Ph.D. in
Informatics, Systems and Telecommunications

CYCLE XXVI

Curriculum: Telematics and Information Society

COORDINATOR Prof. Luigi Chisci

Design, Implementation and Validation of a Middleware Enabling the Exploitation of the Web of Documents

ING-INF/03, ING-INF/05

Ph.D. Student
Turchi Stefano

Tutor
Prof. *Giuli Dino*

Tutor
Dr. *Paganelli Federica*

Coordinator
Prof. *Chisci Luigi*

Years 2010/2013

Acknowledgements

I would like to express my special gratitude to my tutors Professor Dino Giuli, and Dr. Federica Paganelli for encouraging my research and for allowing me to grow professionally and personally. I would also like to thank Dr. Maria Chiara Pettenati for deep and inspiring discussions and her unconditioned, precious friendship. I also want to thank Professor Franco Pirri for stimulating discussions and his insatiable curiosity in science. I would especially like to thank all the workmates who become friends and shared with me these intense years: you know who you are. I would like to thank the students I supervised, who contributed with their work in opening my mind as a researcher.

Finally, a special thanks to Roberto, Fiorella, Francesca and Federica who supported me with patience and shared the joys and sorrows of this long, long journey.

Abstract

This thesis presents and details InterDataNet, a framework for information aggregation in the form of graphs of individually addressable information pieces. The framework pursues the objective of creating a *read/write* Web of Documents, by exposing these graphs as resources called IDN-Documents whose vertexes can be manipulated with CRUD operations. Graphs can be linked together and new IDN-Documents can be built by reusing parts of different IDN-Documents. Moreover, several properties (e.g. licensing, privacy, etc.) are enabled both at IDN-Document and vertex level.

The framework is made up of an information model, called InterDataNet Information Model, which regulates the representation of InterDataNet entities such as IDN-Documents, IDN-Nodes and different types of Links (Aggregation Link, Reference Link, etc.). Such information model is implemented through the InterDataNet middleware, a fully RESTful layered architecture providing services for the management of IDN-Documents, names, history and persistence. A detached module, the Adapter, provides adaptation capabilities towards external data sources to represent outer data as IDN-Documents. InterDataNet is also provided with the Activity Node, a scriptable vertex for dynamically generating data, using other vertexes' contents as inputs.

Finally, the framework includes a tool-suite, to support developers in exploiting framework capabilities. The tool-suite consists of the IDN Java and IDN.js libraries for the server-side and client-side exploitation of the IDN-Document, respectively, a visual document editor called IDN-Studio, and IDN-Viewer a JQuery plugin for rendering IDN-Documents as HTML pages.

InterDataNet has been selected to participate in the SmartSantander FP7 EU project, to provide Web of Resources capabilities in a real-word Smart City scenario. To this end, sensor data coming from the experiment platform has been represented as IDN-Documents, and added-value applications have been built on top of them. This experiment has proven useful for InterDatNet validation, and qualitative and quantitative evaluations have been included.

Contents

Acronyms and Abbreviations

ADPT Adapter

API Application Programming Interface

DAG Directed Acyclic Graph

GUI Graphical User Interface

HATEOAS Hypermedia as the Engine of Application State

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDN InterDataNet

IDN-CA InterDataNet Compliant Application

IDN-IM InterDataNet Information Model

IH Information History

ISE InterDataNet Search Engine

JSON JavaScript Object Notation

JSONPATH JavaScript Object Notation Path

LDNS Logical Domain Naming System

LRI Logical Resource Identifier

LRU Least Recently Used (cache)

MIME Multipurpose Internet Mail Extensions

OWL Web Ontology Language

PES Performance Enhancement System

QoL Quality of Life

RDF Resource Description Framework

RDFS Resource Description Framework Schema

REST Representational state transfer

SGML Standard Generalized Markup Language

SI Storage Interface

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

VR Virtual Resource

W3C World Wide Web Consortium

Wod Web of Documents

WOD Web of Data

WS Web Service

XML Extensible Markup Language

XPATH XML Path Language

Chapter 1

Introduction

Currently, web pages indexed by search engines are almost 2 billion [Kun13] and represent only a part of the attainable information. Such an immense source of knowledge makes really attractive the possibility of aggregating and correlating data from different providers. For this reason, from the middle of the 2000s the so called "mash-ups" have gained popularity. A mash-up can be defined as a Web application generated by combining content, presentation or application functionality from disparate Web sources [YBCD08]. A mash-up is built with the intent of creating valuable integration from existing data and presenting it in the document form, mostly to human users, as HTML pages. Such document is subsequently exposed on the Web, as a drop on the sea of the Web of Documents. The term Web of Documents classically indicates the a global space made of HTML pages interlinked through hypertext links, and is opposed to the Web of Data, a global data space containing billions of assertions, putting data in relation through typed links [BHBL09].

When the mash-up concept was conceived, the major issue consisted in data retrieval methodology. In most cases they were buried in HTML pages and their extraction required an *ad hoc* parsing strategy.

Assuming that the value of a well-structured mash-up increases with the amount of integrated data and considering that their retrieval from sources implies a dedicated implementation, we infer how challenging is to build a good quality mash-up.

In the subsequent years, providers began to adopt a different paradigm for

information delivery: the so called Application Programming Interfaces or APIs, for short. Basically, an API is an endpoint on which is possible to invoke operations (mostly readings) on information objects. These are delivered in a format defined by the service provider, usually XML or JSON which are much easier to be processed by applications.

Leveraging the APIs made available by the information providers, the data extraction procedure is simplified, but the problem of complying with the service interface remains. This means that the developer is forced to implement a client able to interact with the remote service, interpret the reactions (e.g., HTTP requests and status codes) and data formats. This happens for each service participating in the mash-up.

REST architectural style [Fie00a] has been estimated to be the most suitable choice for the exposure of information that will be used in mash-ups [PZL08, GIM12], but genuine RESTful APIs are still hard to find [ASJH11]. The REST architectural style has the foundation on five principles a RESTful API must fully address. Unfortunately, it is common to find APIs which are claimed to be RESTful that limit to the exposure of RPC services on nice URIs.

On the basis on what discussed above, is possible to state that creating new information by putting together the one provided by third party services presents some basic critical points:

1. implementing a mash-up requires considerable resources in terms of money, time and personnel which scales badly with the number of involved information providers;

2. is hard to determine the quality of a mash-up in terms of properties (e.g., licensing, provenance, security, privacy, etc.) of the information aggregate;

3. since REST principles are often disregarded or not properly or fully implemented, the interaction with resources is limited (e.g. data can be barely aggregated, not allowing a full *read/write* interaction).

The solution proposed in this thesis, InterDataNet, is a framework conceived with the intent of making information aggregation and reuse easy and effective. This objective is pursued by defining the IDN-Document, a composable document made up of related information pieces which are reusable and very easy to put together. An apt metaphor is the construction game, where objects of different form and type can be built from the same bricks. The fact that information is immaterial pushes the metaphor further: indeed, actual bricks are bound to the object they belong, and if a brick is reused to build something different, it will be physically taken away from that object. This does not happen in the digital world: parts of different documents can build other documents again and again,

taking advantage of the reference (URI) mechanism.

InterDataNet is made up of an information model, called InterDataNet Information Model, to represent graphs of related (and individually addressable) information pieces called IDN-Documents, a RESTful architecture to expose these graphs as Web resources and let clients interact with them, and a toolsuite supporting developers in building applications providing libraries and visual instruments. InterDataNet pursues the following objectives:

1. information resources are represented via a shared graph model whose vertexes are individually addressable and reusable;
2. *read/write* operations are enabled for every element of the graph;
3. the InterDataNet Information Model is designed with the intent of making information aggregation simple;
4. InterDatNet provides a fully RESTful architecture, enabling a *read/write* Web of Resources;
5. when an information aggregation is performed with InterDataNet, is automatically exposed through RESTful APIs. Thus, a virtuous cycle where new aggregations are ready to be reused in an optimal way is triggered. Moreover, being RESTful provides other benefits such as content negotiation;
6. InterDataNet takes charge of enforcing properties on information resources, such as privacy, licensing, provenance, etc.;
7. is provided a comprehensive toolsuite to support developers in fully exploiting the capabilities of both Information Model and architecture.

It is worth to point out that IDN-Documents capabilities are not limited to the ones of an ordinary representative of the Web of Documents, i.e. an HTML page. On the other hand, even though it is possible to include a single information grain inside the vertex of the graph, it turns out to be very inefficient due to the amount of required metadata. Therefore, it is also inappropriate to associate it with the Web of Data concept.

Because of the objectives of InterDataNet (providing capabilities for building documents from information pieces, and fully interact with them in a *read/write* way), I find more appropriate to consider an IDN-Document as an "enhanced" representative of the Web of Documents, provided that consideration is given of its strong Resource oriented connotation.

The thesis is articulated in two parts, the State of the Art and the Discussion of the Work. In the State of the Art technologies relevant to the topic are discussed,

i.e. Linked Data, Semantic Web and REST. Linked Data and Semantic Web are tackled in chapter 2 since they define a well recognized model for data on the Web (RDF). Even though RDF addresses semantics, which is not the case of InterDataNet, it was an inspiration during the design of the InterDataNet Information Model.

Chapter 3 addresses REST which is the architectural style chosen for InterDataNet for exposing IDN-Document resources to applications and for internal architecture exchanges.

In the Discussion of the Work part the InterDataNet framework is detailed: chapter 4 describes both information and data modeling, while chapter 5 details the InterDataNet architecture focusing on the Virtual Resource and Storage Interface layers, and on the Adapter, a module providing connectivity to remote information providers.

The Toolsuite is presented in chapter 6, along with its four instruments IDN Java Library, IDN.js, IDN-Studio and IDN-Viewer. Finally, in chapter 7 is presented the InterDataNet experiment as a part of the SmartSantander FP7 EU project, where conclusions are drawn on real-world use cases provided by the project. Qualitative and quantitative evaluations are also presented in this chapter.

Part I

State of the Art

Chapter 2

Linked Data and Semantic Web

Since the World Wide Web has become an everyday tool for the general public, we have witnessed a progressive expansion both in terms of produced and consumed contents and social diffusion. What was once considered a technical instrument for few professionals, first become known, then popular. The cultural and economic factors that underlie this evolution are many and different. Enough to think that just until few decades ago even disposing of the access technology was considerably expensive. Little by little, the hardware's cost declined allowing more and more people to join the ranks of the Web users.

It's difficult to determine whether the growing interest in the novel Web has promoted the lowering of the technology or the opposite. Probably what really matters is the role of immense information repository played by the Web with its expansion. Today, the Web is a synonym for information space and, as such, the centerpiece for many debates. It is well known that more than one political regime currently operates a censorship against certain types of web locations, which can be read as a representative indicator of how the media space has gained important in people's lives.

As the Web led to a massive social change, is also interesting to focus on changes encountered by the Web during the course of its evolution. The Web we experience daily is known as the Web of Documents [BHBL09, AHH+10, MJGSB11]: a global information space which can be explored through hyperlinks displayed in HTML [RLHJ+99] pages interpreted by browsers. In this way, the publication and access to documents are very simple operations, encouraging the proliferation of contents available for the community. This structure, very functional for the management of documents, does not show equally efficient for data management. Currently,

data published on the Web are mostly available as tables, CSV [Rep06], XML [BSMY+08] or similar formats that sacrifice much of their structure and semantics.

In the conventional Web, the correlation between two documents is expressed and realized via hyperlinks among HTML pages. However, this procedure does not ensure enough expressive power to connect single entities described in a document through typed links. The need for a solution to this problem has promoted a new concept of Web where data are represented as individually addressable Web resources, belonging to different domains. This information organization allows to move from one data source to another, while preserving the semantic consistency. This new conception of Web, known as Web of Data [BHIBL08, BHBL09, BLK+09], requires the availability of a proper technology substrate capable of enabling flexibility, integrity and consistency features. The implementation of the project is the result of the joint effort of many cooperating scientific organizations which apply to raise and resolve issues ranging from the definition of languages, the problem of security, reliability and trust, to the optimization of the interfaces and communication with the end user.

2.1 Towards the Semantic Web

Within the context of the Web Science, the term "Semantic Web" was introduced for the first time in 2001 by T. Berners-Lee, J. Hendler and O. Lassila in a Scientific American article [BLHL+01]. Since then, it has been associated with the idea of a global information space where intelligent agents, i.e. intelligent applications able to understand data and to guide the user with complete efficiency, operate. The achievement of such a scenario is subject to the possibility of the Web to be interpreted by machines and requires the adoption of a new procedure for the creation of data.

According to [BLHL+01] an intelligent agent should be able to perform these tasks at least:

- to understand the meaning of the texts on the Web.
- to build routes based on the information requested by the user, and guiding him/her towards them.
- on the basis of the information requested, moving from site to site connecting different elements.

In addition, the capability of checking the reliability of the retrieved information through a cross-searches methodology is hoped, although this is one of the most difficult and controversial issues. The idea of the Semantic Web is articulated around the definition of the generic conceptual domains to which data are bound. In other words, we try to represent the semantic characteristics

of a given information piece by linking it to a semantic domain. To this end, Berners-Lee introduces the Linked Data principles [BHBL09], a set of rules for the publication of data in accordance with the requirements of the Semantic Web. Linked Data is about the publication of data on the Web, with the intent of making them readable by machines and connected to other data sets so that they can reach and be reached by them.

The Linked Data principles are the following:

- use URIs [BLFM05] as names for things.
- use HTTP URIs so that people can look up those names towards them.
- when someone looks up a URI, provide useful information, using the standards (RDF [BG04], SPARQL [PS08a]).
- include links to other URIs, so that they can discover more things

While the elementary units of the Web, the HTML documents, are connected with links that don't discriminate on the data type (untyped links), Linked Data documents contain data in a particular format and are able to support connections between objects through assertions on the type (typed links). For example, if this thesis were published on the Web according to the Linked Data principles, it could be traced back to the set of documents dealing with the Web Science, or textual data or other topics. What enables the data categorization is the introduction of a hierarchy of metadata, that is, the information accompanying data which directly define its attributes. Thus, the Semantic Web is a declarative environment, in which you specify the meaning of data and not the way you intend to use them.

2.2 The Enabling Technologies

The Semantic Web depends on several technologies, some of which have been mentioned in the previous section. To ease the comprehension, their contribution to the big picture is introduced first, in an overall description. Then, a more detailed explanation is given.

2.2.1 The Big Picture, Made Short

In order to talk about something, is necessary to assign it a name. While approaching the problem in a global space, some kind of unique identifier should be applied to minimize possible ambiguity issues. According to these considerations, the Semantic Web chose a well known standard for global resources identification: the Uniform Resources Identifiers (URI) [BLFM05]. Although this acronym may not be familiar to non-experts, the use of URIs occurs very frequently in the

everyday life. Indeed, during our daily experience with the Web, we usually deal with a particular type of URI: the Uniform Resources Locator or URL [BLMM+94], better known to the public as "web page address".

The adoption of URIs allows to associate an identifier to any entity (as a resource available on the Web), but in order to move towards assertions formulation, is necessary to find a way to correlate different objects. The technology addressing this issue is the Resource Description Framework or RDF [BG04], which provides the tools for implementing machine-processable statements. An RDF statement is very similar to a normal sentence with the difference that words are replaced by URIs. Each declaration of this type is composed of three elements: a subject, a predicate and an object.

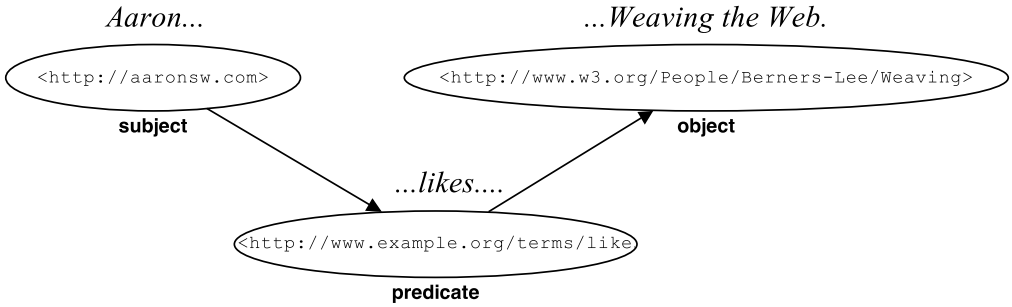


Figure 2.1. *a sample RDF declaration stating that Aaron likes the book "Weaving the Web", taken from [Swa02].*

Fig. 2.1 shows an example from [Swa02] of a simple RDF statement saying that Aaron likes the book "Weaving the Web". The statement is represented as a triple where the entity referred as `http://aaronsw.com` is the subject, `http://www.example.org/terms/like` is the predicate and `http://www.w3.org-/People/Berners-Lee/Weaving` is the object. Please note that each entity refers to a specific domain.

If a URI acts like a pointer, by dereferencing it one should reach a meaningful representation of the pointed resource. This consideration leads to the concept of dictionary, which is a repository of terms definitions. The RDF Vocabulary Definition Language (or RDF Schema or RDFS) is a language that can be used to define the vocabulary (i.e., the terms) to be used in an RDF graph. It is used to indicate that we are describing specific kinds or classes of resources, and uses specific properties in describing those resources.

The RDF Schema defines classes and properties and supports hierarchy of definitions. For example is possible to assert that the entity "School" is a class and the entity "Welsh Establishment" is either a class but subclasses the "School" entity. As another example, is possible to state that "being a staff at" is a property and "being a head of" is either a property and a subproperty of the first. This

RDF entities organization is very important because is the first step to return, say, the "head of a department" as a result of a "find every person in the staff" query.

To achieve a real semantic expressiveness, definitions and properties are not enough. Is necessary to introduce a richer modeling and semantics dealing with axioms defining *transitive*, *symmetric*, *inverse of*, *functional*, *inverse functional*, *equivalent* properties; axioms defining *intersection*, *union*, *disjoint*, *equivalent of classes*; *restriction on classes* such as *some values form*, *all values from*, *cardinality*, *has one value*, *one of*, keys; axiom defining *same as*, *different from*, *all different on individuals*; and *imports*.

The Ontology Web Language [DS04] or OWL is the technology to answer these needs. It supports much richer modeling and provides consistency checking of models and data. By leveraging OWL is also possible to create new knowledge from a knowledge base, by exploiting inference capabilities.

Since everyone is free to create his/her own ontology, a translation system that allows definitions reuse is required. For this reason, is possible to interconnect different vocabularies and ontologies by defining RDF mappings. The use of URIs for identifying resources (with the HTTP protocol [FGM$^+$99] as a mechanism for data retrieval, and the RDF model for their description) allows the Linked Data to rely directly on the existing Web architecture. Therefore, is possible to see the Web of Data as an additional layer rooted in the classic Web [Ran98].

Now, the meaning of the RDF graph mentioned in the previous paragraph gets clearer. By leveraging this technology stack, is possible to create an instance of an entity, say a "Professor", associating it with a unique URI to be retrieved on the Web, and describing it by adding relations to other entities, say a "School", a "Department", other professors or a "Classroom". These entities will be defined with the same criteria, and the glue is the shared ontology. Indeed, the ontology keeps terms definitions and relations as previously discussed. By consulting the ontology, is possible to find that a "Professor" is also a "Member of the teaching staff" or a "Public employee".

The graph can be expanded at will, and every time a new entity is added, because of the semantic contextualization, related entities improve their definition quality. Moreover, the more a graph is complete, the more sophisticated goals (e.g. search, discovery, inferences, reasoning) become achievable.

To sum up, is possible to state that [MvH04]:

- URIs provides a way to reference resources.
- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.
- RDF is a datamodel for objects ("resources") and relations between them, provides a simple semantics for this datamodel, and these datamodels can be represented in an XML syntax.

- XML Schema is a language for restricting the structure of XML documents and also extends XML with datatypes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.
- SPARQL is a SQL-like language to query the RDF graph

The contents in the following sections are a readjustment of the [Las99, MM04, Bec04, MvH04, PS08b] W3C recommendations. These documents have been chosen because are the reference for the topic accepted by the scientific community. While discussing the SPARQL query language in the dedicated section 2.2.7, the [DuC11] book has been also used.

2.2.2 Identifying Resources: URI

A Uniform Resource Identifier (URI) [BLFM05] is a sequence of characters that identifies an abstract or physical resource. According to [BLFM05], the URI syntax defines a grammar that is a superset of all valid URIs, allowing an implementation to parse the common components of a URI reference without knowing the scheme-specific requirements of every possible identifier.

The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment, as shown in the following.

<scheme name>:<hierarchical part>[?<query>][#<fragment>]

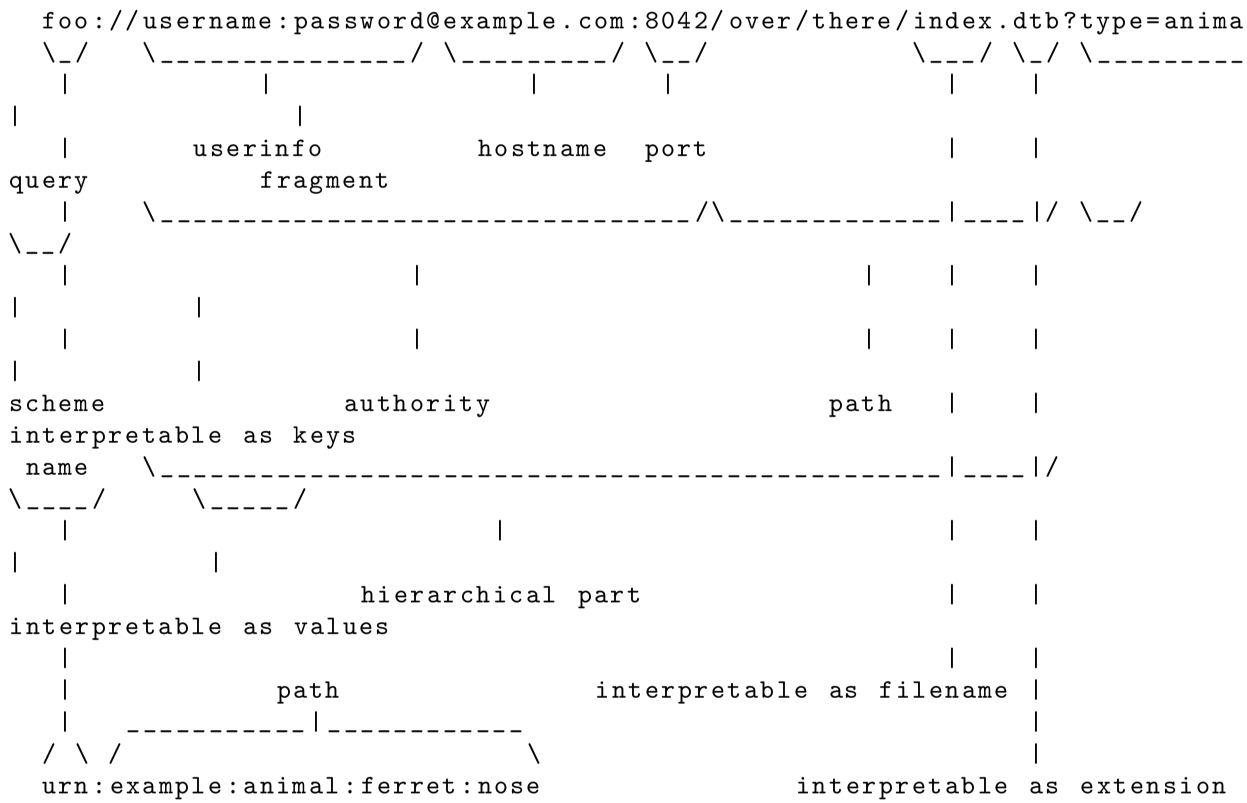
Each URI begins with a scheme name that refers to a specification for assigning identifiers within that scheme. As such, the URI syntax is a federated and extensible naming system wherein each schemes specification may further restrict the syntax and semantics of identifiers using that scheme. The scheme registry maintains the mapping between scheme names and their specifications.

Many URI schemes include a hierarchical element for a naming authority so that governance of the name space defined by the remainder of the URI is delegated to that authority (which may, in turn, delegate it further). The generic syntax provides common means for distinguishing an authority based on a registered name or server address, along with optional port and user information.

The path component contains data, usually organized in hierarchical form, that, along with data in the non-hierarchical query component, serves to identify a resource within the scope of the URIs scheme and naming authority (if any). The path is terminated by the first question mark or number sign character, or by the end of the URI.

The query component contains non-hierarchical data that, along with data in the path component, serves to identify a resource within the scope of the URIs scheme and naming authority (if any). The query component is indicated by the first question mark character and terminated by a number sign character or by the end of the URI.

Listing 2.1 gives a complete representation of the URI scheme (the example is borrowed from [BLFM05]).



Listing 2.1. The URI complete scheme

2.2.3 Providing the Base Syntax: XML

The eXtensible Markup Language (XML) [BSMY+08] is a markup language, i.e. a language based on markers whose syntax allows to define and control the meaning of the elements in a document. As its name says, XML is extensible because supports the definition of custom tags.

As a consequence of the browsers' war in the ninety, when the browsers' vendors released an HTML proprietary extension of the official version, the World Wide Web Consortium (W3C) was forced to follow the individual HTML extensions. The W3C had to choose which characteristics standardize and which characteristics let go. In this context become clear the need of a markup language making the definition of tag easier, still remaining a standard. The XML project started in the early ninety in the context of the SGML Activity [Bry88], and attracted so much interest to convince the W3C to create a work group called XML Working Group and a Commission, the XML Editorial Review Board with with the duty to establish the project specifications, which became a recommendation in the 1998.

Compared with HyperText Markup Language (HTML) [JHR99], XML has a different purpose. While the former describes a grammar for the representation and structuring of hypertexts, the latter is a metalanguage used to create new languages for the description of structured documents. While the HTML has a well defined set of tag, the XML supports their custom definition.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sensors>
3    <sensor>
4      <name>K202</name>
5      <producer>ACME</producer>
6    </sensor>
7    <sensor>
8      <name>K206</name>
9      <producer>Foster</producer>
10   </sensor>
11 </sensors>
```

Listing 2.2. A very simple XML example

Listing 2.2 shows a very simple instance of XML code, representing a set of sensors. In the first line is possible to see the XML version declaration as well as the chosen encoding option. From the example, it is clear the structure of the sensors set representation: the name "sensors" is assigned to a main container, and a set of "sensor" elements appear as its content. These elements are defined in the same way, that is by specifying the content that, in this case, is made of a "name" and a "producer" element. Finally, the leaves of the XML tree contain

some plain text.

XML uses markers called *tags* to transfer a semantic to the text. Tags convey information in two ways: enclosing it or by using parameters. In order to be well formed, an XML document must have a closing tag for every opening tag. This formalism can also be expressed in a concise form with auto-closing inline tags, which conveys information with parameters only.

A tag is delimited with angle brackets, i.e. the less-than <and the greater-than signs >, while a closing tag starts with the characters sequence </ . An inline tag can be also closed ending it with the />characters sequence. Comments are enclosed between the <!—and —>characters sequence. Listing 2.3 shows the XML syntax for tags.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <example>
3   <tag>this is a normal tag</tag>
4   <tag parameter="value">this is a normal tag, with a
    parameter</tag>
5   <tag description="this is an inline autoclosing tag"/>
6   <!-- this is a comment -->
7 </example>
```

Listing 2.3. An XML syntax example

The XML syntax is very strict and while writing an XML document is mandatory to follow precise rules:

- tags can't start with numbers or special characters and can not contain spaces;
- tags must be balanced, i.e. nesting errors are not permitted.

Listing 2.4 shows an unbalanced (and therefore invalid) XML document, since "A" and "B" tags are closed in the inverse order.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <example>
3   <A>
4     <B>
5       </A>
6   </B>
7 </example>
```

Listing 2.4. An invalid XML document

Summing up, to be properly interpreted an XML document must be well formed, i.e. must have the following properties:

- the content must be defined in the very first element of the document, by specifying XML version and encoding;
- the content must be delimited with a beginning and end tag;
- the content must be properly nested.

Since XML supports the definition of custom documents, a different language to describe their content is very useful. This means that if a user create an XML document representing a set of sensors, probably he/she will also need a formalism to define the structure of valid sensors documents.

XML Schema [SMGB+12] was the first separate schema language for XML to achieve Recommendation status by the W3C. Its purpose is to define which elements, data types, hierarchy are allowed for a given XML document. Mostly, this is useful for validating XML document instances against a predetermined model.

Technically, an XML schema document (or schemas) is an abstract metadata collection, consisting in a set of schema components, i.e. attributes and elements declarations and simple or complex definitions. Schemas are organized by *namespaces*: in a document, every schema component is associated with a certain namespace which is a property of the schema as a whole. A schema can include other schemas for the same namespace, and can import other schemas for different namespaces.

Usually, XML schema documents have the ".xsd" file extension. No dedicated MIME type has been created for XML schemas yet, so types most frequently adopted are "application/xml" and "text/xml".

The Listing 2.5 shows an XML schema representing a class of ship orders document instances. As previously discussed, the schema contains element and attribute type declarations. The syntax of the XML schema won't be detailed here, not to encumber the reading. Further details can be found in the official documentation [SMGB+12].

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="stringtype">
4     <xs:restriction base="xs:string"/>
5   </xs:simpleType>
6   <xs:simpleType name="inttype">
7     <xs:restriction base="xs:positiveInteger"/>
8   </xs:simpleType>
9   <xs:simpleType name="dectype">
```

```
10      <xs:restriction base="xs:decimal"/>
11    </xs:simpleType>
12    <xs:simpleType name="orderidtype">
13      <xs:restriction base="xs:string">
14        <xs:pattern value="[0-9]{6}"/>
15      </xs:restriction>
16    </xs:simpleType>
17    <xs:complexType name="shiptotype">
18      <xs:sequence>
19        <xs:element name="name" type="stringtype"/>
20        <xs:element name="address" type="stringtype"/>
21        <xs:element name="city" type="stringtype"/>
22        <xs:element name="country" type="stringtype"/>
23      </xs:sequence>
24    </xs:complexType>
25    <xs:complexType name="itemtype">
26      <xs:sequence>
27        <xs:element name="title" type="stringtype"/>
28        <xs:element name="note" type="stringtype" minOccurs="0"/>
29        <xs:element name="quantity" type="inttype"/>
30        <xs:element name="price" type="dectype"/>
31      </xs:sequence>
32    </xs:complexType>
33    <xs:complexType name="shiporder type">
34      <xs:sequence>
35        <xs:element name="orderperson" type="stringtype"/>
36        <xs:element name="shipto" type="shiptotype"/>
37        <xs:element name="item" maxOccurs="unbounded"
          type="itemtype"/>
38      </xs:sequence>
39      <xs:attribute name="orderid" type="orderidtype"
        use="required"/>
40    </xs:complexType>
41    <xs:element name="shiporder" type="shiporder type"/>
42  </xs:schema>
```

Listing 2.5. An XML Schema describing a ship orders class of XML documents

The Listing 2.6 shows an XML document compliant with the previous XML schema. This document is an element of the class of documents validated by the schema shown in Listing 2.5.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:simpleType name="stringtype">
```

```
4     <xs:restriction base="xs:string"/>
5   </xs:simpleType>
6   <xs:simpleType name="inttype">
7     <xs:restriction base="xs:positiveInteger"/>
8   </xs:simpleType>
9   <xs:simpleType name="dectype">
10    <xs:restriction base="xs:decimal"/>
11  </xs:simpleType>
12  <xs:simpleType name="orderidtype">
13    <xs:restriction base="xs:string">
14      <xs:pattern value="[0-9]{6}"/>
15    </xs:restriction>
16  </xs:simpleType>
17  <xs:complexType name="shiptotype">
18    <xs:sequence>
19      <xs:element name="name" type="stringtype"/>
20      <xs:element name="address" type="stringtype"/>
21      <xs:element name="city" type="stringtype"/>
22      <xs:element name="country" type="stringtype"/>
23    </xs:sequence>
24  </xs:complexType>
25  <xs:complexType name="itemtype">
26    <xs:sequence>
27      <xs:element name="title" type="stringtype"/>
28      <xs:element name="note" type="stringtype" minOccurs="0"/>
29      <xs:element name="quantity" type="inttype"/>
30      <xs:element name="price" type="dectype"/>
31    </xs:sequence>
32  </xs:complexType>
33  <xs:complexType name="shipordertype">
34    <xs:sequence>
35      <xs:element name="orderperson" type="stringtype"/>
36      <xs:element name="shipto" type="shiptotype"/>
37      <xs:element name="item" maxOccurs="unbounded"
        type="itemtype"/>
38    </xs:sequence>
39    <xs:attribute name="orderid" type="orderidtype"
      use="required"/>
40  </xs:complexType>
41  <xs:element name="shiporder" type="shipordertype"/>
42 </xs:schema>
```

Listing 2.6. A ship orders XML document instance

2.2.4 Modeling Data: RDF

The Resource Description Framework (RDF) is a technology to let machines understand data on the Web. The basic methodology is very clean: an information entity is annotated with special markers so that machines can recognize that entity and inspect its meaning. The technical way to accomplish this task is to define a set of metadata, i.e. data describing other data. RDF is a foundation for processing metadata since it provides interoperability between applications exchanging information on the Web. RDF can be applied in different scenarios such as resource discovery, improving the criteria for search engines, cataloging by enhancing the description of resources, just to mention few.

Thus, the broad goal of the RDF is to define a mechanism to describe resources which doesn't require any assumption about a particular application domain, nor defines (*a priori*) the semantics of any application domain.

Imagine trying to state that someone called John Smith created a web page. A simple way of doing it would be the following:

`http://www.example.org/index.html` has a creator whose value is John Smith.

In order to represent the concept expressed by the statement, is mandatory to name or identify three things:

- the thing described by the statement, that is the *subject*;
- the property associated to the subject, that is the *predicate*;
- the value of the property, that is the *object*.

RDF is based on the idea that is possible to describe a thing by declaring some properties and their corresponding values. In order to make statements non ambiguous two requirements must be met:

- the entities must be identified with a machine processable code, with no possibility of confusion with a similar identifier;
- the statements must be declared using a machine processable language.

The existing Web architecture yet provides the technologies to address these requirements: the Uniform Resource Identifier and the eXtensible Markup Language. These two technologies have been introduced in subsections 2.2.2 and 2.2.3, respectively.

2.2.4.1 The RDF Data Model

RDF models statements using the graph notation, associating vertexes with subject and object and edges (directed from subject to object vertexes) with predicates. Therefore, the example above can be represented as shown in Fig. 2.2.

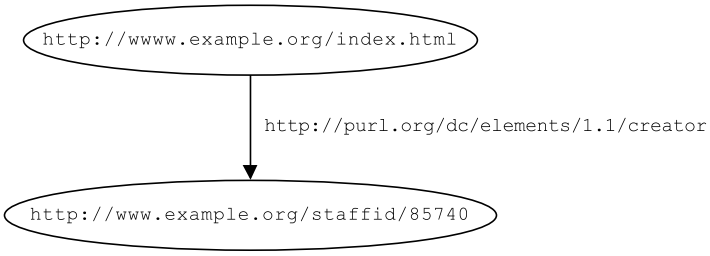


Figure 2.2. *a simple RDF statement.*

Groups of statements map directly to the graph's elements. For example, "`http://www.example.org/index.html` has a creation date whose value is August 16, 1999" and "`http://www.example.org/index.html` has a language whose value is English" are represented in Fig. 2.3.

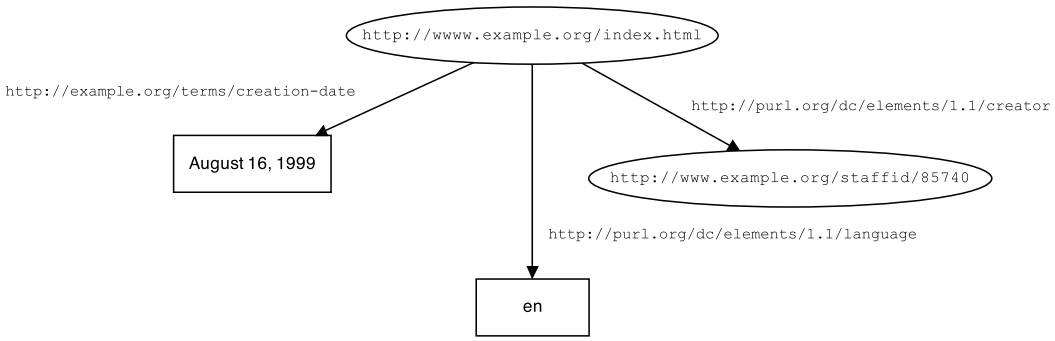


Figure 2.3. *More statements about the same resource.*

Fig. 2.3 shows also that the RDF statements can be either URI and literals value. Literals may not be used as subjects or predicates in RDF statements. In drawing RDF graphs, nodes that are URIs are shown as ellipses, while nodes that are literals are shown as boxes.

The RDF notation requires that subject, object and predicate URIs are surrounded by angle brackets. From a visual point of view, it can be quite complicated. To this purpose is available a simpler notation substituting an XML qualified name (or QName) without angle brackets as an abbreviation for a full URI reference. A QName contains a prefix that has been assigned to a namespace URI, followed by a colon, and then a local name. So, for example, if the QName prefix `foo` is assigned to the namespace URI `http://example.org/somewhere/`, then the QName `foo:bar` is shorthand for the URI `http://example.org/somewhere/bar`.

Using this new shorthand, the previous set of triples can be written as:

Since RDF uses URIs instead of words to name things in statements, RDF refers to a set of URIs (particularly a set intended for a specific purpose) as a

ex:index.html	dc:creator	exstaff:85740
ex:index.html	exterms:creation-date	"August 16, 1999"
ex:index.html	dc:language	"en"

vocabulary. Often, the terms inside a vocabulary belongs to the same namespace so that is possible to guess the adopted vocabulary by the QName. However, this is just a convention. The RDF model only recognizes full URIs; it does not inspects URIs or use any knowledge about their structure. In particular, RDF does not assume there is any relationship between URIs just because they have a common leading prefix.

Sometimes is useful to represent as an RDF entity all the information that pertain a vertex. Let's clarify with an example. As discussed above, to represent an address is possible to adopt the literal form. In this case a valid RDF statement is the following:

exstaff:85740	exterms:address	"1501 Grant Avenue, Bedford, Massachusetts 01730"

Looking at the address information, is reasonable to think that it would be better represented if the address data were split up in separate pieces. This means that it would be necessary to have an empty connector vertex. Fig. 2.4 shows a possible representation of the scenario.

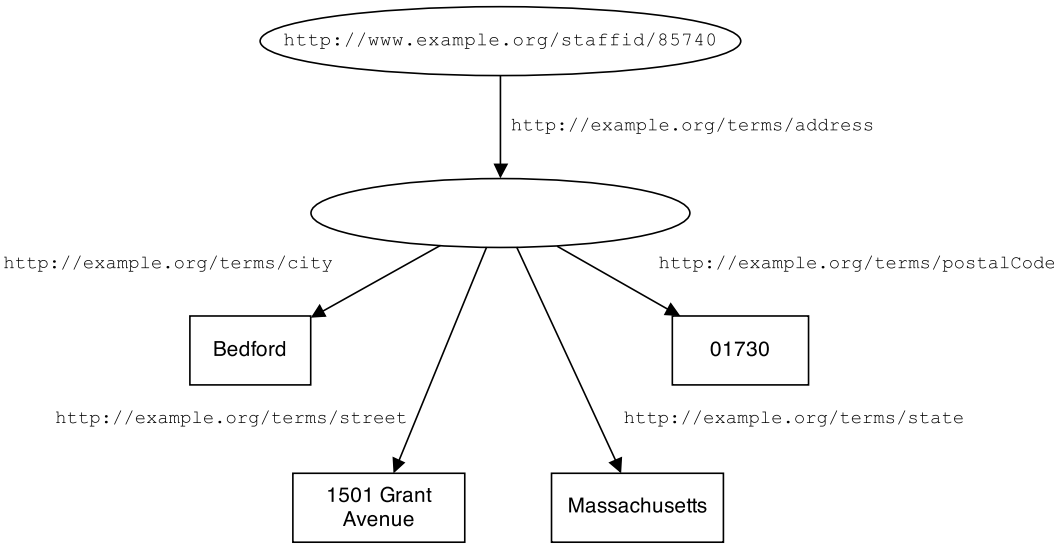


Figure 2.4. *The role of the blank vertex.*

This blank vertex serves its purpose in the drawing without containing a URI, since the node itself provides the necessary connectivity between the various parts of the graph. However, some form of explicit identifier for that node is needed in order to represent this graph as triples. For this reason, blank vertex identifier have a special notation: "`:name`". In the example discussed above, it could be "`_:johnaddress`" and the corresponding RDF triples could be:

```
exstaff:85740  exterms:address       _:johnaddress
_:johnaddress   exterms:street   "1501 Grant Avenue"
_:johnaddress   exterms:street            "Bedford"
_:johnaddress   exterms:street      "Massachusetts"
_:johnaddress   exterms:street               "01730"
```

Since RDF is definitely oriented to machine processing, in order to avoid ambiguity in interpreting attributes' values, it supports data type declarations. An RDF typed literal is formed by pairing a string with a URI that identifies a particular data type. For example, using a typed literal, John Smith's age could be described as being the integer number 27 using the triple:

```
exstaff:85740  exterms:age  "27"^^xsd:integer
```

The RDF model as discussed so far is good at representing entities and their relations. A tile is still missing to the puzzle until terms used in the statements are not described in a vocabulary. The basis for describing such vocabularies is the RDF Schema.

2.2.5 Defining Classes and Properties: RDF Schema

Leveraging RDF is possible to make assertions on entities, by specifying subjects, objects and predicates. As previously discussed, these entities are identified through URIs. What is still missing so far, is a way to define the abstract concept these URIs refer to.

The RDF itself provides no means for defining such application-specific classes and properties. They are described as an RDF vocabulary, using the RDF Vocabulary Description Language [BG04] or RDF Schema.

RDF Schema does not provide application-specific terms, but it provides the instruments to describe these terms and the way they relate (for example, to say that the property `ex3:jobTitle` will be used in describing a `ex3:Person`). In other words, RDF Schema provides a type system for RDF. The RDF Schema type system is similar, in some extent, to the type systems of object-oriented programming languages. Thus, RDF Schema allows resources to be defined as

instances of one or more classes. In addition, classes can be organized in a hierarchical way; for example a class `ex:Dog` might be defined as a subclass of `ex:Mammal` which is a subclass of `ex:Animal`.

RDF Schema instruments are also provided in the form of an RDF vocabulary; that is, as a specialized set of predefined RDF resources with their own meanings. The resources in the vocabulary have URIs with the prefix

`http://www.w3.org/2000/01/rdf-schema#`

which is conventionally associated with the QName prefix `rdfs:`. Vocabulary descriptions written in the RDF Schema language are valid RDF graphs.

A class in RDF Schema corresponds to the generic concept of a Type or Category. Classes are described using the RDF Schema `rdfs:Class` and `rdfs:Resource`, and the properties `rdf:type` and `rdfs:subClassOf`.

For example, suppose an organization `example.org` is willing to use RDF to provide information about different kinds of motor vehicles. In RDF Schema, `example.org` would first need a class to represent motor vehicles. Resources that belong to a class are called its instances. In this case, `example.org` describes entities that are instances of motor vehicles.

In RDF Schema, a class is any resource having an `rdf:type` property whose value is `rdfs:Class`. Assuming that `ex:` stands for `http://www.example.org/-schemas/vehicles`, which is used as the prefix for URI from `example.org`'s vocabulary, `example.org` would write the RDF statement:

```
ex:MotorVehicle  rdf:type  rdfs:Class
```

the property `rdf:type` indicates that a resource is an instance of a class. So, having described `ex:MotorVehicle` as a class, resource `exthings:companyCar` would be described as a motor vehicle by the RDF statement:

```
exthings:companyCar  rdf:type  ex:MotorVehicle
```

The resource `rdfs:Class` itself has an `rdf:type` of `rdfs:Class`. A resource may be an instance of more than one class.

After describing class `ex:MotorVehicle`, `example.org` might want to describe additional classes representing different specialized kinds of motor vehicles. These classes can be described in the same way as class `ex:MotorVehicle`, by assigning a URI for each new class, and writing RDF statements describing these resources as classes, e.g., writing:

```
ex:Van    rdf:type  rdfs:Class
ex:Truck  rdf:type  rdfs:Class
```

and so on. However, these statements by themselves describe the individual classes only. `Example.org` may also want to indicate their special relationship to class `ex:MotorVehicle`, i.e., that they are specialized kinds of `MotorVehicle`.

This kind of specialization relationship between two classes is described using the `rdfs:subClassOf` property. For example, to state that `ex:Van` is a specialized kind of `ex:MotorVehicle`, `example.org` would write the RDF statement:

```
ex:Van  rdf:subClassOf  rdfs:MotorVehicle
```

The meaning of this `rdfs:subClassOf` relationship is that any instance of class `ex:Van` is also an instance of class `ex:MotorVehicle`. So, if resource `exthings:companyVan` is an instance of `ex:Van` then, based on the declared `rdfs:subClassOf` relationship, RDF software written to understand the RDF Schema vocabulary can infer the additional information that `exthings:companyVan` is also an instance of `ex:MotorVehicle`.

This example of the `exthings:companyVan` illustrates the point made earlier about RDF Schema defining an extended language. RDF itself does not define the special meaning of terms from the RDF Schema vocabulary such as `rdfs:subClassOf`. So if an RDF schema defines this `rdfs:subClassOf` relationship between `ex:Van` and `ex:MotorVehicle`, RDF software not written to understand the RDF Schema terms would recognize this as a triple, with predicate `rdfs:subClassOf`, but it would not understand the special significance of `rdfs:subClassOf`, and not be able to draw the additional inference.

The `rdfs:subClassOf` property is *transitive*. This means that given the RDF statements:

```
ex:Van      rdf:subClassOf  rdfs:MotorVehicle
ex:MiniVan  rdf:subClassOf           rdfs:Van
```

RDF Schema defines `ex:MiniVan` as also being a subclass of `ex:MotorVehicle`. As a result, RDF Schema defines resources that are instances of class `ex:MiniVan` as also being instances of class `ex:MotorVehicle`. A class may be a subclass of more than one class (for example, `ex:MiniVan` may be a subclass of both `ex:Van` and `ex:PassengerVehicle`). RDF Schema defines all classes as subclasses of class `rdfs:Resource`. Fig 2.5 shows the full class hierarchy discussed in these examples.

Listing 2.7 shows how this schema could be written in RDF/XML.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE rdf:RDF [<!ENTITY xsd
  "http://www.w3.org/2001/XMLSchema\#">]>
```

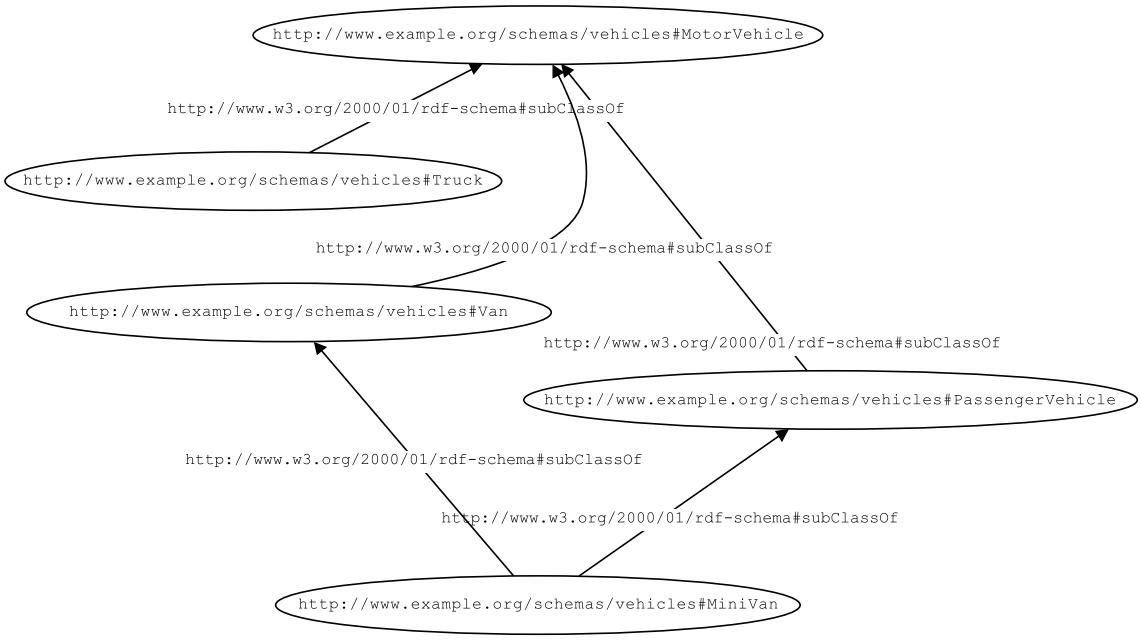


Figure 2.5. *A vehicle class hierarchy*

```
3  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns\#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema\#"
   xml:base="http://example.org/schemas/vehicles">
4
5  <rdf:Description rdf:ID="MotorVehicle">
6    <rdf:type
     rdf:resource="http://www.w3.org/2000/01/rdf-schema\#Class"/>
7  </rdf:Description>
8
9  <rdf:Description rdf:ID="PassengerVehicle">
10   <rdf:type
     rdf:resource="http://www.w3.org/2000/01/rdf-schema\#Class"/>
11   <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
12 </rdf:Description>
13
14 <rdf:Description rdf:ID="Truck">
15   <rdf:type
     rdf:resource="http://www.w3.org/2000/01/rdf-schema\#Class"/>
16   <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
17 </rdf:Description>
18
19 <rdf:Description rdf:ID="Van">
20   <rdf:type
```

```
    rdf:resource="http://www.w3.org/2000/01/rdf-schema\#Class"/>
21    <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
22  </rdf:Description>
23
24  <rdf:Description rdf:ID="MiniVan">
25    <rdf:type
    rdf:resource="http://www.w3.org/2000/01/rdf-schema\#Class"/>
26    <rdfs:subClassOf rdf:resource="\#Van"/>
27    <rdfs:subClassOf rdf:resource="\#PassengerVehicle"/>
28  </rdf:Description>
29  </rdf:RDF>
```

Listing 2.7. The vehicle class hierarchy in RDF/XML

In addition to describing classes, user communities also need to describe properties that characterize those classes. In RDF Schema, properties are described using the RDF class `rdf:Property`, and the RDF Schema properties `rdfs:domain`, `rdfs:range`, and `rdfs:subPropertyOf`.

All properties in RDF are described as instances of class `rdf:Property`. So a new property, such as `exterms:weightInKg`, is described by assigning the property a URI, and describing that resource with an `rdf:type` property whose value is the resource `rdf:Property`, for example, by writing the RDF statement:

```
exterms:weightInKg  rdf:type  rdf:Property
```

RDF Schema also provides vocabularies for describing how properties and classes are intended to be used together. The most important information of this kind is supplied by using the RDF Schema properties `rdfs:range` and `rdfs:domain` to describe application-specific properties.

The `rdfs:range` property is used to indicate that the values of a property are instances of a designated class. For example, if `example.org` wanted to indicate that the property `ex:author` had values that are instances of class `ex:Person`, it would write the RDF statements:

```
ex:Person   rdf:type     rdfs:Class
ex:author   rdf:type   rdf:Property
ex:author  rdfs:range     ex:Person
```

A property, say `ex:hasMother`, can have zero, one, or more than one range property. If `ex:hasMother` has no range property, then nothing is said about the values of the `ex:hasMother` property. If `ex:hasMother` has one range property, say one specifying `ex:Person` as the range, this says that the values of

the ex:hasMother property are instances of class ex:Person. If ex:hasMother has more than one range property, say one specifying ex:Person as its range, and another specifying ex:Female as its range, this says that the values of the ex:hasMother property are resources that are instances of all of the classes specified as the ranges, i.e., that any value of ex:hasMother is both a ex:Female and a ex:Person.

This last point may not be obvious. However, stating that the property ex:hasMother has the two ranges ex:Female and ex:Person involves making two separate statements:

```
ex:hasMother  rdfs:range  ex:Female
ex:hasMother  rdfs:range  ex:Person
```

For any given statement using this property, say:

```
exstaff:frank  ex:hasMother  exstaff:frances
```

in order for both the rdfs:range statements to be correct, it must be the case that exstaff:frances is both an instance of ex:Female and of ex:Person.

The rdfs:range property can also be used to indicate that the value of a property is given by a typed literal, as discussed in Section 2.4. For example, if example.org wanted to indicate that the property ex:age had values from the XML Schema datatype xsd:integer, it would write the RDF statements:

```
ex:age   rdf:type   rdf:Property
ex:age  rdfs:range   xsd:integer
```

The datatype xsd:integer is identified by its URI (http://www.w3.org/-2001/XMLSchema#integer). This URI can be used without explicitly stating in the schema that it identifies a datatype. However, it is often useful to explicitly state that a given URI identifies a datatype. This can be done using the RDF Schema class rdfs:Datatype. To state that xsd:integer is a datatype, example.org would write the RDF statement:

```
xsd:integer   rdf:type  rdfs:Datatype
```

This statement says that xsd:integer is the URI of a datatype. Such a statement does not constitute a definition of a datatype, e.g., in the sense that example.org is defining a new datatype. There is no way to define datatypes in RDF Schema. Datatypes are defined externally to RDF and RDF Schema,

and referred to in RDF statements by their URI. This statement simply serves to document the existence of the datatype, and indicate explicitly that is being used in this schema.

The `rdfs:domain` property is used to indicate that a particular property applies to a designated class. For example, if example.org wanted to indicate that the property ex:author applies to instances of class `ex:Book`, it would write the RDF statements:

```
ex:Book    rdf:type    rdfs:Class
ex:author  rdf:type  rdf:Property
ex:author  rdf:type       ex:Book
```

These statements indicate that `ex:Book` is a class, `ex:author` is a property, and that RDF statements using the `ex:author` property have instances of `ex:Book` as subjects.

A given property, say `exterms:weight`, may have zero, one, or more than one domain property. If `exterms:weight` has no domain property, then nothing is said about the resources that `exterms:weight` properties may be used with. If `exterms:weight` has one domain property, say one specifying `ex:Book` as the domain, this says that the `exterms:weight` property applies to instances of class `ex:Book`. If `exterms:weight` has more than one domain property, say one specifying `ex:Book` as the domain and another one specifying `ex:MotorVehicle` as the domain, this says that any resource that has a `exterms:weight` property is an instance of all of the classes specified as the domains, i.e., that any resource that has a `exterms:weight` property is both a `ex:Book` and a `ex:MotorVehicle` (illustrating the need for care in specifying domains and ranges).

As in the case of `rdfs:range`, this last point may not be obvious. However, stating that the property `exterms:weight` has the two domains `ex:Book` and `ex:MotorVehicle` involves making two separate statements:

```
exterms:weight  rdfs:domain          ex:Book
exterms:weight  rdfs:domain  ex:MotorVehicle
```

For any given statement using this property, say:

```
exthings:companyCar  exterms:weight  "2500"^^xsd:integer
```

in order for both the `rdfs:domain` statements to be correct, it must be the case that `exthings:companyCar` is both an instance of `ex:Book` and of `ex:MotorVehicle`.

The use of these range and domain descriptions can be illustrated by extending the vehicle schema, adding two properties `ex:registeredTo` and `ex:rearSeatLegRoom`, a new class `ex:Person`, and explicitly describing the datatype `xsd:integer` as a datatype. The `ex:registeredTo` property applies to any `ex:MotorVehicle` and its value is a `ex:Person`. For the sake of this example, `ex:rearSeatLegRoom` applies only to instances of class `ex:PassengerVehicle`. The value is an `xsd:integer` giving the number of centimeters of rear seat legroom. These descriptions are shown in Listing 2.8:

```
<rdf:Property rdf:ID="registeredTo">
  <rdfs:domain rdf:resource="\#MotorVehicle"/>
  <rdfs:range rdf:resource="\#Person"/>
</rdf:Property>

<rdf:Property rdf:ID="rearSeatLegRoom">
  <rdfs:domain rdf:resource="\#PassengerVehicle"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>

<rdfs:Class rdf:ID="Person"/>

<rdfs:Datatype rdf:about="&xsd;integer"/>
```

Listing 2.8. Some property descriptions for the vehicle schema

Note that an `<rdf:RDF>` element is not used in Listing 2.8, because is assumed this RDF/XML is being added to the vehicle schema described in Example 24. This same assumption also allows the use of relative URI like `#MotorVehicle` to refer to other classes from that schema.

RDF Schema provides a way to specialize properties as well as classes. This specialization relationship between two properties is described using the predefined `rdfs:subPropertyOf` property. For example, if `ex:primaryDriver` and `ex:driver` are both properties, `example.org` could describe these properties, and the fact that `ex:primaryDriver` is a specialization of `ex:driver`, by writing the RDF statements:

```
ex:driver                rdf:type        rdf:Property
ex:primaryDriver         rdf:type        rdf:Property
ex:primaryDriver   rdfs:subPropertyOf      ex:driver
```

The meaning of this `rdfs:subPropertyOf` relationship is that if an instance `exstaff:fred` is an `ex:primaryDriver` of the instance `ex:companyVan`, then RDF Schema defines `exstaff:fred` as also being an `ex:driver` of `ex:companyVan`.

The RDF/XML describing these properties is shown in Listing 2.9.

```
<rdf:Property rdf:ID="driver">
  <rdfs:domain rdf:resource="\#MotorVehicle"/>
</rdf:Property>

<rdf:Property rdf:ID="primaryDriver">
  <rdfs:subPropertyOf rdf:resource="\#driver"/>
</rdf:Property>
```

Listing 2.9. More properties for the vehicle schema

All RDF Schema `rdfs:range` and `rdfs:domain` properties that apply to an RDF property also apply to each of its subproperties. So, in the above example, RDF Schema defines `ex:primaryDriver` as also having an `rdfs:domain` of `ex:MotorVehicle`, because of its subproperty relationship to `ex:driver`.

Listing 2.10 shows the RDF/XML for the full vehicle schema, containing all the descriptions given so far:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd
"http://www.w3.org/2001/XMLSchema\#">]>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns\#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema\#"
  xml:base="http://example.org/schemas/vehicles">

<rdfs:Class rdf:ID="MotorVehicle"/>

<rdfs:Class rdf:ID="PassengerVehicle">
  <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Truck">
  <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Van">
  <rdfs:subClassOf rdf:resource="\#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="MiniVan">
  <rdfs:subClassOf rdf:resource="\#Van"/>
  <rdfs:subClassOf rdf:resource="\#PassengerVehicle"/>
```

```
25 </rdfs:Class>
26
27 <rdfs:Class rdf:ID="Person"/>
28
29 <rdfs:Datatype rdf:about="&xsd;integer"/>
30
31 <rdf:Property rdf:ID="registeredTo">
32   <rdfs:domain rdf:resource="\#MotorVehicle"/>
33   <rdfs:range rdf:resource="\#Person"/>
34 </rdf:Property>
35
36 <rdf:Property rdf:ID="rearSeatLegRoom">
37   <rdfs:domain rdf:resource="\#PassengerVehicle"/>
38   <rdfs:range rdf:resource="&xsd;integer"/>
39 </rdf:Property>
40
41 <rdf:Property rdf:ID="driver">
42   <rdfs:domain rdf:resource="\#MotorVehicle"/>
43 </rdf:Property>
44
45 <rdf:Property rdf:ID="primaryDriver">
46   <rdfs:subPropertyOf rdf:resource="\#driver"/>
47 </rdf:Property>
48
49 </rdf:RDF>
```

Listing 2.10. The full vehicle schema

Having shown how to describe classes and properties using RDF Schema, instances using those classes and properties can now be illustrated. For example, Listing 2.11 describes an instance of the `ex:PassengerVehicle` class described in Listing 2.10, together with some hypothetical values for its properties.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE rdf:RDF [<!ENTITY xsd
  "http://www.w3.org/2001/XMLSchema\#">]>
3 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns\#"
4            xmlns:ex="http://example.org/schemas/vehicles\#"
5            xml:base="http://example.org/things">
6
7   <ex:PassengerVehicle rdf:ID="johnSmithsCar">
8     <ex:registeredTo
      rdf:resource="http://www.example.org/staffid/85740"/>
9     <ex:rearSeatLegRoom
      rdf:datatype="http://www.w3.org/2001/XMLSchema\#integer">127
```

```
10      </ex:rearSeatLegRoom>
11      <ex:primaryDriver
        rdf:resource="http://www.example.org/staffid/85740"/>
12    </ex:PassengerVehicle>
13  </rdf:RDF>
```

Listing 2.11. An Instance of `ex:PassengerVehicle`

This example assumes that the instance is described in a separate document from the schema. Since the schema has an `xml:base` of `http://example.org/-schemas/vehicles`, the namespace declaration `xmlns:ex="http://example.org/-schemas/vehicles#"` is provided to allow QNames such as `ex:registeredTo` in the instance data to be properly expanded to the URIs of the classes and properties described in that schema. An `xml:base declaration` is also provided for this instance, to allow `rdf:ID="johnSmithsCar"` to expand to the proper URI independently of the location of the actual document.

Note that an `ex:registeredTo` property can be used in describing this instance of `ex:PassengerVehicle`, because `ex:PassengerVehicle` is a subclass of `ex:MotorVehicle`. Note also that a typed literal is used for the value of the `ex:rearSetLegRoom` property in this instance, rather than a plain literal. Because the schema describes the range of this property as an `xsd:integer`, the value of the property should be a typed literal of that datatype in order to match the range description. Additional information, either in the schema, or in additional instance data, could also be provided to explicitly specify the units of the `ex:rearSetLegRoom` property (centimeters).

2.2.6 Building the Logic: OWL

The first level above RDF is an ontology language that can formally describe the meaning of terminology used in Web documents. If machines are expected to perform useful reasoning tasks on these documents, the language must go beyond the basic semantics of RDF Schema.

Ontology Web language (OWL) is part of the growing stack of W3C recommendations related to the Semantic Web, has been designed to meet this need for a Web Ontology Language.

OWL provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users.

- **OWL Lite** supports those users primarily needing a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives, and OWL Lite provides a quick migration path for thesauri and other taxonomies.

Owl Lite also has a lower formal complexity than OWL DL, see the section on OWL Lite in the OWL Reference for further details.

- **OWL DL** supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL.
- **OWL Full** is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

In the following is given a synopsis of the OWL versions previously described. The OWL constructs won't be detailed here not to encumber the discussion. Further details can be found in the reference documentation [MvH04].

2.2.6.1 OWL Lite Synopsis

RDF Schema Features	**(In)Equality**
Class	equivalentClass
`rdfs:subClassOf`	equivalentProperty
`rdfs:Property`	sameAs
`rdfs:subPropertyOf`	differenFrom
`rdfs:domain`	AllDifferent
`rdfs:range`	distinctMembers
Individual	

2.2.6.2 OWL DL and OWL Full Synopsis

The list of OWL DL and OWL Full language constructs that are in addition to or expand those of OWL Lite is given below.

Property Characteristics
ObjectProperty
DatatypeProperty
inverseOf
TransitiveProperty
SymmetricProperty
FunctionalProperty
InverseFunctionalProperty

Property Restrictions
Restriction
onProperty
allValuesFrom
someValuesFrom

Restricted Cardinality
minCardinality (only 0 or 1)
maxCardinality (only 0 or 1)
cardinality (only 0 or 1)

Header Information
Ontology
imports

Class Intersection
intersectionOf

Datatypes
xsd datatypes

Versioning
versionInfo
priorVersion
backwardCompatibleWith
incompatibleWith
DeprecatedClass
DeprecatedProperty

Annotation Properties
`rdfs:label`
`rdfs:comment`
`rdfs:seeAlso`
`rdfs:isDefinedBy`
AnnotationProperty
OntologyProperty

Class Axioms

Class
oneOf, dataRange
disjointWith
equivalentClass(applied to class expressions)
`rdfs:subClassOf` (applied to class expressions)

Boolean Combinations of Class Expressions
equivalentClass
unionOf
complementOf
intersectionOf

Arbitrary Cardinality
minCardinality
maxCardinality
cardinality

Filler Information
hasValue

2.2.7 Querying the Graph: SPARQL

SPARQL Protocol and RDF Query Language (whose SPARQL is the recursive acronym) is a query language for RDF that is used to express queries across different data sources, whether data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The outcome of SPARQL queries can be results sets or RDF graphs. SPARQL shows some similarities compared with SQL, especially as regards the keywords and the overall syntax. According to the semantic technologies discussed so far, SPARQL also adopts URIs as major identifiers and this is a substantial difference with respect to the relational databases where primary and foreign keys are used. SPARQL needs two inputs:

- the SPARQL query
- the data graph

A SPARQL query uses *triple patterns* to describe what subset of the data graph is requested. Triple patterns are very similar to RDF statements but they introduce variables to achieve the proper degree pf flexibility. The result of a query is a solution sequence, corresponding to the ways in which the query's graph pattern matches data. There may be zero, one or multiple solutions. SPARQL is a very comprehensive language and its deep discussion is off topic here. To give an idea of the capabilities of the language we summarize the most basic available keywords and actions. If the reader is interested in a more detailed discussion he/she can refer the official documentation [PS08b].

2.2.7.1 SPARQL, in a Nutshell

To give a sketch of the SPARQL language, the [FP13] tutorial presentation by two members of the W3C SPARQL Working Group is chosen as guidance. A SPARQL query comprises, in order:

- Prefix declarations, for abbreviating URIs;
- A result clause, identifying what information to return from the query;
- Dataset definition, stating what RDF graph(s) are being queried;
- The query pattern, specifying what to query for in the underlying dataset;
- Query modifiers, slicing, ordering, and otherwise rearranging query results.

Fig 2.6 shows the anatomy of a SPARQL query, according to the aforementioned list points. The prefix declarations are identified by the `PREFIX` statement, the result clause is identified by the `SELECT` statement, the dataset definition is identified by the `FROM` statement, the query pattern is identified by the `WHERE` statement and finally at the bottom of the figure appear query modifiers.

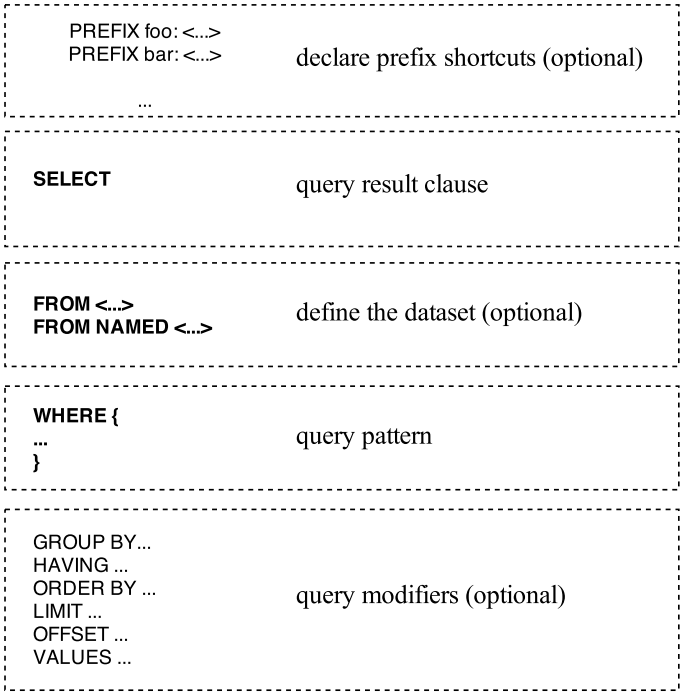


Figure 2.6. *The anatomy of a SPARQL query*

SPARQL queries are executed against RDF datasets, consisting of RDF graphs and a SPARQL endpoint accepts queries and returns values via HTTP. There are also two different kinds of endpoints: a generic endpoint will query any Web-accessible RDF data while a specific one is bound to query against specific datasets. Results of a query can be returned in a variety of formats such as XML, JSON, CSV, RDF and HTML.

In the following example a dataset is queried asking to find all subjects (`?person`) and objects (`?name`)put in relation via the `foaf:name` predicate. The `SELECT` statement precises that names only must be displayed.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name
3 WHERE {
4     ?person foaf:name ?name .
```

```
5 }
```

SPARQL variables start with a `?` and can match any node (resource or literal) in the RDF dataset. Triple patterns contained in the `WHERE` clause are just like triples, except that any of the parts of a triple can be replaced with a variable. Finally, the `SELECT` clause returns a table of variables and values that satisfy the query. To retrieve multiple properties about a particular resource is possible to use multiple triple patterns, as shown in the following example.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name
3 WHERE {
4     ?person foaf:name ?name .
5     ?person foaf:mbox ?email .
6 }
```

SPARQL queries also support a filtering mechanism associated with the keyword `FILTER`. In the following example such filter is declared in the query that assumes the meaning of *find all landlocked countries with a population greater than 15 million, with the highest population country first*. Please note that `lang` extracts a literal's language tag, if any.

```
1  PREFIX type: <http://dbpedia.org/class/yago/>
2  PREFIX prop: <http://dbpedia.org/property/>
3  SELECT ?country_name ?population
4  WHERE {
5      ?country a type:LandlockedCountries ;
6               rdfs:label ?country_name ;
7               prop:populationEstimate ?population .
8      FILTER (?population > 15000000 &&
9             langMatches(lang(?country_name), "EN")) .
10 } ORDER BY DESC(?population)
```

As mentioned before, SPARQL capabilities go far beyond the ones discussed so far. The reader is invited to refer to the official documentation for a deeper insight.

2.3 Linked Data and Semantic Web Criticisms

Linked Data and Semantic Web concepts introduced a new way of conceiving information on the Web. The aim is twofold: on one hand the information has to be directly accessible, on the other hand its intrinsic expressiveness is pushed

forward by defining a framework able to represent the semantic relations between information grains. The expected result is a dense Web of Data whose meaning can be extrapolated by the topology of the Web itself. Because of its semantic structure, this graph of connected data unfolds an incredible amount of information. Indeed, is possible to perform very articulated queries and it also supports cutting edge features such as inferences.

Unfortunately, all this power comes with a considerable overall system complexity. The technologies implementing the Linked Data vision are manifold and their adoption is not trivial at all.

As some reports point out [GGP+10, HDS06, HP02, Jam06] the critical issues span from the usability of applications to complexity in querying, from complexity of the information model to information redundancy. Indeed, when Tim-Berners Lee declared the Semantic Web open for business critics said that RDF standard were too complex and difficult to implement, that named entity mark-up was too labor intensive to be practical, and that creating agreed-upon ontologies to model all the world's knowledge was so gargantuan a task as to be impossible [GB10].

In [GB10] authors argue that the exposure of data as RDF represents a considerable challenge for enterprises and such conversion has become a well-defined problem within the Linked Data community. To this end, many tools for easing the automatic conversion of existing documents in RDF have been released with time. However, only a very small proportion of organizations have made efforts to adopt semantic technologies, and even Tim Bereners-Lee admits that the machine readable Web is still way off.

In [JV09] authors argue that although many Semantic Web related technologies have emerged, much remains to be done. The process of converting the Web's existing unstructured content into a format understandable by machines isn't a trivial or generally solvable task. Another challenging task of ontology engineering is the integration of multiple ontologies into a common one for Web sources and consumers in a domain. Despite the W3C efforts in defining and standardizing the upper layers of the W3C Semantic Web architecture model that refers to logic, inference and reasoning, the research community continues to come out with new Semantic Web languages. Moreover, various newly developed reasoning languages aren't easily supported on existing commercial tools and technologies come from open source communities.

All these complexities are likely to discourage developers of the Semantic Web by worsening the steepness of the learning curve and affecting the widespread of the paradigm. While dealing with data interoperability, the mass effect is highly relevant.

We can borrow an example from the history of telecommunications: the telephone is an invention whose usefulness is definitely related to its diffusion. The value of owning a telephone is directly proportional with the number of people

owning the same device. Probably, if he hadn't had such diffusion it wouldn't have changed the history of telecommunications.

The same considerations apply to the Linked Data and Semantic Web paradigms. That's why the complexity factor shouldn't be underestimated when introducing new technologies and users should be supported in taking advantage of them as much as possible.

Chapter 3

Representational State Transfer

The REpresentational State Transfer , or REST, is an architectural style described in the doctoral dissertation by the author Roy T. Fielding [Fie00a]. It can be considered the Web architectural style since has been conceived during the drafting of the HTTP protocol [FGM+99] of which Fielding is a coauthor. REST is an abstract model while the Web is a concrete architecture that is based on the principles of this model. The model is defined by a set of architectural constraints that may be applied to make systems *RESTful*.

The Web is RESTful not only because meets these constraints, but also because is the specific case-study from which they have been extrapolated.

The first constraint to be fulfilled by a RESTful architecture is the *client-server* interaction between components: the system is made up of server components listening for requests and client components issuing these requests by leveraging *connectors*. A connector is defined as "an abstract mechanism that mediates communication, coordination, or cooperation among components", i.e. en element that encapsulates the resource access activities as well as the transfer of their representations.

A second constraint is the statelessness of interactions between clients and servers. Fielding arguments the constraint as follows: "the client-stateless-server style derives from client-server with the additional constraint that no session state is allowed on the server component. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is kept entirely on the client.". As a drawback, the information exchanges between clients and server can be affected by a performance degradation due to the overhead

caused by the repetition of data. On the other hand, the enforcement of this principle brings a number of advantages: first, it improves the visibility because an interaction which keeps all the information can be easily monitored. Second, it supports scalability, in fact a server which is not forced to keep the state of every interaction with clients, can serve a large number of them without being significantly affected. This is considered one of the major benefits of the RESTful implementations.

To enhance efficiency, RESTful systems can take advantage of caching mechanisms. This means that a client can store responses from the server and use them instead of issuing new requests. It is worth noting that not all resources can be cached because they may change with time without the client is aware of the modification. To this end, resources should be explicitly or implicitly labeled as *cacheable* or *non-cacheable*.

Another peculiar characteristic of REST is the *uniform interface*. Fielding introduces the concept as follows: "The central feature that distinguishes the REST architectural style from other network based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability.

The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an applications needs [...]. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components.

REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self descriptive messages; and, hypermedia as the engine of application state.".

Therefore, this principle defines other constraints on functionalities provided by the interface of a RESTful system. The identification of resources and their management have to happen by the means of a *representation*. A resource is any information provided with a name, such as a document, an image, an actual object (a person, an animal, an house). In REST, *resource identifiers* are used to identify resources involved in interactions between clients and servers which perform actions on them using a resource representation to capture the state.

Finally, a REST architecture is *layered*. This allow to meet scalability requirements of huge network systems such as the Internet. Enforcing this principle means to promote the loose coupling between architectural components and independence of non adjacent layers.

3.1 REST Principles

Regardless of the interpretation of the original definition or subsequent derivations, in synthesis REST must have the following basic properties:

1. Representation Oriented
2. Addressability
3. Uniform Interface
4. Statless Communications
5. Hypermedia as the Engine of Application State (HATEOAS).

From now on, HTTP is assumed as the reference implementation for the REST architectural style. Clearly, being an abstract style, it is not bound to any specific technology. However, according to the purposes of this thesis it is useful to get more concrete and dig into the technologies meaningful for the cases of interest. Therefore, HTTP and all the surrounding technologies such as, URIs, and the different file formats such as HTML, XML, JSON, JPG, etc. will be used.

In the following, excerpts and examples from [Til07] are used to present principles 1. and 3., excerpts and examples from [Sle10] are used to present principle 2., excerpts and examples from [Rot09] are used to present principles 3. and 4., finally, principle 5. has been extracted from [ASJH11].

3.1.0.2 Representation Oriented

A key point in the client-server interaction concerns the handling of the retrieved resource, i.e. clients and servers should be capable of processing resources sent by other system components. The approach taken by HTTP is to allow for a separation of concerns between handling the data and invoking operations. In other words, a client that knows how to handle a particular data format can interact with all resources that can provide a representation in this format.

For example, by leveraging the HTTP content negotiation, a client can ask for a representation in a particular format:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

The result might be some company-specific XML format that represents customer information. If the client sends a different request, e.g. one like this:

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

the result could be the customer address in VCard format. In the example the responses are omitted, but they would contain metadata about the type of data in the HTTP Content-type header.

This illustrates why ideally, the representations of a resource should be in standard formats. If a client "knows" both the HTTP application protocol and a set of data formats, it can interact with any RESTful HTTP application in a very meaningful way. Unfortunately, a standard formats for everything is not available, but probably is possible to imagine how one could create a smaller ecosystem within a company or a set of collaborating partners by relying on standard formats.

Of course all of this does not only apply to the data sent from the server to the client, but also for the reverse direction, i.e. a server that can consume data in specific formats does not care about the particular type of client, provided it follows the application protocol.

There is another significant benefit of having multiple representations of a resource in practice: If an HTML and an XML representation of a resources are both available, they are consumable not only by a paired application, but also by every standard Web browser. In other words, information in the application becomes available to everyone who knows how to use the Web.

There is another way to exploit this: It is possible to turn the applications Web uniform interface into its Web API. After all, API design is often driven by the idea that everything that can be done via the uniform interface should also be doable via the API. Conflating the two tasks into one is a useful way to have a better Web interface for both humans and other applications.

3.1.0.3 Addressability

REST requires that every resource has a name for its identification. To this end, the HTTP implementation of REST leverages URIs (or URLs). A URI is a name for information. Names are how humans identify people, places, things and concepts. If the ability to identify is missing, the ability to signify is missing as well. Having names gives the ability to disambiguate and identify something within a context. Having a name and a common context allows to make reference to named things out of that context. The Uniform Resource Identifier (URI) is the parent scheme. It is a method for encoding other schemes depending on whether resolution information are requested or not.

If the context of a reference systems and information does not changed, resolution information in resource names are very useful. Examples of such names are the followings URLs:

```
http://someserver.com/cgi-bin/foo/bar.pl
http://someserver.com/ActionServlet?blah=blah
http://someserver.com/foo/bar.php
```

The problem with these URLs is that the technology used to produce a result is irrelevant to the consumer of information. In REST it is very advised to keep the focus on the information rather than the technology. In fact, implementation technologies change over time and if they are abandoned, for instance, any system that has a link to the Perl, Servlet or PHPbased URL will break. Despite being fragile, the URL scheme does allow us to disambiguate information references in a global context.

```
http://company1.com/customer/123456
```

is distinct and distinguishable from

```
http://company2.com/customer/123456
```

in ways that a decontextualized identifier like "123456" is not. To ground the concept into a larger information systems framework, a URL can be thought as a primary key that is not specific to a particular database. Using URLs is possible to make references to an item in different databases, documents, applications, etc. and know that same thing is referred because the URL is a unique name in a global context.

The next aspect of URLs that bears discussion is their universal applicability. URLs provide a common naming scheme that allows us to identify documents, data, services, concepts. A careful application of specific guidelines allows to blur the distinctions between these things. Not only are these names useful in order to refer to information, but systems that receive these references can simply ask for them. The "L" in URL (locator) gives the capacity to resolve the object, not knowing anything else about it. Issuing a GET request to a URL representing a document, some data, a service to produce that data or an abstract, *non-network-addressable* concept all work fundamentally the same way. It is possible to manipulate, create, modify or delete these objects using similar means.

A very interesting analysis on the topic can be found in [Bry12] where the author defines some characteristics which are desirable for a RESTful URL. In the following are reported some of them. According to the author, a good REST URL should be:

Short as possible This makes them easy to write down or spell or remember.

Hackable "up the tree" The user should be able to remove the leaf path and get an expected page back, e.g. `http://.../cars/alfa-romeos/gt` the user could remove the `gt` bit and expect to get back all the alfa-romeos.

Meaningful The URI describes the resource. The user should have a hint at the type of resource he/she is looking at (a blog post, or a conversation).

Nouns, not verbs It complies with the role of URIs in RESTful architecture. They identify resources not actions.

Stateless The state of the resource should not be bound to the URL.

Report canonical URIs If two different URIs are available for the same resource, the canonical URL should be put in the response.

3.1.0.4 Uniform Interface

To simplify the overall system architecture the REST architecture style includes the concept of a Uniform Interface. The Uniform Interface consists of a constrained set of well-defined operations to access and manipulate resources. The same interface is used regardless of the resource. If the client interacts with a `Hotel` resource, a `Room` resource or a `CreditScore` resource the interface will be the same.

The Uniform Interface is independent to the resource URI. No IDL-like files are required describing the available methods. The interface of RESTful HTTP is widely used and very popular. It consists of the standard HTTP methods or *verbs* such as GET, PUT or POST which is used by applications to retrieve pages and to send data.

The set of standard methods includes GET, POST, PUT, DELETE, HEAD and OPTIONS. The meaning of these methods is defined in the HTTP specification, along with some guarantees about their behavior. It is possible to imagine that every resource in a RESTful HTTP scenario extends a class like this (in some Java pseudo-syntax and concentrating on the key methods):

```
1 class Resource {
2         Resource(URI u);
3         Response get();
4         Response post(Request r);
5         Response put(Request r);
6         Response delete();
7 }
```

Because the same interface is used for every resource, it is possible to retrieve a representation using GET. Because GETs semantics are defined in the specification, no obligations are required when it is invoked. This is why the method is called

"safe". GET is idempotent, meaning that it is possible to issue a GET request many times and with the exact same effect. Idempotence is also guaranteed for PUT (which basically means "update this resource with this data, or create it at this URI if its not there already") and for DELETE. POST, which usually means "create a new resource", can also be used to invoke arbitrary processing and thus is neither safe nor idempotent.

3.1.0.5 Stateless Communications

A RESTful HTTP interaction has to be stateless. This means each request contains all information which is required to process the request. The client is responsible for the application state. A RESTful server does not have to retain the application state between requests. The Server is responsible for the resource state not for the application state. Servers and intermediaries are able to understand the request and response in isolation. Web caching proxies do have all the information to handle the messages correctly and to manage their caches.

This stateless approach is a fundamental principle to implement high-scalable, high-available applications. In general statelessness enables that each client request can be served by different servers. A server can be replaced by another one for each request. As traffic increases, new servers are added. If a server fails, it will be remove from the cluster.

3.1.0.6 Hypermedia as the Engine of the Application State

Hypermedia as the engine of application state means that neither client nor server needs to keep the state of the exchange in a session, because all the necessary information is stored in the exchanged HTTP messages (in the URI and the accompanying HTTP headers). Defining self-contained links is critical for RESTful Web services, because these links make it possible to traverse, discover, and connect to other services and applications. However this is difficult, because complex interactions translate to complex URIs. Complex applications have many states that the client needs to be aware of.

HATEOAS forces Web services to expose the states as links, which duplicates the internal design of the service. Thats why many RESTful Web services resort to exposing the underlying API of the service even if they know its wrong. Many services require the client to send user-specific information (e.g. user-id) in every request URI. As a result, the same requests from two different clients appear unique to the Web caches, because caches use URIs as keys for the data. Sending user-specific information is often unnecessary (especially when the user sends a generic query), but its used extensively byWeb services providers to limit the number of accesses from each client. Since HTTP caching cannot be used in this case, the service must handle more requests, which defeats the purpose of rate limiting. This seemingly innocuous (but often occurring) lapse violates

two principles the identification of resources and HATEOAS; it also affects cacheability.

3.1.1 Making the Point on REST

Since the REST architectural style was introduced to Web communities, its popularity has grown at a very fast peace. Nowadays many application claiming to be RESTful have been released as well as frameworks for their developments such as Jersey (Java), RESTEasy (Java), Restlet (Java), Spring (Java), Recess (PHP), Routes (Python), CherryPy (Python), Django (Python), RESTfulie (Ruby, Java), Ruby on Rails (Ruby). Moreover, several giant enterprises such as Google, Facebook and Twitter have chosen a RESTful approach in providing services instead of the WS-* [CLS+05] counterpart.

Despite this tremendous interest, many scientists agree that this style has not been well acknowledged yet. In 2011, Adamczyk *et al.*, published an article [ASJH11] where they evaluated some key questions regarding the real and perceived distinctions between REST and WS-* styles, by carrying out a very thorough analysis of Web services exposed by four repositories:

1. xmethods.net
2. webservicex.net
3. webservicelist.com
4. programmableweb.com

Tab. 3.1 is extracted from this study and shows common Web service styles in different repositories. Despite in the article results from 2007 and 2010 are compared, here, the most recent results only (i.e. from 2010) are reported.

Style	**1**	**2**	**3**	**4**
RESTful	0	0	144	1627
WS-*	382	70	259	368
XML-RPC	0	0	21	53
JavaScript/AJAX	0	0	9	130
Other	0	0	26	77
Total (unique)	382 (382)	70 (70)	459 (386)	2255 (2179)

Table 3.1. Web services styles used in public services (year 2010), according to [ASJH11]. Some services are available in two or more styles, the number of unique services are shown in parentheses.

The study also discusses the most common mistakes made in implementing REST services. As regards the identification of resources, authors argue that the most blatant violation of the principle is a Web service defining a single resource where methods calls are openly included in the URI, as in the case of Digg where, in order to access the news, the following URI is used.

```
http://service.digg.com/2.0/user.getMyNews
```

In fact, the URI identifies only one resource and all specific requests are encoded as strings representing a method call (e.g., `getMyNews`).

As regards content negotiation, only one Web Service was found to support it, meaning that such capability is often neglected. Considering the uniform interface principle, authors report that some services defined GET for sending all requests to resources, even if the request had side effects. For example, initially Bloglines, Flickr and Delicious Web service defined GET for making updates. Other services specified that clients could use GET and POST interchangeably, which is a clear violation of the principle.

Some consideration have been made also on available tools for building RESTful services. Although a large number of tools is released, this does not mean that the offered support is actually appropriate. In [ZS12] authors investigate this direction and argue that frameworks still lack an adequate guidance for incorporating REST principles. To this end, they provide practical guidelines for designing frameworks for developing RESTful systems. They argue that frameworks should provide a greater separation of concerns to increase reusability and modifiability. In addition, since complex engineering disciplines should utilize theoretical foundations for practical guidance, frameworks should use simple formal models to provide abstractions that encapsulate REST principles and steer the development process.

These design guidelines are summarized and grouped by theme in the following list.

- frameworks should support system modifiability, so developers can easily export, import, and change any architecture and application element definition;
- frameworks should support the implementation of multiple application-level protocols, such as HTTP, and their simultaneous use;
    - to support separation of concerns, a framework should promote the modularization of protocol implementation definitions, which should consist of the supported request operations, response codes, and possible header names and values;

    - a protocol implementation should contain protocol deserializers and serializers, which expose message elements from a byte stream, and vice versa;

– frameworks should support the implementation of resource identifier namespaces, such as the URI namespace, as well as identifier templates;

    - frameworks should implement template engines that parse and generate identifiers using templates and template variables.

– frameworks should also support the implementation of different media types and their simultaneous use;

– frameworks should support content negotiation for choosing the media type for response representations, based on client preferences and server capabilities.

Nevertheless, in a study led in 2012 [GIM12] involving 69 computer novice developers (science students) in implementing HTTP based client applications exploiting the WS-* and REST approaches, results reported that REST is statistically significantly easier and faster to learn than WS-*. Moreover, a significant number of volunteers agreed that REST was more adapted for Web applications requiring to integrate Web content: "*[for] Web Mashups, REST services compose easily*".

In conclusion REST is still a controversial topic. It is widely accepted that is very useful for the Web, and even reasonably easy to learn and use. However, its implementation still presents problems since most RESTful services are not designed diligently and neglect to follow principles in various ways [ASJH11].

Part II

Discussion of the Work

Chapter 4

InterDataNet Modelling

InterDataNet (IDN) is a framework designed, developed and implemented in the context of this thesis, on the basis of previous works [Inn08, Chi09, Cio10]. Its main purpose is to produce a viable option for the consumption and production of aggregated information, still preserving a substantial ease of use. From the InterDataNet perspective, first class entities are documents (or IDN-Documents) which are graphs of related information pieces. Upon them CRUD operations are enabled with the aim of creating a *read/write* Web of Resources.

In literature the Web of Documents is described as a global space made of HTML pages interlinked through hypertext links, and is opposed to the Web of Data which is a global space containing billions of assertions, putting data in relation through typed links [BHBL09]. During the course of the chapter it will become evident that the definition of Web of Document is too restrictive for InterDataNet resources. Indeed, an IDN-Document is a document made up of information pieces which are individually addressable, manageable and reusable to build new documents and enrich existing ones. A comprehensive set of operations (read, write, delete) are enabled for each information piece, creating something more than an ordinary HTML page, which is the typical representative of the Web of Documents.

However, even though the IDN-Document information pieces could contain single data units, the amount of metadata required by the architecture for each of them is significant, making such representation quite inefficient. Moreover, even though InterDatNet supports relations, they are not comparable with the semantics of the RDF. InterDataNet is designed to pick information pieces (less granular than a single data unit), enable collaboration oriented properties, and

aggregate them to build documents. To this end, I feel more comfortable to associate it with the Web of Document concept, considering it as an "enhanced" representative of this family, provided that consideration is given of its strong Resource oriented connotation.

The InterDataNet framework has three main components:

1. the Information Model
2. the Middleware architecture
3. the IDN Tool-suite development tools

the Information Model defines the rules by which the information is represented in InterDataNet. Basically, it is a formalism to describe the information graph including vertices, edges and their properties. The Middleware architecture is a layered, modular, Web-oriented system enabling the properties defined in the Information Model on the InterDataNet resources. It heavily leverages the REST [Fie00b] paradigm both for the exposure of APIs and for internal communications' exchanges. The development tools constitute a tool-suite for the easy production and consuming of InterDataNet resources as well as the simple development of InterDataNet compliant Web applications. The relevant elements are Java and Javascript libraries and two client-side Web applications for the visual management of information graphs and for their automated HTML rendering.

Fig.4.1 shows a sketch of the InterDataNet framework. The Middleware manages the InterDataNet resources exchanged with the applications in the graph form defined by the Information Model. This exchange is mediated by the InterDataNet Tool-suite in favor of users, to mask the operation complexity.

Even though this schema is conceptual and it is not intended to be exhaustive, it still captures the essence of the system components' relations.

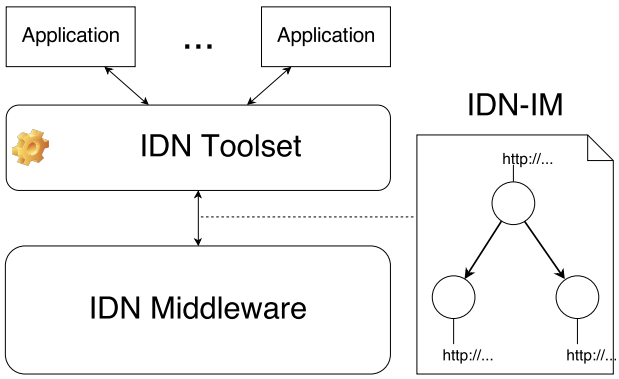


Figure 4.1. *The conceptual scheme of the InterDataNet framework.*

4.1 The InterDataNet Information Model

The InterDataNet Information Model (IDN-IM) is the set of rules that drive the representation of entities in InterDataNet. The Information Model describes the entities from a high theoretical point of view, without taking in consideration implementation details. The document concept is a first class entity of the InterDataNet Information Model (IDN-IM). It is a set of information put in relation through the aggregation concept, and is represented as a graph of uniquely addressed information pieces called IDN-Nodes. Since links between data specify containment relations, every subset of connected vertices can be seen as an IDN-Document. Different IDN-Documents can be combined to form richer documents, just the way a document containing personal data can be aggregated to a document containing descriptions of skills to create a personal *curriculum vitae*.

A suggestive metaphor for the IDN-Document is the construction toy. You have primitive elements in the form of plastic bricks that can be creatively interlocked to build bigger things. The same brick is good to build a house, a boat, a furniture and can be reused at will. In the physical word, once a brick is part of a wall it can't be used to build something different.

The virtual world is not subject to this constraint, since data can be replicated with ease. The same brick can be used to build a boat *while* is being part of a wall. However, all that glisters is not gold: data replication introduces problems in terms of replicas' consistency and synchronization. The Web provides a very elegant solution for this with the concept of *pointer to a resource*.

Basically, is a dereferencable name of the location the resource is hosted in. Given the pointer, is possible to reach the resource in the global space without needing to replicate it locally: indeed the resource is collected directly from the source, as required. This strategy is particularly attractive because it promotes the distribution of responsibilities. Provided that the source is trusted and entitled to deliver the resource, it is possible to transfer some guarantees to the resource itself. For example, if a public employee is looking for the age of a citizen, he/she will trust more the information provided by the registry office of the municipality, rather than the information retrieved form a social network. The technology defining dereferencable pointers on the Web is mature and very widespread: it is a subset of the Uniform Resource Identifier (URI) [BLFM05], the Uniform Resource Locator (URL) [Hof05b, Hof05a]. By using URLs two objectives are pursued at the same time: a global name is given to the resource, and information concerning the location the resource is hosted in, including the authority responsible for the delivery, are embedded in that name.

4.1.1 The IDN-Document

Although the term document occurs very often, its definition is not so trivial. In 1937, the International Institute for Intellectual Cooperation (a League of Nations agency) defines the document to be *any source of information, in material form, capable of being used for reference or study or as an authority. Examples: manuscripts, printed matter, illustrations, diagrams, museum specimens, etc* [Buc97].

Similarly, the International Standard Organization (ISO) defines a document as the ensemble of data and physical supports on which they are recorded, that generally is the permanent and human accessible form.

With the coming of the digital age, the possibility of processing documents takes room and documents enter the realm of immaterial [RMS97]. Generally speaking, a document is an abstraction used to indicate a *structured collection* of information. Humans are very familiar to deal with documents, probably because of their natural aptitude in categorizing things. In the everyday life is quite common to find examples of information structured in document form, for human comprehension: books, invoices, calendars, archive records, driving licenses, ID cards, passports, etc. At the basis of the document abstraction is the ability of structuring information, i.e. discriminating the contents from the container, the similar from the different, the inside from the outside. This paradigm applies to all documents mentioned above.

If we consider the driving license depicted in Fig.4.2 (which is a close reproduction of the one currently in force in Europe) we note is basically a box containing information concerning the owner's personal data and his/her driving capabilities. These two groups can be further divided to reach more detailed information, such as the name of the owner or the type of vehicle he/she is entitled to drive.

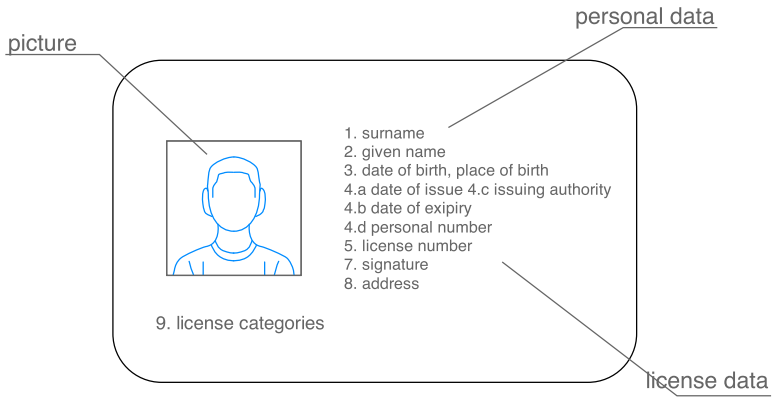


Figure 4.2. *A real world example of document: a driving license.*

Fig.4.3 shows a possible representation of the driving license structure. Even though the sample does not claim to be complete or exhaustive, it brings out the high level of information organization characterizing documents. Indeed, the driving license can be though as a box containing personal and license related information. The former can be either seen as a box containing individual related information such as name and surname, and address information such as ZIP code, street and street number. The example stops at this level of granularity, but the refinement procedure could be easily iterated.

The driving license is a small card, but the information it contains are many and various. Nevertheless, we handle it daily, and the fact they are structured allow us to use it with negligible effort.

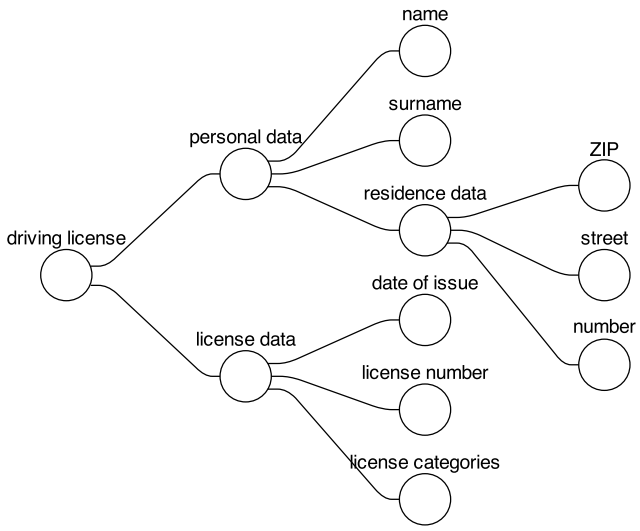


Figure 4.3. *Information structuring of a driving license.*

A last consideration closes the discussion about the natural attitude of humans in exploiting the document abstraction to organize information. Nowadays, the Web pages are major media of information. Is well known that a good design helps people in consuming such information. On the other hand, a chaotic design results to be confusing. This happens because people expect that the information is well structured, adhering to a coherent displacement in the available space within the page. Otherwise, the information can be hidden, decontextualized or at least very hard to consume. Is worth noting that even the language used for Web pages construction is highly structured. HTML [JHR99] defines a set of keywords with the intent of generating containers and contents, and therefore organizing elements within the page.

Documents have an additional interesting property: their meaning derives from the information of which they are composed. Returning to the driving license example, is clear that the essence (and the value) of the license comes from

contained personal data (to bind the license to the driver, the driver to a country location, a residence, etc.) and other data specific for that license (to identify it uniquely, to make it valid for certain vehicles, etc.).

For interesting insights about the notion of document through times, the reader is encouraged to see [Buc97].

For what discussed so far, it is possible to highlight two characteristics of interest for documents:

- humans are very familiar in organizing information according to a container-content hierarchy;
- the meaning of a document is highly related with the information it contains.

Documents, intended as structured aggregate of information pieces, are main entities in InterDataNet. The reason for this, is that a desirable property for the InterDataNet Information Model is to enable people to have a smooth interaction with the framework. The first thing to do in this direction is to define very simple resources, and very simple operations to handle them. Managing InterDataNet resources shouldn't be complicated or cumbersome. Since one of the key objective of the project is to build a very low-entry level barrier framework, it should be intuitive, instead.

Basically, InterDataNet resources are graphs of information grains. Each graph can be made up of one or more vertexes, eventually connected by edges of different nature. Actual data lie inside vertexes and their granularity level is up to the data publisher. Similarly to the RDF model, InterDataNet Information Model provides blank nodes as pivotal connection points of differently specified information.

More graphs can be combined together in order to build a bigger graph, and this procedure can be iterated up to reach a global graph of information. A global identifier is associated with each vertex to make it individually addressable, in a global space.

Fig. 4.4 shows a possible document view of the driving license (it is not a complete representation of the actual document, and it slightly differs from the one described in Fig. 4.3). In the picture, the dashed boxes surround documents. This perspective shows that a document can be composed from different documents, composed from other documents in their turn. Not to complicate the view, leaves of the tree structure have not been wrapped in dashed boxes, even if they could be considered standalone documents, in fact.

In the following, core InterDataNet Information Model concepts will be introduced.

IDN-Document An IDN-Document is a directed graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. The elements of V and E are the

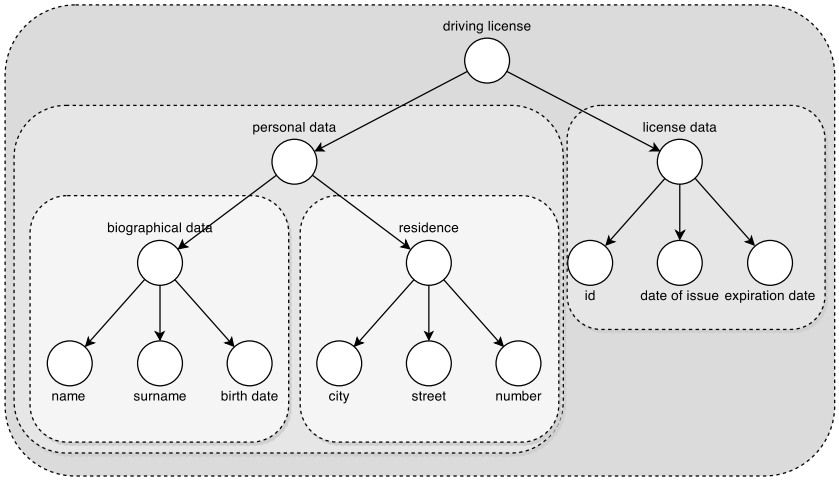


Figure 4.4. *A document view of the driving license.*

nodes containing the granular information (IDN-Nodes) and the relations between IDN-Nodes, respectively.

IDN-Node An IDN-Node is a tuple $T = C, P$, where C is the set of content elements (i.e., data) and P is the set of properties (i.e., metadata) that characterize C.

Aggregation Link The Aggregation Link represents a container-content relation. The node where the edge starts from aggregates and therefore contains the node the edge points to.

Reference Link The Reference Link represents a pointer towards the referred resource. No further meanings are associated with it. To better understand the Reference Link role, it could be somehow compared with the HTML `href` attribute.

Since Aggregation Links define a container-content relation, it shall not be possible to identify cycles following Aggregation Links within an IDN-Document. If this happens, we will be faced with a paradox where an object is either the content and the container of another object. In other words, an IDN-Document is a Direct Acyclic Graph (DAG) with respect to Aggregation Links. The same constraint does not apply for Reference Links because it is legitimate for two nodes to refer each other.

Let D be an IDN-Document modeled as a graph G. Hence, the topology of G expresses the IDN Information Model used to represent D. Through the IDN Information Model is possible to define an IDN-Document as an aggregation of data provided by different information sources. Indeed, an IDN-Node can

be referred to by more than one IDN-Documents, thus favoring the reuse of information across different applications. This is possible since each IDN-Node is associated to an information provider that is authoritative for the information the IDN-Node refers to. Hence, gathering information from the proper sources enforces a responsibility distribution across the information providers, who are responsible for the quality of the provided information. In addition, IDN Information Model can be extended with metadata enforcing properties such as privacy, licensing, provenance, consistency, availability, attached to IDN-Nodes and affecting IDN-Documents. Such features are crucial to support effective and trusted collaboration on real world scenarios. Fig.4.5 shows the Information Model schema.

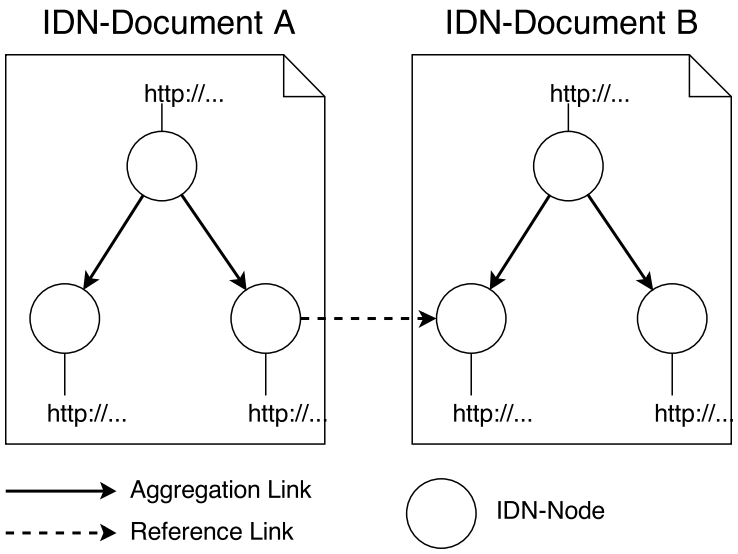


Figure 4.5. *The InterDataNet Information Model schema.*

In addition to these types of edges, a convenience link is also defined:

Back Link The Back Link represents a content-container relation. The node where the edge starts from is the content of the node the edge points to.

The Back Link is the dual of the Aggregation Link, and is used to identify the container from a content, or analogously, a parent from a child. This link can be considered a minor element because, from a conceptual point of view, it doesn't add much to the model. Nevertheless, it has proved to be pretty useful for maintenance issues and so its interest is more in implementation rather than modeling.

4.1.2 The IDN-Node

The IDN-Node is at the heart of the information model. From a graph point of view it is the junction point towards other resources, but it also keeps the information, i.e. the actual data, and the metadata enforcing critical properties such as provenance, licensing or versioning.

IDN-Nodes are identified by logical names called LRIs (Logical Resource Identifier) which are URIs. LRIs referring to the same resource are equivalent in terms of support for operations on the resource. The InterDataNet Information Model allows two classes of LRIs: canonical and alias. The canonical LRI is assigned to the IDN-Node at the time of its creation and is unique. During the life cycle of the IDN-Node, more aliases can be created for reasons of convenience.

Conceptually, the IDN-Node is made up of three major sets of information.

- contents, i.e. data provided by the information source or the publisher;
- links pointing to other IDN-Nodes;
- the meta-information describing the IDN-Node itself and its properties

Fig 4.6 shows a representation of these three main components of the IDN-Node. The center of the node is taken by contents, as they represent the most valuable information and the reason why a particular node is put in place. On such contents are enabled some properties defined in the second ring, and finally, data are put in relation with other data through links represented in the outer ring.

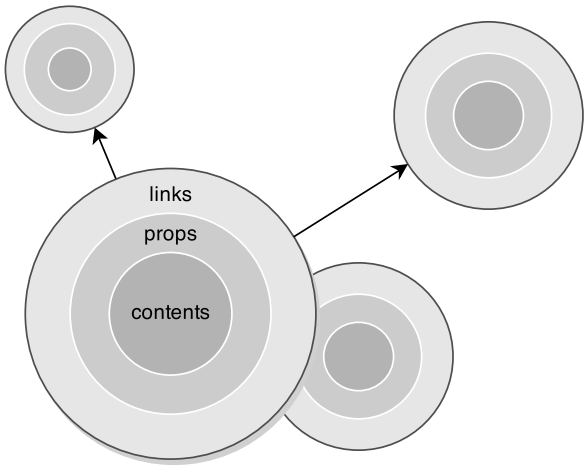


Figure 4.6. *The representation of the three major parts of an IDN-Node.*

4.1.2.1 The Content

The description carried out so far introduces some important questions: what are these data contained in the IDN-Node? Who put them there? Which operations are allowed with them? Let's start saying that data are entered in the node by the publisher, which can be either an agent creating data directly on IDN or a remote information source whose data have been processed to be represented as IDN-Documents. The publisher is likely to be the responsible for that information and is free to enter any kind of data, with the only limitation of encoding them in a string format (Base64 [Jos03] is recommended). However, this is an implementation constraint and doesn't affect the theoretical discussion.

I informally define the granularity level of an information piece as the number of directly addressable constituent information pieces, normalized by the total number of constituent information pieces.

For example, consider the problem of representing a list of employees salaries. It could be represented as a blob with a unique address, say a URL. In this case, to get the salary of Mr. Brown, we need to retrieve the blob dereferencing the URL, extract the content and perform a scan until the characters match the pattern related to that salary.

A different approach would be to give a URL to all employees and their salaries. A possible (RESTful) URL matching the salary of Mr. Brown could be the following:

```
http://www.example.org/employee/brown/salary
```

by dereferencing it, it will be possible to directly access the desired information. In the second example, the granularity level is obviously higher.

The node can host any kind of data: plain texts, images, spreadsheets, etc. The publisher decides the granularity level of data hosted in a node. This means that it won't be possible to get data with more fine grain than the one of the node (this consideration is generally true, although there are some exceptions that will be discussed later). I say this because each and every IDN-Node has a unique identifier. URIs (mostly URLs) are used for this purpose. Since the information held in a node is likely to be used (and re-used), is a good practice to adopt formats allowing information structuring such as XML and Json. Following this practice helps the framework to provide more support to information consumers.

In the following is given a sketch of the content section within the IDN-Node.

- Application data
  - Content
  - Content Encoding

- Content Type
- Content Schema

The main container is called `Application Data` because data put by the publisher are very often produced and exploited by applications. The section `Content` wraps data as they are, while the section `Content Encoding` and `Content Type` specify the algorithm used to encode data (for example Base64) and the adopted format. Allowed formats are the ones specified as MIME Types [FB96b, FB96c, Moo96, FK05a, FK05b, FB96a].

As previously introduced, InterDataNet does not put any constraint for the publisher on the allowed `Content Type`. This means that the `Application Data` entity can host every kind of data format. It goes without saying that structured format support a better exploitation of the information rather than unstructured ones. If the goal of an information provider is to deliver its data, will be a good strategy to adopt those formats which are easy to be parsed and processed such as XML or Json. In order to better support these formats, the section `Content Schema` is provided. As the name says, it contains a reference to (or embeds) the schema describing the information wrapped in the `Content` section. This allows advanced information management and even a grain refinement of the content information.

Since InterDataNet enables a read/write Web of Resources, a client has a significant freedom of action upon IDN-Nodes. It can create a new node, change the content, delete the content, modify each and every attribute, link other resources, delete an existing link, change the properties. In short, it has a complete control over them.

Please note, that at this level of abstraction security is not an issue. The discussion concerns the actions made available by the system regardless they will be actually accessible for every user.

4.1.2.2 Properties

In order to support collaboration around data, and let applications mutually reuse information, a set of properties is enabled for IDN-Nodes: licensing, privacy, provenance, versioning [Cio10] and security.

When the **licensing** property is enforced, an author can specify the conditions by which his/her resources can be reused. In InterDataNet, the licensing property is enforced for both IDN-Documents and IDN-Nodes. The reason for this is that IDN-Documents aggregate information, and the comprehensive view on a document is far more than the sum of partial views.

Since licensing concerns the regulation of creative works, it is important to define what types of data are contained in the IDN-Nodes. Some nodes might

contain factual data such as geographical coordinates or measures in general. Some might contain intellectual properties such as blog posts or music tracks. In addition, there could be nodes that store information to group other nodes in a creative, harmonic way, giving new meaning to data stored elsewhere. Such nodes should also be treated as intellectual products.

For this reason, licensing needs at least two properties:

– the data type, indicating whether data are factual or an intellectual property;

– the license in force, to specify which license applies, if any.

The **privacy** property is enforced when the owner is able to specify which information shouldn't be associated with his/her identity, and, of course, these prescriptions are fulfilled.

This property is critical because affects a core operation such as the linking of IDN-Nodes. To better understand why the privacy property is so highly related to IDN-Documents, let's examine Fig. 4.7. In the left side of the picture is presented a document forbidden for a certain client. It is made up of two different information pieces, A and B, the user is prohibited from accessing in the same context. To get more concrete, you can think of an health record containing personal data (A) and the result for a medical test, say HIV infection (B).

However, these two information pieces can be disclosed in different contexts, such as an employee register and an epidemiological survey. The fault arises when the client is able to recreate the forbidden document by assembling the two. Borrowing the terminology from the database field, this means that there is a *primary key* connecting the two sets buried within data.

This problem has been extensively discussed in [Ohm10], and is still an open challenge. However, some basic countermeasures could be put in place by allowing the publisher to flag identifying data. According to the privacy policy, these identifying data can or can not be assembled to avoid possible reconstructions.

This means that, if data in C were flagged to be identifying, none of the documents marked to be "allowed" in Fig. 4.7 would have been delivered to the client. Conversely, if data in C weren't flagged to be identifying, it is assumed that the forbidden document could not be recreated.

The procedure has two major limitations: it is very conservative since it significantly restricts the delivery of documents, and it relies on the publisher's ability to unerringly flag data.

Despite these limitations, it can provide a basic privacy level. More sophisticated strategies are currently subject of study.

The **provenance** property is enforced when the data consumer has access to information concerning the creation, generation and modification of these data. Since data combination is a key asset, provenance on single information grain is

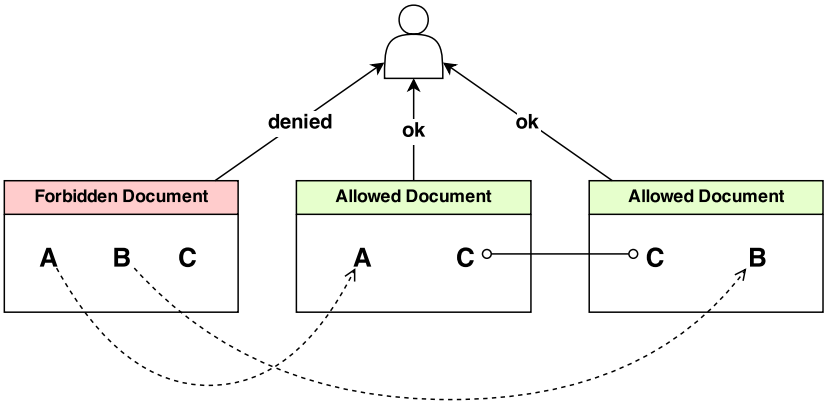


Figure 4.7. *A privacy critical scenario while aggregating data.*

enabled by specifying a set of attributes concerning the information source, the author, the responsible, the time of modification, generation or creation. In Web Science, provenance is still an open issue. Different provenance scenarios have been envisioned, each specifying peculiar requirements [LCFS11, TFKC11, ZSM+11]. This is the reason why a general approach has been chosen while designing the InterDataNet framework.

The **versioning** property concerns the collection of historical data from an information grain. Every time the information grain changes, leaves a state to reach a new one and the set of previous states is kept available. This allows to back track the history from its creation to the present time.

Versioning in InterDataNet is also conceived to support branching, similarly to modern versioning instruments such as Subversion [CSFP04], Git [Swi08], Mercurial [O'S09], etc. This feature supports the cooperation of different stakeholders around a resource because the reviews history is fully tracked. If an edit is not acceptable, is rejected by invalidating the present state and reverting the resource to a previous safe state. It is also possible to cooperate on separate branches and merge the resource in a unique master resource. The thorough discussion of these properties is out of the purposes of this thesis. More detailed insights can be found in [Cio10].

Security property is a complex one. It includes confidentiality, integrity, availability, supported by identification and authorization. **Confidentiality** is enforced when an unauthorized third party can't access a secured content. **Integrity** is enforced when is always possible to detect alterations occurred on data. **Availability** is enforced when data are assured to stay available under certain (system dependent) conditions. **Authentication** is the ability to verify the truth of a statement claimed by the user, while **Identification** is the ability

to associate a user to a precise identity. Often identification is enforced while authenticating a claim of identity.

In a distributed information scenario, where data from different providers are assembled to build new information, security is a challenging issue. A dedicated study, which can't be included in this thesis, has been carried out for this topic.

4.1.2.3 Links

As introduced in the previous sections, the InterDataNet Information Model supports two types of links: Aggregation Links specifying containment relations, Reference Links specifying reference relations. An IDN-Node can connect to one or more different IDN-Nodes through these links, with the meaning defined above. Given an IDN-Node `A`, it can connect to an IDN-Node `B` with an arbitrary link, provided the connection is not in place yet. More links of the same type (and directed in the same way) connecting `A` with `B` are, in fact, the same link, and it wouldn't make sense to duplicate a relation.

Similarly to many entities in InterDataNet, the conceptual displacement of links is borderline. They connect different nodes establishing relations between them, so they should be discussed from the node's perspective. On the other hand, for the same reason, they are the foundation of the document concept, so they should be discussed form an holistic perspective.

Probably, the most effective way to address the problem, is to explain different points of view depending on whether we want to give a general or particular connotation. This is the reason why many concepts are tackled in different parts of this doctoral dissertation.

According to [PS03], the main purpose of an Information Model is to model managed objects at a conceptual level (defining relationships between them), independent of any specific implementations or protocols used to transport data. In this case, the term "information" is very appropriate because the system literally elects the information as its main subject. In compliance with this definition, the Information Model defines relations between data as they are, ready for the delivery and not requiring additional processing by the middleware.

Nevertheless, InterDataNet provides a more advanced link, by introducing the computation concept from behind the scenes. This link is called Active Link, and defines a computation dependency between two IDN-Nodes. This means that the information in a node can be dynamic and requires an algorithm and some inputs to be evaluated. Thus, the node an Active Link departs from is a *scriptable node*.

In the following the InterDataNet support for scripting is discussed more deeply.

4.1.3 The Activity Node

In this section the scripting concept is introduced in InterDataNet by defining Activity Nodes and Active Links. This may perplex because the middleware is claimed to have a strong resource-oriented connotation. From this point of view, the coexistence of the computation with data is delicate and must be precisely explained.

Let's start by saying that InterDataNet *do* have a strong resource-oriented connotation. Even more, InterDataNet *is* definitely a resource-oriented framework. InterDataNet enables a read/write Web of Resources, exposing the document resource leveraging the REST architectural style. Therefore, documents are presented to clients as they are, ready to be consumed or modified. From the clients' point of view, InterDataNet is a black box that presents data on command and graphs of data are the only resource exchanged between the system and clients.

The paradigm holds if the processing features are masked to the client. Thus, the client can request a resource never knowing its state depends on the execution of a script. Doing a little abstraction, is basically what happens in the Web applications' business logic: clients request resources to the server, as if they have always been there. It really doesn't matter how the server is able to provide them, which computations are required, what patterns it implements and if it has to retrieve data elsewhere, as long as these details do not leak out to the clients.

From this point of view, defining a scriptable node is just about taking a little processing out of the server and distribute it on a graph. Of course, the aim of the code contained in the script is to produce information strictly inherent to the node, more precisely the one that will be integrated in the `Content` section of the `Application Data`.

To clear the things up, a typical application of the Activity Node is presented. A major has invested in smart technology for the city, and installed many different sensors in the urban territory. Let's assume that data from these sensors are exposed as IDN-Documents. In order to monitor the citizens' quality of life (QoL), the major hires a team of researchers who find a strong correlation of the wellness with pollution, noise and traffic.

To define the intervention strategy, the administration needs to monitor the level of the QoL in different areas of the city. By leveraging InterDataNet, they decide to create a new QoL location-dependent sensor document which aggregates documents representing the near pollution, noise, and traffic sensors. Such document is likely to be implemented via an Activity Node hosting the script for computing the QoL from the derived measures. Getting even more concrete, the script that implements the QoL could be some kind of function, say a weighted mean $w(p, n, t, d)$, of pollution (p), noise (n) and traffic (t) measurements as well as the distance of the QoL sensor from them (d). In order to provide inputs for

the computation, the Activity Node is connected with the noise, traffic and pollution sensor documents (having data including the corresponding measurements and locations) through Active Links. In this way, when the QoL document is requested, the corresponding Activity Node is inspected, pollution, noise and traffic IDN-Node dependencies are found and resolved, measurement data are extracted and entered as inputs of the script. The computation executes and the output $o = w(p, n, t, d)$ is used to fill the `Application Data` section of the same node. At this point, the QoL node is returned with the QoL measurement in place. Fig. 4.8 shows the QoL example.

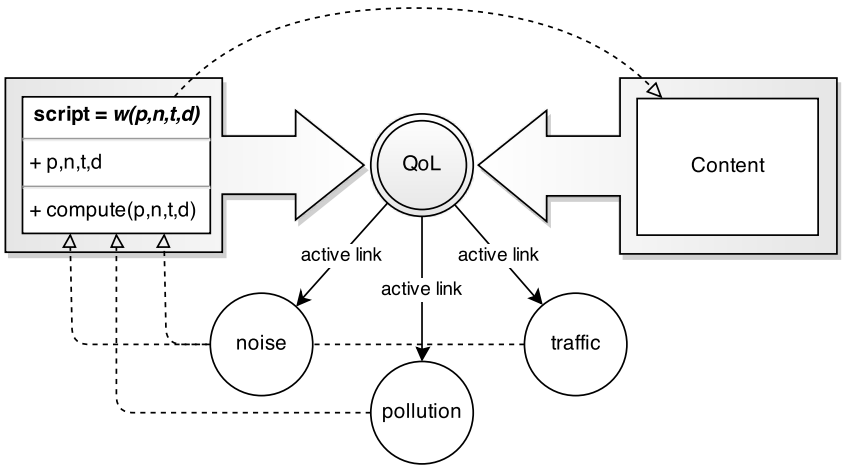


Figure 4.8. *The Quality of Life sensor implemented as an Activity Node.*

Since the middleware is general purpose, it would never have implemented a functionality which is vertical to a domain, in this case the processing of the QoL document. Such processing should have been put on a different application, significantly complicating the scenario (this would have happened for every use case requiring dynamic information). The Activity Node solves the problem elegantly, because is a strategy to let the middleware run *ad hoc* code, still preserving generality.

After having clarified the perspective in which the Activity Node is put and the vision that motivated its conception, more rigorous definitions are given.

Activity Node The Activity Node is an IDN-Node embedded with a script whose intent is making the IDN-Node's `Application Data` (fully or partially) automatically generated.

More Activity Nodes can be chained to get complex processing flows by using Active Links. From the definitions of the Information Model and Activity Node, the processing of the scripts are completely isolated and independent.

Active Link The Active Link defines a direction in the flow of a processing. An Active Link always starts from an Activity Node and points to an IDN-Node (whether it is an Activity Node or not), defining the dependencies to fulfill in order to execute the script.

The processing chain is not visible to Activity Nodes, whose domain is restricted to themselves. This characteristic is strengthened by the fact that when an Activity Node derefrences an Active Link, it doesn't even know whether the architecture is processing an Activity Node or a common IDN-Node. Fig. 4.9 shows the basics and the rationale of the processing actions performed by the Activity Node.

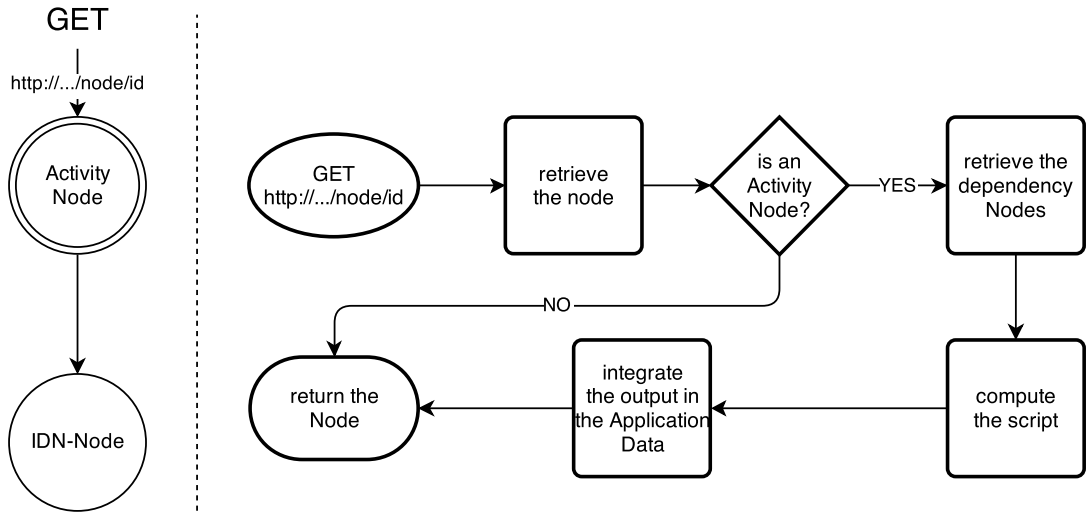


Figure 4.9. *A flowchart exemplifying the basics of the processing actions performed by a Activity Node*

In the left side of the figure is depicted the document involved in the scenario while in the right side takes place the flowchart representing the process triggered by an Activity Node resolution. The example scenario shows an Activity Node connected to an IDN-Node via an Active Link. This configuration means that the IDN-Node is a dependency for the execution of the script, since it contains the required input parameters.

Once a read operation is issued for the Activity Node (in the example, an HTTP GET), the system retrieves it as an ordinary IDN-Node. This means that no special identifiers nor special operations are enabled for Activity Nodes. The interface is exactly the same for all IDN-Nodes, regardless of their nature. If the current IDN-Node is an Activity Node, the system internally recognizes its nature and checks its dependencies (the Active Links). In order to execute the script, all inputs have to be gathered first, and subsequent retrievals are performed by issuing other read operations. When inputs are available, the execution of the

script is attempted and, if it successes, the `Application Data` section of the Activity Node is filled with the script output as well as the the content type and encoding.

The implementation of a consistent Activity Nodes network presents some critical issues and requirements that will be discussed in the following section.

4.1.3.1 Activity Node Requirements and Constraints

The first problem to face while dealing with a processing entity is the definition of inputs and outputs. If they are not clearly defined, will be hard to provide the processing entity with compatible data and to properly interpret its outcome.

This is the reason why an Activity Node supports a strong typed input and output definition. By clearly addressing this issue the externalization of the script can be envisioned to put it in a global accessible space (as currently happens for XML Schemas), fostering information reuse. Just think of a weighted mean algorithm. It is a very general purpose algorithm which is found in many libraries for different programming languages. The externalization is attractive since many different Activity Nodes can import the same algorithm and apply their inputs. This concept allows to create a trusted algorithm source providing stable and optimized code, as currently happens with popular frameworks and libraries. However this is not the case of the current model, even if future works in this direction are planned.

Data types for defining inputs and outputs are borrowed from the XML Schema's built-in data types [BM04], because is a well-accepted and mature standard. In the following, data types chosen for the InterDataNet Information Model are listed.

- dateTime
- date
- boolean
- string
- base64Binary
- hexBinary
- float
- double
- int
- long
- byte
- anyURI

As previously stated, the content of the `Application Data` can be any kind of information object with any granularity level, because the publisher is the one

entitled to decide it. When the IDN-Node referred by an Active Link has some unstructured data, the script is forced to process them as a whole. Conversely, when these data are structured, the script should extract the inputs by leveraging this underlying organization. Therefore, is crucial to have a system for addressing the information grains contained in the data structure. In the Listing 4.1 a sensor is represented using XML. It is very likely that an Activity Node pointing to the sensor IDN-Node would extract the temperature as an input for the computation of the script.

As a consequence, a double mechanism for targeting objects is required. First, is necessary to locate the IDN-Node, then is necessary to locate the information grain(s) contained in the IDN-Node `Application Data` section. Technologies adopted for this purpose are URIs and XPath [Urp08] for XML and JsonPath [Goe07] for Json representations. Currently, no other structured formats are supported.

It is worth to point out that these technologies are used for the internal processing of contents. Later, in section 4.2.2, will be detailed a way for targeting contents by specifying a special URI (IDN-URI). Such URI leverages a XPath syntax (precisely, a sub set of available operations for XPath) for targeting contents. This shouldn't be misunderstood with these content processing issues. It is an higher level syntax to address elements belonging to general structured resources. It has nothing to share with specific representations. Indeed, at time of processing, this syntax will be translated in JsonPath if the content presents a Json format.

If the data schema is unknown, it won't be possible to define the XPath or JsonPath expression to target the information grain(s). This consideration demands for the definition of the data schema for the IDN-Node. It is not mandatory for the publisher to declare the data schema associated with the published data, but it is strongly advised. In fact, making the data schema publicly available is an enabling factor for the cooperation and for the information reuse promotion. Moreover, a well defined content is compliant with the Activity Nodes' targeting requirements.

```
<sensor>
  <location>
    <lat>44.08758502824518</lat>
    <long>11.0302734375</long>
  </location>
  <producer>ACME</producer>
  <physicalQuantity>temperature</physicalQuantity>
  <unit>celsius</unit>
  <value>17.9</value>
</sensor>
```

Listing 4.1. A structured XML document representing a sensor.

The possibility of chaining several Activity Nodes introduces the problem of the infinite resolution. In other words, if there is a cycle in the dependency relations, the retrieval procedure will never end.

Fig. 4.10 shows a cycle in the dependency relations between the `A`, `B`, `C`, `D` Activity Nodes. In such scenario when a retrieval operation is issued to a node, say `A`, the resolution won't end. Indeed, `A` is requested, but since it needs `B` to compute the script, its `Content` won't be ready until `B` is available as an input for the script. Therefore, a resolution of `B` is issued. In its turn, `B` needs `C` to compute its script, behaving analogously. The problem arises when the requests succession involves `A` for a second time: the request from `A` is still pending and will never be fulfilled, so `A` can't be returned in any case and the system fails. Even worse, as a consequence of the very first retrieval operation on `A`, a new resolution sub-cycle is triggered (`A`, still waiting for `B`, issues a second retrieval of `B`), resulting in a senseless resource-draining operation.

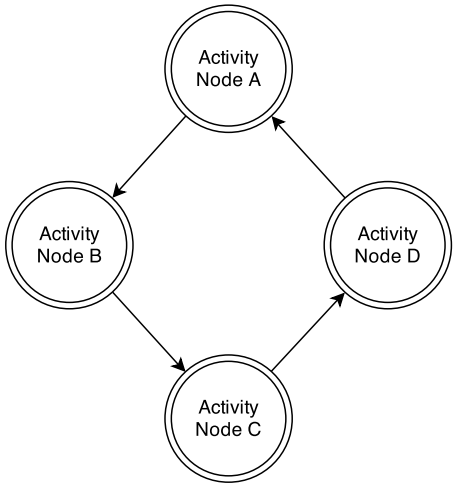


Figure 4.10. *A cycle in the dependency relations between Activity Nodes*

This problem demands for a cycle detection algorithm whose requirements are the following:

- if a dependency cycle exists, it will be detected.
- if a dependency cycle is detected, the system is put in a safe state.
- if more dependency cycles exist, the detection of the first cycle should put the system in a safe state (optimization requirement).

Basically, the rationale of the cycle detection algorithm consists in marking retrieval requests with node specific tokens. A request can be marked several

times with different tokens. When a node receives a marked request, it scans the marks list looking for its token. If the lookup is successful, a cycle is detected.

Listing 4.2 shows the pseudo-code in Java style of the algorithm. In the pseudo-code some resolution details are omitted, since they are not the focus here. For example, line 14 reports an invocation for the resolution of a new dependency, without considering that the call should be performed for every unresolved Active Link: not to complicate the code with inessential details, the `for` loop is omitted.

The guard at line 2, says that the algorithm runs only if the current node is an Activity Node. It makes sense since it is impossible that an ordinary IDN-Node without any processing capability triggers a dependency cycle. Dependencies propagate through Active Links so the guard at line 4 stops the detection if no Active Link is found. At this point, the token for that request is deterministically generated(the advised approach is an hash function, say MD5, of the canonical LRI). When a request is marked with more tokens (i.e. the token list is not empty), a check of the token is performed against the token list (line 6). If the token is found, the cycle is detected and the algorithm returns a positive output. Otherwise, the token is added to the list before issuing a new request (lines 10 and 11).

```
1  cycleDetected = false;
2  if(node is Activity Node){
3          if(node has Active Links){
4                  token = generateToken(node);
5                  if(request has other tokens){
6                          cycleDetected = checkList(tokens, token);
7                          if(cycleDetected)
8                                  return cycleDetected;
9                  }
10                 tokens.add(token);
11                 issueNewRequest(next Active Link, tokens)
12         }
13         return cycleDetected;
14 }
15 end
```

Listing 4.2. The Java pseudo-code of the cycle detection algorithm.

The algorithm is also represented in the flowchart depicted in Fig. 4.11.

For demonstration purposes, the algorithm is tested on a more complex graph having three dependency cycles, as the one depicted in Fig. 4.12. The first retrieval request is issued to the `A` Activity Node. Since the token list is empty, `A` will mark the request with its token and issue the retrieval of `B`. When `B` receives the request, it checks its token against the token list with negative results. Therefore, it adds its token to the token list and issues the request to `C`. When `D` will issue

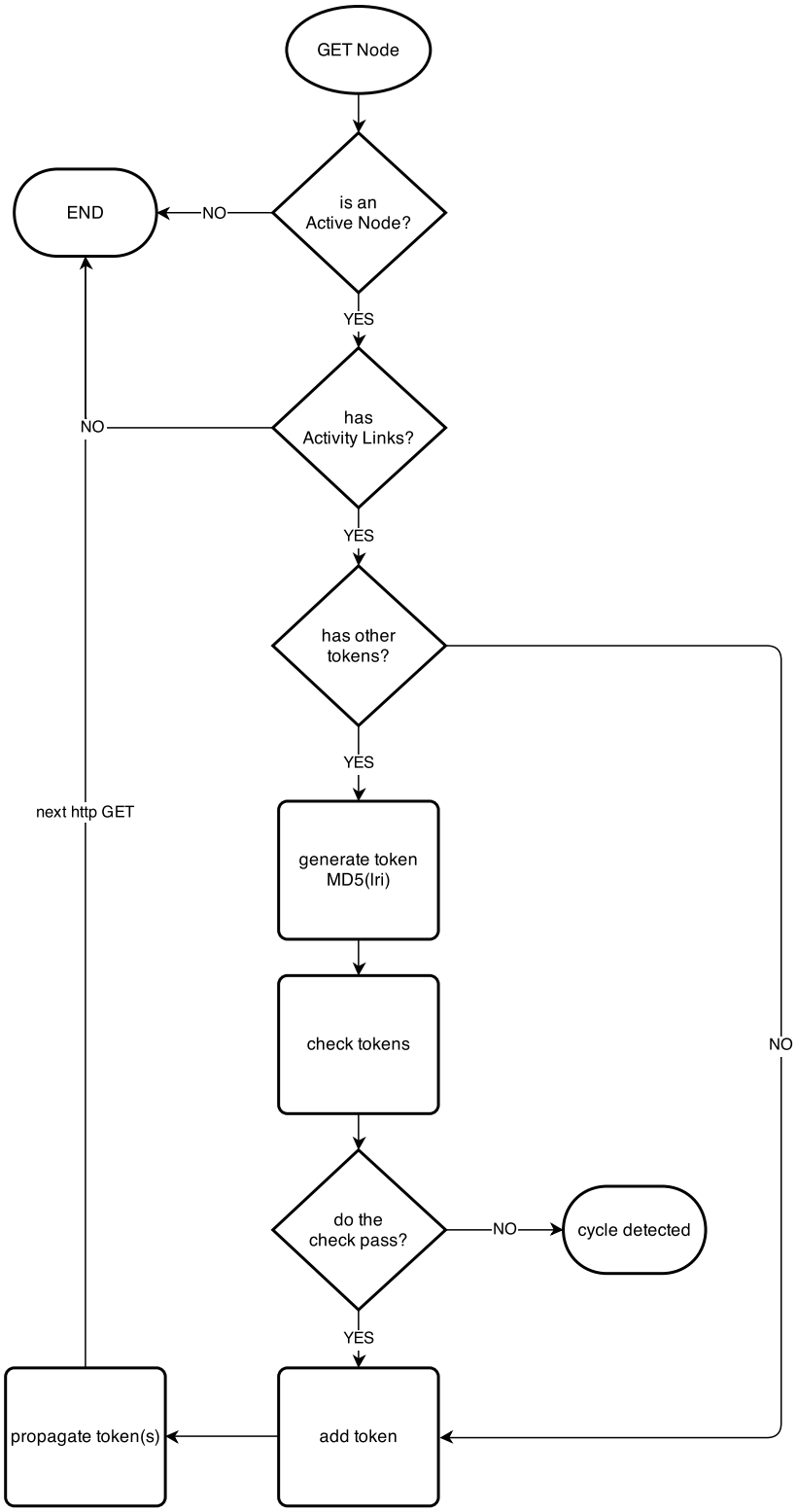


Figure 4.11. *A flowchart representing the cycle detection algorithm.*

the request to `A`, `A` will detect the cycle and will return an error message to `D`, which will be forwarded node by node to `A`.

It is easy to see that the algorithm fulfills all above mentioned requirements. In a modern implementation, it is likely that `B` issues the request to `C` and `F` in parallel. This doesn't affect the fulfillment of requirements since the first cycle to be detected puts the system in the safe state by notifying the error.

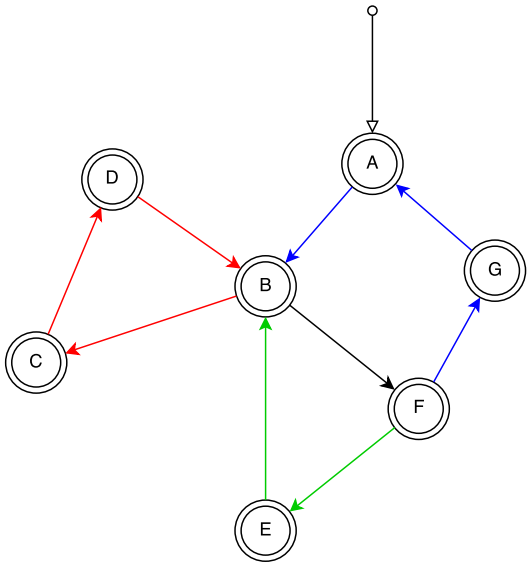


Figure 4.12. *A testing use case for the cycle detection algorithm.*

A constraint is put also on the implementation:

– the implementation of the algorithm should comply with the statelessness REST principle (scalability requirement).

The statelessness requirement says that when a client interact with a server, the servers shouldn't keep any state related to the communication. In other words, the state should be kept within the request. This principle is critical to make the algorithm lightweight and resource-saving. Indeed, if the server would store the state of every request, a significant number of them would result in a resources draining and, ultimately, in a crash.

The transfer of the token list is implemented via a dedicated HTTP header, in compliance with the HTTP standards. In this way, a client (that here is a Web application) can compute the token retrieving the Activity Node from the architecture and running a deterministic, node-dependent function. On the other hand, it can retrieve the token list directly form the request, without needing to store anything.

4.2 The InterDataNet Data Model

To introduce the InterDataNet Data Model, I refer to definitions from [PS03], for a second time: Compared to Information Models, Data Models define managed objects at a lower level of abstraction. They include implementation- and protocol-specific details, e.g. rules that explain how to map managed objects onto lower-level protocol constructs. Therefore, the choice of REpresentational State Transfer (REST) as architectural style will be motivated, and major implementation details such as the protocol and the resources' definition will be discussed.

Since Roy Fielding presented his dissertation *Architectural Styles and the Design of Network-based Software Architectures* [Fie00a] in 2000, a big debate about the best style for designing network architectures has risen. The contenders are the WS-* supporters against the REST supporters. Many analysis have been carried out highlighting the strengths and weaknesses of the two approaches, with the result that what really makes the difference is the purpose of the architecture. This means that there is no universal best choice, it depends on the context and the objectives of the architecture. Many argue that for large enterprise applications, WS-* is more suitable, while for Web oriented applications REST is the best choice.

In one of the most comprehensive comparison [ASJH11], Pautasso *et al.* [PZL08] compare RESTful and WS-* services on 3 levels: 1) architectural principles, 2) conceptual decisions, and 3) technology decisions. Pautasso *et al.* argue that "the main conclusion from this comparison is to use RESTful services for tactical, *ad hoc* integration over the Web (***à la Mashup***) and to prefer WS-* Web services in professional enterprise application integration scenarios with a longer lifespan and advanced quality of service requirements".

Thus, for data aggregation in the Web environment (which is clearly the case of InterDataNet), REST architectural style is strongly advised.

A study led in 2012 [GIM12] involved 69 computer novice developers (science students) in implementing HTTP based client applications exploiting the two approaches (of course, according to the styles, HTTP is used as application and transport protocol, respectively). At the end of the experiment, an interview is submitted to the volunteers to catch their feelings and opinions about REST and WS-*. Results reported that REST is statistically significantly easier and faster to learn than WS-*. Fig. 4.13 is extracted form the work [GIM12] and shows the perceived learning accessibility of REST vs. WS-*.

Moreover, a significant number of volunteers agreed that REST was more adapted for Web applications requiring to integrate Web content: "*[for] Web Mashups, REST services compose easily*".

Since InterDataNet aims to realize a Web of Resources, the core technologies

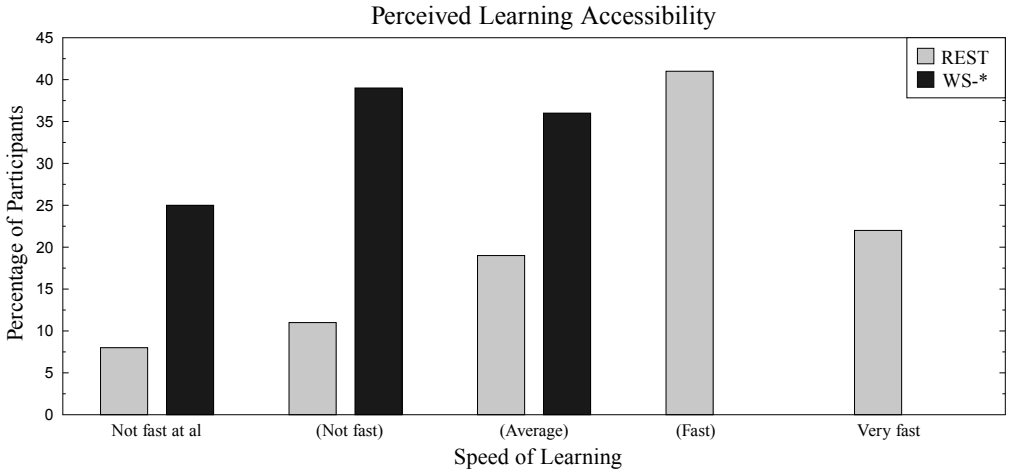


Figure 4.13. *The bar chart from [GIM12] showing that the majority of participants reported that REST as fast or very fast to learn and WS-* as not fast or average.*

for implementing the REST paradigm are *the* Web protocol, i.e. HTTP, and URIs. From now on, REST will be discussed assuming HTTP as implementation.

In the following, REST principles are reminded (a complete discussion is tackled in chapter 3) and is discussed how InterDataNet complies with them. Many works summarize the content of Roy Fielding dissertation to extrapolate and present the REST principles. Among them, [Til07] has been chosen for its completeness, clarity and simplicity of presentation.

REST defines five prinicples:

– Representation Oriented

– Addressability

– Uniform Interface

– Stateless Communications

– Hypermedia as the Engine of the Application State (HATEOAS)

The interaction of clients and servers is **Representation Oriented** since they exchange information through resources representations. This means that information entities are abstract and they are supposed to exist regardless of their representation, that may change with respect of the server capabilities.

Addressability is the idea that every resource in a system is reachable through a unique identifier. In the REST world, addressability is managed through the

use of URIs. It is strongly advised to adopt well structured meaningful URI to name resource.

Every resource should support a **Uniform Interface**, i.e the same set of methods (or operations). HTTP calls these *verbs*. The set of standard methods includes GET, POST, PUT, DELETE, HEAD and OPTIONS. The meaning of these methods is defined in the HTTP specification, along with some guarantees about their behavior.

It is important to stress that although REST includes the idea of statelessness, this does not mean that an application cannot have state, in fact this would make the whole approach pretty useless in most scenarios. The **Stateless Communication** principle mandates that state be either turned into resource state, or kept on the client. In other words, a server should not have to retain some sort of communication state for any of the clients it communicates with, beyond a single request. The most obvious reason for this is scalability: the number of clients interacting would seriously impact the servers footprint if it had to keep client state. (Note that this usually requires some re-design, you cant simply stick a URI to some session state and call it RESTful.).

Hypermedia as the Engine of the Application State principle states that it is important to include hypermedia in a representation, so as to drive the user agent towards the next state of the application.

4.2.1 The Representation Oriented Principle in InterDataNet

InterDataNet is a layered RESTful architecture, therefore resources are defined for every interface. However, the discussion carried out in this section concerns the point of view of the middleware's client and there is no need to deepen technical details pertaining to the inner architecture that will be tackled in chapter 5. This premise is made to clarify that only the interface exposed to the outside is tackled here.

In order to keep the interface as simple as possible, a single resource is defined. InterDataNet exposes the IDN-Document resource only, and by a rigorous definition of the the enabled operations, it suffices for the easy management of the underlying graph of information grains. This statement must not be misunderstood: to elect the IDN-Document as the only resource defined by the interface, it doesn't mean that an IDN-Node couldn't be exchanged between clients and the architecture. Indeed, it is worth noting that the InterDataNet Information Model defines the IDN-Document to be made up of an *arbitrary* number of IDN-Nodes. Thus, the possibility that the document is composed of a single node (or even no nodes at all, in the degenerate case) is absolutely acceptable and compliant with the definition.

The exchange of IDN-Nodes occurs in the form of IDN-Documents containing

one IDN-Node (i.e., graphs with a single vertex).

Fig. 4.14 shows a comprehensive view of the IDN-Node object. The dashed lines represent a list of elements (e.g., aggregation, reference, backlink and active links). The colors highlight the node's main sections with their descendants elements, namely `Application Data`, `Application Meta`, `IDN Meta`, `Management Meta` and `Activity`. All these elements will be detailed in the following.

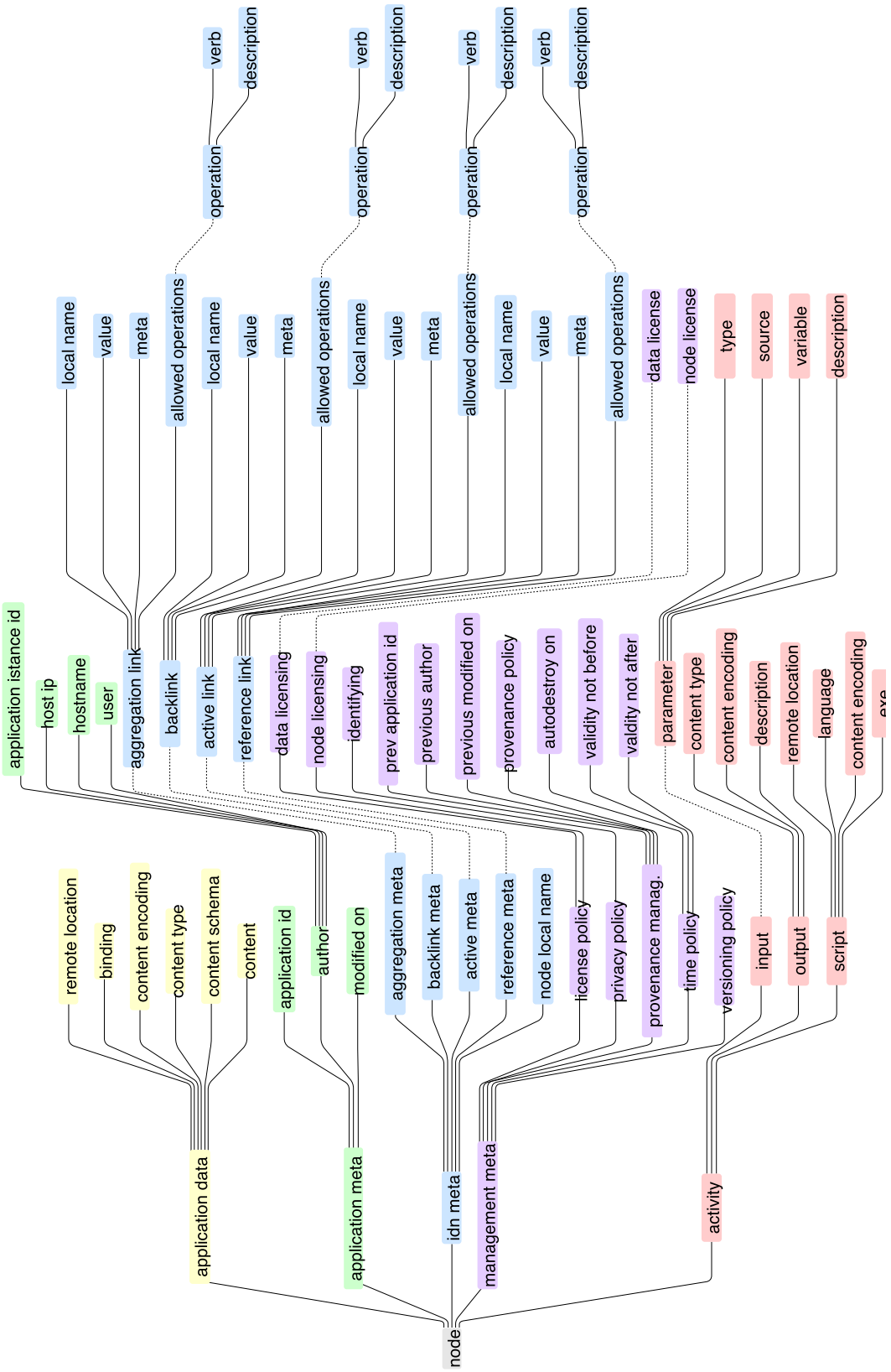


Figure 4.14. *The full node resource.*

In the course of the chapter, I will refer to an XML representation of the document resource, for the only reason XML is more friendly for the human readability and comprehension. This must not be interpreted as a technical preference or constraint on the representation format.

Indeed, in compliance with REST "representation oriented" principle discussed in section 3.1.0.2, InterDataNet fully supports the content negotiation mechanism by leveraging the `Accept` header of the HTTP request. The current implementation supports XML, Json and HTML representation of the document resource, even though additional formats can be included in future releases.

The InterDataNet Data Model requires that an IDN-Document is represented via the `VRDoc` resource. The name stands for "Document handled by the Virtual Resource (VR) layer" (see section 5.2) which is the layer entitled to implement the IDN-Document abstraction.

The `VRDoc` element contains a `VRDocInfo` element and a number of `VRNodeEnvelope`s that map 1:1 with the nodes contained in the document. The `VRDocInfo` is used to vehicle meta information about the document it belongs to. basically, its main purpose is to highlight the presence of problems during the commit of the requested operation. This mechanism is used in combination with the native HTTP Status errors notification, when the latter lacks specificity. Indeed, The `VRDocInfo` contains the `Warnings` and `Errors` sections. Both `Warnings` and `Errors` share a common template: they present an internal InterDataNet-specific `Code` associated to the problem, a human-oriented (plain text) `Message` describing the problem in natural language and a list of `Targeted LRIs` specifying the URIs of nodes affected by the problem. This additional information is very important to understand the cause of peculiar problems. When a problem occurs, the checking flow should proceed in this way: first the HTTP response status message is inspected, then the returned IDN-Document is examined, by evaluating the `VRDocInfo` section. If the root of the problem lies in one or more nodes, it is possible to target them by inspecting the `Targeted LRIs` list element. The LRI acts as a pointer to the nodes whose `VRNodeInfo` (see below) section can be opened for more detailed information. Fig. 4.15 shows the correct error/warning checking flow.

The `Warnings` section is used to notify minor issues, while the `Errors` section is used to notify major problems, probably preventing the successful completion of the current operation. Tables 4.1 and 4.2 report the available warnings and errors defined in the InterDataNet Data Model.

A `VRNodeEnvelope` contains two elements: `VRNodeInfo` and `VRNode`. The former contains the meta information related to the IDN-Node it is paired with. `VRNodeInfo` specifies five attributes: `IsRoot`, `HttpStatus`, `Etag`, `Keywords` and `Location`.

The `IsRoot` attribute identifies the entry node of the graph (informally named

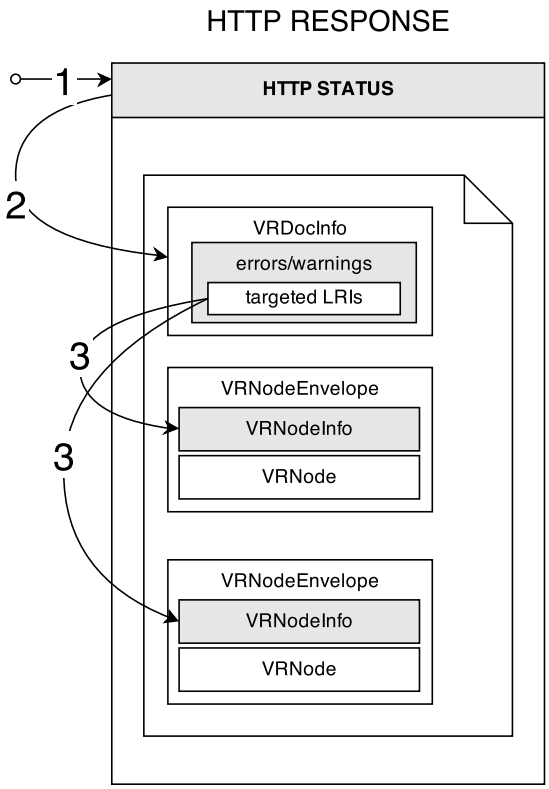


Figure 4.15. *The error/warning checking flow, with the precedence to fulfill in order to obtain a more precise description of the problem.*

Code	Message
Cycle Detected	A cycle has been detected in the document.
Incomplete Write	The WRITE operation has been unsuccessful for one or more nodes. Check the VrNodeInfo section for details.
Missing Etag	One or more required ETAG values are missing.
Malformed Request IDN URI	The URI used for the Request is not a valid IDN URI.
Incomplete Inner Resource Resolution	One or more nodes in the document don't match the request parameters.

Table 4.1. The Warnings defined for the InterDataNet Data model

Code	Message
Incomplete Resolution	One or more errors encountered during the document resolution led a partial resolved document.
Root Uniqueness Error	Two or more nodes in the document are declared to be the root.
Root Not Found	It is not possible to find a node declared as root. Use the IsRoot attribute.
LRI Syntax Error	One or more Lri(s) are not valid.
Aggregation LRI Syntax Error	One or more Lri(s) declared as aggregation link(s) are not valid.
VR-Id Uniqueness Error	The vr-ids must be unique within a list scope.
Bad PUT Document name	The document name (URI) must not contain parameters while performing a WRITE operation
Dependency Cycle Detected	A dependency cycle has been detected preventing the resolution of the document.

Table 4.2. The Errors defined for the InterDataNet Data model

"root"), i.e. the node starting from which the *current document* is defined (for a quick reminder of the concept, refer to Fig. 4.4). Within a document, only one node can be declared as root. If more "roots" are found in a document, the "Root Uniqueness Error" of Table 4.2 is thrown.

The `HttpStatus` and `Etag` elements require a separate explanation. It can

be odd to find inside the resource attributes defined in the HTTP specification, whose handling normally pertain to the protocol. However, it is worth noting that when these attributes are enforced at HTTP response level, they affect the exchanged resource *as a whole*. There is no way to specify that a part of the object (a sub-resource) is affected.

Nevertheless, InterDataNet defines a unique document resource which is an aggregation of more fine grained elements (the nodes). To keep the interface usable, it should be possible to manage sets of related nodes in a sleek way. Truth to be told, defining RESTful interactions on the document resource serves well the purpose, but it is critical to have a more fine grained meta information about such document elements. For example, performing the creation of a document, implies the creation of every single node constituting that document (in this example, every node must be considered completely new to the system). Since the operating environment is the Web, it is well accepted that unavailability problems may occur. In this case, a best-effort strategy is chosen, and the operation is performed to the best, i.e. only a part of the original nodes results to be created.

It is clear that the requesting client must be made aware of such circumstance. To specify which operation went wrong, the `HttpStatus` is used to describe the operation outcome for single nodes. Similarly, the `Etag` attribute is used to represent the state of a specific IDN-Node. The Etag function will be discussed when tackling the Stateless Communication principle.

The `Keywords` and `Location` are dedicated to the indexing system of the InterDataNet Search Engine. It is not the case to introduce the search engine here, briefly, the `Keywords` attribute contains a list of comma-separated keywords to be associated with the node, while `Location` supports geographical indexing of location-related resources. This enables searches requiring spatial constraints, such as "the nearest gas station", or "the church not far more than 3 Km from this point".

Listing 4.3 shows the XML Schema representation of the `VRDocInfo` and `VRNodeEnvelope` elements.

```
1 <xs:schema version="1.0">
2   <xs:element name="VRDoc" type="idnVrDocument"/>
3   <xs:complexType name="idnVrDocument">
4     <xs:sequence>
5       <xs:element name="VRDocInfo" type="VrDocInfoType"
        minOccurs="0" maxOccurs="1"/>
6       <xs:element name="VRNodeEnvelope" type="idnVrNodeEnvelope"
        minOccurs="0" maxOccurs="unbounded"/>
7     </xs:sequence>
8   </xs:complexType>
9   <xs:complexType name="VrDocInfoType">
```

```
10      <xs:all>
11        <xs:element name="Errors" type="ErrorsType" minOccurs="0"
          maxOccurs="1"/>
12        <xs:element name="Warnings" type="WarningsType"
          minOccurs="0" maxOccurs="1"/>
13      </xs:all>
14    </xs:complexType>
15    <xs:complexType name="ErrorsType">
16      <xs:sequence>
17        <xs:element name="Error" type="IdnExceptionType"
          minOccurs="0" maxOccurs="unbounded"/>
18      </xs:sequence>
19    </xs:complexType>
20    <xs:complexType name="WarningsType">
21      <xs:sequence>
22        <xs:element name="Warning" type="IdnExceptionType"
          minOccurs="0" maxOccurs="unbounded"/>
23      </xs:sequence>
24    </xs:complexType>
25    <xs:complexType name="IdnExceptionType">
26      <xs:all>
27        <xs:element name="Code" type="xs:string" minOccurs="1"
          maxOccurs="1"/>
28        <xs:element name="Message" type="xs:string" minOccurs="1"
          maxOccurs="1"/>
29        <xs:element name="TargetedLris" type="lriList" minOccurs="0"
          maxOccurs="1"/>
30      </xs:all>
31    </xs:complexType>
32    <xs:complexType name="lriList">
33      <xs:sequence>
34        <xs:element name="Lri" type="xs:string" minOccurs="1"
          maxOccurs="unbounded"/>
35      </xs:sequence>
36    </xs:complexType>
37    <xs:complexType name="idnVrNodeEnvelope">
38      <xs:all>
39        <xs:element name="VRNode" type="idnVrNode" minOccurs="1"
          maxOccurs="1"/>
40        <xs:element name="VRNodeInfo" type="idnVrNodeInfo"
          minOccurs="1" maxOccurs="1"/>
41      </xs:all>
42    </xs:complexType>
43    <xs:complexType name="idnVrNodeInfo">
44      <xs:all>
```

```
45        <xs:element name="IsRoot" type="xs:boolean" minOccurs="1"
          maxOccurs="1"/>
46        <xs:element name="Etag" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
47        <xs:element name="HttpStatus" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
48        <xs:element name="Keywords" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
49        <xs:element name="Location" type="locationType"
          minOccurs="0" maxOccurs="1"/>
50      </xs:all>
51    </xs:complexType>
52    <xs:complexType name="locationType">
53      <xs:all>
54        <xs:element name="Longitude" type="xs:double" minOccurs="1"
          maxOccurs="1"/>
55        <xs:element name="Latitude" type="xs:double" minOccurs="1"
          maxOccurs="1"/>
56      </xs:all>
57    </xs:complexType>
58  </xs:schema>
```

Listing 4.3. The part of the XML schema representing the IDN-Document `VRDocInfo` and `VRNodeEnvelope` elements.

A very important aspect to point out is that all "`VR*Info`" attributes are *transient*. This means that they don't describe the resource for what it is, but the resource declined in the context of the current communication. An IDN-Node could be marked to be the "root" in a document, while the same IDN-Node could be a simple aggregated node in a different document. Again, the same IDN-Node could be served with a 412 `HttpStatus` attribute one time and a 200 `HttpStatus` another. Moreover, The attributes for indexing, will never be returned as a consequence of a GET operation on the resource.

The real state of the resource, independent form the communication details, is represented by the `VRNode` section (see a XML Schema representation in Listing 4.4). It is marked with the canonical LRI value of the current node (the global identifier) and contains five important elements: `VRApplicationData`, `VRApplicationMeta`, `VRIDNMeta`, `ManagementMeta` and `Activity`.

```
1  <xs:schema version="1.0">
2    <xs:complexType name="idnVrNode">
3      <xs:all>
4        <xs:element name="VRApplicationData"
```

```
      type="vrApplicationData" minOccurs="0" maxOccurs="1"/>
5     <xs:element name="VRApplicationMeta"
      type="vrApplicationMeta" minOccurs="0" maxOccurs="1"/>
6     <xs:element name="VRIDNMeta" type="vrIdnMeta" minOccurs="0"
      maxOccurs="1"/>
7     <xs:element name="ManagementMeta" type="managementMeta"
      minOccurs="0" maxOccurs="1"/>
8     <xs:element name="Activity" type="activity" minOccurs="0"
      maxOccurs="1" />
9    </xs:all>
10   <xs:attribute name="lri" type="xs:anyURI" use="required"/>
11   <xs:attribute name="query" type="xs:string" use="optional"/>
12  </xs:complexType>
13 </xs:schema>
14 </xs:schema>
```

Listing 4.4. The part of the XML schema representing the `VRNode` element.

`VRApplicationData` is the section entitled to hold contents, i.e. the information to be consumed by the applications (refer to section 4.1.2.1). This section contains `Content`, `ContentSchema`, `ContentEncoding`, and `ContentType` subsections which wrap the content itself, the location hosting the schema for this content, the encoding used for the content (for example, base64), and the content type accepting MIME type for content notation (for example, `application/xml`, `application/json`, `text/plain`, `image/jpeg` and so on), respectively. For the sake of completeness, there are two additional sections used when binding a remote resource to the IDN-Document representation: `RemoteLocation` and `Binding`. These are used when data exposed by outer information providers are managed by the InterDataNet middleware to be re-exposed as IDN-Documents. This topic will be discussed in detail when the Adaptation facility will be introduced. Listing 4.5 shows the `VRApplicationData` XML Schema representation.

```
1  <xs:schema version="1.0">
2   <xs:complexType name="vrApplicationData">
3    <xs:all>
4     <xs:element name="RemoteLocation" type="xs:anyURI"
      minOccurs="0" maxOccurs="1"/>
5     <xs:element name="Binding" type="xs:boolean" minOccurs="0"
      maxOccurs="1"/>
6     <xs:element name="ContentEncoding" type="xs:string"
      minOccurs="0" maxOccurs="1"/>
7     <xs:element name="ContentType" type="xs:string"
      minOccurs="0" maxOccurs="1"/>
8     <xs:element name="ContentSchema" type="xs:anyURI"
```

```
        minOccurs="0" maxOccurs="1"/>
9       <xs:element name="Content" type="xs:base64Binary"
        minOccurs="0" maxOccurs="1"/>
10    </xs:all>
11  </xs:complexType>
12 </xs:schema>
```

Listing 4.5. The part of the XML schema representing the IDN-Node element.

The `VRApplicationMeta` section is quite simple: its purpose is to gather metadata about the life-cycle of the IDN-Node. It presents attributes such as the identifier of the application responsible for the creation, the author of the resource, and the time-stamp of the last modification occurred. Listing 4.6 shows the XML Schema representation of this section.

```
1  <xs:schema version="1.0">
2    <xs:complexType name="vrApplicationMeta">
3      <xs:all>
4        <xs:element name="IDNApplicationID" type="xs:string"
         minOccurs="0" maxOccurs="1"/>
5        <xs:element name="IDNAuthor" type="author" minOccurs="0"
         maxOccurs="1"/>
6        <xs:element name="IDNModifiedOn" type="xs:string"
         minOccurs="0"/>
7      </xs:all>
8    </xs:complexType>
9    <xs:complexType name="author">
10     <xs:all>
11       <xs:element name="IDNApplicationInstanceID" type="xs:string"
         minOccurs="0" maxOccurs="1"/>
12       <xs:element name="IDNHostIPAddress" type="xs:string"
         minOccurs="0"/>
13       <xs:element name="IDNHostName" type="xs:string"
         minOccurs="0" maxOccurs="1"/>
14       <xs:element name="IDNUser" type="xs:string" minOccurs="0"
         maxOccurs="1"/>
15     </xs:all>
16   </xs:complexType>
17 </xs:schema>
```

Listing 4.6. The part of the XML schema representing the `VRApplicationMeta` element.

The `VRIDNMeta` section contains two types of information: the node local name and the different types of Link. The first is a *local reference* used to refer the node

internally. In compliance with the InterDataNet Information Model 4.1.1, the Data Model describes Aggregation Links, Reference Links, (convenience) Back Links and Active Links 4.1.3. Each type of Link is represented by a dedicated section built as follows:

– *Meta (contains one or more Link(s))

- Link
  * Local Name
  * Value
  * Meta
  * Operations
    · Verb (GET, DELETE, PUT verbs are supported)
    · Description
  * id

the `LocalName` has the same function of the homonymous attribute defined for the `VRIDNMeta` section: it defines a local reference for the link, to be used internally. The `Value` specifies the URI LRI pointing to the referred resource, the `Meta` is a convenience field for custom descriptions and specification about the resource associated with the LRI, `Operations` defines operations allowed for that resource, specifying the HTTP Verb and the expected operation's outcome. This feature is intended to support the HATEOAS Principle, and will be detailed in the dedicated section. Finally, the `vr-id` element identifies each and every link contained in the IDN-Node. This is extremely useful to disambiguate partial update operation performed via the HTTP PUT verb. Indeed, without an identifier for each link in the list, it would not be possible to tell whether an attribute has been removed or changed (please note that the `Value` holding the LRI is modifiable as well).

Listing 4.7 shows an XML Schema representation of the `VRIDNMeta` section.

```
1 <xs:schema version="1.0">
2   <xs:complexType name="vrIdnMeta">
3     <xs:all>
4       <xs:element name="AggregationMeta" minOccurs="0"
        maxOccurs="1">
5         <xs:complexType>
6           <xs:sequence>
7             <xs:element name="AggregationLink" type="link"
              minOccurs="0" maxOccurs="unbounded"/>
8           </xs:sequence>
9         </xs:complexType>
```

```
10        </xs:element>
11        <xs:element name="BackLinkMeta" minOccurs="0" maxOccurs="1">
12          <xs:complexType>
13            <xs:sequence>
14              <xs:element name="BackLink" type="link" minOccurs="0"
                maxOccurs="unbounded"/>
15            </xs:sequence>
16          </xs:complexType>
17        </xs:element>
18        <xs:element name="ActiveMeta" minOccurs="0" maxOccurs="1">
19          <xs:complexType>
20            <xs:sequence>
21              <xs:element name="ActiveLink" type="link" minOccurs="0"
                maxOccurs="unbounded"/>
22            </xs:sequence>
23          </xs:complexType>
24        </xs:element>
25        <xs:element name="NodeLocalName" type="xs:string"
          minOccurs="0" maxOccurs="1"/>
26        <xs:element name="ReferenceMeta" minOccurs="0" maxOccurs="1">
27          <xs:complexType>
28            <xs:sequence>
29              <xs:element name="ReferenceLink" type="link"
                minOccurs="0" maxOccurs="unbounded"/>
30            </xs:sequence>
31          </xs:complexType>
32        </xs:element>
33      </xs:all>
34    </xs:complexType>
35    <xs:complexType name="link">
36      <xs:all>
37        <xs:element name="LocalName" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
38        <xs:element name="Value" type="xs:anyURI" minOccurs="0"
          maxOccurs="1"/>
39        <xs:element name="Meta" type="xs:string" minOccurs="0"
          maxOccurs="1"/>
40        <xs:element name="AllowedOperations" type="operations"
          minOccurs="0" maxOccurs="1"/>
41      </xs:all>
42      <xs:attribute name="vr-id" type="xs:int" use="required"/>
43    </xs:complexType>
44    <xs:complexType name="operations">
45      <xs:sequence>
46        <xs:element name="Operation" type="operation" minOccurs="0"
```

```
        maxOccurs="1"/>
47    </xs:sequence>
48  </xs:complexType>
49  <xs:complexType name="operation">
50    <xs:all>
51      <xs:element name="Verb" type="xs:string" minOccurs="0"
        maxOccurs="1"/>
52      <xs:element name="Description" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
53    </xs:all>
54  </xs:complexType>
55 </xs:schema>
```

Listing 4.7. The part of the XML schema representing the `VRIDNMeta` element.

The `ManagementMeta` element supports properties described in 4.1.2.2 section. It provides room for `LicensePolicy`, `PrivacyPolicy`, `ProvenanceManagement`, `TimePolicy` and `VersioningPolicy`.

The `LicensePolicy` element is basically made up of two parts: the `DataLicensing` element contains the licensing policy that has to be applied for data contained in the `ApplicationData` section, while the `NodeLicensing` element contains the licensing policy in force for the node when used (aggregated) in documents. More licenses can be specified for both `DataLicensing` and `NodeLicensing` elements. The `PrivacyPolicy` element, basically says if the node is marked to contain identifying information, as discussed in the 4.1.2.2 section. The `ProvenanceManagement` element contains information about the life cycle of the node. It holds data concerning actors who are involved in the previous modification, so that, by leveraging the versioning capability of the corresponding architectural component, is possible to retrieve the full history of the node, including actors who took part in it. The `TimePolicy` element defines `AutoDestroyOn`, `ValidityNotAfter` and `ValidityNotBefore` sub-elements specifying whether the node must be destroyed or invalidated after (or before) a certain date or not. Finally, `VersioningPolicy` specify which versioning policy should be applied to the node.

Listing 4.8 shows the XML Schema representation of the `ManagementMeta` element.

```
1  <xs:schema version="1.0">
2    <xs:complexType name="managementMeta">
3      <xs:all>
4        <xs:element name="LicensePolicy" type="licensePolicy"
         minOccurs="0" maxOccurs="1"/>
5        <xs:element name="PrivacyPolicy" type="privacyPolicy"
         minOccurs="0" maxOccurs="1"/>
6        <xs:element name="ProvenanceManagement"
```

```
      type="provenanceManagement" minOccurs="0" maxOccurs="1"/>
7     <xs:element name="TimePolicy" type="timePolicy"
      minOccurs="0" maxOccurs="1"/>
8     <xs:element name="VersioningPolicy" type="xs:string"
      minOccurs="0" maxOccurs="1"/>
9    </xs:all>
10  </xs:complexType>
11  <xs:complexType name="licensePolicy">
12   <xs:all>
13    <xs:element name="DataLicensing" minOccurs="0">
14     <xs:complexType>
15      <xs:sequence>
16       <xs:element name="DataLicense" type="licenseType"
         minOccurs="0" maxOccurs="unbounded"/>
17      </xs:sequence>
18     </xs:complexType>
19    </xs:element>
20    <xs:element name="NodeLicensing" minOccurs="0">
21     <xs:complexType>
22      <xs:sequence>
23       <xs:element name="NodeLicense" type="licenseType"
         minOccurs="0" maxOccurs="unbounded"/>
24      </xs:sequence>
25     </xs:complexType>
26    </xs:element>
27   </xs:all>
28  </xs:complexType>
29  <xs:complexType name="licenseType">
30   <xs:simpleContent>
31    <xs:extension base="xs:string">
32     <xs:attribute name="vr-id" type="xs:int" use="required"/>
33    </xs:extension>
34   </xs:simpleContent>
35  </xs:complexType>
36  <xs:complexType name="privacyPolicy">
37   <xs:all>
38    <xs:element name="Identifying" type="xs:boolean"
      minOccurs="0" maxOccurs="1"/>
39   </xs:all>
40  </xs:complexType>
41  <xs:complexType name="provenanceManagement">
42   <xs:all>
43    <xs:element name="IDNPreviousApplicationID" type="xs:string"
      minOccurs="0"/>
44    <xs:element name="IDNPreviousAuthor" type="author"
```

```
        minOccurs="0"/>
45      <xs:element name="IDNPreviousModifiedOn" type="xs:string"
        minOccurs="0"/>
46      <xs:element name="ProvenancePolicy" type="xs:string"
        minOccurs="0" maxOccurs="1"/>
47    </xs:all>
48  </xs:complexType>
49  <xs:complexType name="timePolicy">
50    <xs:all>
51      <xs:element name="AutoDestroyOn" type="xs:string"
        minOccurs="0"/>
52      <xs:element name="ValidityNotAfter" type="xs:string"
        minOccurs="0"/>
53      <xs:element name="ValidityNotBefore" type="xs:string"
        minOccurs="0"/>
54    </xs:all>
55  </xs:complexType>
56 </xs:schema>
```

Listing 4.8. The part of the XML schema representing the `ManagementMeta` element.

`Activity` is the section representing the core part of the scripting capability and is present in the representation of Activity Nodes only. It has three main sections, namely `Input`, `Output` and `Script`. The `Input` section puts together the information required to target and process inputs as `Parameters`, such as the `Type`, as explained in section 4.1.3.1, the `Source`, i.e. the Active Link identifying the input required for the computation, the `Variable`, i.e. the variable name used for that input in the scope of the script, and a textual `Description` of the script. The `Source` element creates a binding between the variable name and the object(s) retrieved through the Active Link resolution.

The `Output` section defines properties concerning output of the script, i.e. `ContenteType`, `Content Encoding` and a textual description.

Finally, the `Script` section hosts the actual script within the `Exe` element. There are also a `Language` element to specify the scripting language (currently the architecture supports Groovy [KGK+07]), a `ContentEncoding` with the same purposes described above and an optional `RemoteLocation` for the script externalization (currently not supported by the architecture).

Listing 4.9 shows the XML Schema representation of the `Activity` element.

```
1  <xs:complexType name="activity">
2        <xs:all>
3              <xs:element name="Input" type="input" minOccurs="1"
               maxOccurs="1"/>
4              <xs:element name="Output" type="output"
```

```
                minOccurs="1" maxOccurs="1"/>
5               <xs:element name="Script" type="script"
                minOccurs="1" maxOccurs="1"/>
6       </xs:all>
7   </xs:complexType>
8
9   <xs:complexType name="input">
10      <xs:sequence>
11              <xs:element name="Parameter" type="parameter"
                minOccurs="1" maxOccurs="1"/>
12      </xs:sequence>
13  </xs:complexType>
14
15  <xs:complexType name="parameter">
16      <xs:sequence>
17              <xs:element name="Type" type="xs:String"
                minOccurs="1" maxOccurs="1"/>
18              <xs:element name="Source" type="xs:anyURI"
                minOccurs="1" maxOccurs="1"/>
19              <xs:element name="Variable" type="xs:String"
                minOccurs="1" maxOccurs="1"/>
20              <xs:element name="Description" type="xs:String"
                minOccurs="0" maxOccurs="1"/>
21      </xs:sequence>
22  </xs:complexType>
23
24  <xs:complexType name="output">
25      <xs:all>
26              <xs:element name="ContentEncoding" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
27              <xs:element name="ContentType" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
28              <xs:element name="Description" type="xs:String"
                minOccurs="0" maxOccurs="1"/>
29      </xs:all>
30  </xs:complexType>
31
32  <xs:complexType name="script">
33      <xs:all>
34              <xs:element name="RemoteLocation" type="xs:anyURI"
                minOccurs="0" maxOccurs="1"/>
35              <xs:element name="Language" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
36              <xs:element name="ContentEncoding" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
```

```
37                <xs:element name="ContentType" type="xs:string"
                  minOccurs="1" maxOccurs="1"/>
38                <xs:element name="Exe" type="xs:String"
                  minOccurs="0" maxOccurs="1"/>
39                <xs:element name="Description" type="xs:String"
                  minOccurs="0" maxOccurs="1"/>
40          </xs:all>
41  </xs:complexType>
```

Listing 4.9. The part of the XML schema representing the `Activity` element.

4.2.2 The Addressability Principle in InterDataNet

According to the InterDataNet Information Model, every resource must have a dereferenceable identifier. Since InterDataNet defines only one type of resource on the interface exposed to client applications, it is possible to rewrite the statement as follows: every IDN-Document must have a dereferenceable identifier. This does not mean that only a single identifier is associated to an IDN-Document. There could be aliases URIs resolving the same resource. The point here is that an IDN-Document must have *at least* one dereferenceable, unique name.

The fact that is possible to address a document, doesn't imply that, by issuing two retrievals at different times, the identical object will be returned. In fact, resources change their state with time, as a consequence of the clients update operations. In InterDataNet the document concept is heavily affected by state transitions. Documents are made by nodes which are reused to build other documents, and the expected modification rate may be considerable. Therefore, the document can be considered an *ephemeral* entity. It is important to clarify the point, because in the everyday life we are familiar to deal with very static documents. Papers, driving licenses, id cards, prescriptions are all examples of document that aren't very likely to change. Driven by habit, we can be tempted to assume a document as a stable set of information. There is no guarantee that this assumption is true on the Web. This is even more true for a framework like InterDataNet that encourages information reuse and collaboration around information grains.

Even though documents are ephemeral, this is not the case of the nodes. Unlike documents, nodes are self-containing and self-descriptive and although they can be modified (in terms of content, properties and so on), the change does not affect their structure (e.g., it is not possible to add new sections to a node as it is possible to add new nodes to a document). As previously mentioned, the canonical LRI (the universal, unique identifier) is associated with the nodes, and defined in a dedicated part of their Data Model.

As discussed so far, there are still some points to make clear: the LRI is associated with the IDN-Node, but the resource defined on the interface is the document. How is it possible to link the two concepts? How is it possible to target a document through an LRI?

Let's start saying that by issuing a request on a certain LRI, the request will affect the *document* having the IDN-Node matching that LRI as *root node*. The concept of root node is important because defines the node driving the state transfer for the current communication. It acts as a reference point for the ongoing operation. For example, as detailed below, an HTTP GET issued for a document declaring a certain LRI could be translated in natural language as follows: "resolve the document (traverse the graph and extract a sub-graph) starting form the node with this LRI (assuming it as the entry point).".

Similarly, issuing an HTTP PUT on a certain LRI would means "create/update the document passed in the request body starting form the node with this LRI.".

Returning to the retrieval (HTTP GET) example, it is critical to specify the criteria to use for the sub-graph extraction. This mechanism is embedded in the definition of the IDN-URI (see below), i.e. the sub-class of URI used for documents. However, it is also important to point out that the IDN-Document resolution is performed by following Aggregation Links only. This is natural reminding the container-content semantics of the aggregation relation. By requesting a document, it makes sense to return all the information pieces constituting the document itself.

To serve the purposes of InterDataNet a class of HTTP URIs is defined: the IDN-URI is an HTTP URI with the following form.

```
http://idn_host/VirtualResource/!VR/idn_document_name/$p{idn_inner_path}/
$c{content_path}/$v{idn_version}/$r{idn_resolution_depth}
```

The meaning of the parts of the IDN-URI is explained in the following.

http is the application protocol. HTTP is the only protocol supported.

idn_host is the hostname and port of the endpoint exposing the service.

VirtualResource is the name of the layer implementing the IDN-Document abstraction. It is mandatory.

!VR is a conventional marker for all the IDN-URIs implemented by a Virtual Resource. It is mandatory.

idn_document_name is the name of the IDN-Document (i.e., the name of the root IDN-Node, for the current document). It is a path with no length restrictions. For example, a valid idn_document_name is the following: `ACME/employee/2387/salary/may`. It is mandatory.

idn_inner_path is a path identifying an inner IDN-Node section. For example, it is possible to request a representation of the node showing the `ApplicationData` section only. This will be done by specifying the `idn_document/$p ApplicationData` IDN-URI.

It is also possible to narrow the request by adding a sub-section to the parent section. For example, if the requester is interested in the `Content` only the idn_inner_path will be: `idn_document/$p ApplicationData/Content`. The `$p` selector is mandatory for using this feature. It is optional and it is enabled only for GET requests.

content_path is an XPath expression to target elements contained in the `Content` section. If this parameter is present, it *must* follow a `$pApplicationData/Content` part. This is due to the semantics of the selector. Since it digs in the node content, it must be addressed by specifying the precise location of the `Content` section, within the node.

The reader shouldn't be mislead by the fact that an XPath expression is adopted. The aim here is to provide a selection mechanism which is format independent, i.e. valid for XML, Json, or (hopefully) whatever representation. In order to accomplish this task, has been identified a subset of the XPath capabilities fitting the requirements of the Json language [Goe07]. This subset defines the operations that abstract from the technology and therefore is suitable to our purpose.

Table 4.3 [Goe07] lists the selected syntax for the $c parameter. Please note that there is no mechanism to select attributes which are a feature of XML only. To stay general implies that not all the language peculiarities can be supported. Table 4.4 shows some examples of XPath syntax for the `store` resource shown in Listing 4.10 [Goe07].

XPath	Description
/	the root element
.	the current element
/	child operator
..	parent operator
//	recursive descent
*	wildcard. All objects/elements regardless their names.
[]	subscript operator
|	Union operator
[]	applies a filter (script) expression

Table 4.3. the subset of the XPath syntax chosen for the $c parameter.

XPath	Result
`/store/book/author`	the authors of all books in the store
`//author`	all authors
`/store/*`	all things in store, which are some books and a red bicycle.
`/store//price`	the price of everything in the store.
`//book[3]`	the third book
`*`	wildcard. All objects/elements regardless their names.
`//book[last()]`	the last book in order.
`//book[position()<3]`	the first two books
`//book[isbn]`	filter all books with isbn number

Table 4.4. the subset of the XPath syntax chosen for the $c parameter.

```
{
   "store":{
      "book":[
         {
            "category":"reference",
            "author":"Nigel Rees",
            "title":"Sayings of the Century",
            "price":8.95
         },
         {
            "category":"fiction",
            "author":"Evelyn Waugh",
            "title":"Sword of Honour",
            "price":12.99
         },
         {
            "category":"fiction",
            "author":"Herman Melville",
            "title":"Moby Dick",
            "isbn":"0-553-21311-3",
            "price":8.99
         },
         {
            "category":"fiction",
            "author":"J. R. R. Tolkien",
            "title":"The Lord of the Rings",
            "isbn":"0-395-19395-8",
            "price":22.99
         }
      ],
      "bicycle":{
         "color":"red",
```

```
33          "price":19.95
34        }
35     }
36  }
```

Listing 4.10. A javascript store resource.

idn_version is the selector to specify the version of the current document. The `$v` selector is mandatory for using this feature. This feature is optional and it is enabled only for GET requests.

idn_resolution_depth specifies the number of hops to perform while retrieving the document. For example, given a document with three levels of aggregated nodes, by specifying `this/document/$r1` only the nodes reachable in no more than one hop will be included in the returned document. Fig. 4.16 depicts the document of the example. The edges are annotated with the resolution level number. In the previous example, only the nodes reachable by a 1-labeled edge will be returned. This feature is optional and is enabled for GET requests only.

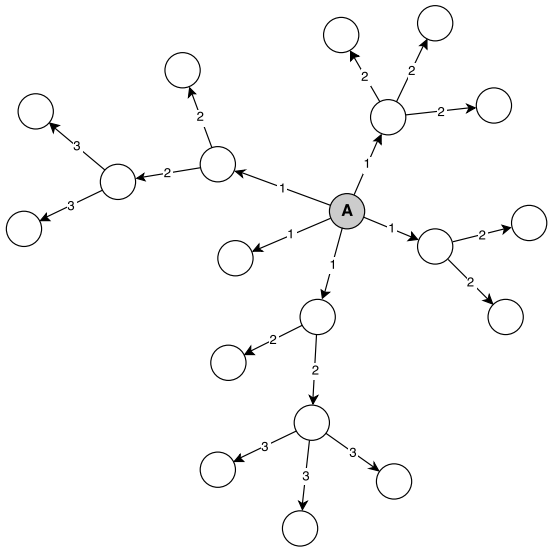


Figure 4.16. *An example of multi-level document. The root node is identified by the A letter, and the edges are labeled with the number of hops needed to reach them from the root.*

Table 4.5 summarizes the characteristics of the IDN-URIs.

While requesting a document resolution, is possible to *chain selectors*, with additive effect. For example, by invoking the following IDN-URI

```
GET http://www.abc.it:8080/VirtualResource/!VR/driving_license/$pVrIdnMeta
/NodeLocalName/$r3
```

the resource returned will be made up of local names of the IDN-Nodes reachable from the driving license root node in no more than 3 hops.

Given a document root node LRI, is always possible to issue a complete document resolution by adding the `/` character at the end of the LRI. This operation is called *infinite GET* and retrieves each and every node reachable by the root node, following Aggregation Links.

In summary, in order to retrieve a document made up of a single node, simply invoke a GET on the LRI of that node. In order to retrieve a document made up of more nodes, two options are available:

1. to issue a complete resolution ending the LRI with a slash `/` character
2. to issue a partial resolution using the `idn_resolution_depth` parameter.

To conclude, the fact that some IDN-URIs are not defined for creation and modification operations doesn't mean that is not possible to reach a similar operational fine-grain level. These operations require a resource specification (in the HTTP request body) and this allows the client to be very specific with the request. Conversely, the deletion operation is defined on single node document only. This aspects will be tackled in the next session.

IDN-URI subpart	**Accepted Values (regex)**	**HTTP Verb**
`idn_document_name`	[^\][/\w]*[^\]	GET, PUT, DELETE
`idn_inner_path`	[^\][/\w]*[^\], iff the character sequence identifies a valid node's section.	GET
`content_path`	a valid XPath expression compliant with the syntax shown in 4.3	GET
`idn_version`	[^.][.\d]*[^.]	GET
`idn_resolution_depth`	[\d]	GET

Table 4.5. Summary table for the IDN-URI, with regex definition of accepted values and HTTP Verbs for which the IDN-URI is enabled.

4.2.3 The Uniform Interface Principle in InterDataNet

In order to keep the interface as clean as possible, only three major operations are defined for the document resource: `read`, `write` and `delete`. Their meaning is pretty trivial, the `read` operation deliveries a representation of the resource to the client, the `write` operation transfers a state on the resource, while the `delete` operation removes the resource from the system.

The `write` unifies the `create` and `update` operations. It works similarly to the `save` of the operating systems. If a resource does not exist, it will be created; it will be updated otherwise. For the sake of completeness, the HTTP OPTIONS verb is also implemented for requesting information about the communication options available on the request-response chain identified by the URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

Table 4.6 shows the mapping between operations and HTTP verbs in InterDataNet.

Operation	**HTTP Verb)**
`READ`	GET
`CREATE`	PUT
`UPDATE`	PUT
`DELETE`	DELETE

Table 4.6. Mapping between operations and HTTP verbs.

When a client issues an HTTP GET for a resource, it should specify the preferred format using the `Accept` header. If this specification is missing, InterDataNet will use `text/html` as default value. The HTTP DELETE verb does not require any special header, while the HTTP PUT verb requires both `Accept` and `Content-Type`. In fact, issuing a PUT on a resource implies to submit an HTTP request with a body which *must* contain a valid resource.

Table 4.7 shows the request-response details of HTTP GET and DELETE requests.

When a client issues an HTTP PUT, it decides the name of the resource to be created or modified. This is a typical RESTful pattern for the PUT. In compliance with REST, a second approach could be chosen, i.e. the POST verb. The POST semantics is slightly different. It is used to create a sub-resource of a pre-existing resource, with the server side managed name [Rot09]. For example, a creation with the POST verb suits well the following scenario: the server exposes a blog post `comment` resource, and allows the client to create new blog post comments as sub-resources of `comment`. To accomplish this task, the

Outcome	HTTP Status
`Positive`	200 ok the resource is returned successfully.
`Client Error`	404 not found the server can't find the requested resource. 415 unsupported media type the server will not accept the request, because the media type is not supported (HTTP GET requests only).
`Server Error`	500 internal server error the request can't be fulfilled for a generic server error. 502 bad gateway the request can't be fulfilled because other distributed components of the middleware have failed. 504 gateway timeout the request can't be fulfilled because other distributed components of the middleware can't be reached.

Table 4.7. Operation outcomes and status messages for the HTTP GET and DELETE.

client issues a POST carrying the new comment in the request body, specifying the URL `http://.../post/827/comment`. The server sends an HTTP response with the `202 accepted` status, including the `http://.../post/827/comment/4` `Location` header.

This approach is not suitable for the document resource, mainly because of the server side managed name. Consider the case a client is willing to create an invoice document for the ACME enterprise. By following the POST approach the best it could do is to issue a request to the `http://.../document` endpoint exposed by the middleware. Since InterDataNet is content-agnostic, it really couldn't expose other resource than the document (e.g. invoices) and it wouldn't make sense to give names different than `http://.../document/{id}`.

However, a very advised practice in REST is to give resources so-called *nice URIs*, i.e. names that are meaningful (to humans) for that resource. For the sake of clarity, `http://.../acme/invoice/23` is a meaningful URI for the invoice resource while `http://.../Hk201/gt56wtf7fd.txt` is not.

Clients may be very interested in assigning nice URIs to resources and the

only way to support this is to delegate them the responsibility of the name choice. It makes sense because the client creating a resource is the one to know what the resource is about (the novel document will contain an invoice rather than weather forecasts).

The PUT verb is also used for updates. The resource wrapped in the request body plays a very important role here. The middleware assumes an "update what you declare" strategy, which means that is possible to modify parts of the document resource by including in the resource representation only the information the client is actually willing to change. This strategy opposes the "update by substitution" strategy where the client is forced to transfer the whole new state, including the unchanged parts.

As Fig. 4.14 shows, the node object is quite complicated, and since the document is an aggregation of different nodes, it can be even more complicated. Implementing the "update what you declare" strategy results in an overall simplification that should not be underestimated. Despite its simplicity, this approach still require strictness in defining the resource carrying the new state. To modify a subsection is mandatory to to specify the context by including its super-section. For example, referring to Fig. 4.14, in order to change the content type to `text/plain`, the object

```
1 <VRNode lri="http://the/node/lri">
2   <VRApplicationData>
3     <ContentType>text/plain</ContentType>
4   </VRApplicationData>
5 </VRNode>
```

must be included in the request body.

Please note that siblings of the `Application Data` and `Content Type` element don't appear in the node representation. If an HTTP GET is issued for the resource, the full representation modified accordingly will be returned. In a scenario enforcing the "update by substitution" policy, the same request would have erased all the sections but the ones specified in the request Body.

Nevertheless, the "update what you declare" strategy introduces some critical issues which must be addressed thoroughly. First, it is possible to obtain a deletion-like effect on sections: if the object

```
1 <VRNode lri="http://the/node/lri">
2   <VRApplicationData>
3     <ContentType/>
4   </VRApplicationData>
5 </VRNode>
```

is submitted, the content of the `Contet Type` section will be emptied. This behavior is coherent because if the client is entitled to change the value of some attributes, it can also set them to `null`. This statement holds both for leaves and for sections of the node tree, meaning that if the object

```
<VRNode lri="http://the/node/lri">
  <VRApplicationData/>
</VRNode>
```

is submitted, the whole `Application Data` section will be erased (further details will be presented in section 5.2.2).

Now, consider the case of a client willing to change a list element, say a `Reference Link` of a document, whose previous state is the following

```
<VRNode lri="http://.../poi/1381504340558">
  <VRIDNMeta>
    <ReferenceMeta>
      <ReferenceLink>
        <LocalName>sensor</LocalName>
        <Value>http://.../sensor/260</Value>
      </ReferenceLink>
      <ReferenceLink>
        <LocalName>sensor</LocalName>
        <Value>http://.../sensor/525</Value>
      </ReferenceLink>
    </ReferenceMeta>
    <BackLinkMeta/>
    <IncomingChangeMeta/>
  </VRIDNMeta>
</VRNode>
```

To accomplish the task, the client submits the resource

```
<VRNode lri="http://.../poi/1381504340558">
  <VRIDNMeta>
    <ReferenceMeta>
      <ReferenceLink>
        <LocalName>sensor</LocalName>
        <Value>http://.../sensor/770</Value>
      </ReferenceLink>
    </ReferenceMeta>
    <BackLinkMeta/>
```

```
10      <IncomingChangeMeta/>
11    </VRIDNMeta>
12  </VRNode>
```

.

The problem here is that the operation is ambiguous. Which goal does the client want to achieve? Does it want to change the `Value` of the first Reference Link? Does it want to change the `Value` of the second Reference Link? Does it want to add a barely new Reference Link? The key point is that it isn't possible to address a specific list item because, in the scope of the node, can't be uniquely identified. This is the reason why the `vr-id` attribute is introduced. The `vr-id` is defined to be an integer which must be *unique in the scope of the list*. Different lists, say `Refernce Meta` and `Aggregation Meta` can both wrap an item having `vr-id=12`, but two items belonging to the same list, say two `Aggregation Link`s, must have different `vr-id` values. With this identification integration, clients can target list items specifically, and the ambiguity problem is solved. In the following, is presented the resource with the `vr-id`s in place.

```
1  <VRNode lri="http://.../poi/1381504340558">
2    <VRIDNMeta>
3      <ReferenceMeta>
4        <ReferenceLink vr-id="2">
5          <LocalName>sensor</LocalName>
6          <Value>http://.../sensor/260</Value>
7        </ReferenceLink>
8        <ReferenceLink vr-id="3">
9          <LocalName>sensor</LocalName>
10         <Value>http://.../sensor/525</Value>
11       </ReferenceLink>
12     </ReferenceMeta>
13   </VRIDNMeta>
14 </VRNode>
```

InterDataNet is a collaborative environment where different actors can cooperate on documents. Such scenario is inherently prone to resources' synchronization and consistency problems. Consider the case of a client requesting a resource for modification. At the time t_0 the client retrieves a resource in a state S_0. At the time t_1 another client retrieves the same resource, modifies it, and submits it with the new state S_1 at the time t_2. At the time t_3, the same operation is performed by the first client (transferring a state S_2) which is not aware of the operation performed by the second client. This results in a state conflict because S_2 assumed a previous state S_0 although the resource was already in the S_1 state.

To tackle this problem, REST advises to use the `Etag` which is a deterministic

server-generated code identifying the current state of the resource. When the resource is requested, is delivered to the client together with the `Etag`. The server won't accept a resource update submission omitting the `Etag`. When the server receives a valid request, it checks the presented `Etag` against the one computed from the current state of the resource. If the check is successful, i.e. codes are equal, the new state is transferred and the `Etag` is updated, otherwise a `412 precondition failed` error is sent.

Fig. 4.17 shows the flow chart of the `Etag` submission and check.

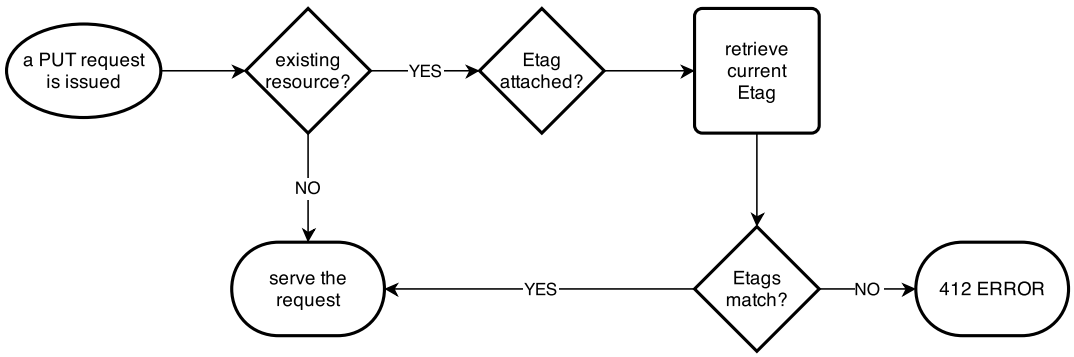


Figure 4.17. *The flow chart of the `Etag` submission and check*

In normal RESTful Web services, the `Etag` is passed in the request header. This is not possible for InterDataNet because, as said before, the exchanged resource, i.e. the document, is ephemeral. A document can be fully or partially resolved using the resolution depth parameter, and, even if the common nodes are equal, they could be considered different resources just because the former contains more nodes than the latter. This interpretation is not wrong in itself, but is useless to address synchronization and consistency problems in the IDN-Document domain. What is really needed, is to convey state information node by node. That's why the `Etag` attribute is bound to every node, within the `VRNodeInfo` section.

Issuing a PUT on a document, means to perform a `write` operation on each and every node included in that document. This can result in a global creation, a global update or a partial creation and update, depending on the server side nodes' state.

Figure 4.18 describes a `write` attempt on a document: first a client requests a document, say issuing a complete resolution. The document is returned with the `Etag` attributes set to the current nodes' state. The client updates the nodes `A` and `B`, adds a barely new node `C` and submits the document to the system via an HTTP PUT. At this point, the system verifies the existence of the nodes and finds out that `A` and `B` are already in the system, while `C` is not. Therefore, the

`write` operation will have different semantics: an update for nodes `A` and `B` and a creation for the node `C`.

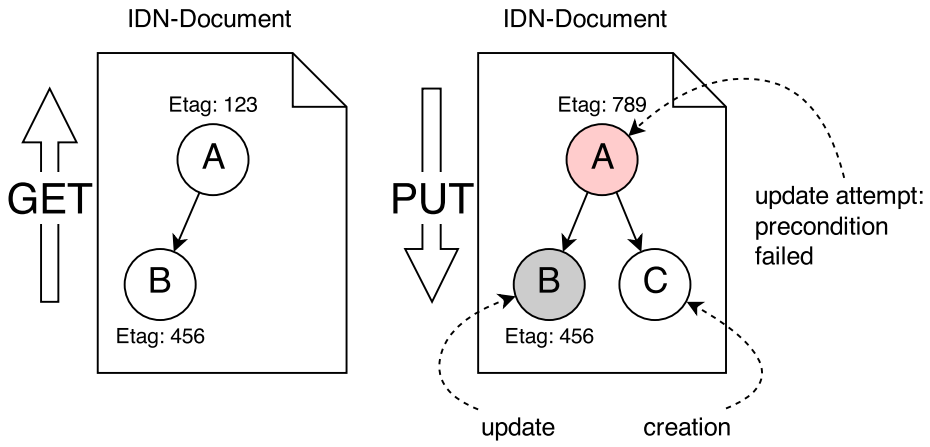


Figure 4.18. *Possible outcomes of a `write` operation performed on a document.*

As said before, the update operation implies a consistency check requiring the `Etag` verification. The system runs the deterministic function for `A` and `B` and checks outputs against `Etag`s passed by the client. The node `B` passes the test and is updated, while the `A` fails the test and a `precondition failed` error message is produced. The node `C` is simply created in the system. From now on, issuing a `write` operation on `C` will require an `Etag` declaration.

Finally, the server prepares the response including the document in the body. Because of the failed update for the node `A`, a `500 internal server error` Status Code is returned. The document will present a `VRDocInfo` section with the `INCOMPLETE WRITE` warning and a TargetedLRI list specifying the `A` node. In the `VRDocInfo` section paired with the `A` node, `HttpStatus` will be set to `412 precondition failed`. Conversely, the `HttpStatus` will be set to `200 ok` and `201 created` for nodes `B` and `C`.

Table 4.8 shows the request-response details of the HTTP PUT request. Please note that these status messages refer to the response as a whole. More details can be found in the dedicated sections of the document resource.

Finally, Table 4.9 specifies the mapping between the errors and warnings described in Tables 4.2 and 4.1 and the status message for the document resource described in Table 4.8.

4.2.4 The Stateless Communication Principle in InterDataNet

The Stateless Communication principle has the major purpose of providing scalability to RESTful architectures. If the server has to maintain a state for every

Outcome	**HTTP Status**
`Positive`	202 created if the document root node is created. 200 ok if the `write` operation is successful for every node in the document.
`Client Error`	400 bad request the request cannot be fulfilled due to bad syntax. 412 precondition failed if the document root node fails the `Etag` check during an update operation. 415 unsupported media type the server will not accept the request, because the media type is not supported.
`Server Error`	500 internal server error the request can't be fulfilled for a generic server error or the server try a best-effort fulfillment, detailing the errors in the request body. 502 bad gateway the request can't be fulfilled because other distributed components of the middleware have failed. 504 gateway timeout the request can't be fulfilled because other distributed components of the middleware can't be reached.

Table 4.8. Operation outcomes and status messages for the HTTP PUT.

Type	Code	HTTP Status
`warning`	Incomplete Write	500
`warning`	Missing Etag	500
`error`	Root Uniqueness Error	500
`error`	Dependency Cycle Detected	500
`warning`	Incomplete Inner Resource Resolution	500
`warning`	Incomplete Resolution	502
`error`	Root Not Found	400
`error`	LRI syntax Error	400
`error`	Aggregation LRI syntax Error	400
`error`	VR-ID Uniqueness Error	400
`error`	Bad PUT Document Name	400

Table 4.9. Warnings and Errors mapping with the Status Messages.

client's request and the their number increases, the server will need more and more resources to avert a crash. Conversely, if every request provides all the information for its management, the server won't have problems in managing a considerable amount of clients.

In InterDataNet every request is self-descriptive and all the information necessary for its fulfillment is contained in HTTP headers and body. As seen in sections 4.2.1 and 4.2.3, different strategies are put in place to vehicle the communication state through the resource, such as `*Info` elements. In fact, the server-side of the architecture does not retain the communication state in any way, in compliance with the REST principle.

The non-implementation specific details for making the communication stateless between clients and the InterDataNet server-side has been already addressed in previous sections, so they won't be repeated here.

4.2.5 The HATEOAS Principle in InterDataNet

Hypermedia as the Engine of the Application State, is a controversial principle. Many argue that is the most misunderstood part of the Fielding dissertation. However, after some clarifications by Fielding, the concept has become clear [Fie08].

Basically HATEOAS means that the state of the application should be driven by hypermedia provided with resources.

Fielding himself gives his definition of hypermedia: "When I say hypertext, I mean the simultaneous presentation of information and controls such that

the information becomes the affordance through which the user (or automaton) obtains choices and selects actions. Hypermedia is just an expansion on what text means to include temporal anchors within a media stream; most researchers have dropped the distinction.

Hypertext does not need to be HTML on a browser. Machines can follow links when they understand the data format and relationship types."

Pragmatically, when a resource is delivered, it should contain the directions to allow the client to take the next steps. InterDataNet resources support this approach by explicitly including in the node edges towards next resources. In section 4.1.2 has been discussed the structure of the node and the fact that all the link types are part of it. Therefore, requesting a document means to have a set of nodes linked with other nodes that, if they aren't part of the current document, can be requested to browse step by step the underlying graph of information.

As detailed in 4.2.1, the node object has a dedicated section for every type of link: `AggregationMeta`, `ReferenceMeta`, `BacklinkMeta` and `ActiveMeta`. They contain lists of link items, having each a `Value` element with the actual URL, and a list of operations available for that URL. To every operation are associated the HTTP verb and a description.

As argued in section 4.2.3, the available HTTP verbs are GET, DELETE, and PUT mapping to `read`, `delete` and `write` operations. Basically, these actions are available for every resource. It is worth pointing out that the possibility of taking an action shouldn't be confused with the clients' actual authorization for undertaking it. This is a separate issue that is addressed by the security component, fully decoupled from the implementation of REST in InterDataNet.

Chapter 5

InterDataNet Architecture

The design and implementation of the InterDataNet architecture has represented a significant effort in the context of this thesis. All core components and services have been studied and prototyped in order to obtain a working proof of concept.

InterDataNet is a layered architecture designed with separation of concerns and information hiding principles [HL95] in mind. It is composed of three main layers with an additional adaptation layer detached from the architecture, and some additional horizontal services, such as security. An InterDataNet full stack requires a Virtual Resource layer implementing the IDN-Document abstraction, an Information History layer implementing the versioning service, a Storage Interface layer implementing the persistence management, and one or more Adapter(s) interfacing with outer data sources. A deployment of a stack is a node of a network of peers, constituting the real InterDataNet cooperation environment. Each peer is responsible for documents created in its domain, and different peers can interact to implement a super-graph of interconnected documents.

Figure 5.1 shows a comprehensive picture of the InterDataNet system, including outer data sources, components of a single InterDataNet instance, the network of peers, the information model and a representation of a document resource. The dashed clouds at the bottom represent outer data sources which provide information to the architecture, in their own custom format. The adaptation layer (ADPT) interfaces with these data sources and performs a transformation of the information to comply with the InterDataNet formalism. Such information proceeds through the architecture until the Virtual Resource (VR) layer, which implements the document abstraction. At this level, data are exposed as docu-

ments which can be composed to build new richer graphs (i.e., other documents). InterDataNet is not limited to the management of data coming from external sources. In fact, is possible to create InterDataNet native data using the RESTful interface. Analogously to the case of information coming from outer providers, these data will be exposed by the architecture in document form.

Different Virtual Resource layers belonging to separate architecture instances can interact, realizing a distributed graph of information. The IDN-Document is depicted as a graph with four vertexes coming from different Virtual Resources to emphasize the distributed nature of the model.

Finally, on top of the model, there is a representation of the document in one of three data formats currently supported by the implementation: HTML, XML and Json.

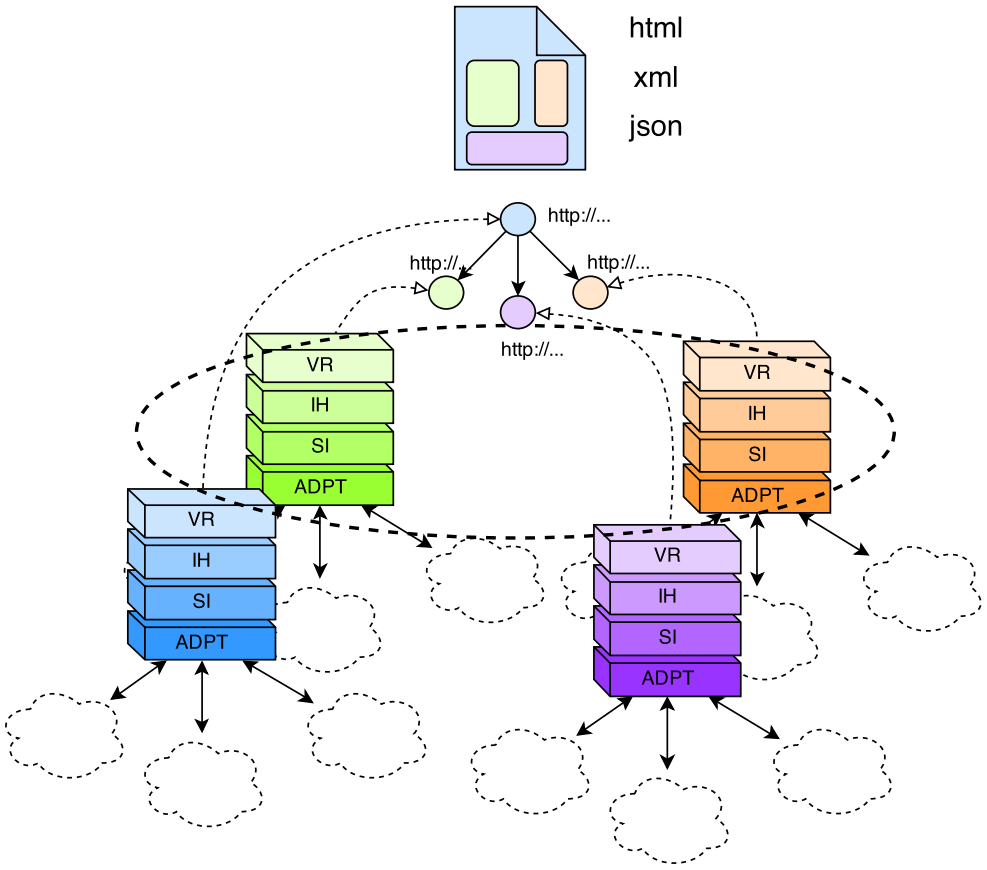


Figure 5.1. *An overall view of the InterDataNet system and the document resource.*

In the course of this chapter the InterDataNet architecture will be discussed. The implemented prototype stack includes Virtual Resource, Storage Interface and Adapter(s) modules. The Information History has not been implemented,

therefore versioning and historization capabilities for information grains are not currently enabled. Nevertheless, the Virtual Resource layer has been designed and implemented to support these capabilities so as a future integration of the Information History layer in the stack will be as simple as possible.

Aside from stack modules, the architecture leverages three additional services, namely, naming system embodied by the Logical Domain Naming System module (LDNS), search engine implemented by the InterDataNet Search Engine (ISE) module, and security framework which has been studied and designed, but not implemented yet.

In the context of the InterDataNet Architecture, my work has focused on the Virtual Resource, Storage Interface and Adapter(s) design and implementation, and the security framework development (not included in this thesis), while I supported the design and implementation of LDNS and ISE modules. Because of this, LDNS and ISE modules will be discussed marginally, only when their functionalities are necessary to the explanation of other modules.

In this architecture implementation, modules are separate Web Applications exposing one or more HTTP interfaces. Therefore, unless otherwise specified, while discussing the architectural components, "module" and "application" terms are interchangeable. Not to confuse the reader, from now on Applications that exploit the document representation exposed by the Virtual Resource module, consuming and producing IDN-Document resources will be called "InterDataNet Compliant Applications", in the following abbreviated as "IDN Compliant Applications" or "IDN-CA". They are called "compliant" because they know the InterDataNet Information Model. It is a general definition to address clients that interact with the upper interface of Virtual Resource.

Finally, a last specification: although in the InterDataNet modeling discussion has been introduced the possibility for an IDN-Node to have one or more alias (see sections 4.1.2 and 4.2.2), this feature is not part of the current implementation where IDN-Nodes have one and only one name, i.e. the canonical LRI.

5.1 An Architectural Overview

In this section will be given a brief architectural overview, before approaching more detailed discussions focused on individual components.

Within the Architecture, services are organized in layers whose interaction is mediated by the REST interface of adjacent levels. Stratification major principles are the following:

Speration of Concerns : a specific function is assigned with each layer. Each layer is concerned with the duties of his competence only, and delegates other

tasks.

Information Hiding : each level exposes to the outside only the information essential to communicate with the adjacent levels.

Every layer is requested to respond properly to calls coming form adjacent layers, and the internal logic processing these calls is not visible form the outside.

The stratification as design pattern has proven to be a winning choice not only for telecommunication network (ISO/OSI) but also for software design [Fow02, MRSS96, AZ05]. Long terms advantages include a better understanding, easing the dissemination of software components, and a reduction of costs compared with different methodologies [ZEWH95] when looking for a scalable solution incrementally implementable, promoting reuse and interoperability. Adopting a stratification technique means to implement a loose coupling of components and this minimizes the impact of interventions on individual parts of the architecture. Indeed, if a component changes, other components don't need to, provided that interfaces remain the same. In [GKT02] authors argue that the stratification is beneficial on a large scale not only for services provisioning, but also for the scalable management and control of resources. Authors advise to describe entities with high level of abstraction in order to achieve a clear separation between system components, and leverage a *divide-et-impera* strategy. Stratification doesn't reduce the problem complexity, but helps in rationalizing and dividing the problem in simpler parts whose solution is delegated to simple architectural components.

In InterDataNet the stratification is applied to leave freedom of implementation for each layer of the stack. The top layer receives information as a complex interrelated object, while approaching the bottom of the stack it is seen as several individual information units.

REST has been adopted also for internal communications exchange, this mean that every layer exposes resources and a uniform interface to manage them. Layers interact adopting the client-server paradigm: every layer is client of the lower layer and server of the upper one. Since HTTP has been used for the implementation of REST in InterDataNet, is possible to argue that the interaction follows a request/response approach. Therefore, it is a pull-type paradigm: as a consequence of a client request, the stack is crossed by a series of requests that proceed from the top to the bottom, and a series of responses that proceed in the opposite direction.

From a general point of view, the following statement is valid: every layer manages information by breaking it down in elementary data units, with the appropriate granularity level, and associating it layer-specific metadata. Therefore, every layer has a peculiar resource representing data in the processing phase. The Virtual Resource layer manages IDN-Documents and, internally, VR-Nodes. The Information History layer manages IH-Nodes and Storage Interface manages SI-

Nodes. in compliance with REST principles, all these resources can be individually addressable and therefore have a layer-specific identifier.

5.1.1 Encapsulation

In compliance with the Information Hiding principle, the information flowing through the stack undergoes an encapsulation process. This means that each layer treats the information coming from adjacent layers as a meaningless blob, focusing on the information necessary for providing its services. Fig. 5.2 shows the encapsulation process through the architecture stack.

In this case, data are produced by an IDN-Compliant Application, which issues a `write` operation to the Virtual Resource layer. Allegedly, the submitted resource is an IDN-Document carrying more than one node. The Virtual Resource layer breaks up the document in VR-Nodes and, for each of these, issues a call to the lower layer.

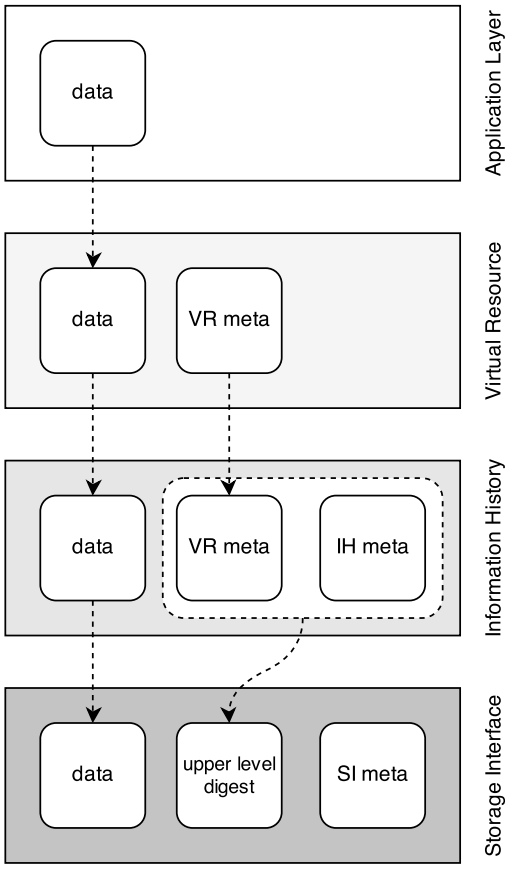


Figure 5.2. *The encapsulation flow through the InterDataNet stack.*

An IDN-Document can be seen as an handful of application data, decorated with InterDataNet metadata including definition for properties and links towards other data. Since these metadata realize the graph of information and are Virtual Resource specific, they are called "VR meta".

At this point, the control is passed to the Information History layer in the form of an IH-Node, i.e. the resource defined for this layer. The information coming from the upper layer is in the form of a digest and won't be decoded here. To allow the Information History to provide versioning and historicization services, other metadata ("IH meta") are associated with the information, and the procedure repeats with the Storage Interface layer. A SI-Node, i.e.the resource defined for the Storage Interface layer, is similarly created, and, again, the Storage Interface layer takes advantage of its own metadata ("SI meta") to enforce persistence capabilities.

Therefore, when the calls flow proceeds from the top to the bottom, the information is encapsulated. Conversely, when the information flows from the bottom to the top (e.g., to fulfill a retrieval request), the lower level envelope is opened and the information pertaining the current level is extracted and used.

It is worth noting that "data" flow freely throughout the architecture without undergoing any encapsulation procedure. This may seems to break the Information Hiding principle. However, data are not part of the concerns of the InterDataNet architecture. InterDataNet provides services for data that never possesses and elaborates. Data comes form IDN-Compliant Applications or remote sources and what InterDataNet really provides is a decoration service for these data, putting metadata on their side which are meaningful for the service provision. Indeed, once data are represented with the Information Model formalism, several properties can be enabled for them.

According to this vision is licit that data pass untouched through the stack.

Not to complicate the description of the encapsulation procedure, the architecture depicted in Fig.5.2 does not include the adaptation layer with Adapter modules. The introduction of the Adapter concept in the architecture, influenced significantly the information flow through the stack. This is one of the contributions to the previous works [Inn08, Chi09, Cio10] made in the context of this thesis. Previously, it was not planned that the information (data, not metadata which are defined on the architecture side only) could have come from external data sources: documents were simply created in the system by publishers, who delivered their own data to the architecture. In other words, it wasn't possible to connect to outer sources and transform their data in documents, without replicating them within the architecture. This led to an encapsulation design different than the current one, depicted in Fig. 5.2, where data and metadata were packed together to form a "digest" for the next layer. Since data were created on the upper interface and packed with metadata for the lower layer, couldn't be possible to connect a data source to the bottom layer without breaking the packing sequence. Fig. 5.3 shows

the Storage Interface resource with the previous encapsulation strategy. It is clear that adding external data at the Storage Interface level would have meant to open two nested envelopes.

The previous scenario was too limiting for at least two reasons: first, data replication introduces a number of problems concerning synchronization an consistency. In addition, it gainsays the philosophy behind InterDataNet, which strongly promotes information reuse.

Second, which is the major limit, it is not compatible with real-world scenarios, where is very unlikely that data providers are pleased to consign their data to a third party.

The chosen strategy is to see the IDN-Document form as a decoration for data, and provide this decoration as a service, instead. Data remain in the hands of the owner, and the system must be able to build an IDN-Document all around them. When such data are managed through the mediation of InterDataNet, they benefit from all the architecture capabilities, conversely they don't. It is just a quality of service matter.

With the introduction of this concept, had to be considered the possibility that data could come from the bottom of the stack, without being created from the top first. Dealing with this aspect still being compliant with the previous encapsulation model would have meant to perform an heavy unpackaging of the envelope at the Storage Interface level because data coming from the outer source should have put inside the envelope generated by Virtual Resource. This definitely would have been a break of the principle (see Fig 5.3).

The awareness that data did not belong to InterDataNet paved the way for a transverse solution: they should have been taken out of the encapsulation and should have crossed the stack aside of the envelope.

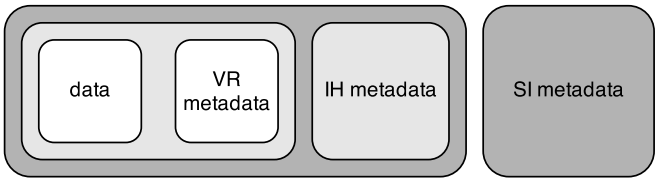


Figure 5.3. *The previous encapsulation strategy. To add data from external sources at Storage Interface level would have meant to open two nested envelopes.*

5.2 Virtual Resource

Virtual Resource (VR) is the core component of the InterDataNet architecture. It implements the graph of information exposed as document resources, and acts as crucial orchestration point towards other architectural services.

Virtual Resource connects to four different services and dispatches the architecture calls to proper components. Fig. 5.4 shows the interaction of Virtual Resource with different components and actors of the system.

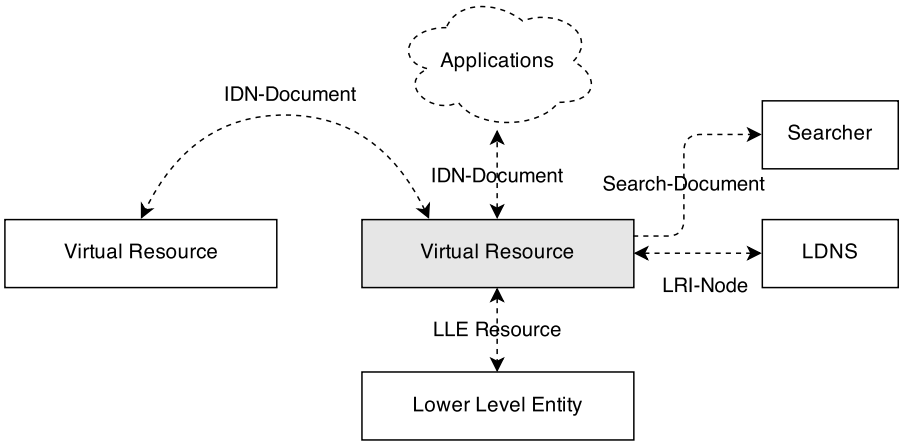


Figure 5.4. *The relations of Virtual Resource with the other actors of the system.*

On the top of the picture are located IDN-Compliant Applications, consuming, producing and modifying IDN-Documents. On the left side of Virtual Resource, a second instance of the same component appears. As mentioned in the previous sections, different Virtual Resource applications interact as peers to realize the distributed graph of information. In the right side of the picture, Searcher and LDNS services are shown. When an IDN-Document is created, Virtual Resource notifies the Searcher to update its indexes and make the new resource searchable. LDNS keeps the mapping between names of VR-Nodes and names of the lower level nodes (as regards the current implementation, SI-Nodes), so that when an operation is issued for a document, it can be propagated through the architecture by involving corresponding resources defined at each layer.

In the following, the Virtual Resource implementation will be detailed. The functionalities of the application will be described together with functional components and classes, in order to get an harmonic picture of the layer.

Virtual Resource is a Java 7 Web Application running on Apache Tomcat Application Server, implemented with the Spring [JHAT09] framework and leveraging different technologies such as Hibernate [BK05], JAXB [OM03] and JSP[Ber03],

to mention few. For the presentation, the Bootstrap framework have been used. Fig. 5.5 shows the Virtual Resource welcome page.

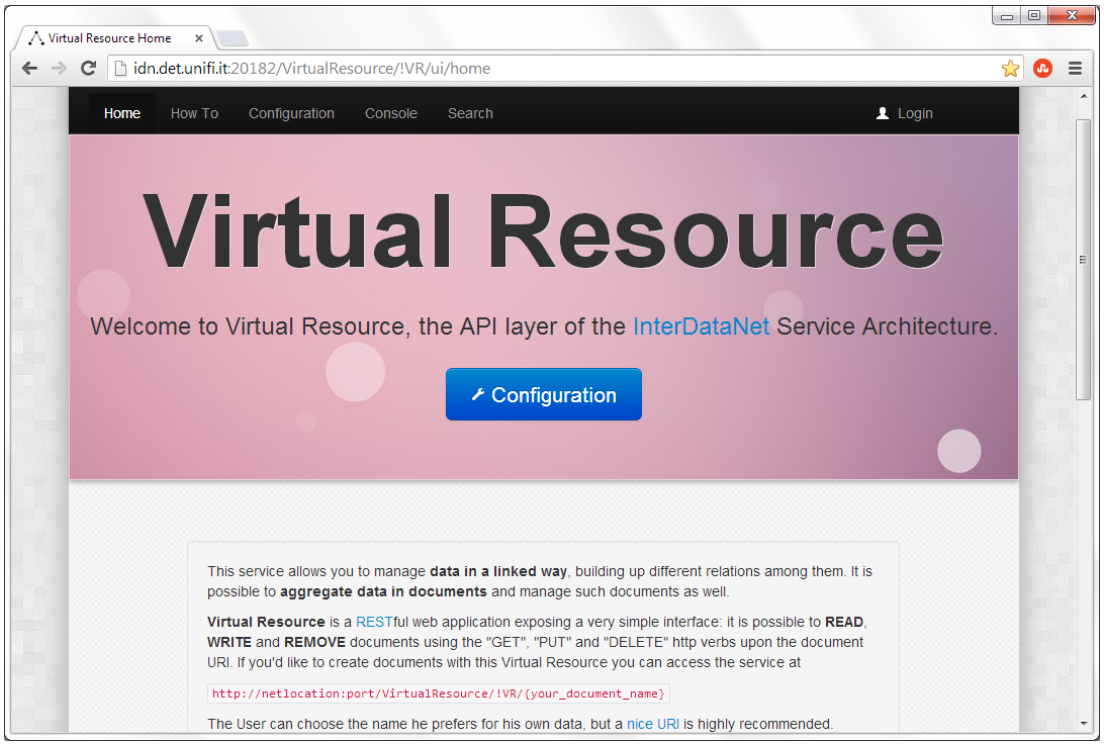


Figure 5.5. *The Virtual Resource welcome page.*

The development of Virtual Resource has been led by leveraging several design patterns such as Dependency Injection, Singleton, Model View Controller [Wol94, Mar03, Fre04] and many others. Because of the distributed, Web oriented nature of the application, it is worth spending few words on the latter.

The Model View Controller [LR01], or MVC for short, is a design pattern enabling a strong separation of concerns between the presentation logic, the business logic and data.

The most common scenario is a client interacting with a Web server. Often, what the client wants is to see some data owned by the server and interact with it. If the server implements the MVC pattern (see Fig 5.6), will treat these data as the model, the way data will be arranged to be served as the view, and the component reacting to client commands as the controller. More precisely, a Web application will expose some URLs to manage resources. The controller component is listening on the invocation of these URLs and will retrieve the model from a database or from a remote source. According to client's preferences, it will serve the requested data in HTML, XML, Json or whatever format, when rendering the view.

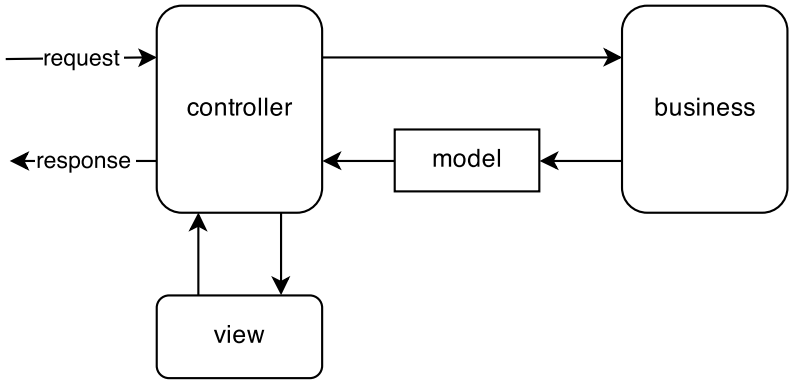


Figure 5.6. *The Model View Controller design pattern.*

Fig. 5.9 shows main classes performing core operations for the layer. Not to complicate the reading, only the essential is shown in the diagram. The Virtual Resource application has several different services doing transverse tasks (some of them have been implemented using the Aspect Oriented programming style [FEC+04]), such as logging, validation, scheduling, polling, services consuming data to and producing data for the InterDataNet Search Engine network, interceptors for performance assessing, persistence management and interface to the database, marshalling and unmarshalling, presentation logic (a GUI is implemented), and many others. Currently, the project consists of approximately 120 Java classes, excluding libraries. Obviously, it is not possible to conduct a full discussion of the application which will be limited to aspects specifically connected with the core functionality of the layer.

The entry point of the application flow is the `DocumentController` class, which, in fact, implements the MVC controller for the document resource. The available operations for the document resource are the ones described in section 4.2.3, namely `read`, `write` and `delete`. The HTTP OPTION verb is also supported.

The `DocumentController` is responsible for managing the interface towards InterDataNet clients, including rendering proper views (HTML, XML and Json) for a request, deciding response Status Codes and dispatching controls to the `DocumentManager` class.

The `DocumentManager` class manages document, i.e. sets of nodes. At this level, documents have a precise identity and structure which is made up of a number of inter-related nodes. Mirroring methods defined for the controller, the `DocumentManager` exposes `getDocumentNode` and `getDocument` methods to fulfill a request for a document made up of a single or more nodes. In the second case, a very important (private) method is fired: `resolveNode`. As the name suggests, it is used to retrieve all nodes constituting the document. This method

is critical because performs a recursion and therefore may significantly affect the time complexity of the operation. The method performs a constrained depth-first visit of the graph along Aggregation Links to discovery next nodes. When a node is discovered, is retrieved and stored in a stack so that, if the same node is referred twice, a second resolution won't be attempted (but, it could be necessary for its children). The visit is constrained by the resolution depth parameter discussed in section 4.2.2.

Fig. 5.7 shows the algorithm on a document instance. Labels on edges define the exploration sequence, while letters in the stack (filled from the bottom to the top) represent the nodes retrieval order.

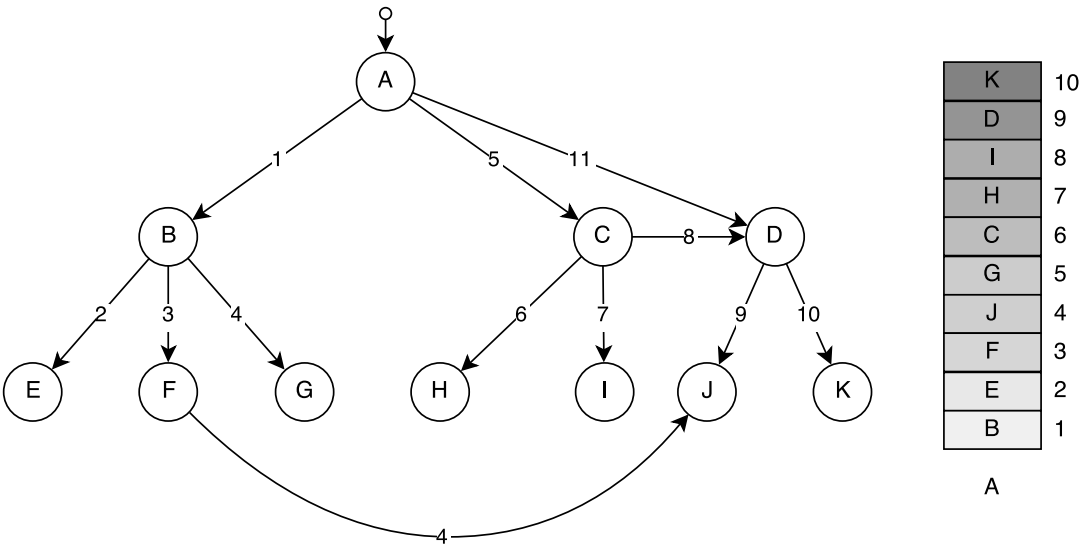


Figure 5.7. *The resolveNodes method run on a document instance*

Actually, the `resolveNode` method is a little more complicated. In section 4.1.3 the Activity Node has been introduced, and this is the point where is managed first. Indeed, the objective of the method is to deliver a full consistent document to the caller, and this means that if an Activity Node is encountered during the resolution procedure, it will have to be handled on the spot.

As previously discussed, Activity Nodes have dependencies that must be resolved before computing the script. To fulfill this requirement, a link precedence is defined for resolution. The rationale behind the extended resolution algorithm is very simple: while attempting a resolution following Aggregation Links, if an Activity Node is found, explore its Active Links *first*, and *then* its Aggregation Links, before returning the control.

This procedure allows for a special case. Consider an Activity Node `A` with an Active Link pointing to a second Activity Node `B`. In this case, `B` must resolve all its dependencies before being returned as an input for `A`. However, if an Aggregation

Link departs from `B` to a third node `C`, it shouldn't be considered for resolution since `C` is not part of the original document. The flow chart of the `resolveNode` method is shown in Fig. 5.8. Please note that, as the figure points out, the diagram doesn't specify whether a retrieval is attempted from the architecture or from a remote source.

To manage the document resource, the `DocumentManager` will end up requiring services from a `NodeManager`. In fact, the approach chosen for managing complex entities such as the document, requires they are broken out in elementary units, and handled separately. The Virtual Resource application leverages two different `NodeManager`s: the `InnerNodeManager` for nodes belonging to the current Virtual Resource instance, and `ForeignNodeManager` for nodes belonging to other instances. Both two extend the abstract generic class `AbstractNodeManager` whose method will be detailed when discussing the `read`, `write` and `delete` flows. For now, will be enough to point out that `NodeManager` classes mirror the operation defined at higher level, specializing for the `Node` entity.

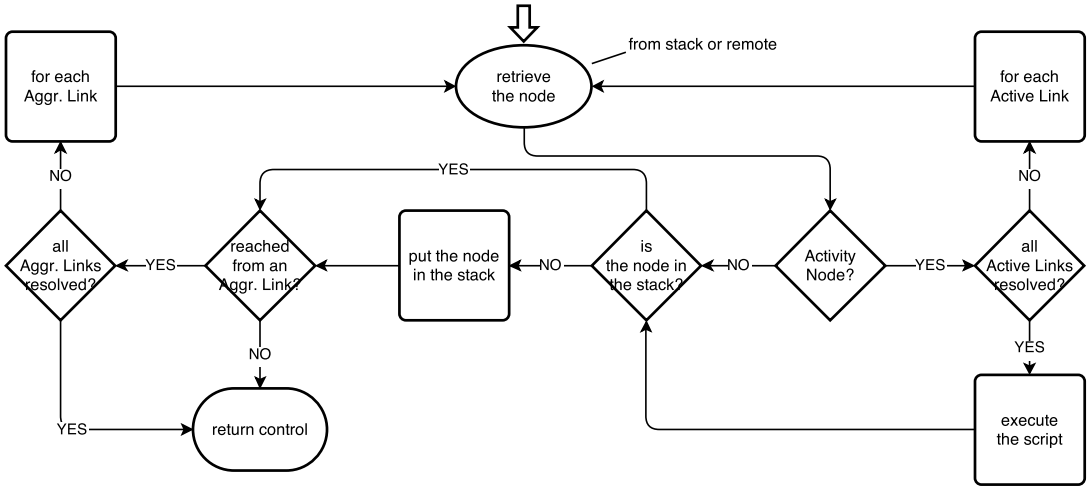


Figure 5.8. *The flow chart of the `resolveNode` method.*

The `InnerNodeManager` class depends on `NameManager` and `SearchManager` services. The first provides a mapping of the current LRI with the lower level name (for the reference implementation, it is a URL associated with a SI-Node), the second contacts the InterDataNet search engine network to request an indexes update.

In the following subsections details concerning the `read`, `write` and `delete` operation flows will be given.

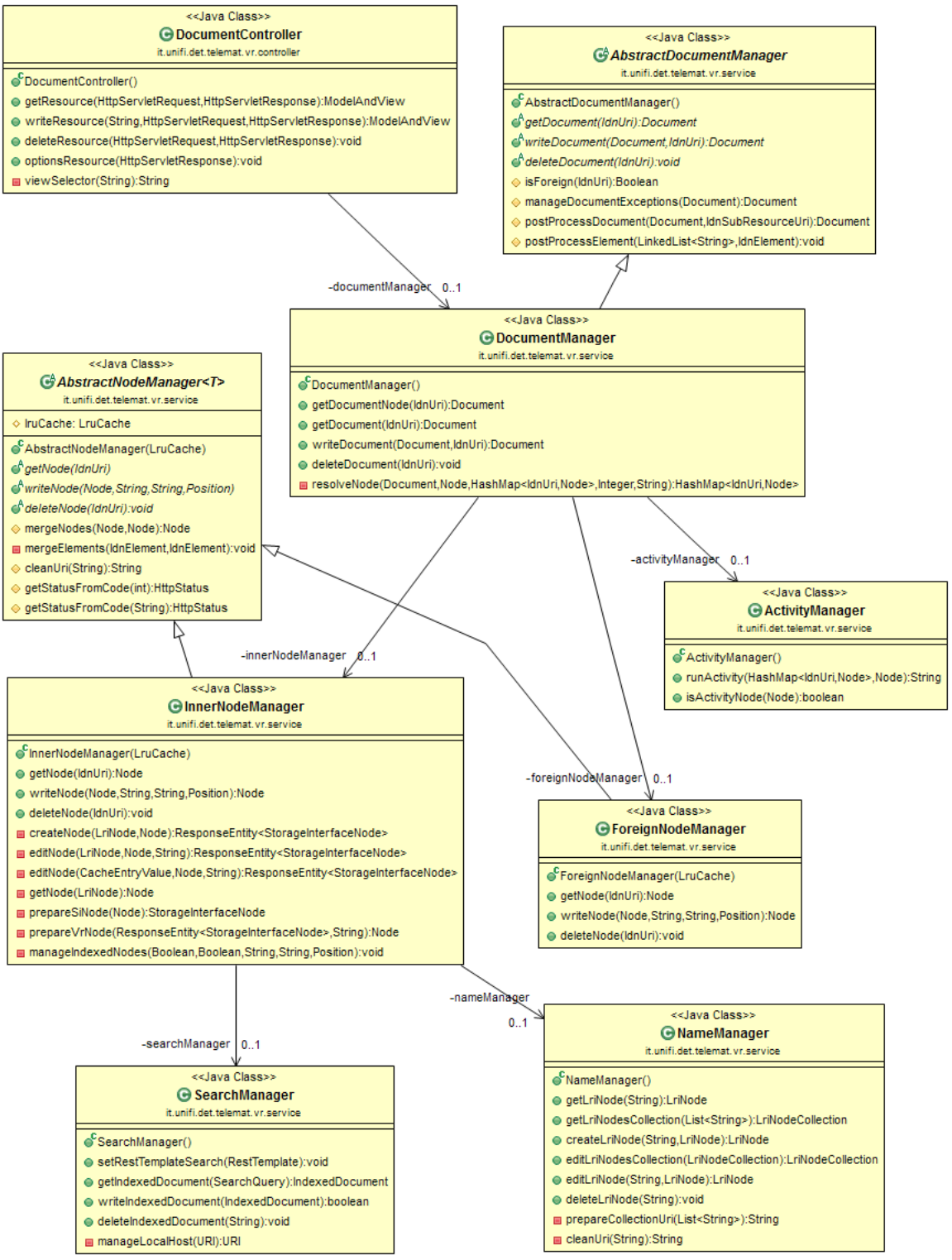


Figure 5.9. *The main classes of the Virtual Resource implementation.*

5.2.1 Reading a Document

The `read` flow begins when a client invokes an IDN-URI (see section 4.2.2) associated with a document. Virtual Resource responds to URIs matching the following ant-like [HLRV03] pattern:

```
http://idn_host/VirtualResource/!VR/**
```

meaning that after the `/!VR/` part, everything (with the exception of the IDN-URI parameters) is interpreted as a document name. The HTTP request sent by the client is intercepted by the `DocumentController`'s `getResource` method that checks whether is the case of a single node or a multiple nodes document `read`. In the first case, the request is dispatched to the `DocumentManager`'s `getDocumentNode`, otherwise it is dispatched to the `getDocument` method. If a single node document is requested, the flow is relatively simple because no other Virtual Resource application is involved. Indeed, if the root node exists, it must be managed by current Virtual Resource instance because the URL that is being dereferenced led to this location.

Thus, the `DocumentManager` invokes the `NodeManager` requesting the resolution of the root node, which, in turn, invokes the `NameManager` service to contact the `LDNS` module. The `NameManager` class takes advantage of the `RestTemplate` client provided by the Spring framework to issue an HTTP GET to the `LDNS` Web Application. The meaning of this operation is querying the naming service for the SI-Node URL corresponding to the root node requested by the client.

In the current implementation, LDNS is paired with a Virtual Resource (because of the loose coupling principle enforcement, more Virtual Resource modules are allowed to refer the same LDNS. In this case, LDNS keeps the name spaces completely disjoint) and it should always be able to resolve a requested URL.

If the response form the LDNS module is successful, the `NodeManager` has now the URL to contact the lower level entity. This is performed by issuing an HTTP GET request to the SI-Node resource exposed by the Storage Interface application. If the operation is successful the SI-Node wrapping the VR-Node of interest will be returned. At this point, the unpackaging operation is immediately performed and the unwrapped VR-Node is first validated and then returned to the `DocumentManager` which performs some document level validation, prepares the document and returns it to the controller.

At this point, the `DocumentController` has a Model object, i.e. the document, to return. Depending on the outcome of underlying operations, the HTTP Status Message is set. After that, the `Accept` request header is checked and a suitable view is used. Now, the HTTP response with the model object and a properly rendered view is ready to be sent to the client.

Fig. 5.10 shows the sequence diagram for a `read` operation issued for a single node document.

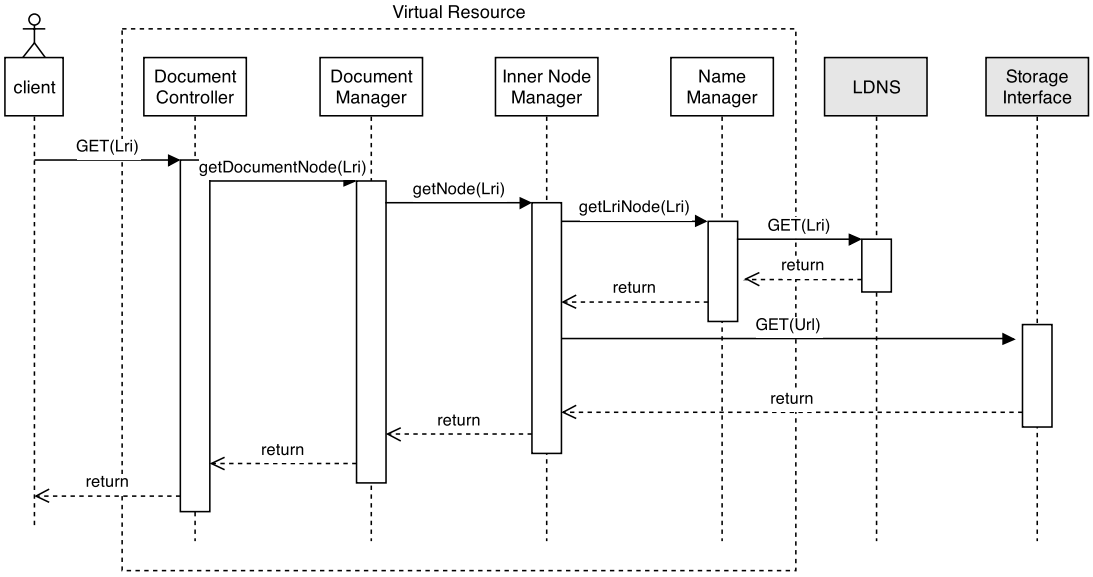


Figure 5.10. *The sequence diagram for a `read` operation issued for a single node document.*

When a request for a document made up of multiple nodes is requested, the `DocumentManager`'s `getDocument` is fired and the flow gets more complicated, basically because of the resolution algorithm described in section 5.2. A more detailed description of the recursive resolution algorithm calls for a code analysis, which is not appropriate for this context. The whole InterDataNet source code is intended to be released open source and the interested reader can delve directly into the code.

However, is worth to point out that the process begins by requesting the `InnerNodeManager` to retrieve the root node, since, as argued above, it *must* belong to the current Virtual Resource instance. This operation proceeds analogously as described for the single node document retrieval. Then, the recursive resolution algorithm invokes the `getNode` method from either `InnerNodeManager` or the `ForeignNodeManager`, according to the authoritative peer specified in the nodes' LRIs. In fact, it is licit for nodes in a document to span across different Virtual Resource instances, and is normal that a resolution operation requires to contact other peers. When this happens, the `ForeignNodeManager` issues an HTTP GET to a peer, requesting for a document. Fig. 5.11 shows the sequence diagram for a `read` operation issued for a document made up of multiple nodes (some of them are retrieved from a peer Virtual Resource).

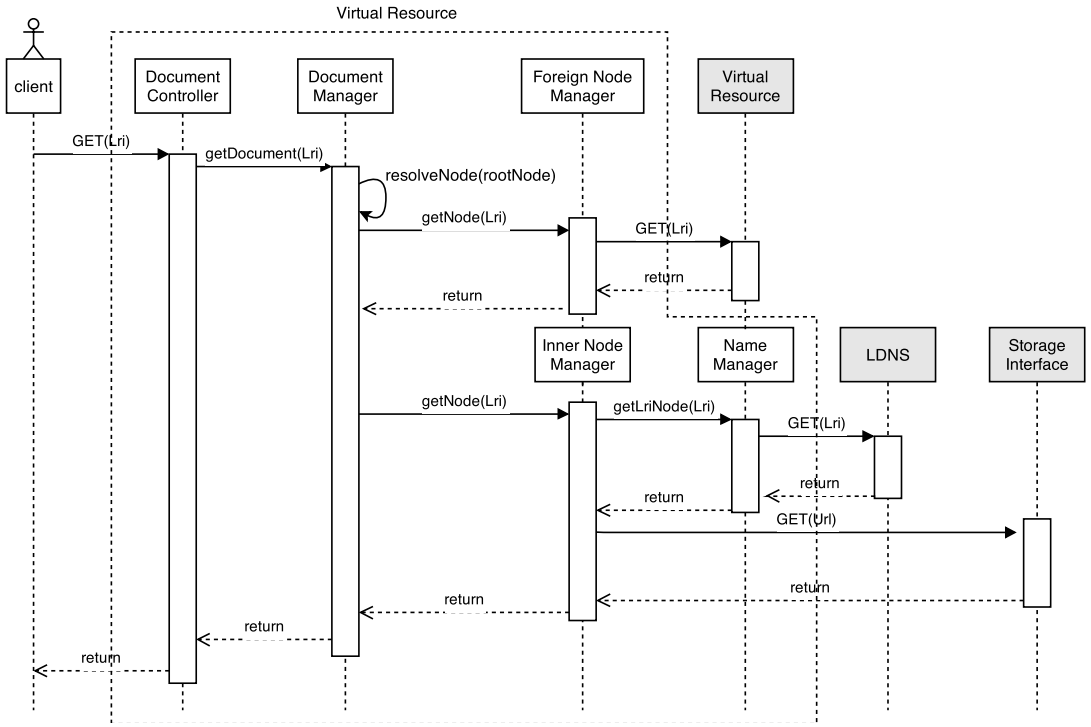


Figure 5.11. *The sequence diagram for a `read` operation issued for a multiple nodes document.*

The fact that a *node* manager requests a *document* resource is not a break of the Separation of Concern principle. Just remember that every Virtual Resource peer exposes a single resource, i.e. the document. When a `ForeignNodeManager` issues a request for a document, it is always a single node one and, at the time is available, such node is immediately extracted and managed as is.

5.2.2 Writing a Document

The `write` operation involves the same classes described for the `read` operation. Briefly, when an HTTP PUT request is issued to the application endpoint, the `DocumentController` reacts and the `writeResource` method is fired. By definition, such request should have a non-empty body containing the resource representing the state to transfer. Unlike the `read` case, the body must be validated and this may result in a `400 bad request` response Status Code. After having checked the (mandatory) `Content-Type` HTTP request Header (the application accepts XML and Json formats) and selected the proper unmarshaller to translate the document representation in a domain object, the `DocumentManager` is called.

As expected, the `writeDocument` method is invoked, which performs a second

document specific validation, and breaks up the document in nodes to check whether they belong to the current architecture or not.

In the first case, they are iteratively dispatched to the `InnerNodeManager` by invoking the `writeNode` method which, again, performs a node specific validation checking the syntax of URIs contained in the `LRI` field and the `value` fields of Links. The `vr-id` section-scoped uniqueness constraint is checked as well.

When a node turns out to belong to a different VirtualResource peer, is be dispatched to the `ForeignNodeManager` that prepares a single node IDN-Document, and issues an HTTP PUT to the URL contained in the LRI field of such node. The peer will handle the request as a `write` operation, as described in this section.

Returning to the `InnerNodeManager`, the system verifies the creation or update nature of the current `write` operation. To accomplish this task, a refined strategy based on the Virtual Resource cache (see section 5.2.4) is adopted, for now the reader can assume that an HTTP GET request is issued to LDNS, asking whether the submitted LRI is present in the index or not. If current operation is found to be a creation, the private `createNode` method is invoked. The `NameManager`'s `createLriNode` method is called and an HTTP PUT request containing the new LRI-Node resource is sent to the LDNS application. In the coming response, is available the SI-Node URL (computed by LDNS) that will be coupled with the newly created LRI. At this point, the `createLriNode` assembles a SI-Node with the current VR-Node, includes it in the request Body and issues a PUT to the URL provided by LDNS.

If the operation is successful, a descriptor is created from metadata contained in `keywords` and `Location` sections (see section 4.2.1) along with the LRI, and it is added to a stack which will be managed by a scheduler to enter the newly created node in indexes of InterDataNet Search Engine network.

The lower level entity is a Storage Interface known by LDNS (that, for this reason, is an architectural trust key point) which takes over the SI-Node, and returns an `Etag` code, representing the current state of the resource. The `Etag` is included in the `NodeIndo` section of the current node, and the control is returned to the `DocumentManager`.

If the `write` operation is found out to be an update, the flow becomes more articulated. As requirement, the updating nodes *must* have a valid `Etag`; if this condition is not met, a `412 precondition failed` HTTP Status Code will be output for that node. It is recalled that the update operation defined for the InterDataNet interface follows the "update what you declare" strategy discussed in section 4.2.3. This means that the client is free to include in the request Body the resource containing *only the elements it's willing to modify.*

From an implementation point of view, this implies a significant effort because the state of the submitted resource must be *merged* in a union-like operation with

the current one.

First, the `editNode` method invokes the `getNode` method to retrieve the current state of the resource from the system, then verifies that the `Etag` for the submitted node equals the one from the retrieved resource (to avoid merging resources in incompatible states) and calls the `mergeNodes` method defined for the abstract class `AbstractNodeManager`, passing resources to be merged as an argument.

The `mergeNodes` method is very complex and required the development of a dedicated library for the abstract management of nodes. This constituted the foundation of the IDN Java Library detailed in section 6.1. Basically, a node is seen as a tree structure of elements containing other elements, up to the leaves. Every "element", "attribute" or "section" introduced in section 4.2.1 is an element of the tree, or an `IDNElement`. Fig. 4.14 serves as a visual sketch of the tree representation. For each `IDNElement` a vast set of "JQuery-style" [MDV04, DV06, BK08] features is enabled, for traversing the tree and for interacting with other elements. The library will be discussed in chapter 6, but for the sake of clarity some examples are given. For instance, given an `IDNElement`, operations such as `addChild`, `setSiblings`, `removeAllChildrenBut` and `removeAllSiblingsBut` are supported, to mention few.

Within the method is performed a union-like operation between `IDNElement`s belonging to two different nodes: if the `IDNElement` belonging to the submitted node has a child which is also present in the corresponding `IDNElement` belonging to the current node, the current `IDNElement` will overwrite its child. Otherwise, it is added to the current `IDNElement`. The procedure is performed recursively and the overwriting is performed only on leaves `IDNElement`s.

Fig. 5.12 exemplifies the merging procedure. The tree structure on the left represents the current state of a node while the tree structure in the middle represents the submitted node state. Not to complicate the example, it was decided to omit the actual node representation. A fake representation with a more readable hierarchical labeling has been used instead. Just remember that given a parent and a child, the first contains the second and that some elements (e.g., the `A.1.1` or `B.2.2`) can have a value.

The tree structure on the right is the outcome of merging the current state with the submitted state. Let's give a closer look at the latter. There are two notable things, here: first, the value of the `A.1.1` element is *changed*. To specify this modification, the whole parent-child hierarchy up to the leaf, is required to get a proper qualification of the element (the `X.y.z...` notation adopted in this example binds the location of the element with its identification, but this is not the case in the actual VR-Node object. In other words, the `AggregationLink` label says nothing about the position of the element in the tree structure).

Second, the tree structure includes an *unchanged* element whose intent is *not* to qualify any *changed* element. This is not interpreted as a redundancy but as a declaration of state transfer. Practically, the client is requesting to substitute

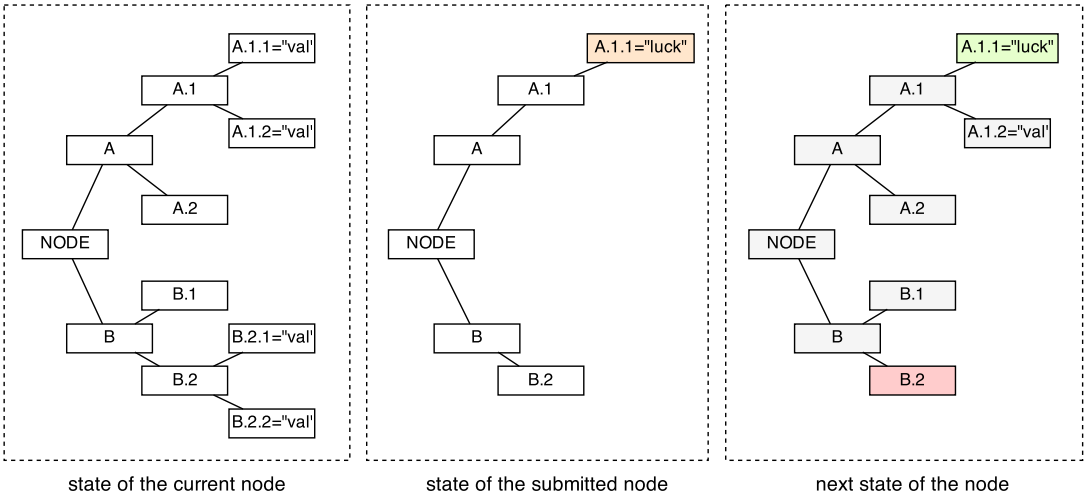


Figure 5.12. *An example of the `write` operation with update semantics.*

the `B.2` element of the current state with the `B.2` element of the submitted state. In fact, this results in a deletion of children `B.2.1` and `B.2.2`, as a side effect of a modification of an higher container. The outcome of the merge operation is depicted on the right side of the picture. As discussed, elements not mentioned in the submitted state are unaffected, while elements explicitly declared are modified. When the merging operation is completed, the resulting node must be committed to the Storage Interface application. To this end, LDNS is contacted to obtain the URL of the SI-Node paired with the VR-Node, and finally an HTTP PUT containing the SI-Node with the novel state is issued to Storage Interface. The `Etag` is updated and communicated as explained above.

After that, the control returns to the `DocumentManager` which prepares the document resource and eventually fills the `Warning` and `Errors` fields, if some problems occurred. The document is then returned to the controller which selects the appropriate view and responds to the client, concluding the communication.

Please note that a `write` operation performed for a document, may include all the cases explained above. Some nodes can be created or updated in the current Virtual Resource instance, and some can be created or updated on one or more peers. The `DocumentManager` dispatches nodes to the `InnerNodeManager` or `ForeignNodeManager` iteratively, until nodes contained in submitted documents are all properly handled. This implies that operations are performed on a single node at a time, and may result in a loss of performance. In fact, in more than one scenario discussed above, the system would benefit from a parallel processing of domain resources.

This consideration motivated the study of the Performance Enhancing System, or PES for short, discussed in section 5.2.5. Unfortunately, the PES system is

not part of the current implementation, because its inclusion would have required a significant code modification for Virtual Resource, Logical Domain Naming System, Storage Interface and Adapter(s) applications. However, future releases of the architecture will include the PES support for parallel processing.

Fig. 5.13 shows the sequence diagram for a `write` operation issued for a document and resulting in a second `write` operation (whether update or creation) for a peer, and an update for the current instance of Virtual Resource. Please note that, from a Virtual Resource point of view, it is not important to know the nature of a `write` operation issued to a peer application (i.e. whether it is a creation or an update). It is not entitled to know the state of resources it does not belong, and it behaves simply as a client, delegating the management of foreign resources to the competent Virtual Resource. In the sequence diagram is shown the GET request issued to LDNS to retrieve the URL for the corresponding SI-Node, fired by the `NameManager`. In case of a creation, Virtual Resource would have issued an HTTP PUT, to create the LRI descriptor hosted by LDNS, for the novel resource.

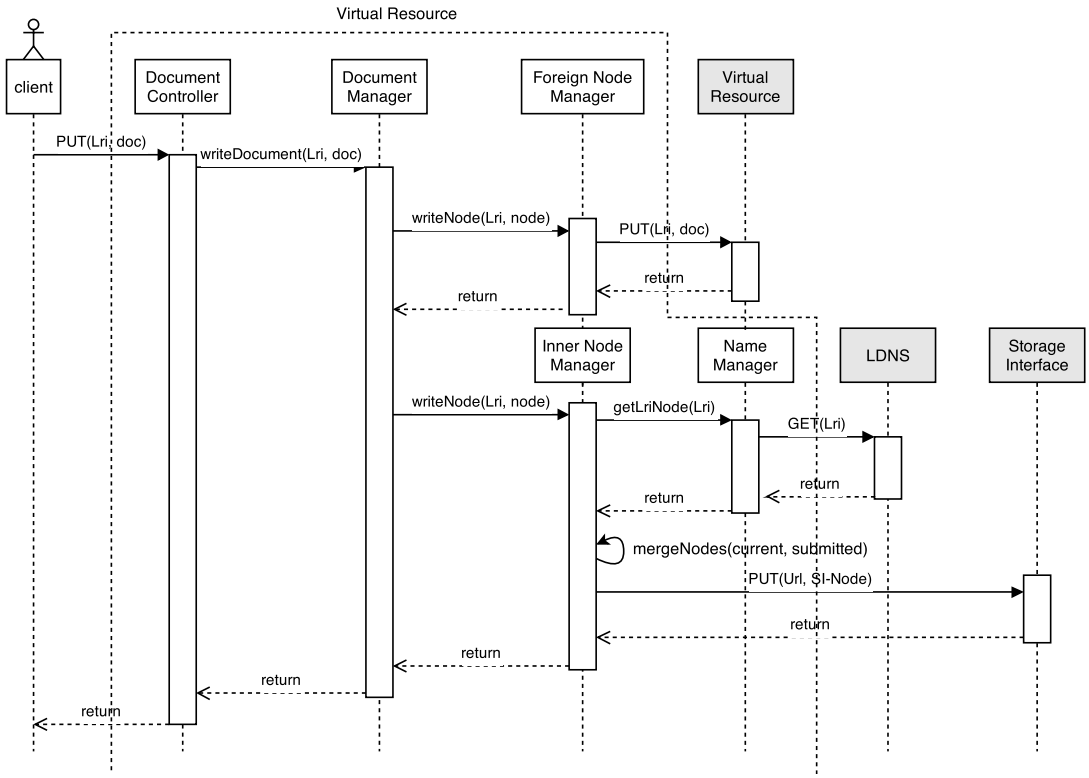


Figure 5.13. *The sequence diagram for a `write` operation for a document resulting on a `write` of a node belonging to a peer, and an update of a node belonging to the current Virtual Resource.*

5.2.3 Deleting a Document

The deletion scenario is the simplest one. the `DocumentController` listens for HTTP DELETE requests issued for IDN-URIs that do not have any selector parameter. Indeed, the deletion is allowed for single node documents, i.e. deletions can be performed on single information units, through the mediation of the document resource. The reason for this is that InterDataNet is a highly collaborative environment. When a document is deployed to the global space, virtually, many different actors can cooperate around it, by changing it, enriching it, and using it to build their own documents. Supporting a massive document deletion mechanism would imply the knowledge of the document state at all times. In fact, the client's knowledge of the document state at a given time, is not guaranteed to be significant in a next time. Many actors could have modified the document, by adding information pieces on which the client has no authority at all. In few words, because of the *ephemeral* nature of the document, a client demanding a complete document deletion really is not in the position to know what will be actually erased.

Please note that this doesn't happen for the `write` operation, where massive document modification are allowed. In that case, the discriminant is represented by the fact that the client, using the HTTP request Body, defines precisely what is to be modified. Massive operations are also allowed for the `read`, just because it doesn't determine any side effect.

Now, the `DocumentManager` dispatches the call to the `InnerNodeManager`. Since the deletion is defined for a single node, the Virtual Resource involved in the operation *must* be the one authoritative for the resource to be deleted. Therefore, the `ForeignNodeManager` won't be involved in the operation, in any case. In turn, the `InnerNodeManager` delegates the `NameManager` which issues an HTTP DELETE to the LRI descriptor exposed by LDNS. After that, the control is returned to the `InnerNodeManager` which inserts the name of the deleted node in a stack that will be managed by a scheduler to update the indexes of the InterDataNet Search Engines network, and this is all. Virtual Resource does not perform any deletion on Storage Interface, because there is no need for a resource to be deleted simultaneously with the client request. Once the descriptor is removed from LDNS, the resource will be unavailable anyway (i.e., a client request for that resource will result in a `404 not found`). LDNS can take its time to run a scheduler to manage the deletion of resources hosted by Storage Interface.

Fig. 5.14 shows the sequence diagram for a DELETE operation performed on a single node document.

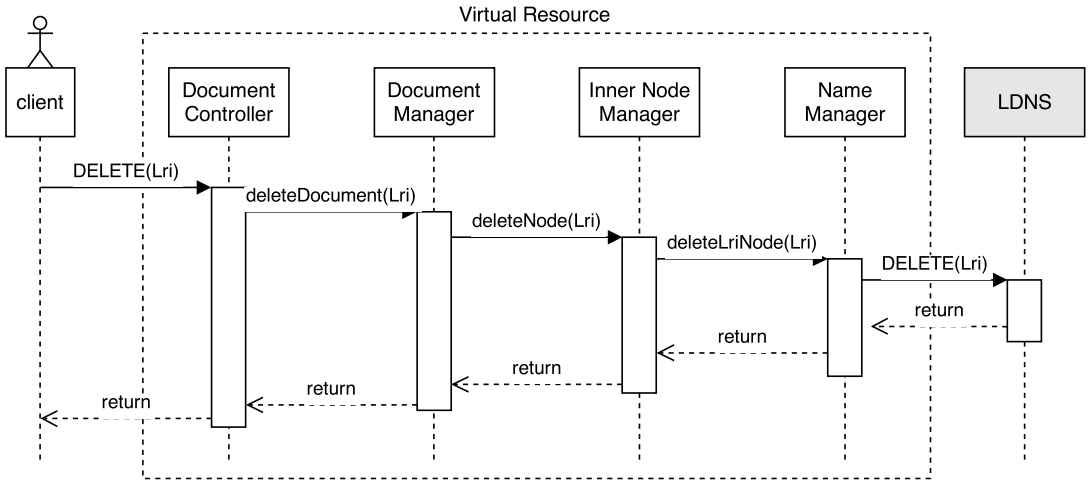


Figure 5.14. *The sequence diagram for a `delete` operation.*

5.2.4 Caching System

In order to save HTTP requests and improve performances, a Caching System has been designed and implemented for Virtual Resource.

It is a Least Recently Used (LRU) cache [RD90] for nodes, because is likely that certain nodes are requested more often than others. The LRU algorithm moves the element that is used more recently to the top of the stack and drops others. The cache takes space form the bottom of the stack, sacrificing the entry that hasn't been accessed for the longest time. In this way, the upper section of the cache tends to collect elements that are accessed more frequently.

Fig. 5.15 explains the LRU algorithm, showing a use case. At the top of the picture is shown the state of the cache at time t_0. At time t_1, the `C` element is hit, becoming the most recently used entry of the cache and is moved to the top position. Entries having a higher ranking than `C` are shifted to the bottom and finally, at time t_2, the cache is in its new state.

Technically, the cache is organized as a map, a `LinkedHashMap` exactly, to take advantage of its very efficient indexing capabilities ($O(1)$ for lookup and insertion operations). `LinkedHashMap` from JDK also provides a doubly-linked list running through all of its entries that defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). However, a special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order) [Mic11].

In this cache implementation, the LRI is the index and the entry record is an object (`CacheEntryValue`) having the node, the URL to the related SI-Node and

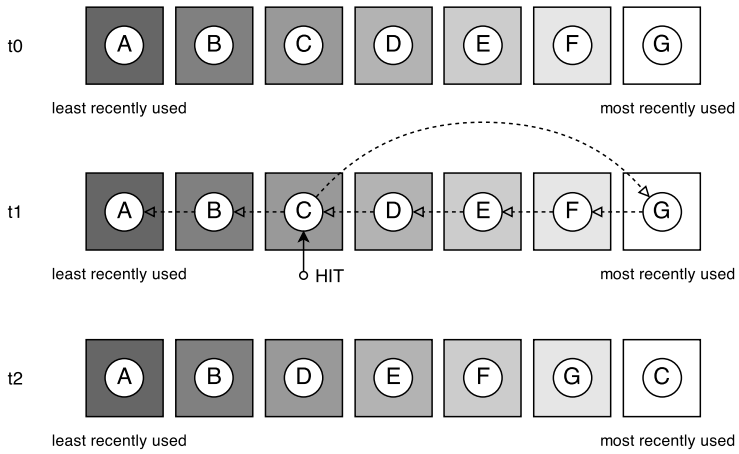


Figure 5.15. *The Least Recently Used caching algorithm.*

the `Etag` value as attributes.

As a matter of fact, this cache is not helpful for `read` operations. When a document is requested, Virtual Resource can not know the actual state of the node that is currently being processed. To be more precise, this statement holds for nodes having dynamic content, i.e. nodes whose data are retrieved from a remote source, on the fly. As regards for static content nodes, i.e. those nodes whose content changes only as a consequences of an explicit `write` call on the architecture, the cache could help. Therefore, as recommended by REST, a `cacheable` attribute could be used to specify which nodes are eligible to undergo a caching procedure. However, this feature is not currently implemented, and will be a matter of future works.

Every time a node is delivered to Virtual Resource, e.g. during a `read` operation, and every time Virtual Resource produces a new state, e.g. during a creation, the cache is updated. When a `write` operation is issued, is performed a lookup in cache to retrieve the node to be written. If the node is found, a quick check of the `Etag` of the submitted node against the `Etag` of the cached one is performed to make sure that nodes have compatible states, i.e. the cache entry is not out of date. If the check fails, the cache entry is removed, and the flow goes on as usual. Otherwise, the cached node is used during the merging operation and the merged node is submitted to Storage Interface. When the operation concludes successfully, the cache is updated with the new state of the node.

If a node is not found in cache, it is retrieved remotely, and when the operation concludes successfully, is added to the cache. The cache is also updated when a resource undergoes a successful creation, since its state can't be outdated.

Finally, when a `delete` is issued, the node is removed by the cache.

It is worth to remark that as long as static resources are accessible through a *unique name*, is possible to benefit greatly from the cache, because every modification happens by a single peer which is always aware of the latest states. The fact that aliases have not been addressed in depth and implemented, has put a brake on the development of the cache which is a purely functional technology.

My opinion is that an alias resolution should ultimately lead to the peer authoritative for that resource, which is a strategy that fully supports a clear distribution of responsibility, and therefore trust.

This approach calls also for a profitable use of the Caching System.

5.2.5 Performance Enhancement System

The Performance Enhancement System, or PES, is a strategy for optimizing invocations between architectural components. When a client interacts with Virtual Resource, the exchanged resource is the document. However, every architectural component requires to exchange unitary information, the so called X-Nodes. This leads to a significant number of calls to manage a single node of a document. To exemplify, consider the retrieval of a document containing four nodes, two from the Virtual Resource application hosting the root node and two from a peer. In this case, the `read` flow will proceed as discussed in section 5.2.1, i.e. a first request triggering the flow is issued by the client to Virtual Resource, then Virtual Resource issues a request to LDNS to retrieve the URL needed to issue a final request to Storage Interface. If the current node belongs to a peer, the LDNS step is skipped, as previously explained.

This set of calls (excluded the request from the client) is performed for every node of the document, i.e. four times, resulting in six calls.

This scenario would be significantly simplified if Virtual Resource would be able to dispatch to middleware components requests involving "architectural documents". Please note that the document concept as discussed so far, involves nodes which are *related* with each others (indeed, Links define these relations). This is *not* the case of architectural documents which are intended more to be "bags" of nodes. Moreover, should be considered the possibility to perform parallel calls, when they are independent.

Fig 5.16 shows an example of the PES strategy. On the left there is the document requested by the client. The nodes are colored with respect of the authoritative peer: the root node and the `B`, `D`, `E` and `F` nodes belong to the (current) `Y` Virtual Resource, the `A` node belongs to the `X` peer and the `C` node belongs to the `Z` peer. Resolving this document requires 10 sequential HTTP GET requests. On the right is shown the PES strategy. The syntax used for labeling edges is the following `[request number].[authoritative peer]`.

At the first level of the document, the `A`, `B`, `C` and `D` nodes can be retrieved by issuing three parallel requests: two independent requests towards the `X` and `Z` peers, and one request to the LDNS application in charge for `Y` to get a multiple names resolution. Now, the `Y` Virtual Resource can follow the same approach with Storage Interface: if `B` and `D` nodes are hosted by different Storage Interface applications, two parallel requests can be sent. Otherwise, a single request for a SI-Document containing both SI-Nodes can be sent. The same approach is adopted for `D` and `E`.

To sum up, adopting the PES strategy, the resolution of the document requires three parallel calls, and one additional subsequent call. If we assume that an HTTP request takes a unitary slot of time (this assumption can be optimistic for requests issued to peers), the time cost of the PES approach with the example document is 2, against the cost of the traditional approach which is 10 time slots, resulting in a 80% speed increase.

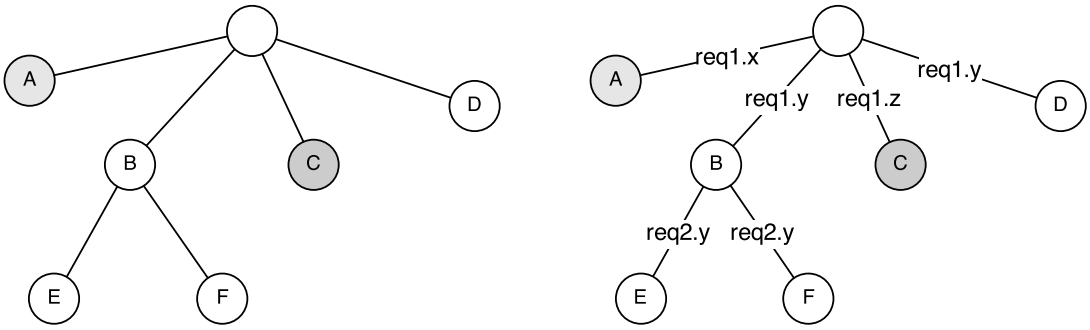


Figure 5.16. *An example of a document processed with the PES strategy.*

A second example inspired to a real use case is provided. To keep things simple, the parallelization is not taken into account here, so as the benefit comes from exchanging architectural documents only. Consider the case that InterDataNet is used in a big hospital to manage the patient hospital record (PHR) as an IDN-Document made up of 150 nodes per PHR instance. Every hospital ward has its own server, for a total of 15 machines. Assuming an homogeneous data distribution, nodes belonging to the PHR are distributed in groups of 10 on each server. In this case, the LDNS application will prepare a document with 10 sets of 15 nodes each, and Virtual Resource will perform 10 different calls.

Therefore, the cost in terms of requests is:

- 1 call from the client to Virtual Resource
- 1 call from Virtual Resource to LDNS
- 10 calls from Virtual Resource to lower level entities

for an overall amount of 11 HTTP requests (excluding the initial request from the client). Currently, with the traditional strategy, the cost in terms of requests is:

– 1 call from the client to Virtual Resource

– 150 call from Virtual Resource to LDNS

– 150 calls from Virtual Resource to lower level entities

for a total of 300 HTTP requests (excluding the initial request from the client). The net savings is about 95%, for this use case. However, consider that for simplicity has been evaluated an architecture in which Storage Interface is directly connected with Virtual Resource. To comply with the theoretical architecture it would be necessary to consider at least an intermediate level (the Information History). Given the combinatorial nature of the calls' tree, the complexity is believed to increase and limiting to the case study represents just a pessimistic approximation (and therefore good exemplification).

Fig 5.17 shows the calls distribution in the theoretical architecture (note the presence of the Information History layer) for the use case.

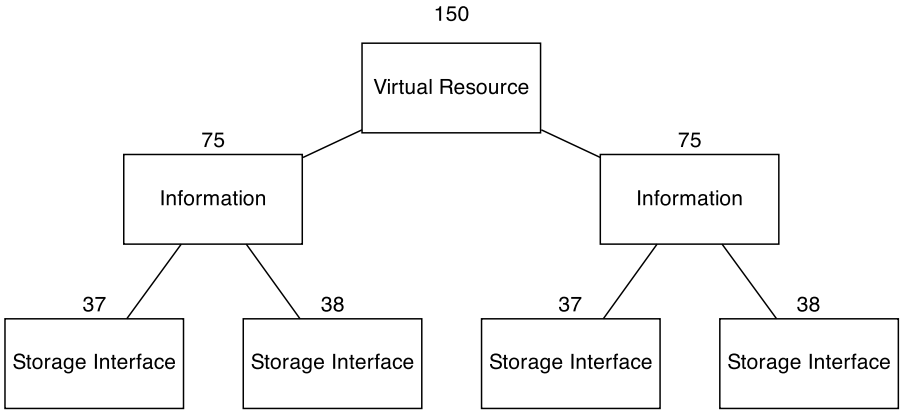


Figure 5.17. *The distribution of the calls through the theoretical architecture, in the traditional case.*

The number of requests (not considering the initial call and the ones to LDNS) is: $(75 + (2 * 37) + (2 * 38)) * 2 = 150 * 3 = 450$ that represents the worst case, and is not dependent on the morphology of the tree (assumed that levels remain the same). Conversely, using PES the number of calls is dependent on the morphology of the tree and in the shown case will be limited to the number of edges, that is 6. Please note that the case of a uniform data distribution represents the worst case because maximizes the number of calls. The ideal case, is the one in which all resources dwell on the same location at each step, in such case the number of required requests is 2. Finally, let

– w, be the number of calls in the worst case

– b, be the number of calls in the best case

– m, be the number of involved entities

– n, be the number of architectural layers

– k, be the number of nodes to manage

given a tree architecture, the general formula for the evaluation of the number of calls is the following.

	Traditional Strategy	**PES Strategy**
`worst case`	$w = ((n-1) * k)$	$w = m - 1$
`best case`	$b = ((n-1) * k)$	$b = n - 1$

5.3 Storage Interface

Storage Interface is the bottom layer of the InterDataNet architecture and is entitled to manage persistence aspects. It manages also the processes involving data retrieval from outer sources, such as an external repository or API, leveraging the mediation of Adapter components. Technologies described for the Virtual Resource application in section 5.2 have been applied also for the Storage Interface implementation. Fig. 5.18 shows the Storage Interface welcome page.

In the current architecture implementation, Storage Interface exposes SI-Nodes to Virtual Resources which use them to persist information units constituting IDN-Documents. The SI-Node resource is a tree structure containing data, a digest from the upper layer and metadata related with persistence aspects such as `Author`, `ModifiedOn`, `AutodestroyOn`, to mention few.

Fig. 5.19 shows the displacement of the Storage Interface application within the system. Basically, Storage Interface manages two different types of resources: already mentioned SI-Nodes, and so called Remote Resources. These are exposed by Adapters and wrap data coming from remote data sources (from which they borrow their name). A Storage Interface may have a complete interaction with an Adapter, in terms of `read` and `write` operations, that will be translated by the latter for external sources. Of course, this depends on the contract between the data provider and InterDataNet: whether certain operations are enabled or not for Remote Resources, depends on the the data provider's will.

From a logical point of view, the information *persisted* by the Storage Interface application can be roughly divided in `data` and `metadata`. The former are

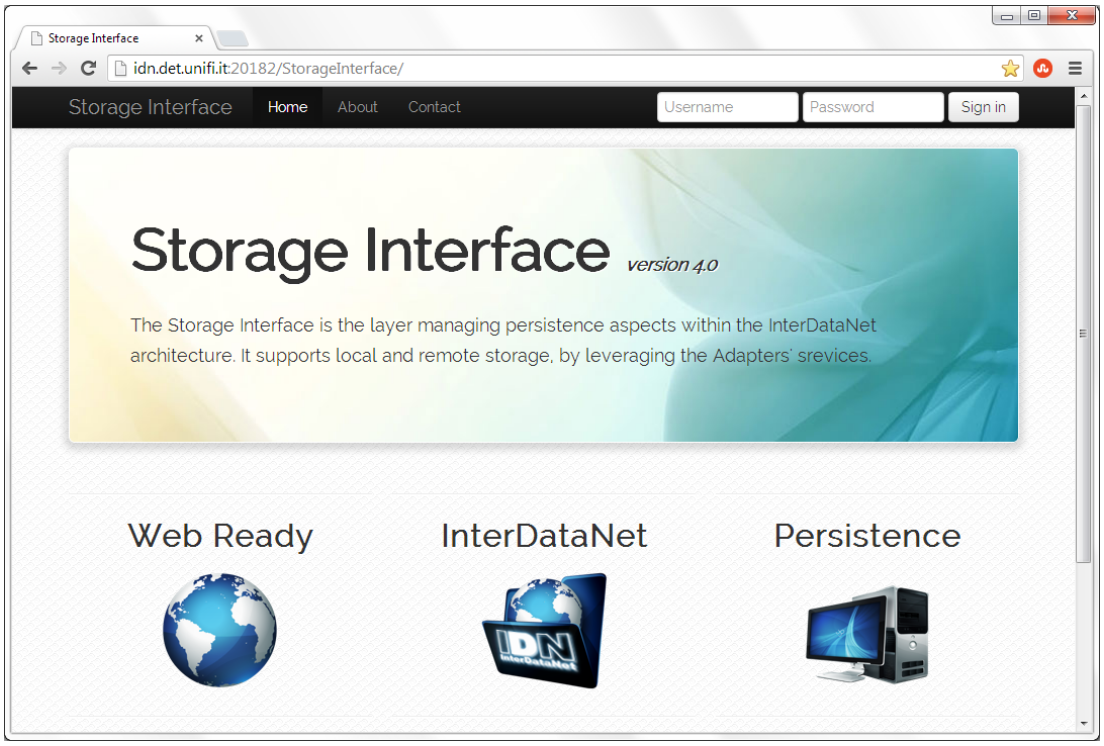


Figure 5.18. *The Storage Interface welcome page.*

those data directly created by an IDN-Compliant Application requesting a `write` operation on the Virtual Resource interface. For example, this is the case of a personal document created from scratch by a user. Those data didn't exist anywhere before, and now they are persisted in Storage Interface databases. Please note that data from external sources are not included in this class because they are persisted out of the architecture.

The latter, are all the architecture-specific information which are necessary to represent IDN-Documents as they are. This includes the digest from the top layer (i.e. properties metadata, Links, Activities, and so on) and metadata from the Storage Interface layer. This separation is represented by the two databases on the right of the figure.

Fig. 5.20 shows the main classes of the Storage Interface Implementation. Not to complicate the reading, as for Virtual Resource, only the essential is represented in the diagram. The Storage Interface application is a RESTful Java Web Application deployed on the Apache Tomcat Application Server and implementing the Model View Controller design pattern.

The entry point is the `NodeController` which exposes the interface and the SI-Node resource to the upper layer. Since no document is defined for this layer, no Document Manager class is present in the class diagram. Instead, a `NodeManager`

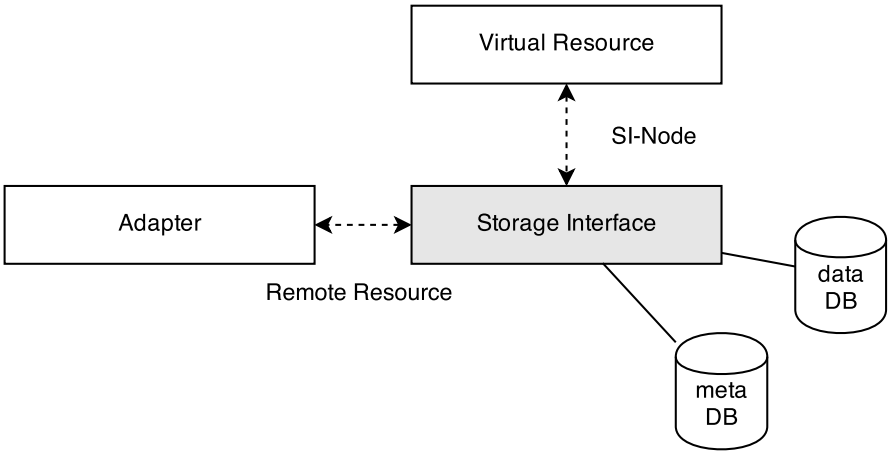


Figure 5.19. *The relation of Storage Interface with the other actors of the system.*

receives calls from the controller and performs the requested operations. As shown in the figure, the `NodeManager`'s methods mirror the ones defined for the `NodeController` and show a clearly CRUD nature

The `NodeManager` depends on the `AdapterManager` which implements a client interfacing with Adapters and manages Remote Resources, and the `Persistence-Manager` that provides capabilities for persistence, managing the storing of data and metadata in a own repository.

In the next section will be detailed the SI-Node resource.

5.3.1 The SI-Node

In the previous sections the SI-Node has been mentioned frequently, now it will be discussed thoroughly.

First, the SI-Node has a unique identifier, which is an URL. Aliases are not envisioned for names belonging to this layer. The reason for this is that aliases are a commodity especially for the application domain. When a document is assembled aggregating parts of other documents, makes sense that a new name reflecting the structure of the new document is used for it. For example, consider a driving license with the LRI `http:\\...\lic\2` aggregating a personal data document from a registry office with the LRI `http:\\...\reg\7\personal` . Probably it would be useful for the owner of the driving license to refer to personal data as `http:\\...\lic\2\personal` . Similar scenarios aren't present in the Storage Interface operating context.

Fig. 5.21 depicts a full representation of the SI-Node where different colors highlight different parts of the node.

`Etag`, `Http Status` and `Url` sections are intended to support the PES strategy.

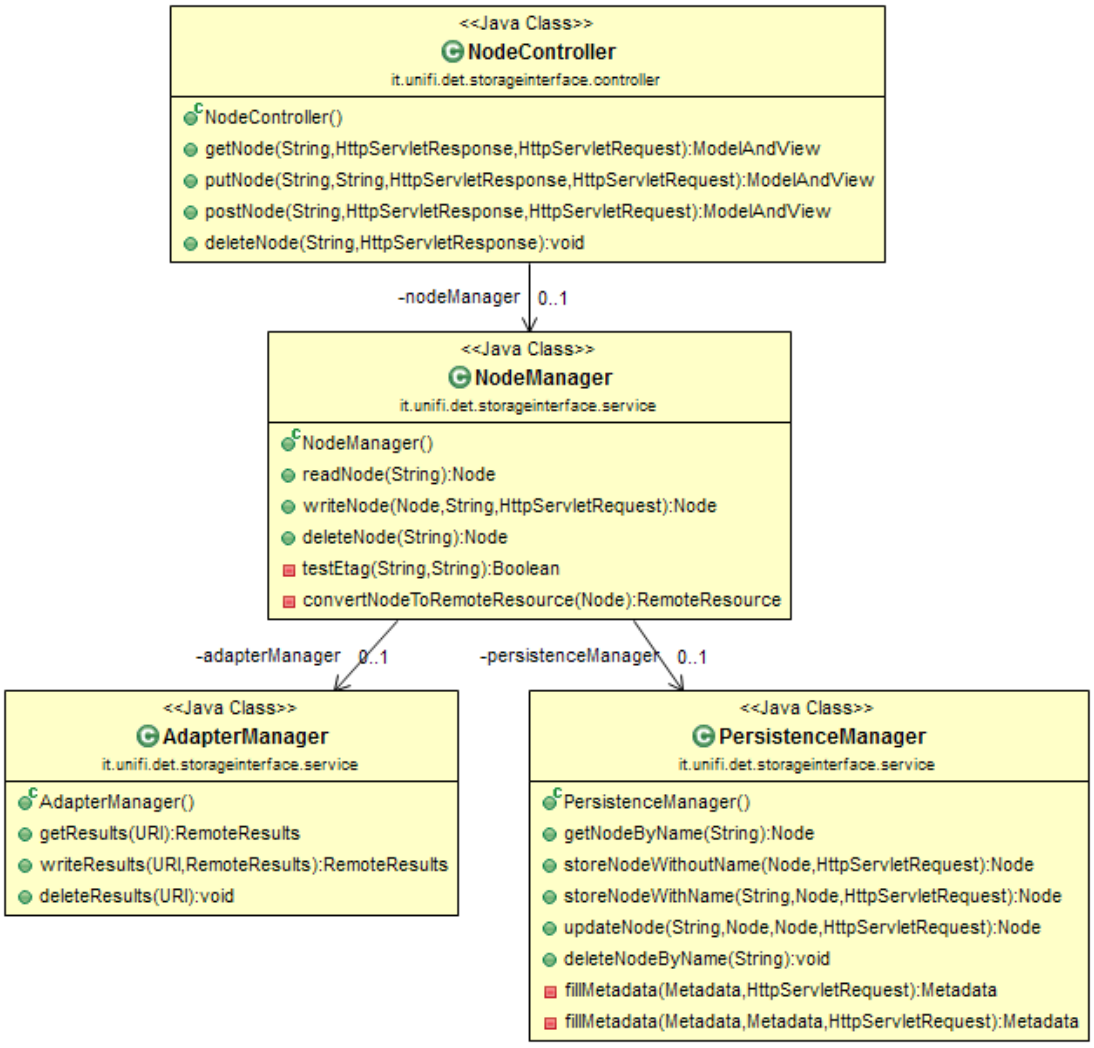


Figure 5.20. *The main classes of the Storage Interface implementation.*

In fact, as mentioned before, introducing PES means to allow architectural components to exchange "bags" of nodes. Within a "bag", the information concerning the management of the single unit must be given, just like the way it is explicated in the `NodeInfo` section of the `VR-Node`. The "bag" resource, must also be exposed (the official name chosen for it is "collection"), but this is still an ongoing work. Currently, Storage Interface supports it in terms of Data Model, but the PES strategy itself is not yet functional. For this reason, this aspect won't be discussed in greater depth.

The `Upper Level Digest` contains the information coming from the upper layer, as represented in Fig. 5.2. This element specifies a `Content Encoding` defining the encoding algorithm chosen for the digest (e.g., Base64) and a `Content Type` (the implemented architecture communicates internally using the XML format).

`Node Meta` includes an `Author` section containing information about the author of that SI-Node from the Storage Interface point of view. This section consists of `Application Instance Id`, `Host IP Address`, `Hostname` and `User` elements. `Autodestroy On` determines whether the life of the current node has a time limit or not. The `Modified On` element specifies the time reference of the current modification while the `Previous Modified On` tracks the time reference of the last modification. The `Owner` field determines the owner of that node while the `Previous Author` specifies the information concerning the executor of the previous modification.

Probably, the most interesting section is the `Content Data` which contains actual data coming from Adapter or Virtual Resource, depending on the operation currently performed. `Content Encoding` and `Content Type` elements have the same purpose of the ones described for the `Upper Level Digest` section and won't be repeated. The `Remote Location` element contains the URL of the corresponding Remote Resource, exposed by Adapter. Practically, when a request is issued for a SI-Node, if the `Remote Location` is set, the Storage Interface application won't attempt a data retrieval from its database but will issue a call to the URL specified in that section, instead. Of course, as a requirement, that URL must be managed by an Adapter since Storage Interface expects to exchange a Remote Resource.

Since the `Content` part is passed in plain from the upper layer, it can carry metadata useful for different architecture layers to assume peculiar behaviors related to the way the content has to be managed. The `Binding` element is one of them. Operations invoked by IDN-Compliant Applications, are propagated through the architecture, so that if an IDN-Compliant Application issues an HTTP GET, it will be propagated to Storage Interface by Virtual Resource (see section 5.2.1) which will propagate it to Adapter, if data are remotely stored. The same happens with the HTTP PUT. However, there is a special circumstance in which

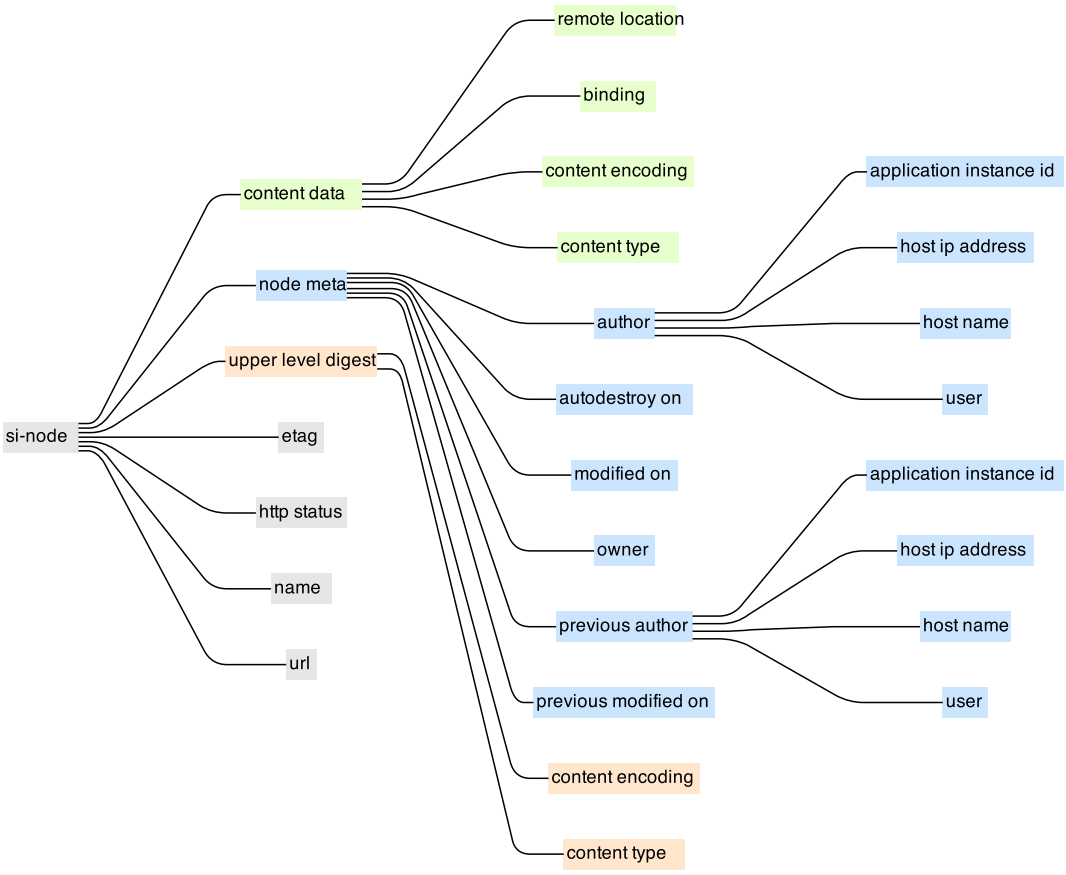


Figure 5.21. *The hierarchy of the SI-Node.*

this propagation should *not* be performed, i.e. when a binding is established for the first time between an information piece provided by a remote data source and its document form. This point will be detailed in section 5.4, for now is sufficient to understand that the difference between a data item and its counterpart in InterDataNet, consists in metadata added by the architecture to enable linking and other properties. This operation is referred to as "decoration": the term well catches the sense of the action, because it emphasizes the fact that data remain unchanged while a structure to enable new properties is built around them.

This means that the first time a data item provided by an external source is entered in the architecture, metadata must be created. If such data item shall be exposed as a document resource, then an X-Node must be created for every layer. As expected, this decoration is initiated via an HTTP PUT, which propagates through the stack as usual. The only difference is that, since no data managed by Adapter are going to be modified, the PUT shouldn't be propagated any further, and this is what the `Binding` flag element is for.

5.3.2 Reading a SI-Node

The read operation starts when the Storage Interface client sends an HTTP GET request to an URI matching the following template pattern:

```
http://idn_host/StorageInterface/!SI/sinode/{id}
```

where the `id` variable can be any sequence of characters. The application supports the content negotiation, therefore the `Accept` HTTP Header is inspected to render the proper view, as seen for Virtual Resource. If the requested format is not supported the application will output a response including an `415 unsupported media type` Status message.

The first step in processing the incoming request, is to extract the SI-Node identifier and to pass it to the `NodeManager`'s `readNode` method. This, forwards the request to the `PersistenceManager` that attempts a node retrieval from the local database. At this stage, a node is made up of metadata and data which can be empty or not. As discussed above, if the information is InterDataNet native, data will be immediately available, otherwise the URL extracted from `Remote Location` metadata will be used to query an Adapter. This check is performed when the node is returned to the `NodeManager`.

If the node is complete is returned to the `Node Controller`, otherwise the control passes to the `AdapterManager`. Now, the `getResults` method is invoked and an HTTP GET request is issued to the Adapter module.

Please note that there is an explicit $n : m$ mapping between SI-Nodes and data items coming from remote sources. Since the data owner has the right to define the way his/her data are represented as documents, it is also important to

let him/her decide the granularity level. In fact, these data can be represented as documents of different form, with IDN-Nodes containing more or less data from the original source.

If the returned node is `null`, the `NodeController` will set a `404 not found` HTTP Status, otherwise, if the request presented a `If_None_Match` HTTP Header, the value from the header is checked against the `Etag`. In case they are equal, a `304 not modified` Status is set. If this condition is not met, the resource is returned as is, with a `200 ok` Status Code. Fig. 5.22 shows the sequence diagram of a GET operation issued to Storage Interface, and triggering the retrieval of data from an Adapter. Please note how the `PersistenceManager` is involved anyway in the process in order to contact the database and obtain the empty node and metadata that will be used to fill it.

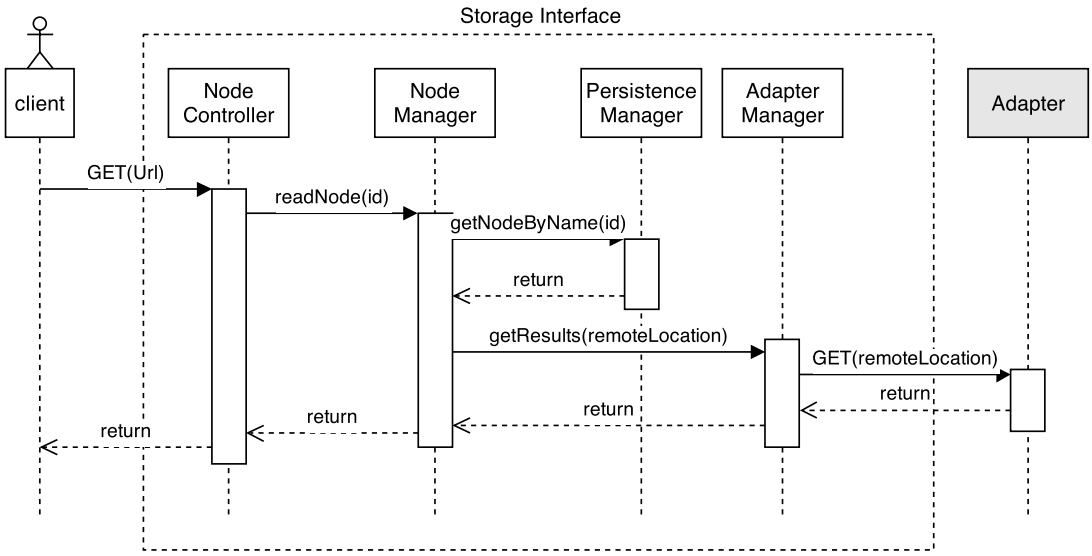


Figure 5.22. *The sequence diagram of a GET issued to Storage Interface, triggering a retrieval from an Adapter.*

5.3.3 Writing a SI-Node

Storage Interface application supports two different types of HTTP verb for creating resources: 1) the POST style, where the resource's name is decided server-side, and 2) the PUT style where the resource's name is decided client-side. As previously mentioned, the architecture takes advantage of this second approach, since LDNS is the module determining the name of the SI-Node that is going to be created. Since the PUT style serves well also for modification, and is more suitable for the unifying `write` approach, is the one to be discussed.

When an HTTP PUT is called on the SI-Node resource, the controller intercepts it and fires the `putNode` method. Since the PUT verb requires a Body, the `Content-Type` Header is mandatory. Because of the states conflict control discussed in previous sections, the `If-Match` Header specifying the `Etag` value is also required.

The controller invokes the `NodeManager`'s `writeNode` method, which attempts a retrieval of the SI-Node, through the `PersistenceManager` mediation. If the SI-Node exists, the `write` operation is interpreted to be an update.

Now, the `Etag` check is performed to assure states consistency, and the `AdapterManager` is invoked to let it issue a modification HTTP PUT at the `Remote Location` URL under the authority of an Adapter. Is not guaranteed that the remote source supports writing operations, and is reasonable to expect few cases of this type (e.g. social network profiles, requiring account credentials). However, it is important that the architecture is able to support all reasonable scenarios. If no corresponding SI-Node is found, the `write` operation is interpreted as a creation. At this point, is critical to discern the creation of an InterDataNet native object from the binding operation. To this purpose, the dedicated `Binding` flag element is inspected and the operation is delegated to the Storage Interface persistence service accordingly. In the first case, the node containing data and metadata is stored in the database by leveraging the `PersistenceService` class. In the second case, the `RemoteLocation` *must* be present, and metadata only are stored in the database. Fig. 5.23 shows a PUT operation resulting in an update of a resource hosted by a remote information provider.

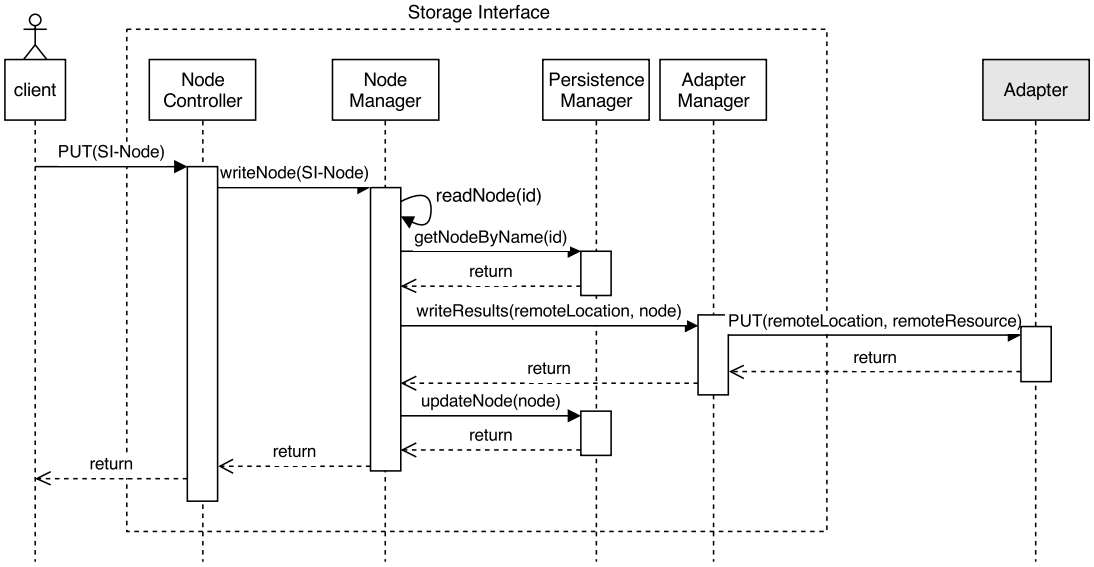


Figure 5.23. *The sequence diagram of a PUT issued to Storage Interface, modifying a resource hosted by a remote source.*

5.3.4 Deleting a SI-Node

As expected, Storage Interface implements the `delete` operation through the HTTP DELETE verb issued for a SI-Node resource URL. This event is caught by the controller which invokes the `deleteNode` method from the `NodeManager`. The `NodeManager` attempts a retrieval of the node to be deleted, by leveraging the `PersistenceService`. If the node can not be retrieved, the control is returned to the `NodeController` that produces a response with the `404 not found` Status Code set. Otherwise, is checked whether data are hosted locally or remotely; in the first case, the node is removed by the database, otherwise, the `AdapterManager` is called to propagate the deletion call to the Adapter module. After the response from the Adapter module is available, metadata are removed from the database. This operation is performed regardless of the outcome of the deletion issued to the remote information provider through Adapter. When a deletion is issued to Storage Interface, it means that InterDataNet is not entitled to keep these data anymore, and they must be removed from the architecture. According to the operation outcome, a proper HTTP Status is generated and returned by the controller to the client.

Fig. 5.24 shows the sequence diagram of a deletion issued to Storage Interface.

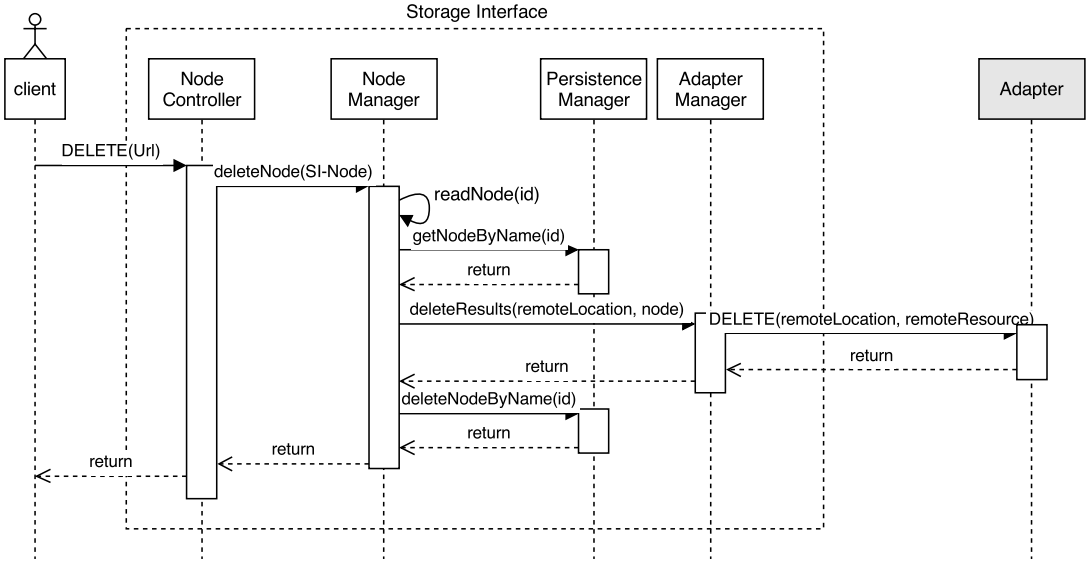


Figure 5.24. *The sequence diagram of a DELETE issued to Storage Interface.*

5.4 Adapter

Adapters represent the last boundary of the InterDataNet architecture towards external data sources. The main purpose of an Adapter application is to put in practice a strategy to ensure that data coming from a remote source are represented in compliance with the InterDataNet Information Model, i.e. as documents. To provide this service is very challenging, mainly because remote sources can significantly differ in many aspects, such as:

Access Strategy : two different sources can require different access channels, such as HTTP and FTP;

Access Requirements : some requirements could be put in accessing a source, such as authorization and authentication supporting confidentiality;

Interfaces : data sources have peculiar interfaces for accessing data. For example, HTTP based services can implement WS-* or REST styles;

Resources : different data sources provide different resources which can be differently mapped to documents. Managed resources types determine also advanced mapping strategies for decorating them.

This high heterogeneity calls for an *ad hoc* Adapter design, depending on the peculiar scenario. In other words, for each data source willing to connect to the InterDataNet architecture, a dedicated Adapter has to be designed, implemented and hooked up to Storage Interface.

While designing Adapter, has been paid great attention in trying to preserve as much generality as possible. Adapters are custom by nature, but this doesn't necessary mean that a general behavioral pattern can not be synthesized. The objective here is to provide a complete *template application*, which can be easily extended and adjusted for particular problems. In such way, analysts and developers are not forced to reinvent the wheel every time a new data source requires the InterDataNet Information Model decoration for own resources.

The design of Adapter template application required a significant abstraction effort, and many Java interfaces have been prepared for supporting developers. This aspects will be discussed in section 5.4.1.

Fig. 5.25 shows the relation of Adapter with other actors of the system. On the right side of the picture is represented the InterDataNet architecture, interacting with Adapter through the Storage Interface layer. On the left side, the external data source is represented as a cloud, to catch its natural heterogeneity. Adapter can interact also with Virtual Resource in order to create new documents decorating data coming form the external data source. This aspect will be detailed in section 5.4.2.

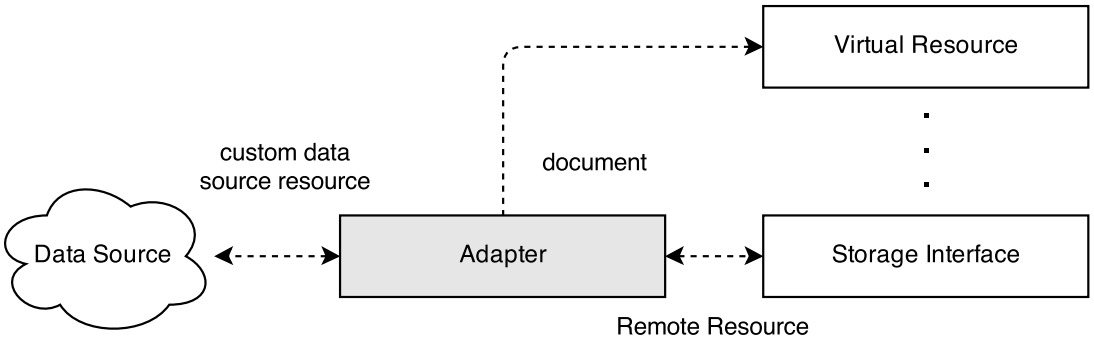


Figure 5.25. *The relation of Adapter with the other actors of the system.*

5.4.1 An Adapter Modular Design

To deal with the data source heterogeneity, the Adapter entity has been abstracted and studied in its different aspects and functionalities. The analysis led to the definition of five main modules, namely the Subscription Manager, the Notification Manager, the Transformer, the Document Manager and the Policy Manager, whose interaction is mediated by a main orchestrator component, the Main Manager.

Fig 5.26 shows these modules and how relate with each other through the Main Manager mediation.

In the following, each module is discussed.

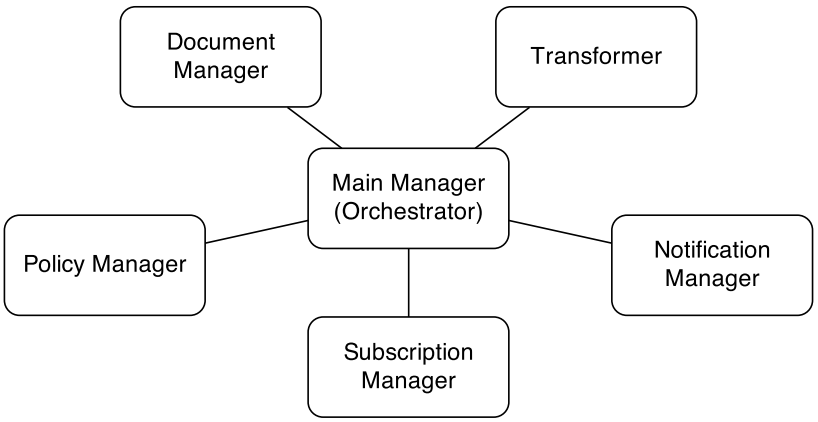


Figure 5.26. *The Adapter modules.*

5.4.1.1 Subscription Manager

When a data source is paired with an Adapter, a contract must be established. Addressing conditions under which this agreement is made is beyond the scope

of this thesis, but a subscription has to be managed anyway. The Subscription Manager module is thought to meet these requirements, either the subscription is managed on the side of the remote source or on the side of Adapter.

Both scenarios could occur, depending on context particularities: since data are provided by the remote source, it makes sense they are transferred only when Adapter undergoes an authentication procedure. Honestly, this is the scenario that is expected to occur more frequently. In this case, valid credentials should be provided to Adapter and a secure authentication and authorization procedure must be enforced. The Subscription Manager module will take care of managing these credentials and supporting the required security standard, eventually demanded by the external source, such as an HTTP over TLS [DA99, DR06, Res00] communication channel.

However, it shouldn't be taken for granted that a data source is willing to invest resources for implementing a secure service. Therefore, the "secure coupling" between Adapter and the source could be delegated to the former. In this case, the Subscription Manager should manage all the security aspects, and the data source could declare its own requirements to be implemented on the Adapter side. It is clear that an Adapter-side enforced security *can not* protect data exposed through the remote resource interface: no security is implemented on it, thus they are just public. However, a limited degree of security will affect the document representation of these data, which is undoubtedly a desirable effect.

It goes without saying that InterDataNet security does *not* rely on this aspect only. A comprehensive Security Framework is in place for its enforcement in terms of authentication, authorization, identification, confidentiality and integrity.

5.4.1.2 Notification Manager

Since data coming from external sources are obviously coupled with their document representation, a notification service can be envisioned to manage notification messages between Adapter and sources.

The problem does not affect data modifications from the side of the architecture. In sections 5.3.3 and 5.3.4 has been explained the adopted strategy for propagating modifications from documents to Si-Nodes to Remote Resources, and consequently, to external data. However, it could be useful for the remote source to be notified when changes in the document structure occur, e.g. when a Link is removed from a node wrapping external data. It is worth noting that this capability may require a notification service available for the architecture. Such feature is not currently available, even though is an ongoing study. In addition, a notification in the opposite direction would be useful, e.g. to make Adapter aware of some changes in mapped resources.

For example, in a real world scenario (see chapter 7), an Adapter is used to decorate sensor data coming from a sensing infrastructure. Available sensors

change with time, and a notification flow from the source to Adapter comes very handy. When new sensors are available, a notification can trigger a procedure on the Adapter module to automatically decorate and publish them as documents.

5.4.1.3 Transformer

The Transformer is a core module for Adapter because manages transformation procedures from the remote source data format in the document data format. The transformation tends to be complex and probably the most relevant part of the Adapter customization. Indeed, at this stage, data from the remote source have to be split up and associated with the proper IDN-Node. Fig. 5.27 shows an example of a mapping between an object from a source and a document.

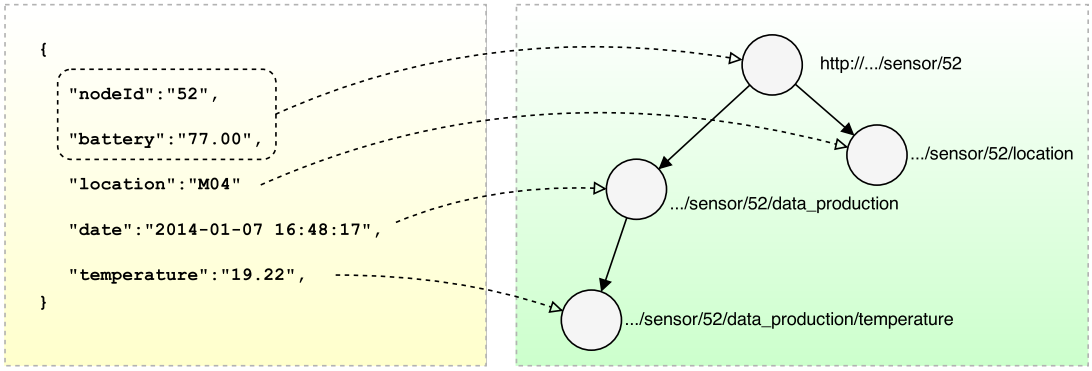


Figure 5.27. *A mapping between the object form a remote source and a document.*

Data shown on the left of the image comes from a remote data source and are a Json representation of a sensor. On the right there is the document representation of the same data. In order to complete the transformation, a model of a document sensor must be given as input to Transformer module. Indeed, different valid document representations can be associated to such data and picking one instead of another is just a design choice. In this case, the sensor `id` has been assigned to the LRI and the root node, along with the `battery` information. Other data units have been associated with individual nodes.

It is also possible to envision the inverse procedure, i.e. that data from different sources or more data units from the same source are aggregated in a single node. Analogously, Transformer will need to be provided with the mapping between the source and destination information. Currently, the mapping is hardcoded, however it would be far a better solution to define a language for prototyping documents. Probably XSLT [Kay07] or a similar technology could serve the purpose. Insights in this direction are planned as future works.

5.4.1.4 Document Manager

The Document Manager module implements a client towards the Virtual Resource interface, to let Adapter interact with document resources. This interaction is strictly defined as a creation HTTP PUT (see 5.2.2) and is performed only once, during the initialization phase. The operation will be detailed in section 5.4.2.

Apart from this, the module does not have additional tasks.

5.4.1.5 Policy Manager

Policy Manager is a core component since translates requests coming from the architecture in requests meaningful for the data source. InterDataNet takes advantage of the REST architectural style, and let modules interact by defining resources and invoking HTTP requests on them. A data source may have a completely different paradigm and a mediation is required. Indeed, is likely that an update operation coming from the architecture, which is implemented as an HTTP PUT request, would map in a different operation, say a POST, or even a GET. It could also possible that requests coming from Storage Interface require to trigger a complex workflow on the side of the external source, comprising different calls to different components.

This is a core configuration point for Adapter, similar to the Transformer's one, in terms of significance. As for Transformer, the mapping between incoming requests and outgoing calls is currently hardcoded. Adapter would benefit considerably by implementing an external configuration point for this.

5.4.2 Dealing with the Variability of Resources

While dealing with document representation of external data, two strategies can be mainly pursued: a dynamic decoration and a configuration-based decoration. Decorating dynamically means to build the document and X-Nodes on the fly, when a request is issued to InterDataNet for that resource.

The system must be aware of the fact that data decoration hasn't been performed, but it reacts as if the document representing these data had always been available within the architecture. In other words, when the external source is connected via Adapter, its data are available as IDN-Documents, even if no IDN-Document has been created for them. The system just builds everything on the fly, when requested.

The other approach is about configuring document resources representing external data *before* they are made available by the architecture. If a sensor external object has to be decorated and exposed as a graph of individually addressable information, a preliminary initialization step in which architectural metadata are created for this object is required. From that moment on, the "sensor document"

is made available by InterDataNet, with all the operations and properties enabled.

InterDataNet is ill-suited to the dynamic creation of IDN-Nodes, for a number of reasons: First, the architecture follows the encapsulation strategy described in section 5.1.1, and the order in which information is generated can *not* be broken. If this happens, is possible to face the case in which a layer envelope generated at a lower level does not wrap the corresponding upper layer resource. For example, if a SI-node is generated inside Storage Interface, it won't wrap any upper level digest, since it is not able to reproduce a VR-Node in any way. In order to reach the result, a VR-Node should also be generated inside Storage Interface, resulting in a major violation of architectural principles.

A second controversial issue is that non-existing resources should be assigned to existing names, because they have to be generated and served as they would exist. Such names could be delivered in explicit form or as a class. In the first case every single LRI would be added to the LDNS index, while in the second case a URI-template [GHNO08] could be used to represent a set of URIs.

Both strategies pose problems. It is not always possible to know in advance the exact objects to decorate, and an explicit enumeration of LRIs could be unfeasible. On the other hand, the definition of "reserved" classes of names leads to a big complexity increase in names management and also a significant waste of them. Consider again the case of the sensors external source. A suitable name for a sensor document is

```
http://.../sensor/{id}
```

where `id` is a path variable identifying a sensor. If this variable is an arbitrary array of characters, an infinite set of names should be reserved just to expose few. It could also be possible that some names belonging to this space are already assigned and can't be reserved. In this case, the class definition should be provided with logical operators such as conjunction, disjunction and negation, implying a complication of the required formalism.

Moreover, consider that an HTTP PUT is issued for the LRI

```
http://.../sensor/145.
```

Since is not possible to know whether the resource exists or not at LDNS level, the nature of the operation remains ambiguous until the data source is contacted (is it a creation or an update attempt?), and this results in a considerable architectural inefficiency.

Last but not least, even assuming to have handled all these problems, it wouldn't still be possible to obtain a structured document. With a dynamic approach, data from eternal sources could be exposed merely as nodes, since the

relation information, i.e. the Links, resides in the VR-Node metadata and can not be inferred by the system and, therefore, generated on the fly.

These considerations have led to undertake the path of the configuration-based decoration, i.e. to generate the architectural resources such as the VR-Node, the LDNS record and the SI-Node before making the decorated content available by the side of InterDataNet. This approach suites well the architectural operation flows, because takes advantage of the natural pattern defined for the content creation.

External resources can be roughly categorized in two groups: countable and uncountable[1]. A data source exposes a countable set of resources when their number is limited. They can be appropriately represented with a URI-Template whose variables assume a reasonable number of values.

When the first case occurs, a pure configuration-based approach can be adopted. The data source connects to Adapter and, according to the $n : m$ mapping defined for the use case, documents are generated and submitted by Adapter to the architecture. These documents contain the required metadata and handles to retrieve corresponding remote resources, as detailed in section 5.3.3. This procedure can be considered, in all respects, an initialization.

Fig. 5.28 shows how this pattern is implemented. A connection between Adapter and Virtual Resource that will be authoritative for novel documents is in place and Adapter uses such connection to perform the initialization procedure. In this way, architectural metadata are entered in the system seamlessly, and the document is readily available. Unlike the dynamic decoration approach, a *full* document with all the relations in place is generated, as depicted in Fig. 5.27.

However, this approach is not a universal solution, due to the variability of external resources. Sometimes, data sources expose uncountable sets of resources, especially when they expose RPC interface requiring input parameters. A trivial example, is a forecasts service. This service could expose a forecast resource named as follows

```
http://.../forecast/{location}/time/{time}/offset/{offset}
```

or, more realistically,

```
http://.../forecast?location=location\&time=time\&offset=offset,
```

being `location` the place where the forecast has to be evaluated, `time` the temporal reference, and `offset` the time span (ten hours, a week, fifteen days) of the forecast. It is clear that it would be senseless to create an InterDataNet entry for all the possible combinations of parameters.

[1]These terms are borrowed by the Set Theory without any pretense of mathematical rigor. They should be considered as informal definitions.

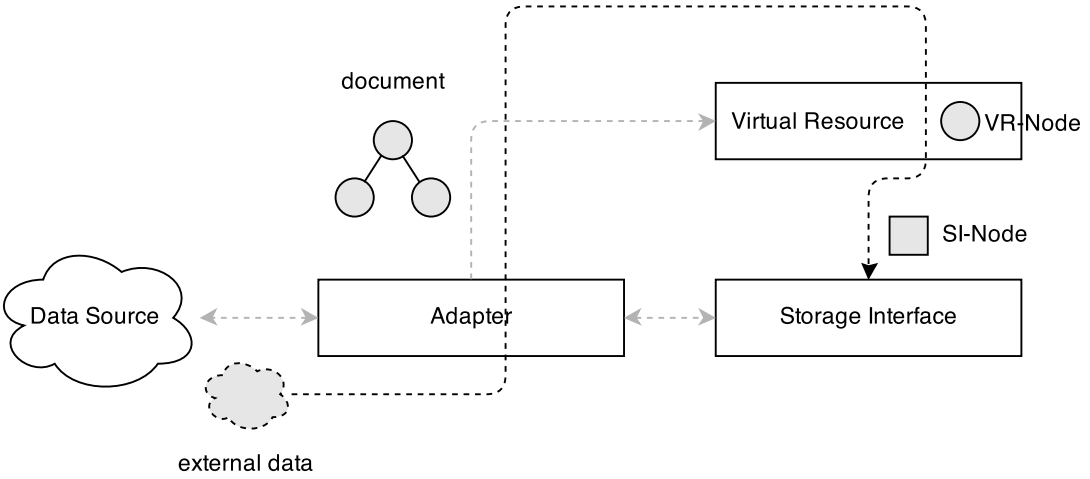


Figure 5.28. *The initialization procedure performed by an Adapter.*

The solution is to use a semi configuration-based approach by supporting dynamics with a document template. For example, in order to decorate data from the forecast source, the following steps are needed:

1. design a shell document for the forecast resource
2. create and publish such document to Virtual Resource
3. fill that pre-existing document on the fly, by leveraging URI query support.

The main advantage of this strategy is that no architectural metadata have to be created on the fly, although, in fact, the document is dynamically filled. LDNS has to manage just *one* entry for every node composing the document, because the variability of external resources is managed with parameters. For example, the LRI of the node representing the forecast in Chicago for the next week from now can be

```
http://.../VirtualResource/!VR/.../forecast?location=chicago&time=now&
offset=week.
```

The drawback is that external data are bound to a shell documents to be filled from time to time. This means that such document can not undergo an instance specific modification. More concretely, if a client is willing to decorate the forecast for Chicago with some pictures of the city, it won't. In addition, a propagation strategy for query parameters while resolving a document must be strictly defined. Such propagation rule can be defined as follows: if a resolution is issued for a shell document made up of n nodes, query parameters will be propagated while

resolving each and every node. However, if during the resolution an Aggregation Link containing a different query is encountered, from that point on, the new query parameters will be propagated instead. This makes sense because, by declaring specific query parameters for an LRI, is intrinsically assumed that a *document*, i.e. a set of related nodes whose meaning is dependent by these parameters, is requested. In other words, returning to the previous forecast example, is possible to request forecasts for Chicago instead of New York only as a consequence of a parameter definition. Logically, these documents are two utterly separate entities.

Fig 5.29 shows the query propagation strategy with an example. In the upper right there is a document shell describing the event in the city of Chicago. It is possible to select the event type, by specifying the `type` parameter. Since the document is inherently bound to that city, make sense to link it with the corresponding forecast document. To this end, a node of the "events" document has a parameterized Aggregation Link pointing to the forecast document on the left. When a client issues a GET request to the `http://.../chicagoevents/?type=concert` the `type=concert` parameter is propagated until it reaches the Aggregation Link to the forecast document. From that point on, it would be senseless to propagate the `type=concert` parameter. In fact, the new `location=chicago` parameter is propagated for the resolution of the forecast document instead.

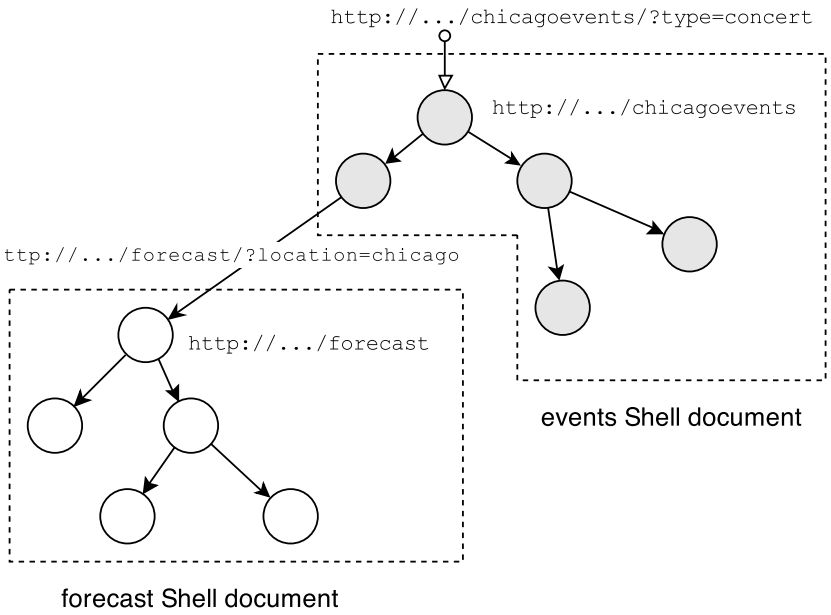


Figure 5.29. *The query propagation strategy.*

Chapter 6

InterDataNet Tool-suite

One of the main requirements of the InterDataNet framework is being easy to use. The technology proceeds at a fast peace, opening to new capabilities and possibilities. However, people should be carefully supported in exploiting technology advances, for at least three reasons. The first is trivial: technology is for people, and therefore it should be accessible. Second, sometimes the great rise of a technology happens when people understand it and start to use it. Communities of contributors have a big impact on a technology improvement. Third, certain technologies are massive by nature (the Web is one of those), and if they fail to spread their intrinsic utility decreases.

In order to keep easy the interaction of users with InterDataNet, a Tool-suite has been designed and implemented. It comprises four tools, dedicated to different types of users. In fact, three different types of users can be identified: the one implementing a web server, the one implementing a client and the one willing to manage documents, without bothering with custom implementations. For users implementing web server interacting with the architecture, a Java library has been released, while a Javascript library is provided to users who prefer a client-side implementation. IDN-Studio is a visual tool that help users to manage documents with a very simple user interface, while IDN-Viewer is a JQuery plugin for rendering IDN-Documents in a human oriented way.

In this chapter these four components of the InterDataNet Tool-suite are discussed.

6.1 IDN Java Library

The IDN-Java Library is a tool dedicated to the Web Application server-side developer. It comprises all the classes necessary to handle the IDN-Document with ease, still preserving flexibility. The main components of the library are the following:

- the document class hierarchy representing the resource exchanged with the architecture
- `IdnElement` classes to traverse and manipulating the document
- a `IDNClient` service to contact the architecture and perform CRUD operations.
- a (un)marshaller for the resources (de)serialization (integration of third party libraries).

The document class hierarchy mirrors the one depicted in Fig. 4.14 and addressed in section 4.2.1. The actual implementation have classes dependencies that are not present in the theoretical representation, this is due to the creation of convenience methods and attributes useful for developers. For example, the `Document` class has a reference to a map of `Node`s, even though the theoretical structure associates nodes to envelopes only. Fig 6.2 shows a minimal representation (including attributes) of these classes.

This section is intended to be an high level description of the library rather than a technical manual, and because of this, the discussion won't dig into the details of the document hierarchy classes. Conversely, more focus is given to the node traversing facility provided by the `IdnElement` classes because it allow a powerful and flexible node processing.

The problem motivating a facility for traversing the node tree is the following. Documents and nodes are complex objects which can be processed in many different ways. When is necessary to perform an heavy nodes processing, is critical to have a strategy to do it in a general way, without relying solely on the `getter` and `setter` manipulation which produces an incredible amount of code, increasing the probability of error and lowering the overall quality.

A good example is the node merge operation. Given two nodes, each and every element of the nodes tree structures is compared to determine how to perform the merge. Without the traversing facility, this operation required to visit every element by explicitly invoking the correspondent `getter`, and eventually invoking the `setter` to modify the attribute value. This forces the developer to write a lot of boilerplate code, performing the very same operation on different node elements. To make the idea, an example of the node structure visit by `getter` invocation is shown in Listing 6.1.

```
1  if(newNode.getApplicationMeta().getAuthor() != null)
2  {
3      if(!newNode.getApplicationMeta().getAuthor().hasElements())
4      oldNode.getApplicationMeta().setAuthor(newNode.getApplicationMeta().
       getAuthor());
5      else
6      {
7          if(oldNode.getApplicationMeta().getAuthor() == null)
8          oldNode.getApplicationMeta().setAuthor(newNode.getApplicationMeta
           ().getAuthor());
9          else
10         {
11             if(newNode.getApplicationMeta().getAuthor().
               getApplicationInstanceID() != null)
12             oldNode.getApplicationMeta().getAuthor().
               setApplicationInstanceID(newNode.getApplicationMeta().getAuthor
               ().getApplicationInstanceID());
13             if(newNode.getApplicationMeta().getAuthor().getHostIpAddress()
               != null)
14             oldNode.getApplicationMeta().getAuthor().setHostIpAddress(
               newNode.getApplicationMeta().getAuthor().getHostIpAddress());
15             if(newNode.getApplicationMeta().getAuthor().getHostName() !=
               null)
16             oldNode.getApplicationMeta().getAuthor().setHostName(newNode.
               getApplicationMeta().getAuthor().getHostName());
17             if(newNode.getApplicationMeta().getAuthor().getUser() != null)
18             oldNode.getApplicationMeta().getAuthor().setUser(newNode.
               getApplicationMeta().getAuthor().getUser());
19         }
20     }
21 }
```

Listing 6.1. A merge operation performed without the traversing facility.

Conversely, by introducing the traversing mechanism, the node can be explored automatically, without forcing the developer to write a case for every element. For the merge operation, this strategy allowed to implement a very compact method, reducing the code length by approximately the 90%.

6.1.1 The IDN Element

The IDN Element implementation is the core of the traversing mechanism. Basically, regardless of their nature, each element participating to the node structure is an IDN Element. Every IDN Element has three main attributes: a parent which is also an IDN Element, a list of siblings and a list of children, as shown in Fig. 6.1.

The IDN Element is implemented as a concrete Java class called `IdnElement`. An IDN Element is an entity located inside an IDN Node, with exception of some

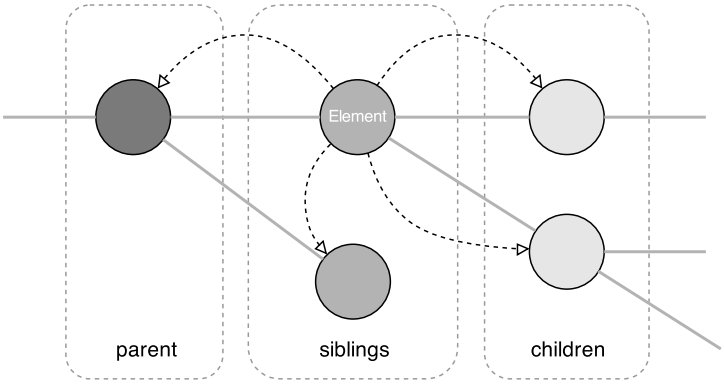


Figure 6.1. *The IDN Element, retaining the reference towards the parent, siblings and children.*

leaves which can not extend the `IdnElement` class. For example, the `String`, `URL`, and `int` types can't be represented as IDN Elements. Nevertheless, such elements are represented in the hierarchy as well, since they necessarily have a parent which extends the `IdnElement` class.

Please note that every operation performed on the `IdnElement` structure is reflected on the underlying element. For example, invoking a `removeChildren` on an `IdnMeta` will cause the deletion of every child, recursively.

Basically, the `IdnElement` class exposes two types of public methods: the ones to traverse the structure and the ones to initialize the parent, siblings and children attributes. This operation is performed by the `buildElement` method that builds recursively the node hierarchy, by leveraging Java Reflection. Java Reflection APIs are a set of classes thought for the dynamic interrogation of class instances. In this way, is possible to know the names of attributes and methods of a generic class (and much more) and even invoke them at the execution time.

The methods to traverse the structure are thought to manipulate the node elements through the parent, sibling and children relations. This is similar to what happens in the JQuery framework for Javascript. In the following the comprehensive set of method is listed.

`getParent()` retrieves the parent element of the current IDN Element.

`getSiblings()` retrieves the siblings of the current IDN Element.

`getChildren()` retrieves the children of the current IDN Element.

`hasParent()` `true` if this IDN Element has a parent, false otherwise.

`hasSiblings()` `true` if this IDN Element has at least one sibling, false otherwise.

`hasChildren()` `true` if this IDN Element has at least one child, false otherwise.

`addChild(Object childToAdd)` when the IDN Element to add the child to is a list, this method can be invoked. Otherwise, the `setChild()` should be used the instead.

`addChildren(List<Object> childrenToAdd)` when the IDN element to add children to is a list, this method can be invoked.

`addSibling(Object siblingToAdd)` when the IDN Element to add the sibling to is a list or a Node, this method can be invoked. Otherwise, the `setSibling()` should be used instead.

`addSiblings(List<Object> siblingsToAdd)` when the IDN Element to add the siblings to is a list or a Node, this method can be invoked. Otherwise, the `setSiblings()` should be used instead.

`setChild(IdnElementType childType, Object childToSet)` it attempts to set the value of the declared child to the one passed as an argument. If the parent class is an `IdnList`, the retrieval of the list element with a matching id will be attempted. If it can be found it will be replaced with the child to set, passed as an argument, otherwise the child to set will be appended to the list, delegating the `addChild()` method. Setting a child to `null` is not allowed, it should be used the `removeChild()` method instead.

`setChildren(Map<Object, IdnElementType> childrenToSet)` it attempts to set the value of the declared children to the ones passed as an argument. If the parent class is an `IdnList`, for every child to set the retrieval of the list element with a matching id will be attempted. If it can be found it will be replaced with the child to set, passed as an argument, otherwise the child to set will be appended to the list, delegating the `addChildren()` method. Since the method works child by child, some children might be updated and some might be added. Setting children to `null` is not allowed, it should be used the `removeChildren()` method instead.

`setSibling(IdnElementType siblingType, Object siblingToSet)` it attempts to set the value of the declared sibling to the one passed as an argument. If this class is an `IdnList`, the retrieval of the list element with a matching id will be attempted. If it can be found it will be replaced with the sibling passed as an argument, otherwise the sibling to set will be appended to the list, delegating the `addSibling()` method. Setting a sibling to null is not allowed, it should be used the `removeSibling()` method instead.

`setSiblings(Map<Object, IdnElementType> siblingsToSet)` it attempts to set the value of the declared siblings to the ones passed as an argument.

If this class is an `IdnList`, for every sibling to set, it will be attempted to retrieve the list element with a matching id. If it can be found it will be replaced with the sibling to set passed as an argument, otherwise the sibling to set will be appended to the list, delegating the `addSiblings()` method. Since the method works sibling by sibling, some siblings might be updated and some might be added. Setting siblings to `null` is not allowed, it should be used the `removeSiblings()` method instead.

`removeChild(Object child)` removes the specified child from this element

`removeChild(IdnElementType childType)` it tries to remove the child by its `IdnElementType`. This operation is possible only when there is no ambiguity on the child type. If so, it will return `false`.

`removeChildren()` removes all the children of this element from the node hierarchy.

`removeChildren(List<Object> childrenToRemove)` removes the children of this element specified in the list from the node hierarchy.

`removeAllChildrenBut(List<Object> childrenToKeep)` removes all the children of this element but the specified ones, from the node hierarchy.

`removeSibling(Object sibling)` removes the specified sibling from this element.

`removeSibling(IdnElementType siblingType)` tries to remove the sibling by its `IdnElementType`. This operation is possible only when there is no ambiguity on the sibling type. If so, it will return `false`.

`removeSiblings()` removes all the siblings of this element from the node hierarchy.

`removeSiblings(List<Object> siblingsToRemove)` removes the declared siblings of this element from the node hierarchy.

`removeAllSiblingsBut(List<Object> siblingsToKeep)` removes all the siblings of this element but the ones specified as an argument from the node hierarchy.

`removeAllSiblingsBut(Object siblingToKeep)` removes all the siblings of this element but the one specified as an argument, from the node hierarchy.

`getChildrenByType(IdnElementType type)` returns a List of children matching the type declared as an argument.

`getSiblingsByType(IdnElementType type)` returns a List of siblings matching the type declared as an argument.

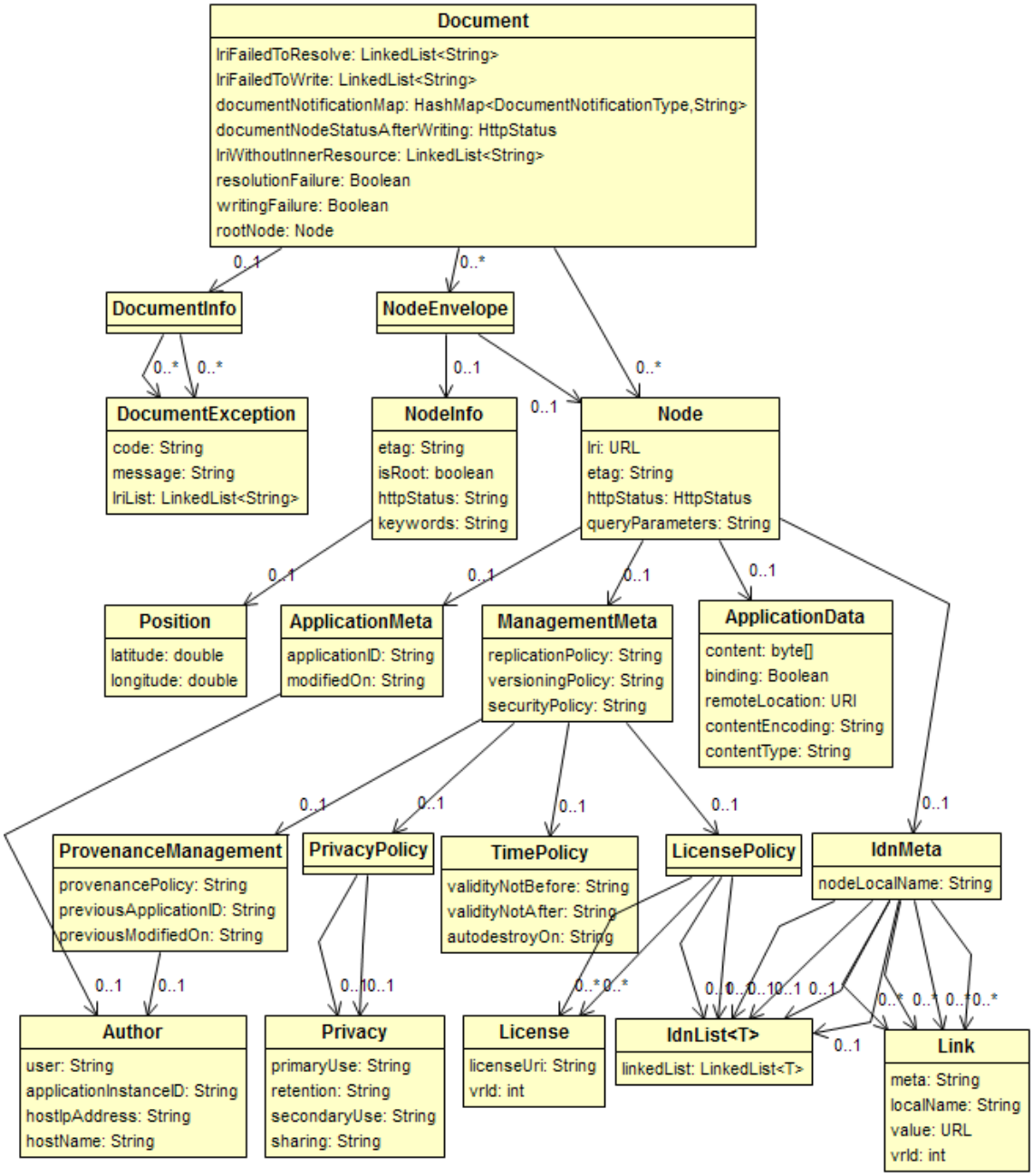


Figure 6.2. *The document class hierarchy.*

6.2 IDN.js

IDN.js is a Javascript library that shares the intent with the IDN Java Library. It is thought to be an alternative for developers who are more familiar with client-side programming. It leverages JQuery [MDV04, DV06, BK08] as a helper library and JSON-js [Cro10] for (un)marshalling document resources (IDN.js embeds an AJAX [G+05] client which accepts Json documents from the InterDataNet architecture).

Basically, IDN.js provides a client to manage calls towards the architecture and a document implementation. The traversing facility are given by the JQuery library.

The class embedding the client is called `Manager` and defines methods to retrieve single node or multiple nodes documents, delete and write documents. The class representing the node is `Node` and exposes method for managing the content links and metadata. APIs are not significantly different than the one discussed for the Java Library, and therefore won't be detailed here.

6.3 IDN-Studio

IDN-Studio [Fai12] is a Javascript Web Application for the visual management of documents, providing the user with a full control over the graph structure and data. In order to be actually effective, it presents a simple and usable interface, allowing the user to focus on the management of the graph and the information contained therein [J+10]. IDN-Studio interacts with InterDataNet through APIs exposed by the Virtual Resource layer (see section 4.2.2) and allow users to create new documents, delete and modify existing documents by adding or removing edges, contents and properties.

A very important feature, is that IDN-Studio is flexible with respect to document resource modifications. Since InterDataNet is a research topic, is very likely that the document definition changes with time. IDN-Studio is designed to be configurable in this sense, taking as input an XML Schema representation of the IDN-Document. This characteristic gives IDN-Studio great longevity.

In addition, the choice of implementing it as a Javascript Web Application implies four major advantages: first, users don't need to install software. The installation is performed once on the server and form that moment on is freely accessible. Second, since the application is hosted on a central server, is very easy to keep up to date. Third, the service is accessible by every client machine connecting to the central server without requiring multiple installations as happens to Desktop Applications and finally, it is portable, because the only support needed to access such it, is a common browser, regardless of the underlying platform or operating system [Jaz07, Dar06].

6.3.1 Graphical User Interface

In order to help the user to focus on documents management, IDN-Studio implements a very minimalist and clean interface [J+10]. Fig. 6.3 shows a mock-up of the Graphical User Interface. The Web Application is divided in two main areas: the toolbar and the canvas. The former is organized in two levels, the upper part has commands to perform operations involving the document as a whole, such as:

New Document to create a document from scratch, specifying its LRI.

Open Document to open an existing document, specifying its LRI.

Write Document the user confirms the modification for the document displayed on the canvas and performs a submission to the InterDataNet architecture.

In the lower section of the toolbar there are commands to perform operations affecting the canvas, such as:

Tool Selection the user can select one of the three available tools, namely hand, pencil and rubber. IDN-Studio will react to the user actions, depending on the selected tool.

Zoom to zoom in or zoom out the canvas area.

Canvas Sizing to resize the canvas. This option is very useful when working with very large documents.

Layout Selection through the dropdown layout menu is possible to choose the layout that suites better the document graph. The available layouts are Spring (for highly connected graphs), Tree (for tree-like graphs) and Random.

Perspective Shift through this dropdown menu the user can select which type of edge represent in the graph. Indeed, due to readability issues, edges are not simultaneously displayed on the canvas.

In addition, a search bar is available for querying the InterDataNet Search Engines network and helping users in retrieving documents. As happens with the latest search facilities on the Web, this search bar sends queries on typing, providing real time suggestions and improving the user experience.

The position and the look and feel of bars and buttons have been chosen according to the Gestalt Principles, similarity and proximity principles [SvE88], in particular.

Much of the graphical interface is dedicated to the canvas where documents are represented as graphs whose vertexes are connected by edges of different nature, according to the selected perspective. To avoid ambiguity, different vertexes have

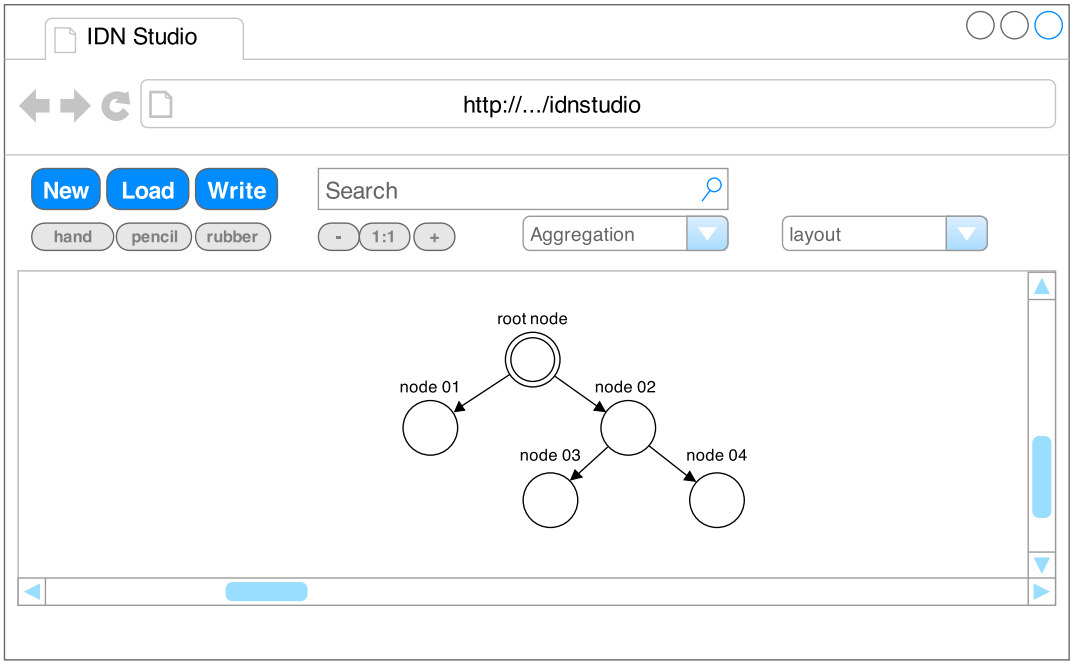


Figure 6.3. *The mock-up of the Graphical User Interface.*

different colors. When a document is displayed on the canvas, the user has a full control over it. He/she can move it with a *drag and drop* interaction, edit it, adding or removing edges of different nature, creating new nodes and documents, link them together, and so on. IDN-Studio has been conceived to preserve the InterDataNet APIs capabilities while masking complexity providing the user with a visual metaphor of the document. A very handy feature is the possibility to load more documents on the same canvas, and interact with them by linking their nodes. This capability make the data composition from different sources really easy and is considered to be one of the main feature of IDN-Studio. Fig. 6.4 shows the IDN-Studio Web Application with two documents displayed simultaneously on the canvas. Documents have different colors to be identifiable at sight.

In Fig. 6.4 is also possible to spot opaque nodes. These are called "ghost nodes" and represent nodes that haven't been retrieved by the application yet. As discussed in section 4.2.2, documents can be retrieved with different resolution depth. This implies that is possible to have nodes with unexplored outgoing edges. In IDN-Studio has been chosen to follow the aforementioned strategy, because results very intuitive for users. Moreover, is possible to interact with a ghost node requesting its retrieval with a double click. If the newly retrieved node presents other outgoing links, they will be represented as ghost nodes as well and the user can repeat the operation. This enables an actual browsing of the document. Of course, the user can browse different types of links by selecting the correspondent

perspective.

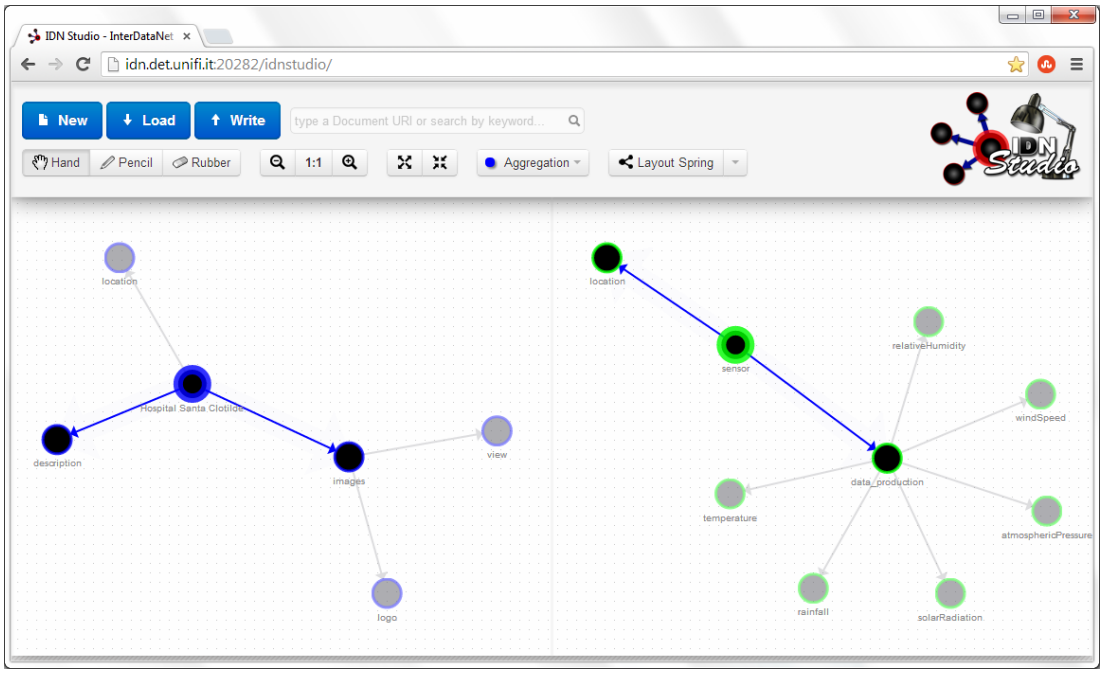


Figure 6.4. *The IDN-Studio Web Application with two documents displayed on the canvas.*

When a document is going to be retrieved or created, the Web Application must be provided with the LRI. Since IDN-Studio manages IDN-Documents, the LRI is also an IDN-URI (see section 4.2.2). To help the user in entering the proper IDN-URI, a modal window containing a form is displayed. Here, the IDN-URI is divided into main components (`host`, `application`, `VR`, `resource` for creation. If the ongoing operation is a retrieval the , `inner path`, `content path`, `version`, and `resolution depth` are also available).

When the document is displayed on the canvas, the user can edit every single section of its nodes. To this purpose, he/she can double click on a graph vertex to enter the node edit mode. A modal window (Fig. 6.5) where every node section is available for editing is presented to the user. All sections can be expanded to reach the the attributes, which are modifiable with a different widget, according to the input type (a text will be rendered as a text area or an input field, a boolean with a check box or a radio button, etc.).

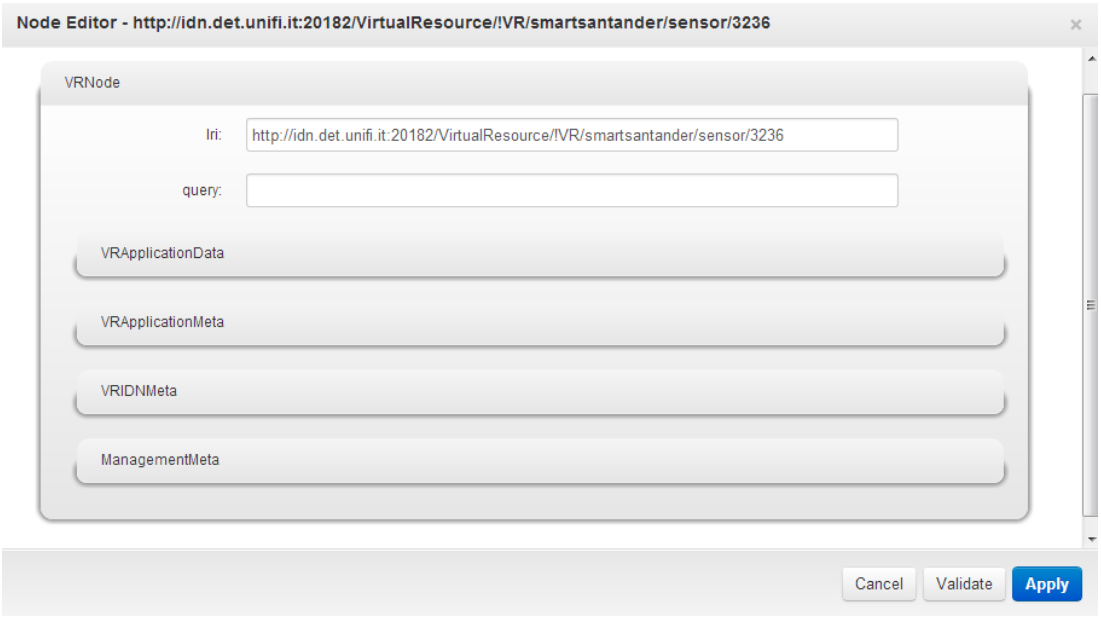


Figure 6.5. *The modal window for editing a node.*

6.4 IDN-Viewer

While IDN-Studio provides an easy way to put information together and manage documents, IDN-Viewer addresses the problem of their visualization. This is the last mosaic tile because is very likely that the ultimate purpose of a user creating a own document is to show it on the Web, i.e. support humans in intuitively accessing it. Also IDN-Studio provides a document human oriented representation, but is specifically thought for editing. Conversely, if a user puts together information concerning events and forecast for Chicago in a document, what he/she wants to see in the end is a Web page including this information. In addition, this information should appear in a user's Web page, not on a different location provided by InterDataNet.

This is why, IDN-Viewer has been conceived as a plugin for the popular JQuery [MDV04, DV06, BK08] Javascript framework. Many efforts have been put in trying to make it very easy to use, and also customizable. Moreover, the plugin is not tightly coupled with any specific CSS framework, allowing the user to adopt the preferred style. Fig. 6.6 shows the plugin in action, rendering the document depicted in Fig 6.7 in an HTML page.

Currently IDN-Viewer is able to choose the proper visualization for many different types of contents. For example structured document such as Json or XML are rendered as tables, images (jpeg, png, etc.) are rendered as clickable thumbnails, plain text is rendered as is. To perform this task, IDN-Viewer

inspects the `Content Type` section of the `Application Data` and selects the proper widget accordingly. Conversely, blank nodes, are not rendered because their unique purpose is to structure the document. As discussed in the following, this is not interesting for rendering human oriented document representations.

Figure 6.6. *The IDN-Viewer plugin the IDN-Document of Fig. 6.7*

A very important thing to point out is that IDN-Viewer does *not* strictly preserve the document hierarchy, while rendering. This means that, given a document, the way the elements are arranged in the space does not necessary follow the container-content relations defined by Aggregation Links. The reason for this is that, since aggregation is a basic relation, an average document is expected to have a significant number of contents nested in container nested in other containers in turn.

To constrain the document view to such a structure, it will end up in a extremely confusing representation, nullifying the *raison d'etre* of the project. Moreover, inspecting in this direction, it was agreed that the most human friendly representation preserving the graph structure is the one adopted by the IDN-Studio, whose intents does not overlap with the ones of the IDN-Viewer at all.

These considerations, motivated to focus the efforts in improving at most the readability of the document, trying to represent it as a browsable Web 2.0 page with contents, links and menus. Fig. 6.8 shows the rationale behind the displacement of elements on the Web page operated by the plugin. The main space, i.e. the foreground, is reserved for the root node contents. Below, are positioned the other nodes (retrieved following Aggregation Links), while on the right sidebar are located the menus and the gallery, if present. The menu is

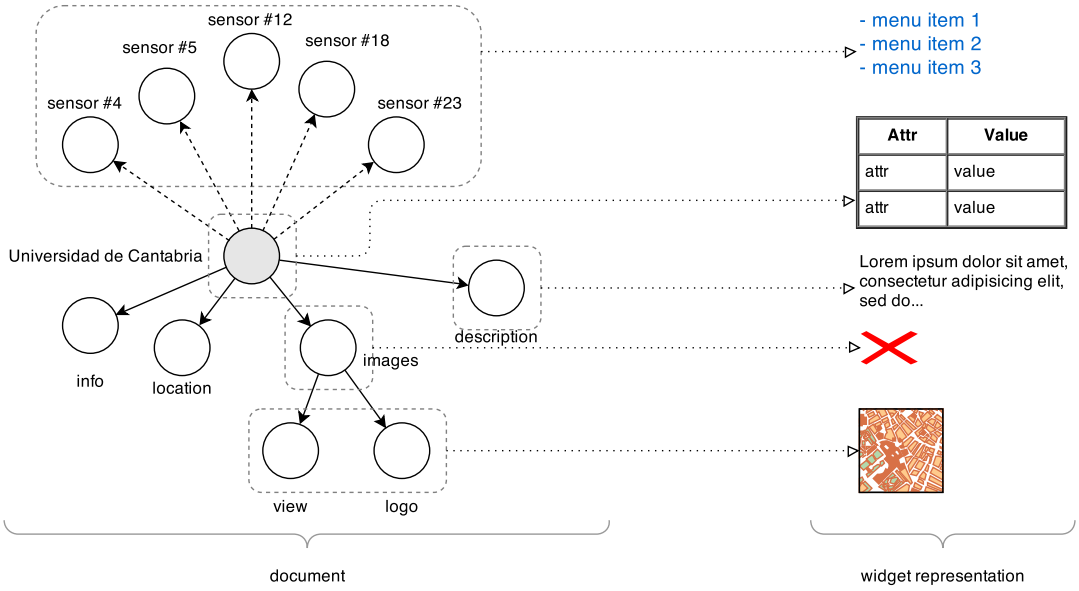


Figure 6.7. *The IDN-Document rendered in Fig. 6.6, with the corresponding widgets. As usual dashed edges represent Reference Links, while solid edges represent Aggregation Links.*

generated from the Reference Links contained in the document. Due to Reference Links semantics (see section 4.1.1), it seemed the perfect solution. If more nodes have Reference Links, the menu is organized in sections, each labeled with the `Local Name` of the node. Menu items are also estracted by the Reference Links `Local Name`s because they are likely to be more human friendly. If no `Local Name` is available, the complete URL is used instead. By interacting with the menu, is possible to browse the document space, because the referred document will be retrieved, and the Web page will be rendered again. To improve the user experience, the event associated with the browser "back" button is intercepted so that is possible to browse the document back and forth, as a usual Web page.

Sometimes IDN-Documents have a large number of nodes and is worthless to render them in a huge page. The problem has been tackled by introducing a "more" button at the basis of the Web page. When a LRI is dereferenced, a number of aggregated nodes are loaded. If the number is small enough to generate a readable page, the document is rendered at once, otherwise only a few number of elements are loaded a button for requesting more content is positioned at the bottom of the page. When the button is activated, the plugin performs an AJAX call to the InterDataNet architecture to retrieve the next set of nodes that will be attached at the bottom of the page.

In Listing 6.2 is shown an example of the IDN-Viewer usage. The user must prepare an HTML element with `id=''canvas''` where the plugin will render the document. Of course a `div` element is advised.

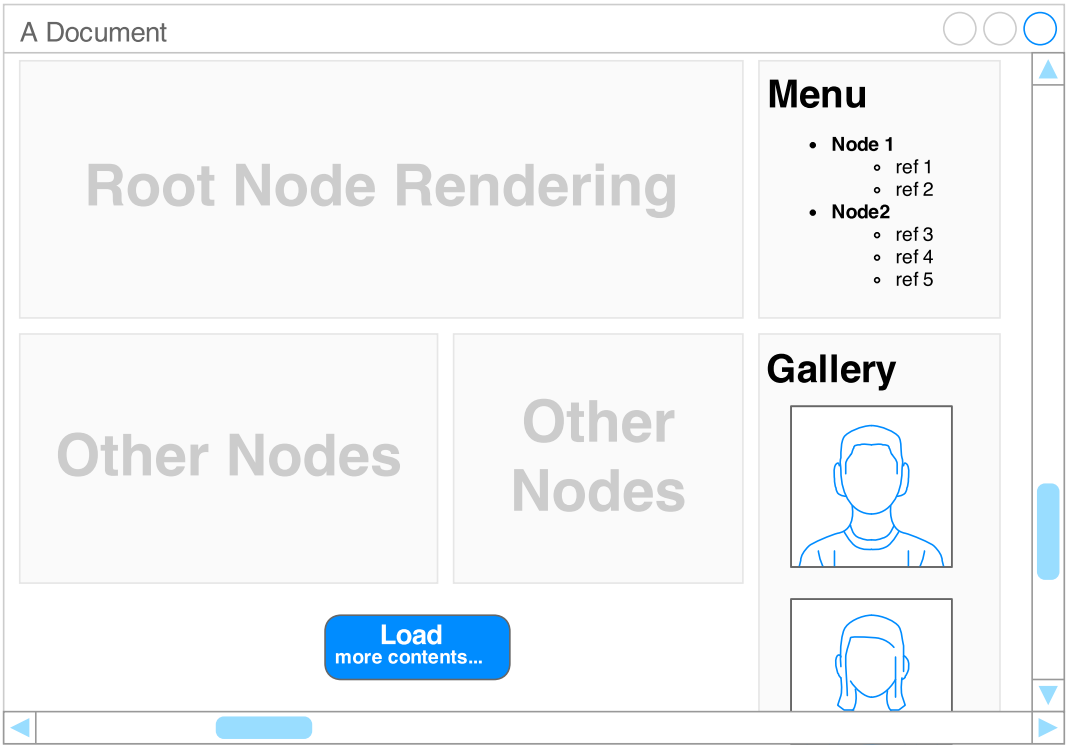


Figure 6.8. *The rationale behind the displacement of elements on the Web page.*

```
1  $("#canvas").idnviewer({
2      lri: "http://idn.det.unifi.it:20182/VirtualResource/!VR/smartsantander
       /sensormap/poi/1378557455142",
3      nodesWrapperClasses: "span9",
4      nodeWrapperClasses: "span4",
5      linksWrapperClasses: "span3",
6      loadMoreButtonClasses: "btn",
7      loadMoreButtonText: "load more",
8      resolutionDepth: 3,
9      resolutionDepthStep: 1
10     rendererssOptions: {...
11     },
12 })
```

Listing 6.2. An example of IDN-Viewer plugin usage.

The plugin options are listed in the following:

`lri` : the Logical resource Identifier of the document to be rendered

`nodesWrapperClasses` : parameters for the specification of CSS classes that will be applied to the elements containing the nodes list.

`linksWrapperClasses` : parameter for the specification of a CSS classes that will be applied to the element containing the Reference Links.

`loadMoreButtonClasses` : parameter for the specification of a CSS classes that will be applied to the "more" button.

`loadMoreButtonText` : parameter for the specification of the "more" button text.

`resolutionDepth` : initial resolution parameter for the IDN-Document. It determines how many nodes are loaded when the document is rendered for the first time.

`resolutionDepthStep` : determines how many nodes are loaded when the "more" button is pressed.

`rendererssOptions` : map indexed on the nodes' content types specifying additional options for renders.

Currently, the IDN-Viewer plugin has been tested for Mozilla Firefox and Google Chrome browsers.

Chapter 7

A Use Case: The SmartSantander Experiment

In this chapter is described the application of the InterDataNet framework to a real world Smart City scenario made available by the SmartSantander [SGG+11, GSG+12, GGS+13, HMM13, SMG+13] project. In this context, InterDataNet modeling and architecture have been used to represent sensor data provided by the SmartSantander Testbed Architecture as graph of individually addressable, composable information units, in the form of IDN-Documents. These sensor documents have been used to build other documents of different nature, such as virtual sensors or points of interests.

The whole experiment represents a validation for InterDataNet, in terms of adequacy of the model, effectiveness of the architecture, development support and ease of use of the Tool-suite.

The SmartSantander FP7 project aims at the deployment of a unique facility composed of Internet of Things (IoT)[Ash09, AIM10] nodes in the city of Santander as well as in Belgrade, Guildford and Lübeck. The main objective of the project is the creation of an infrastructure allowing experimentation on top of it, whilst concurrently supporting service provision related to the different operational domains of the city. On one side, the SmartSantander facility, made by thousands of sensors, calls for tools allowing users to use and share the information provided by the wide-scale sensing infrastructure. On the other side, it represents a unique opportunity for deploying and testing Internet of Things software prototypes developed within an academic environment in a challenging real-world scenario.

In [SMG+13], authors present the objectives of the SmartSantander infrastructure as follows: "the objectives of SmartSantanders deployed IoT infrastructure are two-fold as well as concurrent. As a testbed, it enables experimental assessment of cutting-edge scientific research [...]. It also aims at supporting the assessment of the socio-economical acceptance of new IoT solutions and the quantification of service usability and performance with end users in the loop.

For instance, it simultaneously supports the trial and subsequent provisioning of smart city services. To attract the widest interest and demonstrate the usefulness of the SmartSantander platform, the deployment of the IoT experimentation infrastructure has been undertaken to realize the most interesting and impact-generation use cases.

In this respect, application areas have been selected based on their high potential impact on the citizens, thus enabling the execution of extensive experiments to obtain insights into the uptake of IoT-based services deployed in a live environment. Also taken into consideration in the selection of application use cases are the diversity, dynamics and scale of the IoT environment. All these aspects increase the potential of the testbed for the evaluation of advanced protocol solutions."

Fig. 7.1 shows an overview of the SmartSantander high-level architecture for the experimental facility.

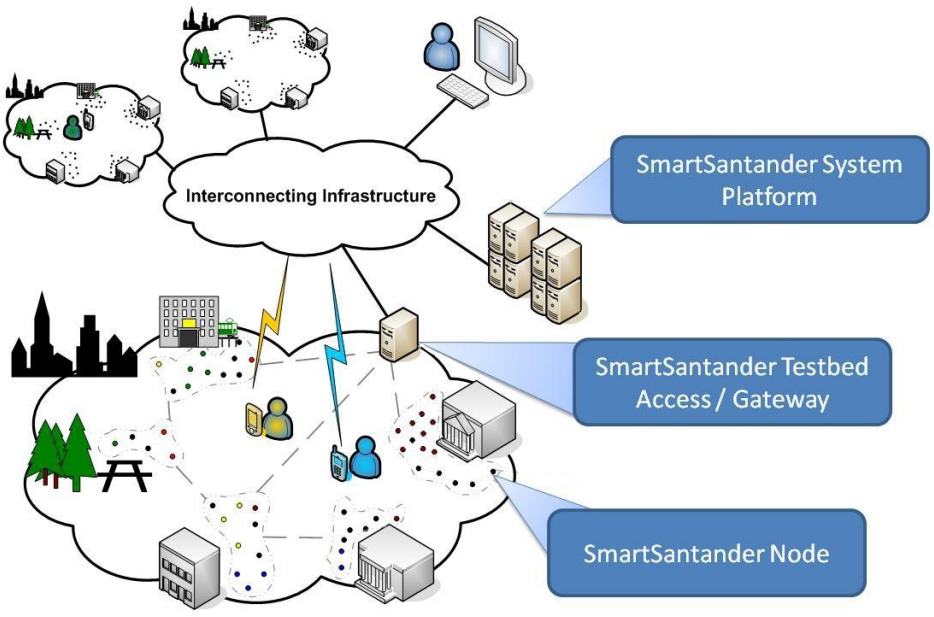


Figure 7.1. *The overview of the SmartSantander high-level architecture for the experimental facility.*

7.1 Testbed Architecture

In this section is addressed the SmartSantander Testbed Architecture in the form of excerpts from [SGG+13].

SmartSantander is a three tiered network architecture, made up of an IoT device tier, a gateway tier and a server tier.

The IoT node tier provides the experimentation substrate consisting of IoT devices. As IoT experimentation is the primary focus for SmartSantander, the IoT node tier accounts for the majority of the devices utilized in the testbed infrastructure. In order to satisfy the expected heterogeneity the IoT node tier consists of a variety of IoT device types, such as diverse mote platforms, RFID readers and tags as well as more powerful platforms such as mobile phones with short range communication capabilities.

The gateway node tier links the IoT devices at the edges of the network to a core network infrastructure. The nodes of the gateway tier are also part of the programmable experimentation substrate, in order to allow experimentation for different interworking and integration solutions of loT devices with the network elements of a current or Future Internet.

The server tier provides more powerful server devices, with high availability, which are directly connected to the core network infrastructure.

Within the scope of the project, a variety of sensing devices have been installed. A comprehensive list is given in the following:

Fixed sensor nodes have been placed attached to public lampposts or to building facades. Apart from the typical low-capacity microcontroller and multimodal sensing unit, a distinguishing feature of these devices is that they have been equipped with two radio transceivers operating at 2.4 GHz frequency. One of the modules implements IEEE 802.15.4 protocol in a native way, and the other one runs IEEE 802.15.4 protocol modified with the proprietary routing protocol Digimesh.

This way is possible to support that these nodes concurrently supports experimentation and service provision as both functionalities are physically separated on each of the networks created.

Multimodal sensing units includes a wide range of observed physical magnitudes: light intensity, noise, carbon monoxide, air temperature, relative humidity, solar radiation, atmospheric pressure, soil moisture, soil temperature, wind direction, wind speed and rainfall. It goes without saying that not all nodes are able to provide information about all these magnitudes but a subset of them on each case. In total more than 2,000 sensing points are available through this kind of nodes.

Mobile sensor nodes are equivalent to the fixed ones in terms of micro-controller and radio transceivers, these nodes are installed on top of public transport buses, taxis and other municipal services vehicles.

These nodes are also equipped with GPS and a GPRS interface so that they can geo-position each of their observations and be permanently reachable respectively. The sensing capabilities of these nodes are focused on air quality monitoring: nitrogen dioxide, ozone, particles in the air, air temperature, carbon monoxide, relative humidity.

GPS also allows them to report speed and course of the vehicle. Some of the nodes installed on public buses have been connected to the CAN-bus so they can monitor a plethora of parameters related to the vehicle status and behavior. 150 vehicles are equipped with this kind of nodes.

Parking monitoring sensor nodes are buried under the asphalt on outdoor parking places at Santander downtown area. A total of 350 parking lots are monitored with this kind of nodes.

Traffic monitoring sensor nodes are buried under the asphalt on the main entrances to the city of Santander. They are monitoring the number of vehicles, the level of occupancy of the lanes, the average speed of vehicles and the average queue length both inward and outward. 40 sensor nodes of this kind have been deployed.

QR and NFC tags are located at city Points of Interest (POI) (e.g. monuments, bus stops, local administration premises, shops, etc.) these tags include both a QR code and a Near Field Communication (NFC) tag. They are a key asset for the Augmented Reality service as they provide information related to the POI. 2,500 of these tags have been placed all around the city.

Embedded PCs act as gateways for the fixed sensor nodes. These nodes are arranged in multihop meshed clusters that depend on one of these Gateways. They are connected to the IoT Nodes through the IEEE 802.15.4 Digimesh interface and to the testbed server tier through WAN interface (e.g. Ethernet, 3G, optical fiber, etc.).

7.2 The InterDataNet Experiment

The InterDataNet architecture experiment aims at contributing to the SmartSantander project objectives by addressing the need for the efficient management and processing of IoT generated data, in order to allow an easy integration of these IoT endpoints into the service layer of the Internet.

The main objective of this project action consists in extending the SmartSantander testbed facilities with the capabilities provided by the InterDataNet

middleware and information model, more precisely, the objectives of the experiment consist in:

- exposing and handling a subset of SmartSantander sensors as a Web of Resources. These Resources will represent sensors descriptive information, functional capabilities (almost static information) and measurements (dynamic information) as well.
- allowing users (i.e., web application developers) to browse this Web of Resources and create new Resources by assembling parts of existing resources and/or adding new sensors and related information. This includes the capability of adding "Virtual Sensors" that can be placed in locations of interest and whose measurements can be derived from the physical sensors outputs (e.g., sensors deployed in the testbeds), according to proper analytical models and/or from citizens direct observation. Analogously, new resources can be added to represent new sensors owned by private citizens (e.g., private weather stations).
- this Web of Resources can be browsed, queried and modified through a Graphical User Interface that allows users to add novel resources.

The experiment will be focused on exploiting the above-mentioned REST APIs and the InterDataNet framework and graphical tools for:

1. creating a set of virtual sensors calculating the apparent temperature in some locations of interest, such as schools, and parks;
2. allow citizens to view physical and virtual sensors and the apparent temperature derived from temperature and humidity values at the points of interest on the map;
3. allow users to browse a web of heterogeneous sensors for a given point of interest.

In the following, the first case study (case study A) complies mainly with points 1. and 2. and is focused on the creation of Virtual Sensors for quality of life monitoring; the second case study (case study B) satisfies mainly point 3. and is focused on the implementation of a manager for points of interest.

7.2.1 Case Study A: Creation of Virtual Sensors for Quality of Life Monitoring

This Case Study aims at demonstrating the usefulness of the Virtual Sensor (VS) paradigm provided by the InterDataNet infrastructure and how it can be exploited to define new value-added services.

Virtual Sensors can be applied to improve the sensors coverage by increasing their geographical resolution or adding new sensor types, i.e. software-based sensors whose output is computed from values acquired through physical sensors (called "feeding sensors"). To this end, a full Web-oriented paradigm is applied to the way sensors are modeled, exposed and manipulated at a granular level by exploiting InterDataNet middleware and information capabilities on top of the SmartSantander platform.

Thanks to the uniform interface and shared semantics of the RESTful APIs, SmartSantander Physical and Virtual Sensors can thus be browsed and navigated as a Web of Resources and new resources can be created and reused by assembling parts of existing resources and/or adding new sensors information. Moreover, the experiment can also leverage the indexing and searching capability implemented by the InterDataNet middleware.

In the reference scenario, these Virtual Sensors can thus be created in order to monitor the behavior of some environmental parameters that can influence the quality of life. For instance, the apparent temperature (or heat index) is a parameter derived from temperature and humidity and represents a significant indicator for measuring the quality of life in open-air areas, especially during summer days.

Indeed, the apparent temperature is a measure of how a human body perceives the air temperature and it can be associated to a level of discomfort. According to the values of the apparent temperature, municipal authorities (i.e. civil protection) can decide to send alerts to the population and suggest proper actions, especially for the care of children and elderly people.

Moreover, further parameters (e.g., ozone and particulate) could be taken into account to provide a more accurate estimation of the quality of life in specific areas of Santander.

Thus, this case study is about the creation and geographical displacement of Virtual Sensors capable of measuring the apparent temperature in strategic locations like schools, public gardens and hospitals. The aim is to leverage the environmental sensors installed in Santander to build a network of Virtual Sensors capable of monitoring the apparent temperature level in the territory.

First it will be crucial to implement the IDN-Document representation for the actual sensor, and secondly the apparent temperature Virtual Sensor (AT-VS) will be designed and implemented. The AT-VS is a Virtual Sensor able to measure the Apparent Temperature in a given location, depending on humidity and temperature values measured by physical sensors in that location. If no proper physical sensor is available in the locations of interests, is possible to create temperature and humidity Virtual Sensors that, on their turn, derive their values from the closest temperature and humidity physical sensors (for instance, with a simple model based on a weighted average operation).

The calculation of the apparent temperature value will be based on the ap-

proximate formula provided by the U.S. National Oceanic and Atmospheric Administration (NOAA):

$$HI = c_1 + c_2T + c_3R + c_4TR + c_5T^2 + c_6R^2 + c_7T^2R + c_8TR^2 + c_9R^2T^2$$

where HI stands for Heat Index, T for Temperature, R for Relative Humidity and $c_1, ..., c_9$ are coefficients provided by the US NOAA [RH90]. New processing models can be added to accomplish more accurate results.

The accuracy of the AT Virtual Sensor output depends on three main factors:

1. the accuracy of inputs;
2. the accuracy of the chosen models for deriving the apparent temperature and for improving the spatial resolution of temperature and humidity sensing;
3. the geographical distance of the feeding sensors from the virtual sensor.

The development of an application for the easy exploitation of data produced by sensors is also envisaged. The application displays a map of the city of Santander with the sensors available for querying. Citizens can interact with the application in an intuitive way, and they learn how to use these data following a guided procedure. The aim is not only to build a rich logical infrastructure of consumable data, but also to promote the citizen participation in producing value for the benefit of the community.

When the application starts, a map of the Santander city is shown; both virtual and actual sensors within the city boundaries are displayed. By clicking on a sensor, the user will access a pop-up window showing information about the sensor, such as its identifier and temperature (real or perceived).

The user can also create new virtual sensors; for example, the user may be interested in setting up a sensor located in his garden, which measures the apparent temperature relying on outputs of the closest sensors. A guided procedure defines the following steps for the creation of the virtual sensor:

1. select the type of sensor;
2. select the analytical model (for example the previously mentioned model, or a simple mean);
3. select the location;
4. select the nearby feeding sensors to be used.

As he confirms the choices, the sensor is created and stored in a global sensors space, where it can be inspected by administrator users.

The user can also modify a virtual sensor, by changing the feeding sensors or delete a virtual sensor he created.

7.2.2 Case Study B: Managing Points of Interest

Case Study B aims at demonstrating how easy is to leverage the Information Model supported by InterDataNet to model the resources representing virtual and physical sensors and exploit them to build a web application.

The target user, i.e. the web application developer, is willing to develop a web application showing information on a given Point of Interest (PoI), harvested by near physical and virtual sensors.

The user performs a search for sensors according to the desired parameters and the InterDataNet Search Engine network returns a list of matching URIs.

The IDN-Studio application (see section 6.3) allows the developer to create a new IDN-Document by modeling the Point of Interest and related sensor-based information. Through the IDN-Studio application, the developer can retrieve an IDN-Document from the system which is presented as an editable graph of information. After a save command, the graph is submitted to InterDataNet for persistence. Moreover, the user can also create a new document from scratch, design its structure and content and submit it to the system.

Thus, IDN-Studio implements an abstraction layer between the user and the InterDataNet architecture. Every user action, such as a creation or an editing is mapped to a RESTful communication handled by InterDataNet, and the system responses are mapped into human readable feedbacks. Finally, the web developer can focus on presentation issues leveraging the IDN.js library or taking advantage of the IDN-Viewer capabilities (see sections 6.2 and 6.4).

7.2.3 Experiment Architecture

This Section describes the architecture of the experiment in terms of the main functional components, the sensor information model and, finally, the granular REST APIs that will be available on top of the InterDataNet middleware.

First, three main definitions are given:

Sensing Element is an object able to output a measurement of a physical quantity, which may require a set of inputs.

Web Sensor (WS) is a graph representation of a Sensing Element, according to the information model shown in section 7.2.4. Each and every vertex of the graph is addressable via a HTTP URI and can be dereferenced on the Web.

Virtual Sensor (VS) is a Web Sensor which derives its output(s) from a number of Web Sensors outputs processed through an analytical model.

A primary objective of this project is to expose some sensors deployed in the city of Santander as Web Sensors. To accomplish this task, a two steps procedure is applied: 1) gain access to the SmartSantander sensors data and 2) process these

data to turn them into a set of Web Sensors IDN-Documents.

The SmartSantander APIs accessible for the experiment activities expose the set of URIs listed in the following[1]

1. `http://.../GetNodes`
2. `http://.../GetLastValuesbyNodeID/{id}`
3. `http://.../GetEnvMonitoringLastValues`

All these URIs return a Json response. More precisely, URI 1 provides information about the fixed sensors, including the position (longitude, latitude), the sensor type and the identifier; URI 2 provides (where available) sensor identifier, battery level, output measurement, date, gateway location and date for the sensor associated to the URI template variable `id`; URI 3 provides last environmental measurements collected from the IoT fixed infrastructure.

These APIs govern the interaction of the experiment software with SmartSantander. Within the aforementioned URI list, URI 3 is used as a hook towards the SmartSantander sensor data and its invocation is a precondition for the creation of the WS IDN-Documents modeled according to the information model presented in section 7.2.4.

In order to integrate data from SmartSantander into InterDataNet, an *ad hoc* Adpater (see section 5.4), called SS-Adapter, has been developed. The SS-Adapter performs an initialization of the resources from the SmartSantander system, according to the procedure described in section 5.4.2. Such resources fall in the group defined as countable, therefore no shell document is used. When the initialization is completed, data coming from SmartSanatander are represented as IDN-Documents exposed through the InterDataNet APIs.

Few additional constraints are needed to enable the Virtual Sensor representation. According to the definition, a Virtual Sensor is a Web Sensor, and so all the mechanisms supporting the creation of a Web Sensor are valid for the Virtual Sensor as well. Basically, the difference between a Virtual Sensor and a Web Sensor lies in the output generation. An ordinary Web Sensor output comes directly from the SmartSantander APIs, while a Virtual Sensor output is the result of a computation with a number of variables.

At the time of the experiment, the Activity Node described in section 4.1.3 was in an early stage of design and it was not possible to apply this concept to the use case. Therefore, a dedicated module, detached from the architecture, has been used instead: VS-Application.

[1]The authority and full path of URIs are omitted for confidentiality issues.

VS-Application is the module dedicated to the Virtual Sensors' output computation. It is provided with the chosen analytical model, and the outputs of the feeding Web Sensors. The Virtual Sensor IDN-Document is used to store (and therefore retrieve) this information. In addition, since VS-Application consumes Virtual Sensor data directly from the IDN-Document and does not rely on a local data-source, the system preserves its consistency at no costs.

From the perspective of the InterDataNet architecture, VS-Application is an ordinary external data source which exposes its resources via a Resource Oriented Approach (ROA) approach. To reconcile the custom resources from VS-Application with InterDataNet, a proper adapter called VS-Adapter is used. Since the resources exposed by VS-Application can be considered countable, the VS-Adapter is of the same type of the SS-Adapter. However, no initialization phase is performed because the creation of a virtual sensor is triggered by a command coming from the outside. This is a special case for adapters, mainly due to the unavailability of Activity Node, which is a much more appropriate and elegant solution. Fig. 7.2 shows the logical view of the system architecture.

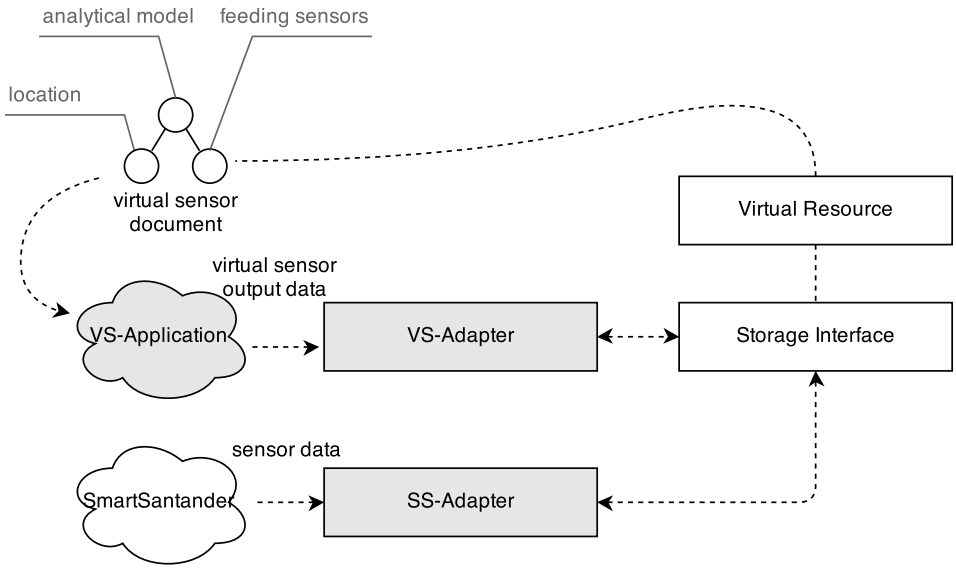


Figure 7.2. *Logical view of the system architecture, from the Adapter perspective.*

7.2.4 Resources Modeling

To expose data in a fruitful way is important to leverage the Web approach, i.e., to put data in a global space, using standard technologies like HTTP protocol and URIs. In addition, it would be very convenient to have data organized in versatile and expressive information structures, e.g. graphs.

While dealing with sensors, often the focus is on the output data. However, it would be useful to model the sensor itself, so developers can get information about the device that generated data they are using.

7.2.4.1 Modeling the Web Sensor

The Web Sensor information model is depicted in Fig. 7.3. It is a graph (more specifically, a tree) where the vertexes identify the information resources and the small dots identify their attributes. The model is valid for both physical and virtual sensors, i.e. Web Sensors.

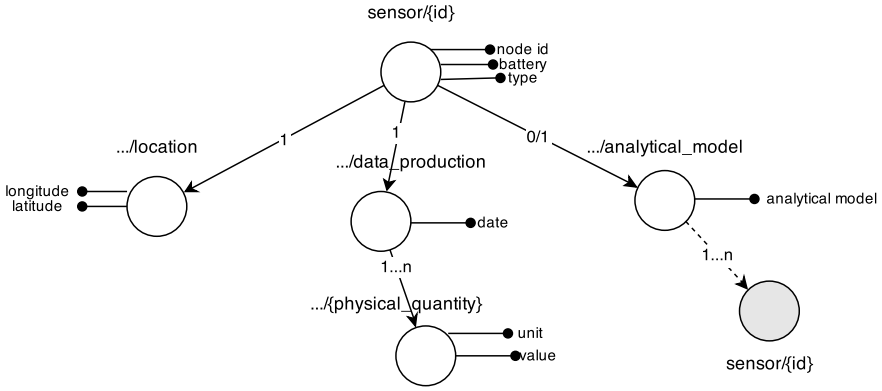


Figure 7.3. *The Web Sensor information model.*

Browsing this sensor representation, is possible to reach a number of resources, such as the measured data, the location, and the analytical model for the Virtual Sensor output. The `sensor/{id}` labeled vertex has the following attributes: identifier, sensor type (e.g., temperature sensor) and battery percentage. This vertex has two children (`location` and `data_production`). The `location` node is used to identify the sensors geographical position, while the sub-tree `data_production` represents the observations associated with the sensor. In particular, the `data_production` vertex has as an attribute representing the last date and time measure, and its children nodes, identified with the generic term {`physical_quantity`} (i.e., temperature, light, etc.), provide information about the value of the associated measure. To preserve the generality of the model, a `data_production` vertex can have more than one child. In other words, if the sensor is equipped with the proper hardware, it can provide different measures simultaneously.

The sub-tree starting from the `analytical_model` vertex represents the virtual sensor specification. The only attribute here is the analytical model with the criteria to provide the output of the virtual sensor. This vertex can have an

arbitrary number of children representing the sensors that feed the VS (in the figure the grey node represent another Web Sensor document). They are included by reference using the Reference Link notation. Table 7.1 shows the REST APIs exposed by the architecture for the Web Sensors.

	GET	**PUT**	**DELETE**
sensor/{id}	physical sensors summary data reading (id, type, and battery).	document creation and modification	physical sensor's summary data deletion
virtual/sensor/{id}	virtual sensors summary data reading	document creation and modification	virtual sensor's location deletion
../location	sensors geographical data reading	document creation and modification	sensor's location deletion
../data_production	sensors temporal output data reading	document creation and modification	sensor's temporal data deletion
../{physical_quantity}	sensors outputs reading	document creation and modification	sensor's physical quantity deletion
../analytical_model	sensors analytical model data reading (for Virtual Sensors only)	document creation and modification (including feeding sensors management for Virtual Sensor output evaluation)	sensor's analytical model deletion
sensor/{id}/	sensors data reading (document starting from `sensor/{id}` vertex)	none	none
../data_production/	sensors data production reading (IDN-Document starting from `../data_production` vertex)	none	none

Table 7.1. the REST APIs exposed by InterDataNet for the Web Sensors.

7.2.4.2 Modelling the Point of Interest

The Point of Interest (PoI) model is depicted in Fig. 7.4, and Table 7.2 shows the REST Apis exposed by InterDataNet for it. As in the Web Sensor case, the vertexes represent the information resources while the small dots are attributes of a resource. A Point of Interest is modeled with the IDN-Document Information Model formalism, as a root node with two attributes: a unique identifier and a string representing the name.

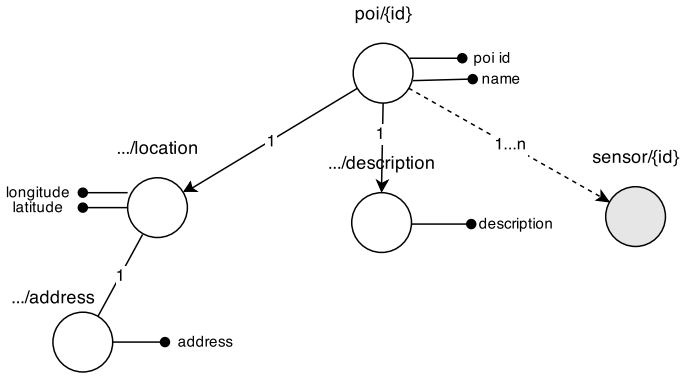


Figure 7.4. *The Point of Interest information model.*

Two children vertexes representing the `location` and the `description` are required for a PoI definition. The former is used to identify the sensors geographical position by longitude and latitude, while the latter represents a generic description (e.g., textual description, photo links, etc.). The `location` node has a child (`address` node) that represent the address expressed as a collection of information used for describing the location.

Finally, is possible to enrich the Point of Interest description by associating it with sensors. This is obtained by adding Reference Links to the root vertex. Each Reference Link matches to sensors information provided by the IDN-Document shown in Fig. 7.3. Table 7.2 shows the REST Apis exposed by InterDataNet for the Points of Interest.

	GET	**PUT**	**DELETE**
PoI/{id}	PoIs summary data reading.	document creation and modification.	PoI's summary data deletion
../location	PoIs geographical data reading	document creation and modification	PoI's location deletion
../address	read operation describing the PoIs address	document creation and modification	PoI's address deletion
../description	obtaining textual description, photo links, etc. related to the PoI	document creation and modification	PoI's description deletion
PoI/{id}/	PoIs data read operation (document starting from `PoI/{id}` vertex)	none	none
../location/	PoIs geographical data read operation (document starting from `../location` vertex)	none	none

Table 7.2. the REST APIs exposed by InterDataNet for the Points of Interest.

7.2.5 MySmartCity Application

MySmartCity is a Web Application designed and implemented to meet the requirements of Case Studies A and B. MySmartCity leverages the fine-grained REST APIS provided by the InterDataNet middleware and the Java and JavaScript libraries, as well as the IDN-Viewer JQuery plugin.

It is composed by a server module and a client module, both interacting with the Virtual Resource layer, to exchange Web Sensor and Point of Interest IDN-Documents. Fig. 7.5 shows the position of MySmartCity application with respect to the whole InterDataNet experiment architecture and tools.

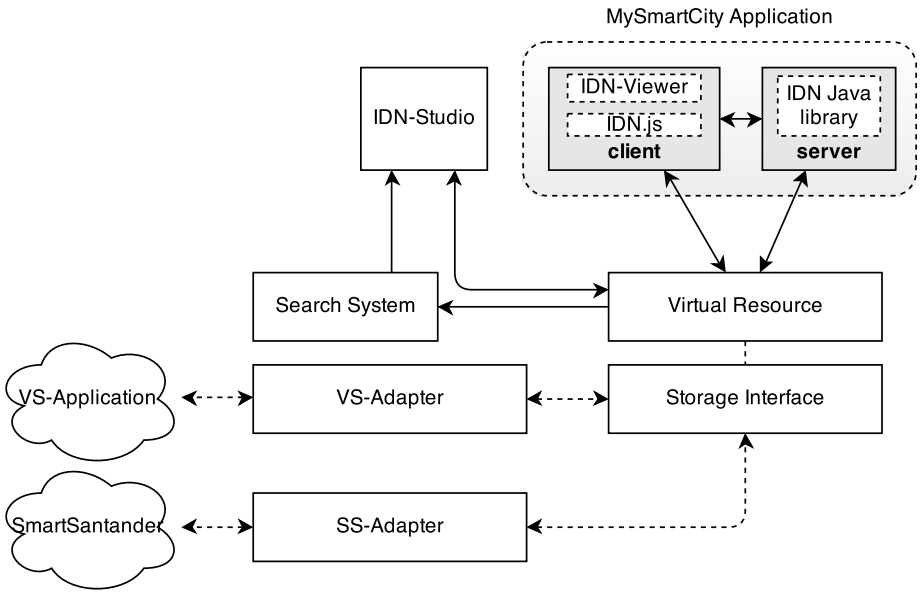


Figure 7.5. *The position of MySmartCity application with respect to the whole InterDataNet experiment architecture and tools.*

The objective of the case study A is to show the usefulness of the InterDataNet information model in the creation and handling of virtual sensors.

MySmartCity application enables the management of virtual sensors, on the basis of the sensors actually installed in Santander; at the startup, it shows a map of the city of Santander where different sensors (whether real or virtual) are located; each sensor is recognizable by its own marker (blue for actual sensors and green for virtual ones). These sensors are retrieved by the InterDataNet in the form of IDN-Documents.

The application supports two different account types: user and administrator. To this end, MySmartCity has dedicated login and registration pages. Each end user have access to the following resources: real sensors; public virtual sensors; his own virtual sensors; all existing points of interest. The administrator can access and manage physical and virtual sensors and points of interest.

In addition, the GUI allows the user to filter the sensors by type (i.e. physical ones or virtual ones) and by measurement type (e.g. temperature, humidity, noise). In order to create a new virtual sensor, an authenticated user can leverage a wizard that guides him with the procedure. When a user selects a sensor on the map, an information balloon will be shown. If the sensor is a virtual one and is owned by the current user, two commands will be shown, Edit and Delete that allow the user to modify or delete that virtual sensor.

Each end user has his/her own personal space for the virtual sensors management. The administrator can see all the virtual sensors created by users and he/she is also able to modify a sensor or make it public. Fig. 7.6 shows the creation wizard for a virtual sensor, shown on MySmartCity graphical interface.

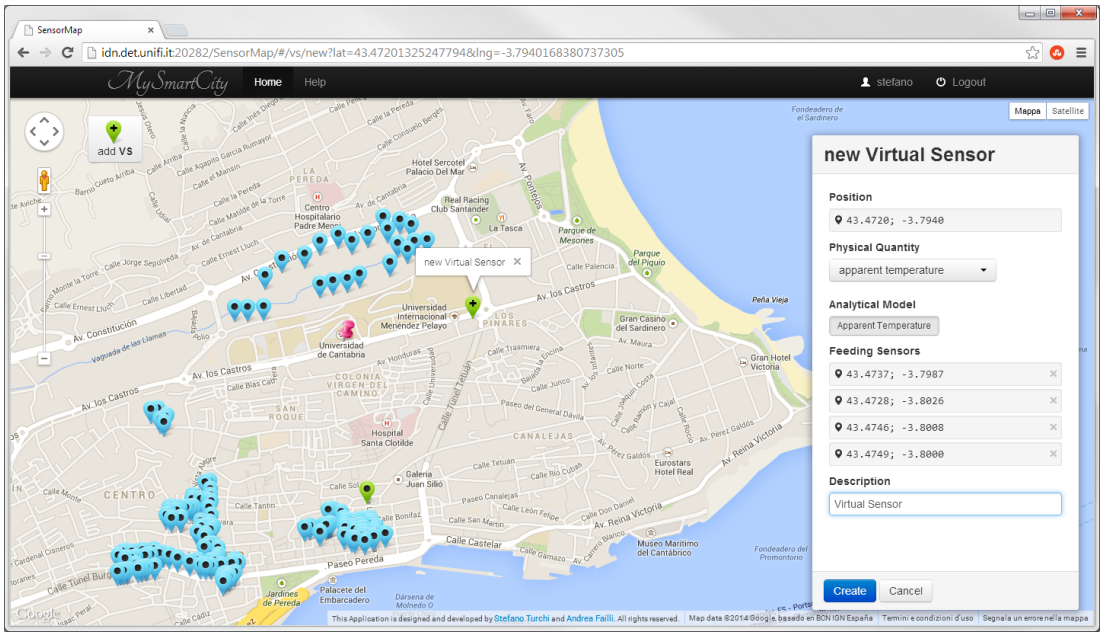


Figure 7.6. *The wizard for the creation of a virtual sensor, on MySmartCity GUI.*

Fig. 7.7 shows the server-side class diagram of MySmartCity. The SensorController class is an implementation of the Spring Controller interface, which provides various methods to manage sensors, such as getSensors() to retrieve the set of sensors actually installed in Santander and virtual sensors that are publicly visible, postVirtualSensor() or deleteVirtualSensor(), respectively to create or delete a virtual sensor. The `SensorManager` performs the role of the main class in managing sensors, and provides utility methods for their retrieval, creation, and modification. The `ModelManager` class produces and manages the analytical model adopted to calculate the output value of a virtual sensor, on the basis of the feeding sensors output values. The `SchedulerService` verifies at regular intervals if the sensors set is changed, by using a `SensorSearcher` service and

eventually build an optimization cache. The `AdapterClient` class, interacts with the Adapter module, while the `IDNClient` implements a client towards the InterDataNet middleware, in order to manage IDN-Documents representing Physical and Virtual Sensors. Finally, the `UserManager` class manages user registration and user login, by the means of the services provided by the `CredentialsManager` class.

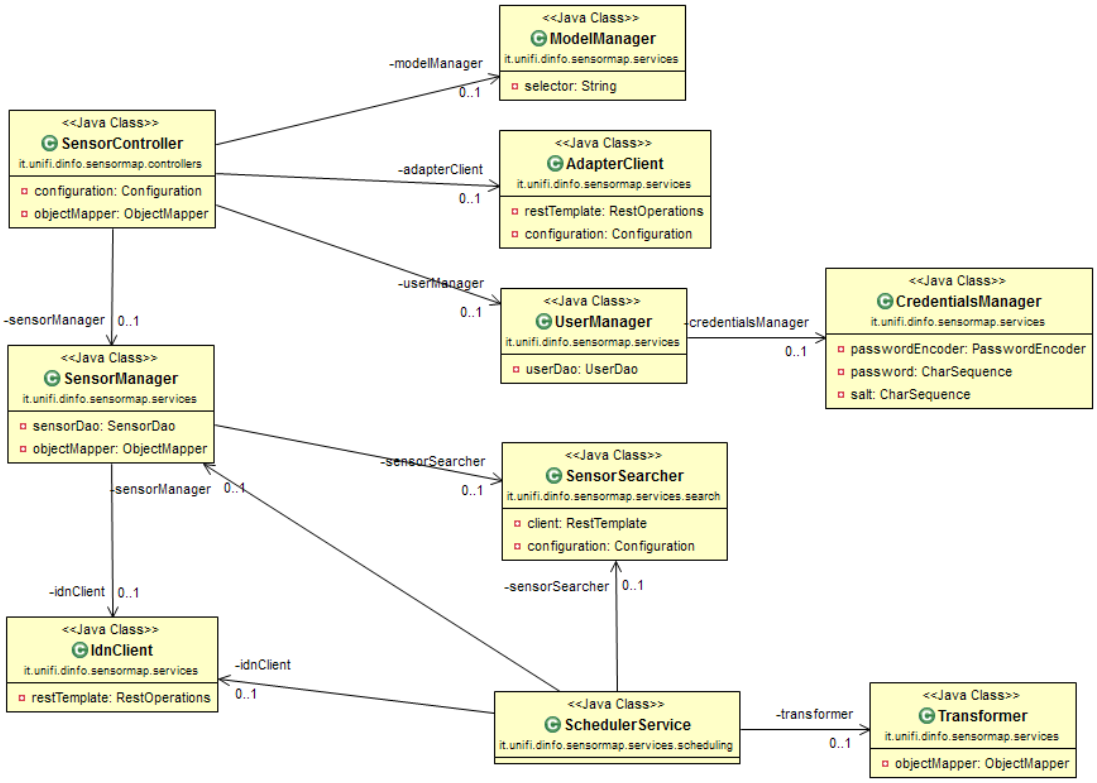


Figure 7.7. *The server-side class diagram of MySmartCity application.*

The objective of this case study is to show how the InterDataNet information model and the related REST APIs can be used to develop web applications allowing end users to browse and handle sensors related information. To this end has been studied the case of a developer willing to implement a web application showing the points of interest of Santander, taking advantage of Web Sensors represented as IDN-Documents.

A Point of Interest (PoI) is a digital representation of a symbolic location (e.g. a dome, park, stadium). It can be characterized by some descriptive information (e.g., tourism information, history, etc.) and its representation can be enhanced with up-to-date measurements of environmental parameters gathered by the Santander facility (e.g. temperature, light, noise). In section 7.2.4.2 is described

the InterDataNet information model for a PoI resource.

MySmartCity application provides the administrator with services for the management of Point of Interest resources; thus only the administrator is allowed to create, edit and delete Points of Interest. Once created, Points of Interest are made publicly accessible. Analogously to the sensors case, when a user clicks on a PoI marker, an information balloon appears, as shown in Fig. 7.8. More precisely, it contains the name and a brief description of the PoI as long as a list of (real or virtual) sensors that are referred to in the PoI model. For each sensor in the list, the available physical quantities and the distance from the PoI are displayed. At the bottom of the balloon there is a view more link that opens a comprehensive information sheet of the PoI. The information sheets are rendered leveraging the IDN-Viewer jQuery plugin.

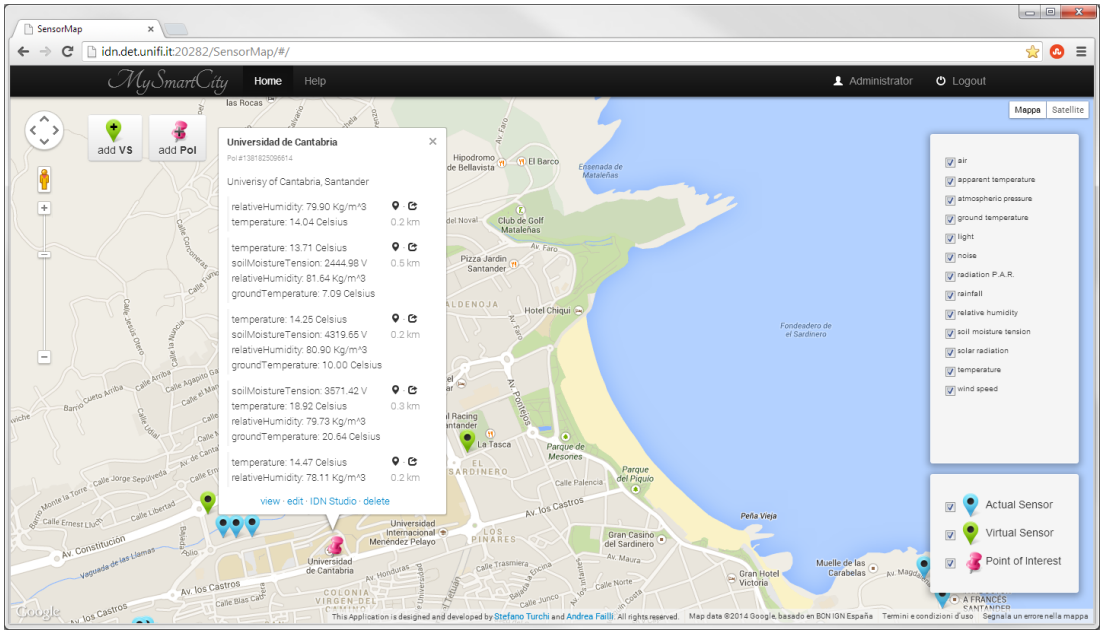


Figure 7.8. *Information related to a PoI shwon on a balloon, in MySmartCity GUI.*

In the following, the main operation involving Web Sensors and Points of Interest are detailed.

7.2.5.1 Retrieving Sensors Information

Not to encumber the reading, in the followings the workflows will be exposed only in the parts concerning the management of sensors. In order to retrieve the sensors information, MySmartCity Application performs a request client-side towards the method `getSensors` provided by the class `SensorController`. This class handles this request by invoking the `getAllPublic` method exposed by the

`SensorManager` class. Finally the `SensorManager` interrogates the persistence layer which manages a cache of the fixed sensors deployed in Santander and their locations. This trick allows to speed up the display operation of the sensors on the map and is based on the assumption that this set of sensors changes infrequently. However, a scheduler implemented by the `SchedulerService` class, checks for changes in the sensors' set at regular intervals of time. The `SensorManager` provides the `SensorController` with the list of all public sensors (i.e. the actual sensors actually installed in Santander and the virtual sensors of public domain); this list is passed client-side from the latter to the application, which shows all the public sensors on the map. Fig 7.9 shows the sequence diagram for sensors visualization.

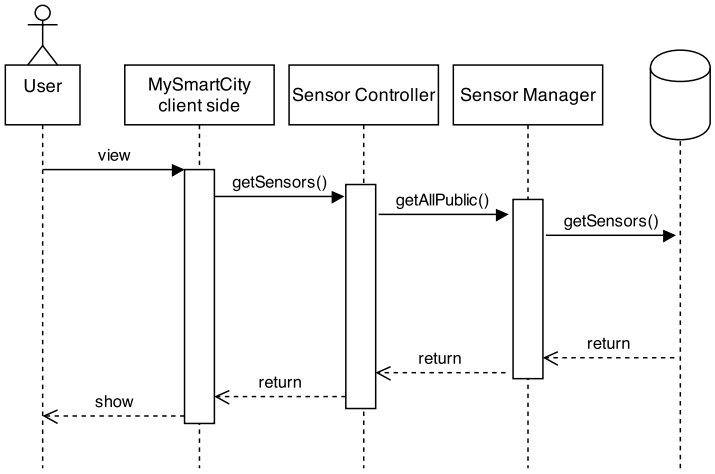


Figure 7.9. *The sequence diagram for sensors visualization.*

When a user clicks on a sensor, either real or virtual, the application performs an HTTP request via AJAX towards the InterDataNet middleware to get sensors descriptive information and last measurements. In this case, there are two different scenarios, depending on the nature of the sensor. If the sensor is an actual one the HTTP request is:

```
GET http://authority/sensor/{id}/
```

and this request is forwarded by the InterDataNet middleware to the SS-Adapter layer, which requests sensor data to the SmartSantander interface.

If the sensor is a virtual one, the HTTP request is:

```
GET http://authority/virtual/sensor/{id}/
```

and this request is forwarded by the InterDataNet middleware to the VS-Adapter layer, which calls VS-Application for the required computation, according to the specified analytical model. Then, the result is returned to the Adapter module, from which is returned to the middleware and finally to MySmartCity, together with the other information specified in the resource representation.

In both cases, the result is returned by the Adapter module (either SS Adapter or VS Adapter) to the middleware and from this to the application; the IDN-Document is processed client-side and the information of interest is captured by using the IDN.js library. Finally, MySmartCity application shows the information in an info balloon.

7.2.5.2 Writing a Virtual Sensor

As the user creates a new virtual sensor, sensor creation request is submitted to MySmartCity server-side. At this stage the `postSensor` of the `SensorController` class is invoked. This class handles the request by calling the `write` method of the `AdapterClient` class, which, on his turn, queries VS-Adapter through an HTTP POST.

As a result of this request, VS-Adapter sends an HTTP PUT request to the InterDataNet middleware

```
PUT http://authority/virtual/sensor/{id}
```

to create the document representing the virtual sensor, and a request to VS-Application to configure it as needed. Fig 7.10 shows the sequence diagram for a virtual sensor creation.

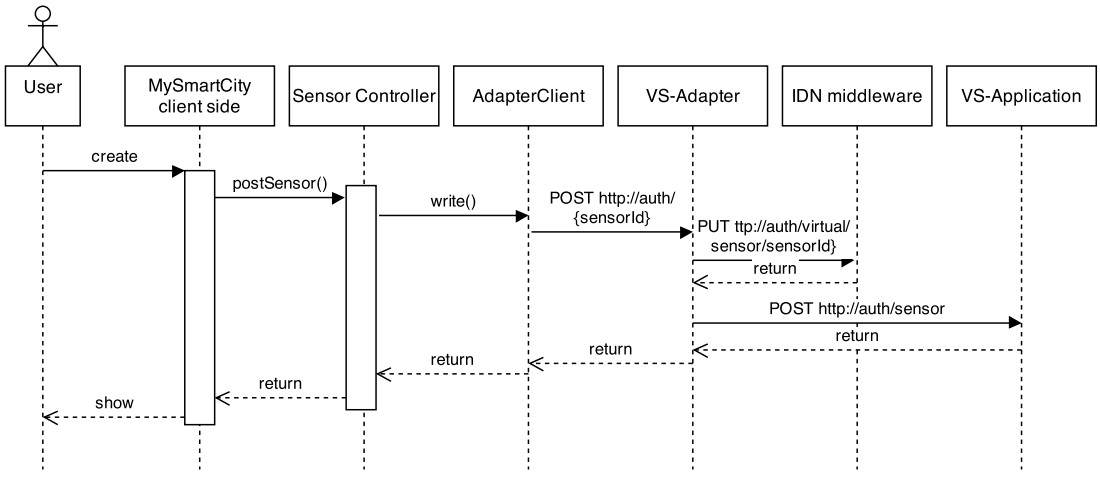


Figure 7.10. *The sequence diagram for a virtual sensor creation.*

7.2.5.3 Deleting a Virtual Sensor

When a user asks to delete an existing virtual sensor, MySmartCity performs a client-side request to the `deleteVS` method of the `SensorController` class. The invocation is handled by calling the `deleteVS` method of the `SensorManager` class; as a result of this, a number of HTTP DELETE requests are sent to the InterDataNet middleware to remove all the document vertexes:

```
DELETE http://authority/virtual/sensor/{sensorId}
```

Once the deletion is made, a confirm response is returned. Fig 7.11 shows the sequence diagram for a virtual sensor deletion.

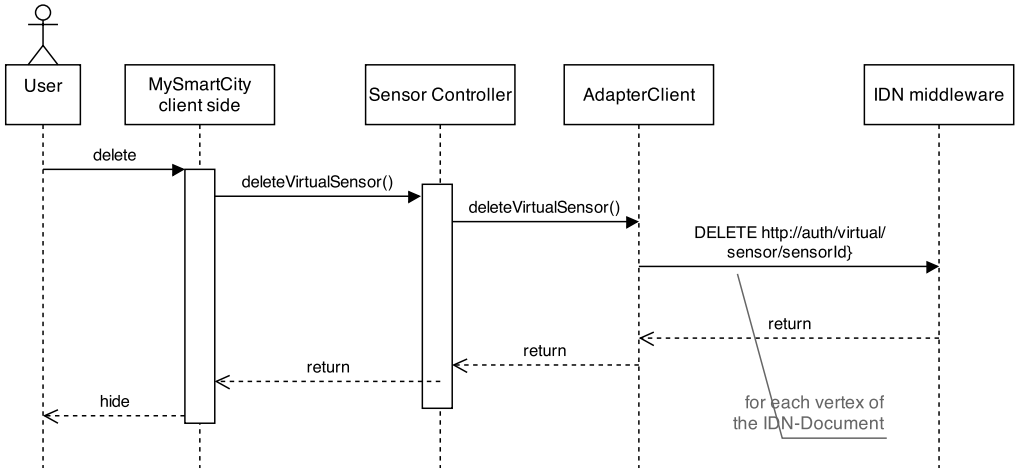


Figure 7.11. *The sequence diagram for a virtual sensor deletion.*

7.2.5.4 Managing Points of Interest

Unlike the Case Study A, the management of the PoIs is fully performed client-side, using the IDN.js library. As discussed in section 6.2, this library provides a set of methods helping the developer in realizing web pages based on the JavaScript language, offering APIs for the client-side management of IDN-Documents.

When a user clicks on a PoI, MySmartCity issues the following request to the InterDataNet middleware client-side, to retrieve the PoI IDN-Document.

```
GET http://authority/PoI/{Id}/
```

Then, the IDN-Document is parsed and the information concerning the PoI is shown on the map. When an administrator sends a modification request for a

PoI, MySmartCity issues an HTTP PUT client-side request to the InterDataNet middleware, as follows

```
PUT http://authority/PoI/{Id}
```

In order to delete a PoI, MySmartCity sends an HTTP DELETE client-side request to the InterDataNet middleware, as follows

```
DELETE http://authority/PoI/{Id}.
```

7.2.6 User Creation of Personal Applications

In this section I show how tools provided in the InterDataNet experiment may help a user in creating "personal" basic Web Applications allowing to browse and handle sensors related information. The user can exploit the provided tools to modify and aggregate existing web resources available as IDN-Documents and create new web resources.

In this context, the following categories are considered as target users: web application developers, who may take advantage of these tools to speed up their design and development tasks; end users experienced with the use of web technologies (i.e. early technology adopters). The users are provided with the IDN-Studio and IDN-Viewer tools.

7.2.6.1 Usage Example: Refining MySmartCity Application

Consider the case that MySmartCity administrator is willing to create a new PoI for the Santander Cathedral (Catedral de Nuestra Señora de la Asunción de Santander). The PoI can be designed to carry some descriptive information about the building and his history and then it can be further characterized by associating some near sensor nodes, say to monitor environmental and pollution parameters (e.g. temperature, light, noise, etc.).

To accomplish this task, the administrator creates the PoI by interacting with MySmartCity Application GUI. In the creation phase, the administrator selects the suitable sensors. After that, the administrator is able to modify the structure of the PoI and to insert the cathedrals history and photos; to this end he opens the PoI in the IDN-Studio application and uses this tool to aggregate a new node that contains a brief summary of the cathedrals history derived from the related Wikipedia page.

Then he/she can also add a photo by creating a new node and filling it with the Base64 encoding of a photo of the cathedral obtained in the Flickr website. Once the administrator has confirmed the desired changes, the IDN-Document

representing the PoI is committed and made persistent; from now on every end user will see the environment data of the Santander cathedral as well as its history and photos. All these contents will be displayed by MySmartCity application using the IDN-Viewer, as shown in Fig 7.12.

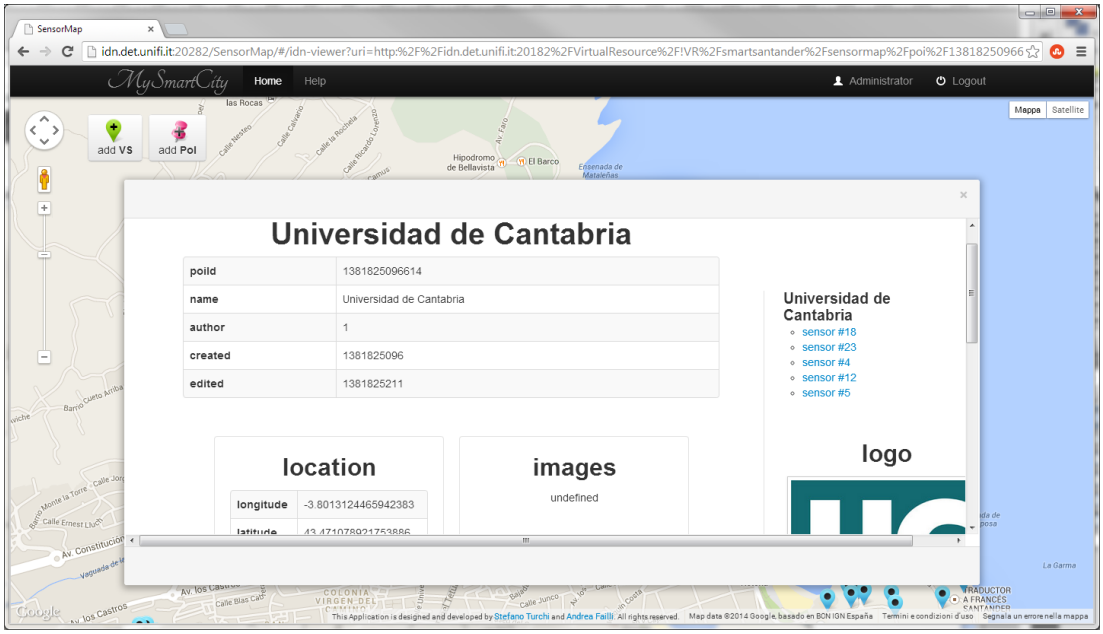


Figure 7.12. *Information related to a PoI shown on a modal window containing the IDN-Viewer rendering.*

7.2.6.2 Usage Example: Easy Web Resources Publishing

The IDN-Studio application and the IDN-Viewer library can also be intended as tools enabling the easy creation of simple Web Applications that model the user personal application domain.

An authenticated user may use the IDN-Studio to create new web resources as graphs of IDN-Documents and enrich this representation by drawing links with existing web resources (e.g. existing sensors, virtual sensors and PoIs). The creation of web resources for implementing a personal application is completed when the end user clicks on the "write" button in the IDN-Studio GUI. These web resources are now accessible from web browsers and can be navigated as traditional web pages. As a matter of fact, when a client invokes a GET operation on the URI of the web resource and specifies the "text/html" content-type , the middleware returns a representation of the resource embedding also the JavaScript libraries (i.e. the IDN-Viewer plugin) for the automatic document rendering.

To demonstrate how the InterDataNet framework can further support this user-centric vision, has been developed a proof of concept application extending

the approach to include also the sensors of the user private sphere. In the demonstration, the sensors integrated in an Android tablet (light sensor, accelerometer, and compass) are used.

A sensor gateway is implemented as an Android application. This application exploits the Android Sensor APIs for interacting with the embedded sensors and implements the features of Virtual Resource and Storage Interface to expose the Android sensors according to InterDataNet graph-based model. Thus, the application turns the user device into a provider of services allowing the user to query and control the sensors through fine-grained REST operations via a standard web browser.

Again, thanks to the features provided by the IDN-Studio tool and the IDN-viewer library, the user can create basic custom web applications that aggregate the information nodes of his own private sphere (i.e. Android sensors) with publicly accessible resources (i.e. Santander sensors, Virtual Sensors, PoIs, etc.). Fig. 7.13 represent a document open in IDN-Studio, aggregating documents representing sensors from SmartSantander and sensors of a personal Android device. Red vertexes represent existing resources from the Santander domain, green vertexes represent existing resources provided by an Android device, blue vertexes are resources created via the IDN-Studio GUI, linking the two documents.

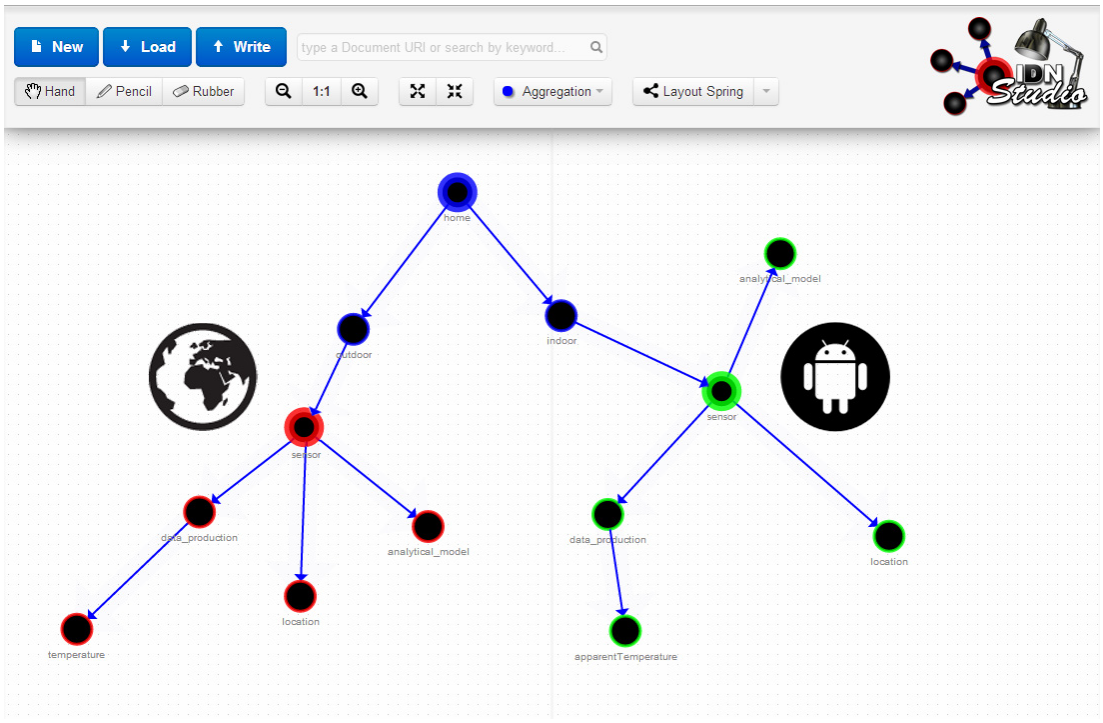


Figure 7.13. *Building documents with data from a personal Android device and from SmartSantander experiment.*

7.2.7 Evaluation

The evaluation process aims at assessing the efficacy and effectiveness of the experiment in achieving the objective of offering services and tools for the efficient management and processing of IoT data, in order to allow an easy integration of these endpoints into the Web. The evaluation is distinguished in two parts:

Qualitative Evaluation : this task consists in evaluating how easily and effectively the experiment achieves the goals of: i) exposing and handling sensors as a Web of Resources and ii) allowing users to browse this Web of Resources and create new Resources (e.g. Virtual Sensors and Points of Interest) by assembling parts of existing resources and/or adding new sensors and related information.

Quantitative Evaluation : We defined and measured a set of parameters modelling the performance of the IDN Middleware and the Sensor and PoI Map application.

7.2.7.1 Qualitative Evaluation

The objective of the qualitative evaluation consists in evaluating how easily and effectively the experiment achieves the goals of:

- exposing and handling sensors (i.e., sensors descriptive information and functional capabilities as well as related measurements) as a Web of Resources.
- allowing users (i.e., web application developers) to browse this Web of Resources and create new Resources by assembling parts of existing resources and/or adding new sensors and related information. This includes the capability of adding Virtual Sensors that can be placed in locations of interest.

The qualitative evaluation has been carried out by involving a group of users. The study had the objective of evaluating the degree of acceptance of implemented features when accessed by users with no previous experience of the applications. The aim of the experimentation was to:

1. evaluate the effectiveness of the tools offered for handling and using sensor digital representations based on web and REST principles and the advantages and/or disadvantages perceived by the users;
2. analyze potential disadvantages and qualify them (e. g., major usability problems and technical bugs);
3. gather subjective user satisfaction evaluation.

The evaluation session has been organized as a half-day workshop. In the first part of the workshop, a member of the InterDataNet experiment team provided an introduction of the SmartSantander project, introduced the experiment goals and described the applications and tools to be evaluated:

- MySmartCity application: we explained the meaning of Virtual Sensors and Point of Interest resources and showed how to view details of existing sensor nodes and how to add new virtual sensors or PoI resources on the map:
  - select resources by types (sensors, virtual sensors and PoIs);
  - smartSantander sensors: view details of a sensor;
  - Virtual Sensors: view details of existing VSs, creation of a new VS, view the new VS on the map, view details of the new VS, share the VS (make public);
  - PoIs: view details of existing PoIs, creation of a new PoIs, view the new PoIs on the map, view details of the new PoIs.
- modification of a web resource via IDN-Studio and visualization of the modified resource: create a PoI via MySmartCity application, the PoI is then modified via the IDN-Studio and the new resource can be accessed.

In the second part of the workshop, the users were asked to use the application and the tools via a web browser to accomplish a sequence of goals:

- creation of a new Virtual Sensor. The user has to create a new virtual sensor and then make it public.
- creation of a new PoI: The user has to create a new PoI that aggregates some sensor information and then make it public
- modification of a personalized PoI through IDN-Studio: the user opens the previously created PoI on the IDN-Studio application. Enhance its representation by adding a picture and some other information (e.g. history). Then the user confirms the operation and commits the changes to the middleware. The end user now goes back to MySmartCity application, clicks on the PoI, selects the "view more" command and checks that the new content is displayed.

Users were provided with a text containing the sequence of goals to be achieved. User feedbacks have been collected through interviews and a questionnaire at the end of the evaluation session.

The evaluation session was conducted with 10 test users. The users were selected among the colleagues in the Department of Information Engineering at the University of Florence. As a consequence, all users are expert in the use of web technologies, some of them (10%) are also expert web developers. None of them has previously used InterDataNet tools and applications. All users are aged within 30-44 years.

Hereafter are reported the results obtained by submitting a questionnaire to the end users after the workshop. Tab. 7.3 and Tab. 7.4 report the feedbacks provided by the users to multiple-choice questions. As shown in Tab. 7.3, the majority of users were quite satisfied with MySmartCity application. Most of them found accessing sensor information and creating new virtual sensors to be quite easy. The application has been judged useful and interesting by most users.

Most users (80%) provided a positive comment on the overall intuitiveness and ease of use of the application interface. However, a large percentage of users experienced a short delay in the application refresh (e.g. after a creation of a virtual sensor). This problem is due to the interaction of the application with the middleware search service. When a new resource is created by the side of the middleware, it takes a little time before is indexed for the search. This issue will be fixed providing the application with a caching strategy and improving the middleware indexing workflow. A different approach could also be adopted by bypassing the search system for the newly created resource, and leveraging the HTTP `location` header to get the resource identifier. Both strategies will be evaluated.

Some users also made suggestions to improve the application and enhance it with additional features. A user suggested adding a feature for converting measured values between different measurement units. Another suggestion consisted in highlighting geographical areas of dense population and/or with locations of interests with different colors to guide a user in easily creating PoIs and locating them in the areas of major interest.

Moreover, they would find useful, while creating a Virtual sensor or a PoI, to automatically visualize the sensors that are closest to the newly created resource.

Tab. 7.4 shows the feedbacks provided on the use of IDN-Studio for modifying and enriching the description of a PoI. The users succeeded in completing the assigned tasks, and 40% considered the application intuitive and easy to use. The majority of users provided positive comments on the visualization of the changes they applied to the PoI on the web browser. Some suggestions were provided to enhance the interface with some help text and brief explanations of labels to guide a first-time user in the use of the tool.

MySmartCity Evaluation	Definitely Yes	Yes	Neutral	No	Definitely No
Is it easy to select and access information on a sensor?	40%	60%	-	-	-
Do you think it is easy to create a new virtual sensor?	20%	80%	-	-	-
Do you think that this web application might be useful in you everyday life, in your city?	10%	80%	10%	-	-

Table 7.3. Results for MySmartCity evaluation.

Personalization through IDN-Studio Evaluation	Definitely Yes	Yes	Neutral	No	Definitely No
Is IDN-Studio intuitive and easy to use?	20%	20%	60%	-	-
Is it easy to modify a PoI through IDN-Studio?	-	40%	60%	-	-
Have you succedeed in the designing the target PoI?	-	60%	40%	-	-
The rendering of your personalized PoI on the web browser was appropriate?	20%	60%	20%	-	-
The rendering of your personalized PoI on the web browser was satisfactory?	-	80%	20%	-	-

Table 7.4. Results for the IDN-Studio evaluation.

7.2.7.2 Quantitative Evaluation

The objective of the quantitative evaluation consists in evaluating the performance of the system architecture described in section 7.2.3.

The different components of the system architecture were deployed on three twin virtual machines (VM), each equipped with 1 GB RAM and running the Debian 6 distribution. These VMs are installed on a HP ProLiant ML350 server with a two Intel Xeon quad-core processors, 17 GB RAM and Debian 6 OS.

We performed a set of tests to profile the main components of the experiment system: the InterDataNet middleware, MySmartCity application, and the Virtual Sensor Adapter. Tests were performed by leveraging the developer suite provided by the Chrome browser [Goo] and the Postman rest client extension for Chrome

[Ast]. During the test phase, six different parameters have been considered:

- request-response time: the time elapsing between the delivery of a request and the related response reception by the client;
- operation time: it is applicable when an operation requires more HTTP requests to be completed. The operation time is the time between the first request and the last response.
- overall request body: it is applicable when an operation requires more HTTP requests to be completed. The size of the overall request body is the sum of every request body.
- overall response body: it is applicable when an operation requires more HTTP requests to be completed. The size of the overall response body is the sum of every response body.
- request body: the size of the request body, if available, expressed in bytes.
- response body: the size of the response body, expressed in bytes.

The same set of tests have been performed three times with different configurations of the client machines. Not to encumber the reading, the results of only one test are shown in this section. The tests reported hereafter were performed on a notebook with a dual-core Intel Core 2 Duo P8400 2.26 GHz processor, 4 GB RAM, accessing the WLAN provided by the University of Florence. Each test has been performed more times and the simple mean and trimmed mean upon the resulting values are computed. The trimmed mean is computed as a simple mean on the measurements set with the highest and the lowest value removed. This operation makes the measurement more robust to outlier perturbations.

As regards MySmartCity application, has been evaluated the time needed for performing the following operations:

- retrieval of a physical sensor;
- creation of a virtual sensor;
- retrieval of a virtual sensor;
- editing of a virtual sensor;
- creation of a PoI;
- retrieval of a PoI;
- editing of a PoI.

Each operation was performed on ten different resources with comparable payload (i.e. retrieval of 10 physical sensors, creation of 10 virtual sensors and so on). The following tables report the mean and trimmed mean values for each operation.

Actual sensor retrieval	**Simple Mean**	**Trimmed Mean**
operation time (ms)	464	456
overall response body (byte)	2212	2193

Table 7.5. Latency for an actual sensor retrieval request.

Virtual Sensor Creation	**Simple Mean**	**Trimmed Mean**
operation time (ms)	1216	1217
overall request body (byte)	1144	1138
overall response body (byte)	1279	1275

Table 7.6. Latency for an virtual sensor creation request.

Virtual Sensor Retrieval	**Simple Mean**	**Trimmed Mean**
operation time (ms)	915	951
overall response body (byte)	3180	3174

Table 7.7. Latency for an virtual sensor retrieval request.

Virtual Sensor Modification	**Simple Mean**	**Trimmed Mean**
operation time (ms)	471	470
overall request body (byte)	1339	1284
overall response body (byte)	1370	1325

Table 7.8. Latency for an virtual sensor modification request.

Has been also measured the time needed for creating a VS-Adapter resource, which is an operation triggered by a Virtual Sensor creation. The test consisted in the submission of the creation request. This operation has been performed

PoI Creation	**Simple Mean**	**Trimmed Mean**
operation time (ms)	714	708
overall request body (byte)	2451	2437
overall response body (byte)	3250	3238

Table 7.9. Latency for an PoI creation request.

PoI Retrieval	**Simple Mean**	**Trimmed Mean**
operation time (ms)	1020	1038
overall response body (byte)	5458	5375

Table 7.10. Latency for PoI retrieval request.

PoI Modification	**Simple Mean**	**Trimmed Mean**
operation time (ms)	82	69
overall request body (byte)	840	756
overall response body (byte)	1357	1354

Table 7.11. Latency for a PoI modification request.

VS-Adapter Resource Creation	**Simple Mean**	**Trimmed Mean**
operation time (ms)	976	974
overall request body (byte)	613	613
overall response body (byte)	777	777

Table 7.12. Latency for a sensor creation request to the VS-Adapter application.

ten times before computing the mean values. The following table reports the measurement results.

The InterDataNet middleware was tested in order to assess the timing related to the following main operations: i) Creation of an IDN-Document, ii) Retrieval of an IDN-Document. Documents composed by 1, 10, 30 and 50 nodes were considered. For each different size, 10 identical IDN-Documents were created and retrieved. The following table reports the obtained results.

Fig. 7.14 show a chart representing the latency for IDN-Document creation and retrieval requests, and their variation according to the number of nodes within the IDN-Document.

Operation	Req.-resp. time (ms)		Exchanged data (KB)	
	Simple M.	Trimmed M.	Req. Body	Resp. Body
1-Node doc. creation	193	192	0.488	0.722
1-Node doc. retrieval	75	73	n.a.	0.722
10-Nodes doc. creation	2375	2253	5.8	5.6
10-Nodes doc. retrieval	626	625	n.a.	5.6
30-Nodes doc. creation	5929	5606	14.8	15.1
30-Nodes doc. retrieval	1756	1704	n.a.	15.1
50-Nodes doc. creation	11565	10748	24.9	25.2
50-Nodes doc. retrieval	2871	2865	n.a	25.2

Table 7.13. Latency for requests on IDN-Documents.

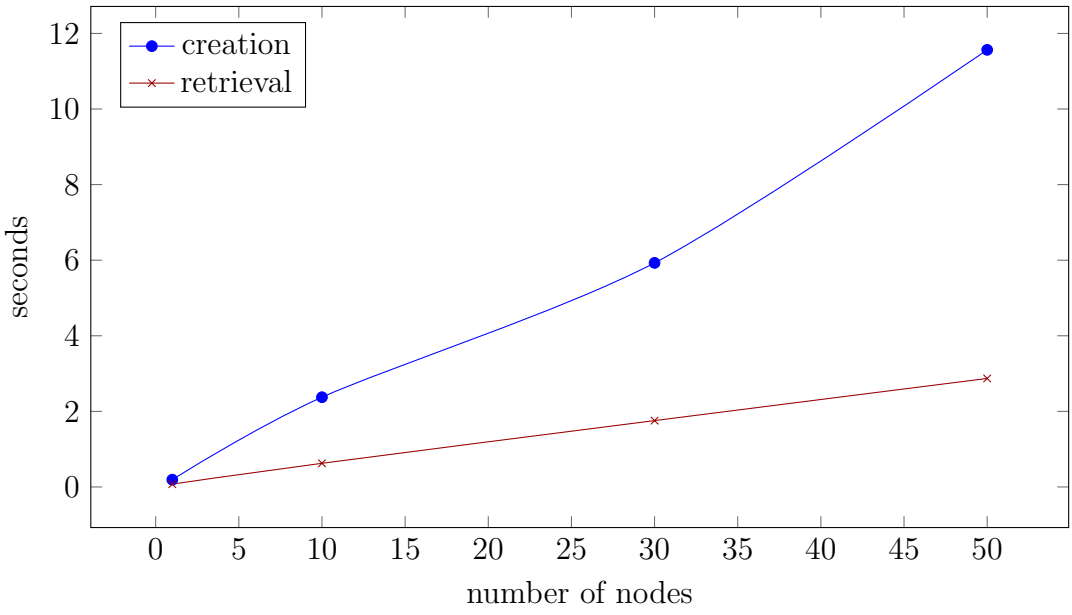


Figure 7.14. The latency (simple mean) for retrieval and creation of IDN-Documents of different sizes (1, 10, 30, 50 nodes).

Chapter 8

Conclusions

This thesis presents and details InterDataNet, a framework for information aggregation in the form of graphs of individually addressable information pieces. It aims at creating a *read/write* Web of Documents, by exposing these graphs as resources called IDN-Documents whose vertexes can be manipulated with CRUD operations.

As a consequence, IDN-Documents can be linked together and new IDN-Documents can be built by reusing parts of different IDN-Documents. Moreover, several properties (e.g. licensing, privacy, etc.) are enabled on the single vertex and on the IDN-Document as a whole.

The framework comprehends an information model called InterDataNet Information Model defining the set of rules that drive the representation of InterDataNet entities such as IDN-Documents, IDN-Nodes and different types of Links (Aggregation Link, Reference Link, etc.).

The Information Model is implemented through the InterDataNet middleware, a fully RESTful layered architecture providing services for the management of IDN-Documents, names, history and persistence. A detached module, the Adapter, provides adaptation capabilities towards external data sources to represent external data as IDN-Documents.

InterDataNet is provided also with the Activity Node, a scriptable vertex for dynamically generating data using other vertexes' contents as inputs.

Finally, the framework includes a tool-suite, to support the developer in taking advantage of the framework capabilities. The tool-suite components are the IDN Java Library for the server-side exploitation of the IDN-Document, the IDN.js

which is a JavaScript client-side version, IDN-Studio, a visual editor, and IDN-Viewer a JQuery plugin for rendering IDN-Documents as HTML pages.

InterDataNet has been selected to participate in the SmartSantander FP7 EU project, to provide Web of Resources capabilities in a real-word Smart City scenario. To this end, sensor data coming from the experiment platform has been represented as IDN-Documents, and added-value applications have been built on top of them. This experiment has served well for InterDatNet validation, and qualitative and quantitative evaluations have been also produced.

The SmartSantander experiment represented a great opportunity to test the framework in a real context and some conclusions have been drawn. First, InterDataNet has proved to be effective in information composition. Documents can be created with ease and immediately made available by the architecture. Second, these documents can be managed in a refined way, by fully interacting with their information grains. In addition, reuse capabilities discussed in this thesis have proved to be actual and effectual. Indeed, during the experiment, IDN-Documents representing sensors from the SmartSanatander facility have been reused to build different types of IDN-Documents (Points of Interest) with the expected ease and benefits. In the same direction, an experiment has been successfully conduced to build a "dashboard" document assembling environmental sensors from SmartSantander with personal sensors from an Android device.

A special mention goes to the toolsuite which played a very important role in simplifying the development of applications consuming and producing IDN-Documents. Moreover, users who tested visual tools (IDN-Studio and IDN-Viewer) provided good feedbacks.

In conclusion, results are encouraging and opportunities coming from the aggregation capabilities of the IDN-Document are promising.

Unfortunately, the Active Node concept couldn't be validated in the experiment due to its recent conception. The strategy adopted to compute the Virtual Sensor output required a dedicated application (VS-Application), and an Adapter (VS-Adapter). Admittedly, this solution is far from being smooth, for two reasons: first, it should be easy to implement dynamic generated contents and two additional modules are too many. Second, this solution complicates significantly the Virtual Sensor related workflows of the main application (MySmartCity).

Improvements are surely required for the IDN-Viewer JQuery plugin to support more types of rendering types.

The InterDataNet architecture performs average, but much could be done in this direction, starting from the Performance Enhancement System implementation.

Probably, the enforcement of properties is the most urgent objective. As detailed in section 4.1.2.2, they are well supported by the model but they have

not been fully implemented yet. More precisely, security among the them is the one that should be addressed first.

As regards the evaluation discussed in section 7.2.7, a more comprehensive assessment should be performed in order to confirm these results. Above all, a wider user base should be involved to make the outcomes more significant.

Future works comprehend the prosecution of research on the Activity Node, which is a topic of recent conception, the implementation of the Security Framework, and improvements of the overall performance of the architecture by implementing the Performance Enhancement System described in section 5.2.5. Additional investments can be done in the direction of the toolsuite, to make it more flexible, easy to use and up to date with the advancements of technology. An interesting topic is also the integration of the Information Model with semantics, in order to provide advanced discovery capabilities. Moreover, the refinement and implementation of a model for IDN-Documents (analogous to the XML Schema) for validation purposes should be addressed.

List of Figures

Bibliography

[AHH+10] Benjamin Adrian, Jörn Hees, Ivan Herman, Michael Sintek, and Andreas Dengel. Epiphany: adaptable rdfa generation linking the web of documents to the web of data. In *Knowledge Engineering and Management by the Masses*, pages 178–192. Springer, 2010.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[Ash09] Kevin Ashton. That internet of things thing. *RFiD Journal*, 22:97–114, 2009.

[ASJH11] Paul Adamczyk, Patrick H Smith, Ralph E Johnson, and Munawar Hafiz. Rest and web services: In theory and in practice. In *REST: from research to practice*, pages 35–57. Springer, 2011.

[Ast] Abhinav Asthana. Postman chrome extension. `https://github.com/a85/POSTMan-Chrome-Extension/wiki`.

[AZ05] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited–a pattern. 2005.

[Bec04] Dave Beckett. RDF/xml syntax specification (revised). W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[Ber03] HHans Bergsten. *Java Server Pages*. O'reilly, 2003.

[BG04] Dan Brickley and Ramanathan Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.

[BHIBL08] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (ldow2008). In *Proceedings of the 17th international conference on World Wide Web*, pages 1265–1266. ACM, 2008.

[BK05] Christian Bauer and Gavin King. Hibernate in action. 2005.

[BK08] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFC 6874.

[BLHL+01] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[BLK+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia-a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[BLMM+94] Tim Berners-Lee, Larry Masinter, Mark McCahill, et al. Uniform resource locators (url). 1994.

[BM04] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[Bry88] Martin Bryan. *SGML*. Addison-Wesley, 1988.

[Bry12] Peter Bryant. Rest-ful uri design. `http://blog.2partsmagic.com/restful-uri-design/`, May 2012.

[BSMY+08] Tim Bray, Michael Sperberg-McQueen, François Yergeau, Jean Paoli, and Eve Maler. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[Buc97] Michael K Buckland. What is a "document"? *JASIS*, 48(9):804–809, 1997.

[Chi09] Davide Chini. *InterDataNet: una soluzione infrastrutturale per il Read-Write Web of Data Proof of concept dellarchitettura*. PhD thesis, Engineering, Florence, 2009.

[Cio10] Lucia Ciofi. *Future Internet:evoluzione in chiave REST per InterDataNet*. PhD thesis, Engineering, Florence, 2010.

[CLS+05] Francisco Curbera, Frank Leymann, Tony Storey, Donald Ferguson, and Sanjiva Weerawarana. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR Englewood Cliffs, 2005.

[Cro10] Douglas Crockford. Json in javascript. `https://github.com/douglascrockford/JSON-js`, nov 2010.

[CSFP04] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion*. O'Reilly, 2004.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[Dar06] Christian Darie. *AJAX and PHP: building responsive web applications*. Packt Publishing, 2006.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.

[DS04] Mike Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210.

[DuC11] Bob DuCharme. *Learning Sparql*. O'Reilly, 2011.

[DV06] Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In *Practical Aspects of Declarative Languages*, pages 88–102. Springer, 2006.

[Fai12] Andrea Failli. Progettazione e produzione di un'applicazione visuale per la gestione di grafi di informazioni granulari all'interno del framework interdatanet, jun 2012. Bachelor Thesis, University of Florence.

[FB96a] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. RFC 2049 (Draft Standard), November 1996.

[FB96b] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231, 5335, 6532.

[FB96c] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard), November 1996. Updated by RFCs 2646, 3798, 5147, 6657.

[FEC+04] Robert Filman, Tzilla Elrad, Siobhán Clarke, et al. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[Fie00a] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[Fie00b] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[Fie08] Roy T. Fielding. Rest apis must be hypertext-driven. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`, 2008.

[FK05a] N. Freed and J. Klensin. Media Type Specifications and Registration Procedures. RFC 4288 (Best Current Practice), December 2005. Obsoleted by RFC 6838.

[FK05b] N. Freed and J. Klensin. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 4289 (Best Current Practice), December 2005.

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[FP13] Lee Feigenbaum and Eric Prud'hommeaux. SPARQL by example a tutorial. `http://www.cambridgesemantics.com/semantic-university/sparql-by-example`, 2013.

[Fre04] Elisabeth Freeman. *Head first design patterns*. O'Reilly Media, Inc., 2004.

[G+05] Jesse James Garrett et al. Ajax: A new approach to web applications, 2005.

[GB10] Lisa Goddard and Gillian Byrne. Linked data tools: Semantic web for the masses. *First Monday*, 15(11), 2010.

[GGP+10] Roberto García, Juan Manuel Gimeno, Ferran Perdrix, Rosa Gil, Marta Oliva, Juan Miguel López, Afra Pascual, and Montserrat Sendín. Building a usable and accessible semantic web interaction platform. *World wide web*, 13(1-2):143–167, 2010.

[GGS+13] Verónica Gutiérrez, Jose A Galache, Luis Sánchez, Luis Muñoz, Jose M Hernández-Muñoz, Joao Fernandes, and Mirko Presser. Smartsantander: Internet of things research and innovation through citizen participation. In *The Future Internet*, pages 173–186. Springer, 2013.

[GHNO08] Joe Gregorio, M Hadley, M Nottingham, and D Orchard. Uri template. *Network Working Group Internet-Draft*, 2008.

[GIM12] Dominique Guinard, Iulia Ion, and Simon Mayer. In search of an internet of things service architecture: Rest or ws-*? a developers perspective. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 326–337. Springer, 2012.

[GKT02] Sven Graupner, Vadim Kotov, and Holger Trinks. Resource-sharing and service deployment in virtual data centers. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 666–671. IEEE, 2002.

[Goe07] S. Goessner. JSONPath - XPath for JSON, February 2007.

[Goo] Google. Google chrome. `https://www.google.com/intl/en/chrome/browser/`.

[GSG+12] J.A. Galache, J.R. Santana, V. Gutierrez, L. Sanchez, P. Sotres, and L. Munoz. Towards experimentation-service duality within a smart city scenario. In *Wireless On-demand Network Systems and Services (WONS), 2012 9th Annual Conference on*, pages 175–181, 2012.

[HDS06] Tom Heath, John Domingue, and Paul Shabajee. User interaction and uptake challenges to successfully deploying semantic web technologies. In *SWUI 2006, the 3rd International Semantic Web User Interaction Workshop (colocated with ISWC2006)*, 2006.

[HL95] Walter L Hursch and Cristina Videira Lopes. Separation of concerns. 1995.

[HLRV03] Erik Hatcher, Steve Loughran, Matthew Robinson, and Pavel Vorobiev. *Java development with Ant*. Manning, 2003.

[HMM13] José M Hernández-Muñoz and Luis Muñoz. The smartsantander project. In *The Future Internet*, pages 361–362. Springer, 2013.

[Hof05a] P. Hoffman. The gopher URI Scheme. RFC 4266 (Proposed Standard), November 2005.

[Hof05b] P. Hoffman. The telnet URI Scheme. RFC 4248 (Proposed Standard), October 2005.

[HP02] Stefan Haustein and Jörg Pleumann. Is participation in the semantic web too difficult? In *The Semantic Web ISWC 2002*, pages 448–453. Springer, 2002.

[Inn08] Samuele Innocenti. *InterDataNet: nuove frontiere per lintegrazione e lelaborazione dei dati Visione e progettazione di un modello infrastrutturale per linterdataworking*. PhD thesis, Engineering, Florence, 2008.

[J+10] Jeff Johnson et al. *Designing with the mind in mind: Simple guide to understanding user interface design rules*. Morgan Kaufmann, 2010.

[Jam06] Anthony Jameson. Usability and the semantic web. In *The Semantic Web: Research and Applications*, pages 3–3. Springer, 2006.

[Jaz07] Mehdi Jazayeri. Some trends in web application development. In *Future of Software Engineering, 2007. FOSE'07*, pages 199–213. IEEE, 2007.

[JHAT09] Rod Johnson, Juergen Hoeller, Alef Arendsen, and R Thomas. *Professional Java Development with the Spring Framework*. Wiley. com, 2009.

[JHR99] Ian Jacobs, Arnaud Le Hors, and Dave Raggett. HTML 4.01 specification. W3C recommendation, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[Jos03] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 3548 (Informational), July 2003. Obsoleted by RFC 4648.

[JV09] Valentina Janev and Sanja Vranes. Semantic web technologies: Ready for adoption? *IT professional*, 11(5):8–16, 2009.

[Kay07] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[KGK+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*. Manning, 2007.

[Kun13] Maurice de Kunder. The size of the world wide web (the internet). *WorldWideWebSize. com http://www. worldwidewebsize. com/(Dec 2013)*, 2013.

[Las99] Ora Lassila. Resource description framework (RDF) model and syntax specification. W3C recommendation, W3C, February 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[LCFS11] Geetika T Lakshmanan, Francisco Curbera, Juliana Freire, and Amit Sheth. Provenance in web applications. *IEEE Internet Computing*, pages 17–21, 2011.

[LR01] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[Mar03] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[MDV04] Edward McCormick and Kris De Volder. Jquery: finding your way through tangled code. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10. ACM, 2004.

[Mic11] Sun Microsystems. Jdk 1.6.0 documentation. sun microsystems, 2011.

[MJGSB11] Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8. ACM, 2011.

[MM04] Frank Manola and Eric Miller. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[Moo96] K. Moore. MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text. RFC 2047 (Draft Standard), November 1996. Updated by RFCs 2184, 2231.

[MRSS96] Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-oriented software architecture, volume 1: A system of patterns, 1996.

[MvH04] Deborah McGuinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[Ohm10] Paul Ohm. Broken promises of privacy: Responding to the surprising failure of anonymization. *UCLA Law Review*, 57:1701, 2010.

[OM03] Ed Ort and Bhakti Mehta. Java architecture for xml binding (jaxb). *Sun Developer Network*, 2003.

[O'S09] Bryan O'Sullivan. *Mercurial: The definitive guide*. O'Reilly Media, Inc., 2009.

[PS03] Aiko Pras and Juergen Schönwälder. On the difference between information models and data models. Technical report, RFC 3444, January, 2003.

[PS08a] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for rdf. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/rdf-sparql-query/.

[PS08b] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[Ran98] Brian Randell. Dependability a unifying concept. In *Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings*, pages 16–25. IEEE, 1998.

[RD90] John T Robinson and Murthy V Devarakonda. *Data cache management using frequency-based replacement*, volume 18. ACM, 1990.

[Rep06] Dominic J Repici. The comma separated value (csv) file format. *Creativyst Software*, 2006.

[Res00] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Updated by RFC 5785.

[RH90] Lans P Rothfusz and NWS Southern Region Headquarters. The heat index equation (or, more than you ever wanted to know about heat index). *Fort Worth, Texas: National Oceanic and Atmospheric Administration, National Weather Service, Office of Meteorology*, pages 90–23, 1990.

[RLHJ+99] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[RMS97] Gail L Rein, Daniel L McCue, and Judith A Slein. A case for document management functions on the web. *Communications of the ACM*, 40(9):81–89, 1997.

[Rot09] Greg Roth. Restful http in practice, 2009.

[SGG+11] Luis Sanchez, José Antonio Galache, Veronica Gutierrez, JM Hernandez, J Bernat, Alex Gluhak, and Tomás Garcia. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network & Mobile Summit (FutureNetw), 2011*, pages 1–8. IEEE, 2011.

[SGG+13] Luis Sanchez, Veronica Gutierrez, Jose Antonio Galache, Pablo Sotres, Juan Ramon Santana, Javier Casanueva, and Luis Munoz. Smartsantander: Experimentation and service provision in the smart city. In *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pages 1–6. IEEE, 2013.

[Sle10] Brian Sletten. Resource-oriented architecture: The rest of rest. *InfoQ, March*, 2010.

[SMG+13] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, et al. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 2013.

[SMGB+12] Michael Sperberg-McQueen, Sandy Gao, David Beech, Noah Mendelsohn, Murray Maloney, and Henry Thompson. W3C xml schema definition language (XSD) 1.1 part 1: Structures. W3C recommendation, W3C, April 2012. http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[SvE88] Barry Smith and Christian Freiherr von Ehrenfels. *Foundations of Gestalt theory*. Philosophia Verlag Munich, 1988.

[Swa02] Aaron Swartz. The semantic web in breadth. *WWW-address: http://logicerror. com/semanticWeb-long*, 2002.

[Swi08] Travis Swicegood. *Pragmatic version control using Git*. Pragmatic Bookshelf, 2008.

[TFKC11] Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of queries on semantic web data. *Internet Computing, IEEE*, 15(1):31–39, 2011.

[Til07] Stefan Tilkov. A brief introduction to rest. *InfoQ, Dec*, 10, 2007.

[Urp08] J. Urpalainen. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. RFC 5261 (Proposed Standard), September 2008.

[Wol94] Pree Wolfgang. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

[YBCD08] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding mashup development. *Internet Computing, IEEE*, 12(5):44–52, 2008.

[ZEWH95] Stuart H Zweben, Stephen H Edwards, Bruce W Weide, and Joseph E Hollingsworth. The effects of layering and encapsulation on software development cost and quality. *Software Engineering, IEEE Transactions on*, 21(3):200–208, 1995.

[ZS12] Ivan Zuzak and Silvia Schreier. Arrested development: Guidelines for designing rest frameworks. *Internet Computing, IEEE*, 16(4):26–35, 2012.

[ZSM+11] Jun Zhao, Satya S Sahoo, Paolo Missier, Amit Sheth, and Carole Goble. Extending semantic provenance into the web of data. *Internet Computing, IEEE*, 15(1):40–48, 2011.