NTNU

Norwegian University of
Science and Technology

# Semantic Web Services in a Network Management System

**Olav Nistad**

Master of Science in Communication Technology
Submission date: June 2009
Supervisor: Finn Arve Aagesen, ITEM

Norwegian University of Science and Technology
Department of Telematics

# Problem Description

Semantic Web Services (SWS) combines semantic web and web-services technology.
Semantic Web technology offers meta-data descriptions which are machine-interpretable and that can be used as a basis for logic reasoning. Web Services technologies both make it possible to find, locate and interact with services offered by other organizations.

The objectives with this research work is to apply SWS technology for a network management system (NMS), which can install SNMP managers during run-time
in systems running TAPAS platform.

1.Analyse the potential benefit of using ontology and reasoning applications
in this system.
2.Specify proposed ontology and reasoning applications integrated with the NMS application. Ontology shall be specified using Protégé-OWL Editor.
3.Specify and implement web-service based applications that makes the reasoning applications from 2 available as WEB-services

Assignment given: 15. January 2009
Supervisor: Finn Arve Aagesen, ITEM

**Abstract**

Semantic Web Services (SWS) are a facility towards full automation of service usage, providing seamless integration of services that are published and accessible on the Web. Based on Semanic Web technology SWS is simply a semantic annotation of the functionalitites and interfaces of Web Services. In the very same way that ontologies and metadata lanaguages will facilitate the integration of static data on the Web, the annotation of services wil help to facilitate the automation of service discovery, service composition, service contracting, and execution.

In this thesis we demonstrate how SWS technology can be applied to a network management system (NMS), which can install SNMP managers during run-time in systems running TAPAS platform. Several reasoning applications are made and integrated with the existing system. In addition, we specify a set of Semantic Web Services described using OWL-S, in order to execute these applications.

# Preface

This paper is the result of the author's masters thesis carried out at the Department of Telematics[1] at the Norwegian University of Science and Technology[2]. The work on this thesis begun in January 2009 and ended in June the same year.

## Acknowledgements

I would like to thank my teaching supervisor Professor Finn Arve Aagesen for much appreciated guidance in the initial stages of the work. Furthermore, I would like to give my appreciation to the participants of the MindSwap OWL-S mailing list[3], who have been of great service during the implemenation stages of the thesis.

Oslo, June 16, 2009

Olav Nistad

[1] http://www.item.ntnu.no/
[2] http://www.ntnu.no/
[3] http://lists.mindswap.org/mailman/listinfo/owl-s/

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Web Service technology and the idea of Service-Oriented Architecture (SOA) for web-based implementation of distributed software systems has experienced a tremendous success [24]. In short time, the SOA approach not only received much praise in the Computer Science research community, but also gained considerable interest by big international players in the IT industry, such as Microsoft, IBM and SAP.

One vital component of the SOA approach is Web Services, which provides a platform- and programming language independent way of achieving interoperability between different parts of distributed software systems.

Semantic Web technology aims at harmonising semantical discrepancies in software systems by providing machine-interpretable semantics, making computers "understand" parts of the information it is processing. This can enable computers to make automated decisions, thus creating more powerful and intelligent applications. The approach of combining Web Services with Semantic Web technology is called Semantic Web Services (SWS). Semantic Web Services applies semantic annotations to the inputs, outputs, preconditions and effects of Web Services - expressed in knowledge representation languages, referring to shared ontological vocabularies. This can enable a higher degree of automation and produce more precise results than conventional Web Services.

In this work we will investigate the potential benefit of applying Semantic Web technology to a network management system (NMS). The network management system was created as part of a project thesis carried out in autumn 2008[17] . The system aims at limiting the amount of computation required by a central managament station as well as reducing network management traffic, by installing SNMP managers during run-time in devices running TAPAS platform [19]. After an investigation of what parts and components of the current NMS applicaiton may benefit from being applied with SW technology, we will propose a new version of the application where ontology and reasoning applications is integrated with the NMS application.

## 1.1 Motivation

Allthough experiencing a slow start since the idea was first released, the Semantic Web has received increased attention and recognition. Several projects

have been researching Semantic Web technology, and multiple promising tools and framworks based on this technology are under development. Semantic Web Services is bringing Semantic Web technology in to the field of service-oriented computing - a field which has become one of the predominant factors in current IT research and development efforts. Issues such as intelligent service discovery or fully automated service composition are subject to widespread ongoing research in many labs.

In the autumn of 2008 we carried out a project thesis[17] where we created a SNMP monitoring application based on the TAPAS Platform[4]. Using this application as a starting point, it will be interesting to investigate how Semantic Web technology can be applied to this system to further improve it.

## 1.2 Reader's Guide

This section describes the structure and content of this thesis. The thesis is divided in three main parts; Part I will give the reader a background on the different technologies and approaches used in this work. Part II will present our new NMS application, including it's architecture and building blocks.

In Part III we will evaluate our work. This includes a discussion of how our result harmonizes with our objectives, a proposal for future work as well as a presentation of related works.

**Part I: Background**

**Section 2: Web Services** This section will give a brief presentation of Web Services, including it's advantages and main building blocks.

**Section 3: Semantic Web Technology** This section will give the reader an introduction to Semantic Web technology. This includes a description of knowledge representation in general as well as different knowledge representation languages. The latter includes a brief introduction to RDF as well as a more in-depth description of OWL. In addition the reader will be given a brief introduction to the Protg-OWL Editor.

[4] `http://tapas.item.ntnu.no/`

**Section 4: Semantic Web Services** This section will describe the main drawbacks of conventional Web Services, and explain how Semantic Web Services can help address them. Following this key concepts of Semantic Web Services will be described.

**Section 5: An Semantic Web Service Approach: OWL-S** The section will give the reader an presentation of the SWS approach that is used in our NMS application, namely OWL-S. The main components of the approach as well as different developing tools used to create OWL-S services will be described.

**Part II: SWS Applied to a Network Management System**

**Section 7: SNMP-based Monitoring Application by using TAPAS Platform** This section will give the reader a brief presentation of the current NMS application created in autumn 2008.

**Section 8: Application Improement Using Semantic Web Technology** This section will pinpoint the different parts of our existing solution that may benifit from being improved using SW technology.

**Section 9: An SW Enriched SNMP-Based Monitoring Application** Here we will present our new NMS application which is based on the current application only extended with SW technology. The different ontology- and reasoning applications will be presented as well as how they are accessed using OWL-S to semantically annotate Web Services. The different drawbacks presented in section 8 will be addressed.

**Part III: Evaluation**

**Section 10: Conclusion** Here we will discuss whether and how we achieved our objectives for this thesis.

**Section 11: Evaluation and Future Work** In section 11 we will give a more general evaluation of our result as well as present some proposals for future work.

**Section 12: Related Work** Here we present some related work.

# Part I

# Background

# 2 Web Services

As distributed software systems are becoming increasingly powerful a wide variety of rich services can be offered. On the other hand such software systems are also becoming more complex and measures needs to be taken to bridge the gap betweeen separated heterogeneous areas. An effective solution to achieve interoperability in such distributed software systems can be realized through the use of Web Services. Web Services is a standardized architecture for modular systems, where new functionality can be made from existing building blocks and where communication can be established between hetereogenous elements. Other approaches that addresses the same goals such as CORBA or Multi-Agent Systems, do exist, but these technologies lack some of the great advantages of the WS approach. Firstly, WS technology is a simple extension of existing Internet standards and based on widely accepted protocols such as HTTP. Secondly, it is platform independent and allows for easy encapsulation of existing code and applications

Web Services allow access to a functionality via the Web using a set of open standards that make the interaction independent of implementation aspects such as the operating system plaform and the programming language used. Web service technology build upon four main components:

- An agreed transport protocol: HTTP
- A platform-independent message description format: SOAP
- A language for Web service interface description that describes which operations and messages a service can offer: WSDL
- A registry for publication and discovery of available services: UDDI

## 2.1 HTTP

The first main component can, in principle, be realized by any of the common transport protocols such as FTP or SMTP. Because of it's "robustness" against firewalls, the most popular protocol in the context of Web services is, however, HTTP. Also, HTTP is ubiquitously available and its built-in addressing and error-handling functionalities are fully covering the needs of the Web Services message description format (SOAP).

## 2.2 SOAP

The second component is realized by SOAP. SOAP is a specification for the exchange of XML-coded messages and specifies the binding to HTTP as an underlying communication protocol between two addressable endpoints. The most important advantage over competing technologies, like Java RMI or CORBA, is thatt SOAP is absolutely independent from a certain operating system, a programming language or special runtime components. SOAP aims to achieve maximum acceptance and flexibility by the provision of a sophisticated extension model in which application-specific information may be conveyed in an extensible manner, without making any up-front commitment to the semantics of application-specific data. A SOAP message itself is just a XML document. Figure 1 depicts the shematic structure of a SOAP message, while the XML representation is given below the figure..

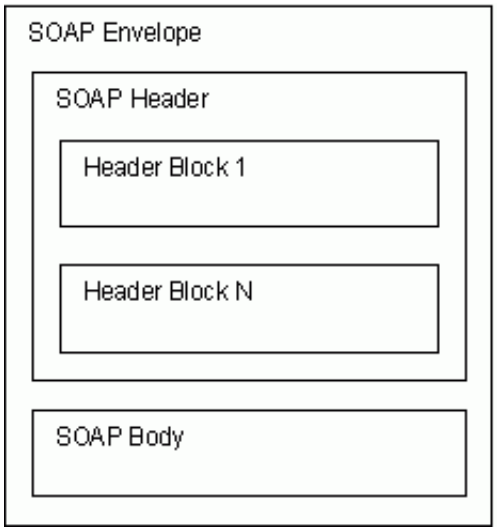


Figure 1: Structure of a SOAP Message

Within a SOAP envelope, an optional header- and a mandatory body element is present. The body-element contains the actual payload of the message. The SOAP specification does not define any constraints about the data in the message body, one can include all kinds of XML data here. The used data format is application specific.

```
<? zml version=1.0 encoding =UTF-8 ?>
        <env:Envelope xmlns:env=http://www.w3.org/2003/05/soap-envelope>
                <env:Header>
                        ...
                </env:Header>
                <env:Body>
                ...
                </env:Body>
        </env:Envelope>
```

## 2.3 Web Service Description Language (WSDL)

The third component enabling Web services as an universal middleware technology is a powerful and well-structured Interface Definition Language (IDL). The basic task of an IDL is to provide an exact and machine readable definition of service interfaces. Also, an IDL allows a distinction between the description of the abstract functionality (operations) that a service provides and the details of how to access the service. A service requester interprets the IDL description of a service provider in order to generate service calls that are compatible with the according service interfaces. The current IDL approach for Web services, WSDL, is structured in five main sections: documentation, types, interface, binding and service. The documentation section contains additional textual information on how to use the described service for humans. Its content is meant as an endorsement to the other sections of the WSDL document, which are mainly meant to be interpreted by machines.

In the types section all data types that will be used in the input and output messages of the service operations are declared. This is typically done using XML Schema. Unlike other XML grammar description languages XML Schema provides a very sophisticated type system which can be directly used for specifying basic data types like integers, strings and dates, as well as compund data types. Furthermore, extensions and restrictions of existing data types can be described.

The interface section is basically the core component of a WSDL document. Here each service operation is listed and its inputs and outputs are specified by referencing the according data type definitions which were specified in the types section. Up to this point the service description is abstract, i.e. independent from a certain messaging format or transport mechanism.

In the binding section we are mapping our abstract service operations to concrete ones. We specify the used messaging format (e.g. SOAP 1.2) and

the protocol used for message transport (e.g HTTP 1.1). The according service operation, declared in the interface section are referenced using the *ref* attribute.

In the service section we finally define service endpoints. An endpoint references a previously defined binding and provides all necessary technical information for accessing its service operations. This is typically done by providing the URL of the Web Service.

## 2.4 Universal Description, Discovery and Integration (UDDI)

UDDI is a framework that provides means to publish (advertise) Web services as well as to browse and query existing Web services. UDDI provides a data model for services and business entities. more concretely, it provides three categories of information: white, yellow, and green pages. This model provides related information to a service such as the name, address, telephone number, and other contact information of a given application; basically, any information that categorizes applications, and technical information about the Web services provided by a given application.

To summarize, the four mentioned core components enable a set of basic interactions required in a Web-service-oriented architecture, as depicted in figure 2

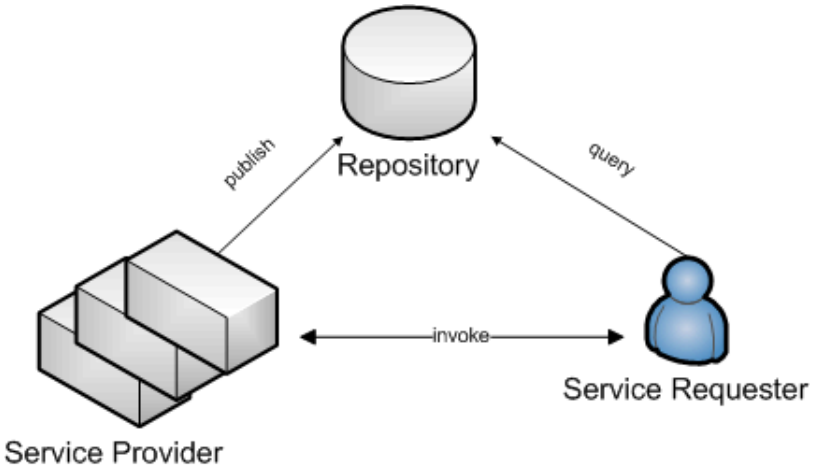


Figure 2: Web Service interaction

This figure illustrates the basic building blocks of a Web-service-oriented architecture. In order for a Web service to be discovered, a service provider must first publish the service in a UDDI. A service requester is then able to issue queries to the UDDI in order to obtain a reference to the desired service. To invoke a service the requester needs to know how to do this (i.e it needs the interface definition). This information is provided by WSDL (Web Service Description Language). After interpreting the WSDL document the service requester can now invoke the service. All communication between the parties are done over SOAP.

# 3 Semantic Web Technology

Today, most web pages in the current Web is built up by HTML. HTML is a language with emphasis on visual presentation that describes a body of structured text interspersed with multimedia objects and interactive forms. The emphasis has been on publishing and presenting the information to a human being. HTML has however limited ability to classify the blocks of text on a page, apart from the roles they play in a typical document's organization and in the desired visual layout. In the current Web one has to know where things are located. For example, the URL http://www.vg.no/ indicates only an address of a web site, but it does not tell you what exactly it contains. The user needs to get information up front about what are contained where, e.g from email, from advertisement, friends etc. The Web contains so much information that it becomes increasingly difficult to find exactly what you look for. Despite the successful introduction of powerful search engines, the Web does not usually function as a content or knowledge management platform. It is difficult to find, sort and catalog all the information that is out there. Due to the fact that the Web itself has limited ability to help users answer complex question or perform many day-to-day tasks, the Web is barely an adequate information retrieval tool. This limitation is rooted in the inability of computers to understand the semantics behind the information it is processing. That is, HTML specifies how information should appear, but ignores the meaning or significance of that information. Tim Berners-Lee and his colleagues at W3C have been addressing these limitations of the current Web and they call the next stage the Semantic Web. While still in the initial stages of development, the project entails adding an additional layer of Web infrastructure to regular Web technology. The key idea behind the Semantic Web is augmenting Web documents with meta-data and rules of logic. The resulting infrastructure helps computers understand Web data in the same way that humans do. Adding semantics to the current Web allows computers to make decisions, form interferences and respond to complex queries. The Semantic Web will enable users to search not only for documents that contain data, but also for the desired data itself, through semantic identification and location techniques. It will support software agents that are able not only to locate data, but also to perform meaningful tasks with data automatically and on the fly that today must be done manually and episodically

by computer users. To accomplish this the Semantic Web uses a set of different technologies. The most important ones will be described in the following subsections.

## 3.1 Knowledge Representation and Ontologies

Common sense for humans is not necessarily, and most often not, common sense for traditional software systems. For example, a computer have no way of knowing that the word "empire" can have the same meaning as the word "realm", in some contexts. Nor can it know that the word "*dog*", found in one location, refers to the same concept as the word "*dog*" found in another location. Real communication can only be achieved between two parties if they both share a common understanding of how the language refers to concepts prevalent in the real world, and if both know which constraints and which background knowledge is typically associated with these concepts. All though knowledge like this is common-sense for most humans, it is typically not available in a computer system. For this reason, there is still a need for manual human intervention in order to interpret the semantics of information residing in software systems.

The aim of the Semantic Web is to harmonize semantical discrepancies in software systems by providing machine-interpretable semantics, enabling a machine to understand and reason about the information carried in the data it is processing. This is realized by creating meta-data for web accessible information. This meta-data is expressed in powerful logic-based representation languages that refer to the controlled vocabulary of shared and quasi standardized domain knowledge models, also called ontologies.

Long considered as one of the principal elements of Artificial Intelligence, knowledge representation and reasoning aim at designing computer systems that are able to reason about a machine-interpretable representation of the world. A knowledge base is a computational model of some domain of interest which contains symbolic surrogates, substitutes of real world concepts, such as physical objects and relationships. In a knowledge-based system these surrogates are formed as statements about the domain, and reasoning can be achieved by manipulating these statements.

If the domain of interest is, for instance "Animals". The knowledge base can then be filled with statements like: "*An animal can either be a predator or a herbivore.* "*A predator is an animal that lives by preying on other animals*".

*"Cheetahs and gazelles are special kinds of animals"*. "*Cheetah's favorite food is gazelles*". From the given statements a knowledge-base system can then deduce that a cheetah is a predator.

In this way, a knowledge-based system can reason about and be able to deduce own conclusions about the domain of animals, similar to what a human would. After filling in more (a lot more in this case) statements it could for example deduce that the African and Asian elephant are both elephants belonging to the same family, but due to a few genetic differences they cannot be interbred.

## 3.2 How to represent knowledge

In an actual knowledge base, the statements can not be written as they appear above. Instead, for enabling a machine to understand and reason about some knowledge, one has to represent this knowledge in a machine-interpretable form, also called an ontology language. The term "ontology" originates from philosophy and has been adopted in the field of Computer Science with a slightly different meaning. Adopting the definition by [1], ontology can be defined as:

*An ontology is a formal explicit specification of a shared conceptualization.*

More precisely, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically concepts (or classes/sets), attributes (or properties), and individuals (or instances).

Ontologies interweave human and computer understanding of symbols. The representational primitives or terms can be interpreted by both humans and machines. The meaning for a human is represented by the term itself, which is usually a word in natural language, and by the semantics relationships between terms. An example of such a human-understandable relationship is a super-concept -sub-concept relationship (often referred to by the term "*is-a*"). Such relationship denotes the fact that one concept, the super-concept, is more general than another - the sub-concept. For instance, the concept Animal is more general than the concept Predator. Figure 3 depicts a simple is-a hierarchy

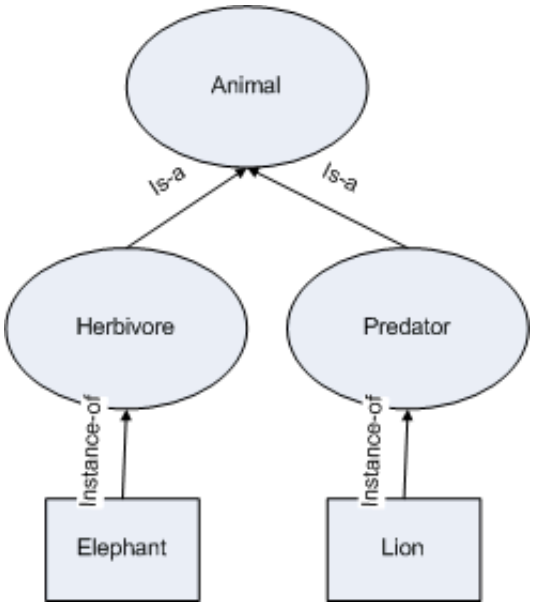


Figure 3: A simple is-a hierarchy (taxonomy)

(also known as taxonomy), where the more general concepts are located above the more specialized concepts.
A concept describe a particular object in the real world. The concept PREDATOR should capture all existing predators in the real world. Since lions, cheetahs, snakes and tigers all are predators, they are captured by this concept. These are called individuals or instances of the concept. In the figure 3 a lion is modeled as an instance of a predator through the instance-of relation. It is important to note that since the concept PREDATOR is a sub-concept of the concept ANIMAL. Any instance of PREDATOR is also an instance of ANIMAL

These relations, which are implicitly known to humans (e.g. a human knows that every predator is an animal), are encoded in a formally explicitly way so that they can be understood by a machine. One can say that the understanding possessed by humans are encoded in a way that ables machines to process it and draw own conclusions based on logical reasoning.

### 3.2.1 Types of ontologies

There exist different types of ontologies, built for different types of use; and they vary in both generality and expressiveness. A very general ontology has a very broad scope, and tries to capture all commonsense knowledge (e.g. space and time). The expressiveness of an ontology refers to the level of detail given in the ontology.

Since an ontology is a specification of a shared conceptualization, domain experts, users and designers need to agree on the knowledge specified in an ontology so that the ontology may be shared and reused. Since such an agreement can be hard to achieve, it is a good idea to layer the knowledge in different ontologies on the basis of generality. Agreement is then required only between specific domain and application ontologies and between the higher/level ontologies that are being used. Hence, a categorization of ontologies can be made according to their subject of generalization. Top level ontologies, also called upper ontologies or foundational ontologies, attempt to describe very abstract and general concepts that can be shared across many domains and applications. Due to their generality, they are typically not directly used in applications but for other ontologies to be aligned to. On the other end you have application ontologies that provide the specific vocabulary required to describe a certain task enactment in a particular application context. These ontologies are limited to knowledge about a particular domain of interest. The narrower the scope of the domain for the ontology, the more an ontology engineer can focus on axiomatizing the details in this domain rather than covering a broad range of related topics. These lower ontologies inherit and specialize concepts and relations from the upper ones, while the upper ones have a broader potential for reuse.

## 3.3 Ontologies in the Semantic Web

The idea of the Semantic Web was boosted in the late 1990's. The general opinion held by W3C was that the Semantic Web needed an ontology language compatible with current web standards and that could be expressed in XML.

A much used illustration of the Semantic Web is the Semantic Web Stack, which is depicted in figure 4. The bottom layers of the stack, Unicode, URI and XML, is built up by existing web standards which forms the syntactical foundation for Semantic Web languages. Unicode provides a character-encoding

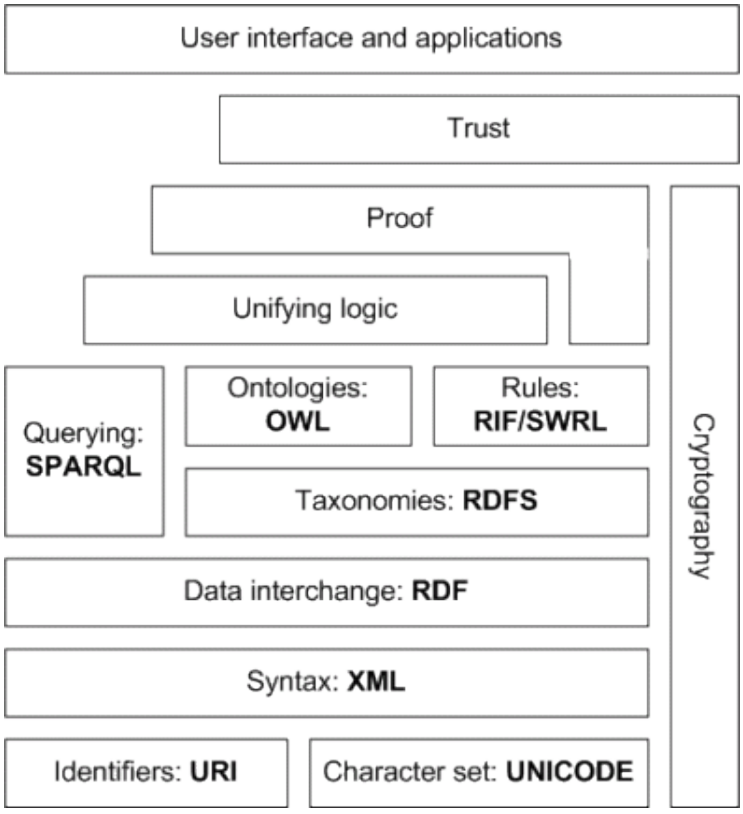


Figure 4: Semantic Web Stack

standard, used by XML. The URI (Uniform Resource Identifier) is used to identify or name a web resource. All concepts and resources used in the above layers can be specified using Unicode, and uniquely identified using URI's. RDF and OWL (Web Ontology Language) are kinds of ontology languages which will be described in the next section. Placing the logic layer on top of the OWL and rules layer has been subject to some disagreement since both OWL and rules are grounded in logic. The proof and trust layers are not well documented, but they do most likely refer to the application and not to any specific language. for instance, the application could prove some statement by using deductive reasoning, and a statement could be trusted if it had been proven and digitally signed by some trusted third party.

### 3.3.1 The Resource Description Framework (RDF)

The Resource Description Framework (RDF)[28] is the first language developed especially for the Semantic Web. RDF was developed as a language, realized in XML, for adding machine-readable meta-data to existing data on the Web. RDF Schema extends RDF with basic ontological primitives such as classes, properties and instances. In addition, the instance-of, subclass-of, and subproperty-of relationships have been introduced, allowing class- and property hierarchies. These primitives are used to create statements about resources (specified as URIs ) on the web. Such statements are formed as subject-property-object triples, also written as P(S,O). For instance, the triple hasAuthor(The Girl With the Dragon Tattoo, Stieg Larsson) is a statement saying that the book *The Girl With the Dragon Tattoo* (subject) has the author (property) Stieg Larsson (object). An object of a triple can, in turn, function as the subject of another triple, forming a directed labeled graph, (figure 5) where the subject and object correspond to nodes, and the edges between correspond to properties. The corresponding XML serialization is shown in figure 6

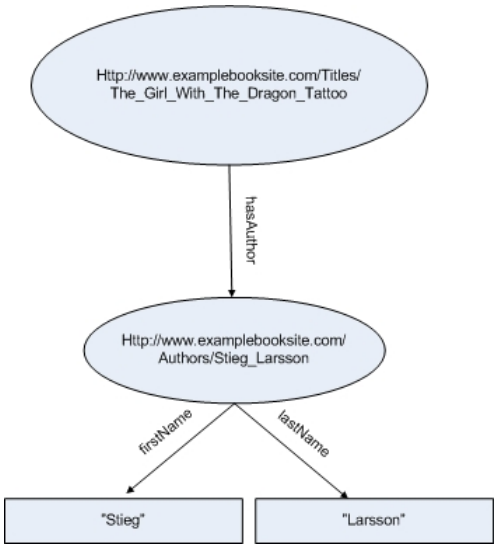


Figure 5: RDF triple example

```
<?xml version=1.0 ?>
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
xmlns:book=http://www.examplebooksite.com/books/#
xmlns:author=http://www.examplebooksite.com/authors/#
xmlns:base =http://www.examplebooksite.com/>

        <rdf:Description          rdf:about=#The_Girl_With_The_Dragon_Tattoo>
                <book:hasAuthor>
                        <author:firstName>Stieg</author:firstName>
                        <author:lastName>Larsson</author:lastName>
                </book:hasAuthor>
        </rdf:Description>
</rdf:RDF>
```

Figure 6: Example XML serialization

### 3.3.2 RDF Schema

While RDF is a language for describing resources with classes, properties and values, it has no way of defining the class hierarchies, property hierarchies and property restrictions. RDF Schema is an extension of RDF that provides a vocabulary for defining the application-specific vocabulary used by RDF. The resources described in a RDF document can be seen as instantiations of definitions in a RDF Schema. A document containing a combination of RDF and RDF Schema is called a RDF-S document. Figure 7show a simple RDF-S doc-

ument.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://www.animals.fake/animals#">

        <rdf:Description rdf:ID="animal">
                <rdf:type                rdf:resource="http://www.w3.org/2000/01/rdf-schema#Clas
        </rdf:Description>
        <rdf:Description rdf:ID="horse">
                <rdf:type                rdf:resource="http://www.w3.org/2000/01/rdf-schema#Clas
                <rdfs:subClassOf rdf:resource="#animal"/>
        </rdf:Description>
</rdf:RDF>
```

Figure 7: An example RDF-S document

RDF-S allows only the representation of concepts, concept taxonomies and binary relations and is therefore lacking expressiveness compared with many other ontology languages. For example, it has no way of expressing disjointness between classes, cardinality (e.g. "exactly one"), equality, rich typing of properties, characteristics of properties (e.g. symmetry) and enumerated classes. Nor does it provide means to specify rules and policies.

This limitation of RDF-S was the major motivation for developing more expressive languages for the Semantic Web. The next subsection will describe the ontology and rules components residing on top of the RDF-S layer in the Semantic Web Stack.

## 3.4 The Web Ontology Language: OWL

OWL [30] is an expressive ontology language which addresses the limitations of pure RDF-S. OWL serves as an extension of RDF-S and adds more vocabulary for describing properties and classes.

The language provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users:

- OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. Compared with RDF-S it adds local range restrictions, existential restrictions, simple cardinality restrictions (only 0 or 1), equality, and property characteristics (symmetric, transitive, inverse).

- OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaran-

teed to be computable) and decidability (all computations will finish in finite time). OWL DL adds full support for negation, disjunction, cardinality restrictions enumerations, and value restrictions.

- OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example a class can be treated simultaneously as a collection of individuals and as an individual in its own right. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full

Each of these sub-languages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. The following set of relations hold. Their inverses do not.

- Every legal OWL Lite ontology is a legal OWL DL ontology
- Every legal OWL DL ontology is a legal OWL Full ontology
- Every valid OWL Lite conclusion is a valid OWL DL conclusion
- Every valid OWL DL conclusion is a valid OWL Full conclusion

Ontology developers using OWL have to consider what sub-language that best fit their needs. The choice between OWL Lite and OWL DL depends on the expressiveness of the ontology to be developed. The choice between OWL DL and OWL Full mainly depends on the extent to which users require the meta-modeling facilities of RDF Schema (e.g. defining classes of classes, or attaching properties to classes). One also have to consider the fact that the syntactic freedom allowed be OWL full may cause unpredictable reasoning.

OWL Full can be viewed as an extension of RDF, while OWL Lite and OWL DL can be viewed as extensions of a restricted view of RDF. Every OWL document is an RDF document, and every RDF document is an OWL Full document, but only some RDF documents will be a valid OWL Lite or OWL DL document. For this reason, some care has to be taken when one wants to migrate an RDF document to OWL. When the expressiveness of OWL DL or OWL Lite is regarded as appropriate, some precautions have to be taken to ensure that the original RDF document complies with the additional constraints imposed by OWL DL and OWL Lite. One such constraint is that every URI that is used as a class name must e explicitly asserted to be of type owl:Class.

```
Class(MargheritaPizza partial
        Pizza
        restriction(hasTopping
                someValuesFrom(Mozzarella))
        restriction(hasTopping
                someValuesFromTomato)))

Class(CheesyPizza complete
        Pizza
        restriction(hasTopping
                someValuesFrom(Cheese)))
```

Figure 8: OWL example using abstract syntax

This is also the case for properties. Also, every individual must be asserted to belong to at least one class, the URI|s used for classes, properties and individuals must be mutually disjoint. These and other constraints can be found in [3]

OWL DL provides maximum support for expressiveness while simultaneously guaranteeing decidability. Because the latter property means that reasoning can be applied, OWL DL has become a popular choice in ontology based applications. In the remainder of this section, we shall focus on this sub-language.
Similar to RDF-S, OWL DL also consist of statements about resources. But whereas RDF-S statements are triples, OWL DL statements are either axioms or assertions. An axiom is either a class definition, a class axiom or a property axiom. Class definitions can be used to define subclass relationships, as well as property restrictions which hold for a particular class. With class and property axioms, one can express more complex relationships between classes and between properties such as boolean combinations of class descriptions and functional, inverse and transitive properties. Individual assertions can be used to express class membership, property values and equality of individuals.
OWL DL is defined in terms of an abstract syntax. However, since OWL is syntactically embedded into RDF, all of the RDF serializations can be used. RDF/XML is the normative syntax and should be used to exchange information between systems. The RDF representation of an OWL DL ontology can be obtained through a mapping from the abstract syntax. Figure shows a simple OWL DL ontology defining the classes MargheritaPizza and CheesyPizza.

This can be interpreted as: 'All Margherita pizzas have, amongst other things, some mozzarella topping and also some tomato topping'. And 'a cheesy pizza is any pizza that has, amongst other things, some cheese topping'. An in-depth definition of the axioms used in OWL is provided in [4].

## 3.5 Components of OWL

An OWL ontology is built up by three components; *Classes*, *Properties* and *Individuals*. Referring to section3.2, these components are analogous to *concepts*, *relations* and *instances*, respectively. These will be described in more detail in the following subsections.

### 3.5.1 Individuals

Individuals represent objects in the domain we are interested in and can also referred to as instances of classes. It is important to note that OWL does not use the Unique Name Assumption (UNA). This means that two different names could actually refer to the same individual. In OWL one therefore has to explicitly state whether two individuals are the same, or not the same as each other. Or else they *might* be the same, or they *might not* be the same.

### 3.5.2 Properties

Properties are the relation between two individuals, that is a property links an individual to another. For example, the property hasBrother can link the two individuals David and Jonas together. A property may be functional, symmetric or transitive. If a property is functional there can be at most one individual that is related to the individual via the property. For example the property hasBirthMother is a functional property (you can only have one mother).
A symmetric property can be defined as follows: If individual A is related to individual B via property P, then, if P is symmetric, B is also related to A via P. An example of an symmetric property is the property hasSibling; if David has a sibling called Jonas, then Jonas also has a sibling, called David.

A transitive property can be defined as follows: If individual A is related to individual B via property P, and B related to individual C via P - then, if P is transitive, A is related to C via property P. The property hasAncestor can be characterized as transitive; if David has the ancestor Gary, and Gary has the ancestor Kate - then we can infer that David is has the ancestor Kate.

### 3.5.3 Classes

OWL classes can be interpreted as sets containing individuals. They are described using conditions that states precisely what requirements needs to be in

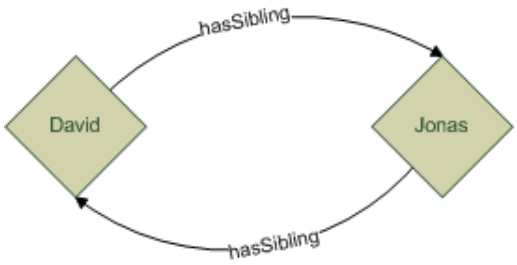


Figure 9: Symmetric property

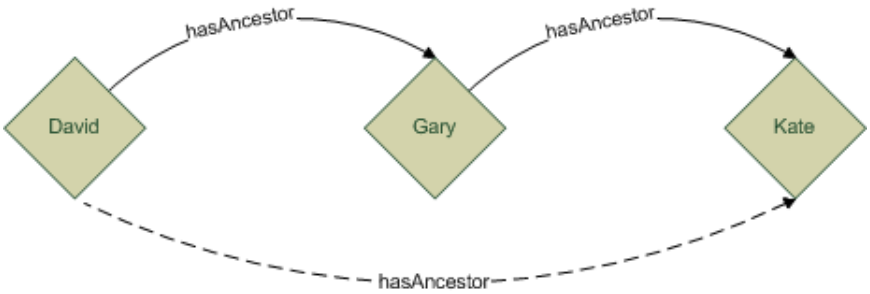


Figure 10: Transitive property

place in order for an individual to be a member of the class. Classes may be organized in a superclass-subclass hierarchy, known as a taxonomy. Using a reasoner, this taxonomy can be computed automatically.

## 3.6 Rules

All though in it's infancy, rules are considered to be a major issue in the further development of the Semantic Web. They can be used in ontology languages, either in conjunction with or as an alternative to description logic's, to draw inferences, to express constraints, to specify policies and/or to react to event/changes. With rules one can express knowledge in the form "IF A THEN B". An example, written in a human readable syntax of the form
*antecedent (body)* $\Rightarrow$ *consequent (head)*, is given below.

$$parent(?x, ?y) \wedge brother(?y, ?z) \Rightarrow uncle(?x, ?z)$$

This example says that if $\boldsymbol{y}$ is the parent of $\boldsymbol{x}$, and $\boldsymbol{z}$ is the brother of $\boldsymbol{y}$, then $\boldsymbol{z}$ is the uncle of $\boldsymbol{x}$.

### 3.6.1 Types of rule languages

SWRL [6] is an extension of OWL DL which adds the expressive power of rules to OWL. The example above can be expressed in SWRL.
SWRL enables Horn-like rules [5] to be combined with an OWL knowledge base. However, whereas Horn rules have a conjunction of atomic formulas in the antecedent of the rule and a single atomic formula in the consequent of the rule, SWRL allows any OWL class description, property or individual assertion in both parts. Since SWRL combines the full expressive power of function-free Horn logic with an expressive description logic language, the key inferences tasks (e.g. satisfiability and entailment) are in general undecidable for SWRL.
Another proposal for a rule language for the Semantic Web is F-Logic [7]. Rules in F-Logic are similar to Horn rules, with the distinction that besides atomic formulas, F-Logic rules also allow molecules in place of atomic formulas.
The main difference between SWRL and F-Logic is that in SWRL, the rule language is seen as an extension of the ontology language OWL DL, whereas in the F-Logic proposal, ontologies are modeled using rules.

## 3.7 Protege-OWL

Writing an OWL directly can be hard and is indeed error prone. Protege-OWL is an open source tool created to support ontology development for the Semantic Web. The tool allows users to edit ontologies in OWL and to use description logic classifiers to maintain consistency of their ontologies. Protege-OWL enables OWL developers to load existing ontologies or to create new ones from scratch using an intuitive user interface where one can visualize classes, properties, individuals as well as SWRL Rules. In addition, the tool is tightly integrated with Jena and has an open-source Java API in which developers can use to create their own Semantic Web applications.
Figure 11 shows a screen-shot of the tool.

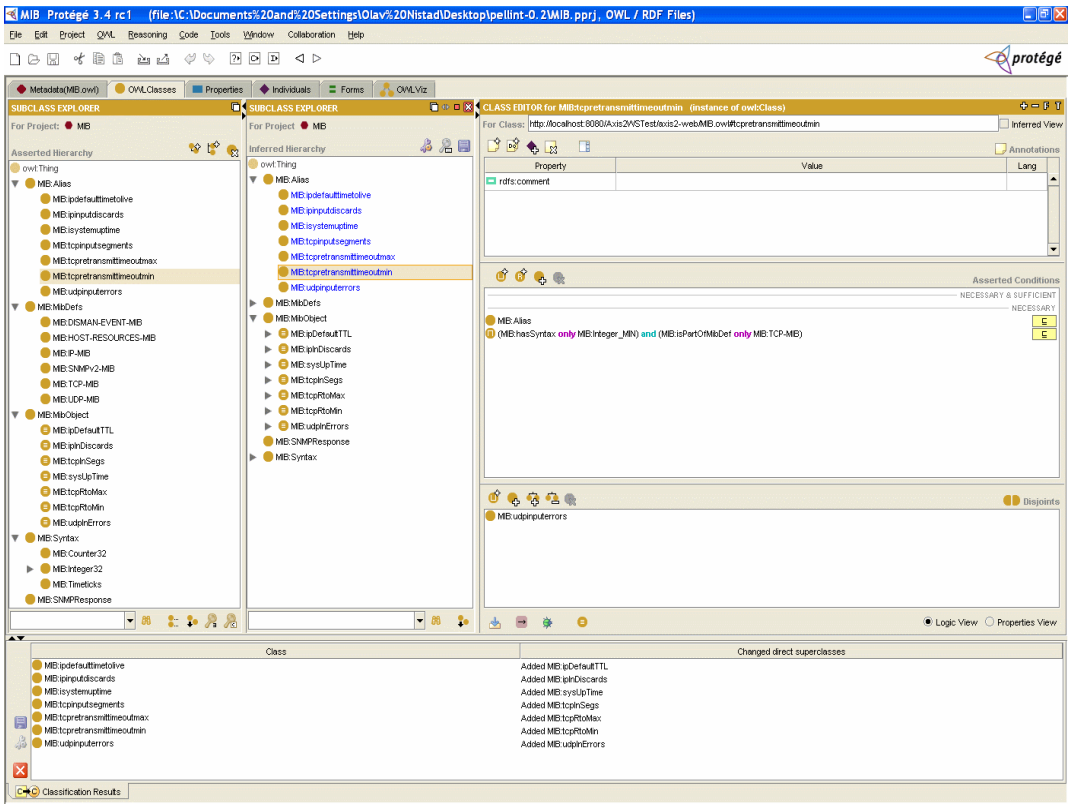


Figure 11: Screen-shot of Protege-OWL

# 4 Semantic Web Services

As we have seen from the last section, the technologies provided by the Semantic Web are working towards a Web where machine-interpretable information is added to enable computers to reason about information and take automated actions. Web Services, on the other hand, are working towards a situation where organizations can make some of their abilities available and accessible via the Internet. This is done by wrapping some computational capability with a Web Service interface and allowing other organization to access it either directly or via some discovery agency (e.g. UDDI). Web Services provides a standard and widely accepted way of defining these interfaces.

Semantic Web Services is an extension of the conventional WS technology where Semantic Web technology combined with traditional Web Services. As we will see in the next subsections such an combination can help provide more precise results as well as a higher degree of automation.

## 4.1 Limitations of current Web Services

Web Service technology has, as earlier mentioned, experienced great success. It is however naive to believe that this is the solution for all problems related to interoperability in heterogeneous systems. All though it provide a communication medium for distributed systems, it have now way of ensuring that all communicating parties "speak the same language" - a feature that is necessary in fully automated system interoperation. As illustrated in figure 12 ,Web Services make use of accepted standards for structure, syntax and vocabulary, but it does not offer the semantics and the pragmatics of the used vocabulary. Nor does it say anything about (in a machine-interpretable form) what the software system does, or what sequence of messages is used to interact with it.

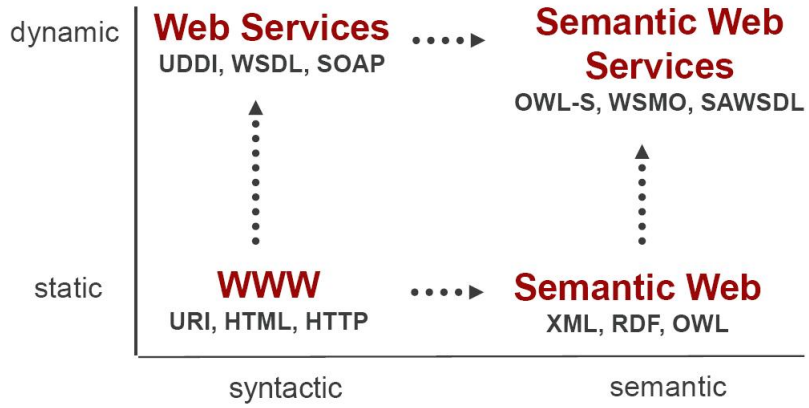


Figure 12: The evolution of the Web

We can overcome this lack by using Semantic Web technology. The term *Semantic Web Services*, stands for the automation of service usage tasks such as discovery, selection composition and enactment of suitable services. This task is accomplished by making the services themselves machine-interpretable. Just like the way in which the Semantic Web promises to make the static content of the Web machine-processable via semantic annotation, the idea of applying similar techniques to Web Services is very appealing. Using Semantic Web Services (SWS) we can annotate software being offered via Web Service interfaces with machine-interpretable descriptions describing what the software does and how it does it. Also, with ontologies able to describe and annotate the various aspects of a Web Service, we are able to automate the tasks of discovering services, composing them, executing them and enabling seamless interoperation between them - thus enabling intelligent Web Services.

Combining these technologies enables many new things to be done. 'Services' as varied as protein analysis, DVD-selling, translation and animation rendering could be advertised and discovered automatically on the Web. A company needing a service could locate a provider they were previously unaware of, set up a short-term business relationship and receive the service in return for a payment. All this could be done automatically and at high speed. Furthermore, several services could be combined into a more complex service, possibly automatically. If one of the component services is unavailable, a replacement could be rapidly

found and inserted, so the complex service can still be provided.

## 4.2 Key Concepts in Semantic Web Services

In the following sub sections, we will describe key concepts used to enable Semantic Web Services, and show how they are related.

### 4.2.1 Service Representation

Before going into the details about Semantic Web Services it is important to have a clear definition of what *service* is. A service can be defined as a something one party has to provide to another when the first party does something for the benefit for the second. For example; a house cleaner may perform the service of doing your home cleaning; a flight attendant may perform the service of bringing you coffee while aboard an aircraft. Formally, one can say that a service is the performance of some actions by one party to provide some value to another party. We call the party which provides the service the service *provider* and the party which is provided the value of the service the service *requester*.

One aim of Semantic Web Services is to carry about a machine-interpretable representation of the service. This representation is referred to as the service *description*. To describe services with semantic annotation, one uses techniques based on knowledge representation. This means that the service has to described in a way which permits reasoning with it. In this regard, we first has to decide what formal language we will use. Should we use horn clause logic, description logic, non-monotonic logic or some other approach? Secondly, what specific concepts and relations should be used to describe the different concepts of the service, what is the meaning of these? This involves the creation of an ontology which provides us with a structured ontological vocabulary, as described in section 3. It is important that the ontology provide a specification of the types of inputs and outputs of the service, as well as the actions the service consists of.

Two parties describing a service may make different choices with regard to the language and ontology used. As a consequence, if one party should reason with a description produced by a different party, then some additional reasoning will be necessary in order to translate between the two approaches. This additional reasoning is called *mediation*.

### 4.2.2 Software agents

It is also important to describe the online representation of a service- provider and requester. If the providing and receiving operations of a service is to be automated, then the two representative parties need some online software component to take care of this. These types of components are called *agents*: a service provider agent will represent the provider, while a service requester agent will represent the requester. It is important to note that the behavior of an agent does not need to be static; the software component can act as a requester agent at one time, and a provider agent at another.

### 4.2.3 Communication

When a service is published and accessible via the Internet, there must be some interaction between the provider and the requester. Such an interaction requires some exchange of messages which follow certain constraints if they are to make sense to both parties. Hence, the message exchange must take place according to some known communication protocol. In this thesis, we follow the definition of the W3C Choreography Working Group [9] and refer to this communication protocol as a *choreography.*

Exchange of messages between two parties according to a certain choreography is referred to as a *conversation.* Furthermore, when two parties engage in a conversation, they must both have one or more communication endpoints to send and receive the messages according to some transport protocol. This is referred to as the *grounding* of the choreography.

### 4.2.4 Orchestration and Service Composition

Choreography puts constraints on the order of messages sent between the requester and the provider. This however is not alone to determine exactly what message is sent when. This responsibility is assigned to the *orchestration*, which is a specification within an agent, of which message should be sent when. In other words, the choreography decides what is permitted of messages, while the orchestration decides what each party will actually do.

The real power of orchestration becomes evident when we look at multiple simultaneous interactions between agents. Instead of a single relationship with one agent acting as a service provider and another agent as a service requester

it is clear that in some circumstances an agent will be involved in several relationships. One service requester can for instance communicate with several service providers and combine and coordinate the different services to produce a larger complex service. The task of composing such a service is called *service composition.*

When a requester have simultaneous conversations with several providers, the orchestration can specify the sequencing of messages with all of these, including appropriate dependencies. Such an specification can be done in several ways. One way is to hard code the integration logic as well as what service providers one wants to use. A more flexible way is to use a workflow language to describe the process of integrating the interaction with the chosen service providers. This approach is used in Business Process Execution Language (BPEL) [10]. The main drawback with this approach is that it depends on reliable and stable service providers. If one of the chosen providers should fail, the overall service orchestration will also fail.

A more failure robust approach, taken by WSMF [11], is not to select the service providers up front, but instead include descriptions of their required functionality. When the orchestration is executed suitable service providers are dynamically discovered and selected at run-time.

Having a explicit definition of a service orchestration means that the orchestration can exist independently of specific service requester, and passed between agents as a data structure. Instead of requiring that only the service requester is responsible for generating an orchestration, this can be done by any party. In particular, in the case where a single provider is hosting several services, it is more convenient that the provider take responsibility for showing how these services can be combined to produce a more complex one. If this is done according some some standard process language, and a service requester is able to interpret that process language, then any such service requester can make use of the complex service. This latter approach is taken by OWL-S [12]. Using OWL-S a service provider can specify how several services can be combined to produce a more complex service. An execution environment constituting an OWL-S Virtual Machine [13, 14] can then be used by the service requester to interpret the process language and interact correctly with the service provider.

### 4.2.5 Life cycle

The life cycle of the relationship between the service provider and service requester can be divided in five different phases: modeling, discovery, service definition and service delivery.

**Service Modeling Phase** Before a service requester can discover a service, it has to create a description of the service it is interested in. Since it is unlikely that the service provider and the specifics of the service is known, only an abstract description will be made. This abstract description specifies the requester's capability requirements of the service. Similarly, service providers create an abstract service capability description representing the service it is able to provide.

**Service Discovery Phase** After a service provider and service requester have created their respective service descriptions, the former has to publish it's description in some registry where the latter can locate ("discover") it. In current Web Service technology, this task is carried through by use of UDDI. All though a powerful service registry tool, the standard version only supports keyword search. In the context of Semantic Web Services, we need a registry which supports semantic annotations of service capabilities via decentralized ontologies, interconnected via logical axioms. In such a registry a service discovery match could be determined through the use of logical inference. The service descriptions could also involve more fine-grained notions such as formal descriptions of *preconditions* and *postconditions*, and of the *inputs* and *outputs* of the service, using terms specified in an ontology. In addition a service discovery match can be based on non-functional properties [31]

**Service Negotiation Phase** During discovery, a service requester may find several services from several service providers that meets it's needs. From the set of providers found, the requester needs to analyze their service descriptions and somehow decide which one is the "best". To decide this a provider has to refine it's abstract service description into a more concrete one. One can think of it as instantiating the abstract descriptions attributes. When a suitable provider has been determined to serve a needed goal, it is necessary to negotiate a service instance from the possibly many services a provider can offer. This may include

establishment of trust policies, determination of payment modalities, selection of offers, etc. where corresponding semantic annotations are required. For the purpose of automating this task, it is important that a semantic service description not only specifies the functional properties of the service, but also the non-functional properties, such as supported policies, and security protocols.

**Service Composition Phase** In cases where a particular goal cannot be achieved by a single service, semantic description can help to determine a composition of several services that combined achieves the goals of a service requester. Composition requires not only the semantic annotations of the overall capabilities of a service, but also a behavioral description of how to interact with the service, in order to achieve a certain functionality.

**Service Invocation Phase** After a - either composed or single - service has been selected, the next and final step is the execution. To this end, possible input and output values need to be extracted from the semantic capability description and adapted to the negotiated message formats and communication protocols.

The full power of Semantic Web Services is achieved when the steps above can not only be fulfilled, but also be (as much as possibly) automated. All though several different approaches exist for achieving this, they all aim at the annotation of Web Service description per service, by extending or complementing current technologies around WSDL, SOAP, UDDI etc with semantic annotation. In the next sub section the approach used in the implementation part of this thesis will be described.

# 5 A Semantic Web Service Approach - OWL-S

OWL-S [12] is an effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, Yale University and SRI International to define an ontology for semantic markup of Web Services. As an OWL-based Web Service ontology it supplies Web Service providers with a core set of markup language constructs for describing the properties and capabilities of their Web Services in an unambiguous, computer-interpretable form.

## 5.1 Atomic and Composite Services

OWL-S is supposed to cover both "atomic" and "composite" services. An atomic service is an indivisible software component that executes small and non-complex operations. Most executions only consist of a single operation in order to respond to the service requester. Examples of atomic services are a service returning the temperature given a zip code, and a service returning the account balance given a bank account number.

An composite service is an software entity that combines several "smaller" operations in order to respond to the service requester. For example, a service which returns both the checking account balance and the savings account balance given a persons ID number, can be defined as a composite service.

## 5.2 Main Tasks Enabled by OWL-S

In the development process of OWL-S, three main tasks have been given special attention :

1. Automatic Web Service discovery
   Automatic Web Service discovery is the automated process of locating a service that can provide the needed service capabilities needed by a service requester, while adhering to some client-specified constraints. For example, the user may want to find a service that sells airline tickets between two given cities and accepts a particular credit card. Currently, this task must be accomplished by a human who might use a search engine to find a service, read the Web page, and then execute the service manually to determine if it satisfies his or hers constraints. With OWL-S markup of services, the information necessary for Web Service discovery could be specified as machine-interpretable semantic markup at the service provider's Web site. Also the service can be advertise itself in OWL-S with a service registry, so that requesters can find it when they query the registry. Thus, OWL-S enables declarative advertisements of service properties and capabilities that can be used for automatic service discovery.

2. Automatic Web Service Invocation
   Automatic Web Service Invocation is the automated process of invoking a service given only a declarative description of that service. This is in

contrast to the situation where the service requester agent has been pre-programmed to call that particular service. This enables the possibility of not only locating a service which offer cheap airline tickets, but also to carry out the purchase of that ticket. OWL-S markup of Web Services provides a declarative, machine-interpretable API that includes the semantics of the arguments to be specified when executing these calls, and the semantics of the output which is returned after execution of the service. The service requester agent should be able to interpret this markup to understand what input is necessary to invoke the service, and what information will be returned. OWL-S, in conjunction with domain ontologies specified in OWL, provides standard means of specifying declarative APIs for Web services that enable this kind of automated Web Service execution.

3. Automatic Web Service composition and interoperation
   This task involves the automatic selection, composition and interoperation of Web Services to perform some complex task, given a high-level description of an objective. For example, the user may want to make all the travel arrangements for a trip to a conference. Currently, the user must select the Web Services, specify the composition manually, and make sure that any software needed for the interoperation of services that must share information is custom-create. With OWl-S markup of Web Services, the information necessary to select and compose services will be encoded at the service Web sites. Software can be written to manipulate these representations, together with a specification of the objectives of the task, to achieve the task automatically. To support this, OWL-S provides declarative specifications of the prerequisites and consequences of application of individual services, and a language for describing service compositions and data flow interactions.

## 5.3 OWL-S Description

The structuring of OWL-S services is motivated by the need to address three aspects of a service:

- What does the service provide for prospective clients?
- How is it used?

- How does one interact with it?

To address these questions OWL-S defines an upper ontology for services with four major elements:

1. Service: This concept serves as an organizational point of reference for declaring Web Services; every service is declared by creating an instance of the Service concept

2. Service Profile Model: The profile provides an abstract description of what the service does, describing its functionality and other non-functional properties that are used for locating services based on their semantic description

3. Service Process Model: The process model describes how the service achieves its functionality, including the detailed description of its constituent processes.

4. Service Grounding: The grounding describes how to use the service, that is how a client can actually invoke the service.

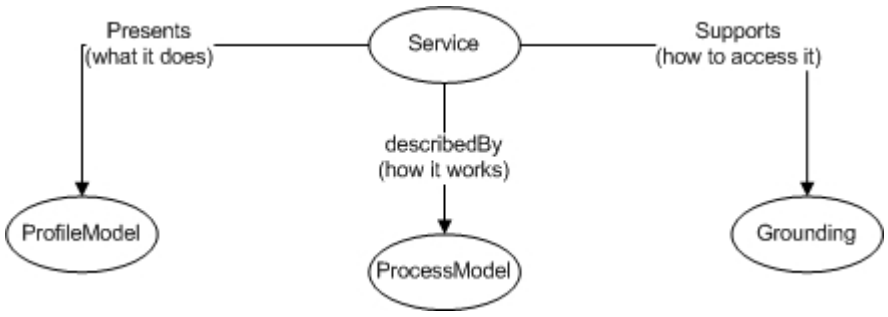


Figure 13: OWL-S Upper ontology

### 5.3.1 OWL-S Service

The *Service* concept in OWL-S links the profile model, process model and grounding of a given service through the properties: presents, describedBy and supports, respectively. Below is an example of the Service concept of a Zip Code Finder service which task is to return the zip code for a given city/state.

**Example 5.1.** ZipCodeFinder - Service description

```
<rdf:RDF xml:base=http://www.examplesite.com/ZipCodeFinder.owl>
        <service:Service rdf:ID=ZipCodeFinderService>
                <service:presents rdf:resource=#ZipCodeFinderProfile/>
                <service:describedBy rdf:resource=#ZipCodeFinderProcess/>
                <service:supports rdf:resource=#ZipCodeFinderGrounding/>
        </service:Service>
        [...]
</rdf:RDF>
```

### 5.3.2 OWL-S Service Profile Model

The profile model describes the intended purpose of the service, both describing the service offered by the provider, and the service desired by the requester. It is thus this description that is used in the publish/discovery phase described in earlier. The profile model gives an abstract description of both non-functional and functional properties.

**Non-functional properties** The non-functional properties includes human-readable information, contained in the properties *serviceName* (of type string; maximum one), *textDescription* (type string, maximum one) and *contactInformation* (of class *Actor*), including information such as name, phone, fax and/or e-mail. A service categorisation is also given, although the classification schemas are not fixed and, therefore, the range of this property is not specified. There are no cardinality restrictions for the categorization, that is, a service can be assigned to none or multiple categories in different categorization schemes. The profile model of the ZipCodeFinder service is defined as follows (service categorization omitted) :

**Example 5.2.** Non-functional properties

```
<profile:serviceName>Find ZipCode</profile:serviceName>
<profile:textDescription>
        Returns the zip code for the given city and     state.
        If several zip codes are associated with the zip        code,
        the first one will be returned
</profile:textDescription>
<profile:contactInformation>
        <actor:Actor rdf:ID=ZipCode Service>
```

```
                <actor:name>ZipCodeService department</actor:name>
                <actor:phone>61177393</actor:phone>
                <actor:email>zipcode@superservices.com</email>
                [...]
        </actor:Actor>
</profile:contactInformation>
```

**Functional properties** The OWL-S profile also specifies what functionality the service provides. The functional properties is split into the *information transformation* performed by the service and the *state change* as a consequence of the service execution. The former is captured by defining the *inputs* and *outputs* of the service, and the latter is defined in terms of *preconditions* and *effects*. Inputs, outputs, preconditions and effects are commonly referred to as IOPEs. Effects are defined as part of a *result*. The schema for describing IOPEs is not defined in the profile, but in the OWL-S process model. Instances of IOPEs are created in the process and referenced from the profile, and it is envisioned that the IOPEs of the profile are a subset of those published by the process [12].
The inputs and outputs describes what information is required to execute the service, and what will be returned. The two types are modeled as subclasses of *parameter*, which is in turn a subclass of a SWRL variable [6] with a property indicating the class or datatype the values of the parameter belong to. Local variables may also be used, and they are modeled as subclasses of *parameter*. Inputs, outputs, local variables have as scope the process where they appear. The inputs and outputs defined in the service process model are referenced from the profile via the *hasInput* and *hasOutput* properties.

In the ZipCodeFinder service, the inputs and outputs are declared as follows:

**Example 5.3.** Functional properties

```
<profile:hasInput rdf:resource=#City />
<profile:hasInput rdf:resource=#State />
<profile:hasOutput rdf:resource=#ZipCode />
[...]
```

The inputs and outputs referenced from the profile are defined in the process as part of the different atomic processes where they appear:

```
<process:Input rdf:ID=City>
```

```
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI>
        http://www.w3.org/2001/XMLSchema#string</process:parameterType>
</process:Input>
<process:Input rdf:ID=State>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI>
        http://www.w3.org/2001/XMLSchema#string</process:parameterType>
</process:Input>
<process:Output rdf:ID=ZipCode>
        <process:parameterType rdf:datatype=http://www.w3.org/2001/XMLSchema#anyURI>
        http://www.daml.org/2001/10/html/zipcode-ont#ZipCode
        </process:parameterType>
</process:Output>
[...]
```

Preconditions are conditions on the state of the world that has to be true for successfully executing the service. That is, if the preconditions are not met, the service will not execute. They are modeled as *conditions*, a subclass of *expression*. Expressions in OWL-S specify the language in which the expression is described (most commonly this is either SWRL[6] or SPARQL [16].) and the expression itself is encoded as a literal. Effects describe conditions on the state of the world that are true after the service execution. They are modeled as part of a result. A result has an *inCondition*, a *ResultVar*, an *OutputBinding* and *Effect*. The inCondition specifies the condition for the delivery of the result. The OutputBinding binds the declared output to the appropriate type or value depending on the inCondition. The effects describe the state of the world resulting from the execution of the service. The ResultVars play the role of local variables for describing results. Conditions, i.e. preconditions defined in the service model, are referenced from the profile via the *hasPrecondition* property and results via the *hasResult* property.

### 5.3.3 OWL-S Process Model

The Process Model of a OWL-S description represents how the service works, that is, how to interoperate with the service. It describes the functional properties of the service, together with details of its constituent processes (if the service is a composite service), describing how to interact with the service.

**Atomic Processes** OWL-S distinguishes between atomic, and composite processes. Atomic Processes can be invoked, have no subprocesses and are executed

in a single step from the requester's point of view. They are a subclass of *process,* and therefore, they specify their inputs, outputs, preconditions and effects. All though the ZipCodeFinder service only defines one atomic process, there are no restrictions on the number of atomic processes inside one OWL-S description.

**Example 5.4.** Atomic Process

```
<process:AtomicProcess rdf:ID="ZipCodeFinderProcess">
        <process:hasInput rdf:resource="#City"/>
        <process:hasInput rdf:resource="#State"/>
        <process:hasOutput rdf:resource="#ZipCode"/>
</process:AtomicProcess>
```

**Composite Processes** OWL-S composites are decomposable into other processes. OWL-S provides a set of control constructs such as *sequence* or *split* which are used to define the control flow inside the composite process. Processes are annotated using the *binding* class. A binding is declared as a process which consumes data from other processes which declares what other process and which concrete process parameter the data comes from. Since the ZipCodeFinder service only has an atomic process, and not any composite, the example below is taken from a service called BravoAirService[5], an imaginary flight booking service.

In the example we se a definition of the composite process of BravoAir for booking a flight. It is a sequence of processes, from which the first one is to perform a log-in, and the second one is to complete a reservation. The process for completing the reservation takes data from the parent process, and uses it as the input for its own ChosenFlight input.

**Example 5.5.** Composite Process

```
<process:CompositeProcess rdf:ID=BookFlight>
 <process:composedOf>
  <process:Sequence>
   <process:components>
    <process:ControlConstructList>
     <list:first>
      <process:Perform rdf:ID=PerformLogin>
       <process:process rdf:resource=#LogIn/>
      </process:Perform>
     </list:first>
```

[5] http://www.daml.org/services/owl-s/1.1/BravoAirService.owl

```
      <list:rest>
       <process:ControlConstructList>
        <list:first>
         <process:Perform>
          <process:process rdf:resource=#CompleteReservation/>
          <process:hasDataFrom>
           <process:Binding>
            <process:toParam rdf:resource=#ChosenFlight/>
            <process:valueSource>
             <process:valueOf>
              <process:theVar rdf:resource=#ChosenFlight/>
              <process:fromProcess rdf:resource=Process.owl#TheParentPerform/>
[..]
```

### 5.3.4 OWL-S Service Grounding

The grounding in a OWL-S description provides the details of how to access the service, mapping from abstract to a concrete specification of the service. OWL-S links a Web Service to its grounding by using the property *supports*. A Web Service can have multiple groundings, but a grounding must be associated with exactly one service. These groundings are associated with the atomic processes defined in the Process Model, although this association is not described in the model but only in the grounding. Therefore, the groundings for the atomic processes of the model can be located only by navigating from the Process Model to the service (via the *describes* property), and from there to the service grounding (via the *supports* property).

OWL-S does not dictate the grounding mechanism to be used. However, the current version of OWL-S provides a predefined grounding for WSDL, mapping the different elements of the Web Service to a WSDL interface. An OWL-S atomic process is mapped to a WSDL operation, and inputs and outputs to the WSDL input and output message parts, respectively.

## 5.4 Tools for developing OWL-S based Semantic Web Services

Development and deployment of Semantic Web Services is a quite complex task, and its adoption within the industry has been relatively slow. An important reason for this is the significant human effort required to create semantic service offer- and request descriptions and then to monitor the invocation and execution of the Web Services. A number of tools and systems have therefore

been developed within the Semantic Web Services community to provide the developer with support for semantic annotation o Web Services as well as their deployment.

In this sub section we will present some of these tools.

### 5.4.1 Semantic Web Service tools

Semantic Web Services is essentially only about adding semantic annotations to Web Services. For this reason, many of the tools for creating Semantic Web Services are extensions on existing and established Web Services tools. In most cases, they are 'tool-lets' rather than tools, being small programs that perform a narrowly defined task, such as automatically generating WSDL descriptions from Java classes. In the following, we will go through some of these tools in detail.

**Java2WSDL and WSDL2Java** Java2WSDL generates WSDL descriptions from Java classes. It is part of the Apache Axis SOAP toolkit [20], an Apache open source software development project. The same toolkit also provides WSDL2Java, which generates Java stubs and skeletons for the Web Service. To create a Web Service, the developer can first create a Java interface of the Web Service, which can then be used to develop WSDL descriptions for the Web Service using Java2WSDL. The resulting WSDL description can then be used to create stubs, skeletons and bindings using the WSDL2Java tool.

**WSDL to OWL-S tool** There exists different implementations [6] [7]of a tool for transforming a WSDL description to an OWL-S description. They all have in common that they convert a WSDL descriptions into OWL-S descriptions by generating a complete OWL-S Grounding, a partial OWL-S Process model and Profile for the WSDL service. The generated Grounding is clearly complete, since the WSDL file contains all the information necessary to invoke the Web Service. However, the WSDL file is only a partial description of the Web Service, so the generated Process Model and Profile are thus only partial and need to be manually enriched with semantic information. This includes defining composite processes in the Process Model, describing the service capability descriptions

[6] http://www.daml.ri.cmu.edu/wsdl2owls/
[7] http://www.mindswap.org/2004/owl-s/api/doc/javadoc/examples/WSDL2OWLS.html

within the Profile and XSLT transformations from the WSDL XSD types to OWL ontologies.

**Java to OWL-S tool** There also exists a tool[8] for a direct conversion from a Java class to a OWL-S description. This tool combines the Axis Java2WSDL converter and a WSDL-to-OWL-S converter to provide a complete OWL-S Grounding as well as partial OWL-S Process Model and Profile.

**OWL-S API** In addition to these tools, several Semantic Web Service environments also make use of an OWL-S API which provides programmatic access to OWL-S service descriptions. Two such API's has been developed independently by both CMU [9] and University of Maryland [10]. These APIs provides Java classes and methods to extract information from an OWL-S description or to generate an OWL-S description. They also contain a execution environment in order to invoke OWL-S described services.

[8] http://projects.semwebcentral.org/projects/java2owl-s/
[9] http://www.daml.ri.cmu.edu/owlsapi/
[10] http://www.mindswap.org/2004/owl-s/api/

Part II

# SWS applied to a Network Management System

# 6 Introduction

In large communication networks there is a need for a Network Management System (NMS) to handle the tasks of monitoring and managing network devices. Typical management facilities in such systems are fault management, configuration management, performance management and security management. In order to perform these management tasks, the system needs to monitor each device on the network. Unfortunately, several of these monitoring schemes have some crucial drawbacks that leads to unsatisfactory performance. The main drawback is rooted in the fact that these systems are typically designed in a way that puts all management computation tasks on a centralized server. This puts large demands on the performance of this one server, and also causes it to be a singel point of failure. Furthermore, as the network grows in size, huge amount of raw data is transfered to the this management station, causing huge traffic on the network.

To accomodate this problem, several efforts have been made to relieve the centralized server from some of the work, and instead delegate some of the management tasks to other computers in the network. [18]. This approach is refered to as the Management by Delegation (MdB) model and is today widely accepted and recognized by the network management community. One of the main features of this decentralized approach is the ability to transfer and remotely control management scripts located on remote entities, which leads to the ability to delegate management functions along the management system, therefore decentralizing the management operations.

The sections in this part are divided as follows:

- Section 7 will present the specification of our network monitoring system.
- The following section will describe discuss which parts of this system may benefit from being applied with SW technology.
- Eventually, section 9 will specify a remake of the system described in section 7 applied with ontologies and SWS technology.

# 7 SNMP-based Monitoring Application By Using TAPAS Platform

In a project thesis [17] carried out during the fall of 2008, we presented an specification for a decentralized Network Monitoring System. An SNMP-based monitoring application was implemented that run on a TAPAS platform [19]. This system is comprised by two main components; a MainManager and a MiniManager. The former component is deployed at a central location - in a device functioning as the management station. When a user wants to monitor on a device, a request is sent to the MainManager which then spawns a MiniManager component which is deployed at another device functioning as a delegated management station, from now on abbreviated as *DMS*. What DMS a MiniManager will be deployed to is based on how "close" the DMS is to the device to be monitored. The system will try to pick a DMS that is located as close to the device to be monitored as possible. *Close* in this context is determined by the number of hops between the DMS and the device. Most preferably the device to be monitored should only reside one hop away from the DMS. The MiniManager works as an independent "micro-NMS", that is, it is able to spawn agents at the devices to be monitored, initiate SNMP requests and analyse the following responses. Only when certain tresholds and/or a pre-set time period is reached, it notifies the MainManager about SNMP query results. If a new device is to be monitored, the MainManager can choose to add a new monitor session to an existing MiniManager, or deploy a new MiniManager at a different DMS. Figure 14 shows the basic architecture of the system. The rationale behind this system scheme is that by deploying MiniManagers at DMSs, monitoring intelligence is distributed to other devices in the network, releaving a central station from all the work. Analyzing and reasoning about management information can be done by the MiniManagers, instead of putting all the responsibility on the central management station. This relieves the management station for much of the computation responsibility, as well as reducing the amount of network trafic caused by management information.

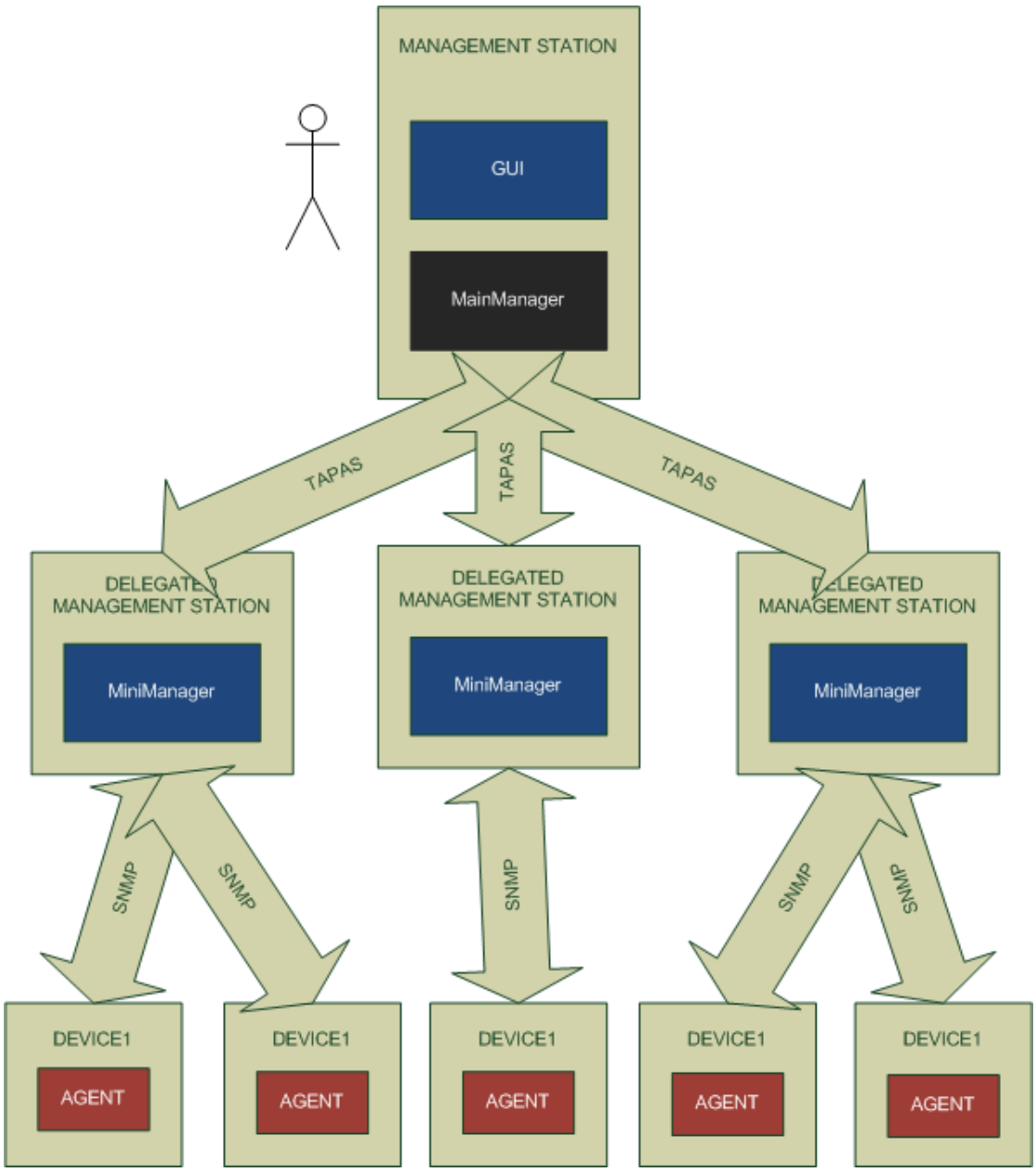


Figure 14: TAPAS SNMP-Based Monitoring Application version 1 - basic architecture

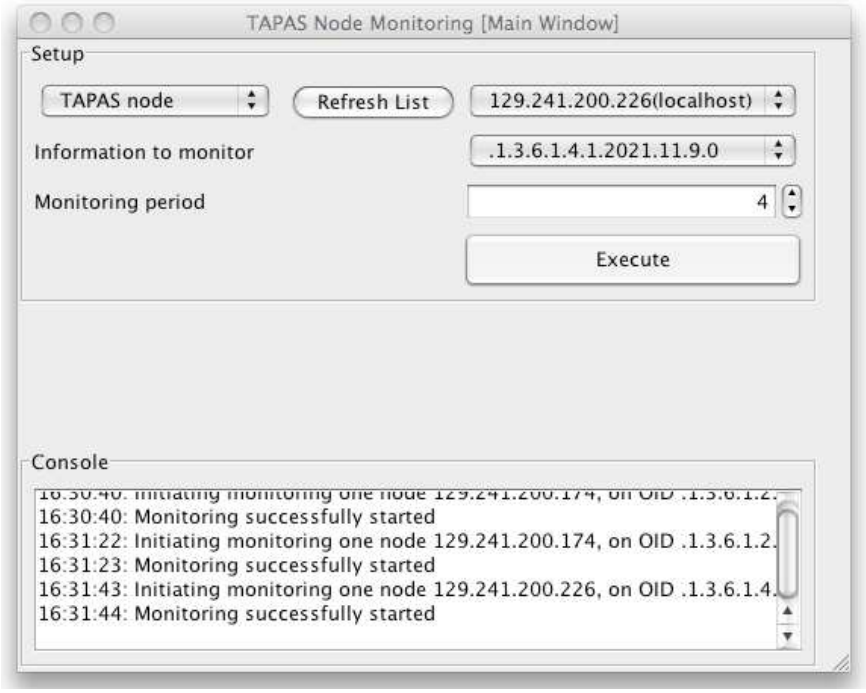


Figure 15: Main window in GUI of version 1

## 7.1 Graphical User Interface

In the curent version of the application the user who launches the application is presented to a simple graphical user interface (fig. 15) which gives him/her several options:

- Select or input what node to monitor
- Select what information to monitor. This information is represented by an Object Identifier (OID) which can be typed either numerical or named.
- Select the time period for monitor feedback

The feedback of a monitor session is presented to the user in an additional GUI window (fig. 16) that pops up when the user initiates a monitor session. For every new monitor session, a new window is opened. Closing one such window is equivalent to ending the corresponding monitor session.

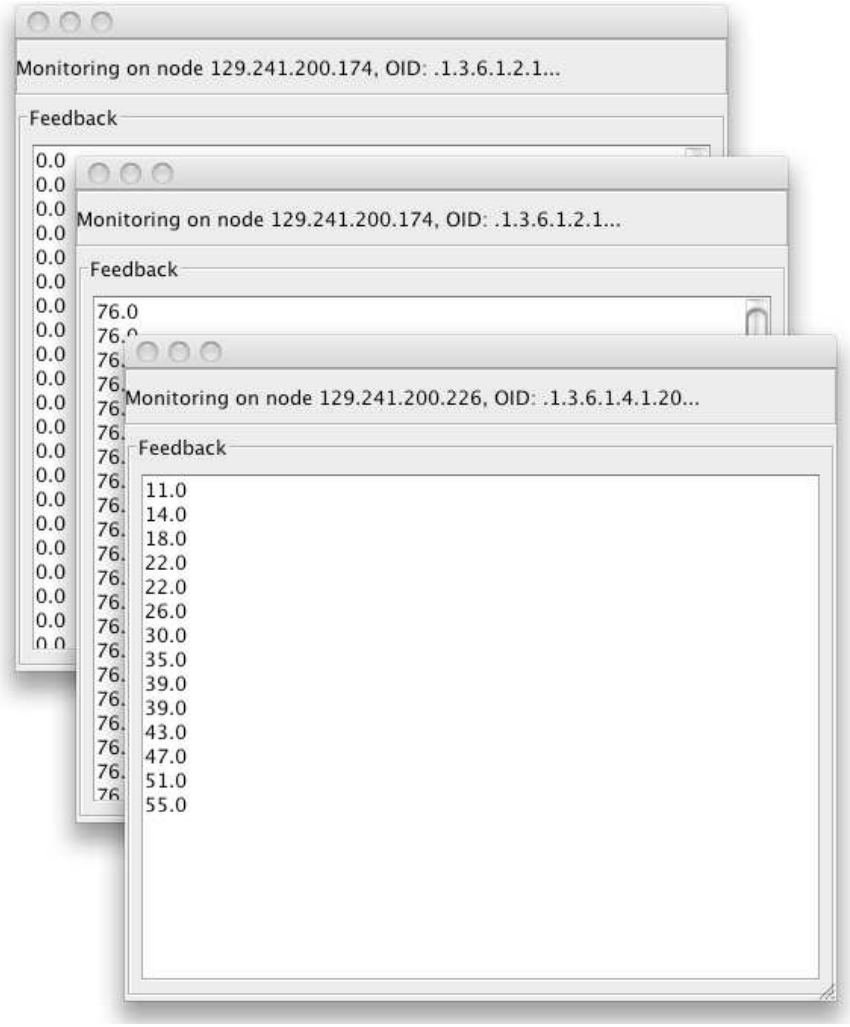


Figure 16: Monitor session window in GUI of version 1

## 7.2 TAPAS: Telematics Architecture for Plug-and-Play Systems

The monitoring system involves running a MiniManager at specific devices. This means that there is a need for easy and automatic deployment and instantiation of this component at the nodes. There exists several different network-based service systems that can handle this task more or less automaticly, and we have

selected TAPAS (Telematics Architecture for Plug-and-Play Systems) for this purpose. TAPAS is a research project where the goal is to develop an architecture for network-based service systems where the main object is to enable dynamic configuration of network components and network-based service functionality. This task is achieved by enhancing flexibility, efficiency and simplicity of system installation, deployment, operation, management and maintenance. The TAPAS architecture is built up by two main architectures; a computing architecture and a service functionality architecture. While the latter architecture has focus on the service functionality and shows the structure of services and service components, the former is a generic architecture for the specification and execution of any service. The TAPAS architecture involves support for dynamic service instantiation that is denoted as the TAPAS platform. The TAPAS platform comprises service creation, deployment, executio and management. [19].

### 7.2.1 Theatre Metaphore

The computing architecture of TAPAS is based on a theatre metaphore: actors play roles according to predefined manuscripts. The actor is a software component that will be part of the TAPAS platform that runs in every node in the networked system. The actor itself does nothing before it is assigned a role. A role is defined by a manuscript and describes a specific behavior that the actor should behave according to. Once an actor has been assigned a role, it becomes a role figure. A play consists of one or more actors playing different roles. Two different role figures can exchange information through a dialogue. A service system can therefore be seen as several actors each implementing a role figure that constitutes a particular service component.

Our application is therefore designed as a TAPAS play, consisting of the two roles MainManagerRole and MiniManagerRole. The behavior of these components is described through two manuscripts that is placed in a repository available to all devices running the TAPAS platform.

### 7.2.2 Plug-and-Play (PaP)

A service system in the TAPAS context consists of several service components which is designed as roles according to a manuscript. A manuscript will, together with other manuscripts part of the same play, reside inside a Play-repository that is located in a web server available to all nodes in the network. In the network

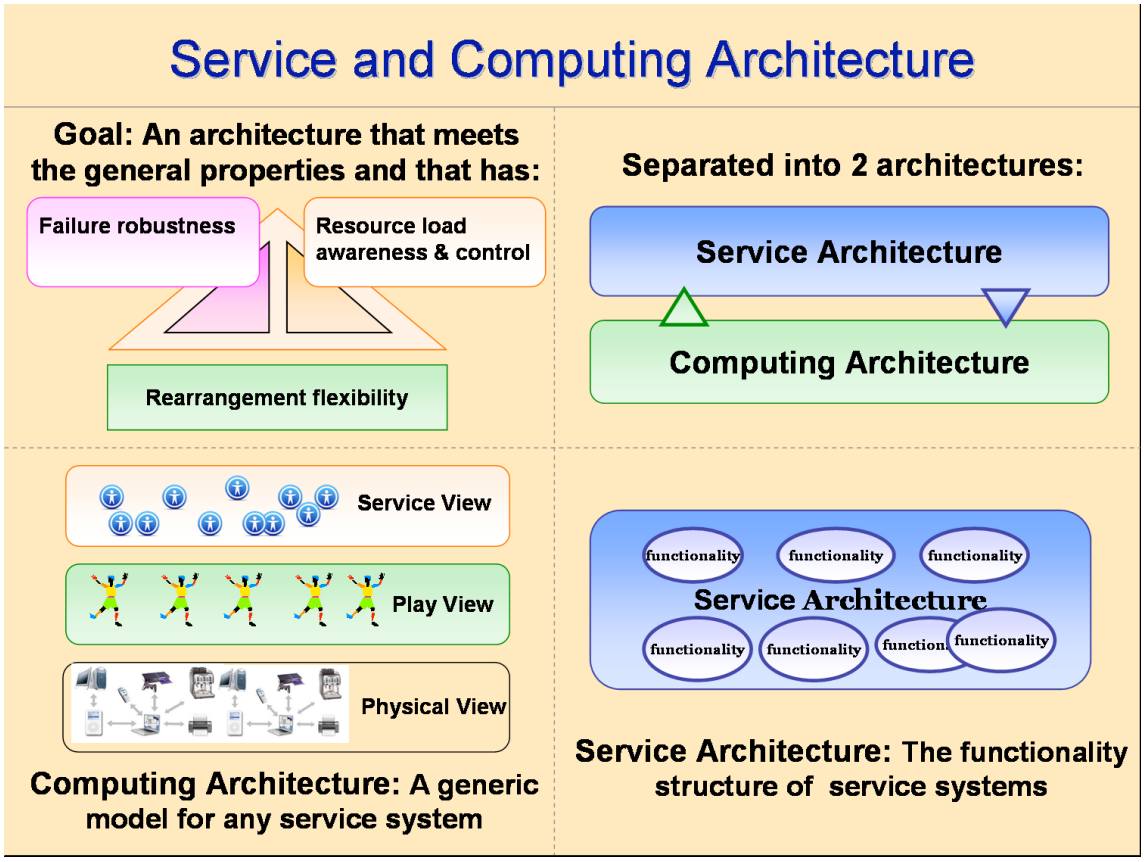


Figure 17: TAPAS Service and Computing Architecture

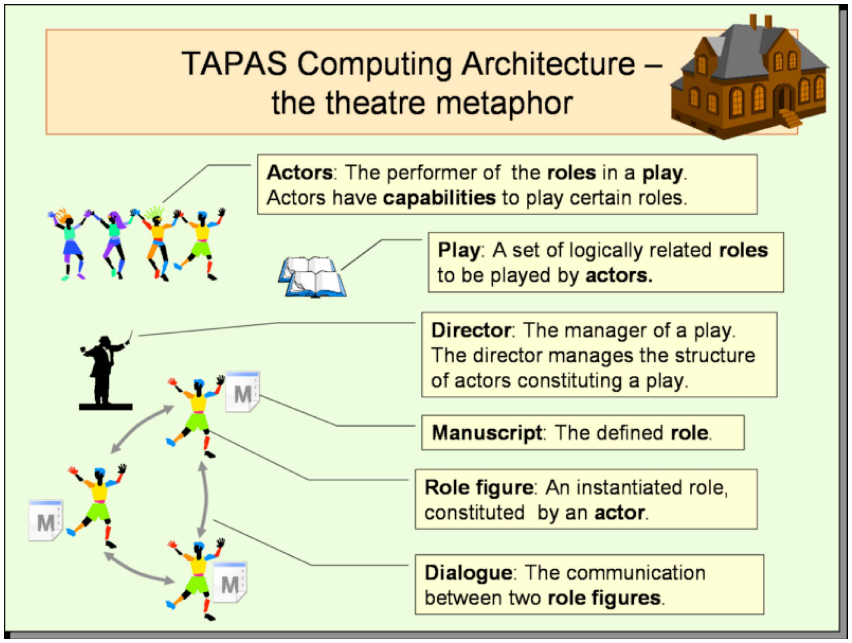


Figure 18: TAPAS Theatre Metaphore

there is also a dedicated server called the Tapas Main Server. This server runs the Tapas Platform as well as an director. The director is a special type of actor that is instantiated and when the Tapas Main Server is launched. The director is responsible for supervising other actors and managing a play. The operations of launching the monitor application and deploying the MainManager and MiniManager are both dependent on the director. That is, the requests to perform these operations are sent to the director which eventually carries out the operations (if they are valid).

In the TAPAS terminology, the process of launching an service system is analogous to launching a "play", or more precisely, to "plug-in" a play. Thereafter, the service system is executed by "playing" it. The task of plugging in a play is as mentioned carried out by the director. A play must consist of at least one manuscript providing a behavior description of a system component (i.e. a TAPAS Role). If specified in the manuscript the director can plug-in and execute the roles at plug-in time. Other roles that are part of the play may be plugged in and executed at later time, but during runtime.

In our SNMP application, the launching of the application causes the MainManagerRole to be plugged-in and executed at launch-time. The component will always be deployed at the device where the plug-in request originated from. The MiniManagerRole(s) will not be plugged in before the MainManagerRole requests it. At any time, the MiniManagerRole(s) can be plugged in and out during runtime. A request to plug-out the MainManagerRole is analogous to plug-out the play (i.e. to terminate the SNMP Application). This is pretty obvious since the application will have no purpose without the MainManager component.

To summarize, our SNMP application consists of two main components - the MainManager and the MiniManager, which again are comprised by several subcomponents. Both of them includes a subcomponent designed as a TAPAS Role. The TAPAS Platform includes plug-and-play functionality meaning that any software component designed as a TAPAS Role is able to be plugged-in (deployed) and plugged-out (un-deployed) - in runtime - at any device in the network running the TAPAS Platform. The TAPAS Platform also includes support for messaging (dialogue) between roles. As you can see from figure 14 the communication between the MainManager and it's deployed MiniManager's are carried out using TAPAS messaging.

# 8 Application improvement using Semantic Web Technology

In this section we will investigate how the current version of the SNMP application can be further improved be using Semantic Web Technology.

From now on, the current version of the SNMP application will be referred to as version 1, and the improved version, which will be described shortly, will be referred to as version 2.

## 8.1 Drawbacks in version 1

Before we can improve the application we first have to locate the areas which can be improved. Of course, version 1 of the application is a fairly simple monitoring application that is far from complete, and several aspects can be further

developed to make the application more powerful. However, for this thesis we have located three specific aspects that will be presented in the following subsections.

### 8.1.1 GUI and MainManager must co-exist at the same device

As depicted in figure 14, the GUI and MainManager is both existing in the same device (management station). In fact, when the user launches the application, the MainManager is first deployed to the device the launch request came from - the first thing the MainManager does is then to create this GUI and make it visible for the user. This was regarded as a smart feature at the time since having both components at the same device, enabled communication between them to take place based on simple and java object invocations. This does, however, impose a severe limitation since it requires the user to be sitting at a fairly powerful computer. In version 2 of the application we have moved the GUI out of the management station and made all communication between the GUI and the MainManager to take place remotely. This enables the user to use any type of computer he wants - in theory he could even use a cellular phone to communicate with the MainManager. Communication between the GUI and the rest of the system will be carried out through invoking Semantic Web Services.

### 8.1.2 Object Identifier input

As one can see from the main window GUI (fig. 15), the application requires the user to know the exact OID for the MIB object to be monitored. For example, if the user would like to monitor disk capacity, he has to enter the OID *hrDiskStorageCapacity.11* (named) or *.1.3.6.1.2.1.25.3.6.1.4.11* (numerical)[11]. Or if he wants to monitor the number of received TCP segments he has to enter the OID *tcpInSegs.0* (named) or *.1.3.6.1.2.1.6.10.0* (numerical). If these OID's are not entered exactly as they are defined in their corresponding MIB-files, a SNMP-query may not be performed successfully. Clearly, it is alot to ask of a user to know all these OID's. No matter how much knowledge the user has about SNMP - knowing OID's by heart is an unreasonable requirement in a monitoring application.

[11]The number after the last dot may vary on different machines. Also, this number helps seperate different disks from each other if the machine contains multiple disks.

In version 2 we have tried to remove this requirement; instead of entering the OID's exactly as they are defined, the user may enter inputs like 'diskcapacity' or 'tcpinputsegments' and let the system itself interpret and translate these inputs to the correct OID's. This approach is accomplished by using Semantic Web Services and will be described in more detail in section 9.

### 8.1.3 Choice of MiniManager

As described earlier, when a user wants to monitor a device for the first time, a MiniManager is deployed to a DMS. Subsequent monitor sessions is realised by adding a monitor session to an existing MiniManager, or locating a new DMS and deploy a new MiniManager. In the version 1 there only exist one type of MiniManager, that is, the same MiniManager component will be deployed in every DMS. This means that the same analyzing operations will be performed regardless of the resource capabilities of the DMS. A network is, however, often very heterogeneous in that it is composed by different types of devices with different resource capabilities. The execution of the monitor session should most preferably as little as possible interfere with other processing taking place at the DMS. At the same time, if the DMS is a powerful device, with a good CPU-power and a alot of free main memory, the MiniManager should be able to take advantage of this and perform more advanced analyzing mechanisms.

For this reason one should make different types of MiniManagers designed for different types of DMSs. By recognizing the difference in resource availability of the different network devices, the MiniManager may enforce flexible and efficient use of resources. For devices with a lot of resources when it comes to memory and CPU, the MiniManager can be developed to handle alot of concurrent monitor sessions with powerful analyzing mechanisms. Therby making full use of local computation. For other devices with fewer resources, simpler MiniManager's may be a better alternative.

The problem in version 1 is not that it isn't possible to create different MiniManager's, the problem is that there doesn't exist any rule one which one to choose. The application could be extended to let the user choose the type of MiniManager to deploy, but it is somewhat unreasonable to require that the user knows the resource-specifics of every DMS in the network and hence what MiniManager to select. A much better approach, that is taken in version 2, is therefore to make the system make this choice. Based on a match between

the MiniManagers capabilities and the DMSs capabilities, the system itself can locate and deploy the "best" MiniManager.

The above mentioned drawbacks has led us to conclude that the application would indeed benifit from being enriched with Semantic Web technology. The next sections we will present the architecture and specifics of our improved application.

# 9 An SWS Enriched SNMP-Based Monitoring Application

Our SNMP Application version 2 is an extension of version 1 in which ontology and semantic information is added to the application. All interaction between the user and the NMS will take place by invoking a set of semantic web services. The following sub-sections will describe version 2 in more detail.

## 9.1 Basic Architecture

Because of the introduction of Semantic Web Services, the architecture shown in figure 14 has been restructured. In version 2 the GUI component has been moved out of the TAPAS network and rather added as part of a component called **MonitorApplication**. This component can in principle be located in any type of device as long as the device has Internet connectivity and runs a Java Virtual Machine. A set of applications is defined in a web server that is accessible to the user by executing Semantic Web Services. The TAPAS Main Server is deployed in the same device as the web server. The MainManager component can be located in any node inside the Tapas network, and will communicate with it's MiniManagers through TAPAS messaging. In the following subsections the different components will be described in more detail.

## 9.2 Server side of the system

The server side of our system consists of a main server in which a web server and the TAPAS Main Server is running. The web server, a Tomcat 5.5 installation, has got several different web applications which are accessible to the client through web services. Some of the web applications are semantic, that is,

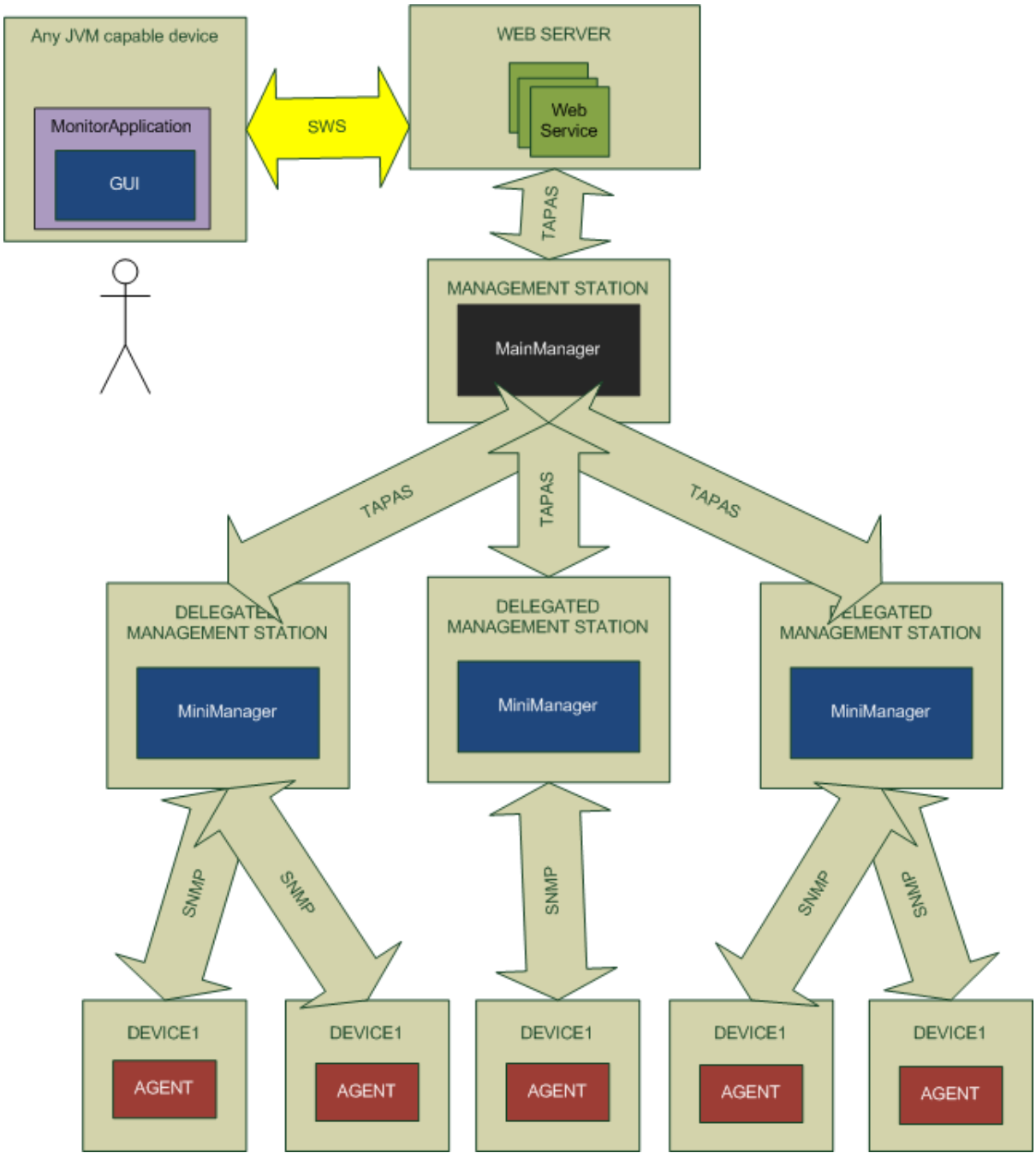


Figure 19: TAPAS SNMP-Based Monitoring Application version 2 - basic architecture

they include logic reasoning using an ontology language. This reasoning, and programmatic access to ontology descriptions is done using Protg-OWL API[12]. The other web applications are plain Java beans with no knowledge reasoning.

In the next subsections we will describe our various web applications in more detail.

### 9.2.1 CreatePlayApplication

Since the NMS system is designed as as a TAPAS Play. the play has to be "plugged in" in order to plug-in any roles. The play is bundled as a jar-file that comprises the MainManager and MiniManager roles as well as their support classes. The CreatePlayApplication will send a pluginPlay-message to the director actor residing in the TAPAS Main Server. The TAPAS Main Server may be any TAPAS node, but in our system it will always reside in the same node as the web server. The message contains the following parameters: <playname>,<playversion>,<playlocation>. The <playname> is the name of the play, that is, the name of our application. If there exists several versions of the same play, the <playversion> parameter specifies what version we want to plugin. The last parameter, <playlocation> specifies where the director can find the play. Below is an example of such a pluginPlay message:

$$pluginPlay(monitorPlay, 1.4, http://129.241.200.232/tapas/monitorPlay/mPlay.jar)$$

### 9.2.2 PluginMainManagerApplication

Before any monitoring can take place a MainManager must be plugged in. The PluginMainManagerApplication takes a string as input that specifies the node in which to plugin the MainManager. After invocation the application will send a pluginActor message to the director actor which contains the following parameters: <node>, <rolename>, <playname>. The <node> parameter corresponds to the node parameter provided in the input of the PluginMainManagerApplication, prefixed with 'tapas://', and specifies in which node to deploy the MainManager. The <rolename> specifies what TAPAS role to plugin, which in this case is "mainmanager". The last two parameters specifies the

[12]http://protege.stanford.edu/plugins/owl/api/

what play role should be plugged into. To plugin the MainManager on a node with IP address 129.241.200.226, we will have to send a pluginActor message looking like this:

$pluginActor(tapas://129.241.200.226, mainmanager, monitorPlay)$

### 9.2.3 PluginMiniManagerApplication

When the MainManager is plugged in, a monitor session may be initiated. As described earlier this will cause a DMS to be identified and a MiniManager to be plugged in. One of the drawbacks in version 1 was that the same type of MiniManager was deployed in the DMS regardless of the capabilities that DMS. In version 2, we have therefore developed a set of three different MiniManagers designed for different DMSs. The different MiniManagers do all have diffent analyzing mechanisms, some more powerful and hence more resource demanding than others. What type of MiniManager that will be chosen is a mentioned determined by the capabilities of the DMS. A TAPAS node's capabilities is revealed for other TAPAS nodes when the node connects to the TAPAS network. An actor called *capabilitymanager* residing in the TAPAS Main Server is keeping track of the capabilities of all TAPAS nodes. By sending the capabilitymanager a request, it is possible to receive these node-capabilities. The actual capability attributes registered involves screen resolution, main memory size, disk space and CPU-clock. Of course there are different kinds of capability attributes one may consider when selecting a MiniManager to deploy, but we have chosen to only consider the CPU-clock attribute as this gives a fairly good indication on a device's ability to execute a program.

**MiniManager ontology** The different MiniManager's are not registered by the capabilitymanager as these are only part of our application (the TAPAS play), and not the TAPAS Platform. We have therfore created an ontology over the domain of MiniManagers, called MiniManager.owl (Appendix B). The ontology language is OWL and is defined as follows:

```
<?xml version="1.0"?>


<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
```

```
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY protege "http://protege.stanford.edu/plugins/owl/protege#" >
    <!ENTITY xsp "http://www.owl-ontologies.com/2005/08/07/xsp.owl#" >
    <!ENTITY swrla "http://swrl.stanford.edu/ontologies/3.3/swrla.owl#" >
    <!ENTITY abox "http://swrl.stanford.edu/ontologies/built-ins/3.3/abox.owl#" >
    <!ENTITY tbox "http://swrl.stanford.edu/ontologies/built-ins/3.3/tbox.owl#" >
    <!ENTITY swrlx "http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl#" >
    <!ENTITY swrlm "http://swrl.stanford.edu/ontologies/built-ins/3.4/swrlm.owl#" >
    <!ENTITY sqwrl "http://sqwrl.stanford.edu/ontologies/built-ins/3.4/sqwrl.owl#" >
    <!ENTITY temporal "http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal.owl#" >
]>


<rdf:RDF xmlns="http://localhost:8080/Axis2WSTest/axis2-web/MiniManager.owl#"
     xml:base="http://localhost:8080/Axis2WSTest/axis2-web/MiniManager.owl"
     xmlns:sqwrl="http://sqwrl.stanford.edu/ontologies/built-ins/3.4/sqwrl.owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
     xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:temporal="http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal.owl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:swrlx="http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl#"
     xmlns:abox="http://swrl.stanford.edu/ontologies/built-ins/3.3/abox.owl#"
     xmlns:swrla="http://swrl.stanford.edu/ontologies/3.3/swrla.owl#"
     xmlns:tbox="http://swrl.stanford.edu/ontologies/built-ins/3.3/tbox.owl#"
     xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:swrlm="http://swrl.stanford.edu/ontologies/built-ins/3.4/swrlm.owl#">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/3.3/swrla.owl"/>
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl"/>
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/built-ins/3.3/tbox.owl"/>
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/built-ins/3.3/abox.owl"/>
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/built-ins/3.4/swrlm.owl"/>
        <owl:imports rdf:resource="http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal.owl"/>
        <owl:imports rdf:resource="http://sqwrl.stanford.edu/ontologies/built-ins/3.4/sqwrl.owl"/>
    </owl:Ontology>
    <swrl:Variable rdf:ID="m"/>
    <owl:DatatypeProperty rdf:ID="hasCapacity">
        <rdfs:domain rdf:resource="#MiniManager"/>
        <rdfs:range rdf:resource="&xsd;int"/>
    </owl:DatatypeProperty>
    <owl:Class rdf:ID="MiniManager"/>
    <MiniManager rdf:ID="MiniManager_1">
        <hasCapacity rdf:datatype="&xsd;int">1000</hasCapacity>
    </MiniManager>
    <MiniManager rdf:ID="MiniManager_2">
```

```
        <hasCapacity rdf:datatype="&xsd;int">2000</hasCapacity>
    </MiniManager>
    <MiniManager rdf:ID="MiniManager_3">
        <hasCapacity rdf:datatype="&xsd;int">3000</hasCapacity>
    </MiniManager>
</rdf:RDF>
```

As one can see, the ontology defines one class, namely the MiniManager class. It also defines one datatype-property[13], called *hasCapacity*. Three individuals of the MiniManager class are also defined, that have different values for the hasCapacity property. The individuals and their hasCapacity properties are shown in table 1.

| Individual name | hasCapacity property value |
|---|---|
| MiniManager_1 | 1000 |
| MiniManager_2 | 2000 |
| MiniManager_3 | 3000 |

Table 1: MiniManager individuals and their hasCapacity values

When the PluginMiniManagerApplication is executed, the user also provides the IP-address of node to be monitored as parameter. Using this IP-address our system then requests the node's capability attributes by sending a message to the capabilitymanager. When the CPU-clock attribute is extracted from the rest of the attributes, the application is ready to find the MiniManager best suitable for this node. This is done by comparing the MiniManagers hasCapacity values with the node's CPU-clock value.

**SWRL to find MiniManager** To compare the MiniManager's hasCapacity value with the node's CPU-clock value, we use Semantic Web Rule Language (SWRL). The reason we use this language instead of other rule languages is that this language is based on OWL and therefore enables us use it directly with our OWL description without any need for conversion. Protg-OWL API comes with well documented and easy-to-use SWRL support as well as support for the for the Pellet reasoner. With Protg-OWL API we are able to write SWRL rules programmatically, which enables us to create rules and queries on the fly. Our SWRL query for finding a MiniManager looks like follows:

[13] A datatype property links an individual to an XML Schema Datatype value or an RDF literal. This is in contrast to an object property which links an individual to an individual.

$MiniManager(?m) \wedge hasCapacity(?m, ?c) \wedge swrlb : lessThanOrEqual(?c, nodeCPUcapacity) \rightarrow sqwrl : select(?m, ?c)$

The query asks for all MiniManager individuals that have capacities equal or less than *nodeCPUcapacity,* which is the CPU-clock value of the node to be monitored. If more than one is returned, only the first will be selected. If none is returned, it means that the CPU-clock of the node is so limited that none of the defined MiniManagers would be able to execute without degrading the node's performance.

After a MiniManager is chosen, the web application sends a message to the already deployed MainManager and tells it to plug-in this MiniManager at the DMS. This will cause the MainManager to send a pluginActor message to the director of the form:

$pluginActor(tapas : //129.241.200.226, minimanagerNO1, monitorPlay)$

### 9.2.4 GetMibDefApplicaiton

When a MiniManager is deployed, the user is able to get monitor (SNMP) - values from the node. As stated in 8.1.2, one of the drawbacks in version 1 is that the user has to know the exact OID of the MIB object he wants to monitor. To relieve the user from having to know this, we have in version 2 allowed the user to enter more intuitive input values, and let the system convert this to the corresponding OID. For example, if the user wants to monitor on the MIB object ***tcpInSegs***, he can enter **tcpinputsegments**, and the system will recognize this string as an alias to **tcpInSegs**. To enable this, we have used Protg-OWL Editor to create an ontology over the domain of MIBObjects, called MIB.owl (appendix A) Since the total number of MIB objects is pretty large, we have only included six MIBObject classes in our ontology. In addition to the class ***MIBObject***, the ontology also defines the classes ***Alias***, ***MIBDefs***, and ***Syntax***, as well as the two object properties ***hasSyntax*** and ***isPartOfMibDef***. Furthermore, each of these classes have several subclasses, as one can see in figure 20.

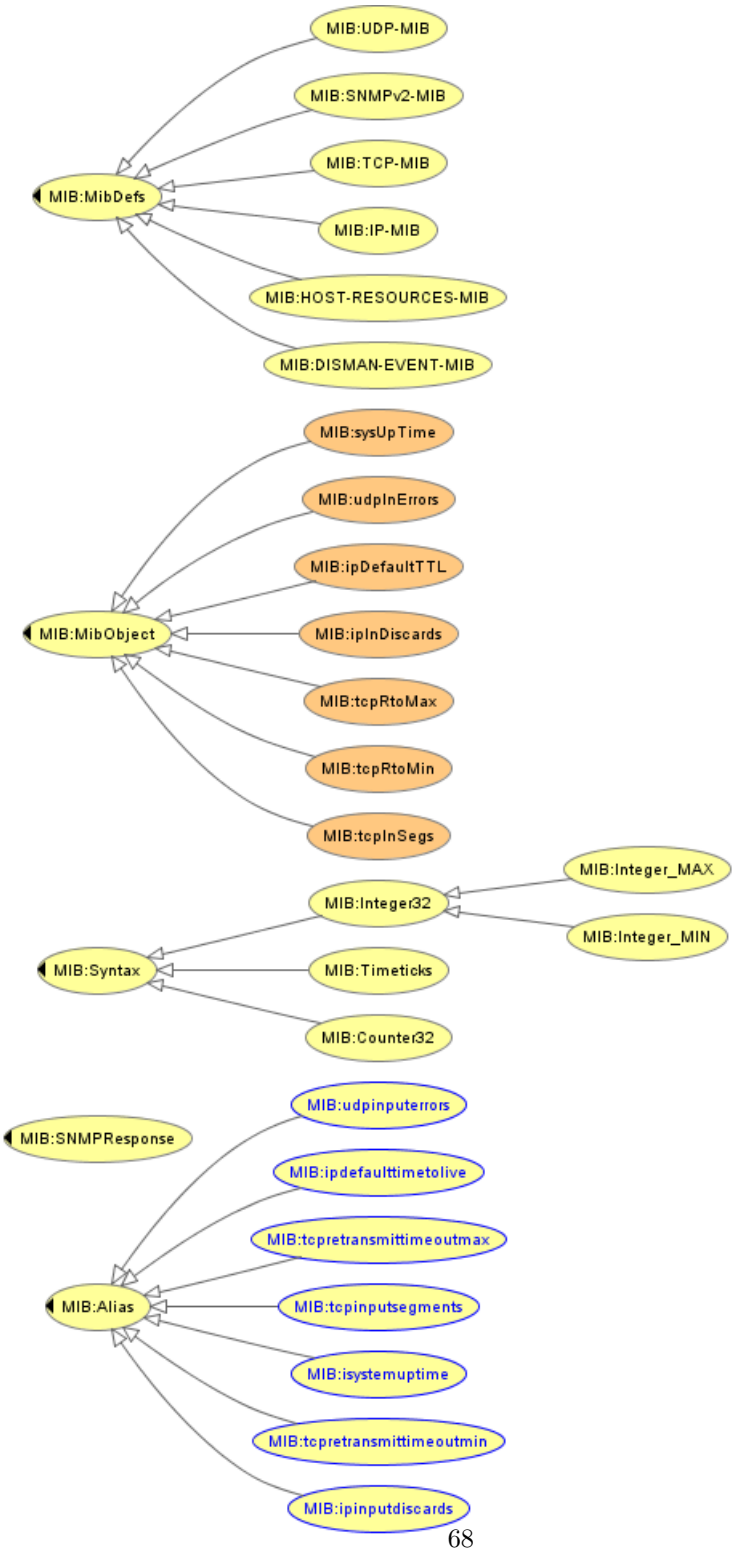




Figure 20: Asserted MIB ontology model

For each of the subclasses of the MibObject class, we have defined a couple of restrictions. Restrictions are used in OWL to restrict the members that belong to a class. For example, we would like to make the Alias subclass **systemuptime** a member of the MIBObject **sysUpTime**. To do that we must restrict all other members of the ontology, such that systemuptime is the only one "letting in". Using Protg, we have therefore created the following restrictions:

**sysUpTime**

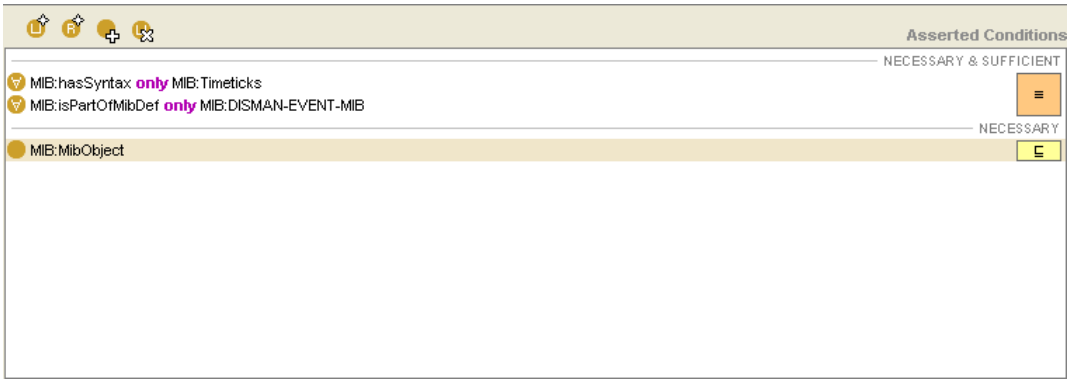


Figure 21: Restrictions of *sysUpTime*

**systemuptime**

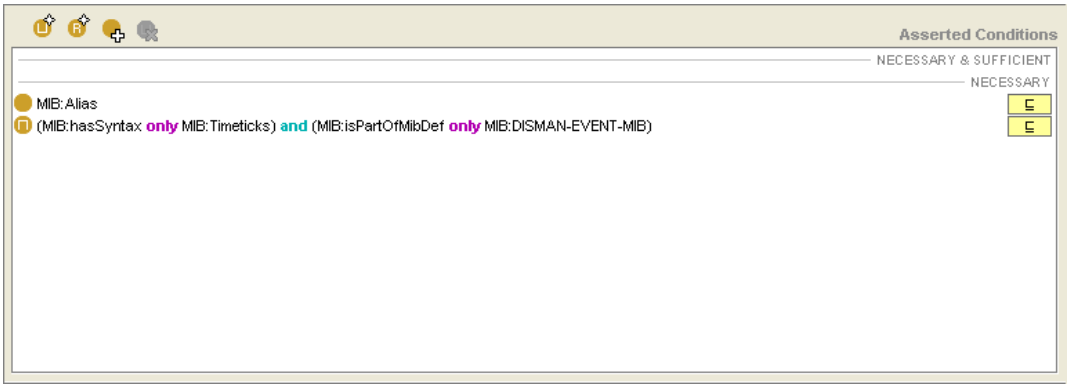


Figure 22: Restrictions of *systemuptime*

As one can see from the screen shots, both systemuptime and sysUpTime have restrictions on the values of the object properties. Since **syUpTime** has

defined the restrictions as NECESSARY AND SUFFICIENT, we are saying that not only are the defined conditions *necessary* for membership of the class **sysUpTime**, they are also *sufficient* to determine that any (random) individual that satisfies them must be a member of the class **sysUpTime.** This means that, since we have said that members of **systemuptime** must have **Syntax** equal to ***Timeticks*** and also be part of ***DISMAN-EVENT-MIB***, all members of **systemuptime** are also members of **sysUpTime.**

In the same way, we define restrictions of the other subclasses of the **MibObject** class and **Alias** class. When having a reasoner classifying our taxonomy, we end up with the following inferred ontology model:


Figure 23: Inferred MIB ontology model

As we see in figure 23, all subclasses of the **Alias** class have also been inferred as subclasses of their corresponding subclasses of the **MIBObject** class. That is, **tcpinputsegments** has been inferred as subclass of **tcpInSegs**, **ipdefaulttimetolive** as subclass of **ipDefaultTTL**, and so forth.

In our application we are using the Protg-OWL API to programmatically

access this inferred knowledge. Provided with a string representing an **Alias** as argument the application uses the inferred ontology model to locate the superclass of this **Alias**, and - if the **Alias** exists in the ontology - return the name of it's superclass. For example, executing the application with with the argument 'systemuptime', will return 'sysUpTime'. If the argument can not be cast to an existing **Alias**, the application will return an error message.

### 9.2.5 SNMPQueryApplication

This is the application that is responsible for returning SNMP values in an ongoing monitor session.

The application does not include any reasoning and will only forward the request to the MainManager by sending a TAPAS message. The MainManager will then contact the MiniManager responsible for monitoring on the provided IP-address and ask for the SNMP value for the given OID. The value is returned to the MainManager which in turn returns the value to the application.

### 9.2.6 PlugoutMiniManagerApplication

When the user wants to plugout a MiniManager from a node, the MonitorApplication will invoke this service. The corresponding web application will send a TAPAS message to the MainManager, forwarding the request. The MainManager will then send a plugoutActor request to the director asking it to plugout the MiniManager on the given node. An example of such a message looks like this:

$plugoutActor(tapas : //129.241.200.226)$

This will cause the MiniManager at node 'tapas://192.241.200.226' to be plugged out.

### 9.2.7 PlugoutMainManagerApplicaiton

When the user wants to plugout the MainManager, the PlugoutMainManagerService application will do as in the PlugoutMiniManagerService and forward the request to the MainManager. The latter then checks whether there are any active MiniManager's "out there". If it is, it first plugs them out, before pluggin out itself.

### 9.2.8 StopPlayApplication

Our last service, will send a plugoutPlay message to the director, requesting to plugout our play - analogous to ending the application. Below is an example of a plugoutPlay message:

$$plugoutPlay(monitorPlay, 1.4)$$

## 9.3 The Semantic Web Services

To make the web applications accessible for our client application a set of Web Services have been created. Using the Java2WSDL[14] tool by Axis, we created a Web Service out of every web application. Using the WSDL descriptions from the created Web Services we further created OWL-S descriptions for each of the services. The tool enabling this is called WSDL2OWL-S Converter, and is part of the Mindswap OWL-S API [15].

[14] http://ws.apache.org/axis/java/ant/axis-java2wsdl.html
[15] `http://www.mindswap.org/2004/owl-s/api/`

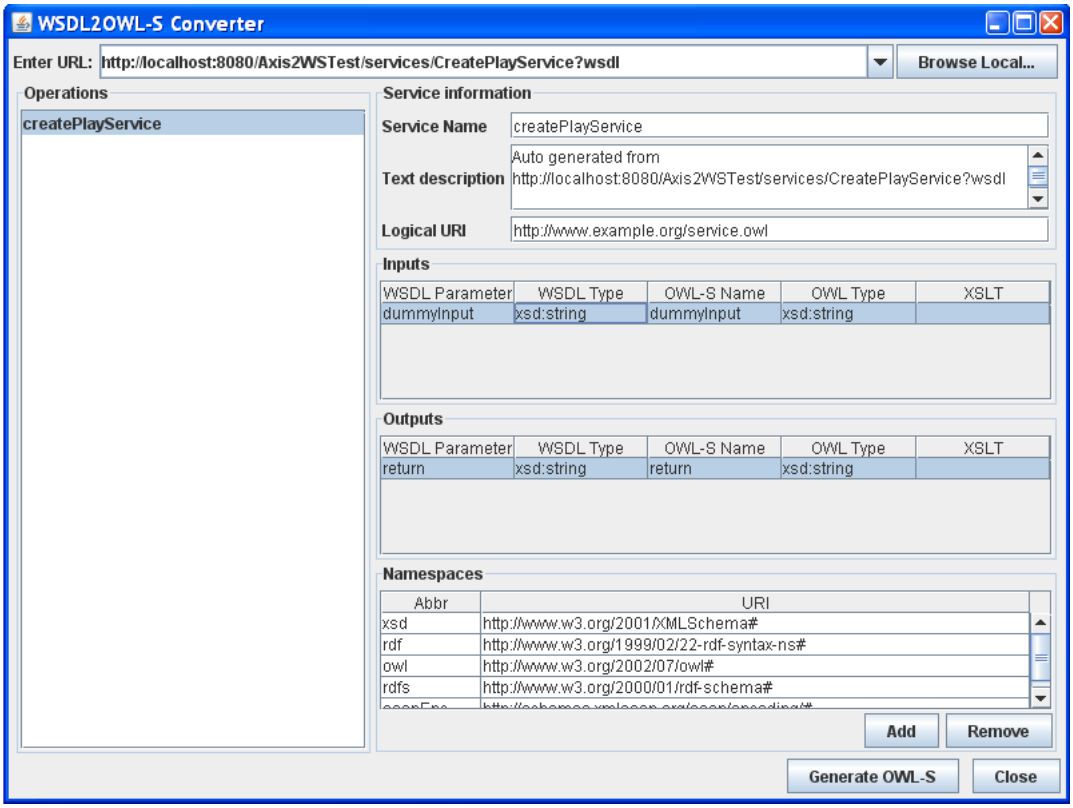


Figure 24: WSDL2OWL-S Converter

Before these services could be invoked we had to do some manual editing of the resulting OWL-S descriptions. This includes editing of the input and output types. Additionally, in cases where the output type of the service is an OWL type, XSL Transformations (XSLT) [23] have been applied as part of the service grounding to transform the output from an OWL type to an XML Schema datatype. This transformation is applied automatically in the execution of the service and is necessary in order for our MonitorApplication to interpret the return values appropriately.

Our resulting OWL-S service descriptions (Appendix C - K) contains one atomic process each, responsible for invoking their respective web applicions. In 2, we have listed our services as well as the input- and output types of their atomic processes.

To make the OWL-S service accessible from our client application they were

| Service | Input type | Output type |
|---|---|---|
| CreatePlayService | N/A | http://www.w3.org/2001/XMLSchema#String |
| PluginMainManagerService | http://www.w3.org/2001/XMLSchema#String | http://www.w3.org/2001/XMLSchema#String |
| PluginMiniManagerService | http://www.w3.org/2001/XMLSchema#String | http://www.w3.org/2001/XMLSchema#String |
| GetMibDefService | MIB.owl#Alias | MIB.owl#MibObject |
| SNMPQueryService | 1. MIB.owl#MibObject 2. http://www.w3.org/2001/XMLSchema#String | http://www.w3.org/2001/XMLSchema#String |
| PlugoutMiniManagerService | http://www.w3.org/2001/XMLSchema#String | http://www.w3.org/2001/XMLSchema#String |
| PlugoutMainManagerService | N/A | http://www.w3.org/2001/XMLSchema#String |
| StopPlayService | N/A | http://www.w3.org/2001/XMLSchema# |

Table 2: OWL-S Services and their input/output types

all deployed at the web server.

## 9.4 Client side of the application: MonitorApplication

Due to the introduction of Semantic Web Services, the client application is now not only a simple GUI, but rather enriched with a OWL-S facilities in order to invoke the OWL-S services described in the previous section. We have therefore created a new client application, called MonitorApplication. A simplified UML class diagram of this application is depicted below.

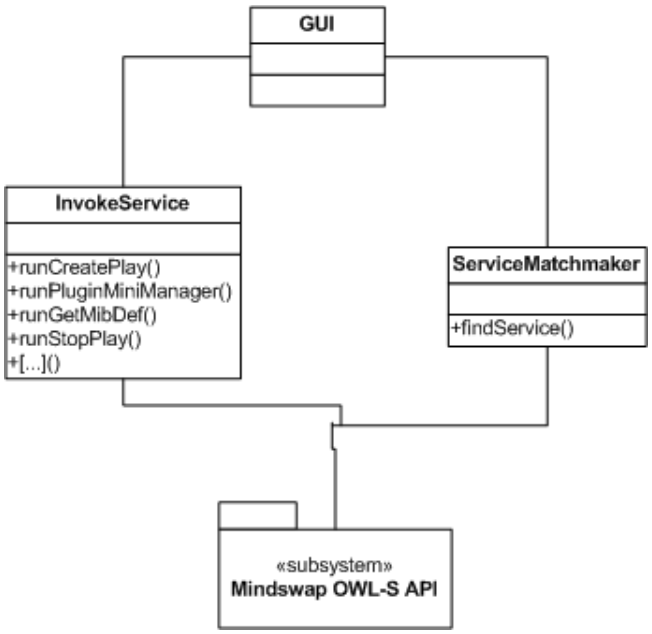


Figure 25: Simplified UML class diagram of MonitorApplication

In the following the different subcomponents of the MonitorApplication will be described.

### 9.4.1 MindSwap OWL-S API

After testing both the OWL-S API's mentioned in 5.4.1, we have found that none of them fulfills all the requirements of this application. The main benefits of the CMU OWL-S API is it's support for embedding OWL-S Profile descriptions as UDDI advertisements. In addition a matchmaker facility is available from their website which enables a client to locate a OWL-S service by searching for OWL typed inputs and/or outputs. However, the execution environment supported by the CMU OWL-S API, namely OWL-S VM, provided us with some problems that we were unable to solve. Nor could we find any proper documentation for this component enabling us to locate the problems.

Similar to the CMU OWL-S API, the Mindswap OWL-S API provides programmatic access to read and write OWL-S service descriptions. In addition a fully functioning execution engine is included in the API. This execution engine enables us to directly invoke our Semantic Web Services, without any need to

create stubs and skeletons. Jena [21] provides the OWL and RDF base for the API, and Pellet is used for reasoning. Contrary to the CMU OWL-S API, the Mindswap API is well documented and also have an active mailing list available from the Mindswap web site. This, in addition to a fully functioning execution environment, made us choose this API for our application.

### 9.4.2 InvokeService class

The execution environment of the Mindswap OWL-S API is reachable through the InvokeService class. This class contains methods for invoking all the defined OWL-S services. Below is the code snippet for invoking the CreatePlayService:

```
public String runCreatePlay() throws Exception{

        OWLKnowledgeBase kb = OWLFactory.createKB();
        kb.setReasoner("Pellet");
        service=kb.readService("http://localhost:8080/Axis2WSTest/axis2-web/CreatePlayService.owl");
        process = service.getProcess();

        values = new ValueMap();
        //Allthough this service does not need an input,
        //we have to include one for the service to execute
        values.setDataValue(process.getInput("someInput"), "dummyInput");

        //execute the service
        values = exec.execute(process, values);

        // get the result of the execution
        OWLDataValue out = values.getDataValue(process.getOutput());

        return out.toString();

        }
```

As one can see, by providing the URI of the OWL-S description file, we have programmatic access to the service. This enables us to set the input(s) of the service, execute the service and catch the result (output) of the execution. Most of the operations a user want to perform can be carried out by invoking one of the services listed in table 2, in the manner described above. However, in order to get a SNMP query value, one first has to execute the GetMibDefService (in order to get a **Mib.owl#MibObject** value), and then execute the SNMPQueryService to get the actual SNMP response value. Instead of having the user to make two separate calls this can be done automatically by creating a composed OWL-S service out the two services (GetMibDefService and SNMPQueryService)

Our new composed service will do the following:

1. Take a MIB.owl#Alias as input,
2. Invoke the GetMibDefService
3. Temporarily store the output (of type MIB.owl#MibObject) of GetMibDefService
4. Invoke the SNMPQueryService using the output from 3. as input
5. Return the final output to the user.

The full OWL-S description of this composed service can be seen in appendix K.

### 9.4.3 ServiceMatchMaker class

A drawback with the Mindswap OWL-S API is that it does not include facilities for creating UDDI advertisements of OWL-S Profile descriptions. In our application, however, this is not regarded as fatal drawback since the system will only offer a fairly limited amount of web services, enabling us to load all service descriptions in the computer's cache at start-up - without any significant performance impacts. A simple matchmaker component, realized in the ServiceMatchMaker class, is created for searching and locating the existing services. The search algorithm in the ServiceMatchMaker class requires an OWL typed input/output pair as parameter. If the algorithm locates a service with exactly this input/output pair, the URI of this service is returned. If not, the algorithm will try to create a list of services from the set of existing services where the input of the first service of the list is equal to the provided input, and the output of the last service of the list is equal to the provided output. The pseudo-code for this algorithm is depicted below.

```
findService(input_A, output_A, List)
        if List == null
                List = new List
        find service in global service list where input == input_A;
        if service found
                add this service to List
                if this service has output == output_A
                        return List
                else
                        output_B = this service's output
                        call findService(output_B, output_A, List)
        else if no service is found: return null
```

Figure 26: Pseudo-code for the findService method of the ServiceMatchMaker class

If a match is found, and the returned list contains more than one service, the MindSwap OWL-S API is used to create a composed OWL-S service of the services in the list. The URI of this composed service will then be returned.

Please note that this matchmaker is currently not used in our solution. Since we only have a limited amount of services in our system we do not have a need to "discover" them using the search algorithm. Instead the URIs of the services used are hard coded in our client application. However, the number of services may increase in the future, it may at some point not be clear what service to use. Also, in our solution we only have one service provider. Hypothetically, there may be several different service providers offering services in the domain of network management. For this reason, we have included the service discovery possibility for future use.

# Part III

# Evaluation

# 10 Conclusion

As a result of this work, we have extended our previous network management system with Semantic Web technology. A new MonitorApplication is created, that includes a graphical user interface that a system administrator can use to perform a set of management operations. These operations are carried out using Semantic Web Services that invokes a set of web applications implemented in a web server. These web applications in turn, communicates with the management system using the TAPAS Platform.

In the following subsections we will address each of our objectives of this thesis, and try to discuss whether we achieved our goals.

## 10.1 Analyze the potential benefit of using ontology and reasoning applications in our NMS system

After investigating our previous solution we located three aspects that we could improve using Semantic Web Services together with ontology- and reasoning applications. These were presented in section 8.1, and included:

**Problem 1: GUI and MainManager in same device** This is regarded as a problem since it places restrictions on the user on what type of device he is using to access and run the NMS application. A better approach would therefore be to separate the GUI from the system itself. The GUI could access the system by invoking (Semantic) Web Services.

**Problem 2: User must know OID by heart** In the version 1 of our application the user had to enter the exact OID for the MIB object he wanted to monitor. A better approach would be to let the user enter intuitive values like "systemuptime" or "mainmemorysize", and let the system interpret these input values and translate them to their corresponding OIDs. This can be accomplished by creating an ontology of MIB objects and corresponding aliases. A reasoning application can then be used to link an alias to a MIB Object in order to determine the OID to monitor.

**Problem 3: Choice of MiniManager** In the current solution the same type of MiniManager was deployed regardless of the capabilities of the DMS.

A better approach would be to create different types of MiniManagers with different capabilities, each designed for a special set of devices. In this way one can - before a MiniManager is deployed - perform a matching between the MiniManager's capabilities and the DMS's capabilities in order to locate the MiniManager that is best suited for the device to be monitored. This can be accomplished using an ontology over MiniManagers and their capabilities. An reasoning application can then be used to perform the matching.

## 10.2 Specify proposed ontology and reasoning applications integrated with the NMS application. Ontology shall be specified using Protege-OWL Editor.

The first problem was addressed creating a new application called MonitorApplication with an embedded GUI. This MonitorApplication serves as our service requester and can be executed independently of the NMS system and only requires it's host to be JVM compatible and with Internet connectivity. The MonitorApplication communicates with the NMS system using Semantic Web Services.

In order to address the second problem , we first used the Protg-OWL Editor to create a domain ontology, called MIB.owl. This ontology defines the OWL classes Alias, MibObject, MibDefinition and Syntax. Furthermore, we created a web application called GetMibDefApplication that accesses this ontology and it's defined classes and properties programmatically. Using the Pellet reasoner to access the inferred ontology model, we are able to locate what MibObject corresponds to the given Alias. That is, when the user enters the argument "systemuptime", our application is able to infer that this alias corresponds to the MibObject sysUpTime.

The third problem was solved by creating three different MiniManager TAPAS Roles, each with different analyzing mechanisms, designed for different types of devices. In addition we created an ontology called MiniManager.owl where we defined the class MiniManager as well has the property hasCapacity. The hasCapacity property ranges from the domain of MiniManager to an integer specifying a CPU-clock capability. Three different individuals of the class MiniManager were created, with different values for the hasCapacity property. Each of these individuals corresponds to one of the MiniManager TAPAS Roles. The

web application PluginMiniManagerApplication, accesses this ontology to find what MiniManager individual - and thus what MiniManager Role - to plug in at a specific node. In this way we can be certain that no MiniManager will require more resources than the node it is deployed to can offer.

We also created some other web applications that do no perform any reasoning, but are necessary in order to perform the other operations required for the network management system to operate properly. This includes applications to plugin / plugout the the TAPAS Play, to plugin /plugout the MainManager as well as performing SNMP queries.

### 10.3 Specify and implement web-service based applications that makes the reasoning applications from 2 available as Web Services

For every web application we created an OWL-S service description. These OWL-S services are invoked from our client application called MonitorApplication through the MindSwap OWL-S API. Most of the operations the user can perform are carried out by invoking one of the defined services through the InvokeService class. The operation of performing a specific SNMP query, however, required two of the services to be executed, namely the GetMibDefService and SNMPQueryService. Instead of requiring two separate calls from the user, we created a composed service out of the two which could be invoked once.

## 11 Evaluation and Future Work

The result of our work is a light-weight management system applied with SW-technology. We have proven that the use of formal ontologies to provide our system with knowledge about the management system enables the system to make “intelligent” and automatic decisions. Since we have created an ontology over MIB objects and corresponding aliases we allow a system administrator to input intuitive arguments to the application, and thereafter make the system translate these arguments to an actual MIB Object. In the current ontology, we have only created one alias per MIB Object, but since these are described in a OWL file - and not in code - new aliases could be added to the ontology at any time, without having to rewrite any code. In the same way new MIB Objects

could be added to the ontology enabling more MIB objects to be monitored.

By creating an ontology over MiniManagers and their capabilities, as well as taking advantage of the facilities provided by the TAPAS Platform to request a TAPAS node's capability attributes - we can compare the MiniManagers capabilities against the nodes capability in order to find the MiniManager that is best suited for the selected DMS. In this way we have created an effective way of recognizing the difference in resource availability of different network devices, and thus enforcing an flexible and efficient use of their resources. Nodes with a high CPU-power, will be granted a MiniManager capable of having a large number of concurrent monitor sessions as well as powerful analyzing mechanisms. In other nodes with less resource capability when comes to CPU-power, a simpler version of the MiniManager will be chosen.

As the network infrastructure changes and devices are removed or replaced, more powerful devices may become available. These changes can easily be accommodated for by creating new MiniManager Roles and creating new MiniManager individuals in the ontology. This can easily be accomplished without bringing the system down.

## 11.1 Proposals for future work

All though the system uses ontology and reasoning applications in order to perform "intelligent" decisions, there are several areas that may be further improved. First, our method for finding the "best" MiniManager to deploy at a node is only based on a comparison of CPU-power. In a future version, this method could be further improved by comparing other capability attributes as well, for example main memory usage. Also as, we mentioned in 9.4.3, our MonitorApplication contains an unused class, namely the ServiceMatchMaker class. As discussed there, we do only consider one service provider in our application, which consists in the same organization as our service requester. For this reason the idea of discovering the services to use have no real purpose in our system, as we can hard code the service URIs in the client application. Referring to section 4.2, this does not exactly harmonize with the idea of Semantic Web Services. Hypothetically, there may be other organizations with a better service for locating a MIB Object given an alias as argument. A future version of our MonitorApplication could accommodate for this by including other service providers in the search and discovery of an "Alias2MIBObject-converter".

In order to realize this discovery, our discovery-algorithm probably needs to be rewritten as one in this case cannot compare the service's inputs and outputs, but rather non-functional properties like the number of defined Aliases the service can convert. In this way the discovery algorithm can locate the service with the highest number of defined Aliases before the MonitorApplication invokes the service. If it turns out that the chosen service cannot convert the provided argument, the MonitorApplication will choose the service with the second most Aliases defined, and so forth.

# 12 Related Work

All though we have failed to find any work that relates directly to our work, there are several projects that relates to our work in some way or another. In the following subsections we will describe some of them.

## 12.1 Semantic Management Meta-Model

The article *Semantic Management: Advantages of using an ontology based management information meta-model [25]* is discussing a way of realizing interoperability between different management domains described in different information models. Interoperability between the different information models (SNMP, CMIP, DMI, WBEM...) have usually been carried out with syntactical translations that do not include the semantic aspects of the defined information. This article shows a way to define a management information meta-model that integrates all the information, that currently belongs to different management domains, in the same model. This is achieved using formal ontology techniques. Having one one such model, network managers can work and reason with an abstract view of the management information, independent of the specific management model used.

## 12.2 The use of Web Services in a Network Management System

The idea of decentralizing network management systems has been around for quite a while, and there have been several proposals of how to achieve this. As with our management system, several of these proposals adopt the management

by delegation (MbD) model [18], which enables one to delegate management functions along the management system, to decentralize the management operations. The work carried out in [26] demonstrates a way of using Web Services in order to delegate these management functions. They base their work on studies showing that Web Services can consume less bandwith than SNMP when a large number of management objects needs to be retrieved from a management entity. The paper specifies a prototype of a WS-based MbD system developed to allow the observations of WS against SNMP.

Continuing with approaches that uses Web service technology to implement management interfaces of managed resources, the paper [27] shows a way to combine several web services to perform composite processes. The composite processes are defined using OWL-S, which allows their formal description. In contrast to our work, where we use a execution engine present in the MindSwap OWL-S API, this paper chooses the Web Services Business Process Execution Language (WSBPEL), in order to execute the services. This requires a mechanism to translate the OWL-S composite processes to WSBPEL before the execution can take place.

# References


[1] Tom Gruber, "Ontology" 2008: http://tomgruber.org/writing/ontology-definition-2007.htm

[2] Princeton University Cognitive Science Laboratory, WordNet, http://wordnet.princeton.edu/

[3] OWL Web Ontology Language Reference, W3C, http://www.w3.org/TR/owl-ref/

[4] OWL DL Class Axioms, W3C, http://www.w3.org/TR/owl-semantics/syntax.html#2.3.2.1

[5] Horn Rule Semantics, W3C, http://www.w3.org/2005/rules/wg/wiki/Horn_Rules_Semantics

[6] SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C, http://www.w3.org/Submission/SWRL/

[7] Logical Foundations of Object-Oriented and Frame-Based Languages, Kifer, Lausen, Wu, http://www.cs.umbc.edu/771/papers/flogic.pdf

[8] Semantic Web Services. IEEE Intelligent Systems, Special Issue on the Semantic Web, S. McIlraith, T.Son and H. Zeng. 16(2):46–53, March/April, 2001.

[9] W3C Web services Choreography Working Group: Charter: http://www.w3.org/2003/01/wscwg-charter

[10] Web Services Business Process Execution Language Version 2.0, OASIS 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[11] The Web Service Modeling Framework WSMF, D.Fensel, C.Bussler, http://www.wsmo.org/papers/publications/wsmf.paper.pdf

[12] OWL-S: Semantic Markup for Web Services, W3C Member Submission 2004, http://www.w3.org/Submission/OWL-S/

[13] OWL-S Virtual Machine, SemWebCentral, http://www.semwebcentral.org/projects/owl-s-vm/

[14] Mindswap OWL-S API, Maryland Information and Network Dynamics Lab Semantic Web Agents Project ,http://www.mindswap.org/2004/owl-s/api/

[15] OWL-S: Some Motivating Tasks: http://www.w3.org/Submission/OWL-S/#2

[16] SPARQL Query Language for RDF, W3C Recommendation 15. January 2009, http://www.w3.org/TR/rdf-sparql-query/

[17] SNMP-based Monitoring Application By Using TAPAS Platform, O. Nistad, F. A. Aagesen, 2008, Department of Telematics, Norwegian University of Science and Technology

[18] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In International Symposium on Integrated Network Management, pages 95–107, 1991.

[19] TAPAS Website: http://tapas.item.ntnu.no/wiki/index.php/Main_Page

[20] Apache Axis, Java platform for creating and deploying web services applications, http://ws.apache.org/axis/

[21] Jena - A Semantic Web Framework for Java, http://jena.sourceforge.net/

[22] Pellet - The Open Source OWL DL Reasoner, http://clarkparsia.com/pellet/

[23] XSL Transformations (XSLT), http://www.w3.org/TR/xslt

[24] D Krafzig, K. Banke, D. Slama, Enterprise SOA: Service Oriented Architecture Best Practices. Prentice Hall PTR, 2004

[25] Jlopez, Villagra, Berrocal, Semantic Management: Advantages of using an ontology based management information meta-model

[26] T. Fioreze, L. Z. Granville, M. J. Almeida, L. R. Taruco, Comparing Web Services with SNMP in a Management by Delegation Environment

[27] J. Fuentes, J.Lopez, P.Castells, An Ontology-Based Approach to the Description and Execution of Composite Network Management process for Network Monitoring

[28] W3C; RDF: http://www.w3.org/RDF/

[29] W3C; RDF-S: http://www.w3.org/TR/rdf-schema/

[30] W3C; Web Ontology Language: OWL; http://www.w3.org/2004/OWL/

[31] S. Jiang, F.A Aagesen; An Approach to Integrated Semantic Service Discovery