WatERP

Water Enhanced Resource Planning

“Where water supply meets demand”

GA number: 318603

WP 2: External System Integration

2.6: Multi-agent Architecture Prototypes

V1.0 29-05-2015

www.waterp-fp7.eu

Document Information

Project Number	318603	**Acronym**	WatERP
Full title	Water Enhanced Resource Planning “Where water supply meets demand”		
Project URL	**http://www.waterp-fp7.eu**		
Project officer	Grazyna Wojcieszko		

Deliverable	**Number**	2.6	**Title**	Multi-agent Architecture Prototypes
Work Package	**Number**	2	**Title**	External System Integration

Date of delivery	**Contractual**	M30	**Actual**	M32
Nature	Prototype ■ Report □ Dissemination □ Other □			
Dissemination Level	Public ■ Consortium □			

Responsible Author	Gabriel Anzaldi	**Email**	ganzaldi@bdigital.org
Partner	BDigital	**Phone**	+34 93 553 45 40

Abstract (for dissemination)	Deliverable 2.6 (D2.6) is focused on presenting Multi-agent architecture with a coarse-grained, describing its components (modules and agents) and interactions between them. Basically, the architecture is divided in three layers that includes the interface layer aimed at defining the public interface whose goal is to be consumed by third parties, the multi agent system layer and their tools which enable the cooperation between agents facilitating its use by managers for Directory Facilitator and Agent Management and external access layer focused on accessing external components such as OGC® WPS, OGC® SOS and Sesame repositories to consume sensor information and processes in an standardised way. Moreover, multiple sequence diagrams are described to present interactions between multi-agent architecture components for more common operations performed by the multi-agent architecture. Finally, a description of the process to carry out the deployment of the architecture is described.
Key words	MAS, agents, directory facilitator, agent management system, yellow pages, white pages, owl-s

List of Acronyms

ACA – Agència Catalana de l'Aigua

ACD – Architectural Context Diagram

ADF – Agent Description File

AMS – Agent Management Service

BDI – Belief Desire Intention

CUAHSI – Consortium of Universities for the Advancement of Hydrologic Science

DF – Directory Facilitator

DMS – Demand Management System

DSS – Decision Support System

DX.X – Deliverable X.X

FOI – Features Of Interest

HMF – Hydro-Meteorological Forecast

HTTP – HyperText Transfer Protocol

JSON – JavaScript Object Notation

MAS – Multi Agent System

OGC® – Open Geospatial Consortium

OMP – Open Management Platform

OWL-S – Ontology Web Language

PIM – Pilot Integration Manager

RDF – Resource Description Framework

REST – REpresentational State Transfer

SOA – Service Oriented Architecture

SOAP – Simple Object Access Protocol

SOS – Sensor Observation Service

SPARQL – SPARQL Protocol and RDF Query Language

SSN – Semantic Sensor Network

SWEET – Semantic Web for Earth and Environmental Terminology

SWKA – StadtWerke KArlsruhe

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

UTF – Unicode Transformation Format

W3C – World Wide Web Consortium

WADL – Web Application Description Language

WAR – Web Application aRchive

WDW – Water Data Warehouse

WFS – OGC® Web Feature Service

WMS – OGC® Web Map Service

WP – Work Package

WPS – OGC® Web Processing Service

WSDL – Web Services Description Language

XML – eXtensible Markup Language

Executive Summary

Deliverable 2.6 (D2.6) is focused on presenting Multi-Agent System architecture with a coarse-grained describing its components and interactions between them. Basically, the architecture is divided in three layers that separate the public interface whose goal is to be consumed by third parties (interface layer), the multi agent system and the tools that facilitate its use such as managers for Directory Facilitator and Agent Management (MAS layer) and the layer to access external components such as OGC® WPS, OGC® SOS and Sesame repositories (external access layer).

Then, the document is initiated with an introduction that includes a description of main outcomes achieved in deliverables referring to agents (D7.2.1 and D7.2.2 “Implementation of MAS”, D2.4 “Agents Goal-Table and Condition Action Rules”). Then, a general view of the WatERP Multi Agent Architecture and how it fits into the project are also depicted. Later, a description of the components involved in the WatERP Multi Agent Architecture is introduced. Referring Interface layer, a complete description of the public interfaces is done by presenting invocation examples and their responses. The second layer, (MAS layer) is introduced by reviewing the agents that conforms the architecture and the tools used by them to manage agents (creation, destruction, search, etc.), services (creation, elimination, search, etc.) and process knowledge. Concerning the last layer (external layer), the created components to interact with external components (non-multi-agent components) are presented. Basically these tools refer to SOS Manager, WPS Manager and Water Ontology Manager. Moreover, multiple sequence diagram are described to present interactions between multi-agent architecture components for the more common operations performed by the multi-agent architecture. Finally, a description of the process to carry out the deployment of the architecture is described.

To understand this document the following deliverables have to be read.

Number	Title	Description
D2.3	Open Interface Specification	This document describes the analysis of the building blocks and the pilots' interfaces in order to understand the general open interface requirements for system integration and interoperability within the WatERP project. The guidelines to integrate a system in the WatERP framework are also described. It also includes the definition of the system integration road-map.
D2.4	Agents Goal-Table and Condition Action Rules	This document describes the implementation of the agent plans and goals. Moreover, the agent definition file (ADF) is depicted for the agents (Sesame agent, OGC® SOS agent, OGC® WPS agent, HF agent, DSS agent, DMS agent and Gateway agent). Also, the use of the OWL-S ontology to centralize the "*Process Knowledge*" of the WatERP MAS and its matching with the OGC® WPS schema is presented in this document.
D7.2.1	Implementation of MAS	This deliverable analyses the requirements to build a subset of the MAS subsystem that was to be implemented once the milestone MS3 "First vertical integration" was reached. Hence, agent types, agents, goals, beliefs and exchanges were identified during the document providing the necessary schemas to implement this part of the SOA-MAS architecture. Moreover, different studies about methodologies to develop multi-agent systems was studied concluding that MAS-CommonKADS methodology is the most suitable to be applied in WatERP MAS design.
D7.2.2	Implementation of MAS	This deliverable analyses the requirements to build a subset of the MAS subsystem that has to be implemented once the milestone MS5 "Second vertical integration" is reached. Hence, agent types, agents, goals, beliefs and exchanges are identified during the document providing the necessary schemas to implement this part of the SOA-MAS architecture.

Table of contents

1. INTRODUCTION 12

2. WATERP MULTI AGENT ARCHITECTURE 13

3. COMPONENTS 17

3.1 INTERFACE LAYER 18

3.1.1 Operational operations 18

3.1.1.1 Logical model operations 18

3.1.1.1 Observation operations 25

3.1.1.2 Process operations 28

3.1.2 Management operations 40

3.2 MAS LAYER 42

3.2.1 Agents 43

3.2.2 WatERP Process Knowledge Service 44

3.2.3 Directory Facilitator Service 47

3.2.4 Agent Management Service 50

3.3 EXTERNAL ACCESS LAYER 51

3.3.1 SOS Manager 52

3.3.2 WPS Manager 54

3.3.3 Water Ontology Manager 57

4. COMPONENT INTERACTION 59

4.1 COMPONENT INTERACTION FOR INTEGRATING A NEW OGC® WPS SERVER 60

4.2 COMPONENT INTERACTION FOR INTEGRATING A NEW OGC® SOS SERVER 61

4.3 COMPONENT INTERACTION FOR QUERYING AN OBSERVATION 62

4.4 COMPONENT INTERACTION FOR EXECUTING A SYNCHRONOUS PROCESS 63

4.5 COMPONENT INTERACTION FOR EXECUTING AN ASYNCHRONOUS PROCESS 65

4.6 COMPONENT INTERACTION FOR QUERYING A LOGICAL MODEL 66

5. DEPLOYMENT VIEWS 67

5.1 MULTI-AGENT ARCHITECTURE DEPLOYMENT 68

5.2 52º NORTH OGC® WPS PROCESSES DEPLOYMENT 69

6. CONCLUSIONS AND FUTURE WORK 71

7. APPENDIX I 71

List of figures

FIGURE 1: "ARCHITECTURAL CONTEXT DIAGRAM (ACD)" 13

FIGURE 2: "WATERP COMPONENT VIEW" 15

FIGURE 3: "MAS COMPONENT DIAGRAM" 16

FIGURE 4: "MULTI-AGENT ARCHITECTURE COMPONENT VIEW" 17

FIGURE 5: "WSDL SCHEMA FOR THE MANAGEMENT OPERATIONS" 41

FIGURE 6: "PROCESS KNOWLEDGE FACADE" 45

FIGURE 7: "OWL-S ONTOLOGY BASE (SOURCE: HTTP://WWW.W3.ORG/SUBMISSION/2004/SUBM-OWL-S-20041122)" 46

FIGURE 8: "SERVICEPROFILE ENTITY OF OWL-S (SOURCE: HTTP://WWW.W3.ORG/SUBMISSION/2004/SUBM-OWL-S-20041122)" 47

FIGURE 9: "DIRECTORY FACILITATOR MANAGER" 48

FIGURE 10: "AGENT MANAGEMENT SERVICE MANAGER" 50

FIGURE 11: "SOS MANAGER CLASS" 52

FIGURE 12: "WPS MANAGER CLASS" 55

FIGURE 13: "WATER ONTOLOGY MANAGER CLASS" 57

FIGURE 14: "MULTI-AGENT ARCHITECTURE COMPONENTS DIAGRAM" 60

FIGURE 15: "INTEGRATE A NEW OGC® WPS SERVER SEQUENCE DIAGRAM" 61

FIGURE 16: "INTEGRATE A NEW OGC® SOS SERVER SEQUENCE DIAGRAM" 62

FIGURE 17: "QUERY AN OBSERVATION SEQUENCE DIAGRAM" 63

FIGURE 18: "SYNCHRONOUS PROCESS EXECUTION SEQUENCE DIAGRAM" 64

FIGURE 19: "ASYNCHRONOUS PROCESS EXECUTION SEQUENCE DIAGRAM" 65

FIGURE 20: "QUERYING A LOGICAL MODEL SEQUENCE DIAGRAM" 67

FIGURE 21: "WATERP DEPLOYMENT VIEW" 68

FIGURE 22 "WPS WEB ADMINISTRATION PROVIDED BY 52º NORTH SERVER" 70

List of tables

TABLE 1: "DESCRIPTION OF THE '/LOGICALMODEL' OPERATION" 19

TABLE 2: "DESCRIPTION OF THE /LOGICALMODEL/{URI} OPERATION" 20

TABLE 3: "DESCRIPTION OF THE /LOGICALMODEL/{URI}/FULL OPERATION" 21

TABLE 4: "DESCRIPTION OF THE '/LOGICALMODEL/{URI}/SINK' OPERATION" 23

TABLE 5: "DESCRIPTION OF THE '/FEATUREOFINTEREST/{URI_LOGICAL_MODEL}/{URI_WATER_RESOURCE}' OPERATION" 24

TABLE 6: "DESCRIPTION OF THE '/OBSERVATION' OPERATION" 25

TABLE 7: "DESCRIPTION OF THE '/OBSERVATION/{URI_OBSERVATION}/DATAWINDOW' OPERATION" 26

TABLE 8: "DESCRIPTION OF THE '/OBSERVATION/{URI_OBSERVATION}' OPERATION" 27

TABLE 9: "DESCRIPTION OF THE '/PROCESS' OPERATION" 29

TABLE 10: "DESCRIPTION OF THE '/PROCESS/{URL_PROCESS}' OPERATION" 31

TABLE 11: "DESCRIPTION OF THE '/PROCESS/{URL_PROCESS}/SYNCHRONOUS' OPERATION" 34

TABLE 12: "DESCRIPTION OF THE '/PROCESS/{URL_PROCESS}/ASYNCHRONOUS' OPERATION" 36

TABLE 13: "DESCRIPTION OF THE OPERATION '/PROCESS/ASYNCHRONOUS/{HASH_PROCESS}'" 38

TABLE 14: "DESCRIPTION OF THE /PROCESS/{URI_PROCESS}/ASYNCHRONOUS/{HASH_CODE} OPERATION" 39

List of listings

LISTING 1: "EXAMPLE TO INVOKE '/LOGICALMODEL' OPERATION" 19

LISTING 2: "OUTPUT EXAMPLE FROM '/LOGICALMODEL' OPERATION" 19

LISTING 3: "EXAMPLE TO INVOKE '/LOGICALMODEL/{URI}' OPERATION" 20

LISTING 4: "OUTPUT EXAMPLE FROM '/LOGICALMODEL/{URI}' OPERATION" 20

LISTING 5: "EXAMPLE TO INVOKE '/LOGICALMODEL/{URI}/FULL' OPERATION" 21

LISTING 6: "OUTPUT EXAMPLE FROM '/LOGICALMODEL/{URI}/FULL' OPERATION" 22

LISTING 7: "EXAMPLE TO INVOKE '/LOGICALMODEL/{URI}/' SINK OPERATION" 23

LISTING 8: "OUTPUT EXAMPLE FROM '/LOGICALMODEL/{URI}/SINK' OPERATION" 23

LISTING 9: "EXAMPLE TO INVOKE '/FEATUREOFINTEREST' OPERATION" 24

LISTING 10: "OUTPUT EXAMPLE FROM '/FEATUREOFINTEREST' OPERATION" 25

LISTING 11: "EXAMPLE TO INVOKE '/OBSERVATION' OPERATION" 25

LISTING 12: "OUTPUT EXAMPLE FROM '/OBSERVATION' OPERATION" 26

LISTING 13: "EXAMPLE TO INVOKE '/OBSERVATION/{URI_OBSERVATION}/DATAWINDOW' OPERATION" 26

LISTING 14: "OUTPUT EXAMPLE FROM '/OBSERVATION/{URI_OBSERVATION}/DATAWINDOW' OPERATION" 27

LISTING 15: "EXAMPLE TO INVOKE '/OBSERVATION/{URI_OBSERVATION}' OPERATION WITHOUT DATE FILTERS" 28

LISTING 16: "EXAMPLE TO INVOKE '/OBSERVATION/{URI_OBSERVATION}' OPERATION WITH DATE FILTERS" 28

LISTING 17: "OUTPUT EXAMPLE FROM '/OBSERVATION/{URI_OBSERVATION}' OPERATION" 28

LISTING 18: "EXAMPLE TO INVOKE '/PROCESS' OPERATION" 29

LISTING 19: "OUTPUT EXAMPLE FROM '/PROCESS' OPERATION" 30

LISTING 20: "EXAMPLE TO INVOKE '/PROCESS/{URL_PROCESS}' OPERATION" 31

LISTING 21: "OUTPUT EXAMPLE FROM '/PROCESS/{URL_PROCESS}' OPERATION" 33

LISTING 22: "EXAMPLE TO INVOKE '/PROCESS/{URL_PROCESS}/SYNCHRONOUS' OPERATION" 35

LISTING 23: "OUTPUT EXAMPLE FROM '/PROCESS/{URL_PROCESS}/SYNCHRONOUS' OPERATION" 36

LISTING 24: "EXAMPLE TO INVOKE '/PROCESS/{URI_PROCESS}/ASYNCHRONOUS' OPERATION" 37

LISTING 25: "OUTPUT EXAMPLE FROM '/PROCESS/{URI_PROCESS}/ASYNCHRONOUS' OPERATION" 37

LISTING 26: "EXAMPLE TO INVOKE '/PROCESS/ASYNCHRONOUS/{HASH_PROCESS}' OPERATION" 38

LISTING 27: "OUTPUT EXAMPLE FROM '/PROCESS/ASYNCHRONOUS/{HASH_PROCESS}' OPERATION" 38

LISTING 28: "EXAMPLE TO INVOKE /PROCESS/{URI_PROCESS}/ASYNCHRONOUS/{HASH_CODE} OPERATION" 39

LISTING 29: "OUTPUT EXAMPLE FROM /PROCESS/{URI_PROCESS}/ASYNCHRONOUS/{HASH_CODE} OPERATION" 40

LISTING 30: "EXAMPLE OF CALL TO REGISTER OPERATION TO INTEGRATE A NEW OGC® WPS SERVER" 41
LISTING 31: "EXAMPLE OF RESPONSE TO REGISTER OPERATION TO INTEGRATE A NEW OGC® WPS SERVER" 41
LISTING 32: "EXAMPLE OF CALL TO UNREGISTER OPERATION TO DISINTEGRATE AN OGC® WPS SERVER" 42
LISTING 33: "EXAMPLE OF RESPONSE TO UNREGISTER OPERATION TO DISINTEGRATE AN OGC® WPS SERVER" 42
LISTING 34: "CREATE A DIRECTORY FACILITATOR SERVICE" 49
LISTING 35: "DELETE A DIRECTORY FACILITATOR SERVICE" 49
LISTING 36: "SEARCH THE AVAILABLE PROCESSES ON THE WATERP FRAMEWORK" 49
LISTING 37: "CREATE AGENT WITH AGENT MANAGEMENT SERVICE" 51
LISTING 38: "DESTROY AGENT WITH AGENT MANAGEMENT SERVICE" 51
LISTING 39: "SEARCH AN AGENT WITH AGENT MANAGEMENT SERVICE" 51
LISTING 40: "SOAP MESSAGE TO INVOKE GETCAPABILITIES OPERATION OF OGC® SOS SERVER" 53
LISTING 41: "SOAP MESSAGE TO INVOKE GETOBSERVATION OPERATION OF OGC® SOS SERVER" 53
LISTING 42: "SOAP MESSAGE TO INVOKE GETOBSERVATION OPERATION WITH FILTERS OF OGC® SOS SERVER" 54
LISTING 43: "TYPE SERVER DEFINITION ON SERVICE IDENTIFICATION DOCUMENT" 55
LISTING 44: "SOAP CALL TO GETCAPABILITIES" 56
LISTING 45: "SOAP CALL TO DESCRIBEPROCESS" 56
LISTING 46: "ASYNCHRONOUS PROCESS EXECUTION" 56
LISTING 47: "SPARQL TO OBTAIN THE LOGICAL MODELS IDENTIFIERS" 58
LISTING 48: "SPARQL TO OBTAIN A LOGICAL MODEL STRUCTURE" 59
LISTING 49: "DEFAULT INTEGRATED WPS AND SOS SERVERS AND SESAME REPOSITORIES" 69
LISTING 50:" KNOWLEDGE PROCESS CONFIGURATION" 69

1. Introduction

A water supply distribution (see D2.1 "*External System Integration Requirements*") is a system of engineered hydrologic and hydraulic components to supply water to consumers. Moreover, a successful water supply system meets the water demand, quality and distribution system requirements such as maintaining pressure to ensure the durability of supply sources. Then, water managers need information from different parts of the system in order to perform a decision making and planning procedures. Therefore, data exchange plays a key role in the water supply distribution system architecture to ensure proper operations and effective decisions.

Nowadays, water supply distribution managers in Europe use separate tools to gather data from many distributed resources, using this data in the decisional systems. All these tools use different ways of communication which are neither standardized nor interlinked. To overcome this problem, the WatERP project proposes to integrate these tools and systems in an interoperable way, creating an open interface (defined in D2.3 *"Open Interface Specification"*). Basically, this Open Interface has been based on OGC® stack taking advantage of OGC® WPS, OGC® SOS, OGC® WFS, OGC® WMS and WaterML2. On the other hand, new knowledge is generated such as financial flow from the real-water and logical representation of the water supply distribution chain, human-entities, and other involved models. This knowledge has been represented and managed by a WatERP knowledge base that is based on best known ontologies in the water and scientific community such as HY_FEATURES, CUAHSI, SWEET, SSN, … (see D1.3 *"Generic Ontology for Water supply distribution chain"* and D1.4.x *"Extension of taxonomy and ontology to the pilots"*).

In order to manage and orchestrate all water information and knowledge, a multi agent system has been designed and presented in the set of deliverables named as D7.2.X *"Implementation of MAS"*. By analysing agents' functionalities and features, it was concluded that the utility-based agent is the most suitable one for developing the WatERP SOA-MAS architecture, using the BDI (Besides Desires Intention) model to foster the MAS cooperation and necessities accomplishment. With the aim of efficiently orchestrating the water-related data and knowledge, six agents were identified: *(i)* Gateway agent; *(ii)* SOS agent; *(iii)* WPS agent; *(iv)* HMF agent; *(v)* DMS agent; and *(vi)* DSS agent.

The present deliverable is focused on presenting the multi-agent architecture with a coarse-grained in order to provide a general view of its operation. Firstly, an overview of the whole architecture is presented in the section 2 in order to contextualize the multi-agent architecture in the WatERP project. Secondly, the section 3 provides a more accurate description of the multi-agent architecture focusing on their components. Moreover, the interaction between the components is presented in the section 4. The section 5 presents the deployment view of the Mas followed by a description of the needed steps to install and configure the whole architecture. Finally, section 6 presents the conclusions and future work. The Appendix I fully details the web service description.

2. WatERP Multi Agent Architecture

The aim of this section is to present the multi-agent architecture developed throughout the WatERP project. This architecture is responsible for managing and orchestrating the available integrated resources like DSSs, DMSs, HMFs and the available observations from water informational systems. The architecture design and development was focused on Task 2.1 Open Interface, Task 2.3 Multi Agent System (MAS) and Task 7.2 Implementation of the Multi-Agent System Architecture (MAS). Based on this performed work, this section firstly presents a contextual vision of the multi-agent architecture, facilitating the understanding of the scope and the relations with other work packages.

The mentioned architecture is composed by the MAS system that actuates as an operational bus. Then, the MAS technically forces the interaction with all involved WatERP WPs developments (WP1, WP3, WP4, WP5 and WP6). The Figure 1 presents an Architectural Context Diagram (ACD) where all interactions with results of the rest WPs are reflected.

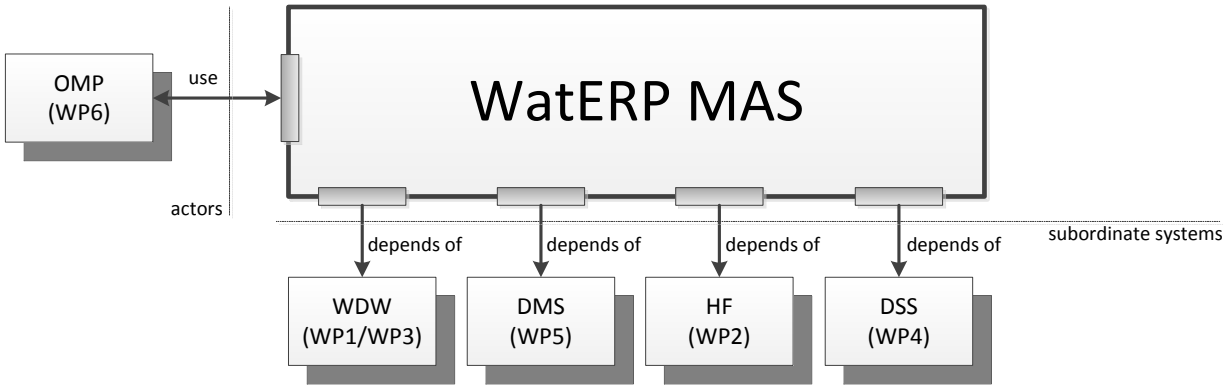

Figure 1: "Architectural context diagram (ACD)"

Mainly, the diagram presents the actors and subordinated systems that cooperate with WatERP MAS. The actors are entities that interact with the system by the production or consumption of information required for processing the user requirements (e.g. manage water resources). In our case, this entity corresponds with the OMP (WatERP WP6). The OMP is the main actor of the WatERP MAS, consuming the information that also processes towards interacting with the users as was presented in the D6.3 *"OMP 2nd prototype"* in Section 4.2.3.1. The subordinate systems (WPs developments) play the role of processing specific requirements that are necessary to complete the functions of the system and providing data/information to the MAS. These subordinate systems correspond with WDW, DMS, HF and DSS. The WDW is the responsible for providing to the WatERP MAS ontological information and processing (WP1, see D3.4 "*WDW Final Prototype*" in Section 2.3) and observational information (WP3, see D3.4 "*WDW Final Prototype*" in Section 3.1). Other subordinate system is the DMS that is able to provide the processing of the demand forecasts. These demand forecasts are used to improve the decisions-making process referring the water allocation, pump scheduling and economical

instruments enabling the decisional evaluation of financial measurements at long-term impact (see D5.5 *"Water Demand Management System and relevant documentation"* in Section 2). The WP2 also contains a subordinate system, the HMF. This subordinate system provides meteorological and hydro-meteorological processing (see D2.4 *"Water Availability Prediction System Integration"* in Section 2) which is used to improve the whole decisions-making (water resource allocation and water distribution) and also facilitates the generation of more accurate demands. Finally, the DSS contributes to the WatERP MAS providing operational processing referring water reallocation and pump scheduling.

A more detailed view is shown in the Figure 2. This figure presents the WatERP component view describing all upper components and their relations. Summarizing, the WatERP architecture is based on seven components: *(i)* the OMP as a graphical tool that allow the users to interact with whole platform; *(ii)* the MAS as a tool responsible for orchestrating all components; *(iii)* the WDW to provide the relational and ontological storage of the data; *(iv)* the HMF provides meteorological and hydro-meteorological processing to improve the whole decision-making (water resource and water distribution processes) and demand generation; *(v)* the DMS provides demand forecast processing to improve the decisions-making referring the water allocation, pump scheduling and economical instruments processing to evaluate decisions with financial long-term impact; *(vi)* DSS provides operation processing such as water reallocation and pump scheduling; and *(vi)* Pilot Integration Manager (PIM) that feed the WDW with the pilot data (water systems information). Referring the relations between the defined components, well-known standards such as OGC® SOS, OGC® WPS, REST and SPARQL has been used as a transfer data protocol or query languages. Moreover, WaterML2, WatERP Ontology and JSON have been also applied as a standard data transfer format. Mainly, these relations between components are: *(i)* the OMP–MAS interaction based on REST/JSON and SOAP/XML which was explained in the D2.4 *"Agents Goal Table and Condition Action Rules"* in Section 3.5 and it is extended in this deliverable Section 3.1; *(ii)* the MAS – WDW interaction based on SOS/WaterML2 and SPARQL which was explained in the D3.4 *"WDW Final Prototype"* Section 2.3; *(iii)* the MAS – Building block interaction (HF, DMS and DSS) which was defined in a generic way in the D2.3 *"Open Interface specification"* in Section 2.3 and contextualized for each building block in each work package; and *(iv)* the WDW-PIM interaction that is based on SOS/WaterML2 in same way as the MAS-WDW interaction (explained more detailed in D7.3.2 *"Implementation of WDW"* in Section 2).

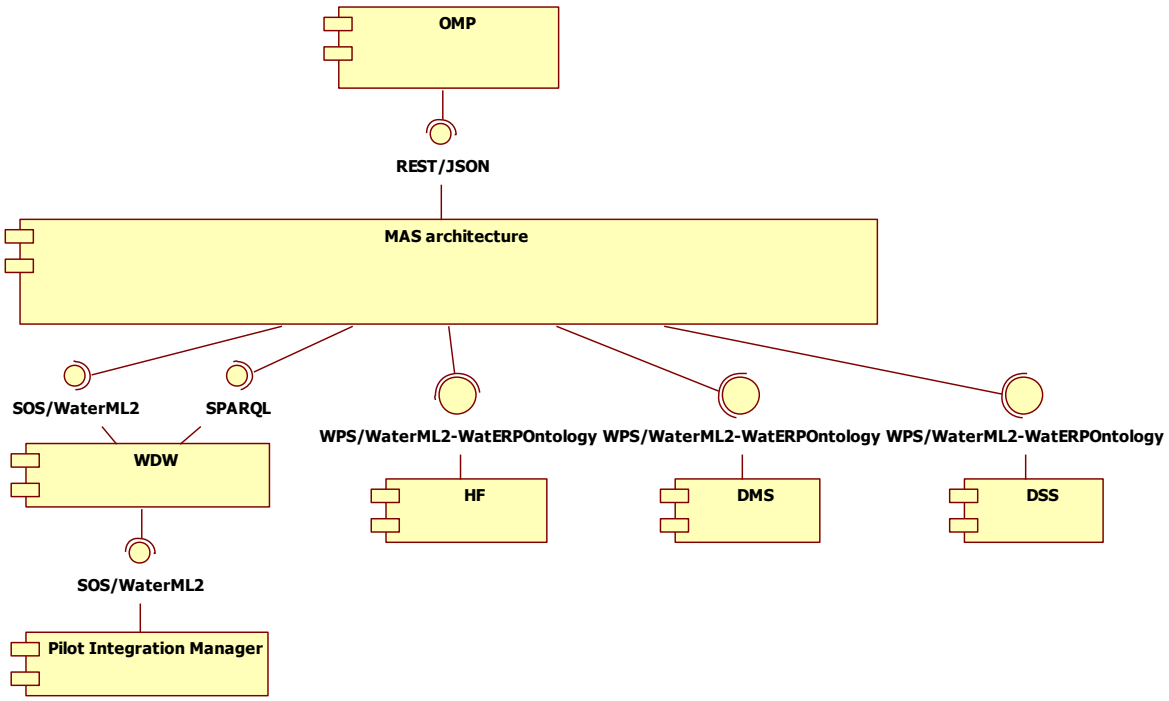

Figure 2: "WatERP component view"

Additionally, the Figure 3 presents the multi-agent architecture from the WP2 point of view. This architecture is divided on different layers, following the software engineering best-practices, which are: *(i)* interface layer; *(ii)* MAS layer; and *(iii)* external access layer.

The Interface layer is the upper level of the architecture in charge of publishing the information of the rest of modules through a REST/JSON and SOAP/XML web services. The second tier, MAS layer, is responsible for controlling the application's functionality by performing detailed processing. It is composed by several modules such as: *(i)* agents; *(ii)* WatERP Process Knowledge Service; *(iii)* Directory Facilitator Service; and *(iv)* Agent Management Service. External Access layer is the last tier of the architecture and it is aimed on accessing the data and processes of the specific described subsystems. Mainly, the modules responsible for managing the access are: *(i)* the SOS Manager to access observations published through servers based on OGC® SOS standard; *(ii)* the WPS Manager to access processes published though servers based on OGC® WPS standard; and *(iii)* the Water Ontology Manager to access ontology knowledge and generate inference over the information currently known.

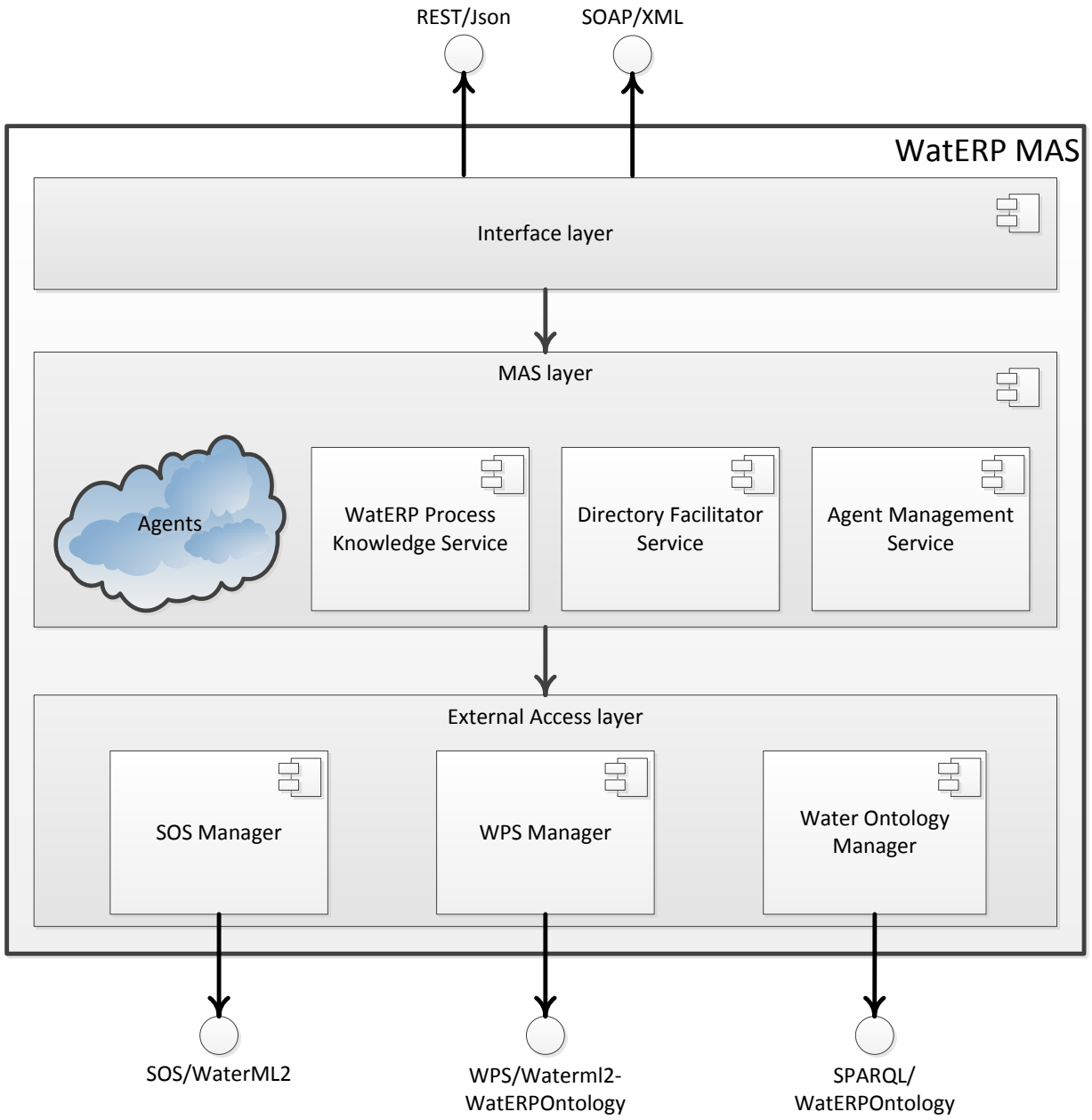

Figure 3: "MAS Component diagram"

As a conclusion of this section, the MAS architecture has been described as the interconnection of the different WatERP-modules in order to perform an efficient orchestration and scalability of the platform. In the following section, a more accurate review of the component involved in the WatERP architecture is presented.

3. Components

The aim of this section is to present all components which are part of the multi-agent architecture. Furthermore, during this section also the singularities of each module are reviewed. For that purpose, the components firstly are introduced and later they are deeply depicted. The Figure 4 shows the layer architecture introduced in the section 2 which is divided in three tiers: *(i)* interface layer; *(ii)* MAS layer; and *(iii)* external access layer. Basically, the figure has been extended presenting the available agents.

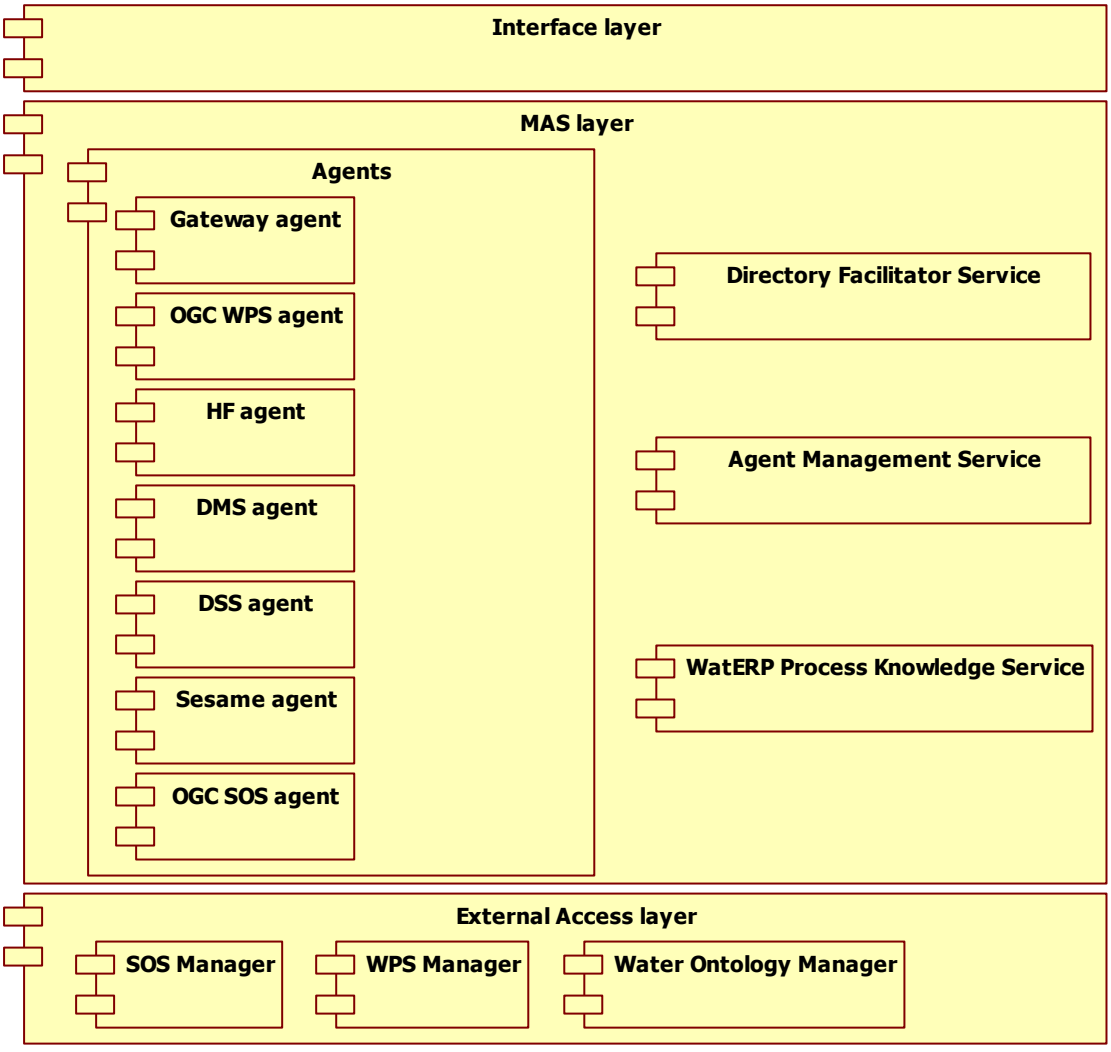

Figure 4: "Multi-agent architecture component view"

Below each component of the architecture is described, Section 3.1 presents the interface layer which is divided in two operations type: operational operations and management operations which are exposed thought web services as commented in Section 2.

3.1 Interface layer

This section is focused on introducing the interface layer component and describing its public web services. These public web services are divided in two technologies: *(i)* REST/JSON for operational operations due to its optimal integration with web environments as the OMP (e.g. reducing the data parsing); and *(ii)* SOAP/XML for management operations.

3.1.1 Operational operations

The aim of the operational operations is to provide essential tools to manage logical models, observations and processes integrated in the WatERP platform. Basically, these operations are used by the OMP with the aim of improving the daily management of the water managers, enabling the reuse of processes and data.

Below, the specific operations for the logical models, observations and processes are described following a common description. Mainly, this description contains for each operation: *(i) an* explanation of the URL used to invoke the operation; *(ii)* an explanation of the operation; *(iii)* a description of the input/s and output/s of the operation*;* and *(iv)* an example of an operation invocation including the request and the response.

Referring the input parameters is important to note that two types of input parameters are used during the invocation: the *path param* and the *query param*. The *path param* is the part of Uniform Resource Locator (URL) that contains the data that fit conveniently into a hierarchical path structure. Instead, the *query param* contains data that cannot be fitted but they are essential in order to perform the invocation. Below, in the following subsections are presented different examples for each case.

3.1.1.1 Logical model operations

This section describes all operations related with the logical model that are accessible by any WatERP MAS client. Mainly, these operations are extracted in conjunction with the work performed under WP6 by analysing the OMP requirements defined in D6.2 *"OMP 1st prototype"* in Section 2 and the information provided by this module. Then, the defined operations regarding the logical models are: *(i)* available logical models; *(ii)* basic logical model information; *(iii)* full logical model information; *(iv)* sink water resources; *(v)* storage water resources; *(vi)* source water resources; *(vii)* transformation water resources; *(viii)* transport water resources; and *(ix)* Feature Of Interest (FOI) of a water resource.

"/logicalModel" operation

This operation is used to get all logical models existing in the WatERP platform. This operation can be only invoked through a GET method. For invoking this operation, no parameters are necessary (see Table 1).

Method	Parameters	Response
GET	None	LogicalModelsResponse LogicalModel [ ] String name String uri

Table 1: "Description of the '/logicalModel' operation"

The Listing 1 shows how to invoke the *"/logicalModel*" operation.

```
HTTP GET http://localhost:8080/MasCore/service/logicalModel
```

Listing 1: "Example to invoke '/logicalModel' operation"

The Listing 2 depicts an example of the result generated by the operation *"/logicalModel*".

```
{"LogicalModelsResponse": [{"logicalModel": [
      {
      "name": "Ter_LLobregat_WaterResourceManagement",
      "uri": "Ter_LLobregat_WaterResourceManagement"
   },
      {
      "name": "SWKAWaterDistributionModel",
      "uri": "SWKAWaterDistributionModel"
   }
]}]}
```

Listing 2: "Output example from '/logicalModel' operation"

"*/logicalModel/{uri}*" operation

The "*/logicalModel/{uri}*" operation (see Table 2) is used to get a basic description of a specific logical model using its URI as an input parameter. This URI parameter is the same as the returned by the "*/logicalModel*" operation. In this operation, the GET method is used to obtain a detailed description of the logical model, including also the water resources defined in this model. As a result, this operation returns water resources which are part of the logical model requested including names, types, URIs and relations.

Method	Parameters	Response
GET	**Path parameters**	LogicalModelResponse

	String URI	WaterResource [ ] String name String type String uri String relations [ ]

Table 2: "Description of the /logicalModel/{uri} operation"

The listing Listing 3 shows how to invoke the '/logicalModel/{uri}' operation.

```
HTTP GET http://localhost:8080/MasCore/service/logicalModel/SWKAWaterDistributionModel
```

Listing 3: "Example to invoke '/logicalModel/{uri}' operation"

Below, an example of the result generated by the /logicalModel/{uri} operation is showed for the requested "*SWKAWaterDistributionModel*" logical model.

```
{"LogicalModelResponse": [{"waterResource": [
      {
      "name": "Si3",
      "type": "Sink",
      "uri": "SWKA_Si3"
   },
...
      {
      "name": "T4",
      "relations":       [
         "SWKA_Si2",
         "SWKA_Si1",
         "SWKA_ST1"
      ],
      "type": "Transformation",
      "uri": "SWKA_TF4"
   }
]}]}
```

Listing 4: "Output example from '/logicalModel/{uri}' operation"

"*/logicalModel/{uri}/full*" operation

This operation (see Table 3) is used to get a complete description of a logical model by using its URI. In the same way as the previous operation, the "*/logicalModel/{uri}/full*" operation is invoked using GET method. This method requires as an input parameter the logical model URI. The response of this method contains the same water resources presented in the previous operation ("*/logicalModel/{uri}*") detailed with: *(i)* the FOIs of each water resource with their names and URIs; and *(ii)* the observations of each FOI with their names, URIs, phenomenon and procedure.

Method	Parameters	Response
GET	**Path parameters** String URI	LogicalModelResponse WaterResource [ ] String name String type String uri String relations [ ] FeatureOfInterest [ ] String name String uri Observations [ ] String name String uri String phenomenon String procedure

Table 3: "Description of the /logicalModel/{uri}/full operation"

The Listing 5 shows how to invoke the "*/logicalModel/{uri}/full*" operation.

```
HTTP GET http://localhost:8080/MasCore/service/logicalModel/SWKAWaterDistributionModel
```

Listing 5: "Example to invoke '/logicalModel/{uri}/full' operation"

Below, an example of the result generated by the "*/logicalModel/{uri}/full*" operation is showed for the "*SWKAWaterDistributionModel*" requested logical model.

```
{"LogicalModelResponse": [{"waterResource": [
      {
      "featuresOfInterests":       {
         "name": "KarlsruheSubAreaA-Land",
         "observations":          {
            "name": "Obs_KarlsruheSubAreaA-Land_Consumption1Hour_AlgorithmCalculated",
```

```
            "phenomenon": "Consumption1Hour",
            "procedure": "AlgorithmCalculated",
            "uri": "Obs_KarlsruheSubAreaA-Land_Consumption1Hour_AlgorithmCalculated"
         },
         "uri": "KarlsruheSubAreaA-Land"
      },
      "name": "Si3",
      "type": "Sink",
      "uri": "SWKA_Si3"
   },
...
      {
      "featuresOfInterests":       {
         "name": "KarlsruheSubAreaB-Land",
         "observations":          {
            "name": "Obs_KarlsruheSubAreaB-Land_Consumption1Hour_AlgorithmCalculated",
            "phenomenon": "Consumption1Hour",
            "procedure": "AlgorithmCalculated",
            "uri": "Obs_KarlsruheSubAreaB-Land_Consumption1Hour_AlgorithmCalculated"
         },
         "uri": "KarlsruheSubAreaB-Land"
      },
      "name": "Si1",
      "type": "Sink",
      "uri": "SWKA_Si1"
   }
]}]}
```


Listing 6: "Output example from '/logicalModel/{uri}/full' operation"

***"/logicalModel/{uri}/sink"* operation**

The "*/logicalModel/{uri}/sink*" operation (see Table 4) is used to access the water resources of "*Sink*" type belonging to a certain logical model. This operation is invoked using GET method. It requires an input parameter which is the logical model URI. The response structure contains only the minimal information of the water resources defined by the name, type and URI of the "*Sink*" water resource.

Method	Parameters	Response
GET	**Path parameters** String URI	LogicalModelResponse WaterResource [ ] String name String type String uri

Table 4: "Description of the '/logicalModel/{uri}/sink' operation"

The Listing 7 shows how to invoke the "*/logicalModel/{uri}/sink*" operation for the "*SWKAWaterDistributionModel*" logical model.

```
HTTP GET http://localhost:8080/MasCore/service/logicalModel/SWKAWaterDistributionModel/sink
```

Listing 7: "Example to invoke '/logicalModel/{uri}/' sink operation"

Below, an example of the result generated by the "*/logicalModel/{uri}/sink*" operation is showed.

```
{"LogicalModelResponse": [{"waterResource": [
      {
      "name": "Si3",
      "type": "Sink",
      "uri": "SWKA_Si3"
   },
      {
      "name": "Si2",
      "type": "Sink",
      "uri": "SWKA_Si2"
   },
      {
      "name": "Si1",
      "type": "Sink",
      "uri": "SWKA_Si1"
   }
]}]}
```

Listing 8: "Output example from '/logicalModel/{uri}/sink' operation"

Following the same pattern, the operations: *(i)* "*/logicalModel/{uri}/storage*"; *(ii)* "*/logicalModel/{uri}/source*"; *(iii)* "*/logicalModel/{uri}/transformation*"; *(iv)* "*/logicalModel/{uri}/transport*" are similarly executed (using the URI of the logical model). The responses of each mentioned operations have the same format, containing the basic description for the water resources that fit with the type used in the call (sink, storage, source, transformation or transport).

"*/featureOfInterest*" operation

Operation that is used to get the features of interest belonging to a water resource of a specific logical model (see Table 5). This operation is invoked using GET method and requires two query parameters: the logical model URI and the water resource URI. The response structure of this operation contains the FOI's information which belong to the required water resource of a logical model.

Method	Parameters	Response
GET	**Query parameters** String uri_logical_model Sting uri_water_resource	FeaturesOfInterestResponse String logicalModel String waterResource FeatureOfInterests [ ] String name String uri

Table 5: "Description of the '/featureOfInterest/{uri_logical_model}/{uri_water_resource}' operation"

The Listing 9 shows how to invoke the "*/featureOfInterest*" operation for the "*SWKAWaterDistributionModel*" and the "*SWKA_Si3*" water resource.

```
HTTP GET

http://localhost:8080/MasCore/service/featureOfInterest?uri_logical_model=SWKAWaterDistributionModel&uri
_water_resource=SWKA_Si3
```

Listing 9: "Example to invoke '/featureOfInterest' operation"

As a result of this request, the Listing 10 depicts the feature of interest contained in the "*SWKA_Si3*".

```
{"FeaturesOfInterestResponse": {

   "featuresOfInterests":    {

      "name": "KarlsruheSubAreaA-Land",

      "uri": "KarlsruheSubAreaA-Land"

   },

   "logicalModel": "SWKAWaterDistributionModel",

   "waterResource": "SWKA_Si3"
```

```
}}
```

Listing 10: "Output example from '/featureOfInterest' operation"

3.1.1.1 Observation operations

This section describes all operations related with observations that are accessible by a WatERP MAS client. Basically, these operations are extracted from the WP6 requirements (See D6.2 "OMP 1st prototype" in Section 2 and they are: *(i)* available observations; *(ii)* observations data window; and *(iii)* observation information.

"*/observation*" operation

The *"/observation*" operation (see Table 6) is used to get the observations of a feature of interest by using its URI. This operation is invoked using GET method and requires three query parameters such as logical model URI, water resource URI and feature of interest URI. Finally, the response of this operation contains information referring the invocation such as logical model URI, water resource URI, feature of interest URI, the observation name, observation URI, observation phenomenon and procedure for the observations that fit with the provided filters.

Method	Parameters	Response
GET	**Query parameters** String uri_logical_model Sting uri_water_resource Sting uri_feature_of_interest	ObservationsResponse String logicalModel String waterResource String featureOfInterest Observation [ ] String name String phenomenon String procedure String uri

Table 6: "Description of the '/observation' operation"

The Listing 11 shows how to invoke the operation "*/observation*" for the "*SWKAWaterDistributionModel*" logical model, the "*SWKA_Si3*" water resource and the "*KarlsruheSubAreaA-Land*" feature of interest.

```
HTTP GET

http://localhost:8080/MasCore/service/observation?uri_logical_model=SWKAWaterDistributionModel&uri_water
_resource=SWKA_Si3&uri_feature_of_interest=KarlsruheSubAreaA-Land
```

Listing 11: "Example to invoke '/observation' operation"

Below, an example of the result generated by the operation "*/observation*" defined in the Listing 11 is showed.

```
{"ObservationsResponse": {
   "featureOfInterest": "KarlsruheSubAreaA-Land",
   "logicalModel": "SWKAWaterDistributionModel",
   "observations":    {
      "name": "Obs_KarlsruheSubAreaA-Land_Consumption1Hour_AlgorithmCalculated",
      "phenomenon": "Consumption1Hour",
      "procedure": "AlgorithmCalculated",
      "uri": "Obs_KarlsruheSubAreaA-Land_Consumption1Hour_AlgorithmCalculated"
   },
   "waterResource": "SWKA_Si3"
}}
```

Listing 12: "Output example from '/observation' operation"

***"/observation/{uri_observation}/dataWindow"* operation**

Operation used to get data window of a specific observation by using its URI. This operation is invoked using GET method and requires three query parameters such as logical model URI, water resource URI and feature of interest URI. Finally, the response structure contains information referring the invocation such as logical model URI, water resource URI and feature of interest URI and the observation name, observation URI, observation phenomenon and procedure phenomenon for the observations that fit with the provided filters.

Method	Parameters	Response
GET	**Path parameters** String uri_observation	DataWindowResponse String first_datetime String last_datetime

Table 7: "Description of the '/observation/{uri_observation}/dataWindow' operation"

The Listing 13 shows how to invoke the operation /observation/{uri_observation}/dataWindow

```
HTTP            GET            http://localhost:8080/MasCore/service/observation/Obs_KarlsruheSubAreaA-
Land_Consumption1Hour_AlgorithmCalculated/dataWindow
```

Listing 13: "Example to invoke '/observation/{uri_observation}/dataWindow' operation"

Below, an example of the result generated by the operation /observation/{uri_observation}/dataWindow is showed.

```
{"DataWindowResponse": {
   "firstDatetime": "2012-01-01T00:00:00+01:00",
   "lastDatetime": "2015-04-19T16:00:00+01:00 "
}}
```

Listing 14: "Output example from '/observation/{uri_observation}/dataWindow' operation"

***"/observation/{uri_observation}"* observation**

Operation used to get time series regarding an observation, using its URI to filter the observation (see Table 8). This operation has two different ways to be invoked, both using GET method. One way, only requires the observation URI and logical model URI. This function returns the full time series from zero to date. At this moment, it is important to note that this form to retrieve time series for a specific observation is obtained at discouraged due to the large amount of data that the time series can be contain. The other operation allows you to filter by defining a start and end date. Hence, the operation requires three query parameters as the logical model URI, the start date and the end date; and one path parameter defined by the observation URI. The response structures for both operations is formed by the logical model URI, water resource URI and feature of interest URI and the observation URI and the data in WaterML2 format.

Method	Parameters	Response
GET	**Path parameters** String uri_observation **Query parameters** String uri_logical_model	DataObservationResponse String logicalModel String waterResource String featureOfInterest String observation String data
GET	**Path parameters** String uri_observation **Query parameters** String uri_logical_model String start_datetime String end_datetime	DataObservationResponse String logicalModel String waterResource String featureOfInterest String observation String data

Table 8: "Description of the '/observation/{uri_observation}' operation"

The Listing 15 shows how to invoke the "*/observation/{uri_observation}*" operation without applying the filter by date. This invocation is performed to the

"*Obs_HBLUR_WaterTable1Hour_SensorMeasured_NorthTankLevel*" observation for the "*SWKAWaterDistributionModel*" logical model.

```
HTTP GET

http://localhost:8080/MasCore/service/observation/
Obs_HBLUR_WaterTable1Hour_SensorMeasured_NorthTankLevel?uri_logical_model=SWKAWaterDistributionModel
```

Listing 15: "Example to invoke '/observation/{uri_observation}' operation without date filters"

The Listing 16 shows how to invoke the operation "*/observation/{uri_observation}*" by applying the filter by date. Thus, using similar example and the explained above the time series has been filtered between "*2015-01-01T00:00:00+01:00*" and "*2015-01-31T23:00:00+01:00*" date time.

```
HTTP GET

http://localhost:8080/MasCore/service/observation/
Obs_HBLUR_WaterTable1Hour_SensorMeasured_NorthTankLevel
?uri_logical_model=SWKAWaterDistributionModel&start_datetime=2015-01-
01T00:00:00+01:00&end_datetime=2015-01-31T23:00:00+01:00
```

Listing 16: "Example to invoke '/observation/{uri_observation}' operation with date filters"

Using the examples explained, the Listing 17 depicts the return structure for the both "*observation/{uri_observation}*". It is important to note that the content of *data* parameter is a WaterML2 which is formatted following the guidelines presented in the D2.3 "Open Interface Specification" in Section 3.2.1.1.

```
{" DataObservationResponse": {

   "waterResource": "SWKA_Si1",

   "featureOfInterest": "HBLUR",

   "observation": "Obs_HBLUR_WaterTable1Hour_SensorMeasured_NorthTankLevel",

   "data": "Waterml2.xml"

}}
```

Listing 17: "Output example from '/observation/{uri_observation}' operation"

3.1.1.2 Process operations

The aim of this section is to present the available process operations in the multi agent architecture and depict how they should be used. The identified process operations are the following: *(i)* get the available operations ("*/process*"); *(ii)* filter the available operations ("*/process*"); *(iii)* describe how to use a process ("*/process/{url_process}*"); *(iv)* execute a synchronous process ("*/process/{url_process}/synchronous*"); *(v)* execute an asynchronous process ("*/process/{url_process}/asynchronous*"), *(vi)* check if an asynchronous process have finished its execution ("*/process/asynchronous/{hash_process}*"); and *(vii)* get the result of an asynchronous process ("*/process/{uri_process}/asynchronous/{hash_process}*").

"*/process*" operation

This operation is used to retrieve all available integrated process in WatERP framework (see Table 9). Basically, the response is a set of process identifiers with its human-readable information (title and abstract).

Method	Parameters	Response
GET	none	ProcessesResponse Process [ ] String identifier String title String abstract

Table 9: "Description of the '/process' operation"

The Listing 18 shows how to invoke the operation "*/process*".

```
HTTP GET

http://localhost:8080/MasCore/service/process
```

Listing 18: "Example to invoke '/process' operation"

As a result of the execution of this process, this operation returns the information of all the processes defined for the WatERP platform. Due to the large amount of defined processes, the Listing 19 only describes the "*HourlyDemandForecastProcess*" and "*DailyTemperatureForecast*" including their specific identifier, title and abstract.

```
{" ProcessesResponse": [{"process": [
      {
      "identifier": "gr.iccs.waterp.dms.process.HourlyDemandForecastProcess",
      "title": "Hourly demand forecast process",
      "abstract": "Hourly demand forecast process"
   },
. . .
      {
      "identifier": "es.hyds.waterp.hf.process.DailyTemperatureForecast",
      "title": "Daily temperature forecast",
      "abstract": "Daily temperature forecast"
   }
]}]}
```

Listing 19: "Output example from '/process' operation"

"/process/{uri_process}" operation

Operation that is used for getting the description of how to use a process (see Table 10). Thus, the process is filtered by URI (parameter for invoking the operation). The response of this operation contains different mechanism (input parameters) to run the same process taking advantage of the MAS orchestration. Thus, this operation returns the process information complemented by the required input parameters and the generated output parameters. The input parameters for the process execution can be other processes, complex or simple data. Similarly, the output parameters for the process execution can be complex and simple data.

Method	Parameters	Response
GET	**Path parameters** String uri_process	ExecutionProfilesResponse Process [ ] String identifier String title String abstract Keywords [ ] String keyword Type String codeSpace String name Input Process [ ] Complex [ ] String identifier String title String abstract String mimeType String encoding String schema Keywords [ ] Simple [ ] String identifier String title String abstract

		String dataType Keywords [ ] Output Complex [ ] Simple [ ]

Table 10: "Description of the '/process/{url_process}' operation"

The Listing 20 shows how to invoke the "*/process/{url_process}*" operation for the "*DailyTemperatureForecast*" process.

```
HTTP GET

http://localhost:8080/MasCore/service/process/es.hyds.waterp.hf.process.DailyTemperatureForecast
```

Listing 20: "Example to invoke '/process/{url_process}' operation"

Below, an example of the result generated by the "*/process/{url_process}*" operation is showed for the "*DailyTemperatureForecast*" example. Mainly, the obtained result presents two ways to execute the "*DailyTemperatureForecast*" procedure. On the one hand, by providing water resource information in "*text/xml+rdf*" format and forecast horizon in "*days*". On the other hand, without input parameters.

```
{
  "ExecutionProfilesResponse": [
    {
      "Process": {
        "identifier": "es.hyds.waterp.hf.process.DailyTemperatureForecast",
        "title": "Daily temperature forecast",
        "abstract": "Daily temperature forecast",
        "Input": {
          "Complex": [
            {
              "identifier": "waterResource",
              "title": "Water resource",
              "abstract": "Water resource",
              "mimeType": "text/xml+rdf",
              "encoding": "UTF-8",
              "schema": "http://www.waterp-fp7.eu/WatERPOntology.owl"
            }
          ],
```

```
          "Simple": [
            {
              "identifier": "forwardTof",
              "title": "Forecast horizon in days",
              "abstract": "Forecast horizon in days",
              "dataType": "int"
            }
          ]
        },
        "Output": {
          "Complex": [
            {
              "identifier": "dailyMinTemperatureForecast",
              "title": "Daily min temperature forecast",
              "abstract": "Daily min temperature forecast",
              "mimeType": "text/xml",
              "encoding": "UTF-8",
              "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd"
            } ,
. . .
            {
              "identifier": "dailyMaxTemperatureForecast",
              "title": "Daily max temperature forecast",
              "abstract": "Daily max temperature forecast",
              "mimeType": "text/xml",
              "encoding": "UTF-8",
              "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd"
            }
          ]
        }
      }
    },
      "Process": {
```

```
        "identifier": "es.hyds.waterp.hf.process.DailyTemperatureForecast",
        "title": "Daily temperature forecast",
        "abstract": "Daily temperature forecast",
        "Output": {
          "Complex": [
            {
              "identifier": "dailyMinTemperatureForecast",
              "title": "Daily min temperature forecast",
              "abstract": "Daily min temperature forecast",
              "mimeType": "text/xml",
              "encoding": "UTF-8",
              "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd"
            } ,
. . .
            {
              "identifier": "dailyMaxTemperatureForecast",
              "title": "Daily max temperature forecast",
              "abstract": "Daily max temperature forecast",
              "mimeType": "text/xml",
              "encoding": "UTF-8",
              "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd"
            }
          ]
        }
      }
    }
  ]
}
```


Listing 21: "Output example from '/process/{url_process}' operation"

"*/process/{uri_process}/synchronous*" operation

Operation used for executing a process in a synchronous way considering the URI of the process (see Table 11). This operation corresponds with a POST method where the message contains extra

information in order to perform the execution of the process. This extra information refers to the nature of the input parameters of the process to be called (simple or complex input parameters or other processes). As a response, this operation contains the output classified on simple or complex data following OGC® WPS guidelines. On the one hand, the simple data provides extra information as data type. On the other hand, complex data can be mime type, encoding and the schema supported by the response.

Method	Parameters	Response
POST	**Path parameters** String uri_process **POST message** ExecuteRequest Process process String identifier Input input Process process [ ] Simple simple [ ] String identifier String data Complex complex [ ] String identifier String data	ExecuteResponse Process process String identifier String title String abstract Output output Complex [ ] String identifier String title String abstract String mimeType String encoding String schema String data [ ] Simple [ ] String identifier String title String abstract String dataType String data [ ]

Table 11: "Description of the '/process/{url_process}/synchronous' operation"

The Listing 22 shows how to invoke the operation "*/process*" for the "*EvaluationProcess*" of the WatERP DSS.

```
HTTP POST

http://localhost:8080/MasCore/service/process/org.bdigital.alim.waterp.dss.process.EvaluationProcess
/synchronous

{
```

```
  "ExecuteRequest": {
    "process": {
          "identifier": "org.bdigital.alim.waterp.dss.process.EvaluationProcess",
          "input": {
            "process": [
              {"identifier": "gr.iccs.waterp.dms.process.HourlyDemandForecastProcess",
              "input": {
                    "process": [
                              {"identifier": "es.hyds.waterp.hf.process.DailyTemperatureForecast"}
                      ]}}]}}}
```

Listing 22: "Example to invoke '/process/{url_process}/synchronous' operation"

Below, an example of the generated result for the "*/process/{url_process}/synchronous"* operation applied to the "*EvaluationProcess*" of the WatERP-DSS, is showed (see Listing 23). As can be appreciated, the WatERP-DSS returns as an output a complex data type composed by the "*HourlyDemandForecast*" codified in WaterML2 format, and the "*TotalEnergy*" encapsulated in a RDF/XML. For this operation, it is important to note that the content of this response is generated on-fly. Hence, it is totally dependent of the executed process. Then, a section of the response is only listed which lets to you view the structure of the response.

```
{"ExecuteResponse": [{"process": {
   "identifier": "org.bdigital.alim.waterp.dss.process.EvaluationProcess",
   "output": {"complex":    [
            {
         "identifier": "hourlyDemandForecast",
         "title": "Hourly demand forecast",
         "abstract": "Hourly demand forecast",
         "mimeType": "text/xml",
         "encoding": "UTF-8",
         "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd",
         "data": "waterml2"
      },
. . .
            {
         "identifier": "totalEnergy3",
```

```
        "title": "Total Energy 1",
        "abstract": "Total Energy 1",
        "mimeType": "text/xml+rdf",
        "encoding": "UTF-8",
        "schema": "http://www.waterp-fp7.eu/WatERPOntology.owl",
        "data": "ontology"
      }
   ]},
   "sync": true
```

Listing 23: "Output example from '/process/{url_process}/synchronous' operation"

"***/process/{uri_process}/asynchronous*" operation**

This operation is used for executing a process asynchronously, taking into account the process URI. This method corresponds with a POST method. Moreover, the message contains extra information in order to perform the execution of the process. This extra information corresponds with the input parameters of the specific process to be called (simple or complex input parameters or other processes). As a response, the operation generates a hash code to identify the process in future MAS queries such as check the status of the process and retrieve the process output.

Method	Parameters	Response
POST	**Path parameters** String uri_process **POST message** ExecuteRequest Process process String identifier Input input Process process [ ] Simple simple [ ] String identifier String data Complex complex [ ] String identifier String data	ExecuteAsynchronousResponse String hashCode

Table 12: "Description of the '/process/{url_process}/asynchronous' operation"

The Listing 24 shows how to invoke the "*/process/{uri_process}/asynchronous*" operation for the "*SimulationProcess*" process of the WatERP-DSS through HTTP POST call. The content of the complex data "*hourlyDemandForecast*", "*hourlyAggregatedDemandForecast*" and "*pumpingScheduling*" have been omitted in order to reduce the information to be shown in the example. The most remarkable aspect of these complex data is that they are based on WaterML2, following the guidelines presented in the D2.3 *"Open Interface Specification"* in Section 3.2.1.1.

```
HTTP POST

http://localhost:8080/MasCore/service/process/0330-fe41-412b-9acd-6a0ed7e765f2/asynchronous/

{

  "ExecuteRequest": {

    "process": {

          "identifier": "org.bdigital.alim.waterp.dss.process.SimulationProcess",

          "input": {

                   "complex": [

                            { "identifier": "hourlyDemandForecast",

                              "data": "hourlyDemandForecast.xml" },

                            { "identifier": "hourlyAggregatedDemandForecast",

                              "data": "hourlyAggregatedDemandForecast.xml" },

                            { "identifier": "pumpingScheduling",

                              "data": "pumpingScheduling.xml" }

]}}}}
```

Listing 24: "Example to invoke '/process/{uri_process}/asynchronous' operation"

As a response, the generated result for the *"/process/{uri_process}/asynchronous"* applied to the "*SimulationProcess*" process of the WatERP-DSS is a hash code as reflected in the Listing 25.

```
{"ExecuteAsynchronousResponse": {"hashCode": "0330-fe41-412b-9acd-6a0ed7e765f2"}}
```

Listing 25: "Output example from '/process/{uri_process}/asynchronous' operation"

"*/process/asynchronous/{hash_process}*" operation

This operation is used for evaluating the execution state for an asynchronous process using the hash process which is retrieved during an asynchronous invocation (see Table 13). Mainly, the operation returns the hash code of the process in form of String and a Boolean indicating the process availability only if the result of the process is available.

Method	Parameters	Response
GET	**Path parameters** String hash_process	AsynchronousResponse String hash_process boolean available

Table 13: "Description of the operation '/process/asynchronous/{hash_process}'"

The Listing 26 shows how to invoke the "*/process/asynchronous/{hash_process}*" operation for the "*0330-fe41-412b-9acd-6a0ed7e765f2*" hash code resultant for the execution of an asynchronous process for the "*SimulationProcess*" of the WatERP-DSS.

```
HTTP GET

http://localhost:8080/MasCore/service/process/asynchronous/0330-fe41-412b-9acd-6a0ed7e765f2
```

Listing 26: "Example to invoke '/process/asynchronous/{hash_process}' operation"

Below, an example of the generated result by the operation "*/process/asynchronous/{hash_process}*" is showed (see Listing 27). This output contains the hash code of the process and the availability represented by a Boolean ("*true*").

```
{"AsynchronousResponse": [{

   "hashCode": "0330-fe41-412b-9acd-6a0ed7e765f2",

   "available": true

}]}
```

Listing 27: "Output example from '/process/asynchronous/{hash_process}' operation"

"/process/{uri_process}/asynchronous/{hash_process}" operation

This operation is used for getting the result of an asynchronous execution, filtering by the URI of the invoked process and by the hash code returned during the asynchronous invocation (see Table 14). The response structure is the same as described in "*/process/{uri_process}/synchronous*" operation.

Method	Parameters	Response
GET	**Path parameters** String uri_process String hash_process	ExecuteResponse Process process String identifier String title String abstract Output output Complex [ ]

		String identifier String title String abstract String mimeType String encoding String schema String data [ ] Simple [ ] String identifier String title String abstract String dataType String data [ ]

Table 14: "Description of the /process/{uri_process}/asynchronous/{hash_code} operation"

The Listing 28 shows how to invoke the "*/process/{uri_process}/asynchronous/{hash_code}*"operation for the "*SimulationProcess*" process of the WatERP-DSS with "*0330-fe41-412b-9acd-6a0ed7e765f2*" hash code.

```
HTTP GET

http://localhost:8080/MasCore/service/process/org.bdigital.alim.waterp.dss.process.SimulationProcess/asynchronous/0330-fe41-412b-9acd-6a0ed7e765f2
```

Listing 28: "Example to invoke /process/{uri_process}/asynchronous/{hash_code} operation"

As a result, the "*/process/{uri_process}/asynchronous /{hash_code}*"operation produces as an output a complex data type composed by the "*Energy*" encapsulated in RDF, and the "*waterTableForecast*" encapsulated in "*text/xml*" (see Listing 29). In the same way of the "*/process/{uri_process}/synchronous*" operation, the content of this response is generated on-fly. Hence, this response is totally dependent of the executed process. Then, a section of the response is only listed which lets to you view the structure of the response.

```
{"ExecuteResponse": [{"process": {
   "output": {"complex":    [
            {
         "identifier": "energy",
         "title": "Energy",
         "abstract": "Energy",
         "mimeType": "text/xml+rdf",
```

```
          "enconding": "UTF-8",
          "schema": "http://www.waterp-fp7.eu/WatERPOntology.owl",
          "data": "energy.rdf"
       },
. . .
             {
          "identifier": "waterTableForecast",
          "title": "Water table forecast",
          "abstract": "Water table forecast",
          "mimeType": "text/xml",
          "enconding": "UTF-8",
          "schema": "http://schemas.opengis.net/waterml/2.0/waterml2.xsd",
          "data": "waterTableForecast.xml"
       }
    ]},
    "sync": true
}}]}
```

Listing 29: "Output example from /process/{uri_process}/asynchronous/{hash_code} operation"

3.1.2 Management operations

The aim of this section is to present the management operations which are responsible to manage the integration of the OGC® WPS servers with the multi agent system architecture. These operations are published through a web service based on SOAP-XML protocol and basically, they are: *(i) register* to integrate new OGC® WPS servers and *(ii) unregister* to unregister OGC® WPS servers.

The *register* operation allows the MAS to integrate new OGC® WPS servers in the WatERP platform. Then, the processes of these incorporated OGC® WPS are available in the platform and are orchestrated by the MAS. As contrary, the *unregister* operation performs the inverse operation. That is, this process removes all processes provided by an OGC® WPS server and decouples the OGC® WPS server from the WatERP platform. Figure 5 presents a schema that describes the definition of the web service WSDL.

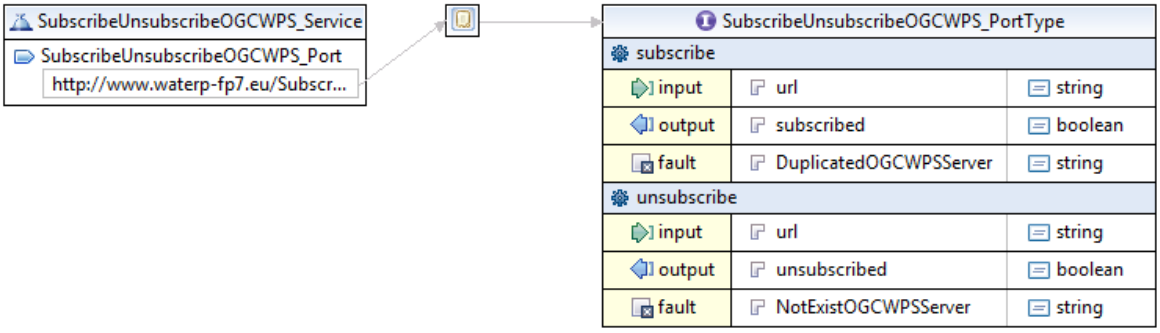

Figure 5: "WSDL schema for the management operations"

Below, two examples referring management operations are presented. One of the examples is focused at depicting the register operation for an OGC® WPS server. The other example is centred on unregistering an OGC® WPS server from the WatERP platform.

Concerning the register OGC® WPS server, the SOAP-XML request shows how to integrate a server referenced by the URL http://localhost:8080/WPS (see Listing 30). The correspondent response is depicted on Listing 31 where an affirmative answer to the integration is returned.

```
<soapenv:Envelope
                    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                    xmlns:ser="http://service.mas.waterp.alim.bdigital.org/">
          <soapenv:Header />
          <soapenv:Body>
                    <ser:registerWPS>
                              <url>http://localhost:8080/WPS</url>
                    </ser:registerWPS>
          </soapenv:Body>
</soapenv:Envelope>
```

Listing 30: "Example of call to register operation to integrate a new OGC® WPS server"

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
          <soap:Body>
                    <ns2:registerWPSResponse

          xmlns:ns2="http://service.mas.waterp.alim.bdigital.org/">
                              <return>true</return>
                    </ns2:registerWPSResponse>
          </soap:Body>
</soap:Envelope>
```

Listing 31: "Example of response to register operation to integrate a new OGC® WPS server"

The unregister operation is requested by indicating to the platform the OGC® WPS server (by URL) to be removed (see Listing 32). As a response of this operation, the SOAP-XML response described in the Listing 33 shows the status of the unregistering a process that in this case is successful ("True" parameter in the "*return*" tag).

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:ser="http://service.mas.waterp.alim.bdigital.org/">
          <soapenv:Header />
          <soapenv:Body>
                    <ser:unregisterWPS>
```

```
                        <url>http://localhost:8080/WPS</url>
                    </ser:unregisterWPS>
            </soapenv:Body>
</soapenv:Envelope>
```

Listing 32: "Example of call to unregister operation to disintegrate an OGC® WPS server"

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
            <soap:Body>
                        <ns2:registerWPSResponse

            xmlns:ns2="http://service.mas.waterp.alim.bdigital.org/">
                        <return>true</return>
                    </ns2:registerWPSResponse>
            </soap:Body>
</soap:Envelope>
```

Listing 33: "Example of response to unregister operation to disintegrate an OGC® WPS server"

3.2 MAS layer

The aim of MAS layer is to orchestrate the processes integrated in the WatERP platform through OGC® WPS and OGC® SOS servers. Hence, one of the purposes is to integrate all information and facilitate the decision making to improve water resource management and energy efficiency in the water supply distribution chain.

As it was presented in the "*D7.2.X*", an overview of the agents and multi agents' technologies previously used in the water domain with a general introduction to matchmaking was described. Also, the main concepts and the different types of agents were defined and analysed. These deliverables concludes that the utility-based agents were the most suitable for the development of the WatERP SOA-MAS architecture using the Belief-Desire-Intention (BDI) model. The utility-based agents provide to the WatERP SOA-MAS architecture the capabilities of: *(i)* representing the goals with high-level knowledge; and *(ii)* understanding previous knowledge of the world which is essential in order to carry out the matchmaking. Moreover, the utility-based agent adds the possibility of weighting the actions. Particularly, weighting the actions was considered important to select the best action once a conflict between goals appears. Furthermore in these deliverables different methodologies to develop multi-agent systems were studied. In this sense, the deliverables conclude that the MAS-CommonKADS methodology (See D7.2.1 *"Implementation of MAS"* section 3.2) was the most suitable to be applied on the WatERP MAS design since it supports most of the utility-based BDI agents' characteristics. Moreover, the MAS-CommonKADS methodology was applied in the D7.2.1 "Implementation of MAS" in Section 4 carrying out the following tasks: *(i)* identify the actors; *(ii)* describe the actors; *(iii)* identify the user stories; *(iv)* describe the user stories; *(v)* identify the software agents; *(vi)* model the tasks; *(vii)* identify the software agents (second round); *(ix)* model the coordination between agents describing the conversations, scenarios and messages processing state diagrams; *(x)* model the organization; and *(xi)* identify the network agents'. The main highlight of this deliverable was the identification of three agents: *(i)* Gateway agent to manage the interaction OMP-MAS; *(ii)* Sesame agent for managing the interaction with ontological knowledge and *(iii)* Yellow Pages agent as base to perform the matchmaking.

In the second iteration of the D7.2, D7.2.2 *"Implementation of MAS"*, a new restatement on the MAS-CommonKADS methodology was done by enhancing MAS system capabilities with the inclusion OGC® WPS and SOS interaction. At this stage, new agents were identified such as: *(i)* OGC® SOS agent; *(ii)* OGC® WPS agent; and *(iii)* specific agents (HMF agent, DMS agent and DSS agent).

During this last version, the MAS is reviewed with a coarse-grained, presenting the different involved components and the interactions between them. For a finer granularity consult D7.2.1 and D7.2.2.

3.2.1 Agents

An agent is just something that acts. Hence, the agent could be considered as anything that perceives its environment through sensors and acts upon that environment through actuators. But computer agents are expected to have other attributes that distinguish them from mere "*programs*", such as operating under autonomous control, perceiving their environment, persisting over a prolonged time period, adapting to change, and being capable of taking on another's goals (Russell & Norvig, 2009).

Basically, the agents identified in the WatERP Multi-agent architecture are: *(i)* gateway agent; *(ii)* Sesame agent; *(iii)* OGC® SOS agent; *(iv)* OGC® WPS agent; *(v)* HF agent; *(vi)* DMS agent and *(vii)* DSS agent.

The **gateway agent** is mainly a dispatcher. This agent is responsible for receiving all external actions with the system through operational operations (see Section 3.1.1) and management operations (see Section 3.1.2). Finally, it propagates the operations to the rest of agents. In the D2.4 "Agents Goal Table and Condition Action Rules" Section 3.5 numerous agent needs, beliefs, goals, plans and events which are essential to carry out the dispatching, were presented. The goals of this type of agent can be summarized in: *(i)* creating a Sesame agent to monitor a Sesame server; *(ii)* creating OGC® SOS agent to monitor an OGC® SOS server; *(iii)* creating OGC® WPS agent to monitor an OGC® WPS server; *(iv)* recovering the available logical models; *(v)* recovering the logical model structure; *(vi)* recovering the FOIs; *(vii)* recovering the observations; *(viii)* recovering an observation; *(ix)* recovering the available processes; *(x)* recovering a process description; and *(xi)* executing certain process.

The **Sesame agent** is responsible for supervising the Sesame servers which contain the WatERP ontology and the semantic instances for each of the cases of study (pilots). Hence, this agent is capable of managing the connections of the Sesame server. Additionally, this kind of agent provides the required ontological knowledge for the rest of the modules when it is required. Summarizing, this type of agents are able to obtain: *(i)* the available logical models; *(ii)* the logical model structure; *(iii)* the features of interest for a water resource; and *(iv)* the observations of a water resource or feature of interest.

The **OGC® SOS agent** is responsible for supervising the OGC® SOS servers which contains the data of the observations in the WatERP framework (See D3.2 section 2.3.1). These agents are aimed at providing the information of different observations for the rest of the platform. Mainly, the services

offered by the OGC® SOS agent are: *(i)* recover data observation (see D7.2.1 "Implementation of MAS" Section 3.2) and *(ii)* recover the data window for an observation (see future D7.2.3 "Implementation of MAS").

The agent responsible for managing the OGC® WPS servers (**OGC® WPS agent**) is aimed at analysing the integrated OGC® WPS services to classify these services and delegate the responsibility to the most suitable agent (HF, DMS or DSS) in order to manage these services. (See D7.2.2 "*Implementation of MAS*" in Section 3.1).

Finally, the **specific agents (HF agent, DMS agent and DSS agent)** are the agents responsible for monitoring and managing an OGC® WPS server once this server has been classified by OGC® WPS agent. Basically, their services are:*(i)* subscribe/unsubscribe processes to the Process Knowledge (see D7.2.2 *"Implementation of MAS"* in Section 2.4.1 and 2.4.2 and D2.4 *"Agents Goal Table and Condition Action Rules"* in Section 3.4 Figure 17); and *(ii)* execute processes (see D7.2.2 *"Implementation of MAS"* in Section 2.4.3, 2.4.4, 2.4.5 and 2.4.6 and D2.4 *"Agents Goal Table and Condition Action Rules"* in Section 3.4 Figure 18).

3.2.2 WatERP Process Knowledge Service

Initially, yellow pages agent (also named as Directory Facilitator (DF) on Jadex Platform) was proposed as a solution to carry out part of the WatERP orchestration by providing the most suitable service to gather an observation or process (See D7.2.1 *"Implementation of MAS"* in Section 4.3). Mainly, these yellow pages were based on DF that allows the agents to publish the service descriptions. Moreover, the DF also permits to perform searches over the agents with the aim of facilitating the discovery of other agents. However, during the implementation of the yellow pages, major shortcomings were detected. These shortcomings refer to restricted stored information of the service (service name, service type and service ownership) and limited search functionalities (based on the matching of string patterns). In detail, the stored information and search functionalities are not enough to orchestrate the integrated services due to the characterization of a process in a string can be a complex task and the matching of string patterns normally is inefficient and almost unusable (See D7.2.2 *"Implementation of MAS"* in Section 4). Based on this, the yellow pages implementation was discarded and a new approach was adopted. The new approach was based on a *"Process Knowledge"* service to support the orchestration. This new service is capable of representing and defining each of the processes published by a certain agent.

Hereafter, *"Process Knowledge"* service is usable by the agents through its facade presented in Figure 6. Basically, it exposes two types of operations: management and evaluation. The management operations are: *(i)* "*init"* to initialize the Façade with the URL of the ontology server and the repository name where *"Process Knowledge"* is stored; *(ii)* "*insertProcesses"* to add new processes to the "*Process Knowledge*" during OGC® WPS integration (See Section 4.1) and *(iii) "deleteProcesses"* to delete processes of unregistered OGC® WPS servers. Referring the evaluation operations, the

published operations used to evaluate the processes are: *(i)* "*hasDependence"* to evaluate if a process previously requires the execution of other processes or observations; *(ii)* "getProcessesDependence" that provides how the dependences of the process should be executed.

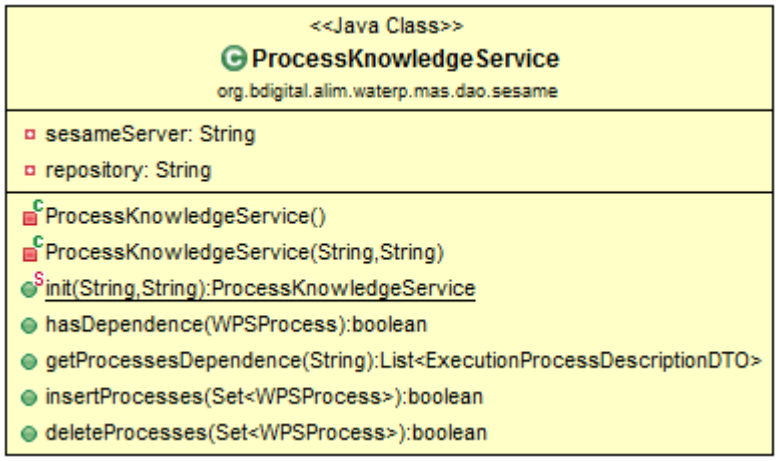

Figure 6: "Process Knowledge facade"

Internally, *"Process Knowledge"* service is based on the service-domain knowledge which is represented through a standard ontology as OWL-S[1] developed by the W3C (see D2.4 "Agents Goal-Table and Condition Action Rules" in Section 2.3). This ontology stores the service processes knowledge and requirements of the MAS. Thanks to make the OWL-S accessible, all the agents minimise the communication overhead during the negotiation by sharing their process in a knowledge-based approach. Therefore, the main aim of OWL-S ontology for the WatERP MAS is to semantically declare and describe services based on three entities as: *(i) ServiceProfile* to define "what the service does" in a way that is suitable for a service-seeking or matchmaking, determining whether the service meets the needs; *(ii) ServiceGrounding* to specify the details of how the service can be accessed (e.g. specify the communication protocol, port numbers, etc.); and *(iii) ServiceModel* that stipulates how to use a service. It is important to note that *ServiceGrounding* and *ServiceModel* entities are not required for the WatERP Process Knowledge because can be assumed that OGC® WPS protocol is used in the WatERP platform. Then, the access service and how to use the service is well-known due to standardization. Hence, it is not need to be represented in the ontology.

[1] http://www.w3.org/Submission/OWL-S/

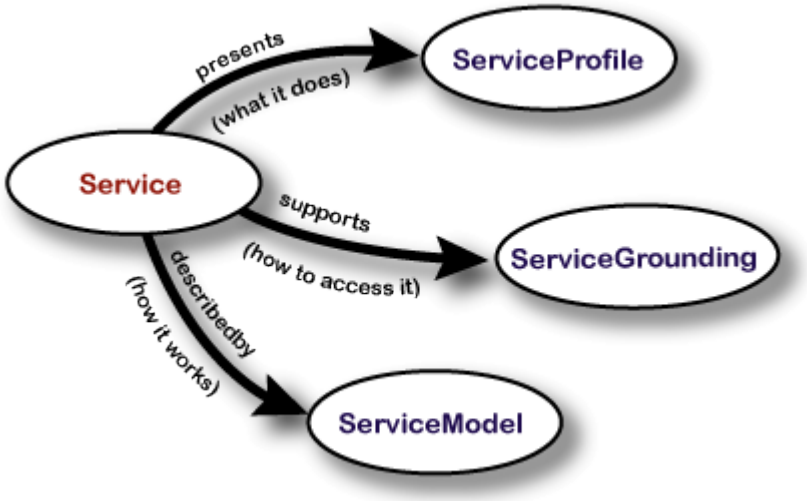

Figure 7: "OWL-S ontology base (source: http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122)"

Below, a more detailed figure of the *ServiceProfile* entity is shown (see Figure 8). In this figure are represented the essential elements to carry out the services orchestration by: *(i)* "*xsd:anyValue"* that corresponds to the input/output definition of the process, allowing any instance to represent the parameter type; *(ii) ParameterType* entity to link concepts with the parameters defined by the OGC® process such as the encoding, mime type, schema and keywords (See Figure 8).

As a conclusion, the WatERP MAS is able to search processes (via "*Service Profile"*) by name and semantically get the required input and output parameters linked with the concepts of the water-domain model represented by the WatERP knowledge base when appropriate.

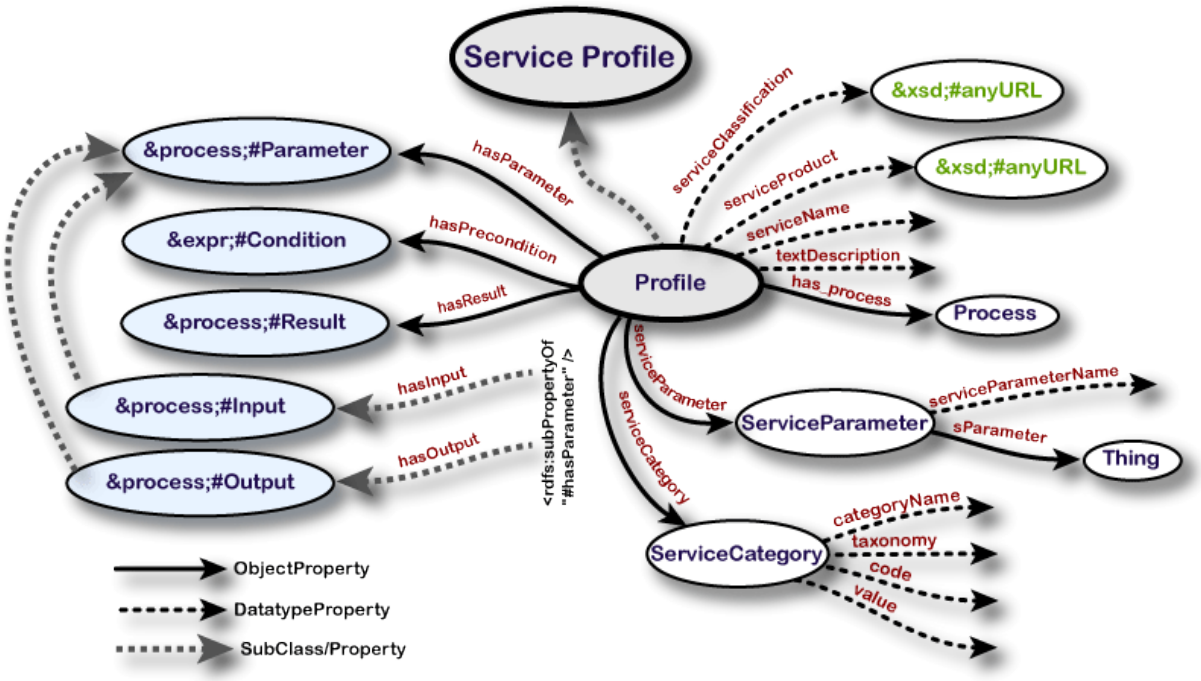

Figure 8: "ServiceProfile entity of OWL-S (source: http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122)"

3.2.3 Directory Facilitator Service

As it was discussed in the previous section, Directory Facilitator (DF) operation has been minimized with respect the first idea presented in D7.2.1 *"Implementation of MAS"* in Section 4.3. Currently, the aim of Directory Facilitator is to provide support during the identification of the responsible agent for managing a service without performing the matchmaking. The Figure 9 presents the class implemented to encapsulate the Directory Facilitator functionalities. Mainly, these operations are used to create and update services in the yellow pages, delete services of the yellow pages and perform searches of these services. Then, the operations implemented are: *(i) search; (ii) create; (iii) modify;* and *(iv) deregister* which has different implementations in order to facilitate their use. These operations are used throughout of the agents' interactions as it is presented in sections 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6

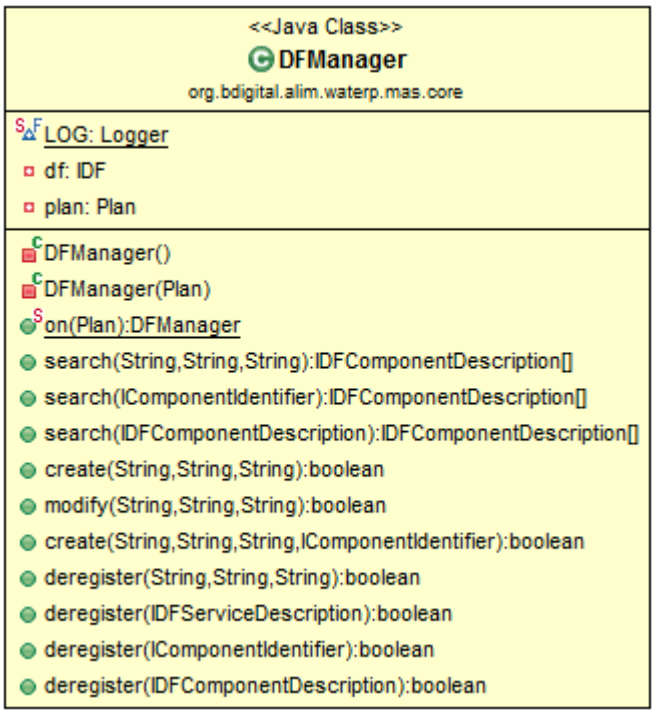

Figure 9: "Directory Facilitator Manager"

DF service information is represented in a table with the following columns: *(i)* name of the registered service; *(ii)* type of the registered service; *(iii)* ownership of the service; and *(iv)* responsible agent of the service.

The ***name of the registered service*** *parameter* depends on the services offered by an agent. For instance, this name is aligned with the operations that each of the services of the WatERP platform can perform.

- Sesame Agent provides services to access to the logical models. Then, logical model identifiers are used in the *name* of the service. Therefore, for each logical model in the sesame server, there is an available service in the DF.
- SOS agent provides services to access the data regarding the observations which this agent supervises. So, the *name* of the service fits with the observation identifier.
- The services published by the HF, DMS and DSS agents are WPS processes. Hence, the *name* contains the process identifier.

The ***type parameter*** of the registered service is used to reflect the service type such as: (i) "*process"* for the processes of the DMS, DSS and HF agents, (ii) "*lm"* for the logical models of the Sesame server and finally, (iii) *"observation"* for the observations of the OGC® SOS server.

The ***ownership parameter*** defines the "owner" of the service. In case of the WatERP project, the main owner of the services is the whole platform. Therefore, this parameter contains the default key *"WatERP"* indicating the origin of the service.

The ***agent responsible for the agent service*** is automatically added by the DF service and it is useful to carry out the matching between service and agent.

Regarding these functionalities, three examples are presented to demonstrate the use of the DFManager class. These examples correspond to (i) register a service in the Directory Facilitator; (ii) delete a service of the Directory Facilitator; and (iii) search a service in the Directory Facilitator.

The "*register service operation"* (Listing 34) aims at registering a service in the DF. To perform this operation, three parameters are required: *(i) name* of the registered service; *(ii)* the *type* of the registered service and *(iii) ownership* of the service. Firstly, the DF is initiated using the static method named as "*on".* Then, the description of the service is created using the "*create*" method defined for the DF. This method needs for the definition of the *service name*, the *type* and the *ownership* in order to register the service. In this case, "*Obs_DW-MP2_Flow1Hour_SensorMeasured*" is used like *service name* because it identifies the offered service, "*observation*" like *type* because the offered service relies on returning data observations and finally, "*WatERP*" like *ownership* because it is the owner of the service.

```
boolean registered =
    DFManager
        .on(this)
        .create("Obs_DW-MP2_Flow1Hour_SensorMeasured", "observation", "WatERP");
```

Listing 34: "Create a Directory Facilitator service"

"Delete operation" (see Listing 35) follows the same procedure as depicted in the register service operation. Then, required parameter types to execute this operation are: *name*, *type* and *ownership*. To unregister a service, firstly the DFManager is initiated using the current plan through "*on"* static method. After that, the *deregister* method is invoked with *service name*, the *type* and the *ownership.* In the example, the unregistered service is the responsible for managing the SWKA pilot instantiation. Therefore, service name contains pilot instantiation name ("*SWKAWaterDistributionModel*"), and type ("*lm*") to identify that the service supervises a logical model.

```
boolean registered =
    DFManager
        .on(this)
        .deregister("SWKAWaterDistributionModel", "lm", "WatERP");
```

Listing 35: "Delete a Directory Facilitator service"

The last operation offered by the DF is the "*search operation*" (see Listing 36). This operation allows the DF to search available service operations inside the registered list of services. For instance, search operation looks for all available process in the DF that fulfil certain "*type*" (e.g. "*process*") and "*ownership*" parameters (e.g. "*WatERP*").

```
IDFComponentDescription []idf =
    DFManager
        .on(this)
        .search(null, "process", "WatERP");
```

Listing 36: "Search the available processes on the WatERP framework"

3.2.4 Agent Management Service

The **Agent Management Service (AMS)** component facilitates the usage of the white pages of Jadex platform. Mainly, it provides the functionalities for: *(i)* creating new agents inside the Jadex platform which is essential during integration tasks such as the incorporation of OGC® WPS, OGC® SOS servers or logical models; *(ii)* destroying agents from the Jadex platform using the unregistering tasks and eliminating the agents that are responsible for managing the unresgistered services; and *(iii)* search an agent inside the Jadex platform in order to know the address to communicate with the agent.

The Figure 10 presents the class implemented to encapsulate the Agent Management Service functionalities. Basically, the operations implemented are: *(i) search; (ii) create; and (iii) destroy.* Furthermore, these operations are used throughout the agents' interactions as it is presented in sections 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6. It is important to note that this class use internally "*cmscap"* capabilities provided by the Jadex platform which are imported using the file *jadex.bdi.planlib.cms.CMS*.

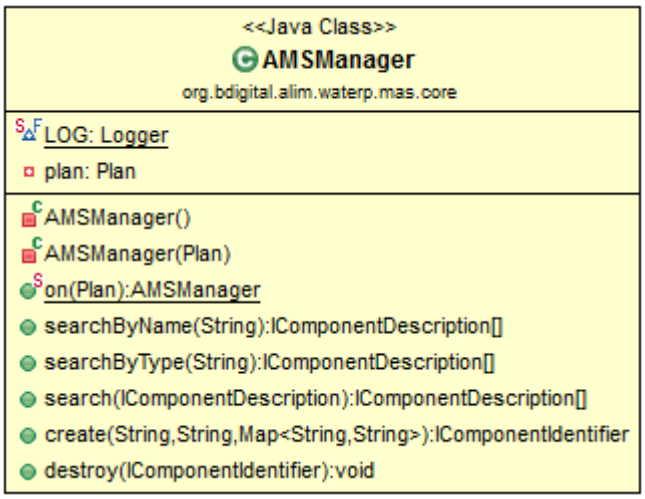

Figure 10: "Agent Management Service Manager"

In order to demonstrate the performance of the AMS, some examples that use the main methods of the AMSManager class are presented: *(i)* create an agent; *(ii)* destroy an agent; and *(iii)* search an agent.

To "*create an agent"* using the AMSManager, three parameters are required: *(i)* name of the agent to be created; *(ii)* agent type, that is, the qualified name to reference an ADF description file inside the project (e.g. org/bdigital/alim/waterp/mas/core/agent/wps/wps.agent.xml), has to be defined; and *(iii)* arguments (e.g., name-value) has to be included. In the WatERP framework, the URL of the integrated server is used as an agent name due to the URL permits to identify uniquely an element inside the architecture. Furthermore, the URL also permits to check if a server is already registered in the WatERP platform by comparing the availability of the URL in the AMSManager.

Listing 37 depicts the instantiation of the OGC® WPS agent at execution time using it URI as name and providing the initialization parameters through the *args* variable. Internally, this class use *cmscap* capabilities executing the goal *cms_create_component* of the Jadex platform.

```
Map<String, String> args = new HashMap<String, String>();
args.put("url", urlSesame);

IComponentIdentifier ci =
    AMSManager
        .on(this)
        .create(
            name,
            "org/bdigital/alim/waterp/mas/core/agent/wps/wps.agent.xml",
            args);
```

Listing 37: "Create agent with Agent Management Service"

"Destroy operation" is used to delete and unregistered agents from the Jadex platform. Mainly, agents monitor a specific server in order to know the operations and functions that can be performed in the WatERP platform. Indeed, the Listing 38 depicts how to destroy an agent using AMSManager class. The *destroy* method only requires the component identifier of the agent to be destroyed which can be provided by the search operation of the same class (see Listing 39). Internally, *destroy* method takes advantage of the "*cms_destroy_component*" goal provided by the *cmscap* capabilities.

```
AMSManager
    .on(this)
    .destroy(componentIdentifier);
```

Listing 38: "Destroy agent with Agent Management Service"

Latter, *"Search* operation" permits to find an agent and checks if the agent found is still active. In case of the agent is active in the architecture, the search operation also returns the component description to access the agent. For this operation, two different methods are provided to search. One method is looking for agents by name and the other is focused on looking for agents by type. Listing 39 depicts the procedure to perform a search by name using *AMSManager* class. In order to perform this operation, the only requirement is the definition of the name of the agent to be searched. Internally, this method taking advantage of the "*cms_search_component*" goal provided by the *cmscap* capabilities.

```
IComponentDescription []cd =
    AMSManager
        .on(this)
        .searchByName(name);
```

Listing 39: "Search an agent with Agent Management Service"

3.3 External Access layer

External access layer is focused on providing the needed tools to access the external components which are used by the WatERP platform to complement the decision-making. Basically, the external components are: *(i)* OGC® SOS servers; *(ii)* OGC® WPS servers and *(iii)* Sesame servers. The following sections presents the implemented classes used to access to these external components.

3.3.1 SOS Manager

SOS Manager class is the responsible for handling the OGC® SOS access by WatERP MAS. Its implementation has been done directly using the Apache Commons[2] instead of "wsdl2java" tools (e.g. apache-cxf, axis2, etc.) due to the WSDL complexity that defines the OGC® SOS web services which generates namespaces incompatibilities during the schema import. Figure 11 presents the SOS Manager class and its operations which are: *(i) getAvailableObservations*; *(ii) getObservation*; *(iii) getObservationWithFilter*; and *(iv) getDataWindow*.

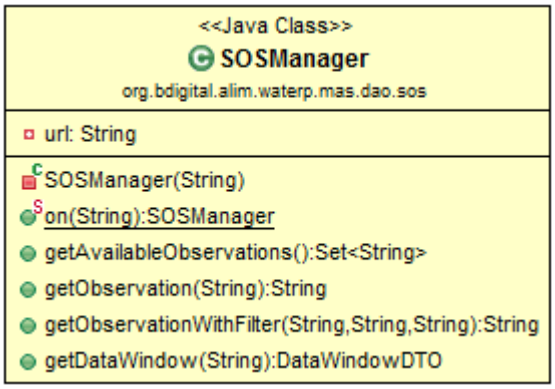

Figure 11: "SOS Manager class"

SOSManager is initiated by the static method "*on*" by providing it the URL of the OGC® SOS Server. Once the SOSManager is initiated, the instance of the class allows this class to invoke the different non-static methods.

The *getAvailableObservations* method is used to obtain the observations that are published by the supervised OGC® SOS server. Internally, this method invokes "*getCapabilities*" operation of the OGC® SOS server with the SOAP-XML message (Listing 40).

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-envelope"
    xmlns:sos="http://www.opengis.net/sos/2.0"
    xmlns:ows="http://www.opengis.net/ows/1.1">
  <env:Body>
    <sos:GetCapabilities
        xmlns:sos="http://www.opengis.net/sos/2.0"
        xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.opengis.net/sos/2.0
http://schemas.opengis.net/sos/sosGetCapabilities.xsd"
        service="SOS">
      <ows:AcceptVersions>
        <ows:Version>2.0.0</ows:Version>
      </ows:AcceptVersions>
```

[2] http://commons.apache.org/

```
      <ows:Sections>
        <ows:Section>OperationsMetadata</ows:Section>
      </ows:Sections>
    </sos:GetCapabilities>
  </env:Body>
</env:Envelope>
```

Listing 40: "SOAP message to invoke getCapabilities operation of OGC® SOS server"

The *getObservation* method is used to obtain time series of an observation. Internally, this method invokes *getObservation* operation of the OGC® SOS server with the SOAP-XML message (Listing 41). In this example, the "*##URI##*" tag is changed by the identifier of the observation to be retrieved. It is important to note that its usage is discouraged due to huge amount of data to be transferred (from the first to the last data). Therefore, the usage of the *getObservationWithFilter* method is recommended due to remedy the problem.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-
envelope">
  <env:Body>
    <sos:GetObservation
        xmlns:sos="http://www.opengis.net/sos/2.0"
        xmlns:fes="http://www.opengis.net/fes/2.0"
        xmlns:gml="http://www.opengis.net/gml/3.2"
        xmlns:swe="http://www.opengis.net/swe/2.0"
        xmlns:swes="http://www.opengis.net/swes/2.0"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/sos.xsd"
        service="SOS" "
        version="2.0.0">
      <sos:procedure>
        ##URI##"
      </sos:procedure>
      <sos:responseFormat>http://www.opengis.net/waterml/2.0</sos:responseFormat>
    </sos:GetObservation>
  </env:Body>
</env:Envelope>
```

Listing 41: "SOAP message to invoke getObservation operation of OGC® SOS server"

The *getObservationWithFilter* method is used to obtain time series of an observation from one to another. Internally, this method invokes *getObservation* operation of the OGC® SOS server adding temporal filter node. Listing 41 presents the used SOAP-XML message where "*##URI##*" tag is changed by the identifier of the observation to be retrieved, the "*##SDATE##*" tag by the initial filter date and the "*##EDATE##*" tag by the end filter date.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-
envelope">
  <env:Body>
    <sos:GetObservation
        xmlns:sos="http://www.opengis.net/sos/2.0"
        xmlns:fes="http://www.opengis.net/fes/2.0"
```

```
        xmlns:gml="http://www.opengis.net/gml/3.2"
        xmlns:swe="http://www.opengis.net/swe/2.0"
        xmlns:swes="http://www.opengis.net/swes/2.0"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/sos.xsd"
        service="SOS"
        version="2.0.0">
      <sos:procedure>
        ##URI##"
      </sos:procedure>
       <sos:temporalFilter>
         <fes:During>
           <fes:ValueReference>phenomenonTime</fes:ValueReference>
           <gml:TimePeriod gml:id="tp_1">
             <gml:beginPosition>##SDATE##</gml:beginPosition>
             <gml:endPosition>##EDATE##</gml:endPosition>
           </gml:TimePeriod>
         </fes:During>
       </sos:temporalFilter>
      <sos:responseFormat>http://www.opengis.net/waterml/2.0</sos:responseFormat>
    </sos:GetObservation>
  </env:Body>
</env:Envelope>
```

Listing 42: "SOAP message to invoke getObservation operation with filters of OGC® SOS server"

3.3.2 WPS Manager

WPS Manager Class is the responsible for handling OGC® WPS access by WatERP MAS. In the same way of the SOS manager, the implementation of this class has been based on Apache Commons[3] in order to avoid namespaces incompatibilities generated during schema import. Figure 12 presents the WPS Manager class and the defined operations: *(i) getServiceType*; *(ii) getCapabilities*; *(iii) describeProcess*; *(iv) executeProcess*; *(v) executeAsyncProcess*; *(vi) isExecutedLongProcess; and (vii) getProcessExecution*.

[3] http://commons.apache.org/

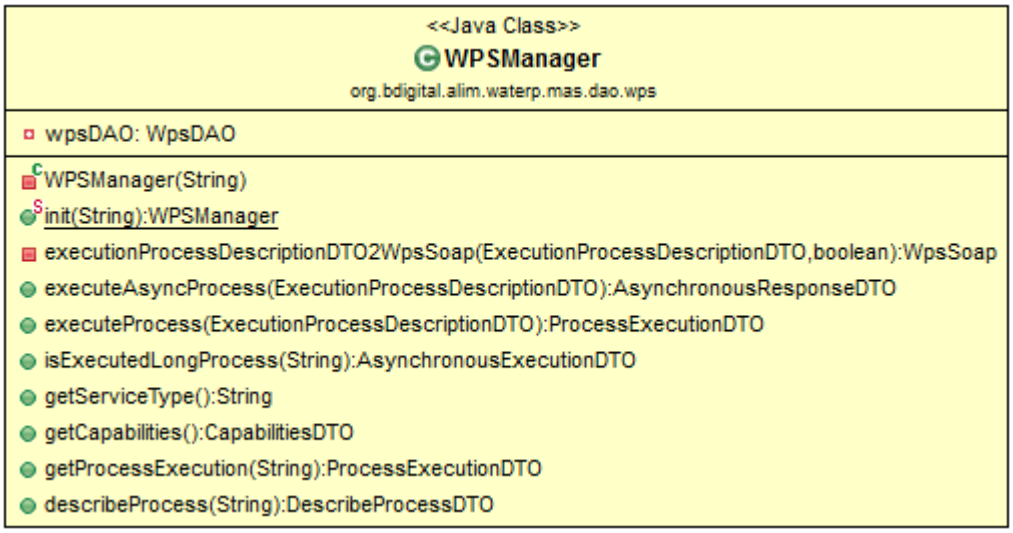

Figure 12: "WPS Manager class"

WPSManager is initiated by the static method called "*init*". For executing this method, an URL of the OGC® WPS Server to be managed is required. Once the WPSManager instance is obtained, the non-static methods can be invoked.

The "getServiceType" method is used to obtain the type of the supervised OGC® WPS server. This information is defined in the in node "*ows:ServiceType"* published in the *ServiceIdentification* document (See D2.3 *"Open Interface Specification"* in Section 3.2.2). This service type is used to integrate tasks between OGC® WPS servers in order to invoke the most suitable agent for managing the OGC® WPS server as is shown in the section 4.1.

```
<ows:ServiceIdentification>
          <ows:Title>DSS Service</ows:Title>
          <ows:Abstract>DSS Service</ows:Abstract>
          <ows:ServiceType>DSS</ows:ServiceType>
          <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
</ows:ServiceIdentification>
```

Listing 43: "Type server definition on Service Identification document"

The second method, "*getCapabilities",* invokes the SOAP-XML request presented in Listing 44. This method is used to recover all available processes in the OGC® WPS server to integrate different tasks (see section 4.1)

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-envelope"
    xmlns:wps="http://www.opengis.net/wps/1.0.0"
    xmlns:ows="http://www.opengis.net/ows/1.1">
  <soap:Header/>
  <soap:Body>
    <wps:GetCapabilities
        service="WPS"
        xmlns:wps="http://www.opengis.net/wps/1.0.0"
        xmlns:ows="http://www.opengis.net/ows/1.1">
```

```
      <wps:AcceptVersions>
        <ows:Version>1.0.0</ows:Version>
      </wps:AcceptVersions>
    </wps:GetCapabilities>
  </soap:Body>
</soap:Envelope>
```

Listing 44: "SOAP call to getCapabilities"

Other method invoked during integration tasks is the "*describeProcess"* which provides information about how a process should be invoked. Mainly, this method is based on the SOAP-XML request presented in the Listing 45.

```
<soap:Envelope
     xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
     xmlns:wps="http://www.opengis.net/wps/1.0.0"
     xmlns:ows="http://www.opengis.net/ows/1.1"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
http://schemas.opengis.net/wps/1.0.0/wpsDescribeProcess_request.xsd">
 <soap:Header/>
   <soap:Body>
     <wps:DescribeProcess
         service="WPS"
         version="1.0.0">
       <ows:Identifier>process_identifier</ows:Identifier>
   </soap:Body>
 </soap:Envelope>
```

Listing 45: "SOAP call to describeProcess"

As was presented in the D2.3 in Section 1.1, the execution of the processes can take huge amount of time (long-processes). Hence the execution of long-processes using asynchronous communication must be supported. Then, the methods to execute processes are divided in two groups, synchronous and asynchronous requests. To execute synchronous request, a single method is available (*executeProcess*) which only requires a description of the process execution to be executed. On the other hand, to perform the asynchronous execution three methods are required. One method is used to throw the asynchronous execution (*executeAsyncProcess*), other to check the finalisation of the execution (*isExecutedLongProcess*), and the last method, *getProcessExecution*, to retrieve the output of the asynchronous execution.

The asynchronous process execution is based on the OGC® WPS implementation to perform the asynchronous request. Therefore, *storeExecuteResponse* attribute of the SOAP-XML request is modified to true Boolean value.

```
. . .
<wps:ResponseForm>
    <wps:ResponseDocument storeExecuteResponse="true">
        . . .
    </wps:ResponseDocument>
</wps:ResponseForm>
. . .
```

Listing 46: "Asynchronous process execution"

3.3.3 Water Ontology Manager

Water Ontology class is the responsible for handling the interaction with WatERP Ontology. Its implementation is based on Openrdf[4] library which is a common framework to process RDF. Furthermore, this library permits to connect and manage repositories and semantic information for a Sesame repository[5]. By using this library, the Water Ontology Manager is aimed at gathering the semantic information of the WatERP repository using SPARQL endpoint. Figure 13 presents the Water Ontology Manager class and its methods which are: *(i) getLogicalModels*; and *(ii) getLogicalModel*.

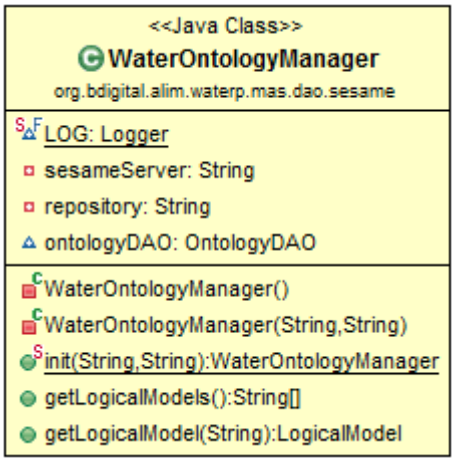

Figure 13: "Water Ontology Manager Class"

In the same way of previous manager classes, WaterOntologyManager is initiated by the static method "*init"* by providing the URL of the Sesame repository and also the repository name. After the instantiation of the class, its non-static methods are callable. Below, these non-static methods are presented.

The "getLogicalModels" method is used to obtain identifiers of the available logical models. Basically, this method is based on the invocation of the SPARQL query presented in the Listing 47. In order to perform the query, this method establishes a connection with the Sesame repository and then, executes the SPARQL query. As a result, this method encapsulates the response in a strings array.

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX geo:<http://www.opengis.net/ont/geosparql#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX WatERPOntology:<http://www.waterp-fp7.eu/WatERPOntology.owl#>
SELECT ?lm WHERE { ?lm a WatERPOntology:WaterResourceManagement. }
```

[4] http://www.openrdf.org/

[5] http://www.w3.org/wiki/SemanticWebTools

Listing 47: "SPARQL to obtain the logical models identifiers"

The other non-static method, *getLogicalModel*, retrieves the full structure of a logical model, filtering the logical model by identifier. In order to obtain the full logical model, the manager executes the SPARQL construct shown in Listing 48. Similarly as mentioned in the above example, the manager establishes a connection with the sesame repository and executes the query. As a result, this query returns a set of triples. This gathered triples are parsed to java beans and are stored in the agent cache in order to minimize the runtime communication overhead (See D2.4 "Agents Goal-Table and Condition Action Rules" in section 3.1).

```
CONSTRUCT {
  ?entityModel
    WatERPOntology:id ?id ;
    WatERPOntology:name ?wrName ;
    WatERPOntology:description ?wrDescription ;
    a ?type ;
    WatERPOntology:hasLogicDependence ?entityModelDependence ;
    WatERPOntology:hasObservation ?wrObservation .
  ?wrObservation WatERPOntology:name ?wrObservationName .
  ?wrObservation WatERPOntology:description ?wrObservationDescription .
  ?wrObservation WatERPOntology:hasProcedure ?wrObservationProcedure .
  ?wrObservation WatERPOntology:hasPhenomenon ?wrObservationPhenomenon .
  ?entityModel WatERPOntology:hasFeature ?foi .
  ?foi WatERPOntology:name ?foiName .
  ?foi WatERPOntology:description ?foiDescription .
  ?foi WatERPOntology:hasObservation ?observation .
  ?observation WatERPOntology:name ?observationName .
  ?observation WatERPOntology:description ?observationDescription .
  ?observation WatERPOntology:hasPhenomenon ?phenomenon .
  ?observation WatERPOntology:hasProcedure ?procedure .
  ?foi geo:hasGeometry ?location .
  ?location WatERPOntology:id ?idloc .
  ?location geo:asGML ?point .
  ?location WatERPOntology:srsName ?srsName .
}
WHERE {
  ?lmodel WatERPOntology:isModelOf ?entityModel .
  FILTER (?lmodel = WatERPOntology:##lm##) .
  ?entityModel a ?type .
  ?entityModel WatERPOntology:id ?id .
  ?entityModel WatERPOntology:name ?wrName .
  ?entityModel WatERPOntology:description ?wrDescription .
  OPTIONAL { ?entityModel WatERPOntology:hasObservation ?wrObservation .
    {
       ?wrObservation WatERPOntology:name ?wrObservationName .
       OPTIONAL {
         ?wrObservation WatERPOntology:description ?wrObservationDescription .
       } OPTIONAL {
         ?wrObservation WatERPOntology:hasProcedure ?wrObservationProcedure .
       } OPTIONAL {
         ?wrObservation WatERPOntology:hasPhenomenon ?wrObservationPhenomenon .
       }
    }
  }
  {
    ?entityModel WatERPOntology:hasLogicDependence ?entityModelDependence .
  }
  UNION
  {
    ?entityModel WatERPOntology:hasFeature ?foi .
    {
      ?foi WatERPOntology:name ?foiName .
      ?foi WatERPOntology:description ?foiDescription .
```

```
      ?foi WatERPOntology:hasObservation ?observation.
      OPTIONAL {
        ?observation WatERPOntology:name ?observationName .
      } OPTIONAL {
        ?observation WatERPOntology:description ?observationDescription .
      } OPTIONAL {
        ?observation WatERPOntology:hasProcedure ?procedure .
      } OPTIONAL {
        ?observation WatERPOntology:hasPhenomenon ?phenomenon .
      }
    }
  }
  FILTER (?type = WatERPOntology:Transport
    || ?type = WatERPOntology:Sink
    || ?type = WatERPOntology:Storage
    || ?type = WatERPOntology:Source
    || ?type = WatERPOntology:Transformation
  ) .
}
```

Listing 48: "SPARQL to obtain a logical model structure"

4. Component interaction

Once the MAS components have been introduced in previous section, this section is focused on presenting the interactions between components in the most common use cases. Firstly, Figure 14 presents a summary of these interactions. Importantly some components such as SOS Manager, WPS Manager, Water Ontology Manager and WatERP Process Knowledge Service always interact with the same kind of agents. For instance, SOS Manager always interacts with OGC® SOS Agent, Water Ontology Manager with the Sesame Agent, and WPS Manager with OGC® WPS agent, HF agent, DMS agent and DSS agent, etc. The main reason for this kind of interactions is that the operations provided by these components are very restricted to a particular case i.e. access to OGC® WPS server. Instead, the usage of Directory Facilitator Service, agent Management Service and Directory Facilitator service is more transversal due to their generic operations such as create an agent, search a service, etc. It is also important that all intercommunication between interface layer and agents is centralized through Gateway agents.

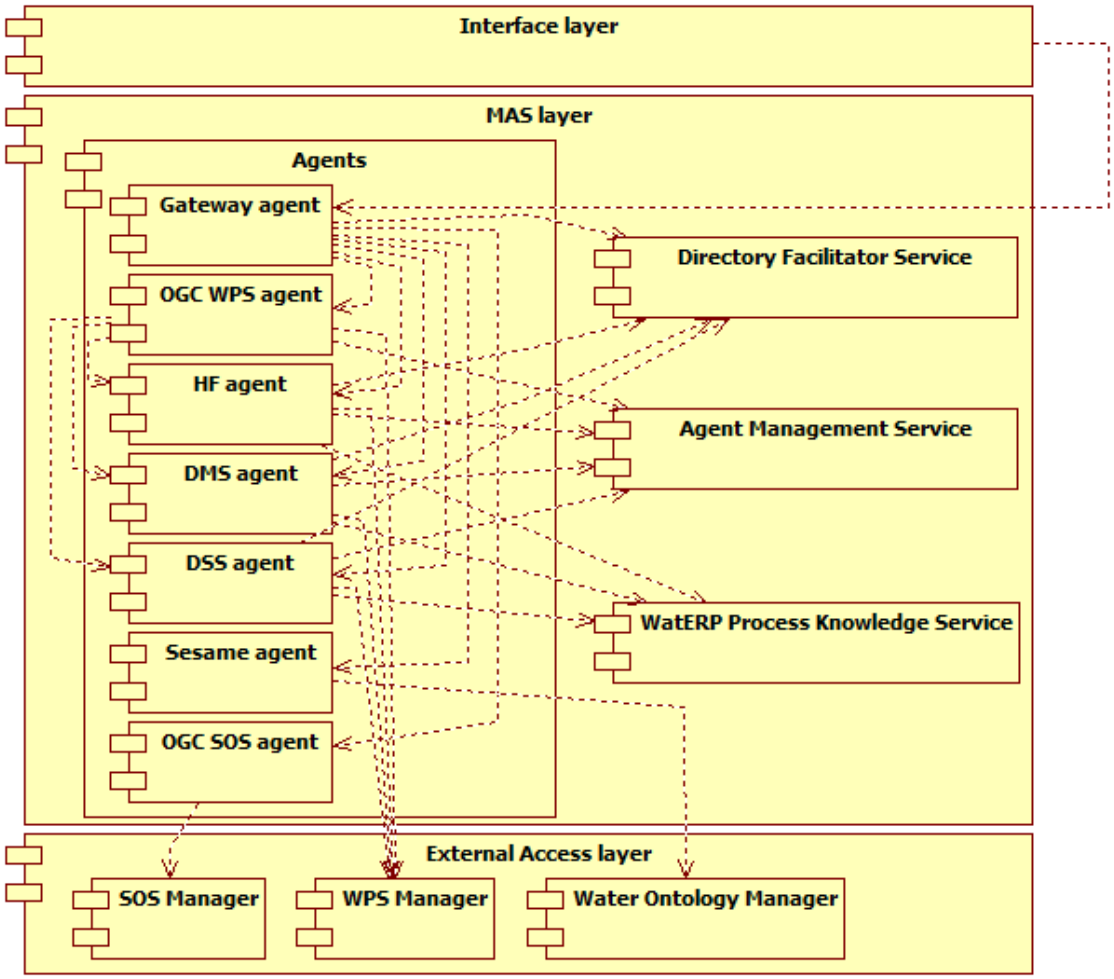

Figure 14: "Multi-agent architecture components diagram"

Below, a set of sequence diagrams are depicted in the following subsections presenting the most common use cases. Basically, they are: *(i)* register OGC® WPS server; *(ii)* register OGC® SOS server; *(iii)* query an observation; *(iv)* execute a synchronous process; *(v)* execute an asynchronous process; and (vi) query a logical model.

4.1 Component interaction for Integrating a new OGC® WPS server

This section exemplifies the interaction between WatERP MAS components when a new OGC® WPS server is registered through management operations (see Section 3.1.1.2).

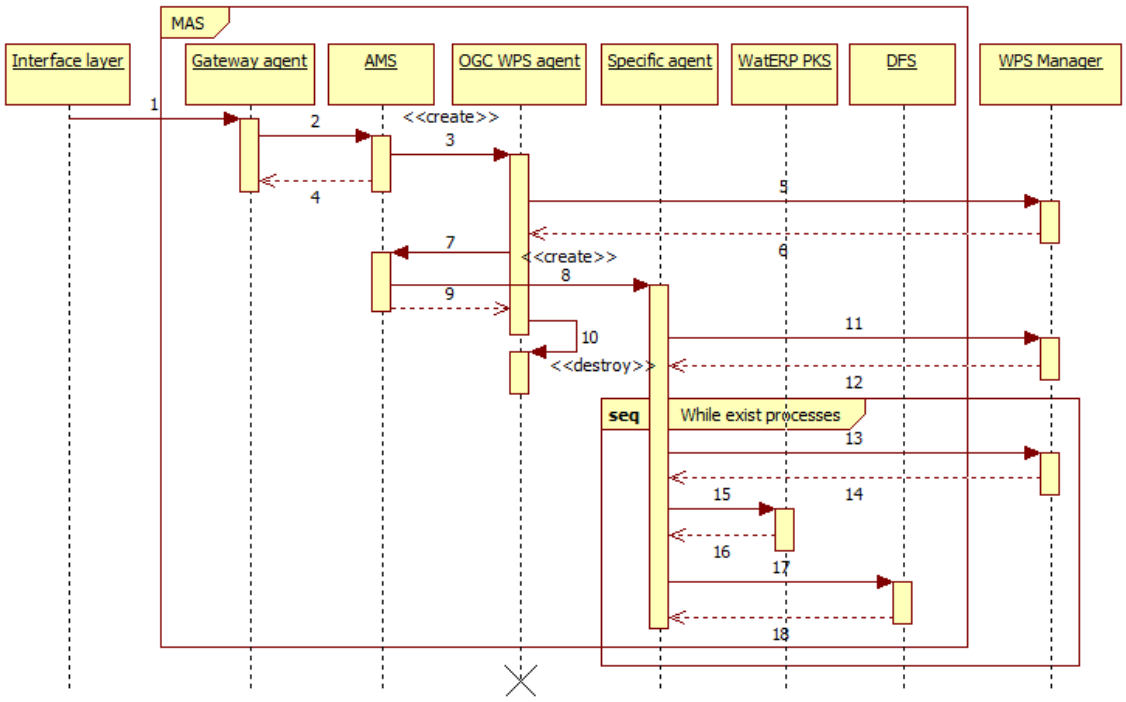

Figure 15: "Integrate a new OGC® WPS server sequence diagram"

The process is initiated from Interface layer throwing *register_wps_server* goal (1) which is interpreted by the Gateway agent (see D2.4 “Agents Goal-Table and Condition Action Rules” in Section 3.5). Then, Gateway agent creates a new OGC® WPS agent (3) to manage the register of the OGC® WPS server using Agent Management Service (2) (see Section 3.2.4). Once the OGC® WPS agent is created, it uses the WPS Manager (5, 6) (see Section 3.3.2) to acquire the OGC® WPS Server type (See section 3.3.2). The server type is used by the OGC® WPS agent in order to decide the new instantiation of the specific agent (DSS, DMS or HMF) which is carried out again through Agent Management Service (7, 8, 9) (see Section 3.2.4). After a more specific agent instantiation, the OGC® WPS agent is auto destroyed (10) (see Section 3.2.4) because the monitoring tasks have been delegated to the specific agent. In parallel, the new agent starts the analysis of the OGC® WPS server through WPS Manager obtaining the available process in the server (11, 12) (see Section 3.3.2). Finally, the specific agent uses the WPS Manager to evaluate each process separately by invoking the *describe_process* operation (13, 14) (see Section 3.3.2). The result of each request is processed and registered in the WatERP Process Knowledge Service (15, 16) and Directory Facilitator Service (17, 18) (see Section 3.2.3).

4.2 Component interaction for Integrating a new OGC® SOS server

This section presents the interaction between WatERP MAS components when a new OGC® SOS server is registered through management operations (see Section 3.1.2).

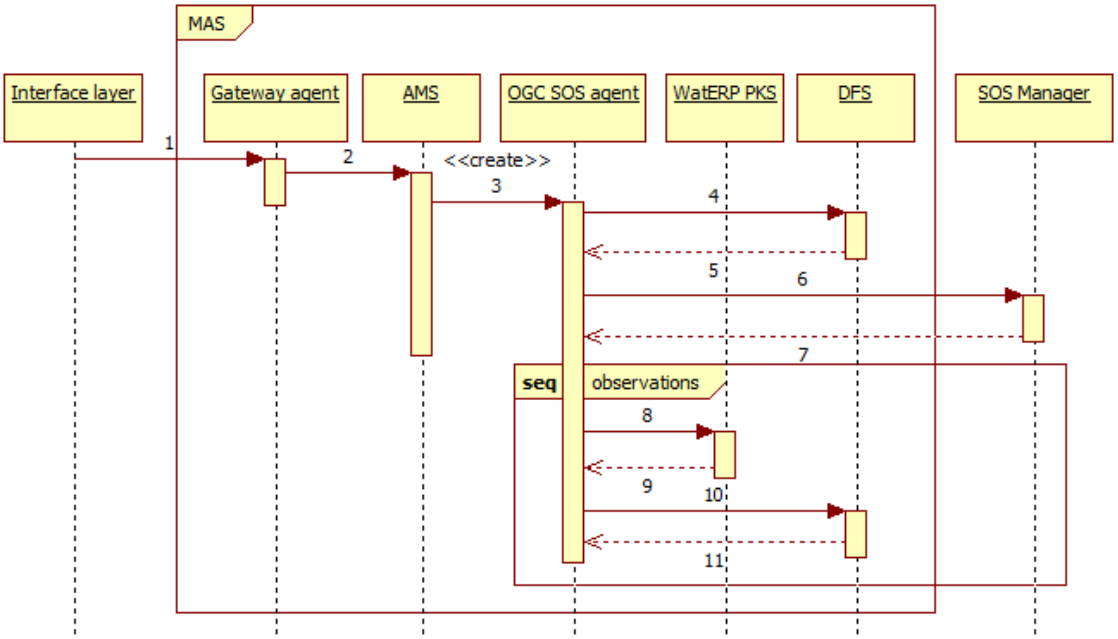

Figure 16: "Integrate a new OGC® SOS server sequence diagram"

In the same way as the previous sequence diagram, the process is initiated from Interface layer. However, the difference relies in the goals thrown that in this case refer to a *register_sos_server* goal (1). This goal is captured by the Gateway agent (see D2.4 “Agents Goal-Table and Condition Action Rules” in Section 3.5). Then, the Gateway agent uses the AMS (2) to create a new OGC® SOS agent (3) in order to manage the register of the OGC® SOS. Once the OGC® SOS agent is created (see Section 3.2.4), the mentioned agent starts discovering and registering tasks in parallel. Firstly, the OGC® SOS Agent registers tasks as an observation service in the DFS (4, 5). After, the OGC® SOS Agent uses SOS Manager (6, 7) to invoke the *getCapabilities* operation (see Section 3.3.1) towards acquiring all available observations of the OGC® SOS Server. Once all available observations are known, the OGC® SOS agent registers each observation in the WatERP Process Knowledge (8, 9) (see Section 3.2.2). These observations are used during orchestration and also are registered as service in the DFS (10, 11) (see Section 3.2.3) in order to facilitate subsequent observation-agent pairing.

4.3 Component interaction for Querying an observation

This section presents the interaction between WatERP MAS components when an observation data is queried (see section 3.1.1.1).

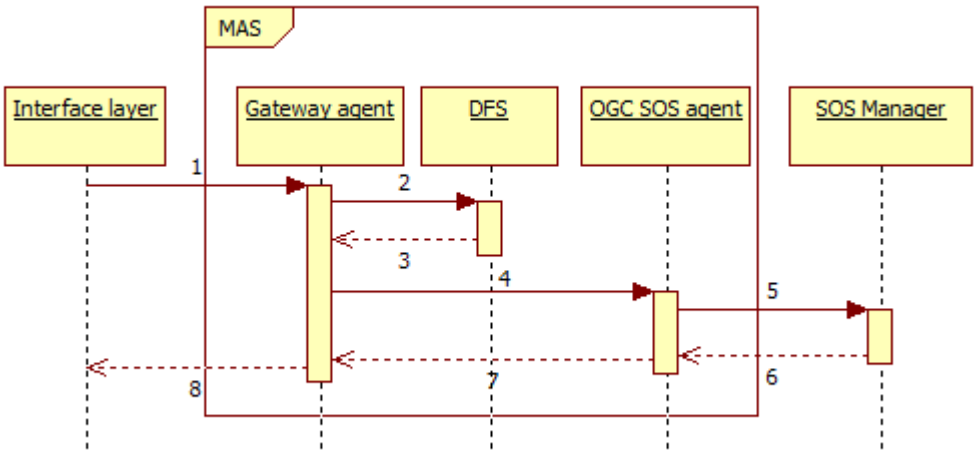

Figure 17: "Query an observation sequence diagram"

The process is initiated from Interface layer by creating an *observation* goal (1) which is managed by the Gateway agent (see D2.4 "Agents Goal-Table and Condition Action Rules" in section 3.5). Then, Gateway agent asks to the Directory Facilitator Service who is the responsible for managing the observation (2, 3) (see Section 3.2.3). Once the Directory Facilitator Service provides the description of the responsible agent, the gateway agents invokes OGC® SOS agent to get data observation (4). Once the OGC® SOS agent receives the invocation using the SOS Manager (6), this agent uplift the data observation that it is supervising to the Gateway Agent and the Interface Layer (see Section 3.3.1) (7,8).

4.4 Component interaction for Executing a synchronous process

This section presents the interaction between WatERP-MAS components that is performed during the execution of a synchronous process (see section 3.1.1.2).

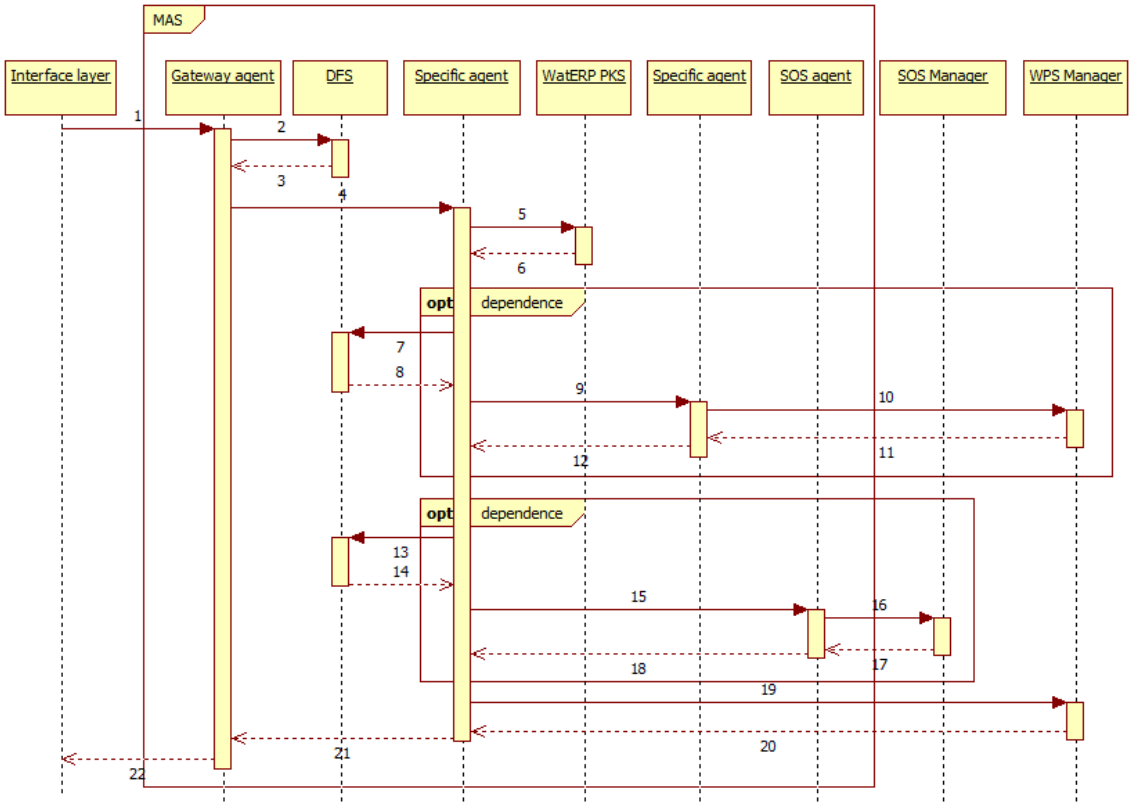

Figure 18: "Synchronous process execution sequence diagram"

Initially, Interface layer inserts an *execute* goal in the MAS. This goal is captured by the Gateway agent (1) (see D2.4 “Agents Goal-Table and Condition Action Rules” in section 3.5). Then, the Gateway agent asks the Directory Facilitator Service for the responsible component in charge of executing the process (2, 3) (see Section 3.2.3) and invoking the responsible agent (4). Once the specific agent is invoked, this agent asks the WatERP Process Knowledge Service (see Section 3.2.2) for the dependencies that the process has defined as for example, observations or outputs of other processes. If this dependencies exists, they are returned to the specific agent (5,6). On the one hand, if the process is dependent of other processes, the specific agent asks the Directory Facilitator Service for the responsible agent of each dependent process (7, 8), invoking each responsible agent (9, 12) to obtain it output. Then, the final output is generated by the specific agent who uses the WPS Manager to execute the process in the corresponding OGC® WPS server (10, 11) (see Section 3.3.2). It is important to note the possibility of generating a chained invocation of agents in order to solve the dependences. On the other hand, if the process is dependent for observations, the specific agent asks the Directory Facilitator Service for the responsible SOS agents to solve the dependence (13, 14). Once these agents are known, the Specific agent invokes the SOS agents in order to get the observations’ data (15, 18). Then, SOS agent uses SOS Manager to recover the observation data (12, 13) in the same way as the

procedure depicted in section 4.3. Finally, the specific agent executes the process through WPS Manager (15, 16) (see Section 3.3.2) once all processes dependences and observations are solved.

4.5 Component interaction for Executing an asynchronous process

This section presents the interaction between WatERP MAS components that is performed during the execution of an asynchronous process (see Section 3.1.1.2).

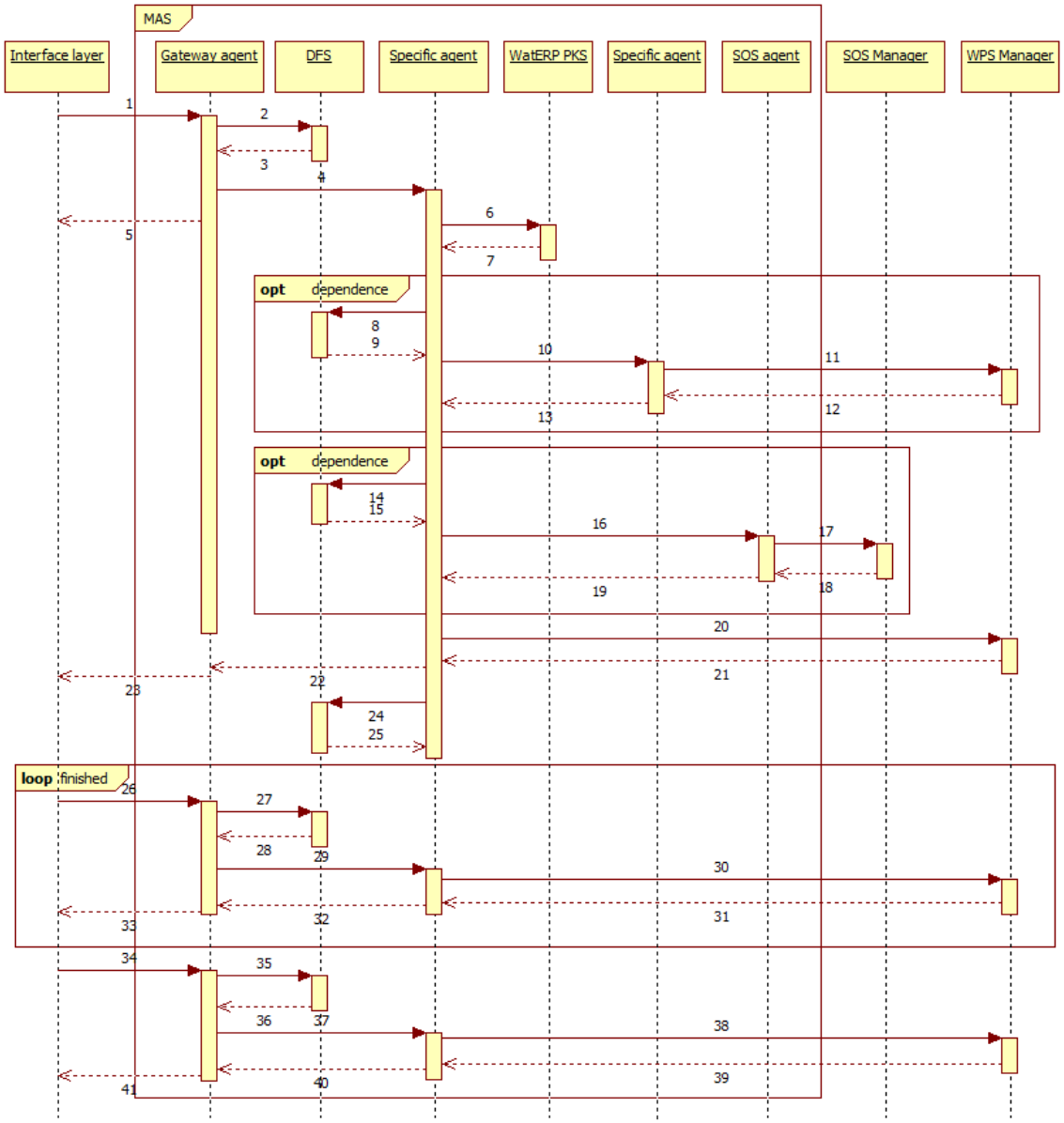

Figure 19: "Asynchronous process execution sequence diagram"

The process starts with the definition of an "*execute_long"* goal in the MAS. This goal definition comes from the Interface layer that indeed is caught by the Gateway agent (1) (see D2.4 "Agents Goal-Table

and Condition Action Rules" in section 3.5). In the same way as the synchronous process execution, the Gateway agent find out for the responsible agent in charge of executing the process, using Directory Facilitator Service (2, 3) (see section 3.2.3). Furthermore, the Gateway agent delegates the process execution to the specific agent (4). At this moment, the specific agent repeats the same procedure used to the synchronous processes (see section 4.4) in order to solve the required dependences (6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19). Once all processes dependences and observations are solved, the specific agent executes the process in an asynchronous mode through WPS Manager (20, 21) (see section 3.3.2). Once the WPS manager returns the process hashcode to the specific agent, this last component send this information to the Interface layer (22, 23). This information is relevant for the Interface layer due to recover the state of the execution (finished or on-going) as well as the asynchronous result. Moreover, this hash process is registered as service in the Directory Facilitator Service with the aim of knowing the pairing agent-long process (24, 25).

The Interface Layer launches *is_executed* goal to evaluate the processes state (26) only at time as this component desire to know the state of the process execution. For that purpose, the Interface Layer initiates a loop that starts with checking the process by requesting it to the Gateway agent. Indeed, the Gateway Agent up-scale the request to the Directory Facilitator Service who is the agent responsible to manage this long process (27, 28). Once the agent is known, it is invoked by the Gateway Agent (29). After the invocation, the specific agent evaluates the state using a WPS Manager request/response (30, 31). At the moment at the specific agent collects the response, this answer is uplifted to the Interface Later, finalising the loop (32, 33). When the checking task is finished ("*is_executed*" goal), the recovering results task is initiated by creating a *recover_long_process* goal in the Interface layer (34), collecting it using the Gateway agent. Then, the Gateway agent asks Directory Facilitator service to the responsible of the long process (35, 26). Once the Directory Facilitator receives the answer, it invokes the responsible to collect the long process output (37, 40) through WPS Manager (38, 39). At time as the Directory Facilitator receive the answer, this is up-scaled to the Interface layer (40, 41) using the Gateway Agent, finalising the whole execution.

4.6 Component interaction for Querying a logical model

This section presents the interaction between WatERP MAS components that is performed during the querying of a specific logical model (see Section 3.1.1.1).

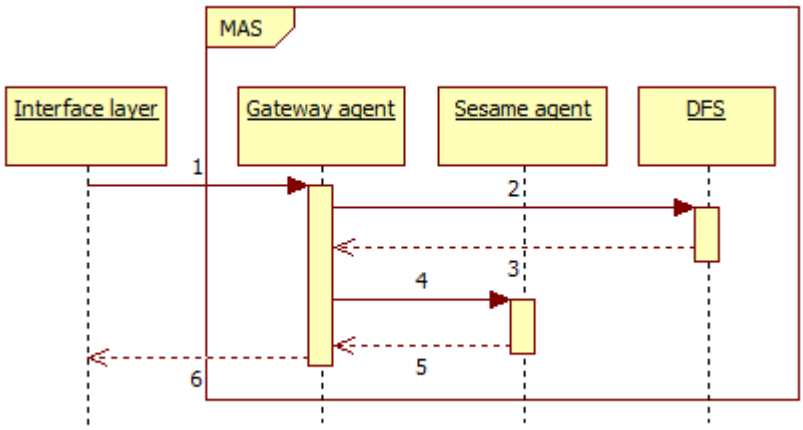

Figure 20: "Querying a logical model sequence diagram"

The process is initiated when the Interface layer insert a *logical_model* goal in the MAS (1) using the Gateway agent (see D2.4 "Agents Goal-Table and Condition Action Rules" in section 3.5). Then, the Gateway agent asks the Directory Facilitator Service (2, 3) (see section 3.2.3) for the Sesame agent responsible for monitoring the specific logical model. Once the responsible agent is known, it is invoked by the Gateway agent in order to recover the logical model (4, 5). This task is solved internally by the Sesame agent using its internal cache which stores a whole model of the logical model in order to minimize overhead communication (see D7.2.1 "Implementation of MAS" in Section 4.2.2). Once the Gateway Agent receives the logical model, this component sends it to the Interface layer, finalising the procedure (6).

5. Deployment views

The WatERP architecture consists of a number of separate services that have to be installed. The deployment diagram in Figure 21 gives an overview over all the components and the existent communication channels.

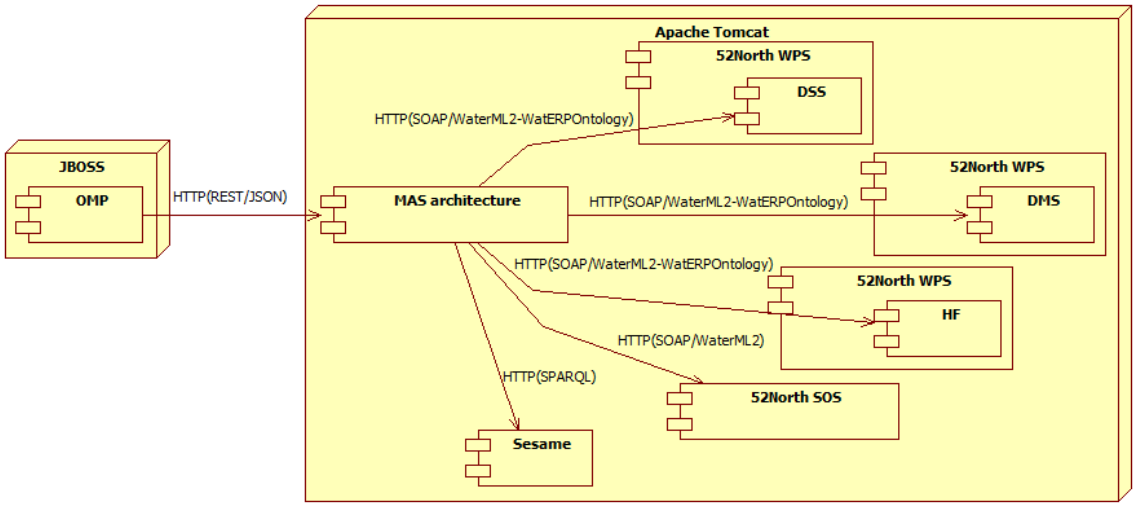

Figure 21: "WatERP deployment view"

More detailed information about the component deployment is provided in the deliverables of each work package.

- Sesame and 52º-North-SOS deployment is described on D3.4 "WDW Final Prototype" in Section 5
- DSS processes deployment is described on D4.3 "Inference and Simulation Engine 2nd Prototype" in Section 3 and D4.4 "Inference and Simulation Engine Final Prototype" in Section 2.1.2
- DMS processes deployment will be described on D7.4.2 "Implementation of the Demand Management System"
- HMF processes deployment is described on D2.5 "Water Availability Prediction System Integration" in Section 4.2

Below is fully described how multi-agent architecture and OGC® WPS processes should be deployed.

5.1 Multi-agent Architecture deployment

This section is focused on presenting the most important steps for the MAS installation which are described below. For more detailed instructions read the installation how-to of each application (Java, Apache Tomcat, etc.).

1. Make certain that the Java Development Kit 1.7.* is installed on the server.
2. Install an Apache Tomcat. For more details about the installation see https://tomcat.apache.org/tomcat-7.0-doc/index.html.
3. Copy MASv1.0.war into the /webapps folder of the tomcat installation.
4. Start the Tomcat Server to unpack the MASv1.0.war into *"<tomcat-root>/webapps/*".
5. Adjust the MASv1.0 configurations.

a. Edit "*<tomcat-root>\webapps\MASv1.0\WEB-INF\properties\register.properties*" to register default Sesame repositories and SOS and WPS servers (see Listing 49).

```
# Registered Sesame to manage the ontology
waterp.mas.sesame.url = http://172.20.10.196:8083/openrdf-sesame
waterp.mas.sesame.repository = useekm-owlim-1
# Registered SOS servers
waterp.mas.sosservers.uri = SOS1
waterp.mas.sosservers.url = http://192.168.45.129:8090/wdwsos/sos
# Registered WPS servers
waterp.mas.wpsservers.uri = DSS, DMS, HydrologicalForecast
waterp.mas.wpsservers.url = http://127.0.0.1:8081/52n-wps-webapp-
3.2.0/services/WPS.WPSHttpSoap12Endpoint,http://localhost:8384/dss,http://172.20.10.196:8385/dms
```

Listing 49: "Default integrated WPS and SOS servers and Sesame repositories"

b. Edit "*<tomcat-root>\webapps\MASv1.0\WEB-INF\properties\general.properties*" to configure the ontological server responsible to manage the knowledge process.
The key "*waterp.mas.kp.url"* contains the URL of the ontological server and the "*waterp.mas.kp.repository"* contains the URI of the repository (see Listing 50).

```
# Knowledge process configuration
waterp.mas.kp.url = http://172.20.10.196:8083/openrdf-sesame
waterp.mas.kp.repository = useekm-owlim-1
```

Listing 50:" Knowledge process configuration"

6. Restart the Tomcat server.

5.2 52º North OGC® WPS processes deployment

This section depicts the generic information to install the 52º North OGC® WPS server that is integrated on WatERP framework. Basically, the installation consists on the following steps:

1. Install an Apache Tomcat[6]. For more details about the installation see https://tomcat.apache.org/tomcat-7.0-doc/index.html.
2. Download the WAR file corresponding to the 52º North OGC® WPS server[7].
3. Copy the WAR file into the "*/webapps*" folder of the tomcat installation.
4. Start Tomcat to force the deployment of the WAR file containing the 52º North OGC® WPS server.

If everything worked correctly, the Tomcat welcome page should appear through "*http://localhost:8080*" and the 52º North home page should be accessible at "*http://localhost:8080/52n-wps-webapp-3.2.0*".

[6] Apache Tomcat is available at http://ftp.cixug.es/apache/tomcat/tomcat-7/v7.0.56/bin/apache-tomcat-7.0.56.zip

[7] 52º North OGC WPS server WAR file is available at http://52north.org/downloads/geoprocessing/wps/52-north-wps-3-2-0/download

Once the 52º North OGC® WPS server is suitably installed, the implemented processes are deployed in the server following these steps:

1. Copy the JAR file generated (for instance "*ProcessWPS.jar*") and any other JAR file linked to the main process into the "*WEB-INF/lib*" folder of the 52º North installation (e.g. "*/TOMCAT_HOME/webapps/52n-wps-webapp-3.2.0/WEB-INF/lib*").
2. Start the Tomcat server
3. Access the Web Admin Console (see Figure 22) provided by 52º North (e.g. "*http://localhost:8080/52n-wps-webapp-3.2.0/webAdmin/index.jsp*", where the default username/password is "*wps"/"wps"*).

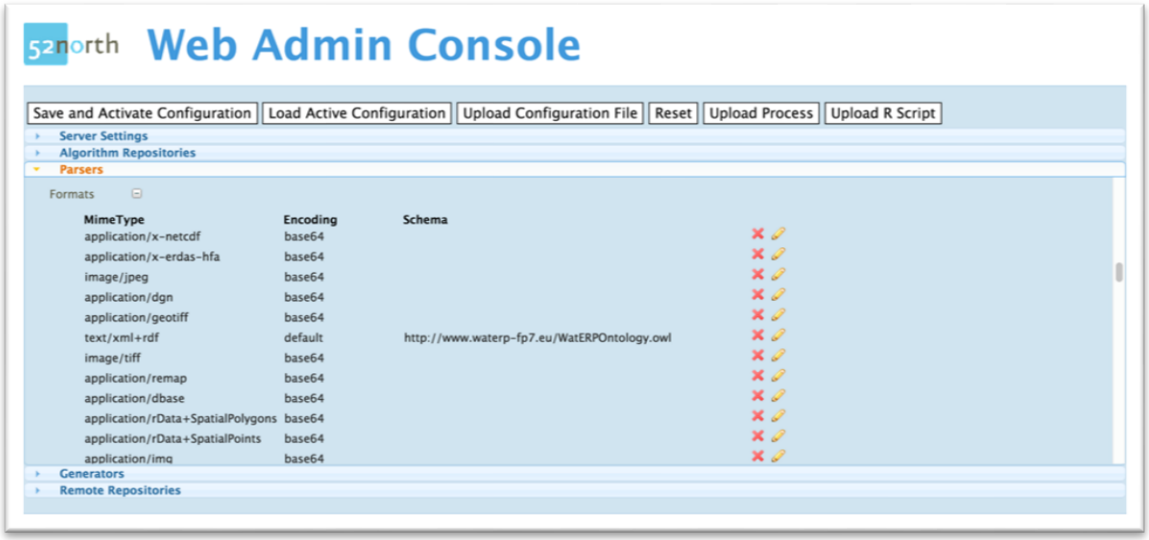

Figure 22 "WPS web administration provided by 52º North server"

4. Go to "*Parsers*", locate "*GenericFileParser*" and add the formats corresponding to your input parameters.
5. Now, go to "*Generators*", locate "*GenericFileGenerator*" and add the formats corresponding to your output parameters.
6. Next, go to "*Algorithm Repositories*", locate "*LocalAlgorithmRepository*" and add your processes.
7. Finally, save all changes by clicking on button "*Save and Activate Configuration*"

When these steps are completed, the WPS server has the information needed to make your processes available and ready to be invoked (e.g. descriptions, configurations and executable files). Moreover, 52º North OGC® WPS server provides a simple interface (called "*WPS TestClient*") to test any registered process.

6. Conclusions and future work

The WatERP MAS's main objective is to orchestrate the building blocks to support the manager's decisions in matching water supply and demand. Therefore, one of the main aspects of the WatERP MAS is to *(i)* embed building blocks to make their functionalities accessible to the other building blocks such as the logical models, observations and processes; and *(ii)* orchestrate the building block content.

In order to accomplish the defined objectives, the WatERP MAS provides a system to register Sesame repositories based on WatERP ontology, OGC® SOS servers and OGC® WPS servers which fulfil the guidelines presented in the D2.3 "Open Interface Specification" in Section 3.2.2.1, allowing to an open system for further building blocks. Moreover, the operational operations presented in the Section 3.1.1 cover all OMP requirements depicted in Section 4.2.3.1 of the D6.3 "OMP 2nd prototype". Also, the sequence diagram described in the Section 4 demonstrates that Multi-agent architecture is able to orchestrate the different information provided by the building blocks (OGC® WPS Servers), Sesame server and OGC® SOS Server by taking advantage of the WatERP Process Knowledge Service, Directory Facilitator Service, Agent Management Service, SOS Manager, WPS Manager and Water Ontology Manager.

Although this is the last deliverable reference to the MAS-SOA architecture, different tasks will be carried out in the comings months in order to ensure proper deployment, integration and performance of the platform. Therefore, the Future work to be accomplished will be focused on supporting deployment and setting of the multi agent architecture in the BDigital Servers and pilot servers. Then, strategy during the next time period will be focused on: *(i)* deploy and set multi-agent agent architecture on BDigital servers; *(ii)* support testing of multi-agent architecture and *(iii)* deploy and set multi-agent architecture on pilot servers.

(i) Deployment and setting of the multi-agent architecture for both pilots on BDigital server in order to validate the platform by experts and analyse performance requirements.

(ii) Working together with the work package 7 to define and implement an set of tests to validate multi-agent architecture operation assuring the correct performance of the platform

(iii) Once the whole platform is validated, support the deployment on pilots' servers.

7. Appendix I

Appendix I – Management operations WSDL

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://service.mas.waterp.alim.bdigital.org/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
  name="BuildingBlockIntegratorService" targetNamespace="http://service.mas.waterp.alim.bdigital.org/">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
      xmlns:tns="http://service.mas.waterp.alim.bdigital.org/"
      elementFormDefault="unqualified" targetNamespace="http://service.mas.waterp.alim.bdigital.org/"
      version="1.0">
      <xs:element name="registerWPS" type="tns:registerWPS" />
      <xs:element name="registerWPSResponse" type="tns:registerWPSResponse" />
      <xs:element name="unregisterWPS" type="tns:unregisterWPS" />
      <xs:element name="unregisterWPSResponse" type="tns:unregisterWPSResponse" />
      <xs:complexType name="registerWPS">
        <xs:sequence>
          <xs:element name="url" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="registerWPSResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:boolean" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="unregisterWPS">
        <xs:sequence>
          <xs:element name="url" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="unregisterWPSResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:boolean" />
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="registerWPS">
    <wsdl:part element="tns:registerWPS" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="unregisterWPSResponse">
    <wsdl:part element="tns:unregisterWPSResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="unregisterWPS">
    <wsdl:part element="tns:unregisterWPS" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="registerWPSResponse">
    <wsdl:part element="tns:registerWPSResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="IBuildingBlockIntegratorService">
    <wsdl:operation name="registerWPS">
      <wsdl:input message="tns:registerWPS" name="registerWPS">
      </wsdl:input>
      <wsdl:output message="tns:registerWPSResponse" name="registerWPSResponse">
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="unregisterWPS">
      <wsdl:input message="tns:unregisterWPS" name="unregisterWPS">
      </wsdl:input>
      <wsdl:output message="tns:unregisterWPSResponse" name="unregisterWPSResponse">
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="BuildingBlockIntegratorServiceSoapBinding"
    type="tns:IBuildingBlockIntegratorService">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="registerWPS">
      <soap:operation soapAction="" style="document" />
      <wsdl:input name="registerWPS">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="registerWPSResponse">
        <soap:body use="literal" />
```

```
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="unregisterWPS">
      <soap:operation soapAction="" style="document" />
      <wsdl:input name="unregisterWPS">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="unregisterWPSResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="BuildingBlockIntegratorService">
    <wsdl:port binding="tns:BuildingBlockIntegratorServiceSoapBinding"
      name="BuildingBlockIntegratorServiceImplPort">
      <soap:address location="http://localhost:8080/MasCore/integratorService" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Appendix I – Gateway WADL

```
<application xmlns="http://wadl.dev.java.net/2009/02"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:prefix1="http://www.waterp-fp7.eu/masservice">
  <grammars>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www.waterp-fp7.eu/masservice" attributeFormDefault="unqualified"
      elementFormDefault="unqualified" targetNamespace="http://www.waterp-fp7.eu/masservice">
      <xs:import />
      <xs:element name="DataObservationResponse" type="DataObservationResponse" />
      <xs:element name="ExecuteRequest" type="ExecuteRequest" />
      <xs:element name="ExecuteResponse" type="ExecuteResponse" />
      <xs:element name="FeaturesOfInterestResponse" type="FeaturesOfInterestResponse" />
      <xs:element name="LogicalModel" type="LogicalModel" />
      <xs:element name="LogicalModelResponse" type="LogicalModelResponse" />
      <xs:element name="LogicalModelsResponse" type="LogicalModelsResponse" />
      <xs:element name="Observation" type="Observation" />
      <xs:element name="ObservationsResponse" type="ObservationsResponse" />
      <xs:element name="ProcessesResponse" type="ProcessesResponse" />
      <xs:element name="WaterResource" type="WaterResource" />
    </xs:schema>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:ns1="http://www.waterp-fp7.eu/masservice" attributeFormDefault="unqualified"
      elementFormDefault="unqualified" targetNamespace="http://www.waterp-fp7.eu/masservice">
      <xs:import namespace="http://www.waterp-fp7.eu/masservice" />
      <xs:element name="FeatureOfInterest" type="FeatureOfInterest" />
      <xs:complexType name="LogicalModelsResponse">
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0"
            name="logicalModel" nillable="true" type="LogicalModel" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="LogicalModel">
        <xs:sequence>
          <xs:element minOccurs="0" name="name" type="xs:string" />
          <xs:element minOccurs="0" name="uri" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="ObservationsResponse">
        <xs:sequence>
          <xs:element minOccurs="0" name="featureOfInterest" type="xs:string" />
          <xs:element minOccurs="0" name="logicalModel" type="xs:string" />
          <xs:element maxOccurs="unbounded" minOccurs="0"
            name="observations" nillable="true" type="Observation" />
          <xs:element minOccurs="0" name="waterResource" type="xs:string" />
```

```
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Observation">
      <xs:sequence>
        <xs:element minOccurs="0" name="name" type="xs:string" />
        <xs:element minOccurs="0" name="uri" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="FeaturesOfInterestResponse">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0"
          name="featuresOfInterests" nillable="true" type="FeatureOfInterest" />
        <xs:element minOccurs="0" name="logicalModel" type="xs:string" />
        <xs:element minOccurs="0" name="waterResource" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="FeatureOfInterest">
      <xs:sequence>
        <xs:element minOccurs="0" name="name" type="xs:string" />
        <xs:element minOccurs="0" name="position" type="xs:string" />
        <xs:element minOccurs="0" name="uri" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ProcessesResponse">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="process"
          nillable="true" type="wpsProcess" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="wpsProcess">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="inputList"
          nillable="true" type="param" />
        <xs:element maxOccurs="unbounded" minOccurs="0" name="outputList"
          nillable="true" type="param" />
        <xs:element minOccurs="0" name="identifier" type="xs:string" />
        <xs:element minOccurs="0" name="title" type="xs:string" />
        <xs:element minOccurs="0" name="abstractTxt" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="param">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="keywords"
          type="keywords" />
        <xs:element minOccurs="0" name="identifier" type="xs:string" />
        <xs:element minOccurs="0" name="title" type="xs:string" />
        <xs:element minOccurs="0" name="abstractTxt" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="keywords">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="keyword"
          nillable="true" type="xs:string" />
        <xs:element minOccurs="0" name="type" type="type" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="type">
      <xs:sequence>
        <xs:element minOccurs="0" name="codeSpace" type="xs:string" />
        <xs:element minOccurs="0" name="name" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="simpleParam">
      <xs:complexContent>
        <xs:extension base="param">
          <xs:sequence>
            <xs:element minOccurs="0" name="dataType" type="xs:string" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
```

```
    </xs:complexType>
    <xs:complexType name="complexParam">
      <xs:complexContent>
        <xs:extension base="param">
          <xs:sequence>
            <xs:element minOccurs="0" name="mimeType" type="xs:string" />
            <xs:element minOccurs="0" name="enconding" type="xs:string" />
            <xs:element minOccurs="0" name="schema" type="xs:string" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="ExecuteResponse">
      <xs:sequence>
        <xs:element minOccurs="0" name="process" type="wpsProcess" />
        <xs:element name="output">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" minOccurs="0" name="entry">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element minOccurs="0" name="key" type="xs:string" />
                    <xs:element minOccurs="0" name="value" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DataObservationResponse">
      <xs:sequence>
        <xs:element minOccurs="0" name="logicalModel" type="xs:string" />
        <xs:element minOccurs="0" name="waterResource" type="xs:string" />
        <xs:element minOccurs="0" name="featureOfInterest" type="xs:string" />
        <xs:element minOccurs="0" name="observation" type="xs:string" />
        <xs:element minOccurs="0" name="data" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="LogicalModelResponse">
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0"
          name="waterResource" nillable="true" type="WaterResource" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="WaterResource">
      <xs:sequence>
        <xs:element minOccurs="0" name="name" type="xs:string" />
        <xs:element maxOccurs="unbounded" minOccurs="0" name="relations"
          nillable="true" type="xs:string" />
        <xs:element minOccurs="0" name="type" type="xs:string" />
        <xs:element minOccurs="0" name="uri" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ExecuteRequest">
      <xs:sequence>
        <xs:element minOccurs="0" name="process" type="wpsProcess" />
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</grammars>
<resources base="http://localhost:8080/MasCore/service">
  <resource path="/process">
    <method name="GET">
      <response>
        <representation mediaType="application/xml"
          element="prefix1:ProcessesResponse" />
        <representation mediaType="application/json"
          element="prefix1:ProcessesResponse" />
```

```
      </response>
    </method>
    <method name="POST">
      <request>
        <representation mediaType="application/xml" element="prefix1:ExecuteRequest" />
        <representation mediaType="application/json"
          element="prefix1:ExecuteRequest" />
      </request>
      <response>
        <representation mediaType="application/xml" element="prefix1:ExecuteResponse" />
        <representation mediaType="application/json"
          element="prefix1:ExecuteResponse" />
      </response>
    </method>
  </resource>
  <resource path="/observation">
    <method name="GET">
      <request>
        <param name="uri_logical_model" style="query" type="xs:string" />
        <param name="uri_water_resource" style="query" type="xs:string" />
        <param name="uri_feature_of_interest" style="query" type="xs:string" />
      </request>
      <response>
        <representation mediaType="application/xml"
          element="prefix1:ObservationsResponse" />
        <representation mediaType="application/json"
          element="prefix1:ObservationsResponse" />
      </response>
    </method>
    <resource path="/{uri_observation}">
      <param name="uri_observation" style="template" type="xs:string" />
      <method name="GET">
        <request>
          <param name="uri_logical_model" style="query" type="xs:string" />
          <param name="uri_water_resource" style="query" type="xs:string" />
          <param name="uri_feature_of_interest" style="query" type="xs:string" />
          <param name="start_datetime" style="query" type="xs:string" />
          <param name="end_datetime" style="query" type="xs:string" />
        </request>
        <response>
          <representation mediaType="application/xml"
            element="prefix1:DataObservationResponse" />
          <representation mediaType="application/json"
            element="prefix1:DataObservationResponse" />
        </response>
      </method>
    </resource>
  </resource>
  <resource path="/featureOfInterest">
    <method name="GET">
      <request>
        <param name="uri_logical_model" style="query" type="xs:string" />
        <param name="uri_water_resource" style="query" type="xs:string" />
      </request>
      <response>
        <representation mediaType="application/xml"
          element="prefix1:FeaturesOfInterestResponse" />
        <representation mediaType="application/json"
          element="prefix1:FeaturesOfInterestResponse" />
      </response>
    </method>
  </resource>
  <resource path="/logicalModel">
    <method name="GET">
      <response>
        <representation mediaType="application/xml"
          element="prefix1:LogicalModelsResponse" />
        <representation mediaType="application/json"
          element="prefix1:LogicalModelsResponse" />
      </response>
```

```
      </method>
      <resource path="/{uri}">
        <param name="uri" style="template" type="xs:string" />
        <method name="GET">
          <request></request>
          <response>
            <representation mediaType="application/xml"
              element="prefix1:LogicalModelResponse" />
            <representation mediaType="application/json"
              element="prefix1:LogicalModelResponse" />
          </response>
        </method>
      </resource>
    </resource>
  </resources>
</application>
```