ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE DI CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

SUPPORTING SEMANTIC WEB TECHNOLOGIES IN THE PERVASIVE SERVICE ECOSYSTEMS MIDDLEWARE

Subject
COMPUTATIONAL LANGUAGES AND MODELS LM

Supervisor
Prof. MIRKO VIROLI

Student
Dr. Eng. PAOLO CONTESSI

Co-Supervisor
Eng. Dr. DANILO PIANINI

Session I
Academic Year 2011/2012

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE DI CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

SUPPORTING SEMANTIC WEB TECHNOLOGIES IN THE PERVASIVE SERVICE ECOSYSTEMS MIDDLEWARE

Subject
COMPUTATIONAL LANGUAGES AND MODELS LM

Supervisor
Prof. MIRKO VIROLI

Student
Dr. Eng. PAOLO CONTESSI

Co-Supervisor
Eng. Dr. DANILO PIANINI

Session I
Academic Year 2011/2012

ABSTRACT

Semantic Web technologies are strategic in order to fulfill the openness requirement of Self-Aware Pervasive Service Ecosystems. In fact they provide agents with the ability to cope with distributed data, using RDF to represent information, ontologies to describe relations between concepts from any domain (e.g. equivalence, specialization/extension, and so on) and reasoners to extract implicit knowledge.

The aim of this thesis is to study these technologies and design an extension of a pervasive service ecosystems middleware capable of exploiting semantic power, and deepening performance implications.

SOMMARIO

La *openness* è un requisito fondamentale nell'ambito dei *Self-Aware Pervasive Service Ecosystems*: i componenti (agenti) di un sistema *open* come questo, il loro comportamento ed i protocolli impiegati non sono staticamente determinati, ma variano continuamente durante l'esecuzione. In tale scenario è fondamentale che ogni parte del sistema sia in grado di gestire informazioni provenienti da diverse fonti ed interpretarle correttamente, anche nel caso in cui non sia possibile utilizzare la stessa rappresentazione. Le tecnologie del *Semantic Web* sono quindi strategiche, perché sono nate per gestire situazioni di questo genere: RDF permette la descrizione delle informazioni, OWL è in grado di modellare le relazioni fra concetti (equivalenza, specializzazione/estensione, ...) e i servizi di *reasoning* sono in grado di far emergere informazioni celate nei dati.

Lo scopo di questa tesi è esplorare queste tecnologie e valutarne l'inclusione in un *pervasive ecosystems* middleware, ponendo attenzione alle implicazioni in termini di prestazioni.

ACKNOWLEDGEMENTS

I would like to thank all the people that contributed to the realization of this master thesis. First of all my supervisor, Prof. Mirko Viroli, who gave me the possibility to work on this project and provided me his experience in the field.

I am also very grateful to Eng. Dott. Danilo Pianini, my co-supervisor, for having shared his knowledge with me whenever I needed, and for having corrected the first version of this volume; thanks to him someone else, other than me, will be able to understand what I have achieved in these months.

Moreover I want to thank my family, my girlfriend and my friends for supporting me in this period: I owe you my graduation.

As a final note I have to acknowledge the debt I owe to the SAPERE project members [Viroli *et al.*, 2012; Zambonelli *et al.*, 2011] for providing documentation and [Miede, 2011; Pantieri and Gordini, 2011] for this volume's style and layout.

I am sure I have forgotten to list something or someone (maybe I have also forgotten english grammar), I am sorry. Thank you all.

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

SAPERE	Self-Aware Pervasive Service Ecosystems
LSA	Live Semantic Annotation
RDF	Resource Description Framework
RDFSchema	Resource Description Framework Schema
N3	Notation 3
Turtle	Terse RDF Triple Language
OWL	Web Ontology Language
OWL-DL	OWL-Description Logic
OWL-EL	OWL-E Logic
OWL-QL	OWL-Q Logic
SPARQL	Simple Protocol and RDF Query Language
SPARUL	SPARQL/Update
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URIref	URIreference
XML	eXtensible Markup Language
XSD	XML Schema Definition
W3C	World Wide Web Consortium
WWW	World Wide Web
OSGi	Open Service Gateway initiative
API	Application Programming Interface
SPI	System Programming Interface
OOP	Object-Oriented Paradigm
CTMC	Continuous-time Markov Chains

1 INTRODUCTION

Nowadays information is everywhere: Internet and Web 2.0 paved the road to remote interaction and knowledge sharing between people and services. When a human being writes or says something to someone each word has a meaning, that is implicitly known or specified through a common vocabulary, while a dialog between two software agents is merely based on *syntax*, so on how each phrase is spelled. In fact *semantic*, which is a synonym of language meaning, is generally provided by the developer at compile time and if concepts and their encoding are not pre-shared communication is compromised [Hebler *et al.*, 2009].

This is the reason why World Wide Web Consortium (W3C) started defining a set of standards, meant to attach a significant to resources on the web, so giving birth to the *Semantic Web*. Using these frameworks means publishing decorated data and enacting each piece of software to understand them wherever they come from, just merging those decorations with common, or application-specific, vocabularies and exploiting inferential capabilities of available *reasoning* services. In other words strong agents acquire the ability to cope with distributed data.

Another wide-spread phenomena of the last years is the appearance of ever more powerful mobile systems and sensors. Computer Science is defining new paradigms, mainly focused on concepts like context-awareness, self-organization and distribution; *pervasive services* is one of them. Applications are designed to exploit connectivity, build a network for data exchange and elaboration, supporting people in everyday life: trying to guess what the user needs, possibly without asking or specifying how the task should be executed.

Once again information understanding and management is vital: how can Semantic Web enhance these kind of systems? In terms of *openness* [Zambonelli *et al.*, 2011]. In pervasive computing boundaries are not defined: components should always know (1) where they are, (2) what are they with and (3) what resources are nearby and available [Wikipedia, 2012; Zambonelli *et al.*, 2011], because next time something could change and application logic should adapt, avoid failure and reach its goal. In this context, for example, recent and older devices would occasionally interact and should be able to understand each other, even if they do not have common data representation. Thinking about developing services that are aware of every past and future protocol they will meet isn't feasible (at least for the future part). On the other side, defining a vocabulary that explains how information is

structured, will allow future applications to deal with them, just defining concepts mapping.

In this thesis Semantic Web technologies will be presented and discussed according to the idea of producing a *pervasive service ecosystems middleware*, capable of providing required abstractions built on semantic notions. In particular the middleware will be designed according to the model defined in the Self-Aware Pervasive Service Ecosystems (SAPERE) project [Viroli *et al.*, 2012; Zambonelli *et al.*, 2011], which is meant to address requirements such as dynamism, mobility, context-awareness and self-organization, through a set of nodes where agents live, manifest themselves and share information by means of LSA-spaces and bio-chemical mechanisms.

Apache Jena and Pellet has been chosen for LSA-space realization: the former provides an efficient and stable RDFStore implementation, letting data be memorized, while the latter offers reasoning capabilities over stored data. The rest of the middleware will supply basic facilities, enabling *(1)* the specification of agents, meant to publish and observe information about their business in the space, *(2)*, the definition of a static topology of computational nodes, each one with a local LSA-space and one or more SAPERE-agents, and *(3)* bio-chemical resembling rules – namely eco-laws – scheduling, through a dedicated entity called *ReactionManager*. Thanks to it, information exchange between SAPERE-nodes is enabled according to the diffusion mechanism, via a *NetworkManager* which will physically handle data relocation.

The resulting platform has been tested, in order to verify requirements satisfaction and analyze performance. A couple of examples have been created in order to demonstrate middleware usage and potential: one focuses data description and interpretation and the other implements a typical scenario in which data are spread in the environment and used by other services to provide useful information to the user. Run benchmarks show that the developed system scales better when the reasoner is not enabled – as expected – but the overall performance is encouraging, considering that all the LSA-space operations have to be serialized, due to Pellet lack of multi-threading support.

The whole architecture has been designed in order to ease future extensions: component-oriented paradigm has been used to obtain modularization and separation of concerns has been stressed in order to simplify the implementation and integration of new behaviors when needed. Possible improvements may concern: (1) eco-laws scheduling policies, (2) network protocol support and (3) performance tuning.

The work is structured as follows:

CHAPTER 2 is dedicated to a survey on Semantic Web Technologies

CHAPTER 3 reports relevant concepts of the SAPERE model for semantic-enabled middleware designing (principally data formalization).

CHAPTER 4 presents problem analysis and solution design according to previous chapters

CHAPTER 5 deepens performance implications of the proposed platform

Finally, conclusions are discussed on Chapter 6.

2 SEMANTIC WEB SPECIFICATIONS AND TECHNOLOGIES

CONTENTS

This chapter is meant to introduce Semantic Web Technologies, starting with specifications from W3C, which represent a global standard in this field, and completing with a survey on some of the most famous Java-based technologies implementing these standards. At the end of this section the reader will have a basic idea of how to realize a Java program which handles Semantic Web Data.

2.1 W3C SPECIFICATIONS

2.1.1 RDF

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web.

According to W3C specification *a Resource is everything that can be accessed via a URIreference* [W3C, 2004], such as pages, things, people and so on.

Each resource can be described in terms of a set of multi-valued properties, whose value can be:

- another **Resource** (obviously identified by an URIreference (URIref))
- a **Literal** , which is *a double-apices-delimited String representing a concrete value* (such as a number or a message and so on) but not a resource

Consequently an RDF Statement is represented as a set of RDF Triples, composed by:

- A **subject** identified by an URIref
- A **predicate** (or property) identified by an URIref
- An **object** which is the value of the property, so a Literal or an URIref to a Resource

Given its structure, as previously stated, an RDF model can be represented as a Graph, in which arcs are properties and nodes are Resources or Literal, providing a graphical format. Generally this representation is only used for little models and surely not for exchange purposes for which other textual alternatives are: RDF/XML, Turtle, N3 and N-Triples.

RDF/XML : the normative format for RDF models exchange (machine-processable)

Terse RDF Triple Language (Turtle) : compact and human friendly format, defined as a subset of the more expressive Notation 3 (N3)

N-TRIPLES : easy to parse format useful for streaming serialization

Details about these formats have been reported in the following. In order to provide comparison terms between each representation here is provided an informal description of an RDF model to be stated:

```
There is a resource (a webpage) "index.html"
created on August, 16 1999 by a developer named Bob Smith.
This page is written in english.
```

ABOUT URIREFERENCES AND LITERALS Let's now go deeply in the definition of these two essential elements.

URIreferences

RDF uses URIref to name subjects, predicates, and objects in RDF statements. A URIref is a Uniform Resource Identifier (URI), together with an optional fragment identifier at the end.

For example, the URIref `http://www.example.org/index.html#Section2` consists of the URI `http://www.example.org/index.html` and (separated by the "`#`" character) the fragment identifier `Section2`. RDF URIrefs can contain *Unicode characters*, allowing many languages to be reflected in URIrefs.

In order to obtain a human-readable identifiers, a namespace can be declared: this way the URI part of an URIref can be mapped in a prefix (also called `QName`, from eXtensible Markup Language (XML)), obtaining something like `<prefix>:<fragment-id>`.

In order to obtain World Wide Web (WWW) compatibility the *number sign* ("`#`"), used as URIref's components separator, can be substituted by a slash ("`/`"); in particular the latter should be used to define a taxonomy of concepts (when the vocabulary is very large) split in several

files, while the former should be adopted when referring to a simple entity[1] [Hebler *et al.*, 2009].

Literals

Literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals.

A literal may be the object of an RDF statement, but neither the subject nor the predicate.

Literals can be plain or typed :

- A *plain literal* is a string combined with an optional language tag. This may be used for plain text in a natural language. As recommended in the RDF formal semantics, these plain literals are self-denoting, meaning that they should be understandable as they are.
- A *typed literal* is a string combined with a datatype URI. It denotes the member of the identified datatype's value space obtained by applying the lexical-to-value mapping to the literal string.

An example of a plain literal with language tag is `"car"@en` (only "car" if language tag is omitted), while a typed literal is something like `"35"^^<http://www.w3.org/2001/XMLSchema#integer>` (or `xsd:integer` if `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` has been declared).

RDF/XML format

As stated by its name, RDF/XML exploits the XML language to represent RDF statements.

This choice make the format really handy for a machine and that's why it has been chosen as normative language for RDF exchange. Despite this, a document of this type cannot be correctly interpreted if it is partial, because some information could be missing: for example if some tags are not closed a parser will fail. In conclusion this format doesn't fit streaming serialization cases, as a counterpart N-Triples format fits better.

RDF/XML syntax is reported in the following (with reference to the presented example).

Listing 1: A RDF/XML Snippet

```
1 <?xml version="1.0"?>
2 <!-- Namespaces declaration -->
3 <rdf:RDF
4     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5     xmlns:dc="http://purl.org/dc/elements/1.1/"
6     xmlns:exterms="http://www.example.org/terms/"
7     xmlns:foaf="http://xmlns.com/foaf/0.1/">
```

1 This convention is useful in order to be compatible with browsers

```
8
9   <!-- Description of a resource -->
10  <rdf:Description
11      rdf:about="http://www.example.org/index.html">
12    <!-- Example of datatype tag -->
13    <exterms:creation-date
14        rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
15      1999-08-16
16    </exterms:creation-date>
17  </rdf:Description>
18
19  <rdf:Description
20      rdf:about="http://www.example.org/index.html">
21    <!-- Literal property -->
22    <dc:language>en</dc:language>
23  </rdf:Description>
24
25  <rdf:Description
26      rdf:about="http://www.example.org/index.html">
27    <!-- Property which points to a resources -->
28    <dc:creator
29      rdf:resource="http://www.example.org/staffid/85740"/>
30  </rdf:Description>
31
32  <rdf:Description
33      rdf:about="http://www.example.org/staffid/85740">
34    <!-- Example of language tag -->
35    <foaf:name xml:lang="en">Bob Smith</foaf:name >
36  </rdf:Description>
37 </rdf:RDF>
```

Looking at listing 1 we can infer some information:

- The root of an RDF/XML is the `<rdf:RDF>` tag, with *namespaces declarations*, which are used to import external vocabularies. A **vocabulary** is an RDF model, expressed in Resource Description Framework Schema (RDFSchema), which defines a set of classes and properties.
- Inside the root some `<rdf:Description>` tags are listed (order doesn't matter): each description is about a resource, specified in the `rdf:about` attribute via its URIref. A description is defined by a set of properties, contained in it
  - Each property is defined by a tag named as the property itself
  - If the property should point to a resource then its URIref should be specified in the rdf:resource attribute
  - If the property expects a literal then the value is written inside property's tags.

- Given the fact that *literals* are String representations of a value, it *is possible to specify the language tag and/or the datatype tag*: the former specifies the language in which the String is written, the latter specifies the original value datatype.

Reducing Verbosity: The typed node abbreviation

This description is really verbose and as the model grows even the number of tags grows too. A first method to reduce the dimension of produced documents is defining the properties, which are relative to the same resource in the same description.

Moreover if resources to be described have a `rdf:type` statement (see RDFSchema specification, on Section 2.1.2), then it is possible to replace the `<rdf:Description>` tag with a specific tag reporting the name of the relative `rdf:Class`. This method is similar to the one used to define properties.

The result is reported in the following (see Listing 2):

Listing 2: A more compact RDF/XML description

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:exterms="http://www.example.org/terms/"
    xmlns:foaf="http://xmlns.com/foaf/0.1/">

  <exterms:WebPage
      rdf:ID="http://www.example.org/index.html">
    <exterms:creation-date
        rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
      1999-08-16
    </exterms:creation-date>
    <dc:language>en</dc:language>
    <dc:creator
      rdf:resource="http://www.example.org/staffid/85740"/>
  </exterms:WebPage>

  <exterms:Developer rdf:ID="http://www.example.org/staffid/85740">
    <foaf:name xml:lang="en">Bob Smith</foaf:name >
  </exterms:Developer>
</rdf:RDF>
```

Obviously `exterms:WebPage` and `exterms:Developer` should be two classes defined somewhere referred by the `exterms` namespace declaration.

Turtle format

This is the most human-readable format, in fact Turtle provides a compact representation of each resource with all its properties reported below it. Despite its structural similarities with RDF/XML (see last RDF/XML example, on Listing 2) the absence of tags, which has been

substituted with indentation, punctuation and brackets, let the focus be maintained on information.

The shared example, formatted in Turtle, looks like in the following (N.B.: sharp-starting lines are comments).

Listing 3: The example expressed in Turtle

```
# Namespaces declarations
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://www.example.org/> .
@prefix exterms: <http://www.example.org/terms/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

# Each resource is followed by an
# indented list of properties, separated by ";"
# and terminated by "." (as each statement)
ex:index.html
  # Example of datatype tag
  exterms:creation-date "1999-08-16"^^<http://www.w3.org/2001/
      XMLSchema#date> ;
  # If a Property has more than one value than value list is comma-
      separated
  dc:language "en" ;
  # URIref are enclosed in angle brackets
  dc:creator <http://www.example.org/staffid/85740> .

# Example of language tag
<http://www.example.org/staffid/85740> foaf:name "Bob Smith"@en .
```

Obviously, in order to declare the type of a class, the `rdf:type` property can be specified. Generally, in this case, type declaration is stated as first and written in the same line of the resource's URIref. A typical shorthand adopted in Turtle is using "a" in place of "`rdf:type`", as shown in the following listing:

Listing 4: A shortcut for type definition in Turtle

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.example.org/> .
@prefix exterms: <http://www.example.org/terms/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

# In alternative rdf:type property can be used
<http://www.example.org/staffid/85740> a exterms:Developer ;
  foaf:name "Bob Smith"@en .
```

Turtle is defined as a subset of N3: in order to empower readability part of its expressive power has been reduced.

N-Triple format

This is the simplest format. It consist in listing all RDF triples that compose the model. This feature make it convenient when a model

needs to be streamed over the network, in fact each line represent a complete statement.

As usual, the example, in N-Triples format, is reported in the following; for the sake of brevity only information about the developer has been listed.

Listing 5: A N-Triples example

```
<http://www.example.org/staffid/85740>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.example.org/terms/Developer>
<http://www.example.org/staffid/85740>
  <http://xmlns.com/foaf/0.1/name>
    "Bob Smith"@en
```

In order to distinguish an URIref from a Literal the former is enclosed in angle brackets (`<...>`), the latter in double quotes (`"..."`); this notation equals to the one used in Turtle.

In this notation no prefix can be declared, according to the streaming-friendly policy, so they have not been used. This way each line is meaningful and does not depends from other ones Sometimes this rule is overcome to obtain a more human-readable format, such as while debugging, but it's not a common practice, because the reader should know in advance Prefix to URIref mapping (otherwise information are useless).

ADVANCED FEATURES What have been presented are the fundamentals of RDF. In the following we will walk to some advanced construct.

Blank nodes

A **blank node** is a resource whose URIref has not been designated to be globally valid; this feature is achieved by omitting URIref's namespace (generally, in Turtle, is indicated with an underscore).

Blank nodes are used to represent *existential variables*, that are variables not bound to a specific resource, because they model an abstract concept. An example is the phrase: "*There's someone special out there for everyone.*": in order to express this information in RDF we cannot use specific resources because the concepts of someone and everyone are generic.

Stating that a person has a residence at a particular address is potentially another use case. In this example, if we use a global variable, we are forced to identify a particular residence and, in case we have two models defining a globally valid residence, their merge would produce a double residence. On the other side, if we use a blank node, the residence, which is only a mean to associate a residency to a person, has only local meaning and the model is correct.

Examples of how to use blank nodes are also reported in the paragraphs about Reification and RDF Containers (see RDF List example, on Listing 9). Here we report only the shorthand provided by Turtle to obtain a compact formalization (with reference to the residency example[Hebler *et al.*, 2009]).

Listing 6: Blank nodes in action

```
@prefix ex: <http://example.org/residences> .
@prefix sw: <http://semwebprogramming.net/resources#> .

sw:Bob sw:hasResidence [
  sw:isInCity "Arlington" ;
  sw:isInState "VA"
  ] .
```

Reification

The **Reification** process consist in *stating something about another RDF statement*. This is an extremely valuable tool for practical Semantic Web systems. It can be employed to qualify or annotate information in useful ways. One application might be to tag information with provenance or with a creation timestamp.

As usual an example is reported in the following.

Listing 7: Reification in action

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix : <http://www.semanticwebprogramming.net/people> .

:Matt :asserts _:stmt .

_:stms a rdf:Statement ;
  rdf:subject :John ;
  rdf:predicate foaf:knows ;
  rdf:object :Ryan .
```

In this RDF model we stated that "Matt says that John knows Ryan". A **RDF Statement** is a resource which always has three properties: a `rdf:subject`, a `rdf:predicate` and a `rdf:object`.

RDF Containers

RDF defines *three types of resources* that are understood to be collections of resources:

RDF:BAG (OR RDF:LIST) is used to represent an unordered grouping of resources

RDF:SEQ is used to represent an ordered one

RDF:ALT a `rdf:Bag` whose elements represents equivalent alternatives

In order to fill the container there are two available options: *(1)* state a relation from the container to the contained resource whose predicate is `rdf:_n`, where `n` is the resource position, or *(2)* state a simpler relation whose predicate is `rdf:li`; in the latter case the ordering determines the actual position.

In the following an example, in Turtle notation, is reported:

Listing 8: RDF Containers: a rdf:Bag

```
@prefix ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
```

```
@prefix people: <http://www.semwebprogramming.org/people> .

ex:Authors a rdf:Bag
  rdf:_1 people:Ryan    # or rdf:li
  rdf:_2 people:Matt    # or rdf:li
  rdf:_3 people:Andrew  # or rdf:li
  rdf:_4 people:John    # or rdf:li

ex:Book ex:writtenBy ex:Authors
```

Unfortunately if we have two models that refer to a homonymous bag like the ex:Authors one, the result of a merge will be a single bag containing all the authors. To solve this problem, in the case this is actually a problem, an `rdf:List` can be used, in fact this class represents a sealed list, defined recursively (see the following listing, in Turtle).

Listing 9: RDF Containers: a rdf:List

```
@prefix ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
@prefix people: <http://www.semwebprogramming.org/people> .

ex:Authors a rdf:List ;
    rdf:first people:Ryan ;
    rdf:rest _:r1 .
  _:r1 a rdf:List ;
    rdf:first people:Matt ;
    rdf:rest _:r2 .
  _:r2 a rdf:List ;
    rdf:first people:Andrew ;
    rdf:rest _:r3 .
  _:r3 a rdf:List ;
    rdf:first people:John ;
    rdf:rest rdf:nil .

ex:Book ex:writtenBy ex:Authors
```

This can be a valuable tool, but it is an extremely awkward, cumbersome and unreadable way to represent RDF lists. Thankfully, Turtle provides very concise shorthand to represent RDF lists; previous example can then be reviewed as follows (on Listing 10).

Listing 10: RDF Containers: a handier rdf:List

```
@prefix ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
@prefix people: <http://www.semwebprogramming.org/people> .

ex:Book
  ex:writtenBy (people:Ryan people:Matt people:Andrew people:John)
      .
```

The syntax of the other structures has not been reported here, because they are similar. More details can be found in [Hebler *et al.*, 2009;

W3C, 2004], which has been adopted as source of material for this section.

2.1.2 RDF Schema

RDF provides a way to express simple statements about resources, using named properties and values. However, RDF user communities also need the ability to define the vocabularies (terms) they intend to use in those statements, specifically, to indicate that they are describing specific kinds or classes of resources, and will use specific properties in describing those resources.

That's what RDFSchema was born for. RDFSchema provides the facilities needed to define those classes and properties. In other words, *RDFSchema provides a type system for RDF* [W3C, 2004]. The RDFSchema type system is similar in some respects to the type systems of object-oriented programming languages such as Java.

The RDFSchema facilities are themselves provided in the form of an RDF vocabulary; that is, as a specialized set of predefined RDF resources with their own special meanings. *The resources in the RDFSchema vocabulary have URIrefs with the prefix* `http://www.w3.org/2000/01/rdf-schema#` *(conventionally associated with the namespace prefix* `rdfs`*)* [W3C, 2004]. Vocabulary descriptions (schemas) written in the RDFSchema language are legal RDF graphs.

The following subsections will illustrate RDFSchema's basic resources and properties.

Describing a taxonomy

In RDFSchema is possible to define a taxonomy of concepts exploiting two simple URIrefs:

RDFS:CLASS which represents a generic type or category

RDFS:SUBCLASSOF which models that the subject class extends or specialize the object one

A new class can be defined asserting that a resource has `rdf:type` the resource `rdfs:Class`; the subject resource will be the new class. For example, in Turtle notation, we can say that (namespaces declaration has been omitted, but are the same as in RDF specification, on Section 2.1.1):

Listing 11: An example of class definition

```
1 @prefix ex: <http://www.example.org/> .
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
3 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4
5 ex:MotorVehicle rdf:type rdfs:Class .
```

Obviously this new class can be the object of a `rdf:type` assertion. In order to specify that a class is subclass of another one (like in "A truck is a motor vehicle") we can state:

Listing 12: Example of subclassing

```
@prefix ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Truck rdf:type        rdfs:Class .
ex:Truck rdfs:subClassOf ex:MotorVehicle .
```

As said while explaining RDF/XML notation (see Footnote 1), once a vocabulary has been defined, it can be used to exploit typed node annotation.

Defining properties

After defining classes, another useful feature is defining which properties characterize those classes. This can be done through the following URIrefs:

RDF:PROPERTY the object of an rdf:type statement used to define a new property (like in class definition)

RDFS:DOMAIN a predicate used to specify the classes (object) which the property (subject) applies to

RDFS:RANGE a predicate used to specify the range of possible property's values, in particular what kind of classes (one, more than one, a subset, and so on)

The latter statement should be interpreted only if present: if no range has been specified nothing should be inferred. Another consideration is that in this way the definition of properties is not part of class definition, like in the Object-Oriented Paradigm (OOP); this choice has some consequences (in agreement with RDF's open world assumption, see Section 2.1.3):

- A defined property can be applied to every classes that wanted to have it just by adding a rdfs:domain, in particular to classes that has not defined yet;
- Once defined a property has a global scope, so it has a unique (set of) range(s) and a unique (set of) domain(s):
  - Is not possible to have locally-different ranges or domains for a property
  - If a new range (or domain) is defined somewhere, then it become globally valid

It's also possible to state the rdfs:subPropertyOf relation, like for classes. A complete example is reported in the following:

Listing 13: An example of property definition

```
@prefix ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf.syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ex:Person   rdf:type           rdfs:Class .

ex:age      rdf:type           rdf:Property .
ex:age      rdfs:range         xsd:integer .
ex:age      rdfs:domain        ex:Person .

ex:legalAge rdf:type           rdf:Property .
ex:legalAge rdfs:subPropertyOf ex:age .
```

Looking at this example a person could think about how to specify the range of `ex:legalAge` property, which is an `xsd:integer` whose value is greater than 18 (or 21 if we're in the USA). It is simply not possible in RDFSchema; in order to have that expressive power OWL is needed (see Section 2.1.3 in the following).

2.1.3 OWL

Web Ontology Language (OWL) is a tool for specifying semantics and defining knowledge models. It can be said that OWL represents an extension of RDFSchema, which provides more expressive power, but, in order to provide that power, it's computational complexity raises and in some cases becomes a not-decidable problem.

With the aim of limiting this problem from the whole specification, called OWL-Full, some subsets and profiles has been defined, providing limited versions of the language, otherwise useless. A diffuse example is **OWL-Description Logic (OWL-DL)**, named after Description Logic because it provides many of the capabilities of this kind of logic, which is an important subset of first-order logic; other examples are OWL-E Logic (OWL-EL), which is designed to perform consistency checks and instance-class mapping in polinomial-time, and OWL-Q Logic (OWL-QL), which is designed to enable the satisfiability of conjunctive queries in logspace with respect to the number of assertions in the queried knowledge base [Hebler *et al.*, 2009].

OWL is merely an ontology language; it is not an application. As such, OWL alone doesn't really do anything, but combined with a reasoner like Pellet (see Section 2.2.2), represent the added value against other approaches for exchanging information. A **reasoning engine** (or reasoner) is a system that *infers new information based on the contents of a knowledge base.* This can be accomplished using rules and a rule engine,

triggers on a database or RDF store, decision trees, tableau algorithms, or even programmatically using hard-coded business logic. Many Semantic Web frameworks perform inference using rules-based reasoning engines. *These engines combine the assertions contained in a knowledge base with a set of logical rules in order to derive assertions or perform actions.* Rules comprise two parts, modeling an if-then statement. The first part is the condition for the rule, and the second part is conclusion of the rule. *Rules can be used to express much of the semantics of OWL and as a tool for users to express arbitrary relationships that cannot otherwise be modeled in OWL.*

An ontology purpose is letting a concept be machine-understandable. Generally these concepts are spread in the WWW, so the knowledge base (expressed in RDF) is considered distributed. In the same way RDF supports distribution, OWL should support it too; this is called **distributed knowledge**. To provide a foundation on which to make valid inferences in this model, we must make two important assumptions:

OPEN WORLD ASSUMPTION : not knowing whether a statement is explicitly true does not imply that the statement is false; in this context new information must always be additive. It can be contradictory, but it cannot remove previously asserted information.

NO UNIQUE NAMES ASSUMPTION : it is unreasonable to assume that everyone is using the same URI to identify a specific resource; *unless explicitly stated otherwise, you cannot assume that resources that are identified by different URIs are different.*

An ontology, expressed in OWL-Full version 2.0, is composed by:

- Ontology header
- Classes and individuals
- Properties: object and datatype
- Annotations
- Datatypes: standard or restricted

Ontology header

An ontology header **is a resource that represents the ontology itself**, containing comments, labels, versioning and a list of the imported ontologies, which is very important because it instructs the reasoner to refer to them to in order to comprehend the expressed concepts and relationships.

Classes and individuals

An OWL class is a special kind of resource that *represents a set of resources that share common characteristics* or are similar in some way. *A resource that is a member of a class is called an individual* and represents an instance of that class.

Like in RDFSchema two classes might be one the subclass of the other; but, when not explicitly stated, a class is automatically subclass of `owl:Thing` (similarly to Java with Object). In OWL `owl:Nothing` is also defined: it is the subclass of every class, so specialized that *it has literally no individuals* (its an empty class).

Another relevant concept is *class equivalence*, in fact in general two classes, even if has the same local name, may express different ideas. *Two classes are equivalent only if explicitly stated* in the ontology, otherwise they aren't; *extension does not imply equivalence* because they are asserted with different unrelated properties.

The individuals of a class may be uniquely identified by a subset of their properties (other than by URIref); these properties determine a **key**, which can be enumerated in a special property called `owl:hasKey`.

Properties

In OWL properties can be of two types, based on the resources used:

OWL:DATATYPEPROPERTY : these properties has a literal object

OWL:OBJECTPROPERTY : these properties state a relation between two individuals

The predicates `rdf:domain` and `rdf:range` globally define the class membership for property subject and object. This implies that a class which does not match with membership definition cannot use those properties, avoiding unwanted and unpredicted inferences; if `ex:name` is a property for `ex:Mammal` and we use the same property for `ex:Institution` we automatically assume that an institution is a mammal, which is false.

As with classes, properties can be arranged into taxonomies using the property `rdfs:subPropertyOf`. **Be careful that asserting that property2 is sub-property of property1 implies that two resources in relationship by property2 are also related by property1**. Similar to `owl:Thing` and `owl:Nothing`, `owl:topObjectProperty` and `owl:bottom ObjectProperty` represents the most general an most specific properties of the taxonomy with reference to `owl:ObjectProperties` domain; `owl:topDataProperty` and `owl:bottomDataProperty` are the same for `owl:DataProperties`.

Other features of properties, descending from mathematics, are:

INVERSE PROPERTIES : a property that states the inverse relationship stated by another property (e.g. if A owns B then B isOwnedBy A)

DISJOINT PROPERTIES : if two resources are related by propX then they cannot be related by propY (e.g. if A hasFather B then not(A hasMother B))

PROPERTY CHAINS : using a couple or more properties two define a new one (e.g. if A hasFather B and B hasBrother C then A hasUncle C)

SYMMETRIC, REFLEXIVE, TRANSITIVE PROPERTIES : like in mathematic relationships

(INVERSE) FUNCTIONAL PROPERTY : each subject (object) is related to only one object (subject)

Negative Property Assertion

While in general a property is stated in order to assert that something is true, it is also possible to define the opposite (that if stated then something is false) using the type `owl:NegativeProperty`. The following example models that is not true that `ex:Daisy ex:hasOwner ex:Amber`.

Listing 14: An example of Negative Property Assertion

```
@prefix ex: <http://example.org/> .
...

[] rdf:type owl:NegativePropertyAssertion;
  owl:sourceIndividual ex:Daisy;
  owl:assertionProperty ex:hasOwner;
  owl:targetIndividual ex:Amber.
```

Annotations

Annotations are statements that describe resources using annotation properties. Annotation properties are semantics-free properties.

Common examples of annotations are: `rdf:label`, `rdf:comment` and `rdf:versionInfo`.

Datatypes

Datatypes represent ranges of data values that are identified using URIs. OWL allows you to use a number of predefined datatypes, most of which are defined in the XML Schema Definition (XSD) namespace.

Otherwise OWL allows the definition of custom datatypes, in two ways:

- by **Datatypes Restriction**, which consists in defining a new `rdfs:Datatype` and specifying:

  OWL:ONDATATYPE : a primitive datatype

  OWL:WITHRESTRICTIONS : a list of facets (constraints) on data value, length, char pattern and so on

- by **Definition in terms of other Datatypes**, in particular by intersection or union of a bunch of datatypes (primitive or custom)

Property Restrictions

A property restriction describes the class of individuals that meet the specified property-based conditions. The restriction is declared using the construct `owl:Restriction`, and the property to which the restriction refers is identified using the property `owl:onProperty`.

Restrictions are applied to a particular class by stating that the class is either a subclass (`rdfs:subClassOf`) or the equivalent class (`owl:equivalentClass`) of the restriction. In the *former* case *all members of the class must meet the conditions* specified by the restriction, in the *latter* case class members must meet the conditions of the restriction and *any individual who meets the conditions of the restriction is implicitly a member of the class* (restriction specifies conditions that are not only necessary but also sufficient).

The OWL specification requires that restriction cannot be named and must be defined using anonymous resources. This is a reasonable condition because restrictions are relevant only to the context of the class in which they are defined and never need to be referred to.

There are *three types* of Property Restrictions: *(1)* Value restriction, *(2)* Cardinality restriction and *(3)* Qualified cardinality description, which is a combination of the former two.

VALUE RESTRICTION A Value restriction can be formalized in three different ways, whose difference is relative to the number of occurrences of the restricted property that must fit the constraint.

OWL:ALLVALUESFROM states that each occurrence of the property (if any) must have the object value in the specified range

OWL:SOMEVALUESFROM states that at least one occurrence of the property must have the object value in the specified range

OWL:HASVALUE states that an occurrence of the property must have the specified object value

Listing 15: Some examples of Value Restrictions

```
@prefix ex: <http://example.org/>.
...

# owl:allValuesFrom example
# This restriction defines a class, ex:CallableEntities,
# whose name have a range of xsd:string
ex:CallableEnties rdfs:subClassOf [
  rdf:type owl:Restriction;
  rdf:onProperty ex:name;
```

```
  owl:allValuesFrom xsd:string
  ].

# owl:someValuesFrom example
# This restriction defines a class whose members
# extends a serializeable class
ex:SerializeableClass rdfs:subClassOf [
  rdf:type owl:Restriction;
  rdf:onProperty ex:extends;
  owl:someValuesFrom ex:SerializeableClass;
  ].

# owl:hasValue example
# This restriction define a class of mammals
# whose owner is ex:Ryan
ex:Mammal rdf:type owl:Class.
ex:hasOwner rdf:type owl:ObjectProperty.

ex:PetsOfRyan rdf:type owl:Class;
  rdfs:subClassOf ex:Mammal;
  rdfs:subClassOf[
  rdf:type owl:Restriction;
  owl:onProperty ex:hasOwner;
  owl:hasValue ex:Ryan
  ].
```

owl:SelfRestriction

There's also another type of value restriction, `owl:SelfRestriction`, which *states that value of the property is the subject itself*. `owl:SelfRestriction` is an alternative to `owl:Restriction`.

Listing 16: Defining an owl:SelfRestriction

```
@prefix ex: <http://example.org/>.
...

# owl:SelfRestriction example.
ex:cleans rdf:type owl:ObjectProperty.

# Self cleaners are all individuals who clean
# themselves (range is implicit)
ex:SelfCleaner rdf:type owl:Class;
  owl:equivalentClass [
    rdf:type owl:SelfRestriction;
    owl:onProperty ex:cleans
    ].
```

CARDINALITY RESTRICTION It provides a way to specify how many occurrences of the property are required. This can be done using the following predicates (obviously N >= 0):

OWL:MINCARDINALITY , #occs >= N

OWL:MAXCARDINALITY , #occs <= N

OWL:CARDINALITY , #occs == N

Listing 17: An example of Cardinality Restriction

```
@prefix ex: <http://example.org/>.
...

ex:Person rdfs:subClassOf [
  rdf:type owl:Restriction;
  owl:onProperty ex:name;
  owl:cardinality 1
  ].
```

QUALIFIED CARDINALITY RESTRICTION It represents a way to define both cardinality and type. It is useful to model a system made of some components of which you need to *assert that there are N components of type X, M components of type Y and so on*.

Listing 18: An example of Qualified Cardinality Restriction

```
# This examples models that a person has 2 biological
# parents: a male and a female.
@prefix ex: <http://example.org/>.
...

ex:Person rdf:type owl:Class;
ex:Male rdf:type owl:Class;
  rdfs:subClassOf ex:Person.
ex:Female rdf:type owl:Class;
  rdfs:subClassOf ex:Person.

ex:hasBiologicalParent rdf:type owl:ObjectProperty.

ex:Person rdfs:subClassOf [
    rdf:type owl:Restriction;
    owl:cardinality 2;
    owl:onProperty ex:hasBiologicalParent
    ];
  rdfs:subClassOf [
    rdf:type owl:Restriction;
    owl:qualifiedCardinality 1;
    owl:onProperty ex:hasBiologicalParent;
    owl:onClass ex:Male
    ];
  rdfs:subClassOf [
    rdf:type owl:Restriction;
    owl:qualifiedCardinality 1;
    owl:onProperty ex:hasBiologicalParent;
    owl:onClass ex:Female
    ].
```

Advanced Class Description

OWL provides a few more methods for describing classes.

ENUMERATING CLASS MEMBERSHIP In some cases you may be interested in the enumeration of the only individuals that can be members of a defined class. This can be done with the predicate `owl:oneOf`.

Listing 19: How to enumerate class instances

```
@prefix ex: <http://example.org/>.
...

ex:Daisy rdf:type ex:Canine.

ex:Cubby rdf:type ex:Canine.
ex:Amber rdf:type ex:Canine.
ex:London rdf:type ex:Canine.

# Each friend of Daisy's is explicitly included in this class
ex:FriendsOfDaisy rdf:type owl:Class;
  owl:oneOf (
    ex:Cubby
    ex:Amber
    ex:London
  ).
```

SET OPERATORS In order to define new classes it could be useful to specify that a class is the union, intersection or complementary of two or more classes. That's why `owl:unionOf`, `owl:intersectionOf`, `owl:complementaryOf` exists. An example of each one is reported in the following.

Listing 20: Using set operators

```
@prefix ex: <http://example.org/>.
....

# Example 1: intersection of
ex:PetsOfRyan rdf:type owl:Class;
  owl:intersectionOf (
    ex:Mammal
    [
      rdf:type owl:Restriction;
      owl:onProperty ex:hasOwner;
      owl:hasValue ex:Ryan
    ]
  ).

# Example 2: union of
ex:isFriendsWith rdf:type owl:ObjectProperty.

ex:FriendsOfRyan rdf:type ex:Class;
```

```
  owl:unionOf (
    [
      rdf:type owl:Class;
      owl:oneOf (
        ex:Daisy
      )
    ]
    ex:FriendsOfDaisy
    [
      rdf:type owl:Restriction;
      owl:onProperty ex:isFriendsWith;
      owl:hasValue ex:Ryan
    ]
  ).

#Example 3: complement of
ex:EnemiesOfRyan rdf:type owl:Class;
  owl:complementOf ex:FriendsOfRyan.
```

DISJOINTED CLASS Similarly to properties case *it is possible to define that if an individual is instance of a class it cannot be also instance of another class, called disjointed*.

Listing 21: Example of Disjoint class

```
@prefix ex: <http://example.org/>.
...

# canine and human are disjoint classes
ex:Canine owl:disjointWith ex:Human.
```

In order to reduce the time spent in defining disjunctions (which is considerably augmenting with the model size), some shorthands has been defined:

OWL:ALLDISJOINTEDCLASSES is an anonymous resource whose owl:members property refers to a list of classes that will be considered disjointed. Listed classes must be defined as subclasses of the same root type.

OWL:DISJOINTUNIONOF let directly define a class as the union of a list of disjointed classes

The drawback of the use of a disjoint union to define a superclass is that any future attempt to incorporate a new subclass will require you to redefine the disjoint union to include the new subclass.

Let's present some examples (Listing 22).

Listing 22: Shortcuts for Disjoint class definition

```
@prefix ex: <http://example.org/>.
...
```

```
3
4  # --> Example 1
5  # canine and human are disjoint classes
6  ex:Canine owl:disjointWith ex:Human.
7
8  # --> Example 2
9  ex:Animal rdf:type owl:Class.
10
11 ex:Bird rdf:type owl:Class;
12   rdfs:subClassOf ex:Animal.
13 ex:Lizard rdf:type owl:Class;
14   rdfs:subClassOf ex:Animal.
15 ex:Feline rdf:type owl:Class;
16   rdfs:subClassOf ex:Animal.
17
18 # Each of the classes is pair-wise disjoint
19 _: rdf:type owl:AllDisjointClasses;
20   owl:members ( ex:Bird ex:Lizard ex:Feline ex:Canine ).
21
22 # --> Example 3
23 # Each of the classes is pair-wise disjoint
24 # and Animal is the union of those classes
25 ex:Animal owl:disjointUnionOf (
26   ex:Bird
27   ex:Lizard
28   ex:Feline
29   ex:Canine
30 ).
```

Equivalence in OWL

At this point the last thing to be told is how OWL states the equivalence between two entities.

There are several ways of defining an equivalence and each one of them involves different type of entities, which involves: Individuals, Classes, or Properties.

EQUIVALENCE AMONG INDIVIDUALS It can be stated in two ways: *(1)* asserting that two individuals are equal, *(2)* assuring that two individuals have nothing in common (not-equals).

In the former case the `owl:sameAs` predicate is used to suggest that two URIref refer to the same resource, even if syntactically different, while in the latter case the `owl:differentFrom` predicate is available. `owl:AllDifferent` can be used In order to reduce the number of declarations, like for disjoint classes:

This property is very important, because the combination of the open world assumption and the no unique names assumption results in an environment where there are very few situations in which you can assume that resources identified by different URIs are actually different.

Listing 23: Example of Equivalence among Individuals

```
@prefix ex: <http://example.org/>.
...

# owl:sameAs example
<mailto:rblace@bbn.com>
  owl:sameAs <http://example.org/people#rblace>.

# owl:AllDifferent example
[] rdf:type owl:AllDifferent;
  owl:distinctMembers (
    ex:Daisy
    ex:Cubby
    ex:Amber
    ex:London
  ).
```

EQUIVALENCE AMONG CLASSES When asserted (via the `owl:equivalentClass`) causes two classes to be considered as a single resource.

EQUIVALENCE AMONG PROPERTIES When you assert that two properties are equivalent (via the `owl:equivalentProperty`), the property descriptions are combined. *Every statement that uses one of the properties as a predicate implicitly exists with the other equivalent property as a predicate as well.*

2.1.4 SPARQL

Simple Protocol and RDF Query Language (SPARQL) is a W3C Recommendation; other alternatives exists (e.g. RDQL and SeRQL) but they are not recommended so they will not become a future standard.

The specification defines the concept of endpoint (or processor) as a service (generally a Web service) that accept and processes SPARQL queries and returns results in different formats, depending on the query form, via HTTP, using the so called SPARQL protocol.

Despite the query language is the most visible and known part of the specification, even the protocol is an important part of it, otherwise clients and effector would not be able to interact. Following the stream we will focus on the query language because it's more relevant: thanks to the Jena framework (see Section 2.2.1) it is possible to perform queries without knowing the protocol.

SPARQL language is very similar to SQL, in particular for what concerns the SELECT statement (obviously adapted in order to handle RDF features). Unlike SQL, this language is read-only, in fact it does not provide any statement that enables the requestor to modify RDF mod-

els; This goal is reached using another language: SPARQL/Update (SPARUL) which is described in Section 2.1.5

The *four foundational query forms* available in SPARQL are:

SELECT : like SQL SELECT it returns a table which reports all instances (in this case statements, individuals) which can be found in places enumerated in the FORM clause and fit a set of conditions specified in the WHERE clause, called graph pattern.

CONSTRUCT : it provides an easy and powerful way to map some RDF information in a different, but legal, form; the result can be added to or merged with other RDF stores.

ASK : it checks if a particular graph exists and returns the verdict as a boolean value

DESCRIBE : it is an interesting tool that enables the interrogation of the RDF store without requiring a specific knowledge of repository's data structure; in fact, given a few information, the endpoint will decide what should be returned.

In the following subsections each form is covered in details, with a particular focus on valid syntax.

SELECT statement

A simple SELECT query is structured as follows:

1. **Preamble**: it reports all prefixes that will be used in the query. An example of prefix declaration is (similar to Turtle): `PREFIX ex: <http://www.example.org#>`. In the same area a BASE can be defined: in this way each time a relative URIref is specified the BASE is taken to obtain the absolute one, like a sort of implicit prefix. The syntax is similar to the previous one, just replace PREFIX with BASE.

2. **SELECT** followed by the list of variables whose bindings should be returned, or a star ("∗") if every binding should be reported. After the keyword is possible to specify some modifiers:
   - **DISTINCT** in order to obtain a table without duplicates
   - **REDUCED** in order to suggest (without imposing it) the application of `DISTINCT` to the endpoint
   - **ORDER BY** in order to require a sorting process: it uses keywords `ASC(?var)` and `DESC(?var)`
   - **OFFSET** and **LIMIT** are useful when returning a large amount of data. Used with `ORDER BY`, which allows a correct handling (it makes result well-ordered and repeatable), it instructs the endpoint to return at most `LIMIT` solutions, starting from the `OFFSET` one

3. **WHERE { }** which encloses a *graph pattern* (a set of partially specified RDF statements) which determines what kind of triples the requestor is searching for. In order to have a valid clause, each line specified should match Turtle syntax; if a term is a variable its name should be prefixed by a question mark ("?").

In order to add flexibility in quering, as the real world requires, some keywords (to be used in graph pattern definition) have been introduced:

FILTER returns a subset of pattern matching triples which verify the specified boolean conditions.

Listing 24: Example of FILTER usage

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?prop ?object
  WHERE {
    ?person rdfs:label "George Washington"@en;
      dbprop:presidentStart ?start;
      ?prop ?object.
    FILTER(xsd:integer(?object) >= xsd:integer(?start) + 4)
}
```

OPTIONAL allows to collect additional information without discarding any triple if it does not have them but leaving, a blank field instead.

Listing 25: Example of OPTIONAL usage

```
# Hopefully all of George Washington's Namesakes!
PREFIX ex: <http://www.example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbprop: <http://dbpedia.org/property/>

SELECT ?l1 ?l2 ?l3 ?l4
WHERE {
  ?person rdfs:label "George Washington"@en.
  ?l1 dbprop:namedFor ?person.
  OPTIONAL { ?l2 dbprop:blankInfo ?person }
  OPTIONAL { ?l3 ex:isNamedAfter ?person }
  OPTIONAL { ?person ex:wasFamousAndGaveNameTo ?l4 }
}
```

If more than a pattern is specified inside an `OPTIONAL` clause, then all of them should match, otherwise all mentioned fields are left blank.

UNION let the specification of the result as all the triples that match at least one of the specified *sub-graph patterns*.

Listing 26: Example of UNION usage

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
  { ?unknown foaf:gender "male" } UNION { ?unknown foaf:gender
      "female2" } .
  {
    ?unknown foaf:interest <http://www.iamhuman.com>
  } UNION  {
    ?unknown foaf:interest <http://lovebeinghuman.org>
  }
}
```

Until now data has been retrieved by a so called *default graph*, that's the repository where the endpoint searches for results. **Generally a query is run against at least one default graph and/or one or more named graphs**, which can be specified by the requestor via the **FROM** and **FROM NAMED** clauses:

FROM followed by an URIref enclosed in angle brackets (`FROM <http://www.w3.org/People/Berners-Lee/vcard>`). Each specified repository is merged with the others and the default graph (if any) and the pattern is applied to the whole merged repository.

FROM NAMED allows the declaration of *subgraph queries*. While defining the `WHERE` clause, it is possible to scope each pattern, applying it to a specific named graph or at each graph separately. In the following example we want to obtain, for each repository, the list of nickname and realname pairs of each known person[2].

Listing 27: Example of subgraph queries

```
SELECT *
FROM NAMED <http://www.koalie.net/foaf.rdf>
FROM NAMED <http://heddley.com/edd/foaf.rdf>
FROM NAMED <http://www.cs.umd.edu/~hendler/2003/foaf.rdf>
...

WHERE {
  GRAPH ?originGraph {
    _:blank1 foaf:knows _:blank2.
    _:blank2 rdf:type foaf:Person;
      foaf:nick ?nickname;
      foaf:name ?realname
  }
}
```

2 The blank node is used as a sort of *hidden variable* which can assume every value but cannot be reported in the result table.

CONSTRUCT statement

This statement runs a `SELECT` and then produces a set of RDF statements using all values bound to the variables.

An example of its syntax is reported in the following.

Listing 28: Example of CONSTRUCT query

```
PREFIX myfriends: <http://www.example.com/2008/myfriends/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
  ?person rdf:type myfriends:Humanoid;
    myfriends:handle ?rname;
    myfriends:homepage ?hpage;
    myfriends:informalName ?nick;
    myfriends:email ?mbox.
  ?mbox myfriends:isOwnedBy ?person.
  ?hpage myfriends:isManagedBy ?person.
}
FROM NAMED ... # The following is like a SELECT query
```

The first part defines the structure of RDF triples that will be created, in the second a `SELECT` query, which will provide values for the variable that has been used, is stated (`FROM`, `FROM NAMED` and `WHERE` clauses).

ASK statement

`ASK` returns boolean values in response to a query. Given a graph pattern, an endpoint can tell you whether or not the pattern exists in the underlying data store. An example is reported in Listing 29.

Listing 29: Example of ASK query

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
WHERE {
  ?person rdfs:label "George Washington"@en;
    dbprop:presidentStart ?startDate;
    dbprop:presidentEnd ?endDate.
  FILTER(xsd:integer('1795') > xsd:integer(?startDate) &&
    xsd:integer(?endDate) > xsd:integer('1795'))
}
```

DESCRIBE statement

A statement of this type is useful in order to obtain a reasonable amount of information about a resource even if we don't know anything about it.

Given a resource like `dbpedia:George_Washington` we can request:

Listing 30: Example of DESCRIBE query

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
DESCRIBE dbpedia:George_Washington
```

Each endpoint has its own policy to handle this type of requests, so the result is not standard and not every processor is able to provide it.

It is also possible to provide a `WHERE` clause and use a syntax like the one in the following, but *the former solution should be preferred*.

Listing 31: DESCRIBE query with WHERE clause

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
DESCRIBE *
WHERE {
  ?person ?anyProperty dbpedia:George_Washington
}
```

Final note

All information that has been presented in this section is referred to version 1.0 of the protocol, but since 05 January 2012 a new version (1.1) has been submitted. It is now a *Working Draft* [W3C, 2012a], supported by last releases of the technologies presented in Section 2.2.

2.1.5 SPARQL Update

SPARUL (a.k.a. SPARQL Update) is a language to express updates to a Graph store; a Graph Store is a repository of RDF graphs managed by a single service. SPARUL is intended to be a standard mechanism by which updates to a remote RDF Store can be described, communicated and stored. It is a companion language to SPARQL and is envisaged to be used in conjunction with the SPARQL query language.

The reuse of the SPARQL syntax, in both style and detail, reduces the learning curve for developers and reduces implementation costs.

SPARUL provides the following facilities:

- Insert new triples to an RDF graph.
- Delete triples from an RDF graph.
- Perform a group of update operations as a single action.
- Create a new RDF Graph in a Graph Store.
- Delete an RDF graph from a Graph Store.

SPARUL is not intended to distribute and exchange modifications among different RDF stores.

In the following we report basic syntax for each statement (`template`, `pattern` and `triple` entities are expressed in SPARQL syntax).

INSERT Inserts a set of triples

Listing 32: SPARQL/Update INSERT syntax

```
{"PREFIX" namespace-declaration}

"INSERT" ["DATA"] ["INTO" uri] "{"
  (template | triples)
"}"
[ "WHERE" "{"
  pattern
"}" ]
```

- If `DATA` is specified, then ground triples (no variables) must be provided; otherwise a pattern should be specified
- This statement is a special case of a `MODIFY` statement

DELETE Removes a set of triples

Listing 33: SPARQL/Update DELETE syntax

```
{"PREFIX" namespace-declaration}

"DELETE" ["DATA"] ["INTO" uri] "{"
  template
"}"
[ "WHERE" "{"
  pattern
"}" ]
```

- Considerations made for the `INSERT` clause still hold.

MODIFY Modifies an RDF graph, deleting and inserting some RDF triples in a single operation

Listing 34: SPARQL/Update MODIFY syntax

```
# UPDATE outline syntax : general form:
"MODIFY" [ <uri> ]
"DELETE" "{" template "}"
"INSERT" "{" template "}"
[ "WHERE" "{" pattern "}" ]
```

- The `WHERE` clause is evaluated only once
- `DELETE` must occur before `INSERT`

LOAD This statement copies all triples from the `remoteURI` to the specified (or default) graph.

Listing 35: SPARQL/Update LOAD syntax

```
"LOAD" <remoteURI> [ "INTO" <uri> ]
```

CLEAR This statement removes all triples in the specified (or default) graph.

Listing 36: SPARQL/Update CLEAR syntax

```
1 "CLEAR" [ "GRAPH" <uri> ]
```

CREATE GRAPH This statement creates a new graph; the provide URI will be its name.

Listing 37: SPARQL/Update CREATE GRAPH syntax

```
1 "CREATE" [ "SILENT" ] "GRAPH" <uri>
```

The `SILENT` keyword is intended to suppress a warning raised, by the endpoint, if creating an already existing graph.

DROP GRAPH This operation removes the specified named graph from the Graph Store associated with the SPARUL service endpoint. After successful completion of this operation, the named graph is no longer available for further graph update operations.

Listing 38: SPARQL/Update DROP GRAPH syntax

```
1 "DROP" [ "SILENT" ] "GRAPH" <uri>
```

The `SILENT` keyword is intended to suppress a warning raised, by the endpoint, if removing a graph that does not exists.

Final note

Reported information has been taken from [W3C, 2008b] (for more detail follow the previous link). On January, 5 2012 a new working draft, which introduces relevant modifications, has been published ([W3C, 2012b]).

2.2 TECHNOLOGIES

The libraries reported in the following are not the only options available for working in the Semantic Web field: on Internet a plenty of solutions exist, each one with its own strengths and weaknesses. *Apache Jena* and *Pellet* have been chosen on the base of previous work from SAPERE team members (WP1 and WP3), and because they're two mature projects, providing a sufficiently stable and easy to exploit framework, because developed in the Java programming language.

2.2.1 Apache Jena: RDF Graph Store

The **Jena Semantic Web Framework**, initially developed by HP laboratories and now moved to the Apache Incubator project[3], provides a set

3 More details on Apache Incubator at `http://incubator.apache.org/`

of RDFStore implementations fitting different requirements and levels of abstraction. At the higher level (the Application Programming Interface (API)) a store is called `Model`: its interface is meant to handle information in the form of `Resources`, properties, URI, `Literals` and `Statements` (each of these concept is explained in Section 2.1).

A `Model` can be created through a `ModelFactory`, which provides a plenty of methods, designed to produce models with different capabilities[4]:

DEFAULTMODEL is an elementary in-memory unnamed model, with no persistence support (it means that model content is not automatically serialized in files or databases when the application is shut down).

INFMODEL is a model meant to be used in combination with a *reasoner* for inferred statements storage

TDBMODELFACTORY is a totally different type of factory, meant to provide a realization tuned for *high-performance* applications, with a database back-end which provides fast and persistent storage with also ACID transaction support, through a *write-ahead-logging* [Foundation, 2011]

Resources can be created an retrieved, calling `createResource` method, decorated with properties (through `createProperty`, `addProperty` and so on) inspected and deleted (`listXXX` and `removeXXX`). An example of usage is reported in the following listing.

Listing 39: Example of Jena API usage [Foundation and HP-Labs, 2010]

```
String personURI    = "http://somewhere/JohnSmith";
String person2URI    = "http://somewhere/JoeBlack";
String givenName    = "John";
String familyName   = "Smith";
String fullName     = givenName + " " + familyName;

Model model = ModelFactory.createDefaultModel();

Resource johnSmith = model.createResource(personURI)
  .addProperty(VCARD.FN, fullName)
    .addProperty(VCARD.N,
      model.createResource()
          .addProperty(VCARD.Given,
            givenName)
            .addProperty(VCARD.Family,
              familyName));

Resource joeBlack = model.createResource(person2URI)
```

4 only a relevant subset of all models has been reported

```
19   .addProperty(VCARD.FN, "Joe Black")
20   .addProperty(VCARD.EMAIL,
21     "joe.black@me.com");
22
23 model.remove(
24   model.listResourcesWithProperty(VCARD.N));
```

In order to obtain *quering capabilties*, the Jena System Programming Interface (SPI) is required: it provides a low-level access where the concepts of `Graph` and `GraphStore` take the place of `Models`. A `GraphStore` can be derived from a `Model` and automatically kept synchronized (in fact they are two different abstractions of the same information): this way the power of SPARQL and SPARUL can be exploited only when required; in case of simple listings the navigation interface, provided by the API, should be used (see Listing 39, line 23).

To run a query two steps are required:

1. **Creation** A factory should be used, in order to parse the query from a String representation:

   QUERYFACTORY creates a `Query` from SPARQL.

   Listing 40: SPARQL query creation

   ```
   1 String sparql    = "SELECT * WHERE { }";
   2
   3 Query query = QueryFactory.create(sparql);
   ```

   UPDATEFACTORY creates an `UpdateRequest` from SPARUL, or programmatically.

   Listing 41: SPARUL query creation

   ```
   1 String sparul    = "INSERT DATA {
   2   <http://www.example.org#subject>
   3     <http://www.example.org#predicate>
   4       <http://www.example.org#object>.
   5   }";
   6
   7 UpdateRequest query = UpdateFactory
   8   .create(sparul);
   ```

2. **Execution** Two other factories are available in order to generate a `Processor` able to visit the `Graph` and run the desired operations:

   QUERYEXECUTIONFACTORY with `Query` objects

   Listing 42: SELECT query execution

   ```
   1 Query query = ...; // From creation phase
   2 Model aModel = ModelFactory
   3   .createDefaultModel();
   ```

```
4
5 ResultSet result = QueryExecutionFactory
6   .create(query, aModel).execSelect();
```

As discussed in Section 2.1.4, SELECT is not the only statement available in the language. Given the specific statement (`ASK`, `DESCRIBE`, `CONSTRUCT`) the right method should be called: `execAsk()`, `execDescribe()`, `execConstruct()`.

After the execution, each match that has been found is returned in the form of a `ResultSet`, which is an iterator over `QuerySolution` items (one per match). Each solution, retrievable through `result.next()` contains the mapping between variables and values, that can be retrieved invoking the method `sol.get(String)` on it and passing the variable name.

`UPDATEEXECUTIONFACTORY` with `UpdateRequests`

Listing 43: SPARUL query execution

```
1 UpdateRequest uReq = ...; // From creation phase
2 Model aModel = ...;
3 GraphStore aGraphStore = GraphStoreFactory
4   .create(DatasetFactory.create(aModel));
5
6 UpdateExecutionFactory
7   .create(uReq, aGraphStore[, bindings])
8           .execute();
```

The `bindings` parameter, whose type is `QuerySolution`, is *optional*: it is meant to be used to set some variables values according to the result of a SPARQL query.

Apache Jena 2.7.0

As an alternative, from Apache Jena 2.7.0, `UpdateAction` can be used. It provides both parsing and execution facility, but does not support the *bindings-passing* option[5].

2.2.2 Pellet: OWL-DL Reasoner

Pellet is a Java-based OWL-DL reasoner (a full list of library features is reported in [mindswap, 2003] and [Clark&Parsia, 2011]). It is available under dual licensing terms: *(1)* an *open-source* license for open-source projects and *(2)* a *proprietary, closed-source* one for commercial application with commercial support.

It is an interesting choice, because it can be easily integrated with Apache Jena in two manners:

5 with reference to with the bindings argument for the UpdateExecutionFactory.create(UpdateRequest, GraphStore, QuerySolution) method

- through the **DIG** interface (see `http://dig.sourceforge.net/`), which exploits an HTTP-based protocol, letting the application communicate with Pellet in a separate process;
- through a **dedicated** interface, which extends Jena's `Reasoner` interface, providing a faster interaction.

The first solution generates a lot of overhead (that's why the second one is generally preferred) but it is safer in case of multi-thread applications. Despite this handicap it's possible to enforce thread-safety by serializing RDF model access with a mutex lock and triggering the classification process[6] in specific spots where that lock has been already acquired (or using DIG and accepting the overhead) [Clark&Parsia, 2011].

Let's now look at how to exploit Pellet capabilities in conjunction with Apache Jena. The process requires the creation of one or more Jena's `Models` and establishing a link to the reasoner[7] Two approaches exist: the former consists in a one-line statement which uses the predefined specification provided by developers, the latter will manually build the required set of models; it is more complex than the former, but allows more customization (if needed).

Check the following listing (Listing 44) for details.

Listing 44: Pellet-Jena usage

```
1  // Approach #1
2  OntModel model = ModelFactory
3      .createOntologyModel(
4        PelletReasonerFactory.THE_SPEC);
5
6  // Approach #2
7  Reasoner reasoner = PelletReasonerFactory
8      .theInstance().create();
9  Model infModel = ModelFactory.createInfModel(reasoner,
10     ModelFactory.createDefaultModel());
11
12 return ModelFactory.createOntologyModel(
13     OntModelSpec.OWL_DL_MEM, infModel);
```

Each time a RDF Triple is inserted, updated or removed the `Reasoner` decides if the Knowledge Base should be refreshed and classified. The same process can be triggered through the interface, calling the `refresh()` method (on `Reasoner`) or, if custom initialization has been used, retrieving the `InfGraph` instance, casting it to `PelletInfGraph` and invoking the `classify()` operation.

6 The classification process which is the computation of inferred subclass relationships. Other features, included in the reasoner, are: (1) Realization, the process that links individuals with classes, (2) Consistency Checking, which checks ontology consistency

7 Pellet is not the unique choice: Jena is shipped with a built-in reasoner and other third-part solutions can be found and, probably, linked to Jena's Models but here we deepen only Pellet integration

3 THE SAPERE MODEL

CONTENTS

The content of this chapter is based on SAPERE project's deliverable D1.1 [Viroli *et al.*, 2011] and technical report TR.2011.06 [Stevenson and Viroli, 2011]. The first section is dedicated to the presentation of framework concepts, while the second one reports a proposal for a semantic-based realization.

3.1 DEFINING SAPERE DOMAIN

The SAPERE framework is meant to satisfy typical requirements of pervasive computing scenarios by adopting a bio-chemical inspired approach. Self-aware pervasive services ecosystems try to address problems like:

MOBILITY SAPERE agents should deal with people movement in physical space, because generally running on mobile devices.

SHARED-ENVIRONMENT as means of coordination between agents, which manifest themselves affecting a local space and perceive other resources manifestations.

SITUATEDNESS Activities of SAPERE agents should depend from the place they are located in. This can be achieved by restricting their actions to a portion of environment their allowed to interact with, so defining a context, they are aware of, by environment perception[Viroli *et al.*, 2011].

SELF-ORGANISING COORDINATION LAWS The environment should handle dynamism and complexity, through a set of local stochastic coordination laws, designed to make global patterns emerge, as in nature. The idea behind the SAPERE project is considering information as *molecule*s which can be bond, as a

form of interaction, and resembled through chemical-like laws (see Section 3.1.2). These molecules could also be diffused in the *neighborhood* like in physics and biology (e.g. ant colonies pheromone gradient).

Presented requirements and proposed abstractions have led to the following *basic ontology*:

LSA A Live Semantic Annotation (LSA) is a structured, semantically founded, and continuously updated annotation reflecting some relevant information for the coordination of a SAPERE ecosystem. It plays the role of the molecule in the bio-chemical abstraction.

SHARED LSA-SPACE Represents the global chemical environment, composed by multiple (local) LSA-spaces: contains all the ecosystem's LSAs.

NODE A SAPERE node is any computational node of the ecosystem. It is the fundamental brick of the virtual topology. Each LSA and agent belongs to a node.

(LOCAL) LSA-SPACE The portion of the SAPERE shared LSA-space that belongs to a specific node is a single LSA-space. LSAs which are local to the node are stored in the space.

AGENT A piece of active software which runs on a SAPERE node. It manifest itself to the rest of the ecosystems through one or more LSAs, published on node's LSA-space.

BOND Similar to chemical bonds, it works as a direct, oriented, connection from a source LSA to a target one, letting the source to observe the target and be affected by it [Stevenson and Viroli, 2011].

ECO-LAW Works like a chemical law: if a a subset of (local) LSA-space's data fits reactants templates, then it *atomically* substitutes them with relative products. Allowed operation are: creation, modification and deletion of LSAs on local space and diffusion to neighbor nodes. Eco-laws are associated to a *rate* that influences its scheduling, so how often is it applied.

3.1.1 Architecture

The resulting architecture is a network of interconnected SAPERE nodes, each one with a (local) LSA-space and a set of SAPERE agents which interacts through it with the rest of the ecosystem.

An example of possible scenario is reported in Figure 1: four nodes, each one with a bunch of agents and a LSA-space, have been installed on two smartphones and two public displays; owned data are represented with the color of the owner agent.

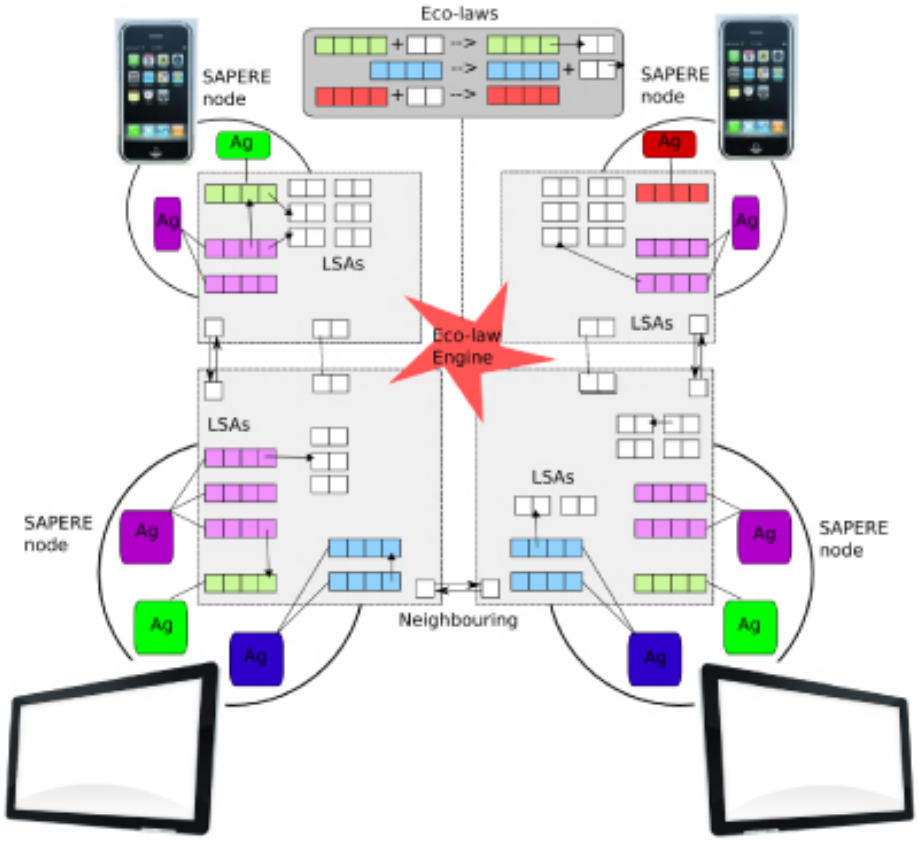

Figure 1: Example of SAPERE architecture [Viroli *et al.*, 2011]

3.1.2 Computational and Operational model

The abstract computational model states some basic ingredients of the proposed framework:

1. Structure and shape of an LSA
2. Role of contextualization
3. Structure and shape of the eco-laws
4. Agents behavior and interaction primitives

It is meant to enclose a set of principles that will be concretized in an operational model, which can have more than one declination, but will remain stick to those principles, providing coherence. In the following, concepts will be expressed in an informal way, simplifying the comprehension: in case of ambiguity the reader should refer to the formal model in [Viroli *et al.*, 2011].

LSAs

LSA is conceived as a set of multi-valued properties, namely a *semantic description*, identified by a unique system-wide identifier, so called *LSA*-

id. The adjective "semantic" implies possibile integration with Semantic Web technologies (see Chapter 2) in order to satisfy the *openness* requirement. Context information should be attached and formalized with the same structure in order to present a uniform and easy to understand content.

LSA-ids are used as a key for content retrieval and bonding support on *by reference* basis. The bonding should be realized keeping simplicity in mind, with the aim of support a weak and directional form of interaction between two LSAs.

A final aspect is the LSA-ownership. LSA's context should contain a reference to the owner, if it is not the case then the underlying system (the middleware) is responsible for its management. This is a tool for additional information reification.

Contextualization: synthetic properties and LSAs

Every contextual information should be reified as an LSA and injected in the LSA-space as part of its semantic description, namely *skeleton description*. Examples of relevant data are current location (mandatory and equal to the node on which it is actually stored), creation and last update time, creator identifier and so on; each one is called *synthetic property*.

Eco-laws

Eco-laws are modeled as chemical resembling rules, like $P_1 + P_2 + \ldots + P_n \xrightarrow{r} P'_1 + P'_2 + \ldots + P'_m (n, m \geqslant 0)$. It is composed by:

REACTANTS The left-hand side of the law. A set of P_i, namely *pattern*s.

PRODUCTS The right-hand side of the law. A set of P'_i, also called patterns.

RATE EXPRESSION which influences law's scheduling frequency.

A *chemical pattern*, which can be a reactant/reagent or product pattern, is defined as $x[F]$: the former (x) is a symbolic *pattern name*, which will be substituted with a LSA-id during the matching phase, while the latter is a filter which is applied to the LSA whose identifier matches the pattern name. When used in left-hand side a filter defines a template and restricts the number of eligible LSAs; whereas on the right-hand side it describes a manipulation of data.

Substantially an eco-law is a template that describes a set of reactions, which differ from it because all variables are instantiated, so defining a unique atomic modification of data. That instantiation is obtained by *matching: a semantic description* D *is said to match filter* F *by substitution* ϕ, *if it matches the application of* ϕ *to* F. Once retrieved a reaction it will be scheduled, according to the actual rate value for *application* in the local

LSA-space: matching data are removed from it and replaced with the one derived from products patterns.

Diffusion process is handled through eco-laws too, even preserving local behavior. In fact it is modeled with a change of the context's *location* property. As a consequence the middleware will arrange the transfer, so to reflect the stated scenario.

CTMC rates

Finally let's spend two words on scheduling rates . The correct abstraction to be adopted is the Continuous-time Markov Chains (CTMC) framework. The main reason is that this is a good, solid model, pervasive in nature, so realistic. Moreover a set of optimizations has been designed and tested over time, in order to support real-time execution. The simplest solution is implementing a scheduler that, once determined the first occurring reaction, waits for the predicted time to come and then triggers the update; if, in the meanwhile, LSA-space content changes, next action is rescheduled. Another approach consists in exploiting dependency-graphs; this way rescheduling is done only if actually necessary, but realizing this logic is really much harder.

Agents behaviour

SAPERE agents are defined as active autonomous entities, belonging to a SAPERE node from birth to death. Their existence is manifested in the (local) LSA-space through a non-empty set of LSA, owned and continuously updated, by them, in order to reflect current status. Other, non-owned, LSAs are perceivable thanks to direct or indirect bonding, but editing is not allowed. The agent who injected an information is the only allowed to destroy it.

3.2 MAPPING TO SEMANTIC FRAMEWORK

Now that a model has been formalized it is possible to analyze possible serializations into concrete languages, grounded by existings semantic web frameworks. The main idea is to translate each LSA into a RDF triples packet (see Section 2.1.1) and each eco-law into queries: SPARQL for the left-hand side and SPARUL for the right-hand one (see Section 2.1.4 and Section 2.1.5).

3.2.1 Live Semantic Annotations (LSA)

Given its structure the simplest way to serialize an LSA is asserting a set of RDF triples of the type $< i, p, v >$, where i is the LSA-id, p is a property name and v a property value. In order to be compliant with RDF specification i and p must be defined as URI, while the value (v) can be both a literal or a resource identifier, based on its nature. In fact it can be a string, an integer number, a date, another resource and,

Nested Descriptions

according to [Montagna *et al.*, 2012], even nested semantic descriptions: in this case a natural translation is a *blank node*, because the underlying meaning is the same.

Listing 45 reports an example of possible LSA and relative RDF triples packet.

Listing 45: LSA Serialization example [Montagna *et al.*, 2012]

```
# ------------------------------------------------
# SAPERE LSA EXAMPLE (part of original example)
# ------------------------------------------------
#
# lsa:exhibition1432 [
#   eco:type museum:exhibition;
#   eco:location "node34165@room131";
#   sos:request [syn:rate "1.0";
#   syn:prop museum:exh_request;
#   syn:syn_prop sos:request];
#   museum:poi_desc "michelangelo" "david" "sculpture"
#   "renaissance";
# ]
# ------------------------------------------------

@prefix ... # Namespaces declaration

lsa:exhibition1432 a sapere:LSA;
  eco:type museum:exhibition;
  eco:location "node34165@room131";
  sos:request [
    syn:rate "1.0";
    syn:prop museum:exh_request;
    syn:syn_prop sos:request .
  ];
  museum:poi_desc "michelangelo", "david", "sculpture",
    "renaissance".
```

3.2.2 Eco-laws

Once LSAs are turned into RDF, it is natural to try to consider existing languages to query and manipulate RDF stores: SPARQL is a good candidate for reactant patterns, in particular a `SELECT` clause, while a sequence of SPARUL statements, `INSERT` and `DELETE`, can play the role of product patterns updating LSA-space content. Variable terms can be mapped into variable names and constraints on their value can be expressed in terms of `FILTER`s and `BIND`ings in the `WHERE` clause.

Listing 46 reports an example of eco-law serialization. The form <vname>! represents a variable, in a SPARUL statement whose value has been determined in the matching phase.

Listing 46: Eco-law Serialization example [Viroli *et al.*, 2011]

```
# ---------------------------------------------
# ECO-LAW EXAMPLE
# ---------------------------------------------
# ?TARGET:[?PROP has ?VALUES] +
# ?SRC:[bond:request has (?BONDREQ); ?B has-not (?TARGET)] +
# ?BONDREQ:[sapere:type has (bond:request_pv);
#        bond:bond_prop=(?B);
#            bond:target_prop=(?PROP);
#            bond:target_value=?VALUES]
# ---->
# ?SRC:[?B has (?TARGET)] + ?BONDREQ + ?TARGET
# ---------------------------------------------

# Left-hand side
SELECT DISTINCT *
WHERE{
  ?SRC bond:request ?BONDREQ .
  FILTER NOT EXISTS {
    ?SRC ?B ?TARGET
  } FILTER NOT EXISTS {
    ?TARGET ?PROP ?o .
    FILTER NOT EXISTS {
      ?BONDREQ bond:target_value ?o
    }
  }
  ?BONDREQ sapere:type bond:request_pv .
  ?BONDREQ bond:bond_prop ?B .
  FILTER NOT EXISTS {
    ?BONDREQ bond:bond_prop ?o.
    FILTER (?o != ?B)
  }
  ?BONDREQ bond:target_prop ?PROP .
  FILTER NOT EXISTS {
    ?BONDREQ bond:target_prop ?o.
    FILTER (?o != ?PROP)
  }
}

# Right-hand side
INSERT DATA {!SRC !B !TARGET .}
```

For details about adopted syntax and translation formalization check [Stevenson and Viroli, 2011].

4 SEMANTIC WEB SAPERE

CONTENTS

This chapter is dedicated to the presentation of the middleware that has been developed.

The resulting piece of software represents an innovation from previous works (see [Desanti, 2011]) because of two main features:

1. The adoption of Semantic Web Technologies, already described in Chapter 2, which represents a breakpoint from the past, from some model and API realization aspects to effective LSA-space implementation;

2. The choice of following Open Service Gateway initiative (OSGi) specification and *component-oriented* paradigm, in order to promote modularization, extension and maintainability.

Main concepts expressed in the SAPERE model (see Chapter 3) have been mapped into a set of OSGi services and an high-level API, which has been consequently implemented in order to fulfill requirements; as long as defined interfaces are not modified single service realization can be substituted providing different *bundles* to the *container*.

Development process

The whole development has taken advantage of a set of tools, meant to standardize a process, ease team collaboration and assure high-quality software; first of all *Apache Maven*[1]. It allowed to define a modular workflow: the middleware has been divided in sub-projects – then deployed as OSGi bundles – each one with its documentation updated at every build. Plugins such as Checktyle, PMD, FindBugs, Cobertura,

1 see http://maven.apache.org

Pax-Exam and Surefire have been included in the process aiming to implement test-driven development – also useful for future extensions – common pitfalls avoidance and documentation completeness.

After the middleware was fully developed some performance tests have been run, as reported in Chapter 5, trying to reach a deeper understanding of the platform capabilities.

4.1 REQUIREMENTS

Semantic Web SAPERE is meant to address pervasive services ecosystems problems, exploiting semantic power to enhance services quality.

The abstract architecture of the middleware has been fixed once the SAPERE project has been chosen as reference. The main, and most complex, component involved in the semantic-enabling revolution is the LSA-space, followed by the formalization of information units, the LSAs. In fact the main requirements can be summarized as follows:

- Use RDF for information representation and, consequently, enact the translation of each LSA in a set of RDF statements and viceversa;
- Let the application logic provide a description of data that will share during its lifecycle, allowing their understanding by the rest of the ecosystem;
- Allow eco-laws to exploit these enriched information for ecosystem regulation;
- Provide a set of extension points for further middleware refinements;
- Support the diffusion mechanism: once defined a virtual topology of SAPERE nodes, LSAs could be relocated from a node to another.

Topology creation and maintenance is out of the scope of this work, it is considered to be provided through a configuration file and considered to be static.

4.2 LOGIC ARCHITECTURE

A logic architecture has been defined as a result of the expressed requirements and consequently mapped into an API, modeled in such a way that multiple implementations can be provided. Section 4.3 will describe one of the possible realizations, focused on Jena and Pellet libraries as enabling technologies, but nothing prevents to choose different ones.

In order to ensure a better understanding of the architecture, it will be presented in a layered way, from the highest level of abstraction to the lowest one:

1. SAPERE nodes and topology structure
2. The SAPERE node sub-system
3. LSA-space components

An overall perspective is given in Figure 2.

4.2.1 The ecosystem as a network of nodes

At the highest level we can think about a Self-Aware Pervasive Services Ecosystem as a multitude of autonomous nodes, each one running on a device of any kind, from a computer, to a smartphone, a sensor platform or a display.

Neighborhood

What these devices must have in common is the ability to communicate with other ones, in order to establish a network and enact information sharing. In particular, in a pervasive context, it is important to define a topology, which means that a node should know both which other nodes it can *speak* to, and its interlocutor approximative distance: in this way they can understand if a proximity relationship subsists, creating the concept of *neighborhood*.

As stated before, topology definition is not the main concern of this work, but it is important that the resulting middleware can provide a basic communication mechanism, on which build more complex behaviors.

4.2.2 Inside the SAPERE node

Let's now go deeper. A SAPERE node is a whole new system, composed by a set of entities each one modeled to satisfy a particular requirement. The central entity is the LSA-space, whose responsibility is storing all data, formalized as LSAs and serialized in RDF[2]. Given its importance next sub-section will describe its internal structure; by now we focus on the interface and the other parts of the sub-system.

Agent's primitives

According to the SAPERE model the space is a shared-environment for some agents, running locally and able of publishing LSAs about their state and willings. Coordination is obtained letting other agents retrieve and observe what other have published. This goal is reached by defining an interface between the agents and the space that provides the following primitives:

INJECT inserts a new LSA in the LSA-space;

2 see sub-sections 3.1.2 and 3.2.1

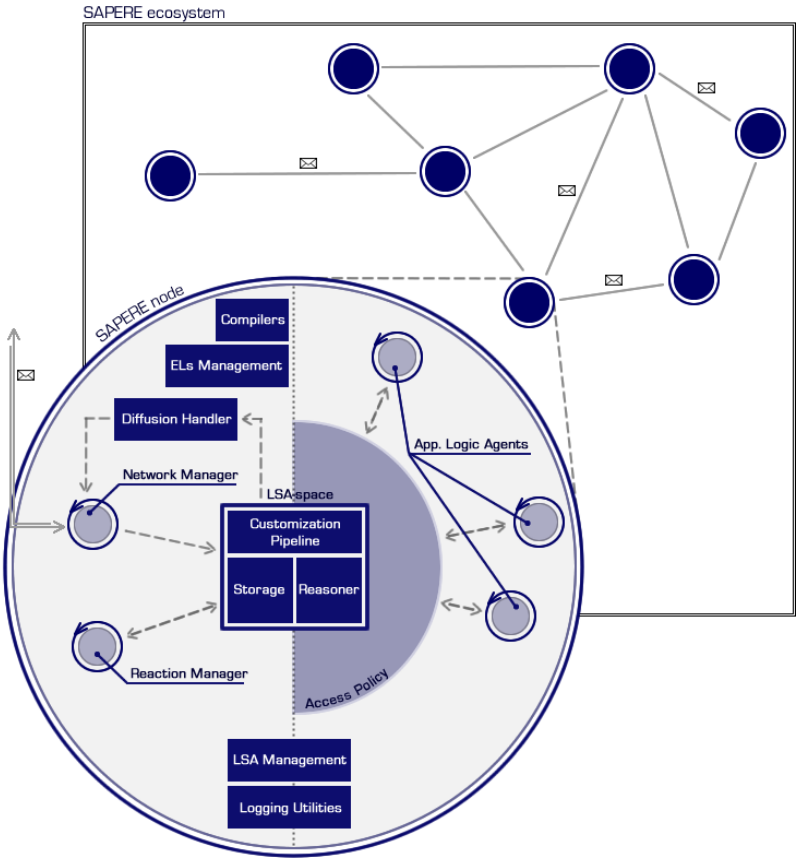

Figure 2: Semantic Web SAPERE: Logic Architecture

UPDATE modifies previously injected LSA content according to the provided one;

REMOVE removes a previously injected LSA from the LSA-space;

READ Given a known LSA-id, find the whole LSA and return it to the agent;

OBSERVE Given a known LSA-id, instruct the system to notify the agent of every modification of the associated content.

Other primitives could be adopted, even if not declared in the model:

IGNORE with the purpose of stating that an agent is no more interested in observing an LSA;

LOADONTOLOGY , in order to address the need of providing data description – in OWL format – so that reasoning can take place. Given the fact that a lot of ontologies are available over internet, their Uniform Resource Locator (URL) should be passed and the content must be managed according to the specific implementation.

System and User Agents

According to previous considerations, two different abstractions of *agent* concept are required. In fact the ones that compose the middleware need to run operations that should not be allowed to *normal* (application logic) agents: modification of data that are not owned by the agent or bonded to it (see Section 3.1), is an example of a potential security leak that could be raised if no regulation is imposed. That's why *System Agents* has been modeled as full-fledged entities, generally used to improve middleware capabilities, while *User Agents* has been provided to the final user in order to build a pervasive services ecosystem, without risking model constraints violations. The component that is demanded to model constraints enforcement is called `AccessPolicy` and is intended as a *mediator* of all the interactions between User Agents and the LSA-space.

LSA-space Access Policy

Network Manager

Recalling what has been told in the previous sub-section, a `NetworkManager` is required in order to provide a mechanism for relocating information from a node to others. In particular a system agent should be designed for listening to the network and injecting what has been received in the LSA-space, while observing what happens in it and determining if an LSA is eligible for *diffusion*. Space observation has been modeled as a passive component, namely the `DiffusionHandler`, which is responsible for Network Manager notification whenever an information is marked as *outgoing*[3]. Another system agent is the `ReactionManager` , responsible for enabling the Eco-laws

Reaction Manager

3 The mark is intended to be expressed via a dedicated LSA's property, called "sapere:location", whose value is "sapere:local" when the information should not be relocated and a neighbor identifier (the destination) when should be moved

execution, as mentioned in Section 4.4.1. In order to accomplish its task two things are required: (1) a way to express and pass eco-laws and (2) a dedicated access to the LSA-space, such as law match and application can be run. These two functionalities has been mapped in two other primitives:

MATCH which tries to bind variable terms to values stored in the space

APPLY which modifies the space content according to what is stated by the law and the bindings that have been retrieved by the previous primitive

The `ReactionManager`'s behavior is cyclic and consists of (1) try to `match` each known eco-law and choose the one that should be scheduled first – according to the actual value of the *Markovian* Rate – then (2) wait for scheduling time to come and (3) `apply`, the law, as soon as it is reached. If the LSA-space status changes before the application phase then match condition should be verified before proceeding.

Other middleware services

Other than LSA-space and agents, a set of services/components has been defined in order to help the latters pursuing their goals; they are listed in the following.

LOGGING UTILITIES are demanded to handle agent's *standard output* and error[4]. This service should be used each time an agent would print a message for the user to be read: this way, according to different levels of detail, single messages will be routed to console and/or log files simplifying monitoring, debug and final deployment.

LSAS AND ECO-LAWS MANAGEMENT facilities are meant to be used respectively for the creation of LSAs and Eco-laws, with reference to the models presented in Section 3.1.

An LSA is composed by an LSA-id, which globally identifies it, and a Semantic Description (also known as Content) which is organized as a set of multi-valued properties, which can refer to URIs, LSA-ids, strings – representing simple text, numbers, date and so on – or, recursively, other Semantic Descriptions. LSA-id and Properties are mapped to URIs, while the rest is mapped to tagged strings, meaning that they are serialized and decorated with metadata that specifies the original type; in order to simplify the management each concept has been mapped to an interface, so to a Java type (see Figure 3).

The same approach has been used for modeling Eco-laws (see Figure 4): they are composed by a set of `Reactants` and `Products` which are specific types of `ChemicalPatterns`, described as a `PatternName`

4 Standard input has been considered irrelevant by now

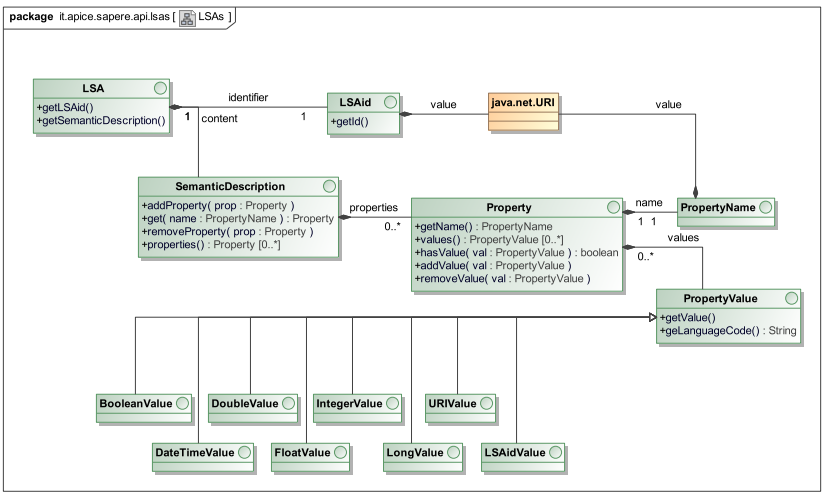

Figure 3: LSA model

plus a set of `Filters`. Filters[5] can be *content-related* such as `clones` and `extends` or *property-related*, such as `assign`, `has`, `has-not` and `matches`.

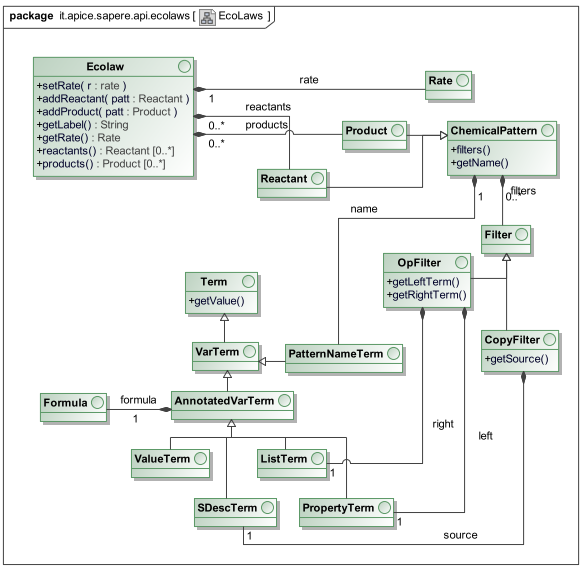

Figure 4: Eco-laws model

`LSAFactory` and `EcolawFactory` has been provided as generators, while `LSAParser` has been defined in order to extract LSAs from a textual representation in RDF/XML, Turtle/N3 or N-Triples language (see Section 2.1).

COMPILERS have been defined, encapsulating the translation from SAPERE to RDF/SPARQL-SPARUL: the former is still under development, while Semantic Web languages are almost standardized. *Com-*

5 More details on types meaning can be gathered in [Viroli *et al.*, 2012; Zambonelli *et al.*, 2011]

piled LSAs and eco-laws rely on those standards stability and let node services be reused over time even if some SAPERE concept changes. This means paying a little overhead (see Section 5.2.2) for improving maintainability: System agents, as part of the middleware, will use compiled versions, while User Agents should not, in order to be compliant with SAPERE requirements.

CompiledLSA

The compilation process should generate a `CompiledLSA` by mapping each LSA property into a RDF predicate and listing property values as objects. The subject of each statement must be the LSA-id (once converted to URI) or a blank-node id, when serializing a nested semantic description. Moreover the creation of empty `CompiledLSA`s and some basic modifications should be allowed, in order to support information update and management at middleware level: just think about the `sapere:location` property in diffusion context. The result is summarized in Figure 5

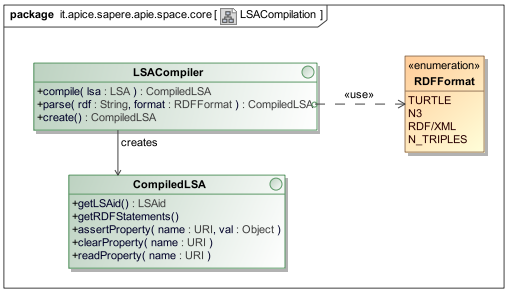

Figure 5: LSAs compilation

CompiledEcolaw and relatives

Supporting eco-laws is more complex. In fact each reagent should compose the `WHERE` clause of a `SELECT` query and each product should be translated in a SPARUL statement. Therefore the variables assigned during the match phase must be reused in the application step, in order to reduce execution time. That's why eco-law compilation has been defined in order to produce a *template* that will be concretized as soon as bindings are available: the structure of the query is still the same, only variable values change. Figure 6 shows the result.

The presented services are provided to each agent with reference to their privilege (system or user); the resulting architecture is detailed in Figure 7, with particular attention to the interaction between components.

4.2.3 The LSA-space

Finally let's look at how the LSA-space is structured.

Storage

The most important component is the one that provides *Storage facility*. This facility will handle CRUD[6] primitives, namely `inject`,

6 CRUD is an acronym which means Create-Read-Update-Delete. It is generally used in order to indicate basic operation that can be executed on a DB.

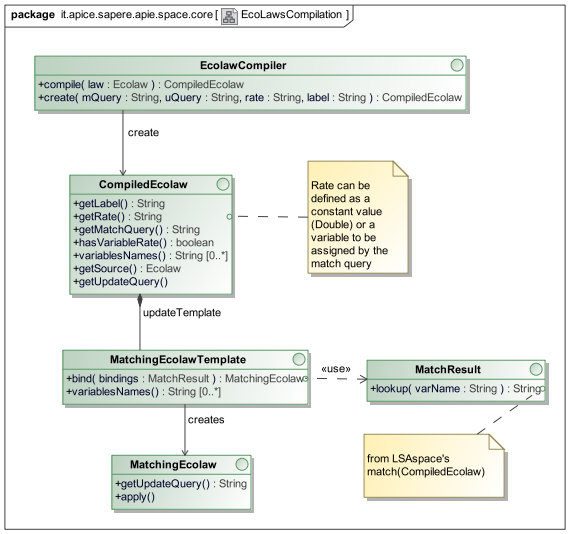

Figure 6: Eco-laws compilation

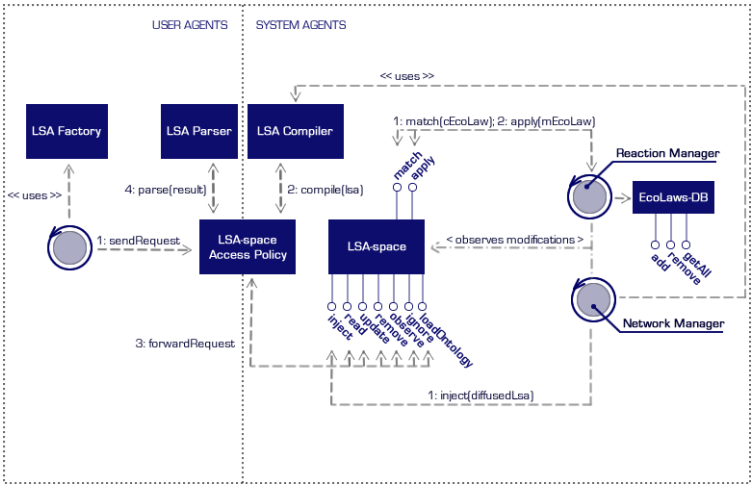

Figure 7: Interaction between agents and LSA-space

`read`, `update`, `remove`. In order to provide thread-safety a *lock* should be acquired before executing each operation, because of the shared-environment assumption. According to what has been previously presented, input is passed already in an RDF format – thanks to the `CompiledLSA.getStatements()` operation – and, in case of read-only primitives, the LSA-id is already provided in URI form. Before storing or retrieving data some requirements must be checked and satisfied. Some constraints – e.g. an LSA cannot be injected twice – are implemented but, according to the idea of allowing future extensions, a *customization pipeline* has been placed before the actual execution of each primitive. This mechanism, modeled through the concept of `CustomStrategyPipeline`, is nothing but an ordered list of steps (`CustomStrategyPipelineStep`), each one able of decorating/modifying parameters and, if some pre-condition is violated, preventing prim-

Customization Pipeline

itive execution. *Synthetic Properties* management is an example of a situation that could be handled this way: just before injecting an LSA the manager can append a context, or `sapere:lastModified` property could be adjusted each time an update is required. Since some middleware services should be reactive to LSA-space changes – e.g. Reaction and Network Managers, see Figure 7 – another important feature provided is space observation. It realizes a *pattern observer* and provide notification each time an operation successfully completes. The `observe` primitive can rely on the same mechanism: when requested, an observer should be notified each time a relevant event is raised; obviously the relevance of perception is different between middleware and agents.

Space observation

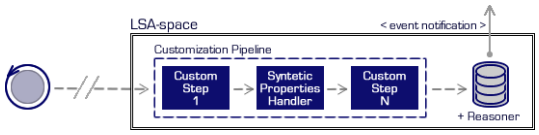

Figure 8: Inside the LSA-space

Reasoning facility

The only remaining component to be presented is the *reasoner*. Its interface and how it interacts with the Storage is specific to the chosen implementation, so it will be explained in the following section. What can be said by now is that, in some way, it should be aware of data modification in order to update the inferences and provide coherent information: using Jena and Pellet will give it almost for free.

4.3 DEVELOPED SYSTEM

This section focuses on design choices and description of the final middleware.

LSA-space: Jena + Pellet

Let's start from the adoption of Jena and Pellet libraries. The former is used as storage facility: the `Model` interface – presented in Section 2.2.1 – allows the creation of a resource for each LSA and its removal each time a `remove` is requested; the update operation is the composition of the previous two.

According to the Jena documentation, RDF resources are handled with the *flyweight pattern*, which means that memory fingerprint is reduced because less objects are created, and also the performances of these features are improved. This feature is exploited in the `read` primitive.

The actual performances, and the linkage to Pellet reasoner, are merely based on the `Models` and `GraphStore` created during the LSA-space initialization: a parameter (`ReasonerLevel`) has been exposed in

order to determine if the reasoner should be used and at what level of inference. Listing 47 shows the real implementation: if `OWL_DL` level is chosen then an inference model – the one which stores all produced inferences – is instantiated and linked to a `DefaultModel`; this pattern has been taken from [Hebler *et al.*, 2009].

Listing 47: Storage/Reasoning initialization

```
Model initRDFGraphModel(final ReasoningLevel level) {
  if (level.equals(ReasoningLevel.OWL_DL)) {
    final Reasoner reasoner = PelletReasonerFactory
        .theInstance().create();
    final Model infModel = ModelFactory.createInfModel(
        reasoner,
        ModelFactory.createDefaultModel());
    setInfGraph(infModel.getGraph());
    return ModelFactory.createOntologyModel(
        OntModelSpec.OWL_DL_MEM,
        infModel);
  } else if (level.equals(ReasoningLevel.RDFS_INF)) {
    return ModelFactory
        .createOntologyModel(
          OntModelSpec.OWL_DL_MEM_RDFS_INF);
  } else if (level.equals(ReasoningLevel.NONE)) {
    return ModelFactory.createOntologyModel(
      OntModelSpec.OWL_DL_MEM);
  } else {
    return ModelFactory.createDefaultModel();
  }
}
```

Thread-safety

A reference to that model is kept because of the thread-safety policy: `read` and `match` primitives use it to trigger the reasoner before retrieving information. Storage access, and reasoning phase, are protected with a *read-write lock* which regulates critical sections, but replaced with a mutex when Pellet is active[7].

In order to speed up `match` and `apply` execution, some optimizations have been introduced. Studying the idea beyond eco-laws mechanism and how queries are managed by Apache Jena a couple of `HashMaps` has been used to provide *caching*. In fact nor the left side of an eco-law neither the right one changes; what is different between two applications is the value of the variables that have been assigned during the match phase. Jena 2.7.0-incubating is able to pre-initialize bindings in a query object, enabling us to cache those bindings and pass them when executing SPARUL statements. Therefore parsing overhead can be avoided

7 Letting Jena + Pellet operations be serialized assures a correct execution, because Pellet is not meant to be used with Jena in multi-threaded scenarios. An alternative could be exploiting DIG interface, but it introduces the overhead of a HTTP protocol stack

after the first time by keeping track of `Query` and `UpdateRequest` objects once created.

Synthetic Properties and Security Policies

About middleware policies, now we discuss the synthetic properties management. Continuing from what has been told in Section 4.2.2, a step – the only one – is dedicated to it in the `CustomStrategyPipeline`. It is responsible for registering timings and location data, but is not able of checking if the requestor is allowed to run the operation. This task is demanded to the `LSAspaceAccessPolicy` component, which is aware of the agent identity: in fact, after calling the `Compiler`, it specifies the owner, during injection, and filters forbidden operations. The exclusive access to synthetic properties is ensured through an additional invariant on the `LSA` model: this way they can only be edited after the compilation phase, which is not accessible to user agents.

Finally let's spend two words on SPARQL/SPARUL expressive power. A feature that has not been presented until now is the extensibility of those languages: thanks to Jena's `Function` abstraction it is possible to add new functionalities, just extending a class and registering it into a `FunctionRegistry`, in association with a URI that will be used to refer them in queries. A new service has been defined in order to support this, namely the `CustomFunctRegistry`: it simply wraps the supplied functionality in order to publish it in the OSGi context.

Reaction Manager

The Reaction Manager business logic has been split in two parts: in fact what concerns the scheduling policy has been encapsulated in a dedicated component – its interface is reported in Listing 48 – which receives notifications about possible matches and decides the next ecolaw to be applied, but also monitors space changes and aborts last decision if found bindings are no more valid.

Listing 48: Reactions Scheduler

```
1  interface ReactionsScheduler extends ReactionManagerObserver
       {
2
3    long ecolawMatched(SchedulableMatchResult match, long
         schedulingTime);
4
5    Entry<MatchingEcolaw, Long> next();
6
7    void checkDependencies(SpaceEvent ev, MatchingEcolaw law)
8        throws AbortException;
9
10   SchedulableMatchResult eval(MatchResult mResult) throws
         SAPEREException;
11
12   SchedulableMatchResult[] eval(MatchResult[] mResults)
```

```
13      throws SAPEREException;
14  }
```

This choice has been taken because the current approach has been implemented in order to optimize execution time, but is prone to *live-locks*: if threads interleaving would let agents modify the LSA-space before a scheduled eco-law is applied then no reaction would occur. In this way future works can deepen scheduling policy optimization – by considering *dependency graphs* implementation and rate adjustment strategies in order to avoid this kind of situation – and plug it in without creating a reaction manager from scratch. Listing 48 shows that an `eval` method has been designed to evaluate the proposed match and return a `SchedulableMatchResult` which contains the scheduling time, calculated according to CTMC no-memory process[8], whose mean value is the actual eco-law's rate. `checkDependencies` operation is the one responsible for space events analysis and abortion handling. Last but not least, `ecolawMatched` allows the scheduling logic to be notified whenever new bindings have been found and to build some optimization mechanism; its return value is meant to be used to modify the update apply time, in fact `ReactionManager` main cycle only cares about the next coming rule.

Network Manager

Network Manager has been designed to offer nodes intercommunication through TCP/IP sockets, but with a little modification of the `register` method (see Listing 49) it could be able of handling any means of transport.

Listing 49: Network Manager

```
1  interface NetworkManager {
2
3    void diffuse(Object to, NodeMessage msg);
4
5    boolean register(String id, InetSocketAddress addr)
6        throws SAPEREException;
7
8    void loadTable(File config);
```

Actual implementation opens a server socket (onto a customizable port) and listens to incoming connections. The `diffuse` operation plays the role of the deliverer: connects to the server on the other node (specified by the `to` parameter) and sends the diffused LSA to it. The diffusion is triggered by the `DiffusionHandler`, which observes the LSA-space and causes a diffusion according to the `sapere:location` property value: if it equals to `sapere:local` then no relocation occurs, otherwise it supplies the identifier of the node to which the LSA must be

8 time = -Math.log(rng.nextDouble()) / rate + currentTime

sent (the `to` parameter). On the server side, at each time message is received, the data are extracted and published in the space (using the `inject` primitive).

The adopted solution is naive but effective. Supporting a dynamic environment and multiple protocols will be a future work.

4.3.1 OSGi bundles

Final architecture has been deployed as a set of bundles. Figure 9 and the following list, show all of them along with their dependencies:

SEMANTICWEBSAPERE-API Contains all the interfaces and enumerations needed to model semanticwebsapere middleware. In practice it contains the definition of the logic architecture presented before (see Section 4.2). No service is published.

SEMANTICWEBSAPERE-REQUIREMENTS Wraps Jena libraries, and relatives ones, in order to let them available in the context. No service is published.

SEMANTICWEBSAPERE-PELLET Wraps Pellet libraries in order to let them be imported in LSA-space component. No service is published.

SEMANTICWEBSAPERE-RDFMODEL Implements APIs for what concerns LSAs and eco-law models: data entities, factories and parsers. The latters are published as OSGi services.

SEMANTICWEBSAPERE-RDFSPACE Implements the LSA-space abstraction, exploiting Jena and Pellet facilities; it also handles LSA (de)compilation. The space, compilers and custom-functions registry are published as OSGi services.

SEMANTICWEBSAPERE-NODE Provides the concept of SAPERE node as computational node in which a set of agents run locally interacting through an LSA-space.

As shown in Figure 9, each application is packed in a bundle and declared as dependent from `semanticwebsapere-node`; for transitivity all other bundles are imported.

Listing 50: SAPEREAgentsFactory interface

```
1 interface SAPEREAgentsFactory {
2
3   SAPEREAgent createAgent(String agentLocalId, final
        SAPEREAgentSpec spec)
4       throws SAPEREException;
5
6   SAPEREAgent getAgent(String agentLocalId) throws
        SAPEREException;
```

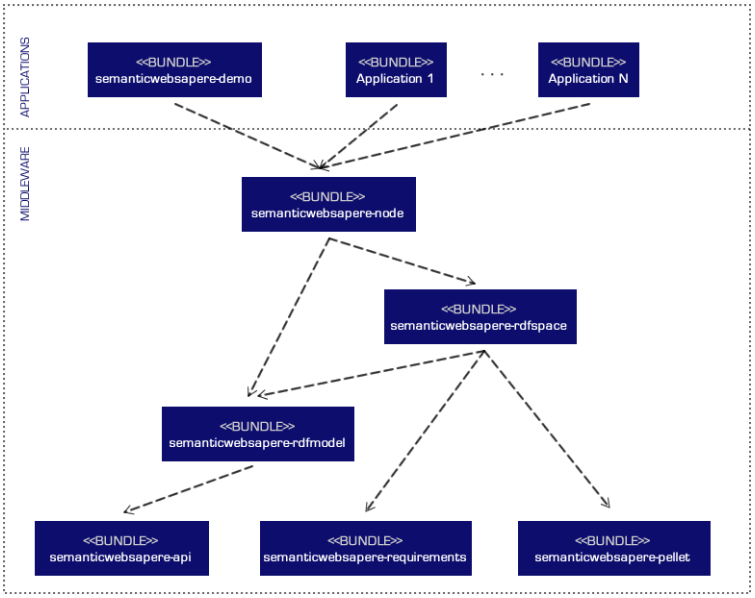

Figure 9: OSGi bundle and dependencies

```
7
8   SAPEREAgent createSysAgent(String agentLocalId,
9       final SAPERESysAgentSpec spec) throws SAPEREException;
10
11  void killAll();
12
13 }
14
15 /** User Agent Specification. */
16 interface SAPEREAgentSpec {
17
18  void behaviour(LSAFactory factory, LSAParser parser,
        LSAspace space,
19      LogUtils out, SAPEREAgent me) throws Exception;
20
21 }
22
23 /** System Agent Specification. */
24 interface SAPERESysAgentSpec {
25
26  void behaviour(NodeServices services, LogUtils out,
        SAPEREAgent me)
27      throws Exception;
28
29 }
30
31 interface SAPEREAgent {
32
```

```
33   void spawn();
34
35   void kill();
36
37   URI getAgentURI();
38
39   String getLocalAgentId();
40
41   boolean isRunning();
42
43 }
```

Agents Factory

In order to simplify the deployment of an agent to the middleware a `SAPEREAgentsFactory` has been defined (see Listing 50). An agent can be created providing a *specification* to the factory method – a subtype of `SAPEREAgentSpec` or `SAPERESysAgentSpec` – with a locally-valid identifier; after that the agent can be spawn. This way there is no need to manually handle services retrieval: this is automatically done by `SAPERENodeActivator`[9], so the factory can pass them to each agent.

Since this is a prototype middleware, the System agent creation is exposed to the public despite in a real context it should be hidden.

The whole node has been tested on Apache Felix platform. It has been chosen because it offered the best debug tools; however according to OSGi specification each OSGi container should be able of running them. A couple of examples have been reported in Section 4.5 and Section 5.1.1.

4.4 MIDDLEWARE USAGE

In this section all the aspects of configuring and launching a SAPERE node are presented and explained, in order to enable future users to exploit its features. Since the platform has been tested on Apache Felix all information reported are relative to it, but other containers could be used in similar way.

First of all, the bundles presented as part of the middleware in the previous section – `semanticwebsapere-api`, `semanticwebsapere-rdfmodel`, `semanticwebsapere-requirements`, `semanticwebsapere-pellet`, `semanticwebsapere-rdfspace` and `semanticwebsapere-node` – should be installed and started in the container. Once started, even the application-specific bundle can be launched; in Felix this can be all done through the `config.properties`, specifying the `felix.auto.start` property.

During the container launch it is possible to set some options provided by the middleware in order to tune it according to what is needed.

9 An activator is what allows to publish and retrieve services in OSGi context.

This can also be done through configuration file, otherwise it can be passed when starting the Java Virtual Machine:

```
java  -D<prop-name>=<prop-value> ... -jar bin/felix.jar
```

Table 1 reports all possible key-value pairs and describes their implications.

Table 1: Middleware Options

Key	Default	Description
sapere.log.console.level	INFO	Level of CONSOLE log, according to Log4j specifications
sapere.log.file.level	INFO	Level of FILE log, according to Log4j specifications
node-uri	<auto>	Sets a URI for the node, otherwise it is automatically generated
sapere.diffusion.port	20021	The port on which the Network Manager listens to incoming connections
sapere.diffusion.config	<none>	Path to a Properties file (XML) which lists all known neighbors
sapere.space.reasoner	none	Selects one of the different levels of reasoning: (none, rdfs, owl-dl)
sapere.space.optimization	true	Enables/Disables the query parsing optimization

The next section will deepen how to define an application on this middleware. In other words how to deploy the application bundle.

4.4.1 Modelling an ecosystem

In order to let the application run on this version of the SAPERE node a `BundleActivator` must be defined. This way the `SAPEREAgentsFactory` service can be retrieved and agents can be created and spawn.

Listing 51: How to spawn an agent on SAPERE node

```
1 void start(BundleContext context) {
2   ServiceReference<SAPEREAgentsFactory> factRef =
3   context.getServiceReference(SAPEREAgentsFactory.class);
4   if (factRef != null) {
5     final SAPEREAgentsFactory agentsFact = context.getService
        (factRef);
```

```
6
7       agentsFactory.createAgent("hello_agent",
8           new HelloAgentSpec()).spawn();
9
10      context.ungetService(factRef);
11    }
12  }
```

Defining SAPERE agents

As already said before, agents can be modeled through the `SAPEREAgentSpec` type. An example – an enhanced version of the classic hello world – is reported in Listing 52. Please note that all the needed services are provided as parameters of the `behaviour` method.

Listing 52: A simple Hello World agent

```
1  class HelloAgentSpec implements SAPEREAgentSpec {
2
3    @Override
4    public void behaviour(final LSAFactory factory, final
         LSAParser parser,
5        final LSAspace space, final LogUtils out, final
             SAPEREAgent me)
6        throws Exception {
7
8        // Print a message
9        me.log("Hello World");
10
11       // Create and populate a LSA
12       LSA lsa = factory.createLSA();
13       lsa.getSemanticDescription()
14         .addProperty(
15           factory.createProperty(
16             URI.create("http://www.example.org#name"),
17             factory.createPropertyValue(me.getLocalAgentId())
                   );
18
19       // Inject a LSA
20       space.inject(lsa);
21   }
22 }
```

Defining Eco-laws

There are two solutions in order to declare some eco-laws to be executed on a node. The first one consists in defining a System Agent,

which has obviously access to all the services, running as an initialization service, then start the rest of the system. The second choice is to directly obtain a reference to `EcolawCompiler` and `ReactionManager` services, then use it directly from the `BundleActivator`.

Since eco-laws language is not yet completed, only a part of the compilation process has been implemented[10]. For this reason the `EcolawCompiler` has a `create` operation, which allows the ecosystem designer to specify rules directly in SPARQL and SPARUL syntax.

Listing 53 shows how to realize the second approach. In order to use the former the user should just ignore the OSGi-related part: services will be retrieved and provided by the platform.

Listing 53: How to define eco-laws

```
void start(BundleContext context) {
  final ServiceReference<ReactionManager> rref = context
        .getServiceReference(ReactionManager.class);
    final ServiceReference<EcolawCompiler> cref = context
        .getServiceReference(EcolawCompiler.class);
    if (rref != null && cref != null) {
      final ReactionManager mng = context.getService(rref);
      final EcolawCompiler cmp = context.getService(cref);

      mng.addEcolaw(cmp.create(getMatchQuery(),
          getUpdateQuery(),
          getRate()));

      context.ungetService(rref);
    }
}

String getMatchQuery() {
  final StringBuilder builder = new StringBuilder();
  builder.append("PREFIX ex: <http://www.example.org/profile
      #> ");
  builder.append("PREFIX xsd: <http://www.w3.org/2001/
      XMLSchema#> ");

  builder.append("SELECT DISTINCT * WHERE { ");
  builder.append("?lsa ex:prop ?value . ");

  return builder.append("}").toString();
}

String getUpdateQuery() {
```

10 Only reactants to SPARQL translation is supported, but not updated to handle nested semantic descriptions

```
29    final StringBuilder builder = new StringBuilder();
30    builder.append("PREFIX ex: <http://www.example.org/profile
          #> ");
31    builder.append("PREFIX xsd: <http://www.w3.org/2001/
          XMLSchema#> ");
32
33    builder.append("MODIFY DELETE { !lsa ex:prop !value . } ");
34    builder.append("INSERT { !lsa ex:prop ?newval . } ");
35    builder.append("WHERE { ");
36    builder.append("BIND(( xsd:integer(!value) + 1 ) AS ?newval
          ) }");
37
38    return builder.toString();
39  }
40
41  String getRate() {
42    return "1.0";
43  }
```

Defining a topology

The `sapere.diffusion.config` property, which should be passed when the OSGi container is launched, is meant to provide a network topology to the node. In particular its value must be a file path and the file must list all the neighbors' s names and TCP/IP address; if no file is specified then the node is considered to be isolated.

Listing 54 shows an example of a neighbors list: the syntax used is the one specified for Java's `Properties` XML serialization: each entry key is interpreted as neighbor name, while the value is the address.

Listing 54: Topology definition

```
1  <?xml version="1.0" encoding="utf-8" standalone="no"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd
       ">
3  <properties>
4  <comment>Diffusion configuration</comment>
5  <entry key="nodeB">nodeB.example.org:20021</entry>
6  </properties>
```

In order to actually establish a connection between nodes, provided URL have to be resolvable or, if an IP is specified, it has to be reachable by the local host. The neighbor's `NetworkManager` is supposed to wait for incoming connections on `20021` , which is the default port. If more than a node is running on the same host then the port must be unique for each one, otherwise a `BindException` is raised; `sapere.diffusion.port` is meant to be used for this purpose (see Table 1).

4.5 A DEMO SCENARIO

As conclusion of this chapter, let's see a demo scenario at work.

This is the first of two cases and is mostly focused on exploiting semantics in a SAPERE node; the next one will show a distributed situation and will be used as test for performance profiling (see Section 5.1.1).

Description

The environment is represented by a room, where a SAPERE node is installed. It keeps track of people moving and eventually approaching to one of two displays. Whenever a person comes next to a display, it shows a welcome message.

People and displays are represented with dedicated agents: the former type shares its name, and updates its current location every time a motion is detected, while the latter observes its owned LSA, once published with its position in the room.

Coordination is obtained defining two eco-laws, whose application modifies the involved display LSA, raising a space event that triggers it:

NEAR Tries to find a person which has become closer to a display. It checks if their distance is less than an arbitrary range – 7.0 meters – and if that person was not already classified as "next-to".

FAR It is dual to the previous one. Checks if a person considered in range has moved out of range.

The rate that has been assigned to both rules is 1.0, which means that they will be scheduled on an average of one time per second.

Some screenshots have been taken and reported in Figure 10. In initial state (top-left screenshot), Bob, Alice and John are in the room and the two displays are off, because no one is close enough (displays range has been marked with a darker circle around the device). Whenever someone becomes in range the `NEAR` eco-law is executed and a message is displayed as consequence of a reaction chain; as shown in the top-right picture that message is a string like "`Welcome {<person-name>}`". As soon as people go away their name is removed from the welcome string and, if no one is left, the display automatically switches off thanks to the `FAR` eco-law.

4.5.1 Realization details

Now that the scenario has been explained, it's time to report how each entity has been realized.

PersonAgent

Let's start with the definition of `PersonAgents` and the information they publish. A person has been modeled as attached to a fake sensor, from which it can retrieve its current location by polling[11]; the LSA it

11 The actual value is obtained by the current position of the related dot on the JFrame. People can be moved by drag and drop

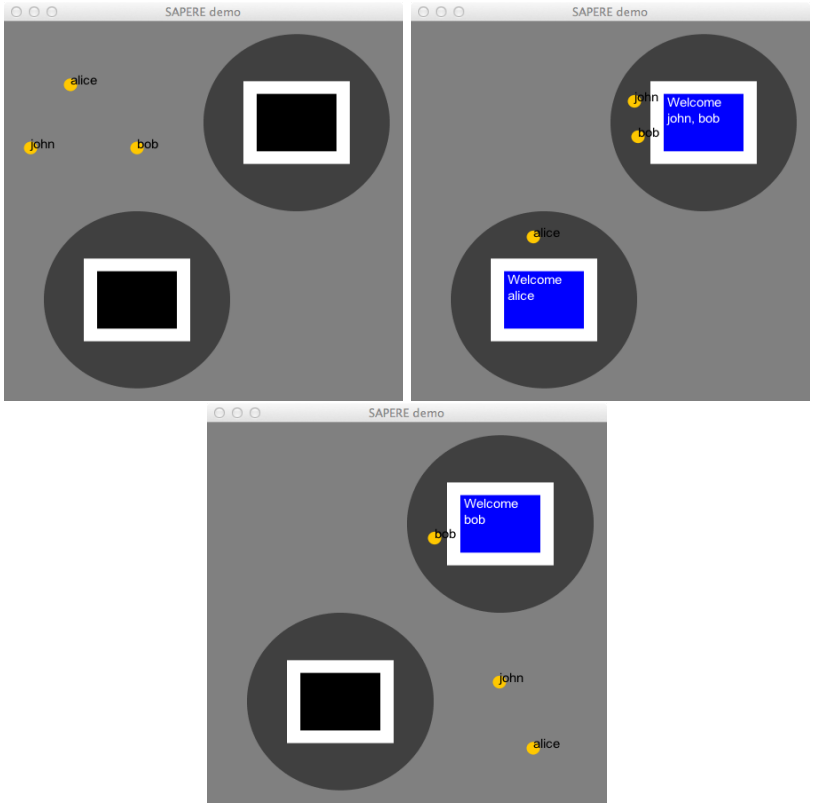

Figure 10: SAPERE demo screenshots

maintains is presented in Listing 55. *Please note that John represents information in a different way: while Bob and Alice declare their name with the* `ex:name` *property, he uses* `foaf:name`[12]. This way we are able to simulate the presence of agents that use different application domains, for describing data, and to show how this situation can be overcome thanks to semantics.

Listing 55: Person LSA (Demo scenario)

```
@prefix sapere : <...> .
@prefix xsd : <...> .
@prefix foaf: <...> .
@prefix ex : <http://www.example.org/demo#> .

# Bob Agent (Alice is similar)
sapere:lsa01234 a sapere:LSA ;
  ex:type "Person" ;
  ex:name "Bob" ;
  ex:x "0.0"^^xsd:double ;
  ex:y "0.0"^^xsd:double .

# John Agent
sapere:lsa12345 a sapere:LSA ;
```

12 Friend-Of-A-Friend is an ontology freely available on Internet, whose intent is describing contacts and their relationships.

```
  ex:type "Person" ;
  foaf:name "John" ;
  ex:x "0.0"^^xsd:double ;
  ex:y "0.0"^^xsd:double .
```

Eco-laws

The eco-laws previously presented has been reported in Listing 56, using SPARQL/SPARUL syntax. `ex:distance` is a custom function that has been implemented ad-hoc, for distance evaluation, and registered in the `CustomFunctionsRegistry` at startup; its code has been reported in Listing 57. The effect of each eco-law execution is the modification of the list of people which is actually using the display and so should be welcomed; in particular the information provided is a set of LSA-ids.

Listing 56: Eco-laws (Demo scenario)

```
# [NEAR] reactants
PREFIX ex: <http://www.example.org/demo#>
SELECT DISTINCT * WHERE {
  ?plsa ex:type "Person";
    ex:x ?px;
    ex:y ?py.
  ?dlsa ex:type "Display";
    ex:x ?dx;
    ex:y ?dy.
  FILTER NOT EXISTS {
    ?dlsa ex:user ?plsa.
  }
  FILTER (ex:distance(?px, ?py, ?dx, ?dy) < 7.0).
}
# [NEAR] products
INSERT { !dlsa ex:user !plsa. } WHERE { }

# [FAR] reactants
SELECT DISTINCT * WHERE {
  ?dlsa ex:type "Display";
    ex:x ?dx;
    ex:y ?dy;
    ex:user ?plsa.
  ?plsa ex:type "Person";
    ex:x ?px;
    ex:y ?py.
  FILTER (ex:distance(?px, ?py, ?dx, ?dy) > 7.0).
}
# [FAR] products
DELETE { !dlsa ex:user !plsa. } WHERE { }
```

Listing 57: A custom function: distance

```
1  package it.apice.sapere.demo.functions.impl;
2
3  import com.hp.hpl.jena.sparql.expr.NodeValue;
4  import com.hp.hpl.jena.sparql.function.FunctionBase4;
5
6  public class DistanceFunction extends FunctionBase4 {
7
8    @Override
9    public NodeValue exec(final NodeValue x1,
10       final NodeValue y1, final NodeValue x2,
11       final NodeValue y2) {
12     return NodeValue.makeDecimal(
13       Math.sqrt(
14         Math.pow(
15           x2.getDouble() - x1.getDouble(), 2)
16         + Math.pow(y2.getDouble() - y1.getDouble(), 2)));
17   }
18
19 }
```

DisplayAgent

`DisplayAgents` publish their location in the space and register as observers for their own LSA[13]. This way a refresh can be triggered each time a user's LSA-id is added, or removed, from values. The agent reads from the space those LSAs and extracts the name property; then it composes the message and shows it.

Exploiting semantic

If the demo is run as it has been presented until now, agent John would never be greeted. In fact `DisplayAgents` do not know what `foaf:name` is and are not able to determine John's name. Thanks to an ad-hoc ontology and the reasoning capabilities of the middleware, it is possible to overcome the problem. As shown in Listing 58 `foaf:name` can be declared as equals to `ex:name` (see Section 2.1.3); this way when the former is asserted the latter is inferred and viceversa. Now the whole scenario works fine.

Listing 58: Demo ontology

```
1 @prefix owl: <http://www.w3.org/2002/07/owl#> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3 @prefix ex: <http://www.example.org/demo#> .
4 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
5
6 ex:name rdf:type rdf:Property ;
7   owl:equivalentProperty foaf:name.
```

13 Lines 20-23 of the previous listing show a template of that LSA

5 PROFILING PERFORMANCE

CONTENTS

This chapter is meant to dive into the middleware performance. The following section will walk through three test cases:

1. a **Distributed Demo**, which realizes a producer-consumer scenario in which a sensor platform produces and diffuses temperature data to an analysis node, which aggregates it in order to determine the maximum temperature ever sensed;

2. a **micro-benchmark** focused on understanding the impact of the **compile operation**, from the LSA model to RDF, and of the **parse** its opposite (see Section 4.2.2);

3. a **micro-benchmark** dedicated to profile the degradation of agent's **READ primitive** performance when the reasoner is enabled and semantic reasoning is exploited.

Section 5.2 is dedicated to the analysis of the results. The distributed demo is the most interesting scenario, because it stresses the architecture and allows determining the maximum diffusion throughput the platform can handle, by checking how far eco-laws scheduling is able to follow the incoming data rate.

Other tests have been run, but they have not been reported here, because not conclusive; in future, this work should be extended in order to reach a better understanding of the potential of this platform.

5.1 PROFILE SCENARIOS SETUP

Let's spend two words presenting the tools that have been chosen to actually profile the middleware, before describing the details of each scenario realization.

In order to avoid the pollution of the code that is under test with the one required to run it, the profiler (VisualVM[1]) has been attached to the runtime environment, thanks to dedicated Java agents. *BTrace* plugin has been adopted: it is a dynamic tracing tool, which instruments target application classes to inject tracing code at bytecode level. In particular it allows the definition of simple methods for JVM events observation; in this way it is possible to collect information such as timings, parameters that have been passed and so on.

5.1.1 Distributed demo

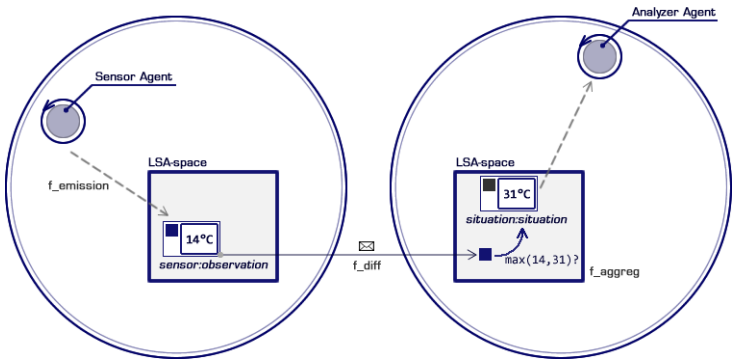

Figure 11: Distributed Demo scenario

This scenario is composed by two SAPERE nodes, which are neighbors. The former (on the left in Figure 11) is a *sensor platform*, in which a `SensorAgent` updates an LSA of type `sensor:observation`, containing the last sensed temperature[2] decorated with some additional information, which may be useful for data interpretation (e.g. Unit of Measurement, Sensor Type, etc.). The sensing operation is programmed to occur at a specific rate, passed on startup.

On the other side there is an *analysis platform*, which runs an `AnalyzerAgent`. It is able to observe an LSA, generated on launch, and print the actual maximum temperature that is stored.

Every time a new temperature is sensed and published, the diffusion mechanism is triggered, causing the LSA to be injected in the LSA-space of the analysis node. As soon as possible an aggregation eco-law verifies if the received value is greater than the maximum registered temperature – if known – and, in that case, updates the aggregated value; finally the incoming LSA is deleted. The update notifies the agent on observation, causing the actual value to be printed.

The goal of this scenario is to test the system with increasing sensor emission frequency and check how long the `ReactionManager` is able to apply aggregation eco-law before a new value is received. In other words we try to determine the maximum rate at which the platform

1 http://visualvm.java.net/
2 Sensing action has been simulated with a Random Generator Number

can work. The same test has been run both with the reasoner enabled and disabled, and results has been compared.

The emission frequency is not the only relevant variable: what happens if more sensors are run on a node? what if the topology is composed by more than two nodes and each one diffuses data to the analysis platform? Considering how the diffusion mechanism is implemented there is no need to run other tests. In fact, the scalability actually depends on the frequency at which information is received by the node, no matter what the source is. As long as the time necessary to schedule the aggregation, and apply it, is less than the period between two diffusions receival, the analysis platform would keep the pace; otherwise some delays should be expected. This is already measured by the proposed test.

5.1.2 Evaluating Parse-Compile impact

The Parse-Compile scenario has been designed in order to measure the time spent on an LSA compilation into a set of RDF statements, and the time that is necessary to parse those statements and obtain the LSA back.

The dataset used as data source has been generated in order to test the operations on an increasing amount of data: Listing 59 shows the structure of produced LSAs.

Listing 59: The "increasing-lsas" dataset

```
@prefix ...

sapere:lsa0-0
   a sapere:LSA .

sapere:lsa1-1
   a sapere:LSA ;
   ex:prop1 "val1-1-1" .

sapere:lsa1-2
   a sapere:LSA ;
   ex:prop1 "val1-2-1" ;
   ex:prop1 "val1-2-2" .

...

sapere:lsa2-1
   a sapere:LSA ;
   ex:prop1 "val2-1-1" ;
   ex:prop2 "val2-1-1" .

sapere:lsa2-2
   a sapere:LSA ;
   ex:prop1 "val2-2-1" ;
```

```
25    ex:prop1 "val2-2-2" ;
26    ex:prop2 "val2-2-1" ;
27    ex:prop2 "val2-2-2" .
28
29 sapere:lsa2-3
30    a sapere:LSA ;
31    ex:prop1 "val2-3-1" ;
32    ex:prop1 "val2-3-2" ;
33    ex:prop1 "val2-3-3" ;
34    ex:prop2 "val2-3-1" ;
35    ex:prop2 "val2-3-2" ;
36    ex:prop2 "val2-3-3" .
37
38 ...
```

The test has been run without the reasoner enabled, because it is not involved in this kind of operations.

5.1.3 Reasoner Overhead

This test tries to highlight the impact that the reasoner has on middleware performance. As explained in Section 4.3, enabling the reasoner means serializing LSA-space operations and triggering the inference process in some specific spots: in particular before executing `match` and `read` primitives.

In this scenario the same dataset as before (see Listing 59) has been used to determine how much performance scales on LSA size; the reasoner has been disabled in first place, in order to provide a baseline, then it has been turned on, so providing a comparison.

Each LSA has been, in order, `injected`, `read`, `updated` and finally `removed`. The `match` operation has not been tested, because it has been already profiled in the Distributed Demo scenario (see Section 5.1.1).

5.2 RESULTS ANALYSIS

The following sections report the results obtained by running previously presented scenarios. Traced data has been aggregated over runs and mean values has been reported.

Other than the Distributed Demo, noise has been a problem and multiple interpolations are possible. Standard deviation has also been reported in order to help the analysis.

Overall performance is compatible with expected results for a prototype implementation of the middleware. Further test should be conducted in order to lead possible improvements design for future releases.

5.2.1 Distributed Demo Results

Data have been collected by running the scenario several times: each time 1000 temperature values has been produced, diffused and aggregated, in order to obtain a reliable mean value. Sensor rate has been imposed at startup, this way samples has been taken in a range from 1 to 1000, which means that the period between temperature sensings goes from 1000ms to 1ms.

Three indexes have been defined and graphed:

1. The rate at which diffusion messages arrive to the analysis platform
2. The rate at which the aggregation is successfully applied
3. How many diffused values has been aggregated after the 1000th temperature is received by the analysis platform

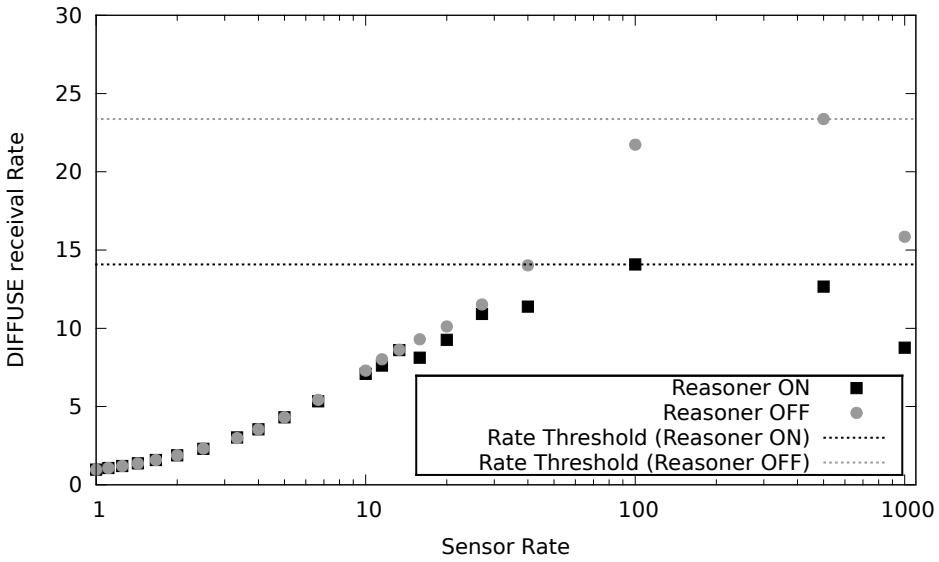

Figure 12: **Frequency of diffusion messages reception over sensor data generation rate.** Data are expressed in s^{-1}. The diffusion mechanism is not able to follow the sensor emission rate, mainly because the TCP connections between nodes are established when needed, and then closed (once the LSA has been sent). Future works should improve performance, for example by caching connections. Moreover the reported sensor rate is the one specified at launch time and it does not highlight possible rate variations, caused by OS threads scheduling policy. Horizontal lines represent the maximum rates that have been measured while running the scenario (the thresholds), respectively 14.08 when semantic reasoning was enabled and 23.37 when it was not.

The first remarkable information is that the diffusion mechanism is not able to support an infinite rate. In fact, as shown in Figure 12, the

maximum diffusion rate is limited by a threshold: when the reasoner is disabled its value is around [20.0; 25.0], while, enabling the reasoner, the maximum value drops to [10.0; 15.0].

Furthermore the diffusion frequency is always less than the one of the sensor emission, even at the lowest rates. This is mainly due to the simple implementation that has been provided: every time an LSA has to be diffused (1) a new connection to the receiving node is established, then (2) the information are sent and (3) the open socket is closed. Future works should try to improve performance, for example by caching connections between nodes. Moreover the reported sensor rate is the one passed at launch time, so it is nominal: since the middleware has not been implemented in a truely real-time environment, the actual sleep time is not accurate and can be greater than expected, with reference to the threads scheduling policy offered by the OS and the Java Virtual Machine.

Thanks to the adoption of the Java Executor Services every relocation event is enqueued, waiting for the Network Manager to be available. In this way different working speeds are balanced and all events are handled.

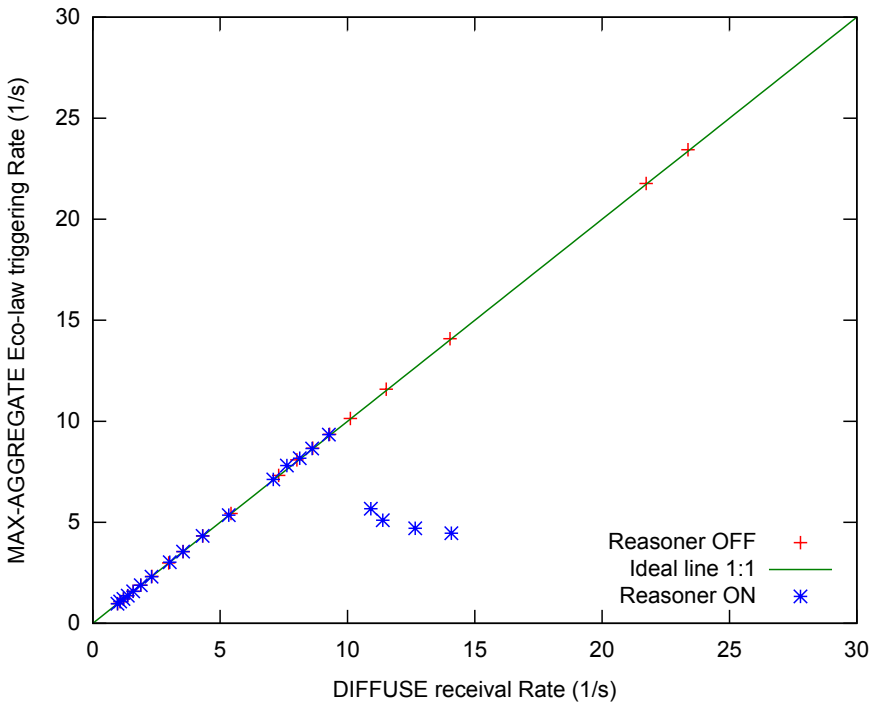

Figure 13: **Frequency of MAX-AGGREGATE eco-law triggering over diffusion messages reception rate.** Even if scheduling rate is ASAP, the effective execution depends on sensor data availability. When semantic reasoning is enabled, match execution takes more time – due to the embedded inference process – and the triggering rate drops down.

The comparison of the frequency at which LSAs are received and the rate at which aggregation is applied (see Figure 13) is ideal when no inference is taking place, in fact the latter follows the former. Otherwise, in the other case, rate 10.0 is a critical value, after whom linearity is lost. This result is enforced by the error rate, calculated as $1 - \frac{\#aggregLSAs}{\#diffusedLSAs}$: 10.0 is, again, the critical frequency beyond which the reasoner cannot work correctly.

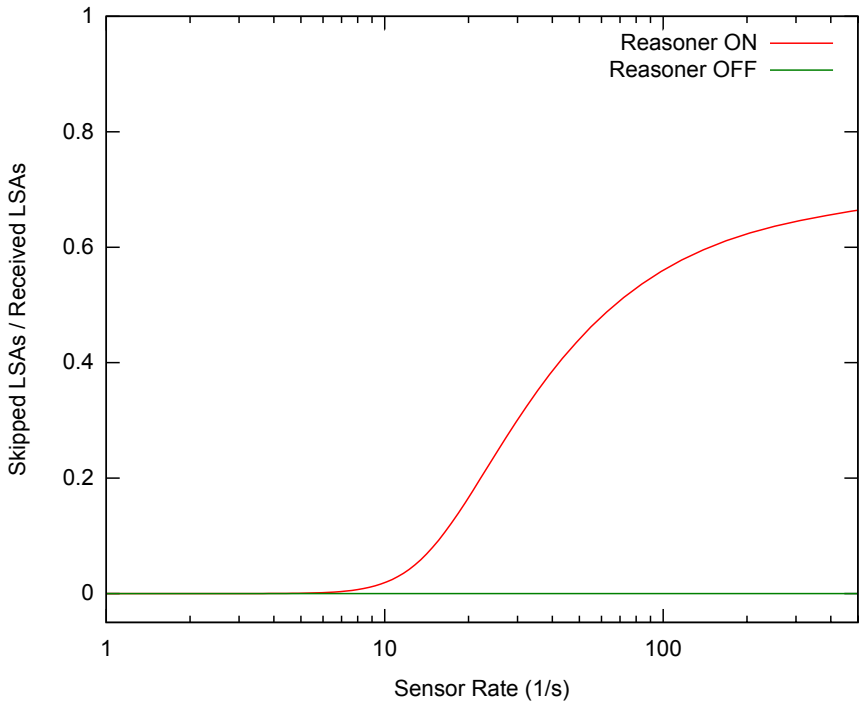

Figure 14: Fraction of sensor data that have not been aggregated after the last diffusion occurred. As consequence of the reduction of the aggregation rate – when the reasoner is on – not all the LSAs are processed in time. When no inference process is run instead, MAX-AGGREGATE is triggered fast enough for completing the elaboration.

Please note that, according to collected data, Network Manager implementation is not a bottleneck when the reasoner is enabled. In the other case, future optimizations should constantly be compared to LSA-space capabilities, in order to find the right balance and put effort in enhancing the critical component.

5.2.2 Parse-Compile Results

Unlike previous results, these show a considerable amount of noise, probably because of the overhead of profiling tools compared to collected values, that are not greater than 60ms. In fact measure sensibility can be estimated around 1ms, according to its source (the system

clock). In addition, factors like memory management, OS policies and other concurrent processes can affect results too.

In order to reduce standard deviation values – that has been graphed in order to have a better understanding of the trend – profiled operations have been run 200 times over each data and then aggregated by LSA size. Results are reported in Figure 15.

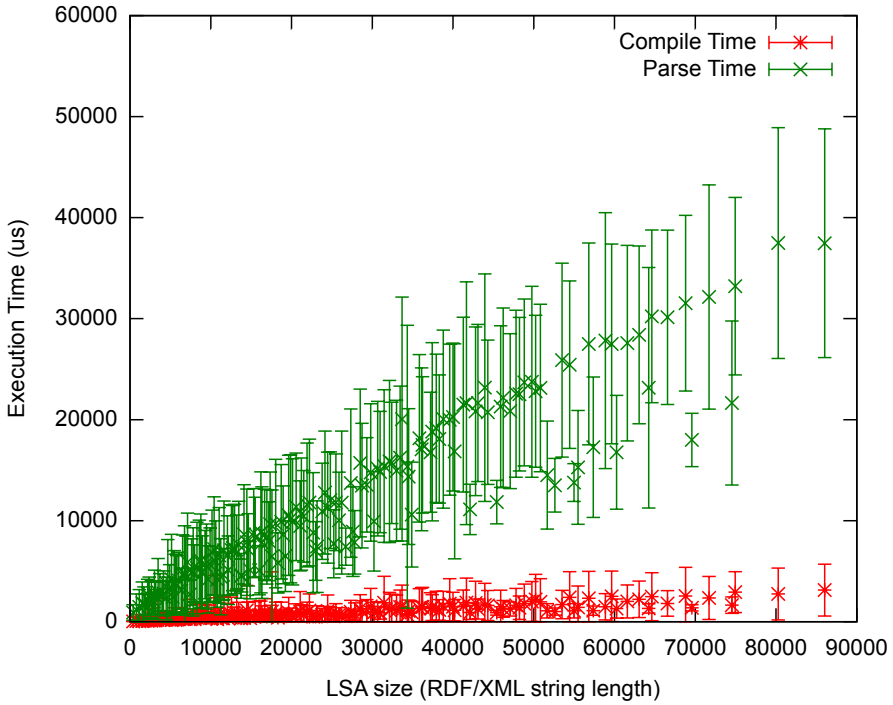

Figure 15: PARSE and COMPILE performance. The compilation process is linear and faster than the parse one. Although the standard deviation highlights a greater uncertainty, the parse operation seems to have linear complexity too (according to mean values).

Compiling an LSA is faster than the parsing operation. It shows a linear trend in function of LSA size, which is aligned to what expected: `compile` exploits *Pattern Visitor* in order to navigate LSA's structure and derive RDF statements.

Information about the `parse` operation are much more noisy and confused, but execution time is always higher. According to mean values, a linear trend can be guessed, also because it is compatible with the implemented behavior: once a RDF model has been populated with statements, all LSA-ids are iterated and related properties retrieved. In future a deeper analysis is required in order to confirm this hypothesis or reject it.

In both cases the operations scale over LSA size, letting us conclude that the choice of providing two models – explained in Section 4.2.2 – and paying a parse-compile cost does not degrades performance too much.

5.2.3 Reasoner Overhead Results

Same considerations previously expressed about noise on data hold here (see Section 5.2.2). In this case they are also stronger than before, because execution time does not exceed 10ms.

In this case the comparison highlights the difference between the execution of LSA-space primitives with or without reasoner enabled. As expected, the only CRUD primitive affected by this variable is the `read` one. The `read` primitive is implemented in order to trigger the inference process while locking the space; that's why, in Figure 16, this is a noticeable constant difference, other than a major oscillation in measures.

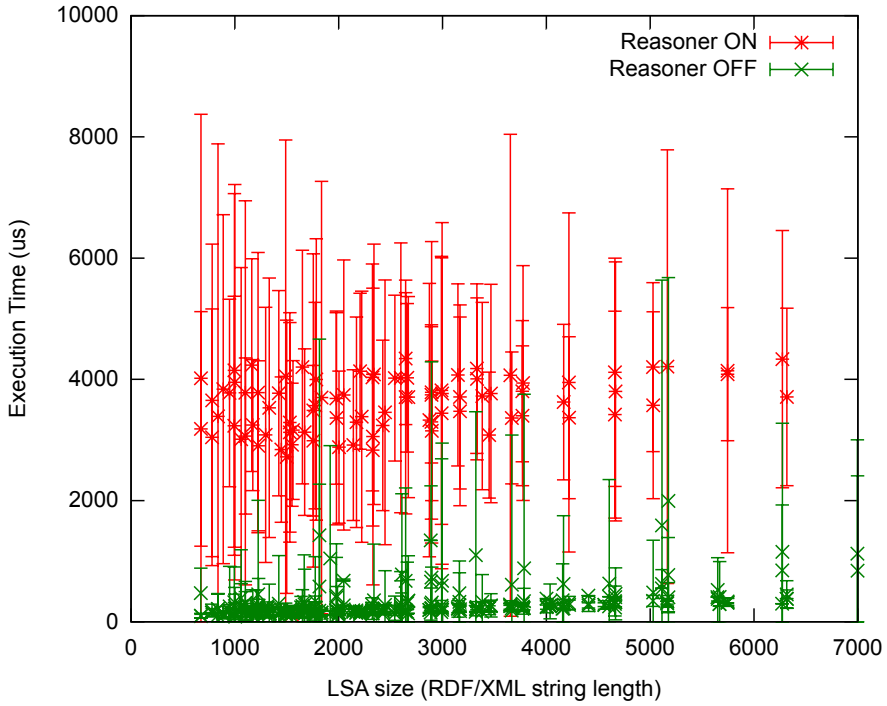

Figure 16: Agent's READ performance. The reasoner overhead slows down the execution of the primitive, despite the uncertainty of data. In both cases the trend seems not depend on LSA size too much.

When the reasoner is enabled reading an LSA has a linear cost and is barely sensitive to its dimension; this confirms the existence of some sort of indexing that speeds up RDF statements retrieval inside Jena Models. Enabling the reasoner implies a greater variation of execution time – due to inference process – but the overall behavior appears similar to the previous situation.

Other primitives data has not been graphed and reported because no remarkable differences is connected to reasoner activation. The `observe` and `ignore` primitives have not been tested, because they do not act on the RDF graph directly.

6 CONCLUSIONS

In the last few months, I have designed and developed a middleware that is able to address the main concerns of Self-aware pervasive services ecosystems, exploiting Semantic Web Technologies for describing information and inferring implicit knowledge that is hidden beyond data.

The SAPERE model has been taken as reference and the abstract architecture has been implemented, in particular the LSA-space, which is the component demanded to store data on each node of the modeled ecosystem. Apache Jena and Pellet – from Clark & Parsia – have been used as enabling technologies, because of their stability and features.

The resulting platform supports the execution of multiple agents on each node, which manifest themselves in the local space, and share data thanks to that space and the diffusion mechanism, which allow LSAs to be exchanged between nodes. Eco-laws definition and scheduling is supported in order to manipulate available information and exploit natural-inspired coordination models. Both LSA and eco-law models have been implemented in a technology-independent fashion and then compiled to Semantic Web languages, respectively RDF and SPARQL + SPARUL. Actually, the translation from eco-laws to semantic web queries has not been completed yet, due to modification of eco-laws model during these months, but some components have been created to support it as soon as a stable formalization is produced. Each agent is also able to provide ontologies – in the OWL format – for data description and openness requirement satisfaction. This way languages, deployed ecosystems and the middleware itself are fully extendable and open to future refinements, also thanks to the adoption of the component-oriented paradigm, realized with OSGi.

By now a working environment has been provided, as demonstrated by the examples described in Section 4.5 and Section 5.1.1, and some tests have been run to check correctness and performance. Results are encouraging, even if enabling the reasoner reduces the scalability. This work should be intended as starting point for future works, which can be focused on (1) scheduling policy optimization, (2) widen and fasten network protocol support, (3) deeper benchmarks execution and (4) case studies realization. This way a better understanding and management of the platform potential could be reached.

BIBLIOGRAPHY

Clark&Parsia

2011 *Pellet: OWL 2 Reasoner for Java*, http://clarkparsia.com/pellet/. (Cited on pp. 36, 37.)

Desanti, Matteo

2011 *Supporto a regole Chimico-Semantiche per la coordinazione di Pervasive Service Ecosystems*, http://apice.unibo.it/xwiki/bin/download/Theses/LSAspace/tesi.pdf, MA thesis, DEIS - Alma Mater Studiorum Università di Bologna (Cesena). (Cited on p. 47.)

Foundation, Apache Software

2011 *Apache Jena - TDB*, http://jena.apache.org/documentation/tdb/index.html. (Cited on p. 34.)

Foundation, Apache Software and HP-Labs

2010 *Jena Tutorial*, http://jena.sourceforge.net/tutorial/index.html. (Cited on pp. xi, 34.)

Hebler, John *et al.*

2009 *Semantic Web Programming*, http://www.semwebprogramming.org, Wiley Publishing Inc. (Cited on pp. 1, 7, 11, 13, 16, 57.)

Miede, André

2011 *A Classic Thesis style*, http://www.ctan.org/tex-archive/macros/latex/contrib/classicthesis/ClassicThesis.pdf. (Cited on p. vi.)

mindswap

2003 *Pellet OWL Reasoner*, http://www.mindswap.org/2003/pellet/. (Cited on p. 36.)

Montagna, Sara *et al.*

2012 "Injecting Self-organisation into Pervasive Service Ecosystems". (Cited on pp. xi, 44.)

Pantieri, Lorenzo

2011 *Introduzione allo stile ClassicThesis*, in Italian, http://www.lorenzopantieri.net/LaTeX_files/ClassicThesis.pdf.

Pantieri, Lorenzo and Tommaso Gordini

2011 *L'arte di scrivere con LaTeX*, in Italian, http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf. (Cited on p. vi.)

Stevenson, Graeme and Mirko Viroli

2011 *A formal translation of eco-laws into SPARQL*, tech. rep., https://sites.google.com/a/sapere-project.eu/sapere-wiki/dissemination/techreports/, DEIS (Cesena) - Università di Bologna. (Cited on pp. 39, 40, 45.)

Viroli, Mirko *et al.*

2011 *Early Operational Model (D1.1)*, tech. rep., http://www.sapere-project.eu, DEIS (Cesena) - Università di Bologna. (Cited on pp. xi, 39, 41, 44.)

Viroli, Mirko *et al.*

2012 "Pervasive Ecosystems: a Coordination Model based on Semantic Chemistry", in *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, ed. by Sascha Ossowski *et al.*, ACM, Riva del Garda, TN, Italy, ISBN: 978-1-4503-0857-1. (Cited on pp. vi, 2, 53.)

W3C

2004 *RDF Primer*, http://www.w3.org/TR/rdf-primer/. (Cited on pp. 5, 13, 14.)

2008a *SPARQL Query Language for RDF*, http://www.w3.org/TR/rdf-sparql-query/.

2008b *SPARQL Update - A language for updating RDF graphs*, http://www.w3.org/Submission/SPARQL-Update/. (Cited on p. 33.)

2009 *OWL 2 Web Ontology Language Document Overview*, http://www.w3.org/TR/owl2-overview/.

2012a *SPARQL 1.1 Query Language*, http://www.w3.org/TR/sparql11-query/. (Cited on p. 31.)

2012b *SPARQL 1.1 Update*, http://www.w3.org/TR/sparql11-update/. (Cited on p. 33.)

Wikipedia

2012 *Context-aware pervasive systems*, http://en.wikipedia.org/wiki/Context-aware_pervasive_systems. (Cited on p. 1.)

Zambonelli, Franco *et al.*

2011 "Self-aware Pervasive Service Ecosystems", *Procedia Computer Science*, 7 [Dec. 2011], ed. by Elisabeth Giacobino and Rolf Pfeifer, Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11), pp. 197–199, ISSN: 1877-0509, DOI: 10.1016/j.procs.2011.09.006, http://www.sciencedirect.com/science/article/pii/S1877050911005667. (Cited on pp. vi, 1, 2, 53.)