UBIWARE Deliverable D3.4:

UBIWARE Platform Prototype v.3.1

December, 2010

Date	December 20, 2010
Document type	Report
Dissemination Level	UBIWARE project consortium
Contact Author	Vagan Terziyan
Co-Authors	Michal Nagy, Michael Cochez, Viljo Pilli-Sihvola, Joonas Kesäniemi, Oleksiy Khriyenko
Work component	UBIWARE Platform
Deliverable Code	D3.4
Deliverable Owner	IOG, JYU
Deliverable Status	Mandatory, Internal
Intellectual Property Rights	Unaffected

Table of Contents

Introduction ... 3
1 New agent classification ... 4
  1.1 Platform infrastructure agent ... 4
  1.2 Application infrastructure agent ... 4
  1.3 Personal user agent ... 4
  1.4 Personal user worker agent ... 5
2 New agent platform infrastructure ... 6
  2.1 Policy agent ... 6
  2.2 Ontology agent ... 6
  2.3 Ubiware directory facilitator ... 7
  2.4 Web interface agent ... 10
  2.5 Application manager agent ... 10
  2.6 User manager agent ... 10
  2.7 Package manager agent ... 11
3 New platform infrastructure web applications ... 13
  3.1 Desktop ... 13
  3.2 Application manager ... 14
  3.3 User manager ... 14
4 New tools for developers ... 15
  4.1 SAPL builder ... 15
  4.2 QueryMessageSenderBehavior ... 16
  4.3 Persistency subsystem ... 17
5 Conclusion ... 20
Appendix A: Glossary ... 21

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.

UBIWARE deliverable D3.3 presented the integrated development results from all the work packages, i.e. the current state of the UBIWARE 3.0 platform prototype. Current deliverable D3.4 follows up and brings a platform update – UBIWARE 3.1.

New version of the platform went through significant changes. The most important of them is the new architecture that follows cloud computing paradigm. Other changes include performance improvements and new features. This report can be understood as a “changelog” between platform version 3.0 and 3.1.

Firstly, we bring a new agent classification system that simplifies the management and policy control of agents. In the new version of the platform there are four types (classes) of agents. Each agent class is being started differently, has different rights (controlled by policies) and different tasks associated with it. Section 1 describes new agent classification in more detail.

Secondly, the new agent classification enabled new platform architecture. The platform infrastructure reported in deliverable D3.3 has been extended and improved. Now, the platform contains seven platform infrastructure agents, each supporting one type of platform functionality. More information about the new architecture can be found in Section 2. Section 2.7 contains information about new packaging system for UBIWARE applications.

Moreover, we developed two new web applications supporting the platform administration. First application is used for user management and the second one is used for management of UBIWARE applications. Section 3 contains more information about these applications.

Finally, we create several new Reusable Atomic Behaviors (RABs) and several libraries that simplify UBIWARE application development. They are described in section 4.

Sections 1, 2 and 3 require only basic technical knowledge. Section 4 is more technical and contains more information related to development of UBIWARE applications.

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

1 New agent classification

With the upcoming shift of Ubiware toward cloud computing infrastructure, there was necessity to revise the agent classification schema. In previous versions of Ubiware, there was just one class of agent. However, in cloud computing architecture, some element are performing functions related to platform runtime and others are related to specific application run by some user. For this reason we create a new classification schema. The schema can be seen in Figure 1. Our new classification involves the following agents classes: platform infrastructure agent, application infrastructure agent, personal user agent (PUA) and personal user worker agent (shortly called worker agent). The following subsections will deal with each agent in more detail.

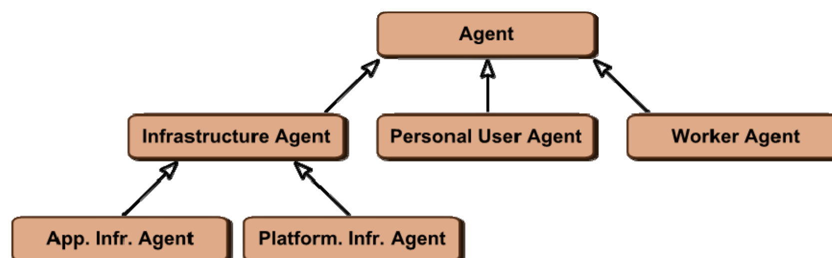

Figure 1: Agent classification schema

1.1 Platform infrastructure agent

Platform infrastructure agent (PIA) is an agent whose main function is to provide generic platform-related services to other agents. There are exactly seven agents of this type and they are available to other agents from platform startup till platform shutdown. Each of these agents has a predefined role in the platform. You can find more information about the agents in section 2.

1.2 Application infrastructure agent

Application infrastructure agent (AIA) is an agent that works for a particular application running on top of Ubiware. One application can have several AIAs, whose role is to provide services to all users who selected ("installed") this application. For example an application related to flower delivery might have one agent who is responsible for delivery scheduling. This agent could be implemented as application infrastructure agent.

1.3 Personal user agent

Every user of the platform has exactly one personal user agent (PUA). This agent is a representative of the user in the platform. If any agent wishes to communicate with any user, the agent will contact PUA of that particular user. Therefore PUA can be understood as an interface between the human user and any other agent on the platform. Every personal user agent can have several "helper agents" who are called personal user worker agents or shortly just workers.

1.4 Personal user worker agent

Personal user worker agent (shortly worker) is an agent bound to a certain application and working for a certain user of the platform. A worker agent is PUA's “minion” and creates a connection between the user and an application that the user wishes to utilize. If the user selected[1] N applications, there will be exactly N worker agents associated with user's PUA. The relationship between an application and a worker can be seen in Figure 2. User A decided to select applications App1, App2 and App4. User B decided to select application App1 and App3. Therefore PUA of user A has three workers and PUA of user B has only two workers.

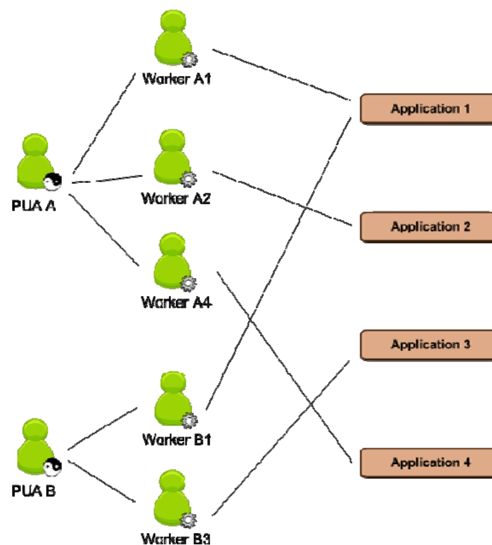

Figure 2: Relationship between PUA and its worker agents

[1] Selection is explained in more detail in section 2. At this point, it is sufficient to say that selection of an application is something similar to installation of an application on a normal operating system. After the selection, the user can start and utilize the selected application.

2 New agent platform infrastructure

The new version of platform must deal with cloud applications and therefore there was a need to introduce new infrastructure that is capable of fulfilling this need. The new architecture contains several agents, each providing a service to support the runtime of cloud applications and platform itself.

We also introduce new terms like application deployment and application selection. Starting from Ubiware 3.1 application can be deployed to the platform using Ubiware packages with *.ubi suffix. A UBI package contains all information about the application, its agents, its web interface, etc. New packaging system resembles packaging system of other mature architectures such as Java application servers (WAR and EAR packages).
Since Ubiware is shifting to cloud computing architecture, we can assume that there might be hundreds of different applications running in the cloud. It is very unlikely that every user of the platform would like to utilize every application available on the platform. Moreover, there might be cases when it is not reasonable for a certain user access a certain application. An example could be security issues. Therefore the user first has to select the application from the list of deployed applications and only after that the application is available to him/her. Logically, only deployed application can be selected. Applications may also be deselected.

2.1 Policy agent

Policy agent was described in detail in deliverable D3.3. In Ubiware 3.1 it has the same function as in Ubiware 3.0. Firstly, it maintains a list of policies related to different agent groups. Secondly, every time an agent wants to execute a Reusable Atomic Behavior (RAB), policy agent checks if the agent has the right to do execute the RAB based on agent's affiliation to a group and the list of policies related to this group.

An important feature is also the ability of Policy agent to register and unregister new group policies. For security reasons, this is allowed only by Package manager agent.

2.2 Ontology agent

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 are used is the message handling ability reported in deliverable D3.3. Each agent willing to act as a message handler 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. They cannot be started by providing a direct path to a SAPL file. An example of such agent is the worker agent.

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. It can only 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 3.

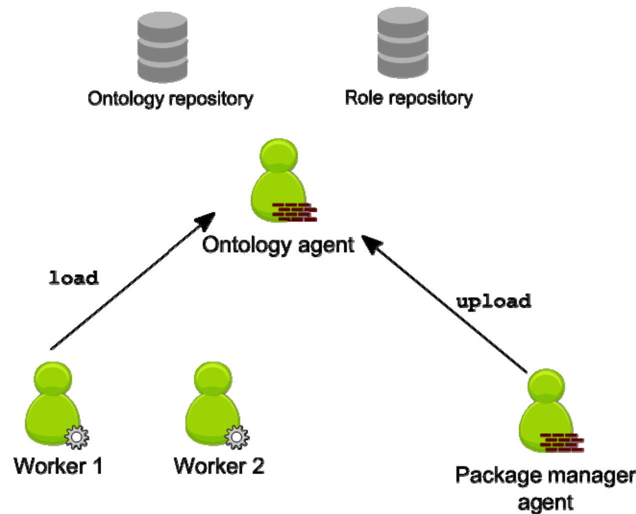

Figure 3: The position of Ontology agent within the platform

2.3 Ubiware directory facilitator

Ubiware directory facilitator (UDF) is an agent that replaces the original directory facilitator (DF) provided by Jade platform. Jade directory facilitator was 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 the fact that the service description was not semantic. It was just a simple string. 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 the original idea of directory facilitator and the result is Ubiware directory facilitator, which is already 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[2]. 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.

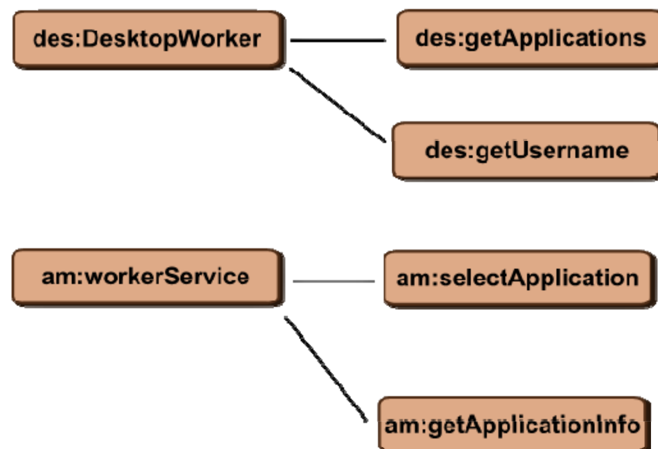

Figure 5: Internal representation of service definitions in UDF

Agents are registering only handlers, not services. Services are always defined by application programmer and are a part of application package description[3]. 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.

[2] 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.
[3] Application packages will be described in section 2.7.

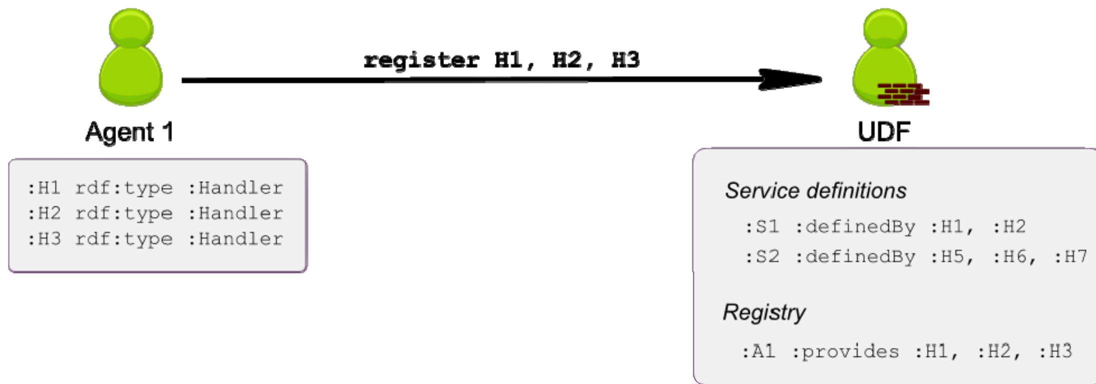

Figure 6: An agent registering its handlers.

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.

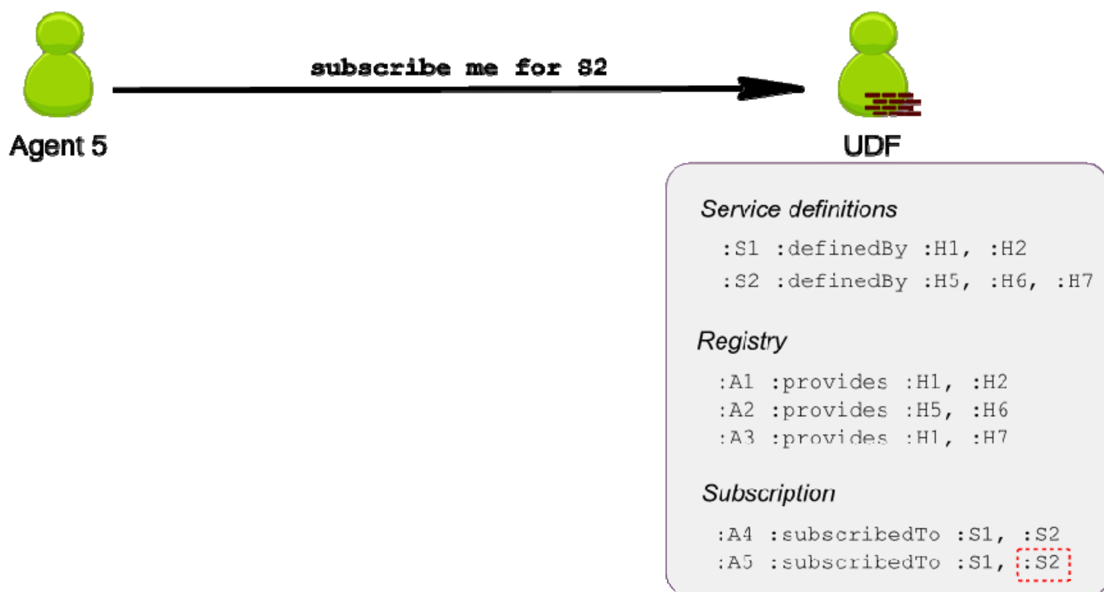

Figure 7: An agent subscribing for service updates.

2.4 Web interface agent

Web Interface Agent (WIA) is responsible for communication over HTTP with the outside world. The communication is arranged by using 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 tickets related to platform-wide, single-sign-on mechanism, used in the Desktop web application (see section 3.1).

2.5 Application manager 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 8.

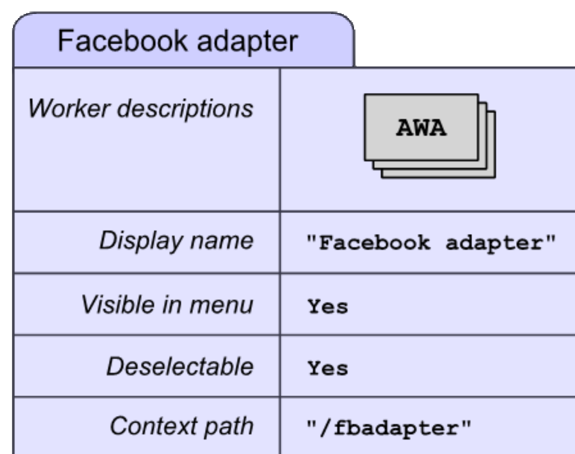

Figure 8: An application description at AMA.

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.

2.6 User manager agent

The most important task of User manager agent (UMA) is to act as user registry. For every user it remembers his/her username, password, corresponding personal user agent name and more. UMA is capable of answering several kinds of requests (Figure 9). 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, UMA is also able to answer the question about the worker agent name that is working for a particular user in particular application. Moreover, User manager agent can provide list of registered users if it asked for it. This question can be asked only by administrator's user manager worker agent.

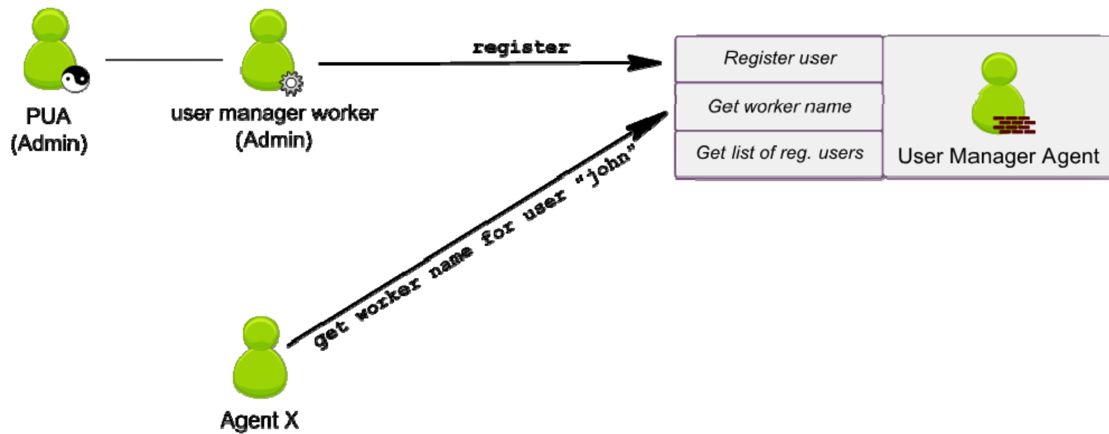

Figure 9: User manager agent and communication that it is involved in.

2.7 Package manager agent

As mentioned earlier, Ubiware 3.1 introduces new way of deploying applications. In the previous version of the platform, the developer had to deploy every piece of code manually. Now we provide an automated deployment mechanism built on top of application package called UBI package (UBIWARE application package). UBI package is essentially a compilation of several files (SAPL script, WAR web applications, etc.) that is packed into one zip file with extension *.ubi.

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 10).

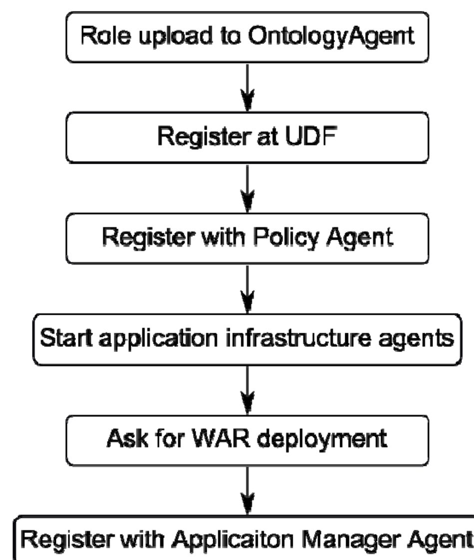

Figure 10: User manager agent and communication that it is involved in.

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.

3 New platform infrastructure web applications

The shift to cloud computing architecture made us introduce new web applications related to basic function of the platform. Since the platform works with users, there has to be a way to manage user accounts. For this purpose we developed User manager application. Also, the platform needs some application management tool where the user can select applications and the administrator can deploy, undeploy and modify applications. This tool is called Application manager application. There is also a need to

3.1 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 11 shows the desktop with the application menu.

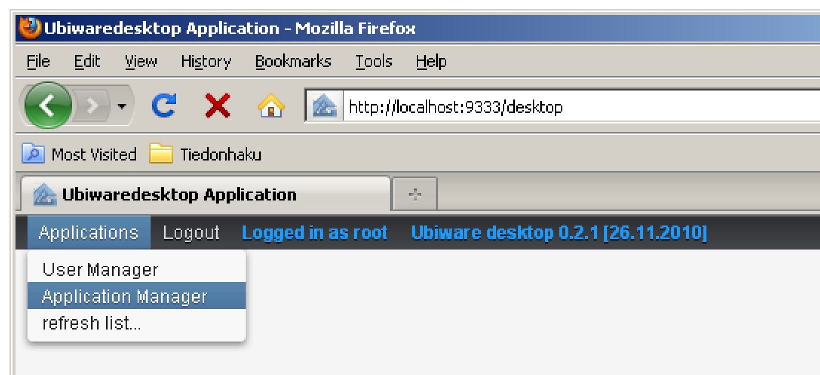

Figure 11: Ubiware desktop web application.

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.

3.2 Application manager

Application manager is a tool available to every user of the platform. Using this tool the user can select and deselect applications (Figure 12). On the left, there is a list of applications that are deployed, but the user didn't select them yet. In other words, these are the applications that are available, but the user doesn't want/need to use them. In the upper right corner you can see a list of applications that are selected, thus available in the application menu of Desktop application. The user can move applications from one list to the other. By doing so he/she selects and deselects applications. In the lower right corner, you can see a list of applications that are "core" applications and are available by default to every user. The user cannot deselect them.

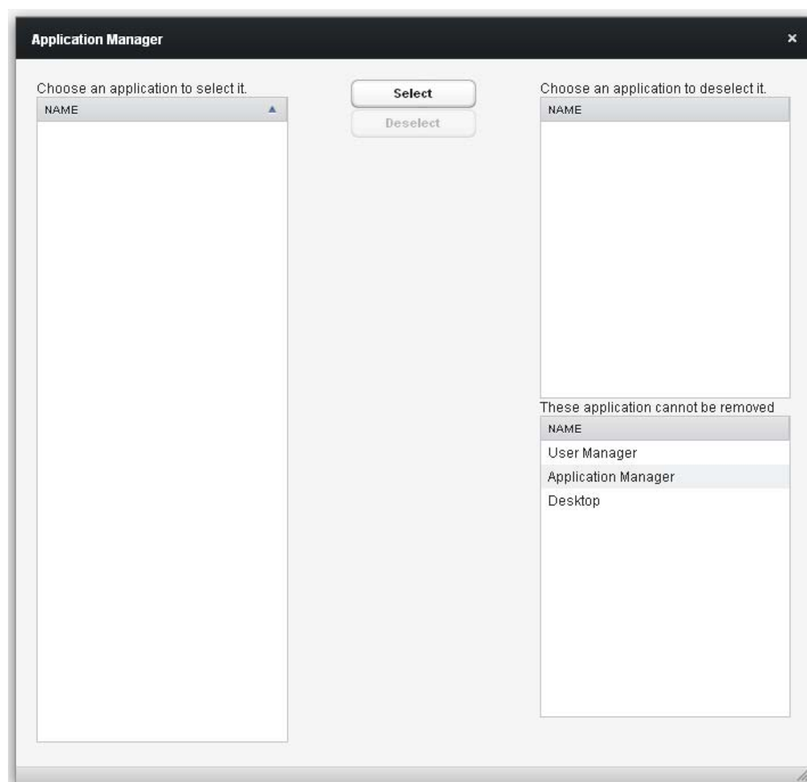

Figure 12: Application manager agent.

3.3 User manager

User manager application is an administrative tool responsible available only to the administrator. The administrator can add, delete and modify users.

4 New tools for developers

This section describes new Reusable Atomic Behaviors (RABs) and tools that simplify development of UBIWARE applications. In order to understand this section you should be familiar with basic UBIWARE and Java application development.

4.1 SAPL builder

SAPL builder is a Java library that simpliefies creation of SAPL code from within a Java class. Prior to SAPL builder, the developer had to write SAPL code by concatenating strings, e.g. using standard Java class `StringBuider` or `StringBuffer`. This approach was error prone. In our experience the developers made two kinds of mistakes. First type of mistake was that they used wrong URIs in SAPL expressions. URIs could have been wrong for two reasons – they were invalid URIs and/or they contained typos (e.g. writing "peple" instead of "people"). Second type of mistakes was syntactic mistakes. Developers forgot to put coma, dot or properly end the container.

SAPL builder solves problems of invalid URIs and syntactic problems, which in our experience collectively stand behind majority of problems and bugs. However, SAPL builder cannot eliminate typos. In order to minimize those, we also introduce new supporting classes that will be explained later. The biggest improvement is that abovementioned mistakes (except for typos) can be checked in compilation time, thus saving significant amount of time otherwise spent on bug fixes.

SAPL builder library consists of several classes, all of them residing in `ubiware.util.saplbuilder` package. Unless specified otherwise, every class mentioned in this paragraph belongs to this package. Therefore we will not specify the full class names. The most important class is `SaplDocument`. An instance of this class virtually represents a document that the developer is going to write. In the beginning the document is empty. The developer can utilize several member methods of this class in order to insert SAPL statements into the document. Among all, the most important one is method `addStatement(Subject subject, Predicate predicate, Object object)`. Three arguments of this method represent subject, predicate and object of the triple that the developer wants to add to the document. Subject and object can either be a container or a resource. Predicate must be a resource. A resource is represented by class `Resource` and a container by class `Container`. A container can be treated as another SAPL document. It even has the same method `addStatement` and it is used in the same way as in case of `SaplDocument` class. The relationship between these classes and interfaces `Subject`, `Predicate` and `Object` can be seen in Figure 13.

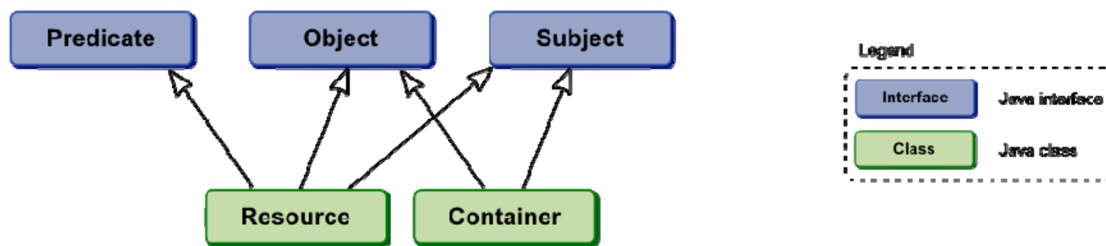

Figure 13: Class hierarchy of SAPL builder classes.

So far the described solution solves syntactic problems. In order to solve problems with invalid URIs, we force the developer use `java.net.URI` class if he/she wants to declare one. The reason is that if a URI is invalid, the class itself will throw an exception. Second step to avoid invalid URIs is to create a set of constants, each representing a frequently used resource. Instead of writing the resource name every time (and rising the risk of making a mistake), the developer can utilize these predefined resource constants. We already provided a set of most frequently used constants. This list can be found in package `ubiware.ontology`. Each SAPL prefix is represented by one Java class containing resource constants. An example of SAPL builder use can be seen in Figure 14.

```
IDGenerator idGen = new IDGenerator("pmaUBIPackage");
SaplDocument sd = new SaplDocument();
sd.addStatement(actionID, rdf.type, pma.UBIDeploymentAction);
sd.addStatement(actionID, pma.pathToUBIFile, new
                          Literal("someString"));
sd.addStatement(actionID, pma.deploymentStatus, pma.UBIIdentified);
String saplString = sd.generateSapl();
```

Figure 14: An example of SAPL builder use in Package manger agent.

4.2 QueryMessageSenderBehavior

This Reusable Atomic Behavior (RAB) is located in `ubiware.shared` package and its main task is to simplify querying and consequently sending message to some agent. Based on our practical experience, many times there is a need to query a certain structure and almost literally copy a part of this belief structure and send it as a message to another agent. This is a typical scenario in case of all agents containing some sort of registry. Among platform infrastructure agents, this is true for User manager agent, Application manager agent and many more. We will try to explain this RAB's function on an example.

Imagine that the agent has a container containing information about some people (Figure 15). The agent remembers name, surname and age for each of each person. We would like to send the content of this container to some agent asking for the list of people. The problem is that the receiving agent wants to receive this list in another form than the form in which the list is stored in sending agent's beliefs. Therefore we need to do a transformation. Without `QueryMessageSenderBehavior` we would have to make a temporary container and with a rule copy the beliefs from the original container in the original form into the destination temporary container in the new form. Then we would send a message containing the content of the temporary container. After that the temporary container would be deleted. This is very complicated and error prone.

```
:peopleRegistry :contains {
:p1 rdf:type fam:Person
  ; fam:hasFirstName "John"
  ; fam:hasFirstSurname "Doe"
  ; fam:hasAge "25" .

:p2 rdf:type fam:Person
  ; fam:hasFirstName "Peter"
  ; fam:hasFirstSurname "Smith"
  ; fam:hasAge "33" .

:p3 rdf:type fam:Person
  ; fam:hasFirstName "Mary"
  ; fam:hasFirstSurname
"Hemingway"
  ; fam:hasAge "32" .
}
```

Figure 15: An example of a people registry.

With use of `QueryMessageSenderBehavior` we are able to query the people container in its original form, transform it and send to the receiver agent. This all can be done in one RAB call. An example of such call can be seen in Figure 16.

```
{
        :peopleRegistry :contains  ?peopleContainer
} => {
        { sapl:I sapl:do java:ubiware.shared.QueryMessageSenderBehavior } sapl:configuredAs {
                p:context sapl:is ?peopleContainer .
                p:query sapl:is {?person rdf:type fam:Person
                                        ; fam:hasFirstName ?fn
                                        ; fam:hasSurname ?sn
                                        ; fam:hasAge ?age
                } .
                p:sendPattern sapl:is {
                        ?person rdf:type computer:User
                        ; computer:hasUsername "?fn?sn "
                        ; computer:hasFirstName "?fn"
                        ; computer:hasSurname "?sn "
                        ; computer:hasAge "?age "
                } .
                p:receiver sapl:is ?sender .
                p:conversationID sapl:is ?id .
                p:performative sapl:is "inform" .
        }
}
```

Figure 16: An example of QueryMessageSenderBehavior use.

4.3 Persistency subsystem

Persistency of agent beliefs brings many benefits to the developer. The typical behavior of the agent is that if it starts, it loads all the beliefs from the SAPL script that was provided. The problem is described in Figure 17. We marked beliefs from the script as B_0.

These are the beliefs that the agent loads. After several iterations the beliefs will naturally change as the agent is performing its task. Therefore after n iterations beliefs B_n might be different than the initial beliefs. If the agent dies or it's terminated at this point, beliefs B_n will be lost, because if the agent starts again, it will again receive B_0, not B_n. In some applications this is the desired behavior, but in some applications it would be beneficial to store the whole set of beliefs B_n or its part. Then, upon the next agent's startup, the agent will not receive B_0, but B_n or a part of these beliefs.

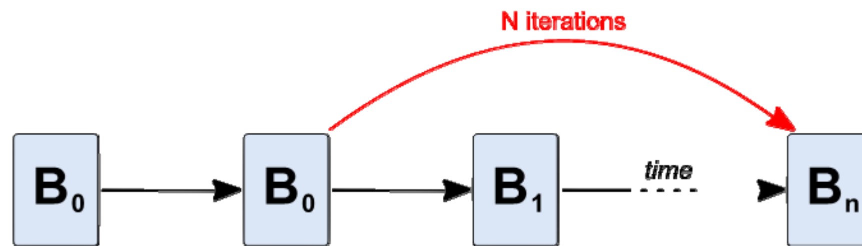

Figure 17: Evolution of agent's beliefs without persistency.

Persistency subsystem allows the developer to turn on persistency for either the whole set of agent's beliefs or just for a specific container(s). The persistency has to be controlled by the developer. The underlying storage system for persistent beliefs is based on a relational database. The developer can mark a container as persistent by declaring it to be a resource of type `per:PersistentContainer`. Based on the persistency ontology, every resource of this type has several properties that are used by persistency subsystem to determine if load or save action from/into the database should be performed. An example of such container is shown in Figure 18.

```
@prefix per: <http://www.ubiware.jyu.fi/persistency#> .
:myPersistenContainer rdf:type per:PersistentContainer
  ; per:hasID "myContainer123"
  ; per:hasState per:clean
  ; per:hasContent {
        :p1 rdf:type fam:Person
          ; fam:hasFirstName "John"
          ; fam:hasFirstSurname "Doe"
          ; fam:hasAge "25" .
        :p2 rdf:type fam:Person
          ; fam:hasFirstName "Peter"
          ; fam:hasFirstSurname "Smith"
          ; fam:hasAge "33" .
        :p3 rdf:type fam:Person
          ; fam:hasFirstName "Mary"
          ; fam:hasFirstSurname "Hemingway"
          ; fam:hasAge "32" .
  }
```

Figure 18: An example of a persistent container.

As you can see in Figure 18, the container has an ID. This ID is used as a unique identifier in the persistent storage. You can also see that the container is in state `per:clean`. This means that the developer considers this container to be fully

synchronized with the information in the persistent storage (relational database). In case the developer wants to store the content of this container to the persistent storage, the only thing that he/she needs to do is to mark this container as `per:dirty` and the persistence subsystem will automatically take care of it. The subsystem is permanently checking if there is a dirty container. If it finds one, it will store its data to the persistent storage and mark it clean. Apart from these two states, there is a state called `per:initial`. While two previous states were mostly used when the agent is running, the later one is used in the agent script (beliefs B_0 in Figure 17). In that case the container does not have to have a value for per:hasContent. A container in the initial state makes the persistency subsystem load its content from the persistent storage. In simple words, the meaning of per:initial is "please load my content from the persistent storage". The transition between different statuses can be seen in Figure 19.

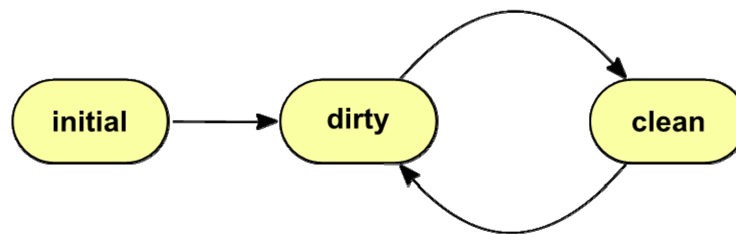

Figure 19: The transition between different persistency statuses.

5 Conclusion

New version of the platform underwent several changes. We created a new agent classification schema that simplifies policy management. Platform architecture was extended with several new platform infrastructure agents. Already existing platform infrastructure agents were modified as well. The platform supports application deployment in form of UBI packages through Package manager agent. User management agent is responsible for user registration. Similar to desktop operating systems, every user has an account, his/her own list of applications and data. Users access the platform through Desktop application which follows the idea of web-based operating systems. User manager agent can be accessed through “User management” application which is available only to the administrator. Application manager agent is responsible for application selection and deselection. Every user has access to “Application manager” web application where he/she can customize the list of available applications. Last of significant platform infrastructure agent changes includes a completely new directory facilitator called Ubiware Directory Facilitator. The underlying philosophy has changed accordingly. The rest of the platform infrastructure agents were changed as well to form a coherent platform.

Appendix A: Glossary

AIA	Application Infrastructure Agent
AMA	Application Manager Agent
PIA	Platform Infrastructure Agent
PUA	Personal User Agent
RAB	Reusable Atomic Behavior
UDF	Ubiware Directory Facilitator
UMA	User Manager Agent