. The corresponding operation intervals are shown in Figure 4.8 and described later on.

The iLAN scenario gives simulation results for the service situated on a server infrastructure and its results can be seen in Figure 4.6. Also the significant high starting transaction time is even higher than in the local scenario about thirty to

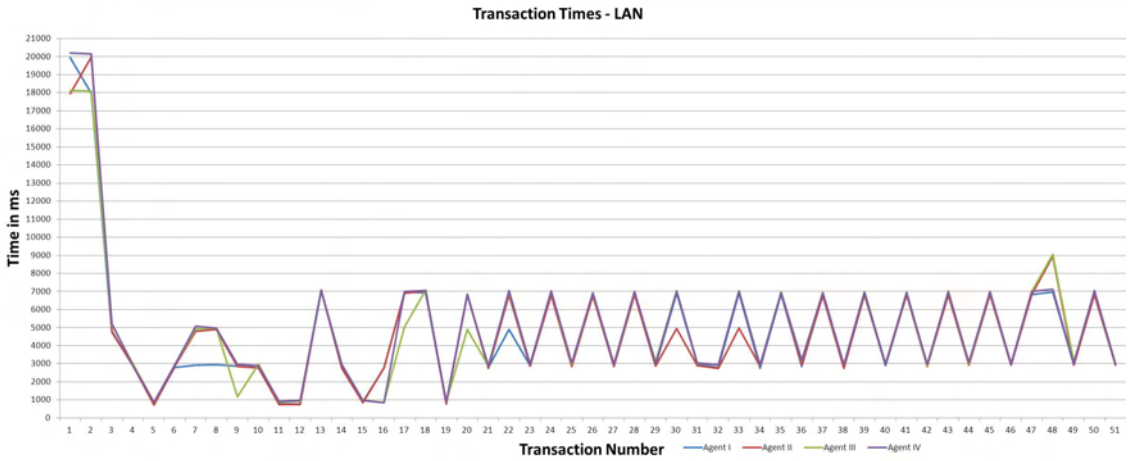

Figure 4.5: Transaction Times LAN

thirty three seconds. The transaction times drop after that but not as fast as in the local scenario. At the eighth transaction the downturn is finished and the fluctuation starts. In comparison to the local scenario the minimum is lower but the margin of deviation higher. The maxima are reached between seven and eight seconds, however most of the transactions are below seven seconds. Compared with the local scenario it is significant that the transaction times are not much higher but the initialization times and the deviations are. These times can be related to the network boundaries and firewalls to be bridged. Concerning reliability, the results are not as good as with the local scenario. There are seven transactions that have not been performed successfully. This is a high loss rate of fourteen percent. Additionally this simulation is not designed to bring the infrastructure into real troubles as the execution periods are quite high.

The iLAN scenario has revealed quite a bad reliability, but how is the infrastructure performing with an even distribution over the Internet in the WAN scenario? The results of this last scenario are illustrated in Figure 4.7. As visible in the scenarios before, the transaction times start with a high value. For the first time there is a distinct difference between the two simulating nodes concerning starting transaction times. On the one hand, the first node's agents are starting around thirty four to thirty five, but on the other hand, the second node's agents are starting around twenty seconds. After the initialization phase which is equal to the previous scenario, the transaction times fluctuate irregularly between two and ten seconds. Concerning reliability, the loss rate is also fourteen percent, which means that seven

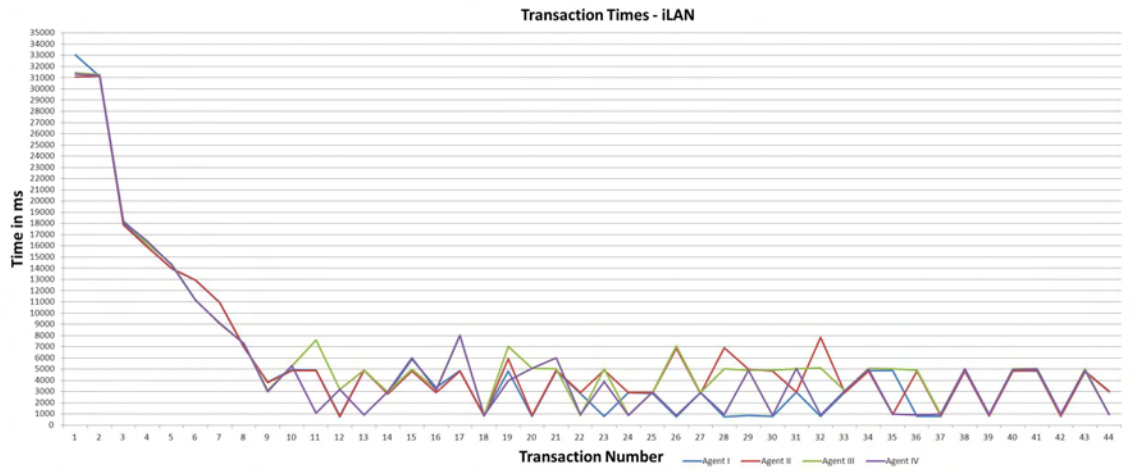

Figure 4.6: Transaction Times iLAN

transactions failed during the simulation process. Although, the simulation was executed over the Internet, the transaction times are not significantly higher than in the previous scenario. Summing up, these three network distribution scenarios

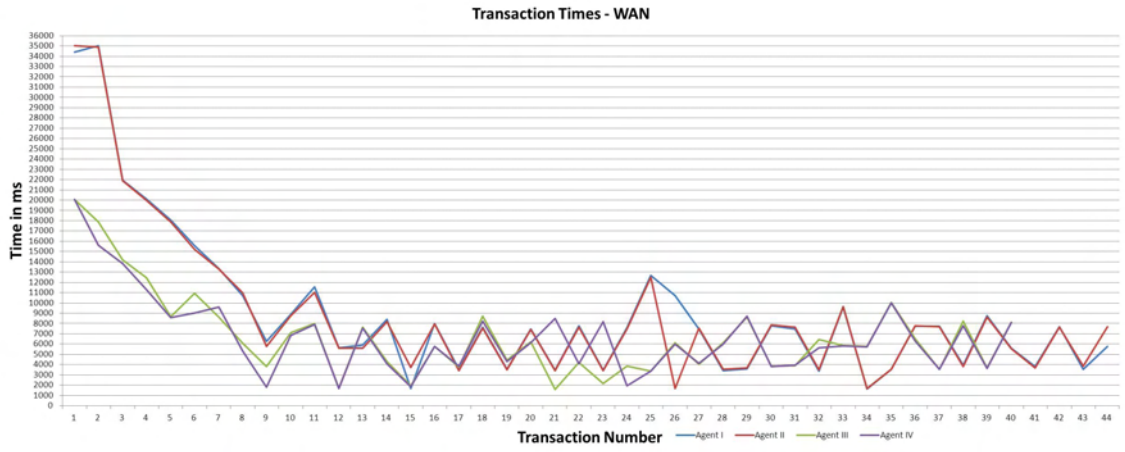

Figure 4.7: Transaction Times WAN

test the performance and the reliability of the infrastructure. Only the local scenario comes up with a full reliability, without a loss of transactions. This means that the infrastructure is not fully developed and needs further optimization. Especially the stability has to be optimized for example by providing a more stable set of *rendezvous peers*. Concerning the independence of the underlying network, the infrastructure is well bridging barriers which are caused by network boundaries

and firewalls. After the description of the second set of results the final evaluation resume is given.

The second set of results is presented that covers the performance and reliability of the document editing service, which is deployed in the above tested infrastructure. The parameters that show how well the service is dealing with document action service requests, are the periods between the arrivals of the requests for the same document. These operation intervals are calculated by taking the deviations from the timestamps of the incoming request for every document handled by the service. This data makes visible whether the service is able to deal even with very short incoming periods and what happens, if two service requests are approaching the service at the same time.

The operation intervals from the LAN scenario are presented exemplary and because of the appearance of a request interference. The presented data is shown in Figure 4.8. The three data sets represent the three documents that were handled by the service and make clear the periods between the actions taken to the document. These periods are widely spread between zero and twenty seconds, but mostly situated below the ten second mark. There are eight situations where the service faces a request appearance for the same document at the same time. As the internal service handling of the request is threaded, this situation can be managed. Nevertheless, at the eighteenth operation period, a zero time occurs and the service edits the document accordingly. However, the two threads are interfering, which produces the output displayed in Listing 4.2. In line two the interference is clearly visible as the two edit lines are combined and the information of the first one is partly destroyed. This event shows that the service has no internal scheduling for the requests and that an overlay of the requests is possible. This emphasizes the need for a locking model to assure that the document is edited properly without the interference of another request. Concerning the stability and reliability the service performed well, as all the other requests that were reached the service were managed properly. The data sets listed in the appendix basically look quite similar to the one presented, but as the simulation is probability based, vary enormously in terms of operation numbers.

```
1 DocumentAgentI,Edited,-303306,Thu Apr 15 16:04:51 CEST 2010,1271340291781
2 DocumentAgentI,Edited,6813,DocumentAgentII,Edited,6797,Thu Apr 15 16:04:57 CEST
      2010,1271340297781
3 DocumentAgentI,Edited,-303255,Thu Apr 15 16:04:58 CEST 2010,1271340298562
```

Listing 4.2: Service Request Interference

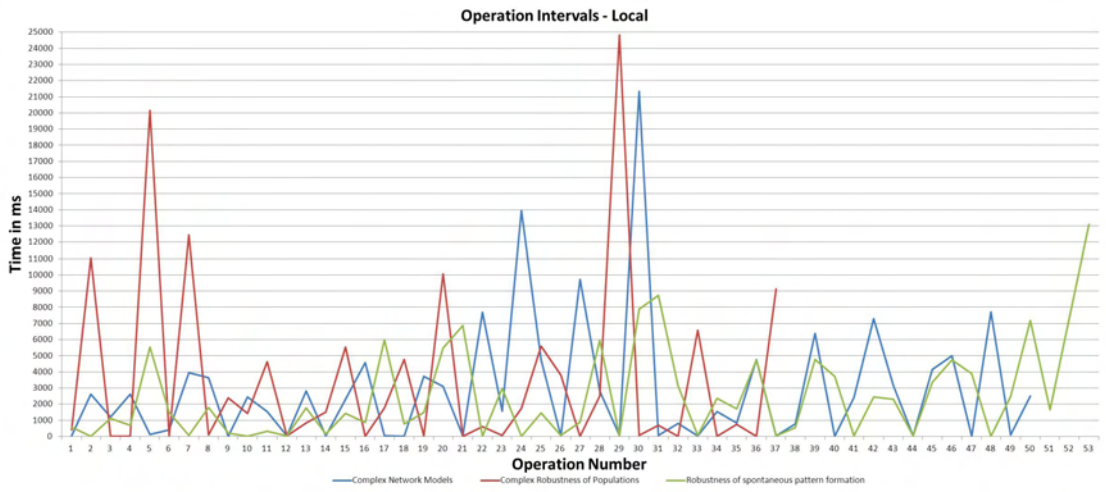

Figure 4.8: Service Operation Intervals LAN

After describing the service behavior under normal circumstances, a second series of simulations was executed in order to test the service behavior under stress. The following simulation results were conducted in the iLAN scenario, which is described in Section 4.7.1. It shows very similar results as the WAN scenario, but is much easier to accomplish. The setting is described as two agents are maintaining three documents on the service, but due to low editing periods the service is put under stress. The following three tests demonstrate different ways of putting the service under pressure.

In the first stress test, the agents start with a low editing period (200 ms) and reduce it with one ms per period. The Figure 4.9 illustrates that the service periods circulate on a very high level of approximately 60 seconds in comparison to the transaction times given in the figures above. Therefore, the service has problems dealing with requests that are arriving constantly at very short notice. It is flooded with requests and needs lot of time to recover and execute them properly as well as send back the result. Additionally, the 200 requests taken do not reveal transaction periods below 40 seconds which is an extremely high period. Although being flooded, the service achieves a correct execution of the editing request. To conclude, starting with a short period of requests brings the service into trouble.

In the second stress test, the agents start with a higher editing period (500ms) and reduce it with one ms per period. The outcome of the simulation shown in Figure 4.10 shows that the service is capable of dealing with the requests properly until the editing period reaches approximately 250 ms. From that point on, transaction times

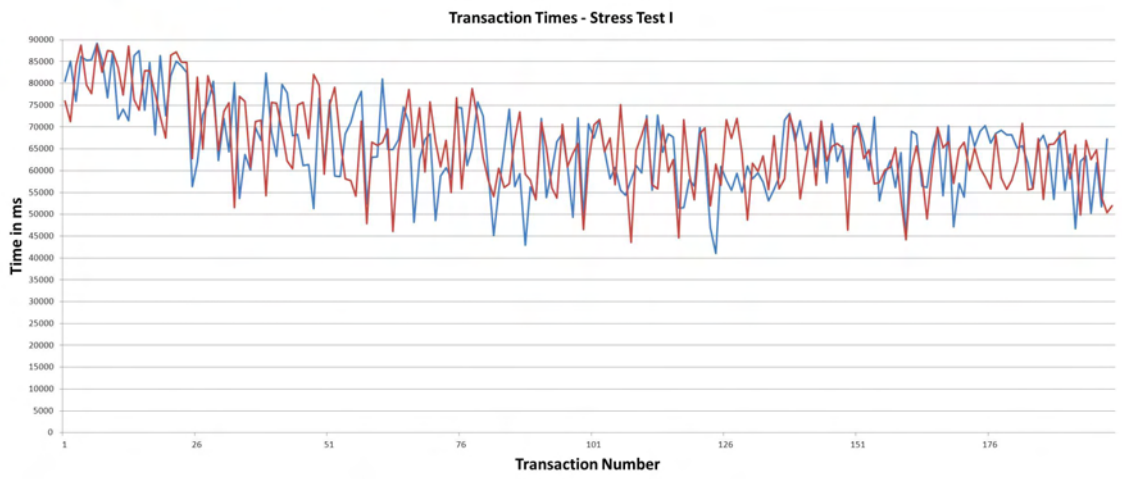

Figure 4.9: Transaction Times Stress Test I

are ten times higher. They are not reaching the level of the first test, but circulate around 40 seconds. This simulation exposes that the service is more capable of shorter editing periods, if there is a slower approach to them. To sum up, the second test discovered the critical editing period. When this period drops below 250 ms, a severe transaction time increase is very likely. Moreover, it pointed out that the service deals better with that after a softer period shortening.

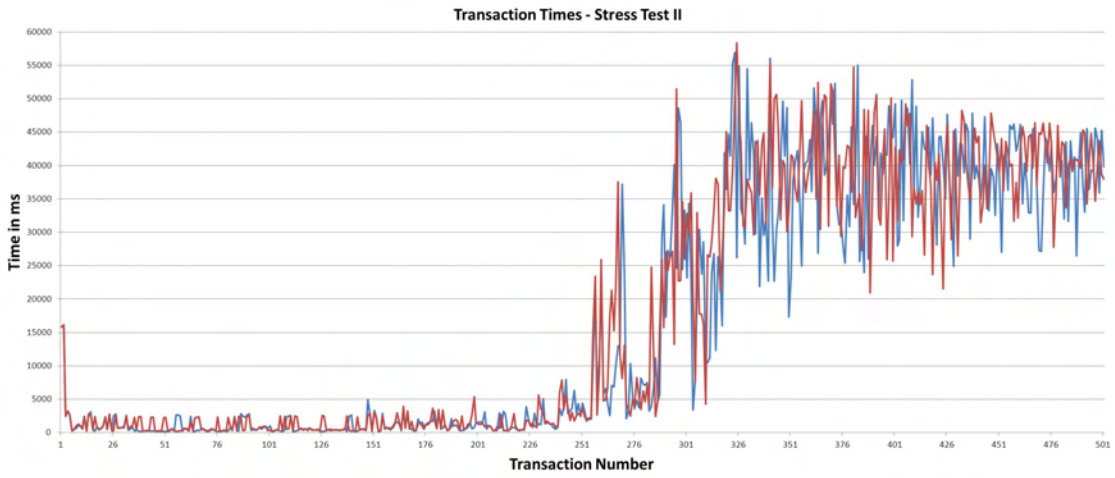

Figure 4.10: Transaction Times Stress Test II

In the last stress test, the period is set even higher at 1000 ms and as usual is reduced with one ms per period. The produced data is visualized in Figure 4.11. The editing period starts with 1000 ms, is decreased very slowly, giving the service the

time it needs to deal with it. Therefore, the transaction periods circulate around two to four seconds even when reaching the critical time of 250 ms. After this point the transaction time increments but far less tremendously than in the test beforehand. To conclude, these tests show that the service is highly dependent on how fast the periods of the editing request change. The higher the starting period and the slower the time change per period, the more likely it is that the service is not suffering in terms of transaction periods.

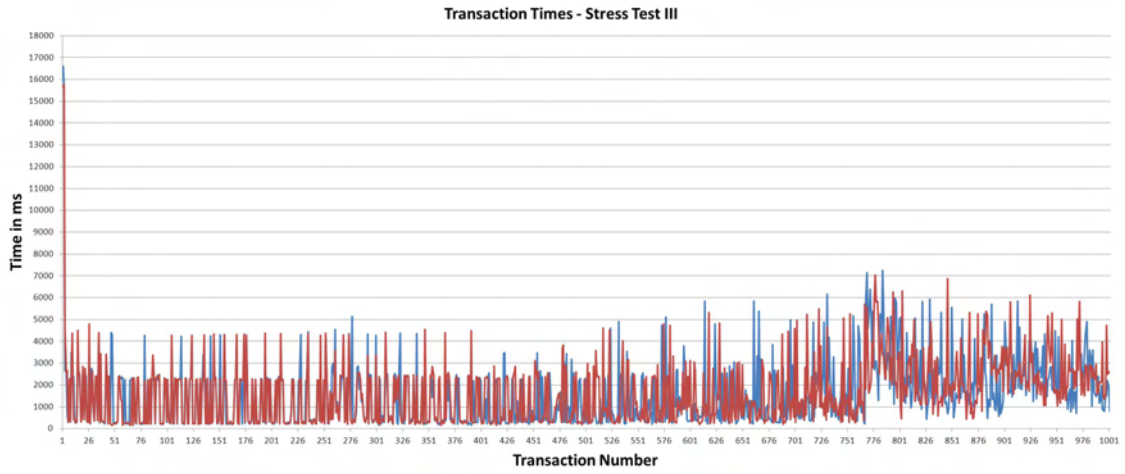

Figure 4.11: Transaction Times Stress Test III

Figure 4.12 demonstrates that the service manages the service requests properly. It illustrates the editing operations taken by the third test. Although, these 1000 editing operations are executed within a very short time, the service is able to handle them and produces a correct file output. In a nutshell, the test identifies the transaction time as the crucial factor for services within the *Flypeer* network.

This chapter finalizes the report and displays the outcomes of the simulation use case design for testing the *Flypeer* infrastructure and the document editing service. Concluding the described simulation results, the infrastructure performed well in terms of network boundaries and firewalls as they do not harm the operation of the infrastructure but do influence the transaction times needed. This is the main advantage that can be derived from the simulation results. Concerning the actual performance in terms of transaction times the infrastructure performs quite badly. As the design was not intended to be rapidly executable, but more focused on the implementation of a transactional model, this performance is understandable.

However, the infrastructure certainly needs optimization before being used in

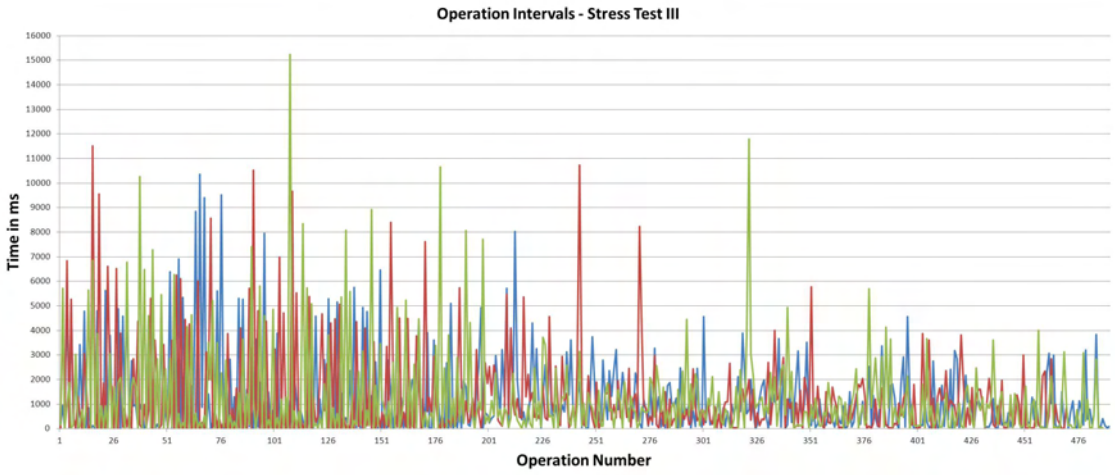

Figure 4.12: Operation Intervals Stress Test III

an industrial environment. Also the loss rate is too severe for industrial usage of the infrastructure. The document editing service performed adequately and handled even simultaneous request properly with only one exception where interference happened. This problem is solvable by introducing a lock model for the data. The stress tests revealed that there is a critical level of editing period time and that the approach to this level influences the way the service is able to deal with it.

5 Acquisition of Social Network Data

This chapter describes the acquisition of social network example data out of real world sources and the further processing with the Java Universal Network and Graph framework [3] (JUNG). It contains the agents that are representing data from the social networks Facebook[1] and Guigoh[2] as well as the graph visualization with JUNG and also components included in this agents such as a timer, connections with friend agents, the community and the task to consume a collaborative document editing service.

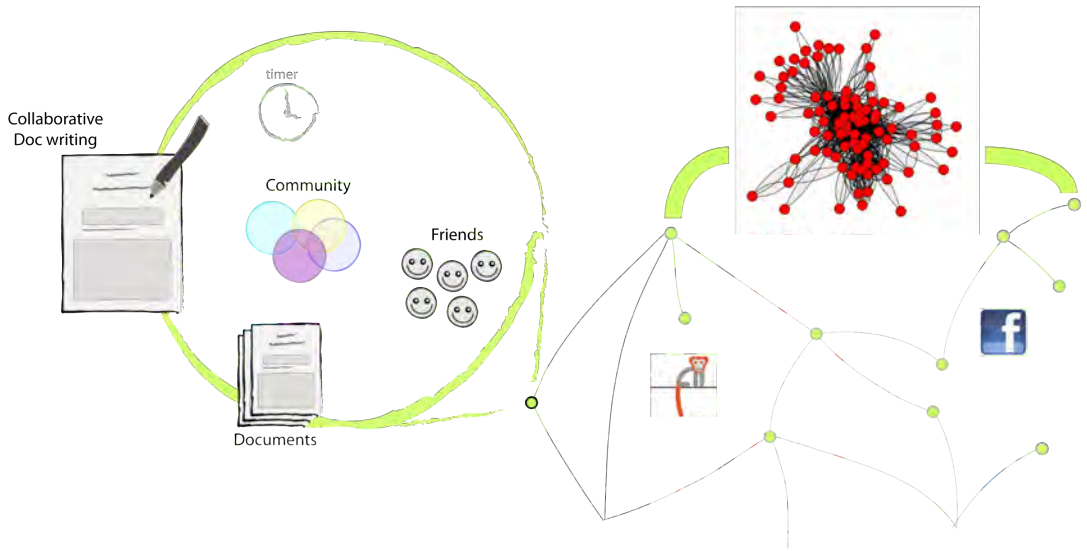

Figure 5.1: *EvESim* SNA Model adapted from [24]

[1] `http://www.facebook.com/`
[2] `http://www.guigoh.com/`

5.1 Acquisition of Social Network Data

As a starting point for the framework extension, which is described in the thesis at hand, there has to be network data that can be used as a basis for SNA research. This data can be either randomly generated or reused from existing real-world systems. In this context, a real-world system means an online social network. There are several online social networks hosted in the Internet with various objectives. In order to get data from one of these networks it has to fulfill two requirements. First, it has to be easy and cost-free to become a part of the network. Second, the online social network has to provide an Application Programming Interface (API) to enable the acquisition programmatically.

Two representatives of online social networks were chosen because they comply with the requirements and they provide easy to use APIs. On the one hand, the online social network Facebook (400 million users[3]) and on the other hand, the one provided by OPAALS named Guigoh. The structure and appearance they have is described in Section 5.1.1. The process of acquisition is also explained in that section.

An online social network is a web-based service that builds a community. It allows participants, who are living individuals, to construct a public or semi-public profile within a delimited system and a list of other users with whom they share a connection. This connection basically means a relationship which can either be some kind of friendship or working relationship, or even an interest both of them share [25]. Each participant is creating his own small sub-network of connections within the entire network of the community. This is exactly the network data used for demonstration of the SNA extension. The data consists of nodes which are the participants in the network, and the connections between them. To enable the framework to visualize the data in a geographical way, every node has location information included. In order to store and transfer that data in a standardized way an XML data model, named *NetworkDescription*, in form of an XML Schema (XSD) was developed. This schema specifies the way of saving the network data information in XML format [26] and it is shown in Listing 5.1. It is one node taken from a network that has a geographical location represented by coordinates and a connection to another user.

```
1    <node type="Person" uri="Christoph.Ruecker" name="Christoph Ruecker">
2     <name>Christoph Ruecker</name>
```

[3] http://www.facebook.com/press/info.php?statistics

```
3       <location lat="47.724339" lng="13.08631" name="Puch, Urstein"/>
4       <connections>
5         <connection strength="4" to="Thomas.Kurz" type="paperContribution"/>
6       </connections>
7      </node>
```

Listing 5.1: NetworkDescription.xml

Before the framework has been redesigned, the internal data model and the *NetworkDescription* data model were equal. During the redesign of the *EveSim* framework its internal data model has been adapted to EMF but to assure compatibility to formerly created data the *NetworkDescription* model was not adapted. This results in the need for a new data parser to link the two models which is described in Section 4.4. To sum up, the network data used for the simulation case is described, its origin and requirements are stated and the model created for the framework is presented. The next section deals with the actual acquisition of the data from the real-world sources.

5.1.1 Guigoh and Facebook Data

Guigoh is an environment that allows its participants to be part of virtual communities, to create their own profile, join forums, chats and extend their own network. The main page of a Guigoh profile is shown in Figure 5.2. The platform is based on

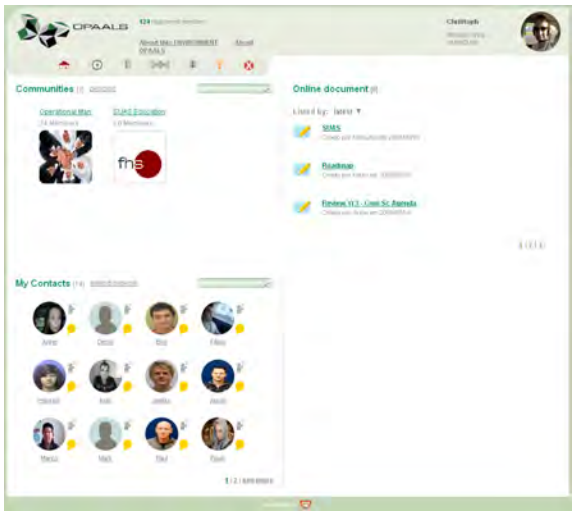

Figure 5.2: Guigoh Social Network

a software created by the Instituto de Pesquisa em Tecnologia e Inovaçã (IPTI) and it is named after a small monkey in risk of extinction in northeast of Brazil. Guigoh

has been developed in Java programming language and is licensed as open source software under the GPL [27]. It was developed to give the OPAALS community the ability to connect with each other and to share and edit documents. For this purpose Guigoh has a collaborative document tool which should have been the service to test in the thesis. As the service is exposed only inside the Guigoh network, the framework is not able to test it.

Therefore, the simulation case includes a self-implemented document editing service which can be replaced with the one from Guigoh in future. To acquire data from this social network, a specified service was created by IPTI. This service uses a valid user name and password to first connect to the system, second parses the data from the network in JavaScript Object Notation (JSON) format and finally, converts it into *NetworkDescription* XML, as described in Section 5.1. The service already existed for the former version of the framework, but was adapted to be used in the *Flypeer* environment. In addition, it calculates the geographical coordinates from the Guigoh user's friends' home places with the Google Maps[4] API for mapping places to coordinates.

Facebook was founded in February 2004 and is an online social network that empowers people to communicate more efficiently with their friends, family and coworkers. The platform emphasizes the development of technologies that facilitate the sharing of information through the social network, the digital mapping of people's real-world social connections. As anyone can sign up for free and interact with the people they know and share interests, the network has become very popular. It is one of the most-trafficked PHP sites in the world, and one of the largest MySQL[5] installations anywhere. It is built on a lightweight but powerful multi-language RPC framework that allows the platform to seamlessly and easily tie together subsystems written in any language, running on any platform. In addition, it includes a development platform that enables companies and engineers to integrate their programs with the Facebook website and therefore, gain access to millions of users [28].

The provided API is used in order to create an application that is able to export the social network into a *NetworkDescription* XML file. The only prerequisite is a valid user is logged in and agreed that the application has the right to read from the user's data to generate the network information. Facebook can be accessed by every valid user at `http://apps.facebook.com/evesimexporter/`. In principal,

[4] `http://code.google.com/apis/maps/documentation/reference.html`
[5] `http://www.mysql.com/`

the application uses the PHP API from Facebook to traverse through the user's friends' data and generates the *NetworkDescription*. Given the opportunity to access the data of the user, the application can also identify the relationships between the user's friends' to generate an accurate digital mapping of the user's social network. To acquire the geographical coordinates of the user's given position data, the Google Maps API is used as well.

Summing up, this section describes the data model for the framework as well as the types of data that are stored. Furthermore, two real-world social networks are shortly explained and the possibilities of acquiring data are stated. Finally, the service for data acquisition from Guigoh and the application for data retrieval from Facebook are mentioned. As the data needed for showing SNA structures has already been acquired, the next chapter elaborates its visualization in graphs and the algorithms used for metrics calculation.

Figure 5.3: *EvESim* Exporter Facebook

5.1.2 Metric Calculation and Visualization

In order to realize the visualization and metric calculation, the JUNG framework is applied [29]. It enables the user to build-up a graph and visualize it accordingly. The gained data described in the section above is parsed to determine the vertices and edges. After this process is finished the graph visualization is generated. Using the parsed data enables the JUNG framework to calculate the specific metrics for the network. All metrics are put into relation to its equivalent calculated from a random graph which has the exact number of edges and vertices as the one displayed. The

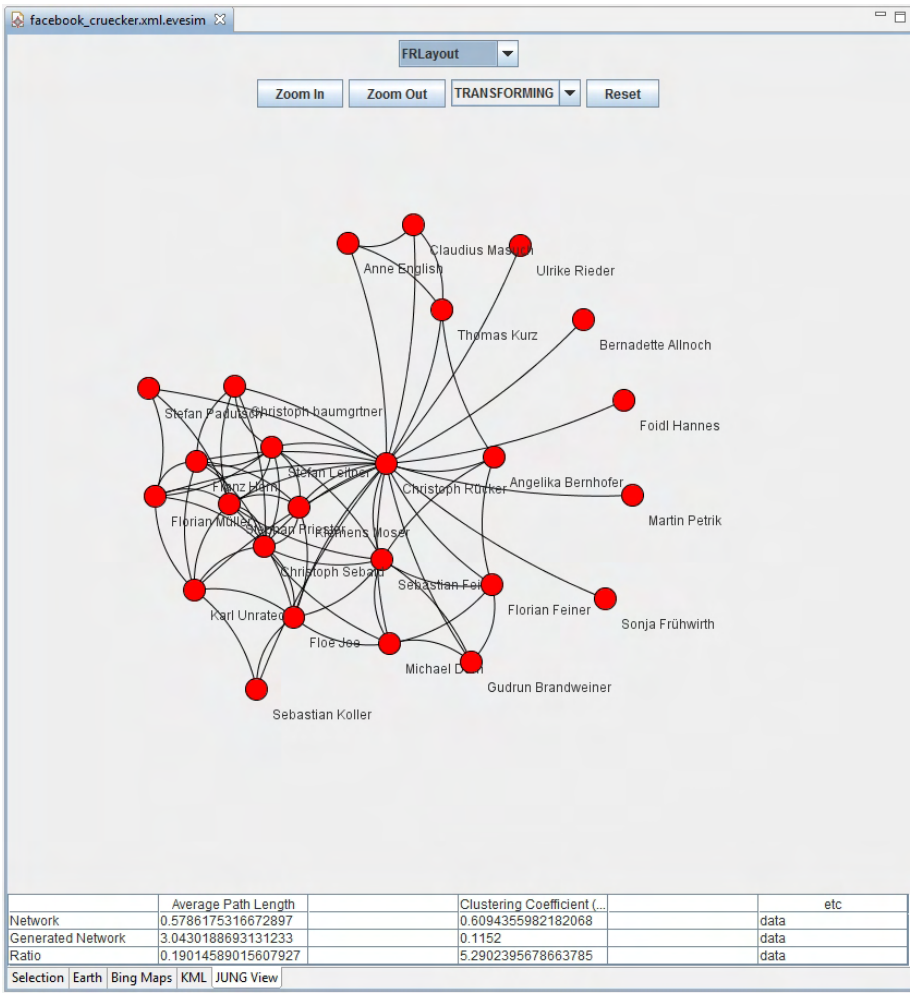

Figure 5.4: JUNG Visualization with Metrics

complete embedded JUNG view including graph and metrics is shown in Figure 5.4. Further development will result in an extension of the calculated metrics.

6 Evesim for Social Network Analysis Practitioners

In deliverables 3.7 and 10.11, from phase II, some conceptual and theoretical issues about SNA were presented, including case studies of many different types of networks, from co-authoring of scientific papers to interactions among discussion forum participants. In these, different measures and properties were used to analyze the networks, mainly their small-worldness and scale-free properties, and in some cases the modeling of their dynamics over time using statistical models. Other deliverables from OPAALS also presented SNA for different purposes, mainly D11.01, D11.20 and D11.08.

In all those cases, the analyses illustrated have some similarities, although they focus on different ways to look at the network. This happens due to the analysts point of view, the type of network, or both. A SNA toolkit provides a diverse set of methods and metrics, for the actors or the structure, summarizing properties in numbers or presenting them in visual graphs.

In all cases, computer programs were obviously the main resource to conduct research in networks, since all algorithms or calculations are complex and time consuming to perform without a computational implementation. In case of D3.7 and D10.11, were these authors had lead and/or collaborated, 4 different softwares and packages were used to provide all informations required to answer the scientific questions. So, is the development of a new toolkit for SNA really necessary? Is EvESim contributing in some way to the SNA world, since a lots of computational resources are already available? This was one of the reviewers questions in the Phase II review, held in LSE - London (UK). And this chapter intends to justify, more than just answer, the necessity of an integrated computational solution for SNA practitioners.

6.1 SNA Practicing

Who are the network (not just social) analysists nowadays? What type of questions, researchers want to answer with SNA? Since the development of the first techniques for graph analysis, in early 50's, these methods started to be applied in a huge variety of researches, involving human interactions, companies business and partnerships, computer networks, DNA analysis, metabolic networks, web pages connections, and many other types. So, it is easy to imagine that the practitioners profiles varies a lot, since we are talking about social scientists, biologists, computer scientists, mathematicians, statisticians, and researchers from many other scientific areas.

In earlier studies, the size of networks in study were not too big, rarely reaching hundreds of nodes. Size was mostly limited by nature of the interactions of interest, mainly social and friendship interaction, and of course by the data collection process, having questionnaires applied to actors in interviews or sent by mail as the main way to collect network data. As said, great part of those studies were related to social questions, trying to explain power or authority by means of centrality measures, homophily based on social characteristics of nodes, resilience examining the effect on the network structure when drooping node(s), among others. Visualization was also extensively used, having the human eyes and perception as the main tool to discover and describe motifs and sub-structures.

Recent years however have witnessed a substantial new movement in network research, with the focus shifting away from the analysis of single small graphs and the properties of individual vertices or edges within such graphs to consideration of large-scale statistical properties of graphs. This new approach has been driven mainly by the availability of computers and communication networks that allow us to gather and analyze data on a scale far larger than previously possible. Where studies used to look at networks of maybe tens or in extreme cases hundreds of vertices, it is not uncommon now to see networks with millions or even billions of vertices. In these cases, questions dealing with effects of a dropped node and measures of centrality loose their meaning slightly, and visual inspection is mainly impossible. So, how to analyze a network structure without seeing it, or even if it could be seen but is not understandable?

Methodological development tried to follow the demand of new network analysis interests, reaching answers for the new thousand or million actors network. Uncertainty is now part of the game, and statistical models for network metrics and newly developed properties started providing new perspectives for SNA. Any of the metrics,

used in SNA, could be modeled as statistical models provided feasible assumptions, such that, Degree Distribution, Clustering Coefficient Distribution, estimators for average path length and maximum degree. Also, new properties were developed, like assortativity, preferential attachment and modularity. As Newman [30] states, statistical models answer the question, "How can I tell what this network looks like, when I can't actually look at it?"

Another aspect that statistical models permitted, as a natural development, was the study of dynamics of networks, longitudinal aspects of the network evolution over time. Before this, just the structure final of a network has had been studied, and sometimes just different structures in different discrete time points were compared, but the underlying process of the evolution had never been assessed. Tom Snijders and collaborators could be considered as the main group in this recent development [31, 32].

If it is now possible to look back the history and understand the dynamics, or even follow the network in real time extracting evolution properties, so look forward is again a natural extension for the SNA practitioners. So, simulate further steps of the evolution, based on what was observed and learned, could bring new insights about scalability and resilience (for example), the simulation and observation of new scenarios assuming some disturbances in the underlying process could help in understanding the impact of interventions in the network, among other research interests.

6.2 A brief review on SNA Computational Resources

The computational skills of those SNA researchers also varies, as long as their science area does not demand general or specific knowledge on computer programs or programing languages. The International Network for Social Network Analysis (INSNA), presents in its webpage an extensive list of softwares available for SNA (http://www.insna.org/software/index.html). This list cover 28 stand alone softwares or library toolkits (that works under another software like R statistical package or Excel).

Huisman et al. [33] made a very helpful and serious review based on the same list, but around 6 years ago, presenting features of 29 softwares, but analyzing extensively just the main six, namely, UCINET, PAJEK, STOCNET, NetMiner, MultiNet and STRUCTURE. They state in the conclusion that

"Obviously, we try to present an objective, substantiated view, but we admit that we cannot give a completely unbiased opinion. We also stress that it is impossible to make a fair comparison between the packages, because their objectives are different, which leads to different functionalities"

and after a scoring on the six software analyzed, they conclude:

"It remains, however, hard to compare the different packages, as we already pointed out at the beginning of this section. We leave it to the reader of this chapter to decide which software to use for the social network analysis s/he wishes to do."

But, for the purpose of this deliverable chapter, it is sufficient to say that no one software gather all available functionalities, all types of modeling, visualization and data handling, and given the great development of SNA methods and techniques, it is unlike that a unique software will once in a close future, cover all specificities, satisfying the general audience of SNA practitioners.

In the deliverables cited above, the authors used 2 different stand alone softwares and 3 different R [34] packages, each one attending to a specific demand. Pajek [35] was used to handle raw data, where specific subsets of actors were used, and also for some visualizations when it resulted in better results. To perform scale-freeness and small-worldness assessment, packages *igraph* [36] and *netmodels* [37], running under R that read directly from Pajek ".net" data format. For longitudinal modeling, SIENA program [38], that run in StocNET [39] software, was the the tool, using data exported from R, once they were imported from Pajek and transformed in form of adjacency matrix.

All the analysis steps depicted above demanded, despite SNA concepts previous knowledge, experience with data manipulation in Pajek (that might be considered a user friendly application) and programing skills in R (not so friendly). For practitioners from social sciences, for example, without familiarity with structured programing, it could be a hard task, making research efforts even more difficult.

6.3 Evesim perspectives from a SNAnalyst

The development of EvESim started during the DBE project, aiming to provide a simulator of agent (or actors) behaviors in an evolutionary environment, but also to provide researchers in the project, a framework to collaborate and test their findings. As at DBE, OPAALS also comprises the same science areas, namely Natural Sciences, Social Sciences, Business and Computing and therefore EvESim was further

developed. The new version of EvESim available for download is a bit different from the one presented at DBE, despite the fact that its main purpose remains almost the same. Although, new perspectives and demands raised from OPAALS researches and experiences, this new version presents a more analytical framework, including SNA metrics and methods.

As cited in the EvESim website (www.evesim.org), the stakeholders descriptions make the new purposes more clearly. There are three main types of potential users:

Analyst

We refer to analysts as experts from social network analysis. The target stakeholders in this area have usually little or no programming skills but they have much knowledge as regards the analysis and interpretation of network data. Analysts have already well-established simulation frameworks and analysis tools at hand, but in some cases they run out of computing power or they need customized visualization and analysis functionalities beyond the level these current tools offer.

Catalyst

In order to implement customized simulation cases, analysts take advantage of simulation catalysts. Catalysts are maintaining and extending the framework. By implementing easy-to-use front-ends for the simulation and emulation of social networks, they extend on the one hand the present framework in order to allow analysts to run simulations on different distributed infrastructures. On the other hand they are exposing an interface for the instance creators for providing different infrastructures.

Instance Creator

The so-called instance creators are providers for implementations of distributed service architectures. To give an example, this could be a provider for a distributed social network or a P2P network implementor. The interest of this group in the present framework might be to access data of existing social networks in order to emulate their infrastructure based on existing network topologies and behaviors.

Also, a good example of collaboration:

Research and development of a medium-sized enterprise needs to develop a collaborative document editing tool that works on a P2P basis. The analyst evaluates the behavior of potential users, collaborating on documents and papers. One existing research network is used for acquisition of real-world data as input for the simulation network. In the following, the peer-to-peer system is chosen as infrastructure, which is also used for the potential underlying infrastructure of the distributed collabora-

tive document editing tool to be developed. Now the enterprise can run emulations of the document editing tool based on social network analysis data on top of different peer-to-peer systems in order to decide on the best configuration and optimum infrastructure.

EvESim empowers its stakeholders to import data, learn properties and simulate from them, integrating this three steps in one framework. However, nobody could claim that it is a *panacea*. Specificities of research, new methods, different views of the same problem, the multidisciplinarity of the 21st century science, and many other idiosyncrasies will always push up the horizon for networks analytical tool developments (like the horse's carrot).

7 Related Activities

This chapter intends to introduce and describe the various additional activities that were conducted around EvESim to emphasize the multi-domain usabilities and capabilities.

7.1 BioTec Visualization Ireland

In the course of a research exchange with the National University of Ireland, Maynooth in May 2010, EvESim has been reconfigured to serve as a visualization tool for the BioTec industry in Ireland. There were several challenges that have been solved to facilitate this capability. The specific data provided was mainly given in plain text files or in social network common Pajek [35] format. To transform this data into EvESim readable one, wrappers were implemented to conduct this process automatically. As the data from the Irish BioTec industry in divided into different parts the wrapper was kept generic to be easy adaptable. Three different scenarios were transformed into EvESim readable format. All of them use the companies as a basis and vary in the nature of their interconnections. The first one defines the connections between the companies via the directors which are working for more than one company. This dataset is shown in Figure 7.1. The second one deals with interconnections based on researchers which are or have been working for different companies. Finally, the last one is setting up connection over the patents that are created by different company collaborations. All in all, these little adaption of EvESim allowed to visualize the BioTec industry of Ireland to not only determine social network properties but also geographical features of the network.

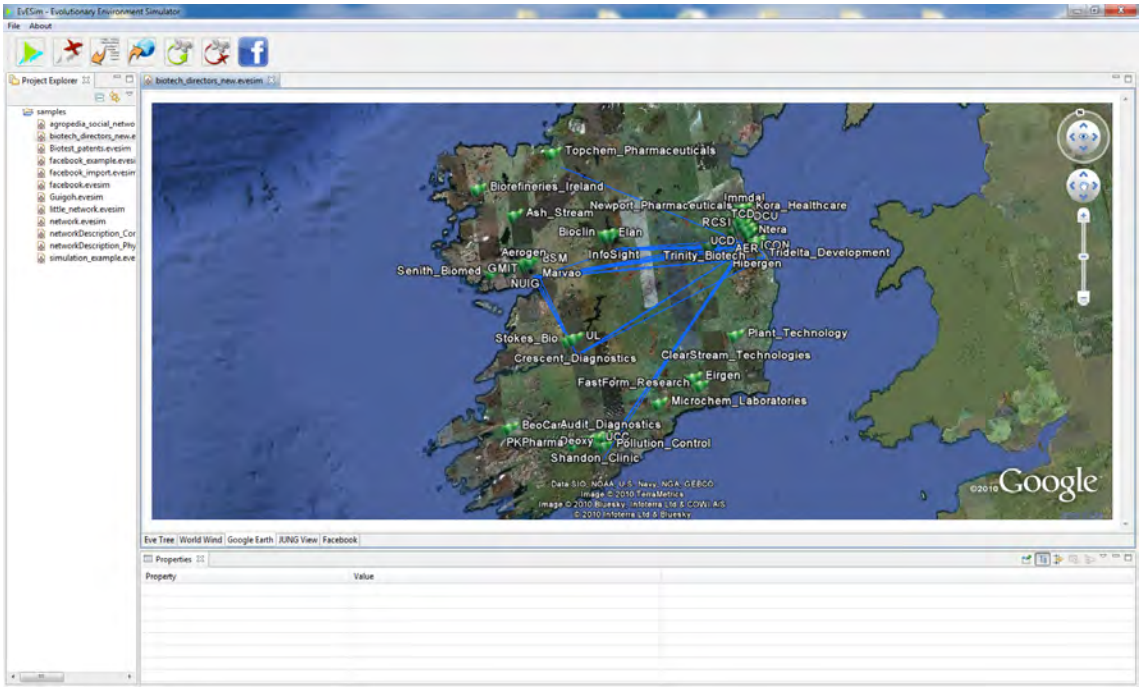

Figure 7.1: BioTec Visualization Ireland

7.2 Agropedia Social Network Visualization India

To show EvESim's capability to visualize a huge dataset there had been a cooperation with the Indian Institute of Technology in Kanpur (IITK). To enable the visualization they provided an interface to export data from their Agropedia social network. It extracts the needed data such as geo-coordinates and interconnections of the participants and encodes it into EvESim readable format. This exporter interface can be reached at `http://agropedia.iitk.ac.in/data_sheet/agropedia_sheet.php`. After small adaptations the data has been used to create visualization showing that the social network Agropedia is not limited to India but has become a network with worldwide participants. The final Agropedia geovisualization output is shown in Figure 7.2. Presenting this example for large scale data visualization would not have been possible without the special activity of IITK for providing the export interface.

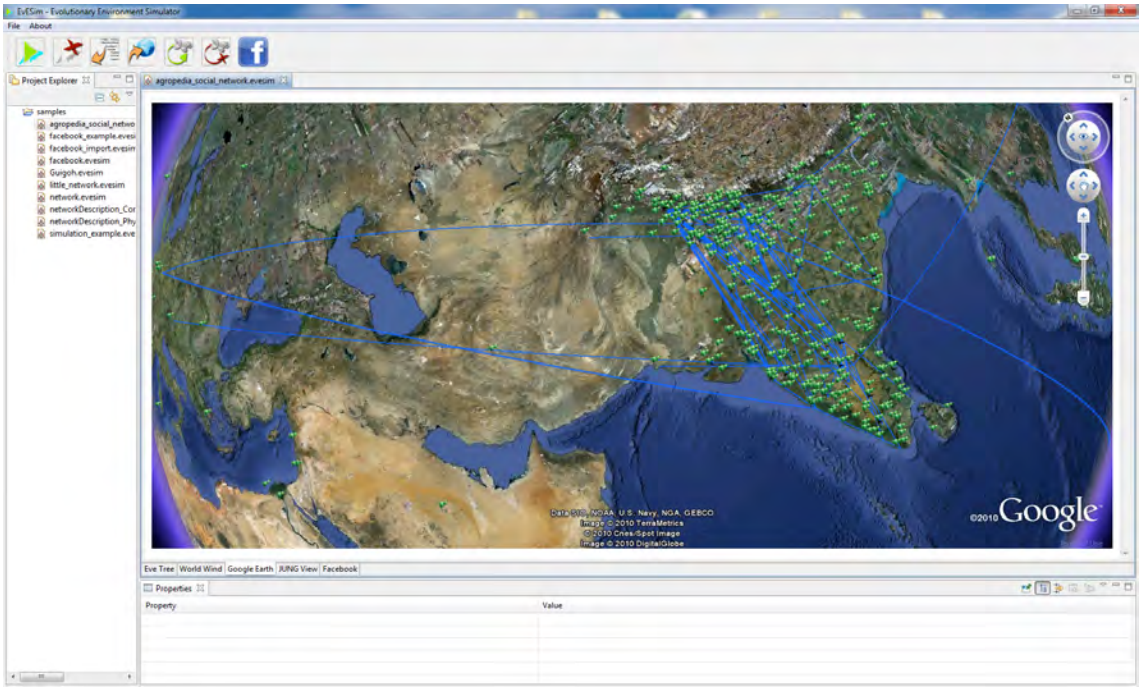

Figure 7.2: Agropedia Social Network Visualization

7.3 Monitoring

Additional to the simulation use cases, the current EvESimulator version contains monitoring functionality, which is depicted in Figure 7.3.

The monitoring is implemented via agents of type EveMonitor configured in the EvEUI. The default implementation of these EveMonitor nodes poll EvENI, which serve as agents providing monitoring information. As can be seen in Figure 7.3, the selected node is configured to poll the local EvENI (*http://localhost:8080/monitor*). Fully functional nodes are marked with green color and have the node status *ok*, whereas erroneous nodes are red and a node status of *error*. If an erroneous is available again, the node status is set to *recovered* and marked in yellow color. After a defined time period in the *recovered* state, the node state is set to *ok*. More implementation details can be found in Section 2.4.3.

If requested monitoring functionality differs from this default implementation (such as monitoring of a dedicated service or network node), a subclass of EveMonitor can be implemented and integrated into the EvESimulator monitoring framework as well.

With the resources within OPAALS for the monitoring only this monitoring

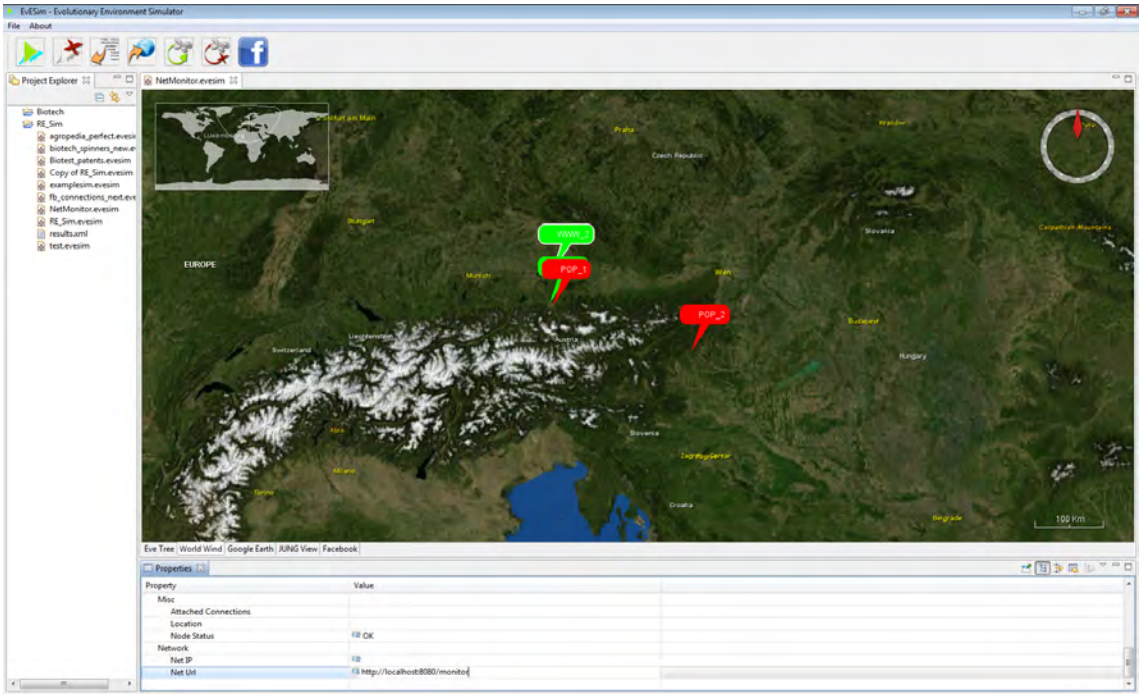

Figure 7.3: EvESim Monitor Example

example use-case could be implemented. Flypeer integration would be an feasible option for providing monitoring mechanisms within the DE software platform providing a high level view onto the globally running software infrastructure.

7.4 Flypeer Search Service

During the testing of the *Flypeer* infrastructure the missing of service search capabilities became obvious. To create these a search service has been created which is able to traverse the P2P network and list all the deployed services. Additionally, the output has been available on a webpage to guarantee a public available service listing even for non *Flypeer* network users. As *Flypeer* was in a premature and unstable state the service was not included in the official builds. During the Zaragoza Code Camp in March 2010, the functionalities were included directly into the *Flypeer* source and became part of the official releases.

7.5 Gnutella vs Freenet Search by EvESim

Parallel to the development of a simulation framework, so-called simulation cases were executed as proof-of-concept of the framework and for finding key factors for configuration and applicability of biologically-inspired *digital ecosystems*. Simulation cases are implemented simulations which are reflecting use-cases in reality. They usually are designed to emulate this use-case by letting the agents take over the actions of the actors in real life. One of the first simulation cases was the search for the most relevant factors for determination of critical mass of services and actors for digital ecosystems and self-organized clustering of SME networks to enable bootstrapping of the service environment.

The most recent simulation case in the framework is a combination of different semantic search and recommending mechanisms. It is intended to combine best practice methods for searching with biologically inspired approaches for data distribution in digital ecosystems. Here again it is intended to replace the simple keyword search component later on with P2P-based semantic search components that are currently under development.

7.5.1 Network Exploration

For the search algorithm we are using two traditional approaches from P2P systems for node lookup and utilize them for service search. First, the *breadth-first search* (BFS), which is also used by Gnutella is used as an example. The search begins at the requesting agent and explores all neighboring nodes. From there on it also explores the next level – neighbors of neighbors, until a given Time To Live (TTL) of the search request is exceeded. For Gnutella the typical maximum TTL is 7. According to [40], about 95% of nodes can be reached with a TTL of 7 [41],[42].

And second, a modified *depth-first search* (DFS), which typically is implemented in Freenet [43] is compared to the BFS. For Freenet it is typical that the network is explored node-by-node until a goal node is found. Iterative deepening is used in this simulation case in order to (i) search for services first at all direct neighbors, (ii) assess the results, and (ii) if a service with fitness higher than a given threshold is found, the search ends at this level. If the fitness threshold is not satisfied, the network is explored further in depth until again a TTL boundary is reached.

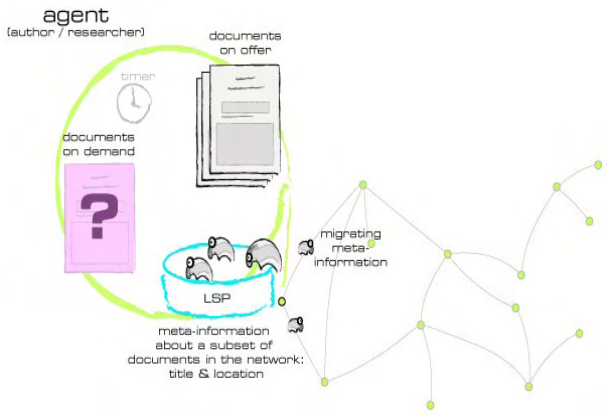

Figure 7.4: Agents using Local Service Pools.

7.5.2 Semantic Search Simulation Model

Fig. 7.4 shows the basic model of the semantic search simulation case. Each agent in the network has a timer which triggers the searches. Additionally, each agent owns a list of keywords on demand, which are requested by the agent when the timer triggers the search. Both the on-demand list and the on-offer list of documents are abstracted service descriptions, which are represented by the title of the document. Consequently, if a request for a document is triggered, the agent searches the network by calling the previously described keyword search component. This component takes as parameters the title of the requested document and a list of candidate document titles. The return value is a ranked list of documents based on the matching keywords. Currently, the agent picks the highest ranked document and remembers it together with the request. In order to enhance future searches, a new connection is established between the requesting agent and the owner of the document. This behavior leads to a dynamic change of the network infrastructure based on the documents found in the network. Connections are set up between nodes which have common interests and thus future searches speed up.

The candidates for the search in this simple scenario are the documents in the on-offer list of an agent. That means that every actor / agent in the network has a list of documents which were written by this actor. This list is searched for the requested document until the TTL of the request is reached. The results of the searches are returned to the originally requesting agent. We refer to this search as *pull-approach*, because the documents are distributed by pulling for a requested document.

Contrary to the pull-approach, we introduced a biologically inspired push-approach. This work is inspired by one of the principles of the Evolutionary Environment (EvE) detailed in [44]. The underlying concept is the one of migrating individuals in nature. In the simulation case at hand, each agent has a local service pool (LSP) (see Fig. 7.4), which represents the environment documents can 'live' in. Living in this case means that the documents, i.e. document titles, which are most probably interesting for that agent concentrate in the LSP of an agent. The interest and therefore the search requests of an agent define in that case the environment of documents. The idea in that case is to have a browseable LSP that acts as a dynamic recommender for possibly interesting readings and in parallel as the starting point for searches.

The name pull in this case is chosen because if an agent searches for a document and finds it in the network, this agent copies the document description in its own LSP and pushes it also to all its neighbor LSPs. This bases on the assumption that the neighbors of a node have also similar 'reading interests'. Additionally to this migration / propagation of document descriptions to neighbors, we introduced also the propagation to a number of random nodes in the network. That is needed in order to connect island networks.

7.5.3 Example Application

The following simulation results are initialized with a social network of co-authorship from Brazilian researchers with other authors all over the world. The data was collected by the Instituto de Pesquisas em Tecnologia e Inovação[1], in Brazil. We restricted the area of publications to Physics-related publications which lead to a network of 476 researchers. The first author of a paper is the owner of the paper and the respective agent gets an entry managing the paper title in its on-offer list. The on-demand lists of the agents are initialized with on-offer lists of other agents in the network. The LSPs are filled randomly with document descriptions in the network with a total redundancy of five. This means that every document on offer in the network is copied randomly into five LSPs in the network.

For all simulation runs, a threshold of 50% fitness is assumed. That means that an agent accepts only search results if the fittest service has at least 50% matching of all keywords within the result. If the threshold is not reached, the search will be started again later in the simulation.

Only searches which use the local service pools (LSPs) for service discovering

[1] http://ipti.org.br

reach fitness values exceeding 0.4, whereas search methods without using LSPs not even achieved a value more than 0.1. Because matched services are migrated only to other agents' local service pools, better services located further away than seven hops cannot be found. This leads to a low average service fitness for searches without using LSPs. In addition it could be observed in the simulations that the average number of hops for successful network search decreased with the number of searches executed. This is only true for searches also using local service pools.

8 Conclusion

This deliverable summarizes the activities in the last phase of OPAALS as regards the EvESimulator based on the version submitted within the code deliverable *D10.19 - Extension of Google maps visualization for large scale social networks.*

In the first chapter, the new architecture is outlined and the implementation is covered on a very high level. The dedicated focus is the *EveComponent* model which – at this stage – supports multiple application areas such as SNA and monitoring. A core benefit of this new architecture is the extensibility by using a well-defined component model conforming to state-of-the-art MDA tools and methodologies.

The second chapter of this deliverable focuses on the geospatial visualizations integrated into the EvESimulator user interface. Two virtual globe applications are integrated prototypically: Google Earth and NASA WorldWind. The ability to visually analyze results of simulations also in a geographical context is a relevant benefit. WorldWind and its Java-only API enable users to interact directly with the simulator user interface which is currently not possible with Google Earth.

Within the third chapter, a complete simulation use case, implemented with and for the new version of EvESim is described. It exemplifies the testing of a document editing tool with the utilization of distributed software agents. Furthermore, all capabilities of EvESim acting as a multi-agent simulation framework with reusable components are outlined.

Covering the social network analysis features, chapters four and five present on the one hand the acquisition of social network data and on the other hand the advantages social network analysts are retrieving from EvESim. Additional activities related to the social network analysis are given in chapter six, for example the visualization of the Irish BioTec industry or of the Indian social network Agropedia.

At the end of this paper, here a few concluding and outlooking remarks. Starting with the intention of a simulation framework for the Evolutionary Environment in the DBE project, EvESim became a major strength of activities at SUAS for many

years. Although the initial simulation ideas are not the main focus of EvESim any more, it acts now as an interdisciplinary framework for three main stakeholders. On the one hand, social network analysts are able to visualise and analyse the changing structure of international networks. Beside the visualisation capabilities, EvESim provides interfaces to well known visualisation and calculation libraries. On the other hand, computer scientists and software engineers, referred to as instance creators, are able to test and monitor their distributed software systems. This testing is possible, because the third group of stakeholders, the catalysts, provide the access to the social network data of the analysts as well as implement use cases as multi-agent systems. Instead of testing manually, the instance creators can use agents to call their systems and act on the basis of findings and data from social network analysis.

Envisaging the future of social networks as distributed systems which allow restricted privacy conditions for users, the EvESim can act as a perfect testing, monitoring and analysis tool for such systems. Beside that, the idea and benefit of Digital Ecosystems and self-organisation can be shown by dynamic visualisations on the virtual planet.

We also want to emphasise here, that the technological infrastructure, utilising the Eclipse Modeling Framework, offers a modular system, which, together with the component based structure of EvESim, is easy to extend and adapts to new challenges in the simulation, emulation and testing of distributed systems.

Although the simulation characteristic of EvESim shifted to social network analysis and infrastructure testing over the years, it attracted interest in the research community, which resulted for example in an invitation for a talk and future collaboration with the University of Edinburgh. In October we will present the EvESim to the School of Informatics and Business School, where they want to use EvESim in a Marie Curie IAPP. They have strong connections to the eHealth sector in Barcelona, Spain, and are very interested in the Digital Ecosystems and Digital Business Ecosystems research.

Bibliography

[1] W3C. Web Services Activity. `http://www.w3.org/2002/ws//`, Last accessed: 22.7.2010.

[2] Salzburg University of Applied Sciences. EvESimulator. `http://www.evesim.org`, Last Accessed: 15/04/2009.

[3] JUNG Framework Development Team. Java Universal Network/Graph Framework. `http://jung.sourceforge.net/`, Last accessed: 24.6.2010.

[4] C. Rücker. *Use Case and Feature Extension for a Multi-Agent Simulation Framework (Diploma Thesis)*. Fachhochschule Salzburg GmbH, Salzburg, Austria, June 2010.

[5] Oracle Corporation. Javadoc. `http://java.sun.com/j2se/javadoc/`, Last accessed: 22.7.2010.

[6] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[7] T.J. Heistracher R. Eder T. Kurz. D10.7 - Visualisation Service for P2P infrastructure and EvESim based on GoogleMaps. OPAALS Project, November 2008.

[8] Jukka Hutamäki, Christoph Rücker, Thomas Kurz, Thomas J. Heistracher, and Raimund Eder. D10.19 - Extension of google maps visualisation for large scale social networks. OPAALS Project, July 2010.

[9] The OPAALS Consortium. Flypeer - Dynamic P2P Infrastructure. `http://kenai.com/projects/flypeer`, Last accessed: 25.5.2010.

[10] OPAALS Consortium. Open Philosophies for Associative Autopoietic Digital Ecosystems (OPAALS). `http://www.opaals.org`, Last Accessed: 15/04/2009.

[11] A Universally Unique IDentifier URN Namespace (RFC 4122). http://tools.ietf.org/html/rfc4122, Last accessed: 25.4.2010.

[12] Google. Google Earth. http://earth.google.com, Last accessed: 21.05.2009.

[13] Google. Google Earth-Plug-in. http://earth.google.com/plugin, Last accessed: 21.05.2009.

[14] Google. Keyhole Markup Language Reference. http://code.google.com/apis/kml/documentation/kmlreference.html, Last accessed: 21.05.2009.

[15] Open Geospatial Consortium. Keyhole Markup Language Specification. http://www.opengeospatial.org/standards/kml/, Last accessed: 21.05.2009.

[16] The Apache Software Foundation. Apache Velocity. http://velocity.apache.org, Last accessed: 21.05.2009.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[18] NASA. World Wind Java SDK. http://worldwind.arc.nasa.gov, Last accessed: 10.6.2010.

[19] NASA. NASA OPEN SOURCE AGREEMENT VERSION 1.3. http://worldwind.arc.nasa.gov/worldwind-nosa-1.3.html, Last accessed: 10.6.2010.

[20] WorldWind Central. http://worldwindcentral.com/, Last accessed: 10.6.2010.

[21] JogAmp Community. Java™Binding for the OpenGL®API. http://jogamp.org/jogl/www/, Last accessed: 10.6.2010.

[22] Vladimir Silva. Embed the NASA World Wind Java SDK in Eclipse. http://www.ibm.com/developerworks/java/library/j-wwj/, Last accessed: 10.6.2010.

[23] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, Inc. New York, NY, USA, 2nd edition, 2009.

[24] Kurz, T. and Colugnati, F.A.B and English, A. and Lopes, L.C.R. D10.8 - Report on cross-domain social networks. OPAALS Project, 2009.

[25] D.M. Boyd and N.B. Ellison. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, Volume 13, Issue 1, 2007.

[26] C. Adelberger. *Evolutionary Framework Simulation (Diploma Thesis).* Fachhochschule Salzburg GmbH, Salzburg, Austria, 2008.

[27] Siqueira, P. and Barretto, S. Guigoh - online social network. `http://www.opaals.org.br/`. Last accessed on 15/05/2010.

[28] Facebook Incoporated. Facebook - social network. `http://www.facebook.com/press/info.php?factsheet`. Last accessed on 15/05/2010.

[29] J. Madadhain, D. Fisher, P. Smyth, S. White, and Y.B. Boey. Analysis and visualization of network data using jung. *Journal of Statistical Software*, 10:1–35, 2005.

[30] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167 – 256, 2003.

[31] T.A.B. Snijders. *The Statistical Evaluation of Social Network Dynamics.* Sociological Methodology. Boston and London: Basil Blackwell, 2001.

[32] P.J. Carrington, J. Scott, and S. Wasserman. *Models and Methods in Social Network Analysis.* Cambridge University Press, Cambridge, 2005.

[33] M. Huisman and M. A. J. Van Duijn. Software for social network analysis. In S. Wasserman P J. Carrington, J. Scott, editor, *Models and Methods in Social Network Analysis*, pages 270 – 316. New York: Cambridge University Press, 2005.

[34] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[35] V. Batageli and A. Mrvar. Pajek - program for analysis and visualization of large networks. Technical report, University of Ljubljana, Slovenia, 2010.

[36] Gabor Csardi and Tamas Nepusz. *igraph library.* MTA RMKI, Konkoly-Thege Miklos st. 29-33., Budapest 1121, Hungary.

[37] Domingo Vargas. *netmodels Package v0.2.* http://cran.r-project.org/web/packages., 2009.

[38] T.A.B. Snijders, C.E.G. Steglich, M. Schweinberger, and M. Huisman. *Manual for SIENA version 3.2.* Groningen: ICS, University of Groningen Oxford: Department of Statistics, University of Oxford.

[39] P. Boer, M. Huisman, T.A.B. Snijders, and E.P.H. Zeggelink. *StOCNET: an open software system for the advanced statistical analysis of social networks. Version 1.4. Manual.* Groningen: ProGAMMA / ICS.

[40] Schahram Dustdar, Harald Gall, and Manfred Hauswirth. *Software-Architekturen für verteilte Systeme.* Springer, 2003.

[41] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.

[42] Clip2 and The Gnutella Developer Forum. The gnutella protocol specification v0.4.

[43] I. Clark. *A Distributed Decentralised Information Storage and Retrival System.* University of Edinburgh, 1999. `http://freenetproject.org/papers/ddisrs.pdf`, Last Accessed: 15/04/2009.

[44] G. Briscoe. Digital ecosystems: Evolving service-oriented architectures. In *IEEE First International Conference on Bio Inspired mOdels of NETwork, Information and Computing Systems*, 2006.