Ubiware infrastructure guide

Authors:

Michael Cochez

Michal Nagy

Organization:

Industrial Ontologies Group (IOG)

University of Jyväskylä

Version: 1.2

July 24, 2012

Changelog

Date	Who made change	Version	What changed
13.12.2011	-	1.0	-
23.07.2012	minagy	1.1	Formatting, typos, extra clarifications
24.07.2012	minagy	1.2	Added (copied) section about how to work with WIA

Table of Contents

Preface

This document describes the infrastructure built on top of the Ubiware platform providing support for implementing ubi packages. This documentation is needed if one wants to develop new or improve the existing infrastructure for the Ubiware platform. This documentation is not required in order to create ubi packages. For information about ubi packages, see HERE.

The Ubiware infrastructure consists of several Ubiware agents and agent types which each have a specific set of tasks. This document outlines which tasks each infrastructure agent performs and in where the implementation of this functionality can be found. This document does not provide a deep discussion on the actual implementation of the functionality since the exact implementation is subject to frequent improvement. There are also four web applications which are part of the infrastructure. These are the desktop, the user manager, the application manager and the package deployer. We will not go into details on how the end-user uses these applications. This information can be found *Application user guide*.

Introduction

Ubiware is a type of middleware. Therefore it consists of a middleware platform (or just platform) and applications running on this middleware platform (or just applications). There is a group of agents associated with running the platform. We call these agents platform infrastructure agents (PIA). There is also a group of agents taking care of applications and users. This group of agents can be divided into three sub-groups: Personal user agents (PUA), Application worker agent (AWA) and Application infrastructure agents (AIA). This document mostly concentrates on PIA, but it also shortly explains other groups of agents.

Application- and user-related 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 will have only one personal user agent in the platform. The task of the PUA is to ensure the authentication for the user. The PUA can for example be asked whether an external user providing certain credentials is indeed the external counterpart of this PUA. PUAs are started by the User Manager (see further) when a new user gets registered to the platform.

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 application services to other agents. This type of agents is also in use when the application is not user-centric. Application infrastructure agents are started when the ubi package is deployed to the

system. Currently the starting is done by the Package Manager Agent. This responsibility might in the future be transferred to the Application Manager Agent.

Application worker agent (AWA)

This agent is the bridge between the personal user agent and the application 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. AWAs are started by the application manager agent. When started, each AWA gets information about which application it belongs to and which PUA it is associated with. Furthermore, it receives two scripts automatically on top of the scripts specified for this worker in the package description. These scripts a) register the worker to the UDF as being a worker for its PUA b) register and unregister all the handlers which the worker has in its global context c) subscribe for other worker agents of the same PUA which provide a certain service. (For more information, see the Ubiware Directory Facilitator.) Figure 1 shows the relation between AWA and PUA.

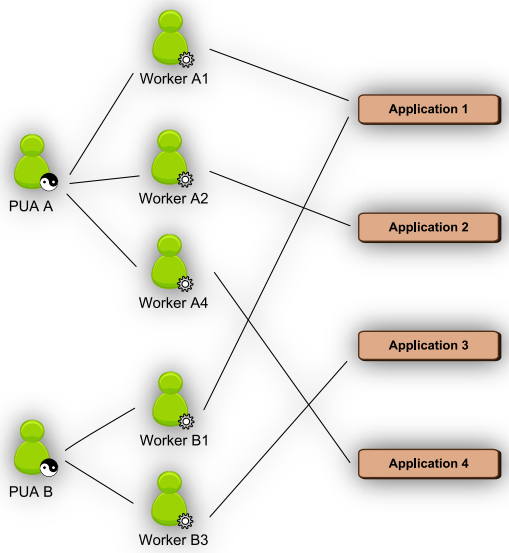

Figure 1: Relation between AWA and PUA

Platform infrastructure agents

Next to these three groups of agents of which there can be a variable amount in different platform instances, there are also a 7 agents providing platform services. These agents are called Platform Infrastructure Agents (PIA). In this subsection we shortly describe the platform infrastructure agents and their roles in the platform. These agents are application-

independent and are designed to handle the platform operation. In *Figure 2* one sees a general overview of the architectural agents on the platform.

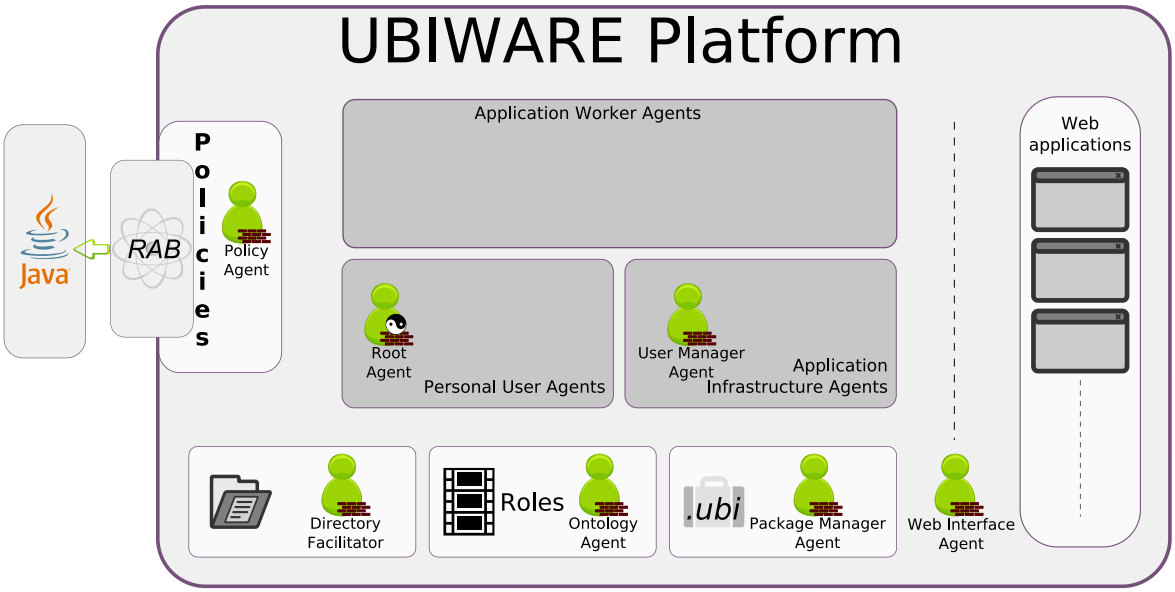

Figure 2: General overview of the architectural agents on the platform

Policy Agent (PA)

Description: The policy agent is the agent responsible for policy checks. 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 and would thus execute external behavior. The policy check resolves whether it is allowed to perform the action. The check is done in a few stages (see Figure 3).

In order to enforce policies, each UBIWARE agent has a policy checker object which is responsible for the check. 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.

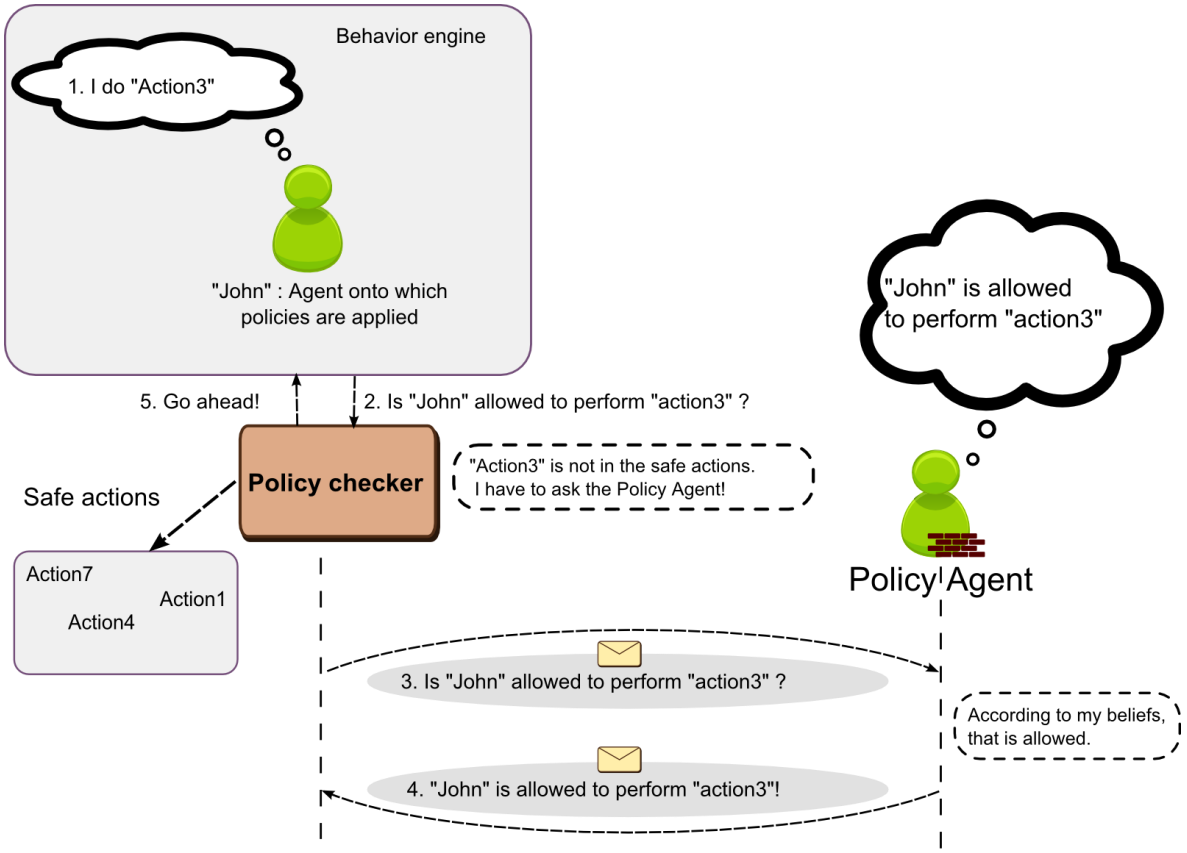

Figure 3: Policy check stages

Next, only if the first check fails, the parameters from the commitment statement are transformed into a request to the policy agent. This request also contains the name of the action and the name of the agent performing the request. This query message is then 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.

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.

UBIWARE DF Agent (UDF)

Ubiware directory facilitator (UDF) is an agent that replaces the original directory facilitator (DF) provided by Jade platform. Jade directory facilitator is capable of mapping agent names to agent capabilities. Each agent was able to register itself with a capability that it was providing. The limitation of Jade DF was a) the fact that the service description was not semantic and b) the directory facilitator had to be asked for updates in the registry i.e. a pull service. In this traditional version of directory facilitator, the agent can register a service, unregister it and it can also search for services provided by other agents.
We improved this original idea of directory facilitator and the result is Ubiware directory facilitator, which is based on semantic technology and is more modular than the original version. In order to understand UDF, it is necessary to understand the new way of defining services. We understand a service as a coherent set of functionalities. As an example we can take "Facebook service". Let's say that by definition this service should allow you to access your Facebook account and all data associated with it. Every service consists of several functions. In case of Facebook service, there is a functionality that allows you to authenticate yourself, receive a list of your friends and get detailed information about your friend. If an agent provides Facebook service, it must provide all functionalities associated with it. Therefore this agent must be capable of handling any request involving user authentication, request for friends list and request for friend's details. In other words it must have three handlers – one for each functionality[1]. Therefore we can say that a service is defined by a set of handlers. The meaning of this kind of definition is that every agent willing to provide this service has to implement those handlers. The relationship between a handler and a service can be seen in Figure 4.

Figure 4: The relationship between a handler and a service

In UDF we also extended the register, unregister and search actions of the old directory facilitator. As we mentioned before, a service is defined by handlers. UDF stores definition of each service. In other words, it stores a map of handlers and services. An example can be seen in Figure 5.

[1] It is possible that an agent implements one functionality with several handlers, however in order to keep this example simple, we assume that each functionality has only one handler associated with it.

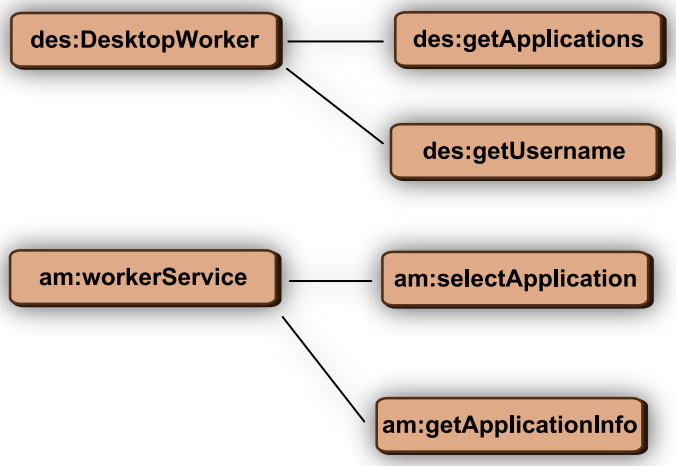

Figure 5: Internal representation of service definitions in UDF

Agents are registering only handlers, not services. Services are always defined by application programmer and are a part of application package description. In Figure 6 you can see agent A1 registering three handlers – handler H1, H2 and H3. UDF knows that service S1 is defined as composition of handlers H1 and H2. Therefore UDF knows that agent A1 provides service S1. Handler H3 is not mentioned in any service description. Therefore no other conclusions about A1's services are drawn. Now, if any agent asked who provides service S1, UDF would answer that agent A1 provides it. Naturally, unregistration is possible as well.

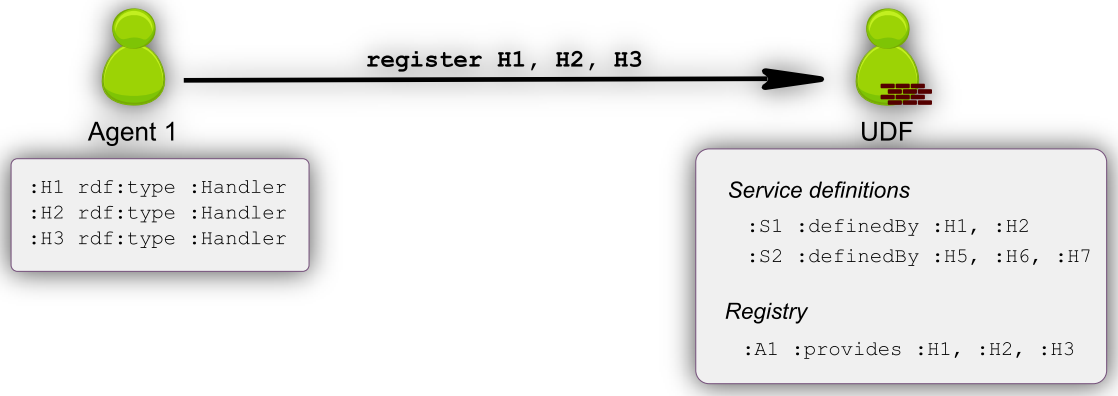

Figure 6: An agent registering its handlers

Another action related to UDF is service search. In the previous version of DF, the agent had to ask explicitly if there is an agent that provides a certain service. In our new implementation, the agent subscribes to a certain service notification and UDF sends updates whenever a new agent registers or an old agent unregisters. For example, in the old system, if some asking agent wanted to be sure that it has an up to date list of Facebook agents, it had to constantly ask DF for Facebook service providers. Every time an answer was received, the list of Facebook agents from DF was compared to the previous list of Facebook agents and whenever there was a new agent on the list, the asking agent knew that there is a change. In the new implementation, the asking agent just subscribes at UDF with information which service updates it wants to receive and every time there is change, asking agent will be notified. The schema for better understanding can be found in Figure 7.

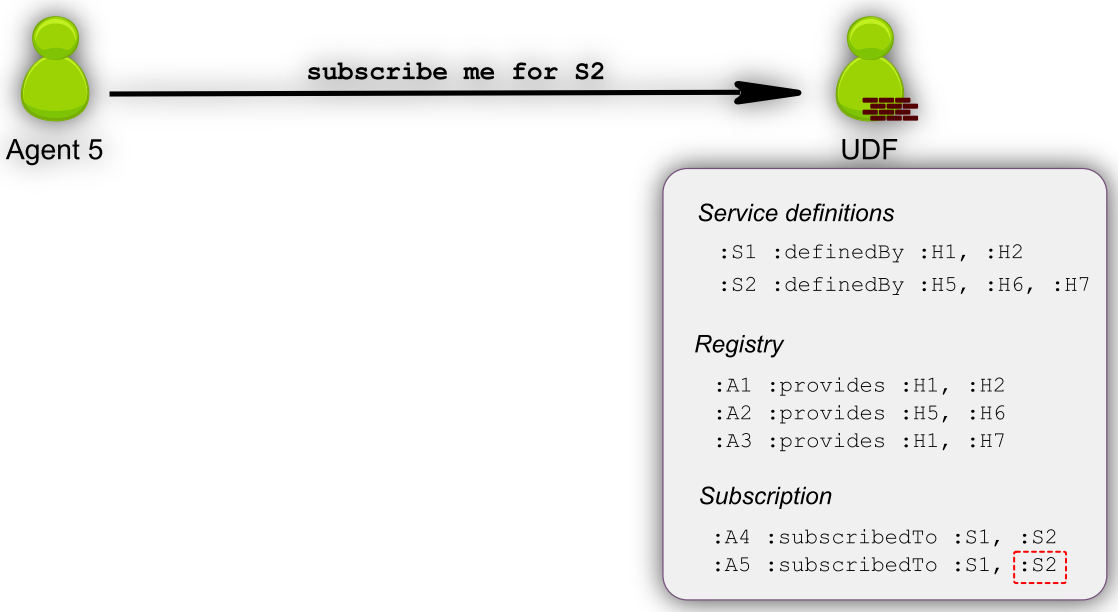

Figure 7: An agent subscribing for service updates

The UDF also provides services to only find out which agents provide a certain service and are working for a certain personal user agent. This way it is possible to compose services with components belonging to a single user.

Web Interface Agent (WIA)

Web Interface Agent (WIA) is responsible for communication over HTTP with the outside world. The communication is arranged by using an embedded web server. The web server 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 worker 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 sessions related to platform-wide, single-sign-on mechanism for external users. the single sign-on is handled with the DoLogin and DoLogout servlets which the WIA has deployed by default.

The 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. It then deploys the login and logout servlet. When the server is up and running, WIA deploys the default 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. From then on, WIA accepts new web applications for deployment from the application manager infrastructural agent.

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. More information about working with the platform from a web perspective can be found in the *Application user guide*.

How to work with WIA

This document describes the Ubiware platform from a web application developer point of view. The interaction which the web application developer has is on purpose very limited. All access to the platform is done trough an object of the type Ubiware.infrastructure.web.PlatformInterface.

To obtain a reference to this object, the web application has to get the platform interface from the request attributes. This can for example with the following code:

```
PlatformInterface platforminterface = (PlatformInterface) request.getAttribute(PlatformInterface.PLATFORM_INTERFACE_KEY);
```

An object of the PlatformInterface type has the following methods:

- boolean isLoggedIn();
  This method indicates to whether this application is in use by a user which is logged in to the platform.
- AgentWrapper getWorkerAgent() throws NoSuchWorkerException;
  This method provides an AgentWrapper to the worker agent which has all the handlers required by the service definition of this web application. The worker agent is only available if a user is logged in since it is user specific. This method throws a NoSuchWorkerException if there is no worker agent found.
- InfraAgentWrapperProvider getAIA();
  This method provides an object from which references to Application Infrastructure Agents can be obtained. by calling its getAIAWrapper method which will return an object of type AgentWrapper if this agent is accessible for this web application. If the agent is not accessible for this webb application, then a NoSuchAIAException is thrown.

The agentWrapper class provides the following two methods which can be used by the web application.

- String syncAction(Resource, ActionArgumentContainer)
  Sends a request to the agent behind the wrapper and return the answer as a String. The arguments to this method are a resource being the identifier of the action and an ActionArgumentContainer containing the arguments to the handler.
- MessageContent subscribe(Resource, ActionArgumentContainer, UpdateHandler)
  This method can be used to implement a push service. The web application subscribes to the agent and processes all messages (updates) which the agent sends. Subscribing happens similar to the syncAction with as an extra parameter the UpdateHandler. The first messages which the agent sends back is the return value of the method and will not be send to the handler.
  The UpdateHandler has the following interface:
  - abstract void handleUpdate(MessageContent)
    The method which the web application developer must override to handle the updates sent by the agent.
  - start()
    must be called when the web app is ready to start receiving updates. The UpdateHandler will buffer the updates till this message is called.
  - MessageContent unsubscribe(Resource, ActionArgumentContainer)
    This method must be called by the web app when no updates are needed any longer from the agent. The contract is the same as syncAction.

Package Manager Agent (PMA)

Package manager agent is the agent that is responsible for deployment of UBI packages. The agent accepts a package and unpacks it to a temporary folder. Then it goes through all package components (scripts, RABs, WAR files) and handles them in one-by-one fashion. The whole process of deployment can be summarized in 6 steps (Figure 8).

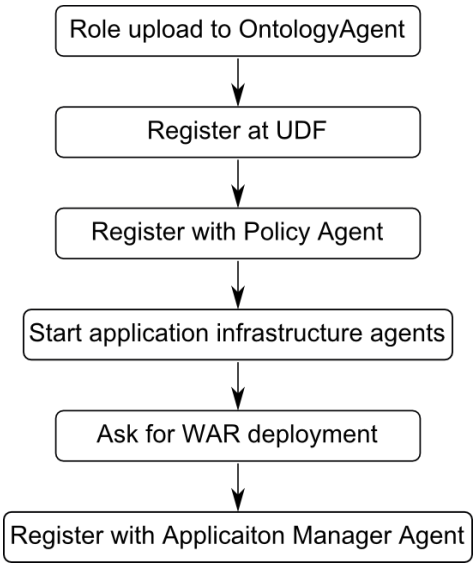

Figure 8: User manager agent and communication that it is involved in

Firstly, all roles specified in the package are read and uploaded to Ontology agent. They will be used later on for application infrastructure agent startup and worker agent startup. Secondly, the application has to register all service definitions if there are any provided. Moreover, all policies have to be registered with Policy agent. Only after that application infrastructure agents can start. Next step is to ask Web interface agent to deploy a WAR file so that the web interface is available. Lastly, the application is registered at Application manager agent and can be selected by users.

Ontology Agent (OA)

Ontology agent is responsible for ontology and agent role management. The agent has access to two repositories – ontology repository and role repository. In general there are two ways to provide a script to the agent. One way is to directly provide the path to the script file. This however cannot be used in case the agent is running in a different container than the container where the script is located. The second way is load the script as a role.

Role is an SAPL script that corresponds to an organizational role. An example where roles can be used is for agents willing to act as a message handler. They will load a role called "action". This role is downloaded from the ontology agent. This can be done even if the agent is in a different container than the ontology agent, because the role (script) is being sent as an ACL message. Some agents are capable of starting only by loading a role script

Ontology agent is serving not only as role provider, but also as role "acceptor". It is possible to contact Ontology agent and ask for role upload. This can for instance be done by Package manager agent. This step will be explained in the section related to Package manager agent. The position of Ontology agent within the platform can be seen in Figure 9.

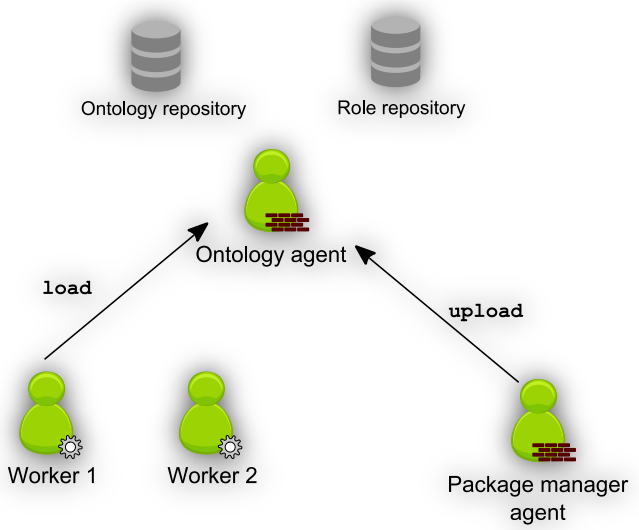

Figure 9: The position of Ontology agent within the platform

User Manager Agent (UMA)

The most important task of User manager agent (UMA) is to act as user registry. For every user it remembers his/her username and corresponding personal user agent name and more. UMA is capable of answering several kinds of requests (Figure 10). Firstly, it is the user registration request. It can come only from worker agent of the administrator, because only the administrator has the right to add new users to the platform. Secondly, User manager agent can provide list of registered user. This question can be asked only by administrator's user manager worker agent.

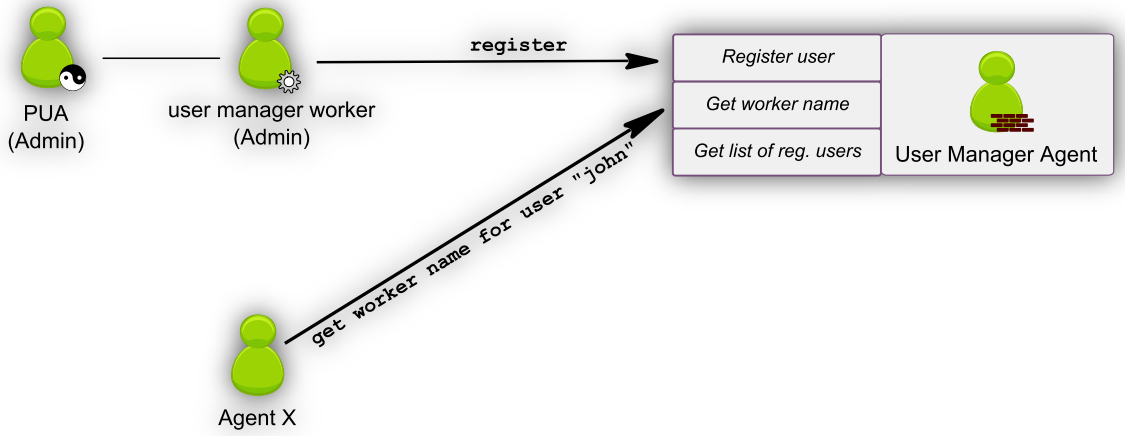

Figure 10: User manager agent and communication that it is involved in

It should be clarified that the user manager agent is working as an application infrastructure agent for the user management application. The interaction to the agent happens trough an application worker agent.

Application manager agent (AMA) acts as application registry. Every application deployed to Ubiware platform has to be registered at AMA. The information about application description can be seen in Figure Figure 11.

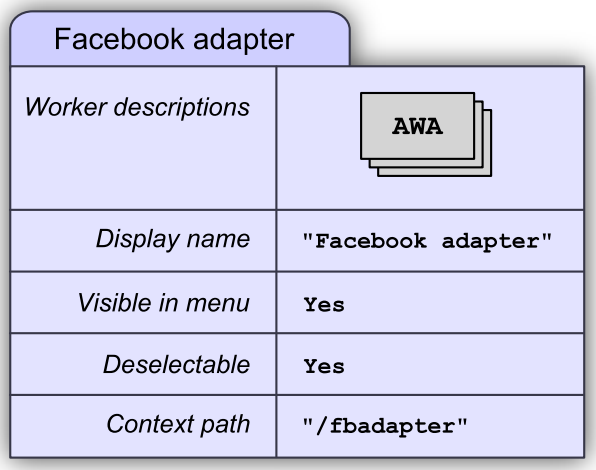

Figure 11: An application description at AMA

Apart from acting as a registry, AMA is also capable of answering several requests. The most important is that it can accept a request to register an application. The request must contain application name, context path, worker agent definitions, etc. Application name is the name that the user sees in the menu. Context path is the path that will form a URL, where the application can be reached. Worker agent definition describes which roles an agent has to load if it wants to act as a worker agent of this application. This is important in selection process.
Also in this case the AWA is AIA for the application selection application.

Root Agent (root)

The root agent is not an infrastructure agent per se. It 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 for users. 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 is then also able to create other users of the system.