UBIWARE Deliverable D3.3:

UBIWARE Platform Prototype v.3.0

August, 2010

Date	Aug 31, 2010
Document type	Report
Dissemination Level	UBIWARE project consortium
Contact Author	Vagan Terziyan
Co-Authors	Oleksiy Khriyenko, Sergiy Nikitin, Michal Nagy, Joonas Kesäniemi, Michael Cochez, Atte Pulkkis
Work component	WP1, WP2, WP3, WP4, WP5
Deliverable Code	D3.3
Deliverable Owner	IOG, JYU
Deliverable Status	Mandatory, Internal
Intellectual Property Rights	Unaffected

Table of Contents

Introduction 3
1 Platform Development 4
1.1 New Agent Platform architecture 4
1.1.1 Application agents in UBIWARE 5
1.1.2 Platform infrastructure agents 5
1.1.3 One-click platform startup 9
1.2 UBIWARE towards modern web 10
1.2.1 A new UBIWARE 3.0 Web Architecture 11
1.2.2 Web application architecture details 12
1.2.3 UBIWARE Desktop 14
1.2.4 Developing Web Applications with UBIWARE 15
1.2.5 Administrator's interface 17
1.3 Policies in UBIWARE 21
1.4 Core Platform Improvements 22
1.4.1 RDF2BEAN 22
1.4.2 Semantic Action Script 23
1.4.3 Development "under the hood" 25
2 A Use Case: Mashupper – Agent-enabled Social Web 30
2.1 Social Ontology 31
2.2 Agent Architecture 32
2.3 User Interface 33
2.3.1 OAuth authentication 34
2.3.2 Building Personal User Network 35
2.3.3 Using Personal User Network 37
2.3.4 Geographical status updates panel 38
2.3.5 Activity timeline 39
2.4 Conclusions 40
3 Smart Interfaces: Context-aware GUI for Integrated Data (4i technology) 41
3.1 Background 41
3.2 ResourcesCloseness_RDFConvertor - general RDF adapter for 4I Browser 41
3.2.1 Adapter functionality and architecture 41
3.2.2 GUI of the convertor 42
3.2.2.1 "Simple text" and "Simple numeric" fields 43
3.2.2.2 "Keywords field" 44
3.2.2.3 "Complex text field" 44
3.2.2.4 "Interval field" 45
3.3 Conclusions and future work 45

Introduction

The UBIWARE project aims at a new generation middleware platform which will allow creation of self-managed complex industrial systems consisting of distributed, heterogeneous, shared and reusable components of different nature, e.g. smart machines and devices, sensors, actuators, RFIDs, web-services, software components and applications, humans, etc. The technologies, on which the project relies, are the Software Agents for management of complex systems, and the Semantic Web, for interoperability, including dynamic discovery, data integration, and inter-agent behavioral coordination.

Work in this project is divided into seven work packages which are running in parallel:

1. Core agent-based platform design
2. Managing Distributed Resource Histories
3. Security in UBIWARE
4. Self-Management and Configurability
5. Context-aware Smart Interfaces for Integrated Data
6. Middleware for Peer-to-Peer Discovery
7. Industrial cases and prototypes.

Work-packages 1 through 6 are research work packages; however, the research efforts are combined with agile software development processes. Prototypes of the UBIWARE platform, integrating the work in these 6 work packages at different levels of their readiness, are developed during each project year, as UBIWARE 1.0, UBIWARE 2.0 and UBIWARE 3.0. Due to the prolongation of the project, we will have one more version of the platform release – UBIWARE 3.1, which will be reported in the final steering group meeting.

UBIWARE deliverable D3.1 reported on the research results from work packages 1, 2, 4 and 5. At that time it was decided to postpone the research on work packages 3 and 6. Nevertheless, we are presenting in this report the current status of the research and development findings for all work packages – from 1 to 6. As a demonstration case of the platform improvement and evolution we have chosen a Social Network integration scenario, which can be considered as an industrial case and thus falls into the work package 7. This deliverable, D3.3, presents the integrated development results from all the work packages, i.e. the current state of the UBIWARE 3.0 platform prototype. Naturally, during the development stage, solutions described previously in D3.1 have undergone some changes and improvements.

This deliverable consists of the software itself and an accompanying report.

UBIWARE Deliverable D3.3:
Workpackages involved WP1, WP2, WP3, WP4, WP6:

1 Platform Development

1.1 New Agent Platform architecture

In the year 3 of the platform evolution we have put our main effort to the platform usability issues. To make the platform attractive as a middleware solution, we have to offer a set of platform features that are comparable with other software development middleware available on the market today. Furthermore, to be able to demonstrate the benefits of the platform, we have to show a clear add value the platform may offer.
This deliverable has brought the UBIWARE platform to the qualitatively new level of the middleware solution – the platform now combines the features of the application server, the semantic web platform and the agent-driven platform, where agent-driven semantic applications can serve end customers with the high quality web-based GUIs, enhanced user-friendliness and responsiveness. The platform has become an application-independent runtime environment, where special infrastructure agents take care of the platform itself, not of the applications being run on it. At the same time, we introduce personal user agents, thus making the platform user-oriented (see Figure 1.1).

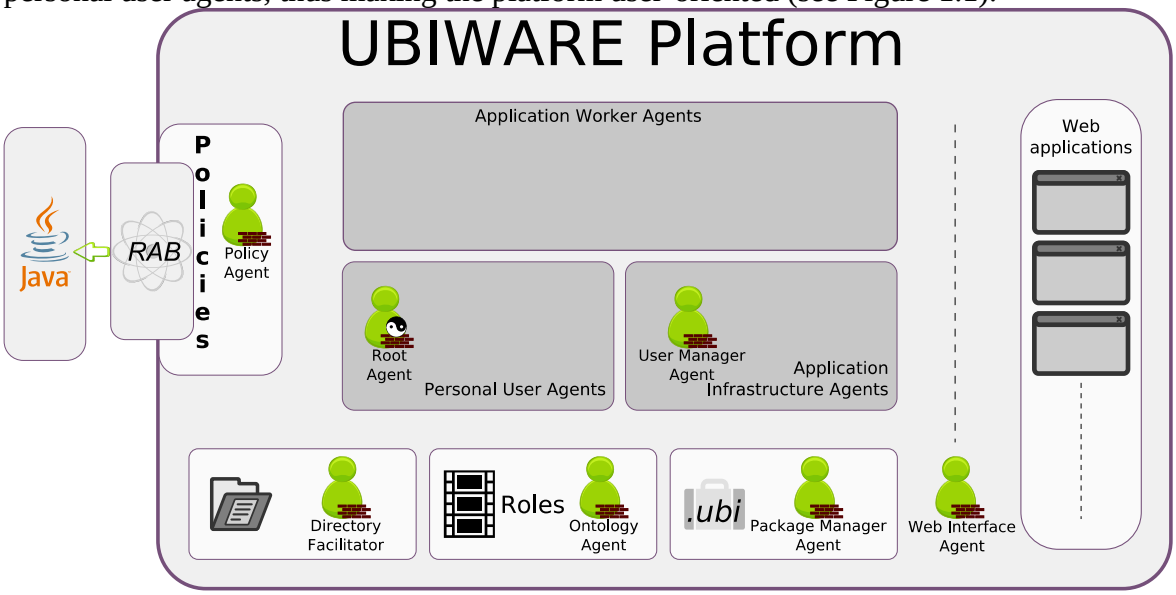

Figure 1.1 – UBIWARE 3.0 platform architecture

We design the UBIWARE platform infrastructure for creation of various kinds of applications. Those applications have a freedom to use a web front-end, on-the-platform user management and other infrastructure or define their own platform components depending on the needs of the application.
In the UBIWARE 3.0 architecture we identify two groups of agents. The first group includes the agents which are application-specific, whereas the second group gathers infrastructure agents providing services to those application-specific ones. In the next section we shortly introduce the new structure of the UBIWARE application and then discuss the role of the infrastructure.

1.1.1 Application agents in UBIWARE

We start with describing how an agent-enabled application should be designed to fit the UBIWARE 3.0 platform infrastructure. We introduce three different agent sub-types for application agents:

- *Personal user agent (PUA):* In the view of the unified approach used in UBIWARE, every external resource is represented with an agent. Humans, which we call users of our platform, are represented by personal user agents (PUA). Each platform user may have only one personal user agent in the platform.

- *Application worker agent (AWA):* This agent is the bridge between the personal user agent and the application infrastructure agents (not to be mixed up with the platform infrastructure agents). It is used as a representative of the PUA towards the application. The main reason for having this mediator is robustness –if the application code resides in the main user agent (PUA) and it is not written well then it might corrupt the operation of the PUA. Therefore, we put code of each application into the separate agent called worker. Having AWA has another important advantage - it gives application developers safety of storing data concerning a specific user into the worker agent, being sure that no other application will be able to retrieve or modify this data.

- *Application infrastructure agents (AIA):* These agents are building blocks of applications which do not have to be replicated for each individual user. AIA's represent applications on the platform and may provide aka "application services" to other agents. They can provide certain functionality to web applications, for example, in the desktop login when the actual user is not even known yet. This type of agents is also applicable when the application is not user-specific or does not have a web interface. As an example, we can consider the Fingrid industrial use case where agents are doing background work continuously without user interaction.

1.1.2 Platform infrastructure agents

In this subsection we shortly describe the infrastructure agents and their roles in the platform. These agents are application-independent and are designed to handle the platform operation.

- **Policy Agent(PA)**

  - *Description:* The policy agent is the agent responsible for policy checks. Its use will be further described in the chapter about policies.

  - *Implementation:* The policy agent has two responsibilities. At first, it processes information messages about agents being allowed to perform certain actions and about agents being in a certain state (Which would then allow them to perform a certain action). The second task performed, is checking of allowance queries. Whenever the Policy Agent receives a message containing a question in the form `{<agent> pol:isAllowedToDo <action>} sapl:configuredAs <parameterList>` it will start the investigation procedure to check whether this requestor agent is allowed to perform this action. The check procedure matches the request against the responsibility design patterns. The check may involve a sequential chain of smaller checks. If a link in the chain is not able to tell whether this action is allowed, it gives the responsibility of deciding to the next link in the chain. If a link is able to decide whether the action is allowed or disallowed, the chain is stopped and the result is sent. The last checker in the chain is final and sends as a result that the action is denied. The following checkers (links) are currently implemented:

    - Infrastructure agent check (Infrastructure agents have all rights

    - Type check (Agents of certain types have certain rights, for example application worker agents are allowed to register themselves to the directory facilitator.)

    - Application context checks (Agents working in the context of an application are allowed to perform application-specific actions. For example a worker agent of the Facebook adapter is allowed to perform actions to interact with the Facebook platform)

    - Advanced message sender checker

      - Any agent is allowed to send messages to itself.

      - Communication between master and slave agents is allowed.

      - A personal user agent is allowed to communicate with its worker agents.

      - Worker agents are allowed to interact with application infrastructure agents.

      - There can be specific exceptions, for example, configuration application agents can be allowed to send messages to infrastructure agents.

- *Interaction:* The policy agent is indirectly interacting with any agent on the platform which wants to perform an external action. For more details on policies see section 1.3 of this report.

- **UBIWARE DF Agent (UDF)**

  - (Under development) UBIWARE Directory Facilitator Agent will keep the ontology of all the concepts used on the platform.

- **Web Interface Agent (WIA)**

  - *Description:* Web Interface Agent (WIA) is responsible for all communication over http with the outside world. The communication is arranged by using embedded web server that handles the incoming requests and provides web application with the access to the agent platform. It works as a gateway for all the messaging between web applications running on the server instance and the application-specific or infrastructural agents. WIA is responsible for the lifecycle management of the web server and provides services for deploying and removing web applications from the server. WIA is also responsible for managing the tickets related to platform-wide, single-sign-on mechanism, used in the UbiwareDesktop (see Section 1.2.3).

  - Implementation: Current WIA implementation uses Jetty as its embedded web server. Jetty is small and versatile server hosted by Eclipse Foundation that works as a basic HTTP server as well as a Servlet container. WIA starts up the web server with the certain port configured, as part of its own startup routine. When the server is up and running, WIA deploys the configured web applications as WAR files to the server. DeployWarBehavior allows WIA to hot-deploy WAR files to the server without the need to restart the servlet container.

  - Interaction: Web applications use WIA as a mediator when sending messages to the agents. However, WIA does not provide direct access to the messaging facilities. Web applications use WIA through a wrapper that allows them to interact with one specific agent. Wrappers are available for application infrastructure agents of the application in question and for the user specific worker agent. When wrappers are created, WIA queries the directory facilitator agent for the worker agent associated with certain logged in user and application. WIA accepts new web applications for deployment from the application manager infrastructural agent. Same agent can also request a web application to be removed. One special case of interaction for WIA is the authentication for the UBIWARE Desktop. When the user provides his or hers credentials, WIA interacts directly with the user manager agent and tries to authenticate the user.

- **Package Manager Agent (PMA)**

  - *Description:* This agent is responsible for deployment of UBI packages (UBIWARE application packages). The agent accepts a package and

unpacks it to a temporary folder. Then it goes through all package components and handles them in one-by-one fashion. It registers all the roles to the ontology agent, sends policy rules to the policy agent, starts application infrastructure agents and asks web interface agent to deploy the Web application WAR file. It also properly registers the application at application manager agent.

- *Implementation:* The deployment process consists of two steps. In the first step the agent uses a Reusable Atomic Behavior (RAB) that reads a UBI package, verifies it, unpacks it to a temporary folder and creates a descriptor object in the agent beliefs. In the second phase, the agent reads the descriptor object and the above mentioned procedures are performed. Infrastructure agents are notified and particular components of the UBI package are redistributed among them.

- *Interaction:* During the package deployment process, package manager agent communicates with platform infrastructure agents. Large files like WAR archives are interchanged between agents by saving those to a temporary folder and referring to them using a file path. The rest of the messages are transferred in SAPL language.

- **Ontology Agent (OA)**

  - Used by other agents for loading agent roles

- **User Manager Agent (UMA)**

  - *Description:* The user manager is responsible for managing human users on the platform, i.e. storing information about user names, passwords and other personal information. UMA is also responsible for personal user agents and it stores the lists of applications selected by the user, as well as information about applications to be selected by default. UMA is also starting an AWA (see subsection 1.1.1) if the user selects an application to use.

  - *Implementation:* UMA agent knows about application worker agents belonging to applications, it gets this information from the package manager agent when applications are deployed. When a user decides to select an application for use (or the application is configured to be automatically selected), UMA starts a suitable worker agent for the user of the application. Internally, this agent keeps a container of application configurations. Since this agent is also application infrastructure agent for the user manager application, it is possible to interact with it using that application. Whenever a user is added, this agent will keep beliefs about the user's existence and link user name to his/her personal user agent. UMA also serves as infrastructure behind the login screen.

  - *Interaction:* The interaction with this agent is, as mentioned before, mainly from within web applications, i.e. the authentication, user registration and application selection.

- **Root Agent (root)**

  - *Description:* The root agent is exactly the same as any other personal user agent. The only exception is the amount of rights it has and the moment when it is created. Root Agent has all possible rights available on the platform. Giving such privileges has been inspired by the root user in Linux systems. One of the reasons to give such privileges was to provide a way within the production environment to interact with any component in the system. Example use cases are: malfunctioning or obsolete components which should be discarded or a platform shutdown. The root agent is created first and then also creates other users of the system.

  - *Implementation:* The implementation of the root agent is exactly the same as the implementation of normal personal user agents.

  - *Interaction:* The root agent interacts, just like normal PUA's only with its application worker agents.

1.1.3 One-click platform startup

In the previous version of UBIWARE, the platform was started up using a list of batch files. The batch files contained commands to run particular parts of the platform. The included instructions about which agent in which container should be started. This solution had several disadvantages. First disadvantage was the dependency on the operating system. Each operating system uses different scripting language and therefore batch files had to be adjusted to a particular system – bat files for Windows, shell scripts for Linux, etc. There were dependencies among the batch files and the user had to be aware of them and start them in a particular order. The second disadvantage was the portability. If a user wanted to run the same scenario on another operating system, he/she had to rewrite the scripts to another scripting language and change library paths.
The current version of UBIWARE solved both problems. The platform uses several universal scripts – one for each operating system. No matter what kind of scenario is run, the same script is used every time. Instead of having the platform configuration in the script, now the configuration is stored in an RDF configuration file. The configuration file is called startup.rdf and it contains a list of several RDF resources such as agent, container, script, role, etc. These resources are interconnected using object properties and they form a pseudo-tree with platform resource as its root. The ontology depicted in Figure 1.2 describes the pseudo-tree structure. We introduce an agent platform object which can have several container objects connected to it. A container can be either a main container or peripheral container. In containers agents are running. Each agent has a name and a list of scripts and/or roles associated with it. Every role has a name and every script has its location. It is a pseudo-tree, because two different agents may refer to the same script or role.

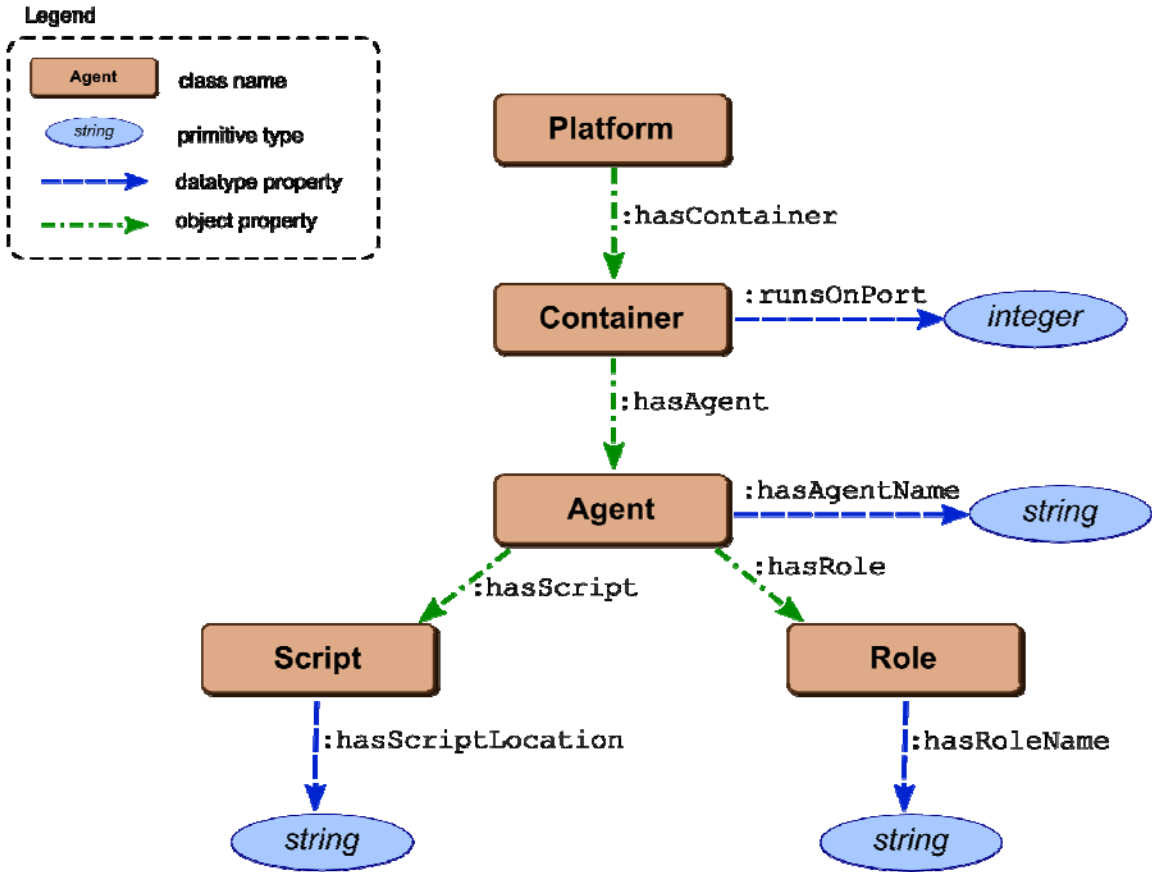

Figure 1.2 - Platform startup ontology

The existence of a startup script solves the problem of cross-platform portability. Instead of moving the whole list of scripts and rewriting them to a different scripting language, the user has to copy just the startup configuration file and execute a different script file, which is already available.

Inner working
In the current implementation the startup mechanism reads the configuration from the startup file. In the future the startup mechanism can be a service callable by an agent. This way an agent can provide a startup semantic annotation and then call a service. As a result a new platform is started.

1.2 UBIWARE towards modern web

In the previous version of the UBIWARE platform the only way for agents to offer the communication with external systems was the AgentServer reusable atomic behavior and ServerEvent class. AgentServer and ServerEvent provided a way of sending and receiving messages through TCP sockets. This was convenient for machine to machine interaction and for simple user interfaces, but was lacking features, such as thread pooling, caching and support for existing tools, - those needed for building more sophisticated web based applications. The new web application architecture based on embedded Jetty HTTP server is designed to take UBIWARE to the modern web.

1.2.1 A new UBIWARE 3.0 Web Architecture

The main idea behind the new web architecture is to allow web developers to use their old and reliable and/or new and shiny web development tools and frameworks for the UI development while providing easy and unobtrusive integration to the agent platform. The high level general architecture for UBIWARE web application is depicted in Figure 1.3.

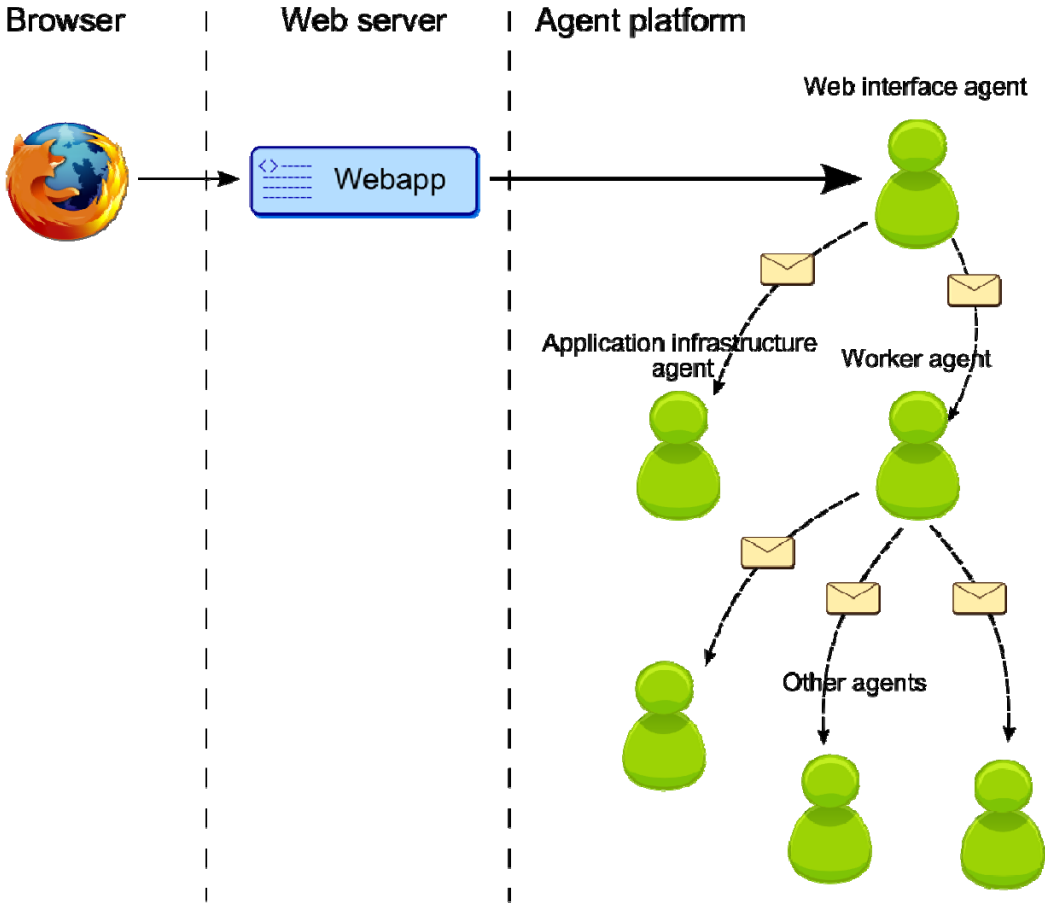

Figure 1.3- UBIWARE Web Application Architecture

Web applications running on one Jetty web server instance use specialized web interface agent (WIA) to communicate with other agents. WIA currently only works as “dummy” mediator, but it could also be used to monitor or restrict traffic from web applications to the agent side. Possible response from the agent is routed back to the right web application, which can use the returned data to modify and render its GUI.

WIA stands between the web application and agent development worlds. Since it is an agent, WIAs can be cloned and moved from one machine to another for example for load balancing purposes. Current version of the UBIWARE platform only has one type of web interface agent available that uses Jetty servlet container as a web server. The platform could however also provide different kinds of WIAs for different and more specific purposes. In addition to servlet based web applications coded with Java, another WIA could be able to host dynamic web applications build on Ruby, Python or PHP. Or, if the application only needs to serve simple static content, it could use WIA with a simple

HTTP server with no support for dynamic web applications. One could also say that the agent using old AgentServerBehavior RAB is a kind of web interface agent.

The possibility to add, move, remove and configure agent based WIAs in a very dynamic manner provides ground work for self-configurable web based agent application platform.

Although modern web applications are usually based on dynamically created content, nothing prevents application from using Jetty based WIA to distribute static content like images, video, html pages. In this kind of scenario the worker agent could still be used to update or generate that content. For example, instead of generating complex statistics report on demand, the worker agent could periodically generate updated version of the report to certain place, where the web server can serve it as a static asset, effectively moving the responsibility of handling mundane data transfer task from application agent to the web server.

1.2.2 Web application architecture details

This section gives a more detailed description of the inner workings of the new web application integration for UBIWARE platform. The solution is based on filters that have been part of the Java Servlet specification since the version 2.3. A filter can be configured to intercept requests and response and to transform or use information contained in them. Common use for filters is authentication, logging and data compression. In this case a special filter is used to inject request with an object that wraps the web interface agent, allowing web application to use its messaging capabilities in a controlled manner. The main components of the web integration are AgentWrapper, PlatformConnection, Servlet filters and RABs for deploying applications.

AgentWrapper

Since it would not have been safe to let any web application to send messages to any agent, web applications are forced to communicate with agents using the instances of AgentWrapper class. It takes Jade agent and the name of the agent that is going to receive the messages as the constructor parameters. AgentWrapper provides three **synchronous** methods for communicating with the wrapped agent recipient:

inform() method is used to send message to the agent without expecting any kind of response. This method can be used to, for example, inform the agent about non-critical (i.e. no confirmation required) state changes of the web application.

requestResponse() method sends a message to the agent and waits for the response. If no response is received within the timeout period, then the exception is thrown.

syncAction() this is a convenience method for sending ACL messages that conform to the action protocol

PlatformConnection

PlatformConnection interface is the source of AgentWrappers. Platform connection is injected as part of HTTP request where web application can retrieve it. Interface has methods for getting the AgentWrappers for worker agent responsible for the web application and for the application infrastructure agents.

Filters

UBIWARE provides three filters for different kind of application scenarios.

PlatformConnectionFilter

This is the main filter that must be used by all the web applications that are deployed under the UBIWARE desktop (see subsection 1.2.3). The filter first checks if there is a valid ticket in the request. If the ticket is valid, it tries to locate the worker agent responsible for handling the web application in question for the given ticket. Finally the filter encapsulates reference to the WIA, the name of the worker agent and a list of possible application specific infrastructure agents into an instance of PlatformConnection which is then stored in the HTTP request.

UbiwareAgentBridgeFilter

This filter is intended for testing and building web applications with UBIWARE platform without the UbiwareDesktop. Filter can be configured to provide AgentWrapper for any named agent. The name of the wrapped agent is given as initialization parameter to the filter.

```
<filter>
	<display-name>UbiwareFilter</display-name>
	<filter-name>UbiwareFilter</filter-name>
	<filter-class>ubiware.web.UbiwareAgentBridgeFilter</filter-class>
	<init-param>
		<param-name>agent</param-name>
		<param-value>worker</param-value>
	</init-param>
</filter>
```

AuthenticationFilter

Authentication filter can be used with the web applications targeted for UbiwareDesktop to restrict access to the certain URLs only to the logged in UbiwareDesktop users.

```
<filter>
	<display-name>auth</display-name>
	<filter-name>auth</filter-name>
	<filter-class>ubiware.web.AuthenticationFilter</filter-class>
	<init-param>
		<param-name>exceptions</param-name>
		<param-value>/publicServlet</param-value>
	</init-param>
</filter>
```

Filter takes initialization parameter “exceptions”, that is a comma separated list of URLs that should be excluded from the filter. Exceptions can be used for example to provide public callback URLs to otherwise restricted applications.

Behaviors

DeployWarBehavior

This is the Behavior used by the WebInterfaceAgent to deploy WAR files to the Jetty servlet container. WARs can be hot-deployed while the server is running. Behavior adds PlatformConnectionFilter with the mapping ‘/*’ to every deployed application so it should only be used in conjunction with the UbiwareDesktop.

DebugWarBehavior

DebugWarBehavior is identical to the DeployWarBehavior with the exception that it does not add any filters to the deployed applications.

1.2.3 UBIWARE Desktop

The idea behind UbiwareDesktop was borrowed from the web desktop environments. Web desktop is a desktop environment embedded in the browser. Web desktops like eyeOS (http://eyeos.org/) offer many of the functionalities and applications available on basic Windows, OSX or Linux desktop environments, such as productivity suites and file management. Some of the benefits of moving desktop to the web are high availability, server-side session management and centralized software management.

UbiwareDesktop is a web application that is distributed with the UBIWARE platform. In its current version the UI acts as simple launcher for other web applications deployed as part of the desktop environment. Applications can be opened as windows inside the desktop or in a new browser window. Figure 1.4 shows the desktop with two application windows open inside single browser window.

Currently the main benefit of using UbiwareDesktop as the deployment target for applications is the user management and authentication services provided by the desktop. As in any multi-user desktop environment, UbiwareDesktop requires users to login. After the user has successfully logged in, the UbiwareDesktop creates a ticket for the session, which is used to authorize the subsequent requests. Ticket can be used as kind of single-sign-on system, since other web applications can use the same ticket as a way to authenticate users. Users can be managed using another web application that is automatically available for all the administrative users.

In the future releases of the UBIWARE platform, desktop is envisioned to facilitate semantic drag-and-drop between applications. In order to make that possible without browser plugins, UbiwareDesktop could work as an intelligent, agent-driven mediator between source and target applications.

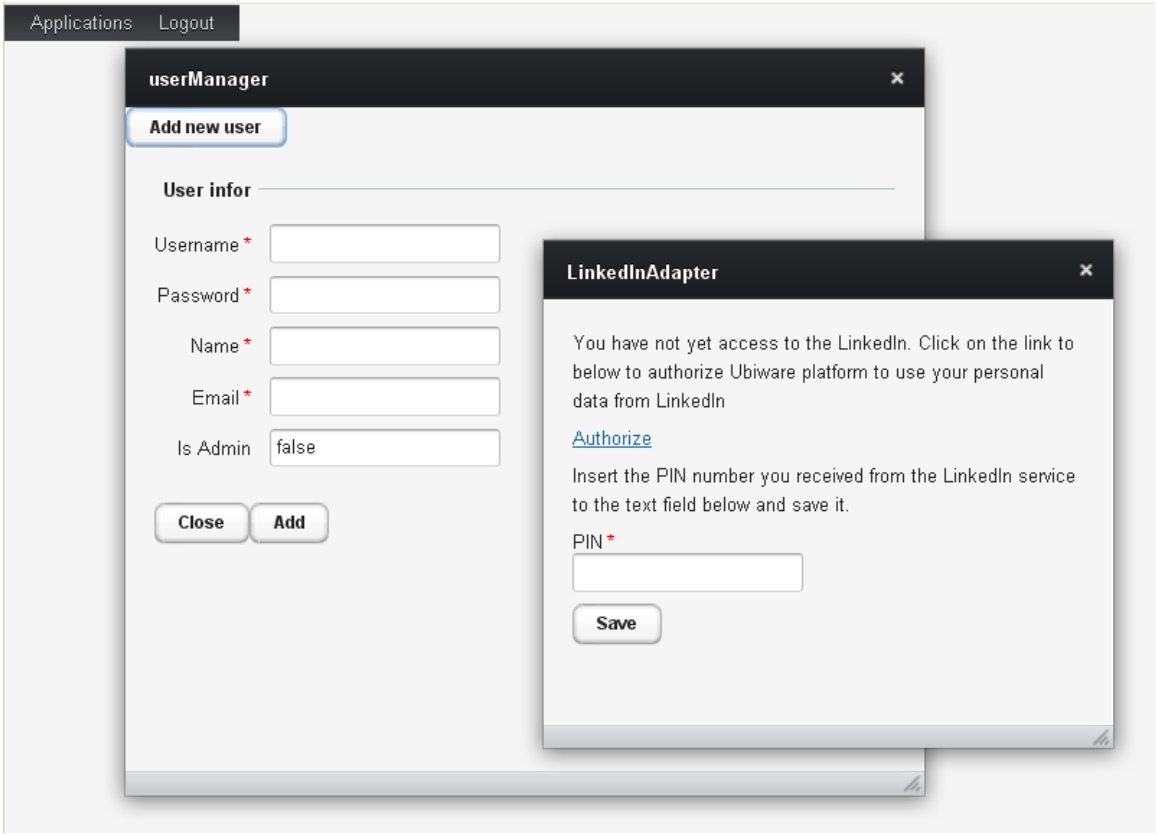

Figure 1.4 – Ubiware Desktop with running applications

1.2.4 Developing Web Applications with UBIWARE

This section describes everything needed for creating web applications that take advantage of the UBIWARE agent platform. When starting to develop a web application with UBIWARE platform the first thing one has to decide is whether the application will be running under the UbiwareDesktop or not. The latter approach will be from here on referred as the standalone approach. Applications targeted for UbiwareDesktop use the PlatformConnectionFilter, which is added automatically by DeployWarBehavior, and AuthenticationFilter whereas the standalone applications rely on UbiwareAgentBridgeFilter for bridging the gap between java code and agents, and DebugWarBehavior for deploying the application to the WIA.

The current Jetty based implementation of WIA requires all web applications to be packaged as web application archives (WAR). This basically means that the server side code of the dynamic web application must be developed as least partly using Java technology. WAR file has a very specific hierarchical structure that is described as part of the Java Servlet specification (http://jcp.org/en/jsr/detail?id=53).

Accessing the platform

Inside the UbiwareDesktop

Access to the UBIWARE platform is provided through the PlatformConnection interface, which provides methods for retrieving agent wrapper for the worker agent or application infrastructure agents of the web application. PlatformConnection instance can be retrieved from the HTTP request using the following piece of code:

```
PlatformInterface pi = (PlatformInterface)
 httpRequest.getAttribute(PlatformInterface.PLATFORM_INTERFACE_KEY)
```

After that, PlatformInterface can be used to retrieve AgentWrapper, which provides methods for communicating with the wrapped recipient.

```
AgentWrapper aw = platformInterface.getWorkerAgent();
aw.inform(“:icecream sapl:is :good”, “SAPL”);
```

In a standalone application

Standalone web applications can access the AgentWrapper directly without going through the PlatformInterface.

```
AgentWrapper wa = (AgentWrapper)
      httpRequest.getAttribute(
            UbiwareAgentBridgeFilter.WORKER_AGENT_WRAPPER);
wa.inform(saplContent, "SAPL");
```

Authentication

Web application deployed to the UBIWARE is by default accessible to any who knows the right URL. In order to make the application or part of the application available only to the users who have logged into the UBIWARE platform, one can use the AuthenticationFilter as described above (see subsection 1.2.2). For example, the web application can be divided into public and restricted sections by adding the following lines to the web.xml:

```
<filter>
  <display-name>auth</display-name>
  <filter-name>auth</filter-name>
  <filter-class>ubiware.web.AuthenticationFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>auth</filter-name>
  <url-pattern>/restricted/*</url-pattern>
</filter-mapping>
```

With this kind on configuration, everything is publicly available with the exception of URLs starting with /restricted.

Alternative way for accessing PlatformConnection

Some of the web frameworks, like Vaadin for example, do their best to shield the developer from the low-level details of HTTP request and response cycle. This might mean that the developer does not have a convenient access to the HTTPServletRequest object, where the PlatformConnection instance is stored. RequestContextFilter from the

Spring framework project (http://www.springsource.org/) provides easy access to the request parameters by storing them to the ThreadLocal. Parameters can be retrieved using RequestContextHolder class from the same source. This approach is used in all the Vaadin based web applications created by the UBIWARE project using the following filter configuration:

```
<filter>
  <display-name>SpringFilter</display-name>
  <filter-name>SpringFilter</filter-name>
  <filter-class>
org.springframework.web.filter.RequestContextFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>SpringFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Now, the PlatformInterface and worker agent wrapper can be accessed in the following manner:

```
RequestAttributes ra = RequestContextHolder.currentRequestAttributes();
PlatformInterface pi = (PlatformInterface)
   ra.getAttribute(    PlatformInterface.PLATFORM_INTERFACE_KEY,
                       RequestAttributes.SCOPE_REQUEST);
AgentWrapper aw = pi.getWorkerAgent();
```

Example web application

In order to make the web development with UBIWARE as easy as possible, the distribution will include a skeleton web application. Sample application includes two agents: WIA and worker. The web application consists of simple html-form for sending messages to the worker agent and one servlet that acts as the server-side handler for the form.

1.2.5 Administrator's interface

The purpose of developing the administrator interface has arisen from the need not to only observe the platform state, but to be able to affect the platform agents on the low level. Having a possibility to affect the platform agents, makes it more resistant to the application malfunctioning or failures that might lead the platform to the uncontrolled state. Another reason for having such interface is to have a possibility for low-level configuration and platform maintenance without stopping it.

Architectural overview

The "Agent Ontology Manager" is developed as a web-based interface. The interface is constructed with HTML and JavaScript technologies, using Qooxdoo open-source JavaScript framework. The UI communicates with the UBIWARE platform via HTTP, and interacts with an agent that is being configured and/or managed.

The User Interface itself is divided into two tabs. The Ontology tab allows the user to browse and edit classes, instances and properties of the ontology that defines the agent beliefs structure. The interface allows not only browsing, but also the editing and immediate testing of the changes – i.e. it allows the user to invoke the executable agent components that are listed in the Instances branch. User can further define conditions of the invocation and execute it. The results of instance invocation are shown to the user.
Agent Ontology Manager (AOM) is based on the Metso-case Agent Component Manager (ACM) developed as industrial case for Metso Automation in year 2010. Thanks to efficient application code structure, we were able to re-organize majority of previous functionality into a more efficient and generic package.

User's Guide

To navigate through the UI, click the tab icons in the upper left corner (see Figure 1.5).

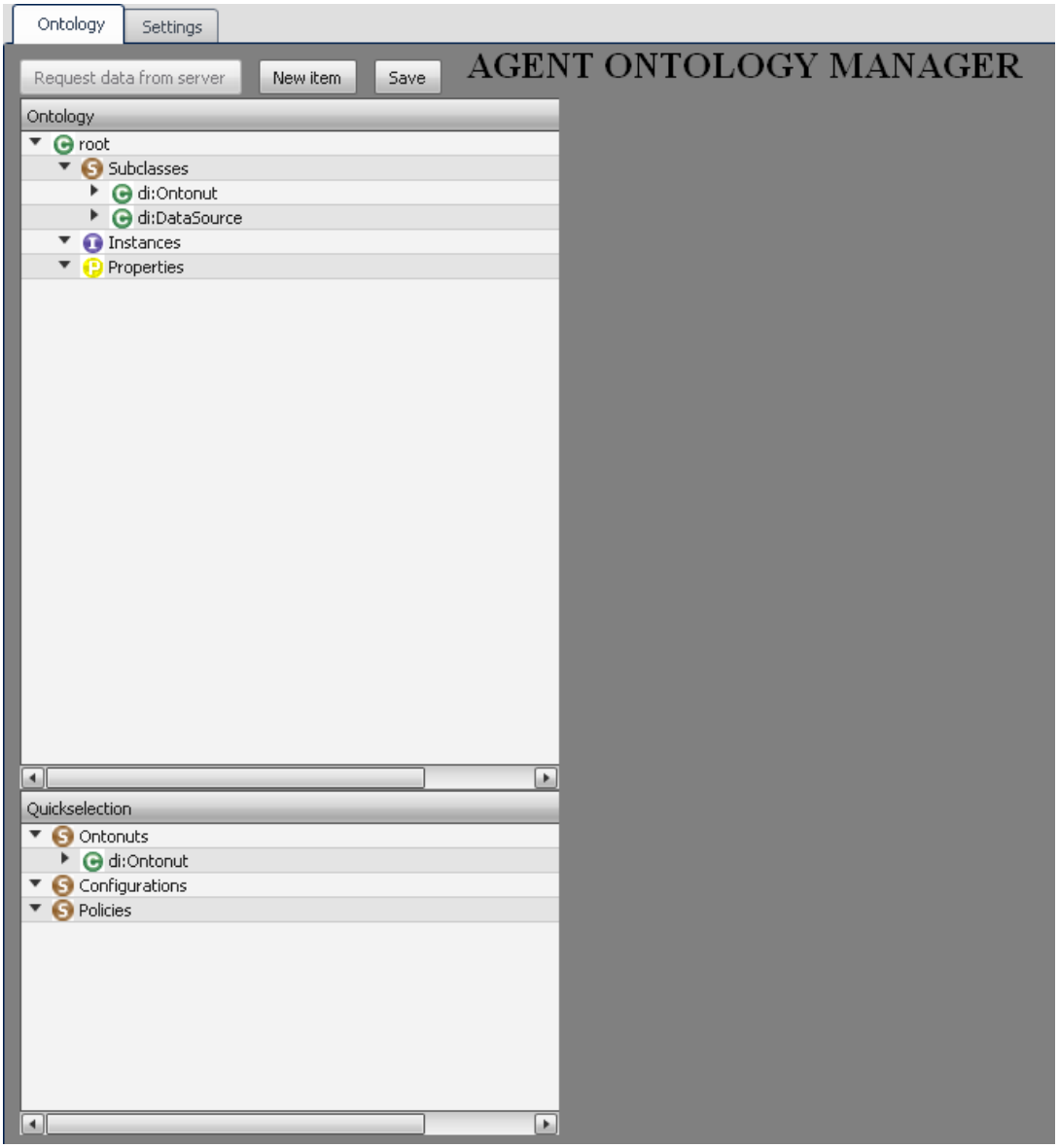

Figure 1.5 – Agent Ontology Manager interface in the initial state

- "Ontology" –tab is where majority of work is done. This tab contains the tools for adding, editing and removing items into and/or from the ontology.
- "Settings" –tab is under construction, and for now only allows changing the address of target agent.

- Ontology -

Navigating the tree is simple: click the black arrows on the left to open and close branches of the tree. To select an item, click anywhere on the row the item is located in. Doing this will open an editor window for the right element (see Figure 1.6).

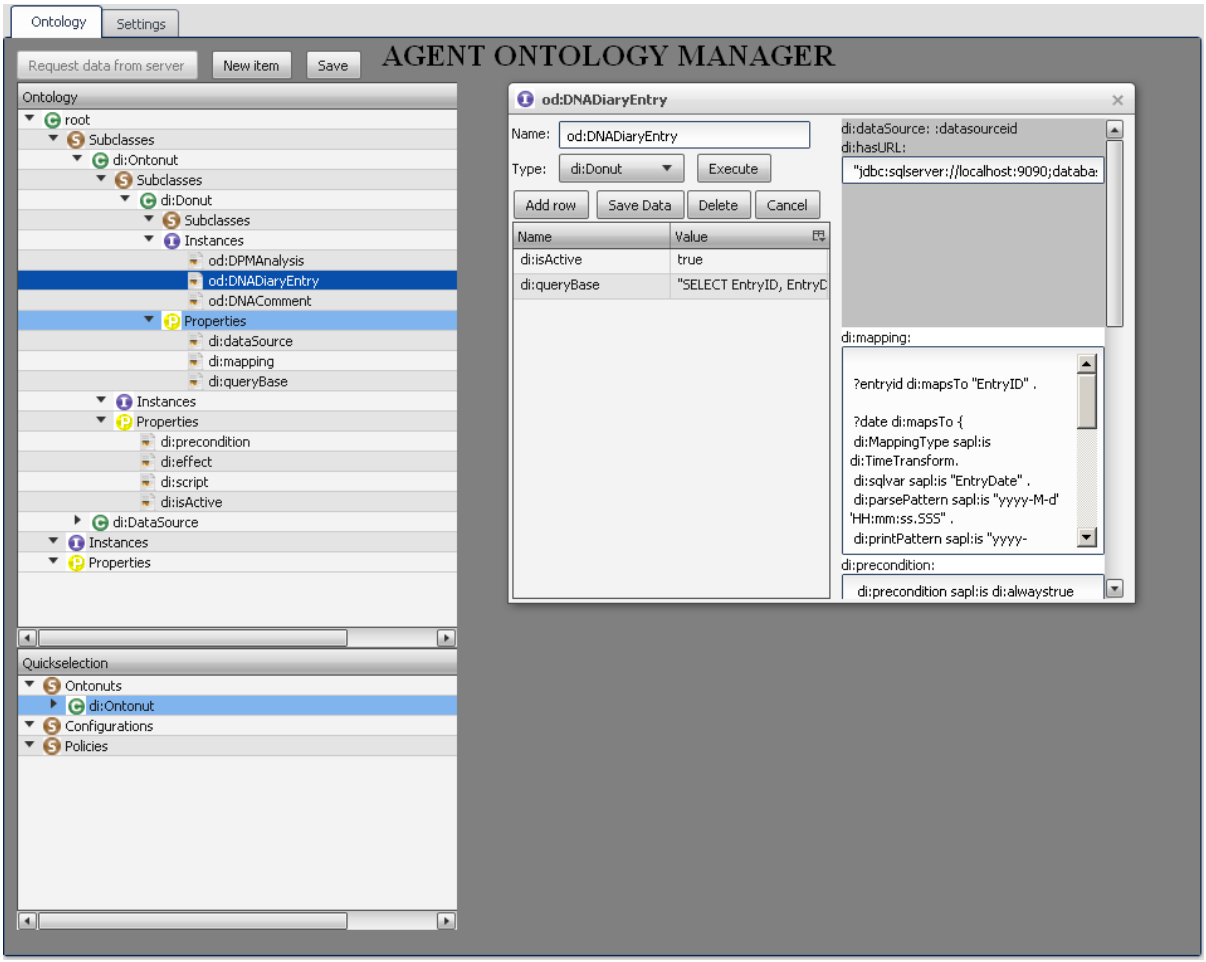

Figure 1.6 – Instance editing

Saving changes happens by clicking "Save"-button or "Save Data" -button. Settings will not be saved if the page is refreshed, user selects a new component, editor window is closed before saving or user clicks the "Cancel"-button. After you are done with editing, pushing the "save" button above the tree will save changes permanently.

Cancelling changes can be done by clicking "Cancel"-button, or closing the editor window by clicking the 'x' in the upper right corner. Also, all unsaved settings will be discarded in event of a page refresh.

New items can be created with "New Item" –button. Clicking this button opens a dialog that queries item type from the user. After choosing the type, a new editor window

appears for giving details about the type of element you are about to create. Item will be created instantly after pushing “Create”.

Deleting items is accomplished by pushing the “Delete”-button found in all editor windows. To open an editor window, click any item in the tree.

A component invocation can be done by clicking the “Execute” –button in instance editor window. A new window opens, that allows user to define specific conditions for the invocation. Users can add, edit and remove conditions. Editing conditions is started by clicking any row, after which condition window opens. After editing is done, instance is invoked with the “Execute”-button, and the results are shown in a new window.

AOM application architecture

This part will describe the internal working logic of the Agent Ontology Manager functions and components (see Figure 1.7).

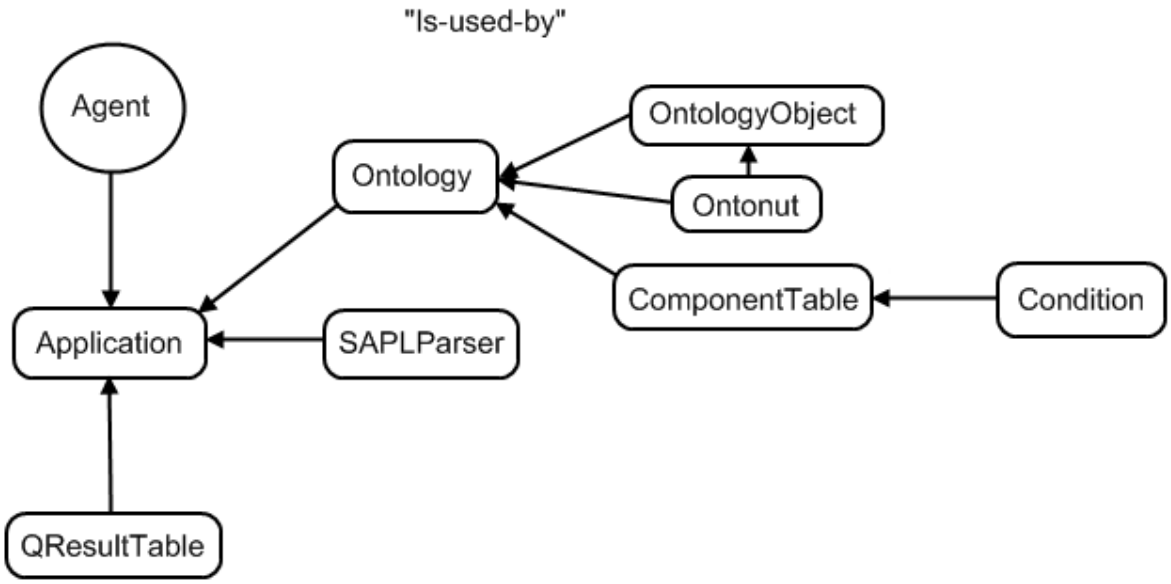

Figure 1.7 – Agent Ontology Manager interface in the initial state

- *Initialization*-
When the index.html –file is opened, first thing the JavaScript of the application does is constructs the User Interface elements. Second task of this initialization is retrieving two pieces of information from application’s Worker Agent: settings and ontology. Settings include user-customized settings for the application. For now the only available setting is the location of the worker agent the application uses. Ontology contains the platform ontology in Notation-3 –format plain text RDF.

After the information is retrieved, *Application* parses the ontology into manageable pieces and hands it to *Ontology*. *Ontology* creates the required objects (*OntologyObject*/*Ontonut*) based on the parsed information, and organizes the information into a coherent tree-like structure that can be easily handled and visualized.

-Managing data-
While *Application* handles creating the UI elements and the communication with the agent, *Ontology* takes care of the data management of AOM. Windows related to editing, creating and deleting data are created in *Ontology*. This way *Ontology* has the control over data management.

Searching the data tree structure is simple, since it is composed of *OntologyObjects*, that contain all the data. This is possible due to all information in the used ontology being based on classes. This way all the searches for data can be done fast recursively.

Adding, modifying and deleting data happens in a very similar way. When user triggers certain action, first the location of data in the data tree is found by searching for it. After the location is found, either the new data is added, or the old data is replaced with new one, or the data entry is deleted.

-Invoking instances-
The administration interface of AOM is designed to invoke three types of instances: Ontonuts, Policies and Configurations. Currenlty only the Ontonuts execution is fully supported, whereas other two types are partially handled by AOM.

When the instance is invoked, the application sends the invocation information to the agent and agent then creates the S-APL definitions needed to invoke the component in question. Component executes automatically after the required definitions are given, and agent then returns the results to the application, for visual representation.

-Saving the data-
Saving the data is also very simple, thanks to the data structuring. Application is able to collect the data with a simple recursive function. Application then sends the to-be-saved data to the agent. Since the user of the AOM has very high degree of freedom in modifying the ontology, the changes have to be validated by the agent before saving them. This validation process is done with a Reusable Atomic Behavior that checks the given RDF Ontology for errors. If the validation is done successfully, the ontology can be sent for further processing and/or storage.

1.3 Policies in UBIWARE

Policies in UBIWARE are implemented by limiting the external actions any agent on the platform is able to perform. This approach is inspired by the fact that agents are autonomous, i.e. they can act without direct intervention from humans or other software processes and have their own actions and internal state. It is thus not needed to imply restrictions on the internal beliefs' structure of the agents. Policies are enforced every time when the agent has an unconditional commitment statement in its beliefs. The policy check resolves whether it is allowed to perform the action. The check is done in a few stages (see Figure 1.8).

At first, the UBIWARE agent has a policy checker object which is responsible for the check and the type of this checker is dependent on the type of agent. Infrastructure agents, for example, have a policy checker object which allows them to perform any action

without performing any actual check. Other agents first check the so called safe set of actions which they are always allowed to perform.

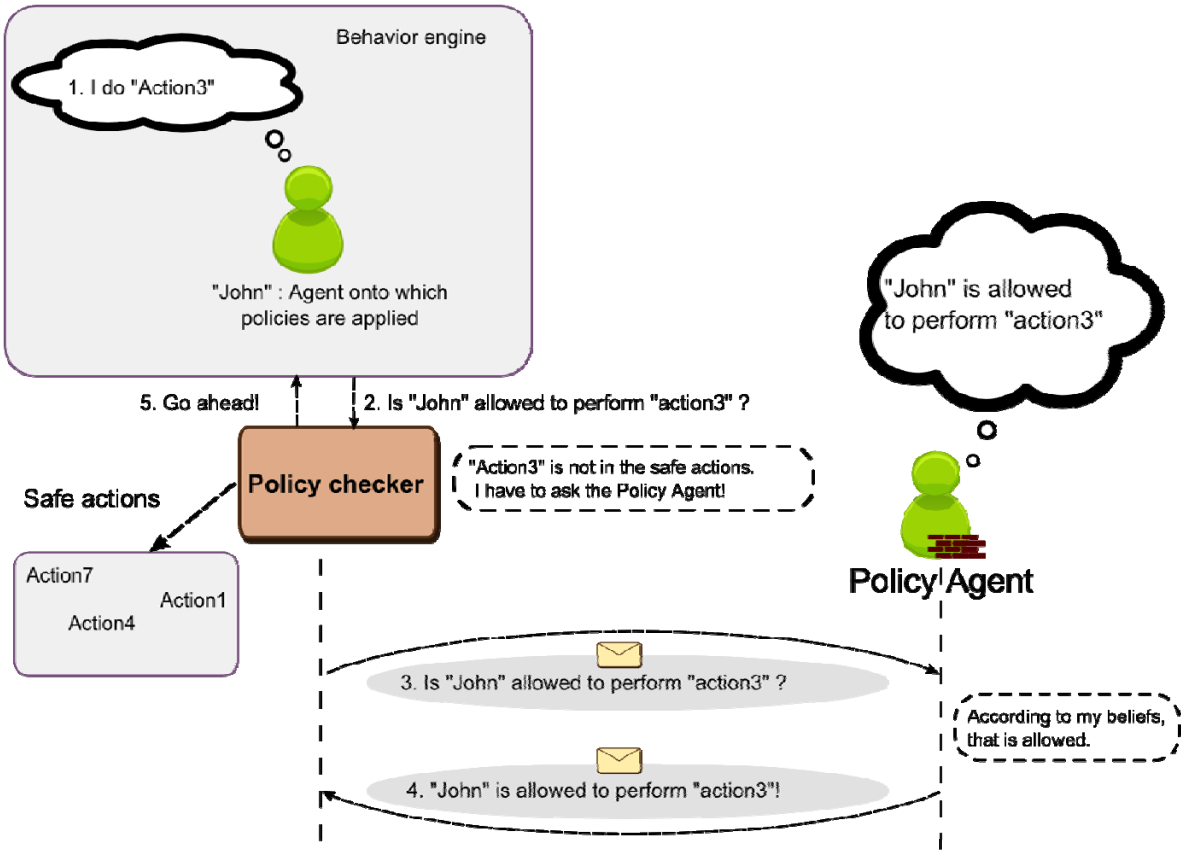

Figure 1.8 – Policy enforcement mechanism

Next, only if the first check fails, the parameters from the commitment statement are transformed to S-APL in textual representation. Then they are combined with the name of the action and a query message is sent to the policy agent which is responsible for replying to policy queries. When the response comes back and the action is allowed, it will be performed. Otherwise, the object container of the sapl:Denied sapl:add parameter is added to the agents beliefs.

1.4 Core Platform Improvements

The core platform improvements are one of the most significant ones in the platform, although they are hardly visible or even presentable. In the 3.0 version we have resolved a lot of performance critical bottlenecks and have made several extensions that facilitate the application development on the platform.

1.4.1 RDF2BEAN

When building Java-based web applications a problem arises how to transform RDF code utilized by agents into Java code and vice-versa. One solution could be to load RDF code

as a string and parse it in Java. There are several libraries capable of doing it. The problem is that the Java developer has to work with RDF on a low level – statement by statement. This way the code becomes more difficult to read and it doesn't correspond to the object-oriented nature of Java language.
A possible solution to this problem would be a creation of library that could transform RDF resources into Java objects and vice-versa. The best type of Java object would be a bean that would have several fields corresponding to object and datatype properties of a resource. There is a 3rd party library called Jena beans that accomplishes this, but it has one significant limitation. The library uses Java annotations for bean class definitions and field definitions. However, when the library reads an RDF code, it converts it into instances of corresponding Java beans, but it doesn't remember their URIs (Uniform Resource Identifiers), which is a potential problem source. For example, if the user changes the structure of bean instances and wants to save them back to the RDF form, he/she will not receive the same URIs as those that were read and therefore the generated document will not correspond to the rest of the document still residing in the agent or anywhere else on the platform.
This disadvantage encouraged us to implement our own library for RDF-bean conversion. The library introduces the concept of storage. It is an object that represents storage for RDF resources. The Java developer uses a storage object together with Java bean classes written by him/her that correspond to the ontology of the data that will be read. Each Java bean class corresponds to a class in the ontology. We say that the bean class handles an ontology class. The Java developer creates an instance of a storage and registers relevant Java bean classes. After the registration, the user reads the RDF data by providing a string represented in Notation3. The storage converts all instances of registered classes into Java objects. These objects can be retrieved from the storage based on their URI or based on their type. The developer may change properties of objects and subsequently save them back into RDF form without changing the URI.

1.4.2 Semantic Action Script

In the previous version of the platform, the communication between agents was handled by two Reusable Atomic Behaviors (RABs) – MessageSenderBehavior and MessageReceiverBehavior. They allowed the agent to send and receive an ACL message. The user was able to set conversation identifier (to match sending and receiving message), ontology (to match different conversation), content and more. The content could have been a string or a container containing SAPL code. This wide variety of options gave a lot of freedom to the agent developer. On one hand this was beneficial, but on the other hand, it was important to agree precisely on the communication model. In addition to that, user had to write a set of rules that were handling incoming messages. If an agent was able to receive many types of messages, it had to have many rules to handle them. Since all the rules share the same context called G context, the developer had to be cautious and not produce data that could trigger an unwanted rule. With increasing number of rules, the problem became more difficult to manage. Also, message handling rules looked very similar and many times the developer had to copy basically the same piece of code just with small modifications. If it was possible to pack this rule into a more compact form, it would improve the readability of the code.

The platform already introduced concepts as Listener, Informer and Believer. These were SAPL roles that an agent could download in order to receive certain capability. An example could be Believer role, which made the agent capable of accepting other agent's code. This way one agent could tell the other agent what to do, which was beneficial for coordination. The problem was that the Believer role made the agent to believe (meaning store) anything that other agent sent. This was a form of code injection, which is not considered a safe method of communication, since malicious code can be injected as well. Some filtering from the believing agent's side was needed. This brought us to the idea of creating a script that would give an agent the capability to provide some interface to the outer world, but at the same time to fully control what will happen after the message was received through this interface, not just blindly "believe" (store) it.

The Action script allows the agent developer to specify so-called action handlers that handle particular type of action message. In order to understand a handler, an action message will be presented first. Action message is an ACL message with a predefined structure. Its content is SAPL code, more precisely the content is exactly one RDF resource of type com:Action. The ontology of an action resource can be seen in Figure 1.9.

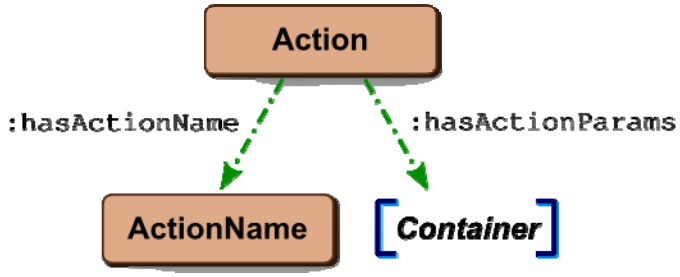

Figure 1.9 - Action message ontology

The action resource has an action name (referenced to by point to an ActionName resource) and list of parameters associated with it. There is no "executable" code included as it was in case of Believer script. The parameter list depends on the ActionName. Usually ActionName and parameter list a matter of agreement between the action sender (agent that requests an action) and action receiver (agent that performs the action). The developers agree on particular ActionName resource they will use to mark the messages and the parameter list that they will exchange.

The agent receiving an action message has an action handler prepared. An action handler is an RDF resource with 3 properties (Figure 1.10). In order to distinguish between handlers, ActionName has to be specified. This is done through *:handles* property. Property *:params* points to a container that contains a list of parameters that are expected and variables bound to parameter values. These variables are then used in a container that is bound to the handler by the *:code* property. As the name suggests, this container keeps the actual code of the action and may use variables defined in *params* container. This way the action sender agent only can specify the parameters and the code execution is fully controlled by the action receiver agent.

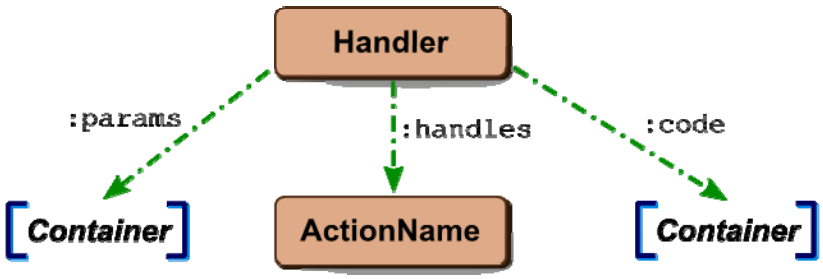

Figure 1.10- Action handler ontology

An example of an action object can be seen below. It is an action for querying all messages of some facebook user. The action object has action name *soc:getAllMessages* and 4 parameters specifying sending agent (in order to know who to respond to), facebook profile ID and time period which should be queried.

```
:action1    rdf:type com:Action ;
            com:hasActionName soc:getAllMessages ;
            com:hasActionParams {
                  com:sender sapl:hasValue “agentPeter” .
                  soc:profileID sapl:hasValue “54981618” .
                  soc:fromDate sapl:hasValue “01-06-2010 00-00-00
            EEST” .
                  soc:toDate sapl:hasValue “10-06-2010 12-00-00
            EEST” .
            } .
```

Partial code of a typical handler can be seen below. In *params* section it queries the parameter values, bounds them to variables and in the code section it uses them as part of the code.

```
: getAllMessagesHandler rdf:type com:Handler ;
      com:handles soc:getAllMessages ;
      com:params {
            com:sender sapl:hasValue “agentPeter” .
            soc:profileID sapl:hasValue “54981618” .
            soc:fromDate sapl:hasValue “01-06-2010 00-00-00 EEST”
      .
            soc:toDate sapl:hasValue “10-06-2010 12-00-00 EEST” .
      } ;
      com:code {
            // . . . code . . .
      }
```

1.4.3 Development “under the hood”

Within the core improvements section we distinguish those improvements that are not immediately visible even by platform developers, but still have significant impact on the platform operation. The list of the changes is given below:

- Commands and the synchronization infrastructure
  - The agents provided by Jade have one thread which executes the behaviors of the agent with a round-robin scheduling algorithm. The same thread is used to maintain the beliefs' structure of the agent. If another thread would access the beliefs of the agent concurrent with the agent's thread, the internal representation of the beliefs might get corrupted. Those inconsistencies would in general be noticed by ConcurrentModificationExceptions. Version 2 of the UBIWARE platform provided two specialized solutions for external modifications of beliefs. The first one could be achieved by using the cyclic possibility of behaviors. The behavior was then executed and checked whether the other thread created or managed by the behavior was ready to modify the beliefs structure. If this was the case, the actual modification was performed in the agent's thread. The second solution concerned server and GUI events. These events received special attention from the agent in a therefore created behavior. This behavior checked whether an even arrives and if that is the case, it processes the event and provides a result back by for example writing the result to the java.io.OutputStream provided with the ServerEvent or modifying the GUI which sent the GUIEvent. The new approach unifies the two before mentioned solutions by using the command pattern. Using design pattern syntax, we could say that any client (other thread) is able to create a command (ubiware.core.commands.UbiwareAgentCommand) which can be received by the UBIWARE agent. The UBIWARE agent has a behavior which will be the actual executor of the action. This behavior is scheduled in the agent and is thus executed in a thread safe way. An extension of this system is the blocking command which allows another thread to block till the actual action is executed inside the agent's thread.
- Applications of the synchronisation infrastructure
  - *RabRunnable:* On top of this framework, a concurrent agent wrapper is created. This wrapper is called RabRunnable and can be used as a normal java.lang.Runnable. The advantage of the RabRunnable is that from within the runnable, calls to the agent can be made in a thread safe manner while preserving the possibility to schedule this Runnable in whatever execution service is desired for the application. An example of the use of RabRunnable is the HttpDataFetcherBehavior which fetches files from the web to the agent. When the file is big or the connection is slow and the input output operation would block often, it is beneficial to start a separate thread to fetch the file. Doing this with RabRunnable allows that thread to send the result of the fetch straight to the agent whenever it is ready without concurrency problems.
  - Initial Beliefs
    - When a UBIWARE agent is started from within another agent, it was not possible to supply the agent with initial beliefs dynamically. We could only give the agent certain roles and scripts which the newly created agent would then load from the repository. Using this command, we are now able to dynamically add beliefs to the agent upon creation. This means that we are for example able

to tell an application worker agent who its personal user agent is in the creation phase of that agent.

    - GUI implementation
        - The new way of implementing local GUI's for agents is using the command model too. Whenever a user action is performed in the GUI (Which is in case of Swing running in the AWT event queue.), a command object will be created and send to the agent. The result of the command object is that the agent gets in thread-safe way knowledge about the event. Then the agent can decide about consequent actions. An example of this system can be found in the buttonGUI where button presses are converted to agent.

- AgentServerBehavior
    - If an individual agent wants to run its own web server, it can do so easily by using the AgentServerBehavior. This was also possible in the old version of the platform but is now optimized by using the concurrency tools. When an agent starts the behavior, it will get beliefs injected about arriving web requests. Upon that, the agent has to prepare a response to the events and start for example HTTPResponseBehavior using its response as a parameter. Internally, Jetty (an enterprise level web server) is used to receive the HTTP request. This ServerEvent is then placed on the blackboard of the agent. After that the agent gets information about the request to its beliefs. When answering the request, the ServerEvent from the blackboard is taken and the answer is sent using that ServerEvent object.
- Enhanced Reusable Atomic Behavior
    - Version 2.0 of the platform has provided the possibility for developers to develop own reusable atomic behaviors. In the version 3.0 we took this approach to the next level by providing the enhanced reusable atomic behavior class. The structure of this class was born from the fact that most Reusable Atomic Behaviors work in the same fashion which tempted us to use the template method pattern. Its functionality could be divided in two parts. The first part is parameter initialization and the second one is the actual action code. Whenever the parameter initialization fails, the action code will not be executed. Another aspect of this class is a frame for the parameter initialization itself. Parameters coming from the agent must be checked for correctness. The enhanced behavior provides convenience methods which return the parameter value checked against the requested criteria. The bottom line of this class is to provide a convenient and programming error-prone infrastructure for behavior developers. Approximately 2/3 of the currently existing RAB's are written with enhanced technology.
- SaplBuilder
    - While programming agents, it is often needed to produce S-APL code which is given to the agent. This used to be done by producing an S-APL String representation which was then provided to the agent. The new approach allows for creation of S-APL code in an object oriented fashion. Statements are assembled from Subject, Predicate and Objects and containers and documents from statements. This way, we can assure that the generated S-APL code will always conform to the S-APL syntax and

thus we avoid hard to find programming errors. Another benefit is that in the future, there will be stronger support for this way of providing S-APL code to the agent. The result of this will mainly be seen in performance since the S-APL code does not have to be parsed after creation with objects.

- S-APL compilation and production
  - Production: The beliefs of the UBIWARE agent are composed of S-APL statements. Internally, the beliefs are stored in a way which allows the agent to transform them easily. But whenever the agent is communicating with the outside world or with other agents, the internal representation has to be transformed to a textual representation. This textual representation is then used in the communication. When communicating about data sets bigger as we had been using in the second version of the platform, we noticed that the speed of generating a textual representation of the internal S-APL code was too slow. This was solved by using a different algorithm for producing the S-APL text. Theoretically; with *s* being the number of statements and *d* as the depth of the code's container structure; the older implementation worked with an efficiency of magnitude *O(s+s*(4s)) = O(s^2)* and created *O(d)* string builders during the process. The algorithm was very copy intensive. Potentially, for every statement the whole result had to be copied 2 times from one buffer to another one, resulting in the whole result being copied 2*s times. This gave also problems regarding memory usage and waiting for garbage collection. The new algorithm performs its actions in O(s) efficiency and uses exactly 1 string builder during the process. An old burden; ID's which were generated even tough not used in any further reference in the generated S-APL document; was also removed during re-implementation. The speed improvement is not only visible theoretically, - the practical gain of using new algorithm and better use of recursion, resulted in major speed ups. In a concrete case when converting a bigger data set (183262 statements) from the agent's beliefs to a flat textual representation, we saw in the old implementation that 30 minutes was not enough to do the job. In the new implementation, the generation takes a couple of seconds! One more speedup and bug fix was realized by reusing a buffer of white spaces. When the data set was nested too deeply in the old implementation, an IndexOutOfBounds exception was thrown because the end of the buffer was reached. In the new implementation, the buffer is growing if the situation demands it. The improvement of this conversion can be noticed trough out the whole platform, because any message which is sent between agents is now generated with the more efficient algorithm. Other places, where this speed up is noticed is in the debug view and backup of beliefs.
  - Parsing: Parsing S-APL has been speed up by making parts of the compiler static instead of recreating them on every parse action. The result is less significant as the speed-up in S-APL production but still makes a difference for message receiving and adding beliefs to the agent.
- Generating unique instances
  - In version 2 of the platform, the current time of the Java virtual machine was often used to generate unique ID's for all kinds of resources. Even inside S-APL code, the *sapl:Now sapl:is "time"* belief of the agent is often

used to create a new ID or URI for a resource. The problem with using the current time is that getting the current time from the operating system requires a trap to kernel (or similar) which always implies a delay. Another problem is that two consequent calls to the operating system are not guaranteed to result in different values. These problems are solved by creating unique instance generators. These generators use an internal counter which can only be incremented in a synchronized way. Using the unique instance generator from S-APL code trough the ID() expression call, we can create a unique ID or resource name in S-APL. Whereas the "current time" function can be used only for its direct purpose.

- UBIWARE agent internals
  - The UBIWARE agent code has become cleaner by extracting the Blackboard and resource prefixing capabilities to dedicated objects. Furthermore, methods used to query the agents' beliefs structure are no longer exposed to the internal representation of the beliefs. Instead they receive a set of bindings from which they are able to find the values of their specified variables. This gives us also the possibility to improve the internal representation without having to change all code using the agent.
  - When loading RAB classes, the old implementation used reflection in a straight manner to load the class for a certain RAB. Then the class object was stored in the agent for later reuse. Version 3 of the platform uses a dedicated RAB class loader which we call RABLoader. This class is a java.lang.ClassLoader which is guaranteed to only load java classes which inherit from Reusable Atomic Behavior. The RABLoader is shared among all UBIWARE agents in the same virtual machine, making RAB loading only needed once.
- General improvements
  - Trough out the whole code base, we started using better programming techniques and use a smarter choice of collections for working with data. Examples are using unsynchronized types where there was no need for synchronization (usage of ArrayList instead of Vector and StringBuilder instead of StringBuffer). One more efficiency gain is realised by having String constants for the most used URI's.

UBIWARE Deliverable D3.3:
Workpackages affected WP7, WP6:

2 A Use Case: Mashupper – Agent-enabled Social Web

The Mashupper application uses data from three prominent social network platforms: Facebook, LinkedIn and Twitter. As the largest player in the field with more than 500 million users, Facebook is becoming a virtual world of its own. It fulfills the need for general socializing in the web and covers both leisure and work. LinkedIn on the other hand is very much profiled to the business and professional side of the social networking. Then there is the Twitter, which has capitalized on the people's need to hear and to be heard, preferably in real-time. The social connections in Twitter are looser than in LinkedIn or Facebook. Everything you write to Twitter (or tweet) is public and can be read by anyone. In Twitter, user can start to follow his or her friends or actually anyone who seems interesting enough, in order to receive updates from those people in real-time.
Facebook, LinkedIn and Twitter is a good set of services to start with, but there are many other interesting social networking sites out there. Adding new information sources to Mashupper is relatively simple task, thanks to the agent-based architecture of UBIWARE platform, given of course that the social network service provides some kind of API for external services.
The scope of the Mashupper is built on top of the concept of Personal User Network (PUN), which is defined as the combination of different kinds of human connections, which a particular user may have in the social web.

PUN is used to link the different online identities or profiles into one and same connection, through which the user can observe the integrated presence of that connection in the current web2.0 landscape.

Mashupper web application was developed using web development framework called Vaadin (http://vaadin.com/). Vaadin is a mature framework based on Google's web toolkit. It is developed and maintained by IT Mill company. Vaadin includes good collection of UI components for creating snappy looking web interfaces with relative ease. One of the main reasons why Vaadin was selected was the fact that the web applications can be written entirely in java, without having to bother with web related technologies such as HTML, CSS or Javascript. Vaadin made it easy especially for developers with experience in building Swing based Java applications.
In order to make a successful UBIWARE-driven application we start with the common domain model construction and build a Social Ontlogy.

2.1 Social Ontology

Social ontology is a domain specific simple ontology for the social web. The main concepts and their properties are depicted in the Figure 2.1. PersonalUserNetwork can basically be thought of as set of people that the user has some kind of connection with. These connections are modeled using HumanConnection class. The name was chosen to emphasize the fact that connection refers to a single human being. This person can then be represented in the social web by multiple digital identities, modeled in the ontology as SocialNetworkProfiles. SocialNetworkProfile is the common parent of more service-specific profiles such as FacebookProfile, LinkedInProfile and TwitterProfile.

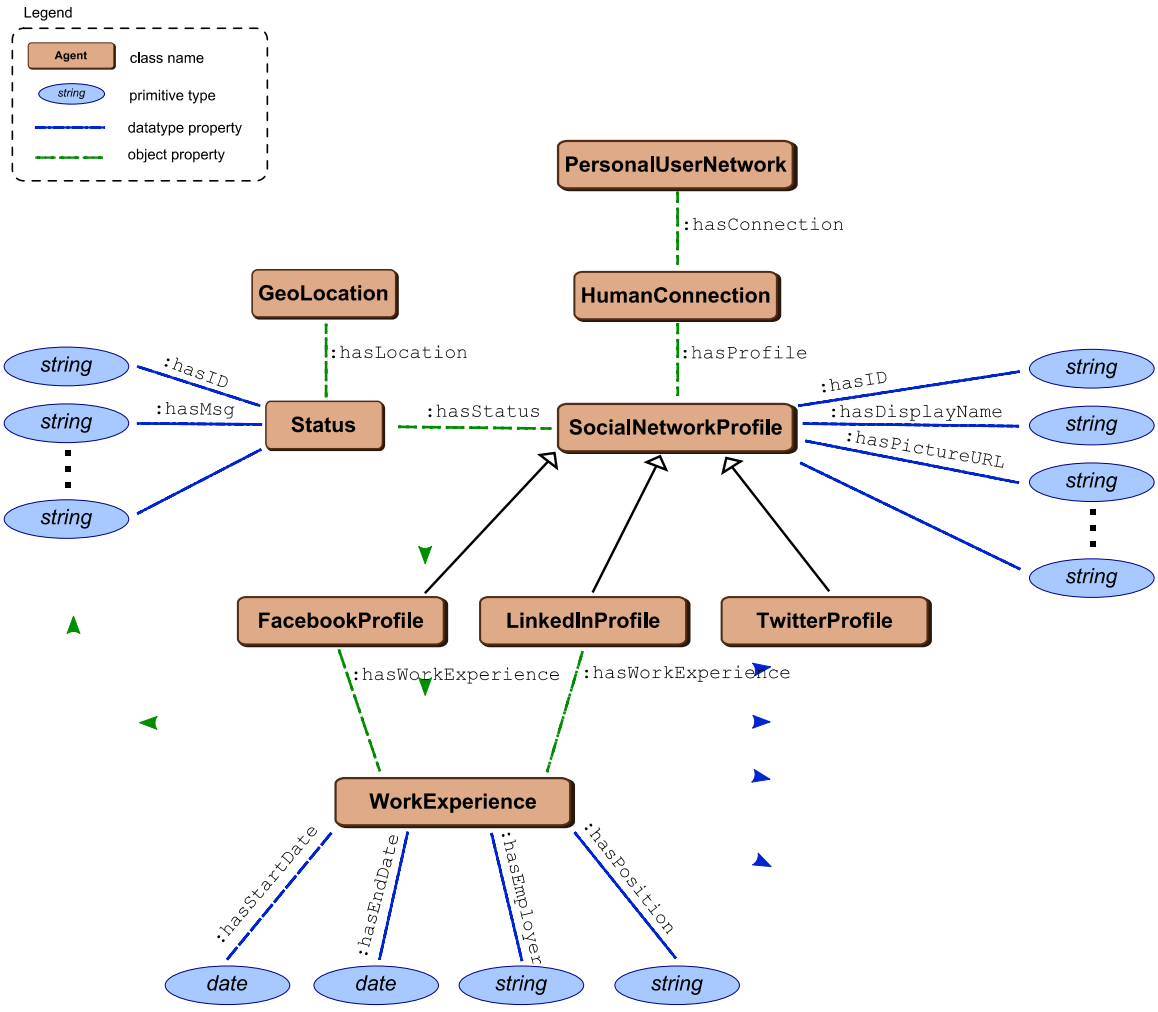

Figure 2.1- Social Ontology

The concept Status refers to any kind of state information available. It is used to bring together status updates from Facebook, network updates from LinkedIn and tweets from Twitter. After all, they are all used to represent the current state, or status of mind of the particular HumanConnection at a particular moment in time. Location, as an increasing important factor of the social web, can also be included as part of the Status using GeoLocation class.

2.2 Agent Architecture

Mashupper consists of four applications. In addition to the web application providing main Mashupper UI, there is one web application for every social network adapter. Adapter UI is used to authorize adapters.

Mashupper is designed to run under UbiwareDesktop. It means that every user receives four worker agents, each responsible for one of the web applications. Normal infrastructure agents are naturally also running. Figure 2.2 shows all the agents involved and their roles.

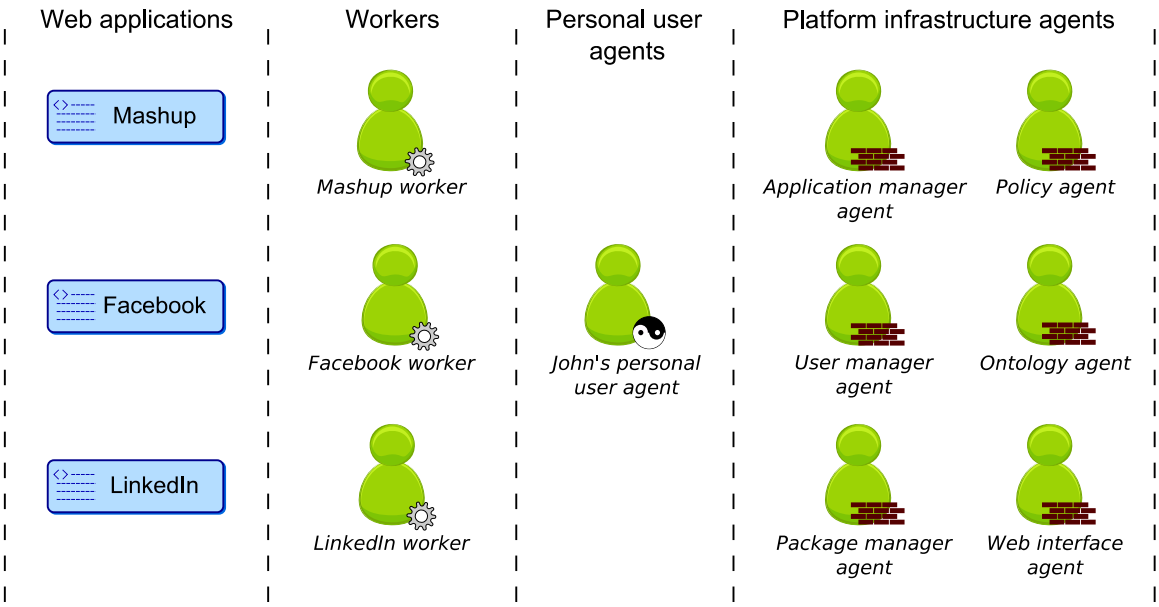

Figure 2.2- Mashupper Application Agents

MashupWorker is responsible for storing the knowledge about the Personal User Network of the user and works directly with Mashupper web application. It coordinates all the message interaction between involved agents. Adapter workers handle all the information related to the external service API, such as URLs of the service interface, client ID, keys for signing and access tokens for authorizing requests. They also know how to access the external service using API-specific RABs. There is one worker for Facebook, LinkedIn and Twitter.

Figure 2.3 shows an example of agent interaction coordinated by the MashupWorker, where web application requests for detailed profile information for a certain HumanConnection. In this case the HumanConnection has been linked to both Facebook and LinkedIn social network profiles. The MashupWorker queries worker agents of both services, waits for their results and returns the combined information to the web application in RDF format.

In the current use case Mashupper is the only agent using the services provided by social network adapters. Nothing stops other agent applications in the platform from taking the advantage of the same information. Or even better, they can use the Personal User Network stored at the Mashupper agent and target their actions to a certain

HumanConnection instead of a single service. In other words, Mashupper can be reused as a component in future application scenarios.

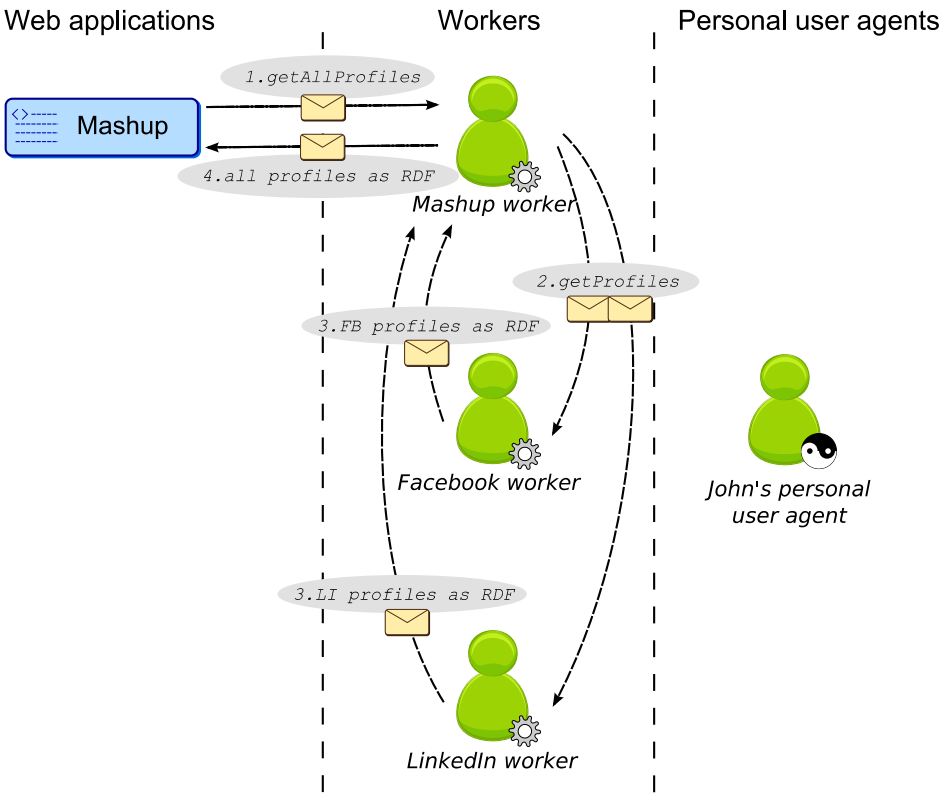

Figure 2.3- Mashupper agent interaction

2.3 User Interface

When a user starts Mashupper for the first time, the PUN is naturally empty as shown in Figure 2.4.

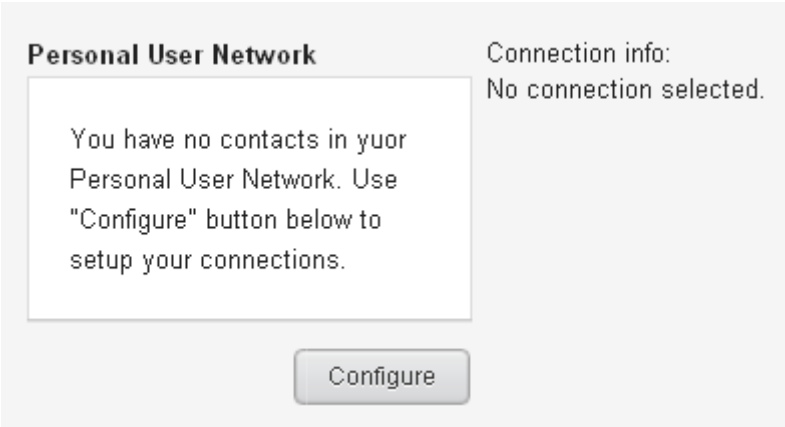

Figure 2.4- Personal User Network is initially empty

User should initiate the process of Personal User Network construction manually by starting the configuration, which can be accessed by clicking “Configure” button. Configuration view shows the list of available social network adapters and the content of

the users of current Personal User Network. Initially, all the adapters are in the offline or non-authenticated state (see Figure 2.5). User must activate adapters by granting the access to its accounts in social networks as explained in the next section.

Figure 2.5- The Adapters are initially offline

2.3.1 OAuth authentication

All three available social adapters use APIs that utilize Open Authentication (OAuth, http://oauth.net/) as their authentication mechanism. Social network adapter can be activated by right-clicking over the adapter and selecting "Activate" link. This opens up the UI of the adapter with the instructions how to authenticate and authorize the selected adapter (see Figure 2.6).

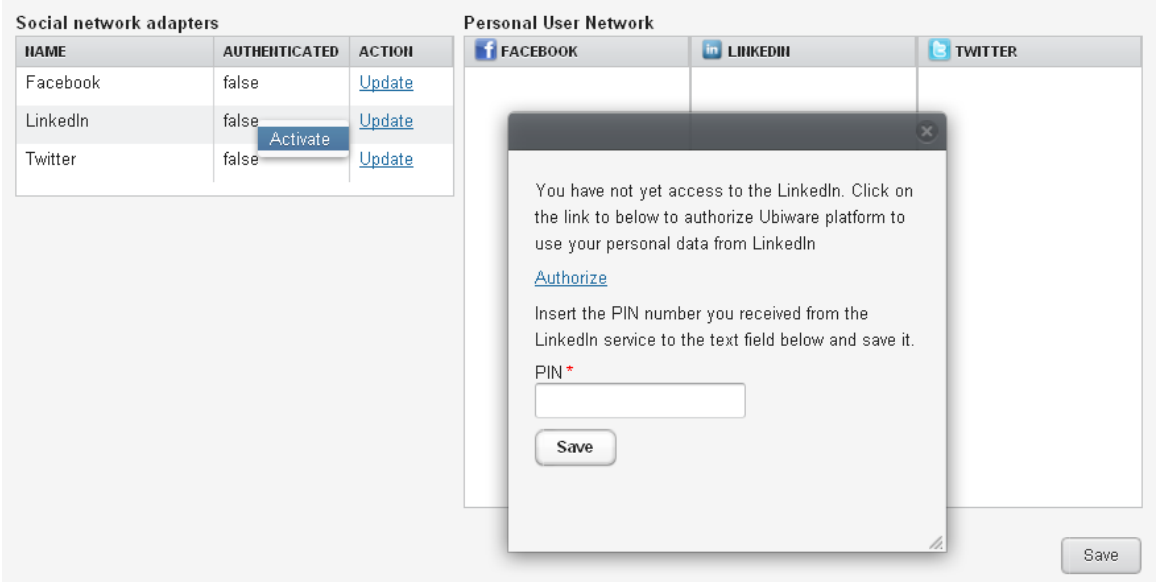

Figure 2.6- Activation UI for the LinkedIn adapter

After following the "Authorize" link and saving the PIN provided by the external service, the social network adapter UI displays the updated state of the adapter to the user (see Figure 2.7).

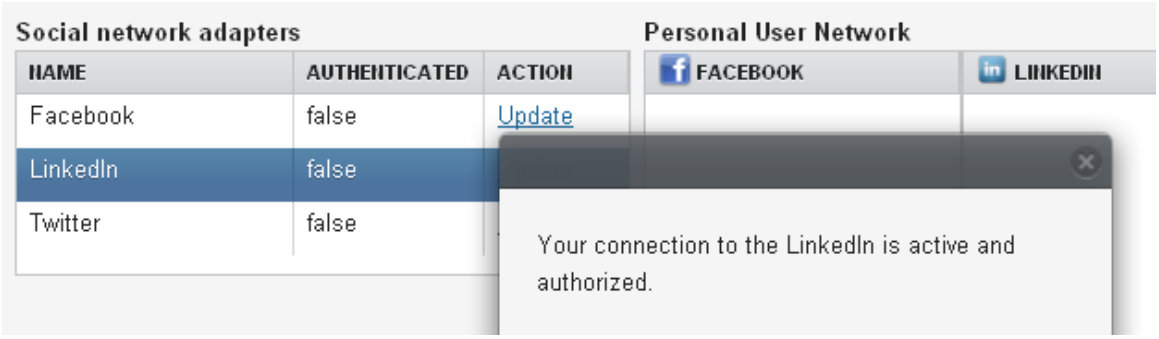

Figure 2.7- LinkedIn activation was successful

When the user closes the social network adapter UI, Mashupper application refreshes the list of adapters, now showing the LinkedIn adapter as authenticated and ready for use (Figure 2.8). Authentication process should be completed with all the adapters from which the user wants to retrieve data from.

Figure 2.8- The list of social network adapters refreshed

Adapter remains active and authenticated until the user manually revokes the access token it has received in the authentication process.

2.3.2 Building Personal User Network

When the appropriate adapters have been authenticated the user can start building his or hers Personal User Network. The user can retrieve the list of people with whom he or she has any kind of contact using the "Update" link associated with every adapter (Figure 2.9). In Facebook this means the list of friends, in LinkedIn, - the connection or contacts of the user and in Twitter – the list of people the user is following (Figure 2.10).

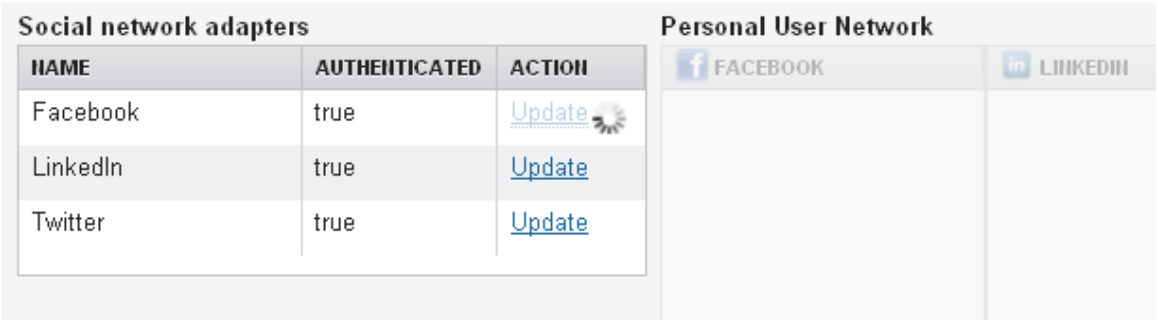

Figure 2.9 - Updating Personal User Network with friends from Facebook.

Figure 2.10 - After successful import, the friends are added to Facebook column of the Personal User Network.

Figure 2.11 shows the state of PUN after user has retrieved contacts from all the available adapters. Every row in the PUN table represents one human connection. Mashupper supports simple merging of the data using the display name. In the example it has automatically merged Pete Smith from Facebook and LinkedIn into the same connection.

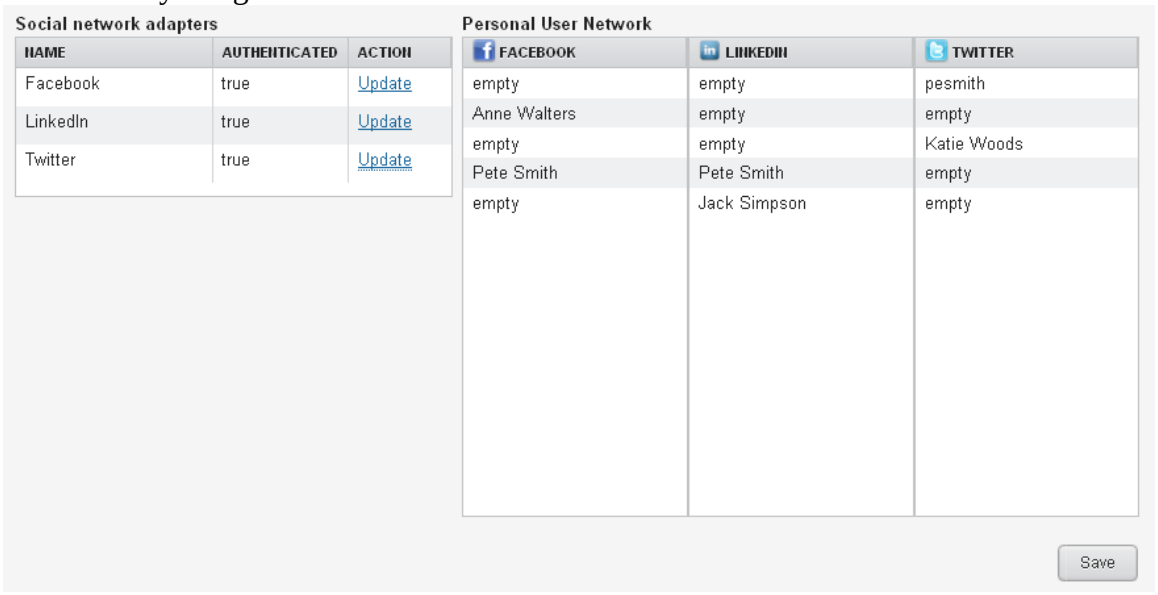

Figure 2.11 – State of the PUN after all adapter updates

The same Pete Smith is also found in the Twitter column, but under the name “pesmith”. The user can now manually link “pesmith” in Twitter with the “Pete Smith” in Facebook and LinkedIn by drag-and-dropping the name “pesmith” onto to the same row with two other Petes. Mashupper will replace the “empty” label with “pesmith” and remove the first, now empty, connection (Figure 2.12).

After user has created all the necessary links between different online identities, the PUN is saved to the MashupperWorker agent by clicking the “Save” button.

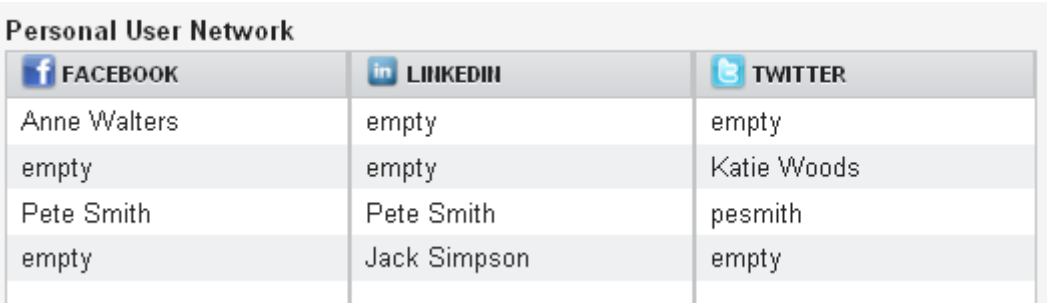

Figure 2.12 – Organizing PUN by linking profiles from different networks

2.3.3 Using Personal User Network

When the user opens up the Mashupper application, the content of his or hers Personal User Network is shown in the left column. Table lists the names of HumanConnections in the Personal User Network (Figure 2.13). Every connection is linked to one or more social network profiles as described in the previous section. When user selects one of the connections, Mashupper uses the associated profiles to retrieve information about that person from the external services. In the current implementation, information sources are Facebook, Linked and Twitter. The retrieved information is shown using three different panels.

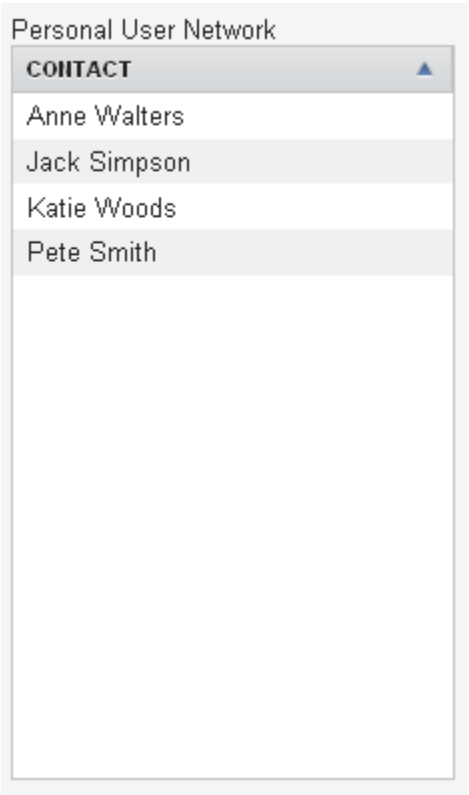

Figure 2.13 – PUN lists the human connections

Profile panel

Profile panel shows the combined profile information about the selected person (Figure 2.14). There are also links to person's profile page in all the "adapted" services. The current version of the profile panel includes only basic personal fields. Work experience is retrieved from both Facebook and LinkedIn and combined into single list. If user hovers the mouse pointer over the values in the table, Mashupper shows the source of that particular piece of information. For example in Figure 2.15 the name "Pete Smith" was found in both Facebook and Linked and merged as single value, but in Twitter the person uses alias "pesmith".

Figure 2.14 – A profile panel view

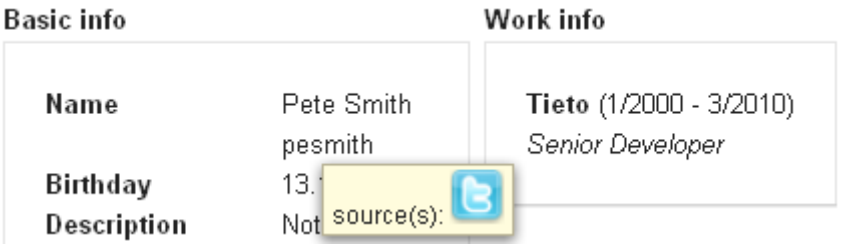

Figure 2.15 – The source of the information is shown by pointing the mouse over it

2.3.4 Geographical status updates panel

The next panel uses Google Maps to display geo-tagged status updates on a map (Figure 2.16). Panel is automatically updated every minute and the new updates with geographical information attached to them are added seamlessly to the map. User can click on the marker to display the status message.

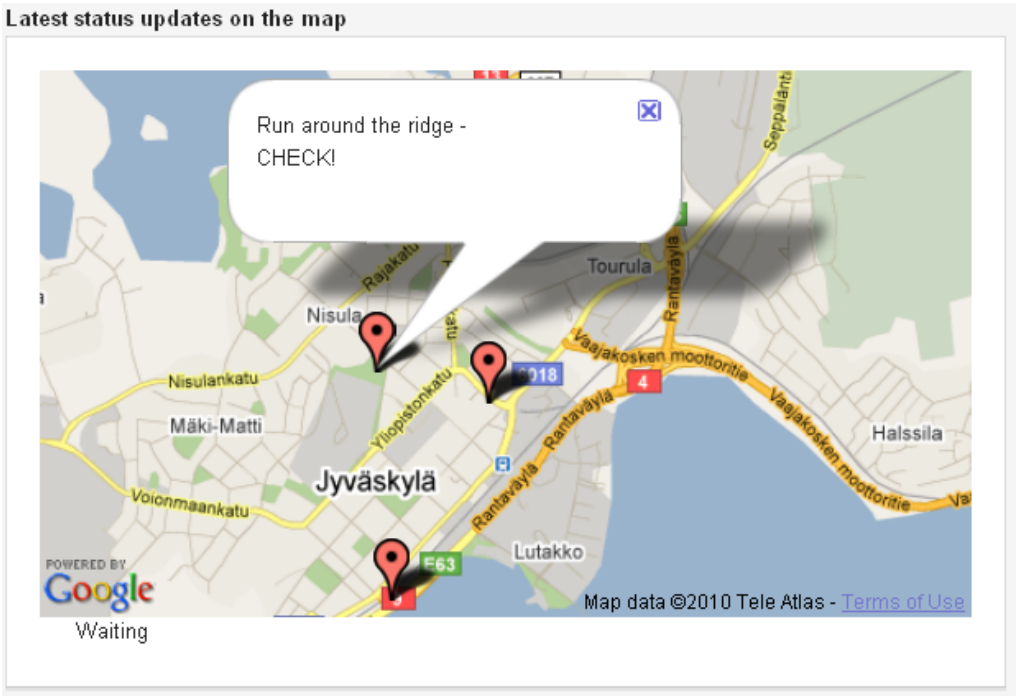

Figure 2.16 – Latest status updates on the map

2.3.5 Activity timeline

Activity timeline panel uses the Javascript widget from SIMILE (http://www.simile-widgets.org/timeline/) to visualize the stream status updates on a timeline. User can scroll the timeline back and forth and click on updates to display the whole message. Timeline is updated automatically every minute with the new status updates from Facebook, network updates from LinkedIn and tweets from Twitter (see Figure 2.17).

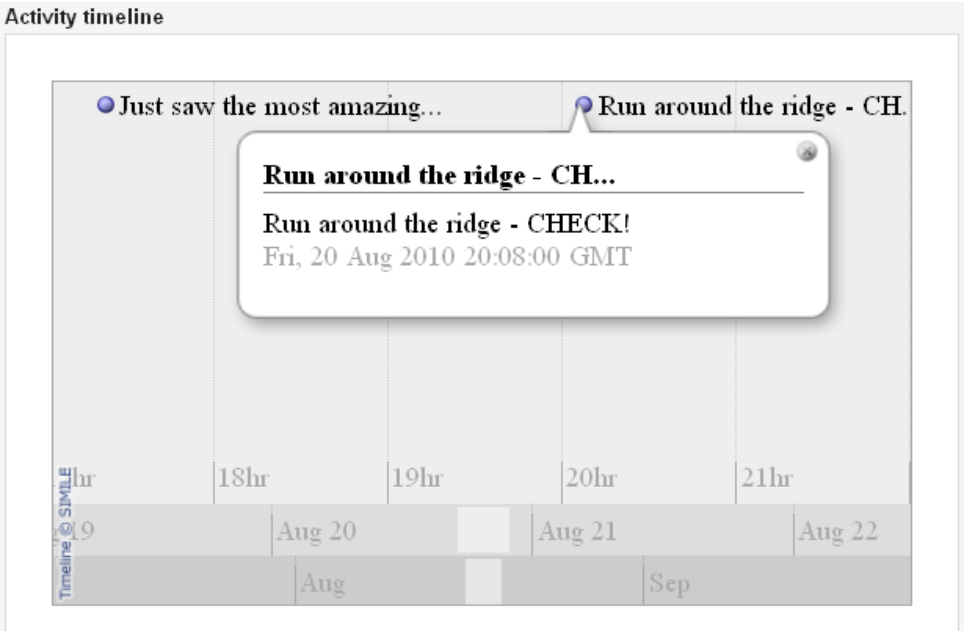

Figure 2.17 – Activity timeline

2.4 Conclusions

By developing the Mashupper application we tried to demonstrate the applicability of the UBIWARE 3.0 platform to the variety of tasks, not necessarily related to the heavy industry domain. The easy implementation of the Social Networking demo has proven that UBIWARE is genuinely middleware solution – with globally broad scope of application areas. Whatever we develop within the platform, we can reuse further in absolutely different application cases and scenarios. The social networking blocks and components are now available as reusable parts for any new application case.
In the nearest future we plan to extend the platform by giving the user more freedom in combining different information sources and functional blocks in the simple Lego-looking way, thus allowing the user-driven customization and even creation of agent tasks.

UBIWARE Deliverable D3.3:
Workpackage WP5:
Task T3.2_w5:

3 Smart Interfaces: Context-aware GUI for Integrated Data (4i technology)

3.1 Background

According to the vision presented in Deliverable D3.1 this year, source data adaptation is on of the challenges that we have to solve to make 4I Browser more or less working system. 4I Browser is kind of engine that provide context-sensitive visualization of resources via MetaProviders, it provides interoperability between different resources and services and adds some additional functionality. Thus, repository of resource descriptions is an input data for the Browser. Current prototype has default imbedded sample repository of resource descriptions in internal format. To make Browser able to work with any external repository, we have to elaborate general adapter that enable to convert data from any format to the required one. New data formats appear all the time and will require new adaptation modules. Thus, optimal way for such module elaboration is to make it extendible, be able to add new adaptation sub-module for new data format transformation.

Regarding to the plan of Inno-W industrial case, we elaborated adaptation sub-module to convert their RDF (N-triple) repository into internal format. It was just one hard-coded adaptation sub-module that transforms all the instances of “Proposal” class to internal format with a respect to supported data fields of internal format to be browsed through 4I Browser in the context of close/similar resources. Following this approach and with a purpose to allow user import and transform any repository (in RDF or N-triple format) him(she)self, we elaborated general adapter with graphical interface for necessary transformation specification.

3.2 ResourcesCloseness_RDFConvertor - general RDF adapter for 4I Browser

3.2.1 Adapter functionality and architecture

According to the general architecture of 4I Browser, adaptation of data sources is done via importing the source to the Browser and converting the data to the internal format. In

the same time with converting the original data format to internal one, adapter have to build all necessary visualization context descriptions and provide visualization module (MetaProvider) binding information.

ResourcesCloseness_RDFConvertor as well as a previous InnoW_Idea_RDF_Convertor is developed as remote services with three input parameters (“resultType”, “servletURL”, “sourceFileURL”) and works with both input formats: RDF with XML serialization and RDF in N-triple format. Description of the outputs and responses are pretty the same as in previous convertor (see Deliverable D3.2). The main difference is a provided graphical user interface that allows convertor configuration for appropriate source transformation.

With a help of convertor user makes a copy of the repository that contains instances (instances of the selected by user resource classes) with the properties of 5 main field/property types supported by the distance measuring function (elaborated during the previous year as a part of visualization module that shows resource closeness). We consider 5 types of the properties/fields: *simple text field* (string), *keywords* (set of simple text fields), *complex text field* (finite set of text sub-fields with the values from the defined set of them), *interval field* (with specified beginning and end of the interval) and field with *simple numerical* value. Thus, graphical interface allows user to specify necessary amount of the properties/fields and define correspondent values.

3.2.2 GUI of the convertor

To import a new source to the Browser, user should click “Import” button in the top left part of the Browser main window. In the popped-up window user has to specify a source (URL or local location of the source) and select appropriate convertor from the list of available in the Browser convertors. If the ResourcesCloseness_RDFConvertor was selected, user gets a window for convertor configuration (see Figure 3.1).

Figure 3.1 – Convertor configuration interface

After the launch, convertor checks a source and provides a list of resource classes mentioned there. User may transform all of the classes or necessary amount of them. With selected resource class, user gets a list of available (mentioned) in the source resources properties in the “Resource Properties” box on the left side of the window. On the right side user has “Transformed Properties” box that contains a list of the transformed properties/fields. To start filling this box with transformed properties, user has to select one or several (specification of the different transformed property/field types may require/support different amount of initial resource properties) properties in the “Resource Properties” box and choose one of the transformation property/field types (see Figure 3.2).

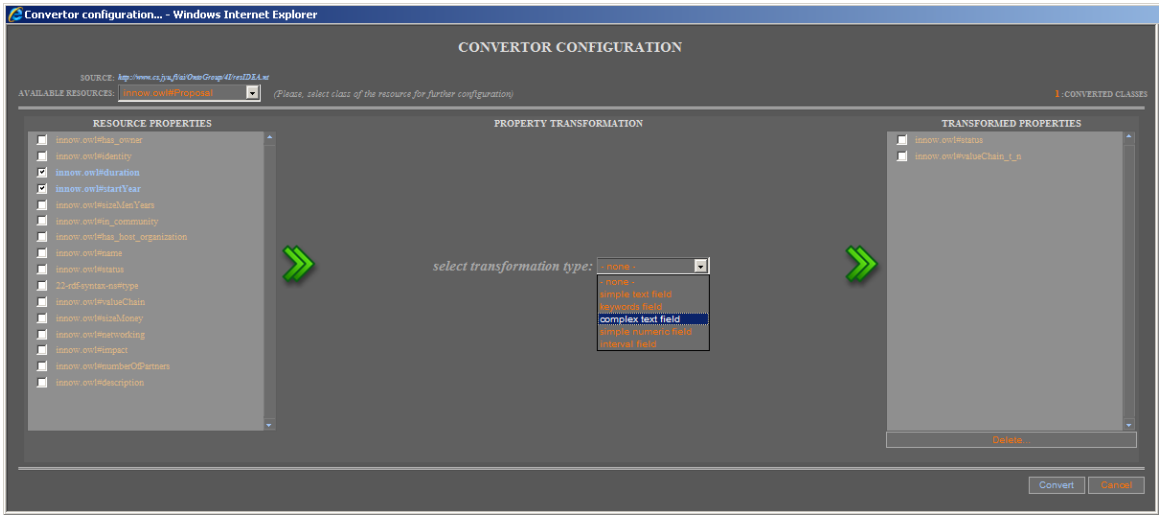

Figure 3.2 – Transformation property/field type selection

3.2.2.1 “Simple text” and “Simple numeric” fields

“Simple text field” and “Simple numeric field” are very simple transformation fields that require from user just to specify a field name. The value of the field is a value of the property selected in the “Resource Properties” box. Current implementation supports only one-to-one type converting for those fields (see Figure 3.3).

Figure 3.3 – “Simple text” and “Simple numeric” fields transformation

3.2.2.2 "Keywords field"

"Keywords field" is more complex transformation field. Here we can have one-to-one or many-to-one types of transformation. The value of the field is a value of one of the property selected in the "Resource Properties" box or merging of the values of several selected initial resource properties. At the same time, value of each property can be taken as it is (without modification) or it can be tokenized by default tokens or user specific ones defined by user (see Figure 3.4).

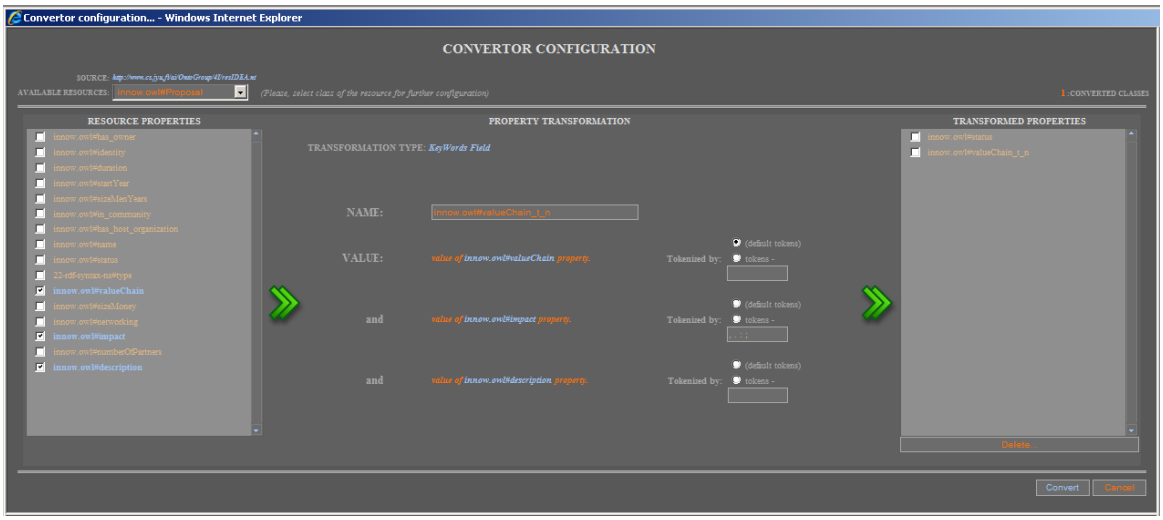

Figure 3.4 – "Keywords field" transformation

3.2.2.3 "Complex text field"

"Complex text field" is a finite set of text sub-fields with the values from the defined set of them. Amount of sub-fields equals to amount of selected initial resource properties. User has to specify a field name as well as the names of the sub-fields. Convertor provides a list of possible values for each sub-field (there are values taken from all the relevant resources from a source). Additionally, user can add a new possible value for the sub-field or deselect available values to reduce amount of them. (see Figure 3.5).

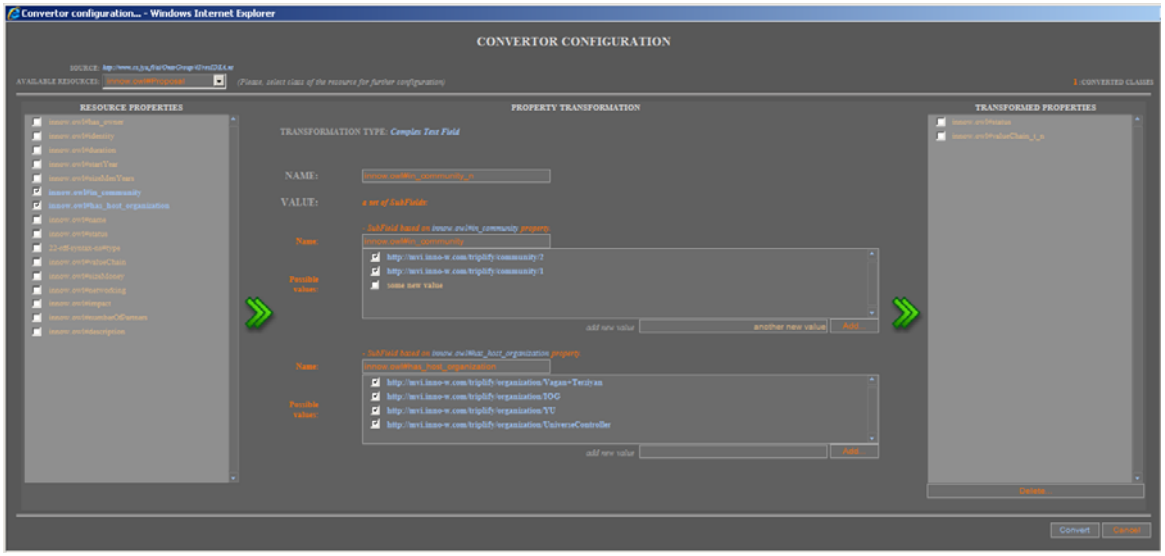

Figure 3.5 – "Complex text field" transformation

3.2.2.4 "Interval field"

"Interval field" has fixed two parameters – beginning/start and end of the interval. Interface provides a possibility to define the values for these parameters in a form of formula that includes the values of the selected initial resource properties. During the converting process final values of the beginning/start and the end of interval will be calculated based on provided formula and the values of mentioned resource properties (see Figure 3.6). User has pay attention to normalization of the properties values and be sure that the beginning/start of the interval is less then the end.

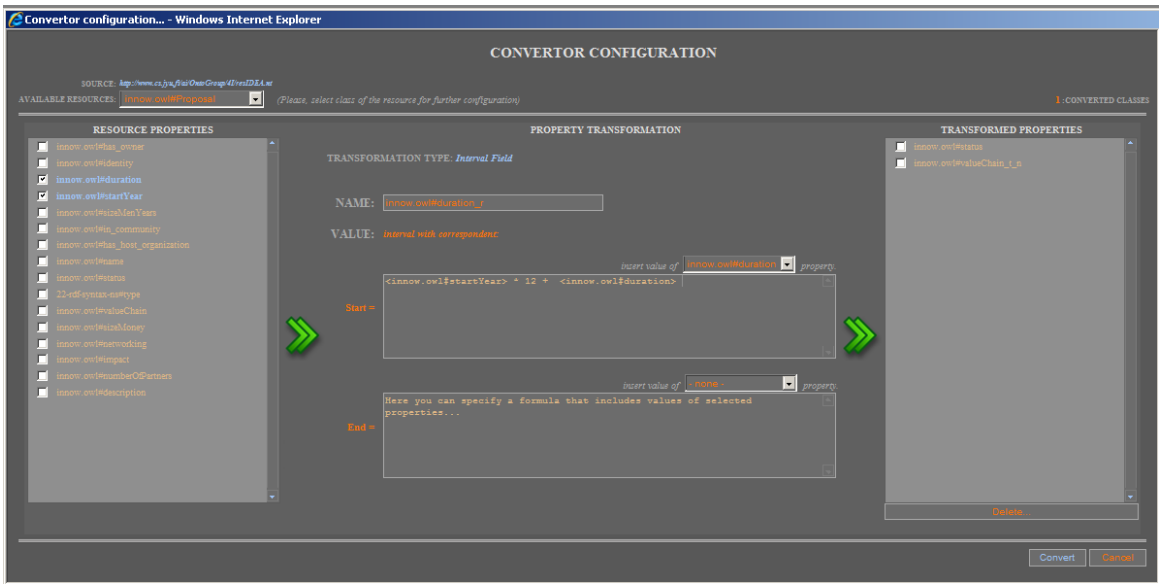

Figure 3.6 – "Interval field" transformation

As a result, convertor generates a file that contains converted resource repository in internal format and all necessary extensions for visualization contexts, MetaProvider descriptions. From that point user can continue with a Browser as before: search for the resources within an imported repository, manipulate with the contexts, visualize, etc.

3.3 Conclusions and future work

Current implementation is a working version, but due to the short development period, current version does not contain all planned functionalities and is not tested well. For the next period we are planning to make a comprehensive test of the current version and to complete development of such functionalities as: storing and reuse/modification of the convertor configurations, storing of the converted repositories (to avoid unnecessary adaptations), import/connection of the new adaptation sub-modules to the Browser.