NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 – "Semantic-based knowledge and content systems"

D 5.2.1 NeOn Protocols for Exchanging and Sharing Ontologies

Deliverable Co-ordinator: **Óscar Muñoz García**

Deliverable Co-ordinating Institution: **UPM**

Other Authors: **Carlos Buil Aranda (ISOCO),**
Saartje Brockmans (UKARL), Carola Catenacci (CNR),
Miguel Esteban-Gutiérrez (UPM),
Raúl García-Castro (UPM),
Asunción Gómez-Pérez (UPM),
Peter Haase (UKARL), Jos Lehmann (CNR),
Holger Lewen (UKARL), Pilar López Atau (UPM),
Ángel López (UPM), Elena Montiel (UPM),
Raul Palma (UPM), Valentina Presutti (CNR)
Marta Sabou (OU),
Mari Carmen Suárez-Figueroa (UPM),
Walter Waterfeld (SAG), Moritz Weiten (ONTO),
Yimin Wang (UKARL)

Document Identifier:	NEON/2007/D5.2.1/v2.3	Date due:	February 28, 2007
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	March 30, 2007
Project start date:	March 1, 2006	Version:	V2.3
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities, grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta} @open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 28 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.upm.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Richard Benjamins E-mail address: rbenjamins@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SI-1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 655 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Koblenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Martino della Battaglia, 44 - 00185 Roma-Lazio, Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Asociación Española de Comercio Electrónico (AECE)** C/Alcalde Barnils, Avenida Diagonal 437 08036 Barcelona Spain Contact person: Jose Luis Zimmerman E-mail address: jlzimmerman@fecemd.org
Food and Agriculture Organization of the United Nations (FAO) Viale delle Terme di Caracalla 1 00100 Rome, Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org	**Atos Origin S.A. (ATOS)** Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed writing parts of this document:

CNR

ISOCO

SAG

ONTO

OU

UPM

UKARL

Change Log

Version	Date	Amended by	Changes
0.1	21-09-2006	Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez	Initial Draft
0.2	2-10-2006	Mari Carmen Suárez-Figueroa, Miguel Esteban-Gutiérrez	Included subsection about Grid Protocols
0.3	4-10-2006	Mari Carmen Suárez-Figueroa, Elena Montiel	Included definitions of exchanging and sharing
0.4	5-10-2006	Mari Carmen Suárez-Figueroa, Holger Lewen	Included subsection about Agent Languages
0.5	6-10-2006	Mari Carmen Suárez-Figueroa, Raul Palma	Included subsection about P2P Protocols
0.6	10-10-2006	Mari Carmen Suárez-Figueroa, Raúl García-Castro	Included subsection about ICE Protocol
0.7	23-11-2006	Óscar Muñoz-García, Asunción Gómez-Pérez	Included new organization and protocols
0.8	4-12-2006	Óscar Muñoz García, Pilar López Atau	Included J2EE
0.9	14-12-2006	Óscar Muñoz García	Note on Sources and Original Contributions
1	19-12-2006	Óscar Muñoz García, Holger Lewen, Carlos Buil Aranda	POP3, IMAP, SMTP, CVS, SVN, RPC, RMI, CORBA, IDL, JDBC, ODBC and Jabber
1.1	21-12-2006	Óscar Muñoz García, Marta Sabou	Web Services, Semantic Web Services
1.2	22-12-2006	Óscar Muñoz García, Peter Haase	Included DIG
1.3	13-2-2007	Óscar Muñoz García	Document changes according to Bled meeting
1.4	14-2-2007	Óscar Muñoz García, Walter Waterfeld	WebDAV, UDDI, ebXML, Subversion, SOAP, WSDL
1.5	19-2-2007	Óscar Muñoz García, Walter Waterfeld, Yimin Wang	Included XQuery, XML-Schema, SPARQL
1.6	20-2-2007	Óscar Muñoz García, Walter Waterfeld, Jos Lehmann, Carola Catenacci	Included XQJ, Social protocols through the literature on collaboration

1.7	21-2-2007	Óscar Muñoz García, Jos Lehmann, Carola Catenacci, Moritz Weiten	Conclussions on social protocols, OSGI
1.8	26-2-2007	Óscar Muñoz García, Walter Waterfeld	SDO, SCA, SA-WSDL, WS-Policy, WS-Security,
1.9	1-3-2007	Óscar Muñoz García	Language corrections
2.0	16-3-2007	Asunción Gómez Pérez, Raúl García Castro, Óscar Muñoz García	Changes in Executive Summary, Introduction and Conclusions
2.1	21-3-2007	Óscar Muñoz García, Saartje Brockmans, Holger Lewen, Peter Haase	Queries in F-Logic, Languages for working with Agents, DIG
2.2	25-3-2007	Óscar Muñoz García, Carlos Buil Aranda, Walter Waterfeld, Miguel Esteban, Marta Sabou, Ángel López	Changes in: RPC, IDL, CORBA, RMI, JDBC, ODBC, WebDAV, XML Schema, Web Services, WS-Security, WS-Policy, SA-WSDL, SCA and SDO, WS-DAI, Web services, JDO, Hibernate
2.3	30-3-2007	Óscar Muñoz García, Valentina Pressuti	Requirements related to collaborative aspects

Executive Summary

Within the task T5.2 (in WP5), we intend to list protocols and techniques for exchanging and sharing ontologies and related metadata. These protocols and techniques support collaborative construction and dynamic evolution of networked ontologies. The networked ontologies are in environments where geographically distributed teams carry out re-engineering, alignment, merging, learning and evaluation activities. The identification of protocols and techniques for exchanging and sharing information is crucial for this task.

The title “NeOn protocols for exchanging and sharing ontologies” could be seen a bit misleading because it could seem that we are going to describe or define protocols to be used in the NeOn project in this deliverable. It will be in WP6 where protocols and techniques that will be used in the NeOn architecture will be defined. So, the aim of this deliverable is to enumerate and describe some protocols, formats and standards that could serve as input to WP6. Some of them are already chosen in WP6, others could be taken into account in a future for implementing new features according to future needs.

The method used for deciding which type of protocols and techniques for exchanging and sharing information in the literature are going to be included in this deliverable is to analyse the requirements already established in D6.1.1 “Requirements on NeOn Architecture”. The analysis of D6.1.1 allowed us to identify a list of potential NeOn needs related to protocols, formats and standards for exchanging and sharing information (e.g., protocols for accessing remote services, version management protocols, etc.). For each particular type of protocol and technique, we have included widely used and/or standardized realizations as well as new ones that are being currently proposed in other research projects. Out of the scope of this deliverable is to review and analyse them in depth and to provide a recommendation to WP6.

Another important aspect is to analyse protocols from a social component of collaboration. Since in WP2, deliverable D2.1.1, include a section on social protocols trough the literature on collaboration, with the goal of doing the deliverable self-contained, we have also included a summary of the content presented there. Again, it is out of the scope of this deliverable to decide the collaboration process to be used in NeOn, which is the goal of WP2.

In summary, the role of this document is to analyse the state of the art of technical and social protocols and techniques and related issues (such as formats and standards) for exchanging and sharing information according to the needs identified in D6.1.1. To define the technical NeOn protocols for exchanging and sharing ontologies and social protocols are out of the scope of this deliverable, so, they will be defined in WP6 and WP2 respectively.

Note on Sources and Original Contributions

The NeOn consortium is an inter-disciplinary team, and they need to have deliverables self-contained and comprehensible to all partners; therefore some deliverables thus necessarily include state-of-the-art surveys and associated critical assessment. Where there is no advantage in recreating such materials from first principles, the partners follow standard scientific practice and occasionally make use of their own pre-existing intellectual property in such sections. In the interests of transparency, we identified below the main sources of such pre-existing materials in this deliverable:

- Sections 3.3.1.6 and 3.3.1.6.1 contain material adapted from (Sabou, 2006)
- Section 4 contain material summarized from NeOn Deliverable D2.1.1.

Table of Contents

List of figures

1. Introduction

The goals of WP 5 are to provide two different methodologies, a methodology to support the collaborative construction and dynamic evolution of contextualized networked ontologies in distributed environments and a methodology for the development of large scale Semantic Web applications.

These methodologies will be described in detail in deliverables D5.4.1 and D5.5.1, due for months 24 and 30 respectively, in the middle of the project duration. Nevertheless, work has already started and guides have to be produced for other work packages. Therefore, the goal of WP5 is also to provide some previous guidance that can help both in the ontology development tasks of the use cases as in the development of the NeOn platform before having full-defined methodologies.

The deliverables produced in month 6 include the requirements of the NeOn toolkit (D6.1.1) and the requirements for the use cases (D7.1.1 and D8.1.1). These requirements clearly show a need for exchanging ontologies and data both in the case of the NeOn toolkit and in the case of ontology development in the use cases.

Although new protocols and techniques can be developed for exchanging these ontologies and data, there are existing alternatives that can be reused to minimize effort and to increase standardization in the NeOn products.

Therefore, this deliverable provides an enumeration of existing protocols and techniques for exchanging ontologies and information so the participants in the other work packages can analyze them either for reusing them or for learning from others experiences facing similar problems.

The enumeration of protocols and techniques presented in this deliverable do not pretend to be exhaustive, only the most relevant protocols and techniques for each topic have been selected. We also have selected from all the existing protocols and techniques those more relevant to the use cases and the NeOn toolkit: data access protocols, version management protocols, collaboration protocols, service access protocols, network communication protocols, notification protocols, and syndication protocols.

In this stage of the development of the methodologies for developing NeOn ontologies and NeOn toolkit, it is not possible to recommend from these protocols or techniques which ones to use in the current NeOn scenarios. Therefore, it is out of the scope of this deliverable analyzing them in depth and is in the hands of the implementers of the NeOn toolkit and of the use cases to choose or adapt the ones that better suit their scenario.

The deliverable is organized as follows:

- Chapter 2 identifies those requirements in D6.1.1 that require exchanging or sharing of information for being implemented inside the context of the NeOn Toolkit. The classification is the following:
  - Accessing remote services.
  - Accessing remote files.
  - Accessing relational databases.
  - Accessing XML sources.
  - Accessing ontological resources.
  - Having a directory of resources and services.
  - Having version management capabilities.
  - Having notification capabilities.
  - Having reasoning and querying capabilities.
  - Having remote plug-in installation capabilities.
- Chapter 3 describes some technical protocols or techniques that could solve the exchanging and sharing needs of the NeOn Toolkit.
- Chapter 4 presents a summary of social protocols through the literature on collaboration as a state of the art in social protocols for exchanging and sharing information.
- Finally, we conclude the document.

2. Sharing and exchanging needs

Starting from the requirements in D6.1.1 we will identify those that are related with the needs of sharing and exchange data or information. We will also explain briefly why for solving the specific requirement a protocol is needed. We start with the definition of sharing and exchange.

2.1 The concepts "share" and "exchange"

Since the role of this document is to analyse the state of the art (SoA) of protocols and techniques for exchanging and sharing information, we first define a protocol as "a set of rules determining the format and transmission of data".

*To support the **sharing** and reuse of formally represented knowledge among AI systems, it is useful to define the common vocabulary in which **shared** knowledge is represented. A specification of a representational vocabulary for a **shared** domain of discourse — definitions of classes, relations, functions, and other objects — is called an ontology* (Gruber, 1993). In this excerpt from a Gruber's research paper, we find the most repeated use of "share" in the Artificial Intelligence literature, i.e., to "share knowledge". To "share" is defined by the Merriam-Webster Online Dictionary[1] as "to have, to get or to use in common with another or others. Share usually implies that one as the original holder grants to another the partial use, enjoyment, or possession of a thing". If we share a parcel of knowledge or a domain of discourse, that means that we have some knowledge in common or that we pertain to the same domain as the others. To "share ontologies" or "share protocols" are as well common collocations in the AI domain, which means that we have or use the same, totally or partially, ontologies or protocols as the others. To put it in simple words: a piece of information is there and an A user can use it for its own purposes, but others users (B, C, etc.) can also take advantage of it for their own purposes. Or, as the second part of the definition of "share" assumes, the A user permits B and C users the partial use or the possession of that piece of information.

In the following cite by the same author, the term "share" is illustrated in a clear way: *Collaboration on the net is more than teleconferencing. Whenever we communicate or cooperate, we **share context and content**. On the net, however, some of the shared context and content is represented in the digital medium. As a consequence, the network infrastructure can give us more than a place to create and **share information**. It can also tell us what others are doing in that space; help us inform each other of relevant changes; track how information is found and used; and offer a context for discussions about **shared content** and how it is used. We don't just **share information**; **we share the process of accessing it, searching it, evaluating it, and using it*** (Gruber, 1995). This definition adds a new component, i.e., that every user (A, B, and C) contributes with a piece of information and put it at the disposal of the rest, who can benefit from it.

The word "exchange" is very often combined with "information", "data", as in the following definition of the XML/RDF languages: *(…) XML/RDF were investigated as a part of the evaluation effort because of the significance of the web and web-based applications. It is clear that the web is rapidly becoming the primary method for the **exchange of information and data**, and that XML is currently the leading candidate for a generic language for the **exchange of semi-structured objects** (…)*[2].The term "exchange" is defined by the MWO as "to part with, give, or transfer in consideration of something received as an equivalent" or "to have replaced by other merchandise", among other meanings. To the question of *Why do we create ontologies?* Gruber answers: (…) *to enable data exchange among programs*. Then, if A, B and C want to exchange something (data, for example), A gives data to B and receives data from B as consideration; and B gives data to C and receives it from B, and so on. One can also infer from Gruber's statement that we need a common platform or space (or even specific requirements) so that the exchanging process can

[1] Merriam-Webster Online: http://www.m-w.com/

[2] An Evaluation of Ontology Exchange Languages for Bioinformatics: http://xml.coverpages.org/OntologyExchange.html

take place successfully, what means that we need to "share" the same bases that enable the "exchanging".

Finally, we want to consider the next example, in which "share" and "exchange", are both combined with the word "knowledge": *Short for **K**nowledge **Q**uery and **M**anipulation **L**anguage. KQML is a language and* protocol *for **exchanging information and knowledge**. (…) It is both a message format and a message-handling protocol that supports* run-time ***knowledge sharing*** *among* agents*. KQML can be used as a language for an application program to interact with an intelligent system or for disparate intelligent* systems *to **share knowledge** in support of cooperative problem solving. (…)*[3]. In this sense, we can even understand a chronological relation between both concepts, since we need to "exchange knowledge" in the first place, so that we have "the same knowledge", in order to "share it" afterwards.

As a conclusion, we could state that both concepts are very closely related. The clearest difference between them is that "exchange" implies a transmission of something, going in one or the other direction (A gives to B or B receives from A, or both), and "share" means that all partners move in the same direction, making use of one thing or a set of things which have been placed in the centre, to which all have access and can even use at the same time. We could represent both concepts in the following schematic way:

2.2 Technical Requirements

In this section we present a table in which we have identified those requirements identified in D6.1.1 that are related somehow with protocols and techniques for sharing and exchanging information. We have made a classification according to different needs of exchanging and sharing. Some requirements are related to more than one need because we can infer different needs from them at the same time. The classification is the following:

- Accessing remote services.
- Accessing remote files.
- Accessing relational databases.
- Accessing XML sources.
- Accessing ontological resources.
- Having a directory of resources and services.
- Having version management capabilities.
- Having notification capabilities.
- Having reasoning and querying capabilities.
- Having remote plug-in installation capabilities.

[3] Webopedia, the Online Encyclopaedia for Computer Technology: http://www.webopedia.com/TERM/K/KQML.html

To ease the reading of this section, we created for each of the above categories a table where:

- Column “Req#” presents the requirement number already identified in D6.1.1.
- Column “Title” gathers a short description of the requirement identified in D6.1.1.
- Column “Description” includes the full description of the requirement identified in D6.1.1.
- Column “Relation with protocols” explains why for solving the specific requirement a protocol is needed.

2.2.1 Requirements for accessing remote services

Req #	Title	Description	Relation with protocols
2.1.1.4	Inter-layer communication mechanism	The NeOn architecture layers shall be at the same time service functionality clients and producers for each other. Natural flow of architecture service invocation goes from the higher level layers to the lower level layer. On the other hand, push services, e.g. information refresh may be triggered upwards by lower level layers, in this case, the distributed repository layer. The necessary infrastructure, e.g. a service bus, shall be implemented to allow this kind of service invocation	The services in every layer that are accessible in the network must be implemented to allow remote invocation.
2.1.1.6	Annotation service	The second layer of the NeOn architecture shall include annotation mechanisms and associated services	The associated services must be available in the network according to some protocol.
2.1.1.7	Text mining service	The second layer of the NeOn architecture shall include text mining services for human language information retrieval	The text mining services must be available in the network according to some protocol.
2.1.1.8	Summarization service	The second layer of the NeOn architecture shall include an informational service briefing ontology information	The informational service must be available in the network according to some protocol.
2.1.1.9	Context service	The second layer of the NeOn architecture shall include a service to gather and process context information for later exploitation	The service for gather and process context information must be available in the network according to some protocol.
2.1.1.10	Reasoning service	The second layer of the NeOn architecture shall include reasoning mechanisms and associated services.	The reasoning services must be available in the network according to some protocol.
2.1.1.11	Query service	The second layer of the NeOn architecture shall include query mechanisms and associated services.	The querying services must be available in the network according to some protocol.
2.1.1.12	Question formulation Service	The NeOn toolkit shall provide a question formulation service that eases query formulation by allowing more natural ways of expressing queries, e.g. natural language	The question formulation service must be available in the network according to some protocol.
2.1.1.13	Provenance service	The NeOn middleware layer shall include a service which keeps track of the precedence of ontologies, i.e. how, when, and by whom ontologies evolve across their lifecycle.	The provenance service must be available in the network according to some protocol.
3.2.1.1	Distributed Repository access API	NeOn distributed repository API shall provide access to knowledge contained in the repository	The NeOn distributed repository API must be accessible in the network by some kind of service.
3.2.5	Middleware layer accessible by an API	The middleware layer API shall provide programmatic access to the NeOn distributed components	The NeOn middleware layer API must be accessible in the network by some kind of service.

2.2.2 Requirements for accessing remote files

Req #	Title	Description	Relation with protocols
2.1.1.5	Support for integration of heterogeneous information sources	The following types of distributed information resources shall be integrated in the NeOn knowledge model: - Unstructured content, e.g. text. - Structured knowledge with no clear semantics, e.g. DBs, catalogues. - Formal knowledge (ontologies and data) - Semantic annotations	Some information sources use to be expressed in files (e.g. text), so files should be accessible remotely.
2.1.1.15	Stand-alone server	Ontologies and data shall be loaded in a stand-alone server, based on OntoStudio and OntoBroker, which implements the backend for the reasoning service and the ontology editor.	It should be possible to obtain the ontologies and data in a file format from a remote server.

2.2.3 Requirements for accessing relational databases

Req #	Title	Description	Relation with protocols
2.1.1.5	Support for integration of heterogeneous information sources	The following types of distributed information resources shall be integrated in the NeOn knowledge model: - Unstructured content, e.g. text. - Structured knowledge with no clear semantics, e.g. DBs, catalogues. - Formal knowledge (ontologies and data) - Semantic annotations	The NeOn toolkit will access to relational databases, so a protocol for accessing relational databases must be used.

2.2.4 Requirements for accessing XML Sources

Req #	Title	Description	Relation with protocols
2.1.1.5	Support for integration of heterogeneous information sources	The following types of distributed information resources shall be integrated in the NeOn knowledge model: - Unstructured content, e.g. text. - Structured knowledge with no clear semantics, e.g. DBs, catalogues. - Formal knowledge (ontologies and data) - Semantic annotations	The NeOn toolkit will access to XML sources, so a protocol for accessing XML sources must be used.

2.2.5 Requirements for accessing ontological resources

Req #	Title	Description	Relation with protocols
2.1.1.5	Support for integration of heterogeneous information sources	The following types of distributed information resources shall be integrated in the NeOn knowledge model: - Unstructured content, e.g. text. - Structured knowledge with no clear semantics, e.g. DBs, catalogues. - Formal knowledge (ontologies and data) - Semantic annotations	For accessing ontological resources some protocol is needed.
2.1.1.15	Stand-alone server	Ontologies and data shall be loaded in a stand-alone server, based on OntoStudio and OntoBroker, which implements the backend for the reasoning service and the ontology editor.	This stand-alone server based on OntoStudio and OntoBroker must be covered with services that implement some protocols for accessing the ontological resources contained in it.

2.2.6 Requirements for having a directory of resources and services

Req #	Title	Description	Relation with protocols
2.1.1.1	Distributed repository layer	Information resources will be potentially stored in a distributed repository managed by this architecture layer. NeOn shall access these resources transparently to their location	The distributed repository must have a directory that describes the resources in it. This directory must be accessible in the network, according to some protocol.
2.1.1.2	Middleware layer	This middleware layer between the NeOn toolkit and the distributed repository shall implement a number of services which allow users to exploit the information stored in the distributed repository	For exploit the information stored in the distributed repository first an identification of the resources in it must be done. So the middleware layer must connect with a directory according to some protocol.
3.2.1	Transparent access to information resources	NeOn shall allow access to ontological and non ontological information pieces transparently from their location	A federation of registries or repositories can give us the mechanisms needed to abstract the resources independently of their location. There are protocols that cover this functionality.
3.2.1.2	Resource transparent storage	NeOn shall provide transparent access to resource storage. The NeOn Distributed Repository shall internally manage the physical location where the resource is actually stored.	Directories internally manage the physical location where a resource is actually stored itself.
3.2.1.3	Resource transparent load	NeOn shall provide a virtual file system which abstracts away the actual resource location, offering a virtual unique storage space	A directory can be view as a virtual file system.
3.2.2	Access to the different kinds of knowledge	NeOn shall provide access to the main four different kinds of information i.e. unstructured content, structure knowledge without a clear semantics, formal knowledge (ontologies and data), and semantic annotations	The directory must have descriptions of resources that belong to this main four different kind of information.

2.2.7 Requirements for having version management capabilities

Req #	Title	Description	Relation with protocols
3.2.11.1.7	Ontology collaborative development	NeOn shall allow collaborative ontology development	Protocols that support collaborative development are needed.
3.2.11.1.7.1	Adoption of CVS development model	NeOn shall adopt the CVS metaphor for collaborative work on shared ontological resources	Protocols that have at least the same behaviour as CVS with respect to collaborative work are needed.
3.2.11.1.7.1.1	Synchronization of ontology concurrent updates	NeOn shall allow synchronization of ontologies edited by multiple users	Protocols that allow synchronization of resources edited by multiple users are needed.
3.2.11.1.7.1.3	Ontology versioning	NeOn shall provide support for ontology versioning based on the CVS metaphor	Protocols that have at least the same behaviour as CVS with respect to versioning are needed.
3.2.11.1.7.1.3.1	Global awareness of local ontology versioning	NeOn shall ease the adoption of new versions of an ontology in an already existing network of ontologies	Protocols that support versioning are needed.

2.2.8 Requirements for having notification capabilities

Req #	Title	Description	Relation with protocols
3.2.11.1.7	Ontology collaborative development	NeOn shall allow collaborative ontology development	When a change is made there should be capabilities for notify the changes made to all the stakeholders. There are a lot of protocols for notifying.

2.2.9 Requirements for having reasoning and querying capabilities

Req #	Title	Description	Relation with protocols
2.1.1.15	Stand-alone server	Ontologies and data shall be loaded in a stand-alone server, based on OntoStudio and OntoBroker, which implements the backend for the reasoning service and the ontology editor.	Protocols and languages for making queries to the reasoners are needed.
3.2.12	Reasoning with networked ontologies	The NeOn backend shall support reasoning capabilities using the aggregated knowledge contained in a set of networked ontologies	The queries must be done at the same time to a set of ontologies.

2.2.10 Requirements for having remote plug-in installation capabilities

Req #	Title	Description	Relation with protocols
2.1.5.1	Service oriented architecture	NeOn shall offer a service oriented interface in order to ease the inclusion of new components into any of the three architecture layers	The NeOn toolkit must implement protocols that allow the remote installation of new features.

2.3 Requirements related to collaborative aspects

Req #	Title	Description	Relation with protocols
2.1.1.1	Distributed Repository Layer	Information resources will be potentially stored in a distributed repository managed by this architecture layer. NeOn shall access these resources transparently to their Location.	This layer should manage multiple and concurrent access to the repository
2.1.1.2	Middleware layer	This middleware layer between the NeOn toolkit and the distributed repository shall implement a number of services which allow users to exploit the information stored in the distributed repository.	Concurrency should be supported by this layer.
2.1.1.3	NeOn toolkit layer	User frontends and high level services shall be supported by this layer.	This layer should support social protocols associated with services for collaborative workflows.
2.1.1.4	Inter-layer communicatio n mechanism	The NeOn architecture layers shall be at the same time service functionality clients and producers for each other. Natural flow of architecture service invocation goes from the higher level layers to the lower level layer. On the other hand, push services, e.g. information refresh may be triggered upwards by lower level layers, in this case, the distributed repository layer. The necessary infrastructure, e.g. a service bus, shall be implemented to allow this kind of service invocation.	Protocols for supporting concurrency should be implemented.
2.1.2.1	NeOn Editor	The NeOn toolkit layer shall provide an ontology editor based on OntoStudio.	NeOn editor should support collaborative editing.
2.1.2.2	NeOn Browser	NeOn shall allow ontology browsing. Might be integrated with the NeOn editor.	NeOn Browser should support multiple users browsing the same ontology.
2.1.2.3	NeOn GUI	NeOn shall offer a user-friendly GUI to functionalities and services	NeOn GUI should provide users with easy access to collaborative environments.
2.1.4.1	Compatibility with several platforms	The NeOn toolkit shall be compatible with several platforms.	Protocols which support configurations of users collaborating from different platforms should be implemented.
2.4.1.4	Testing	Unit and functional tests shall be used to check the software.	Unit and functional tests should include collaborative aspects.
2.4.1.4.1	Stress testing	Stress testing using large, real-world ontologies	Also include large, real-world teams.
3.2.1	Access to Distributed repository	Transparent access to information resources.	Protocols for concurrent access has to be considered.
3.2.4	Semantic annotation	NeOn shall allow to Annotate information sources	Support for collaborative annotation should be implemented.
3.2.8	User profiling	NeOn shall accumulate information about user profiles during system interaction and provide a customized set of functionalities according to that profile.	Information about collaboration between users should be used for user profiling.
3.2.11	Lifecycle support for ontology development	NeOn shall support knowledge acquisition and ontology development.	NeOn ontology development is meant to be collaborative. Hence, support for coordination, cooperation, and in generl collaborative protocols should be provided.

3. Technical Protocols and Techniques for Exchanging and Sharing Information

In this chapter we include for each need identified a set of protocols, languages, formats and standards that aid to give a solution to the concrete need. The following table makes a relation of each need with the correspondent set of protocols.

Need	**Set of protocols and techniques**
Accessing remote services	Access remote services
Accessing remote files	Access remote files
Accessing relational databases	Access relational databases
Accessing XML sources	Access XML sources
Accessing ontological resources	Access ontological resources
Having a directory of resources and services	Access service directories and registries
Having version management capabilities	Version management protocols
Having notification capabilities	Notification and syndication protocols
Having reasoning and querying capabilities	Query languages Agent communication languages
Having remote plug-in installation capabilities	Remote plug-in installation protocols, standards and platforms

Also, for this deliverable we considered relevant P2P protocols since the applications and technologies developed within NeOn for the management of Networked Ontologies and Metadata (e.g. Oyster, KaonP2P) are distributed systems that rely on some P2P technology/protocol. The last section talks about P2P protocols. In this chapter we make a selection of technologies that have or had achieve success inside the software development community or those new technologies that provide new value added function with respect to the previous.

3.1 Data access protocols

3.1.1 Access remote files

In this section we include a list of protocols that allow for accessing remote files. These protocols are potentially relevant for NeOn. There are a lot of protocols that let networked applications access remote files. In this deliverable we make a selection of historical and well know technologies commonly used in the Internet and several new technologies such as WebDAV or grid protocols.

3.1.1.1 GridFTP

GridFTP is a protocol extension of the file transfer protocol (Postel, et al., 1985) for grid environments (Bester, et al., 2003). GridFTP is a grid-enabled implementation of the FTP protocol that provides high-performance, secure and reliable data transfer functionalities based also in standards (Horowitz, et al., 1997) (Hethmon, et al., 1998), which is optimized for high-bandwidth wide-area networks. The current GridFTP protocol specification is now a "proposed recommendation" document in the Open Grid Forum.

GridFTP uses basic Grid security on both control (command) and data channels. Other features include multiple data channels for parallel transfers, partial file transfers, third-party (direct server-to-server) transfers, reusable data channels, and command pipelining.

This grid-based file transfer protocol is used for moving massive amounts of data between remote entities, that provides an added value to this transfers leveraging on the grid infrastructure, and could be of interest for the NeOn project.

FTP Extensions

The GridFTP specification extends the traditional FTP protocol defined in (Postel, et al., 1985) providing two new ways of interaction between servers and clients:

The *parallel data transfer* method allows several data transfers between different entities to happen simultaneously, i.e. to transfer two different files at the same time.

The *striped data transfer* method allows several data transfers of the same data source to happen simultaneously, i.e. to transfer different parts of the same file at the same time.

In order to realize these new transfer methods, the specification defines new commands, options, features and a new data transfer mode.

New options and features are provided for fine tuning these commands, and for enabling the *parallel* and *striped* data transfer methods.

In order to realize the striped and parallel data transfer methods a new data transfer mode is defined: *the extended transfer mode*. This transfer mode supports out-of-sequence data delivery, and partial data transmission per data connection. The extended block mode extends the predefined FTP block mode header to provide support for these, as well as large blocks and end-of-data synchronization.

3.1.1.2. ByteIO

The ByteIO specification (Morgan, et al., 1995) describes a set of interfaces that provide users with a concise, standard way of representing data resources as POSIX-like files. This provides a level of access transparency which is important in a distributed system. Clients can leverage these interfaces to provide users with a convenient way of interacting with data in a way which does not require them to adapt to a new model of data access or, in some cases, does not even require them to realise that their data resources are indeed on the Grid.

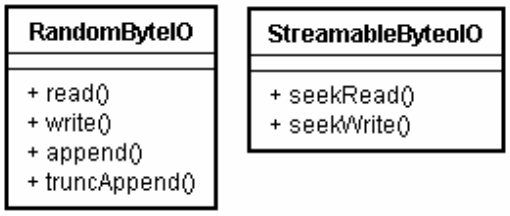

Figure 1: ByteIO interfaces, taken from (Berry, et al., 2006)

NeOn I/O components could leverage these interfaces to seamlessly accessing remote files in a fine-grained way without explicitly having to deal with communication details such as establishing a connection, keeping it alive and closing it when finished.

3.1.1.3. WebDAV

In the context of NeOn ontological resources might be spread around different locations. Because the development of these resources cannot be controlled in a central place or in a central controlling component it is important to have systems that maintain information about the history and current state of a resource. WebDAV is a protocol that provides such means and thus it is important in the context of

- Requirements for accessing remote files,
- Requirements for accessing ontological resources,
- Requirements for having a directory of resources and services,
- Requirements for having version management capabilities.

WebDAV (Web-based Distributed Authoring and Versioning) is a general-purpose protocol for accessing documents on a remote server. As a standard for collaborative web authoring, WebDAV is based on HTTP. Common procedures such as the creation, retrieval and deletion, as well as the organization of documents are supported. Users can edit web resources with the same ease as they can edit resources in a local directory. Example application areas are Instant Web publishing, Workgroups, Content Management, and Strategic File Management.

As it is considered as an extension HTTP the WebDAV standard is owned by the Internet Engineering Taskforce (IETF). It consists of the following parts:

- HTTP Extensions for Distributed Authoring – WebDAV[4].
- Versioning Extensions to WebDAV: DeltaV protocol[5].
- Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol[6].

[4] RFC 2518: HTTP Extensions for Distributed Authoring - WebDAV

[5] RFC 3453: Versioning Extensions to WebDAV: DeltaV protocol

[6] RFC 3744: Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol

- WebDAV SEARCH.

The WebDAV protocol has the following important functionality:

- **Collections.** In contrast to HTTP's URL resource access of a single file, WebDAV offers the concept of organizing any number of individual resources into WebDAV collections. A collection can be compared to a file system directory, providing efficient means for accessing and structuring resources. Managing collections and resources of remote repositories opens a vast field of new possibilities for authoring documents and tools in network environments.
- **Locking.** WebDAV offers various concepts for collaboration on resources which may be accessed by any number of users simultaneously.
- **Properties.** To help to make resources more valuable, WebDAV allows properties, or "metadata", to be assigned to any type of data. This is where XML's inherent extensibility is particularly suited. Here again, the possibilities exceed the scope of helpful retrieval mechanisms, or references, by far, and leave adequate room for further developments.
- **Security (ACL).** "In distributed authoring scenarios resources may be accessible by multiple principals. To control how these principals can access and alter a resource, access controls are needed. These controls define what actions a particular principals is allowed to exercise on a particular resource."[7]
  The ACL standard defines privileges on a resource basis. Each resource can be individually secured by different access rights.
- **Versioning and Configuration Management (DeltaV).** The DeltaV protocol is an extension to the WebDAV protocol offering remote versioning and configuration management of documents stored in a web server. It offers a quite comprehensive set of versioning features.
  - explicit versioning
  - automatic versioning for versioning-unaware clients
  - version history management
  - workspace management
  - baseline management
  - activity management
  - URL namespace versioning

  As shown in Figure 2 it defines four implementation levels, implementing 11 features. There is a core package which implements base functionality for version control of resources and reporting resource changes. Basic functionality for labelling, updating, checking in and out resources and maintaining their version history are provided in intermediate layers. More enhanced functionality like merging different versions of resources are labelled as advanced features in a top layer software component.

[7] WebDAV Access Control Goals: http://www.webdav.org/acl/goals/draft-ietf-webdav-acl-reqts-00.txt

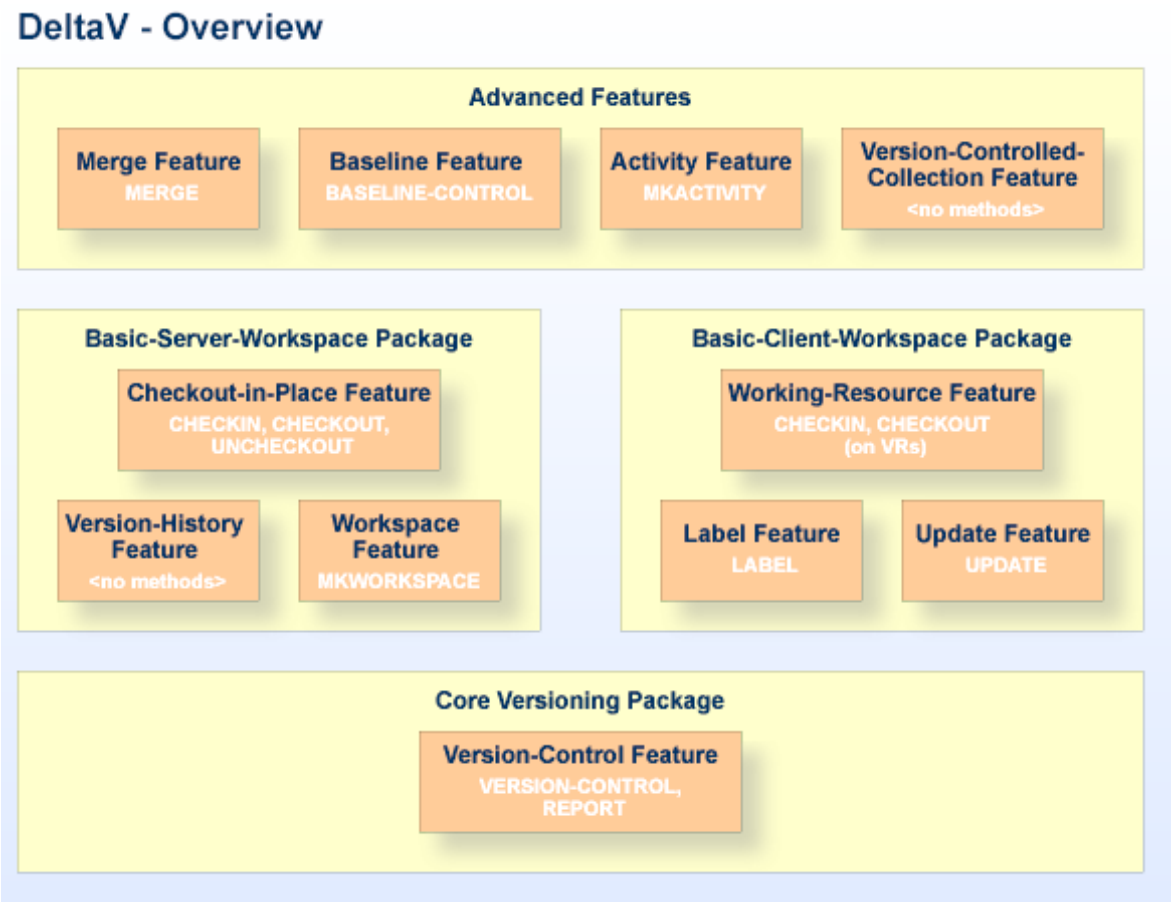

Figure 2: DeltaV – Overview

Interestingly the popular advanced version control system subversion (SVN) uses a subset of the DeltaV protocol.

- **WebDAV SEARCH.** The WebDAV SEARCH standard defines a WebDAV method and a query language to search with a WebDAV namespace for resources based on Boolean expressions on properties and content.

3.1.2 Access relational databases

In this section we include a list of standards that allow for accessing relational databases. ODBC is historically the first standard for giving a uniform access to databases management systems and it is well established inside the Microsoft oriented programmers community. JDBC is the analog standard inside the Java community.

3.1.2.1 WS-DAIR

The *"Web Services Data Access and Integration – The Relational Realization (WS-DAIR) Specification, Version 1.0"* (Antonioletti, et al., 2006) is a realization of the WS-DAI base specification aimed to providing data access to relational data resources, that is, the realization defines a web service based mechanism for accessing relational databases. The specification is being standardized in the Grid community, specifically in the Open Grid Forum.

Organization

The WS-DAIR specification provides the means for querying remote relational databases using SQL and exploiting the results in the client side. The specification defines a set of interfaces for dealing with the results of such queries with different granularity, as it is shown in Figure 3.

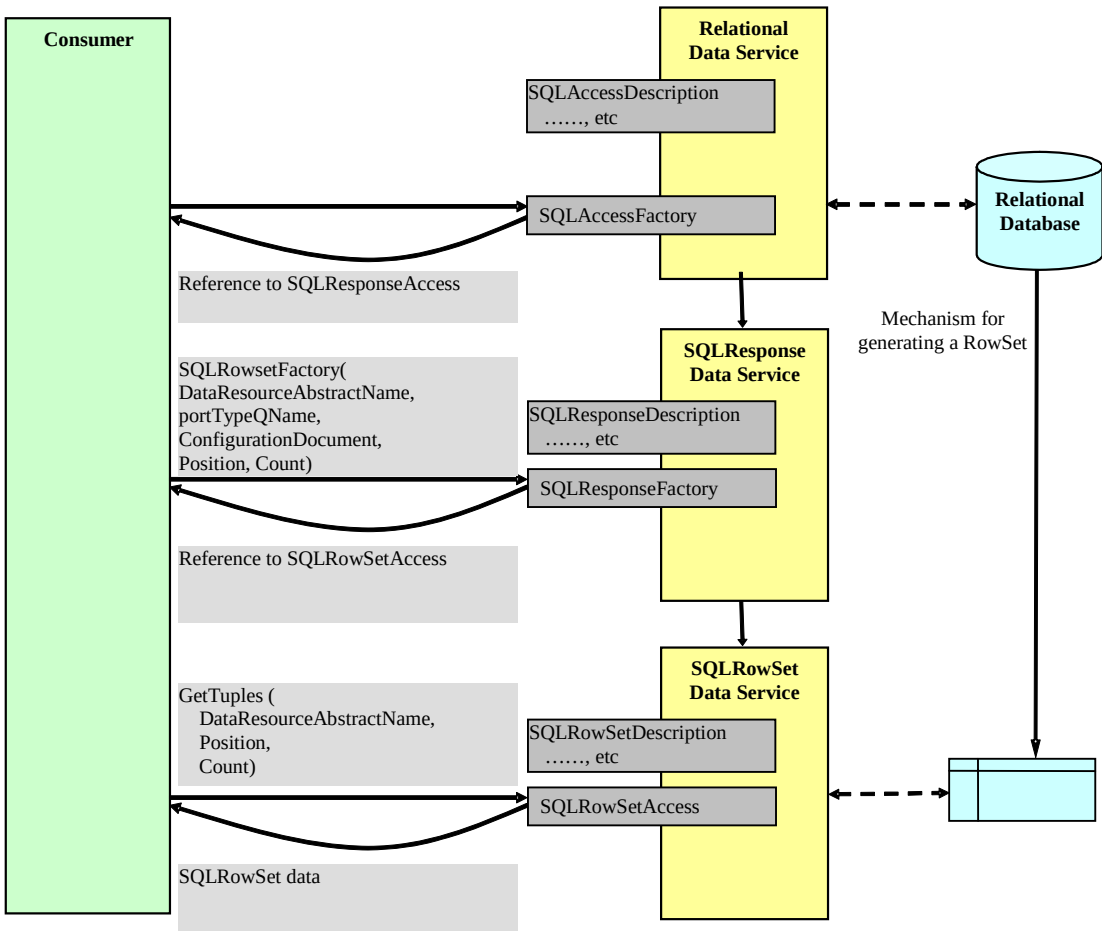

Figure 3: Simple WS-DAIR usage example, taken from (Antonioletti, et al., 2006)

The specification provides three access alternatives for accessing the results of an SQL query that has been pushed to remote relational databases. The first alternative relies in the SQLAccess interface, which provides direct data access functionalities for accessing relational databases by means of SQL queries, that is, the results sets obtained from the query are directly returned to the client embedded in the response.

The second alternative consists in providing direct data access to rowsets of a result set obtained from a query, that is, instead of returning the complete result set to the client (as in the previous alternative), the client browses the result set as needed, having the rowset as the data transfer unit. In order to do so, the client has to use the indirect data access capabilities offered by the SQLAccessFactory interface, which provides the means for making available the result set through services which implement the SQLResponseAccess interface. This latter interface provides direct data access functionalities to the result set.

The last alternative follows the same scheme of the second one, but this time at rowset level: instead of direcly accessing to a rowset, providing direct data access to the different columns of a rowset. The client would use the SQLResponseFactory interface for providing indirect data access to a rowset, making it available through a service which implements the SQLRowsetAccess interface, an interface that provides direct data access functionalities to the columns of a rowset.

Figure 4 shows the interfaces defined in the WS-DAIR realization, and the messages and properties provided by each one of them.

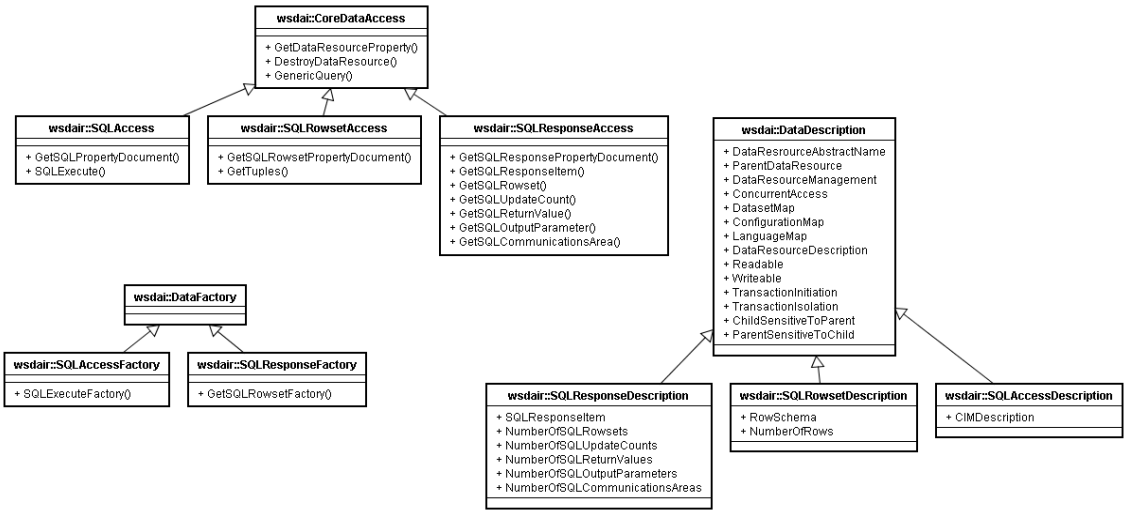

Figure 4: WS-DAIR interfaces, taken from (Berry, et al., 2006)

3.1.2.2. JDBC (Java Database Connectivity)

JDBC is an API that allows the users to perform operations to databases using the Java programming language. JDBC allows the users to do these operations independently from the language in which the users have to access the database management system. The JDBC API offers an API for application writers and a lower level API for driver writers. Figure 5 shows how the applications can access the databases via the JDBC API using pure Java JDBC drivers. The left side of Figure 5 shows how a driver converts directly the JDBC calls into the network protocol used by the DBMS. The right side of the figure converts the calls into a middleware protocol which is translated to a DBMS protocol by this middleware. Applications can also connect to databases using ODBC drivers and other existing database drivers via JDBC.

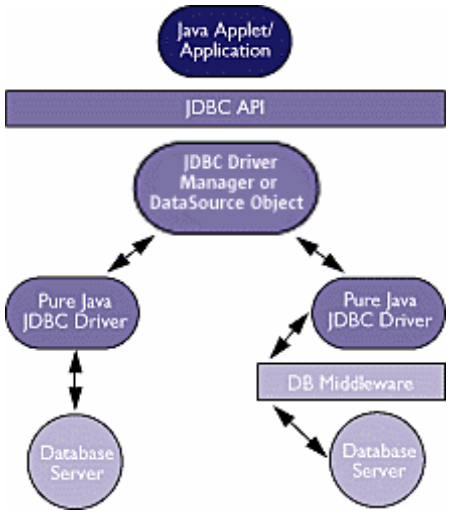

Figure 5: JDBC Architecture

3.1.2.3 ODBC (Open Database Connectivity)[8]

Open Database Connectivity provides an API method for using database management systems. The goal of ODBC is to provide access to those database management systems independently of the programming languages, operating systems and database systems. A user is able to write a program that access to databases without knowing the particularities of the DBMS or the system in which is running the server. An implementation of ODBC contains the applications that will access to the database, a core library that acts as interpreter between the applications and the database management system and one or more database drivers that allow the bridges to access the databases. ODBC also allows accessing other types of data sources rather than databases, e.g. Excel files. Figure 6 depicts the ODBC architecture. In this figure an application can access through the ODBC driver to different DBMS. The ODBC driver connects to several controllers that will access to the databases.

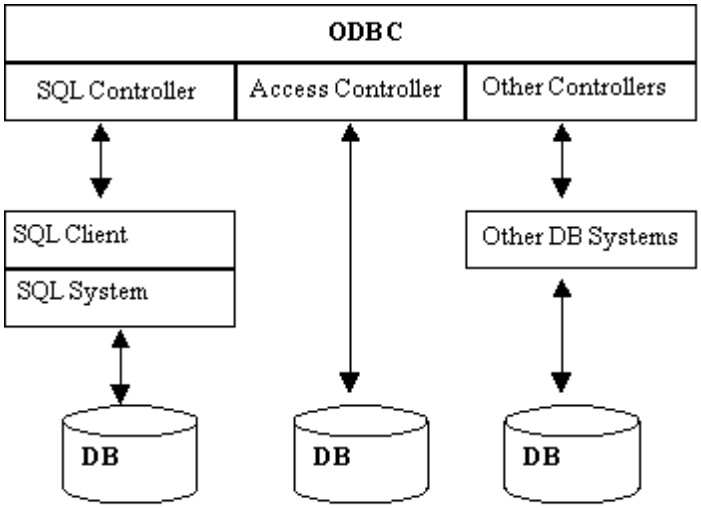

Figure 6: ODBC Architecture

3.1.2.4 JDO

Introduction

In object-oriented programming, when utilizing Java, data is transient, meaning that it usually only resides in memory and will be destroyed when the application ends. In developing many applications, this is a problem because a requirement or goal is for the data to remain persistent, meaning continuing to exist outside of the (single) application instance. Traditional data stores such as databases, files systems, and so forth are generally used for persisting data. In relational databases, data is stored in tables containing columns. In Java, an object-oriented language, data is manipulated as objects. Clearly defined relationships between Java objects and database structures do not exist. As a result, programmers must create relationships between objects being utilized in their applications and the data stores the data actually resides in.

Java Data Objects (JDO) is a **specification** developed to enable transparent persistence of Plain Old Java Objects (POJO[9]). It is a high-level API that allows applications to store Java objects in a transactional data store by following defined standards. This API can be used to access data on platforms such as desktops, servers, and embedded systems. JDO provides a means for treating

[8] Available from: http://www.uv.es/jac/guia/gestion/gestion3.htm

[9] A term coined by Martin Fowler, Rebecca Parsons, and Josh MacKenzie in September 2000

data stored in a relational database as Java objects and provides a standard means to define transactional semantics associated with these objects.

JDO is being developed as a Java Specification Request in the Java Community Process. The original JDO 1.0 is JSR-12[10] and the current JDO 2.0 is JSR-243 [11]

Architecture

The high-level JDO API is designed to provide a transparent interface for developers to store data, without having to learn a new data access language (such as SQL) for each type of persistent data storage. JDO can be implemented using a low-level API (such as JDBC) to store data. It enables developers to write Java code that transparently accesses the underlying data store, without using database-specific code.

The two main objectives of the JDO architecture, which is shown in Figure 7, are to provide Java application developers a transparent Java technology-centric view of persistent information and to enable pluggable implementations of data stores into application servers.

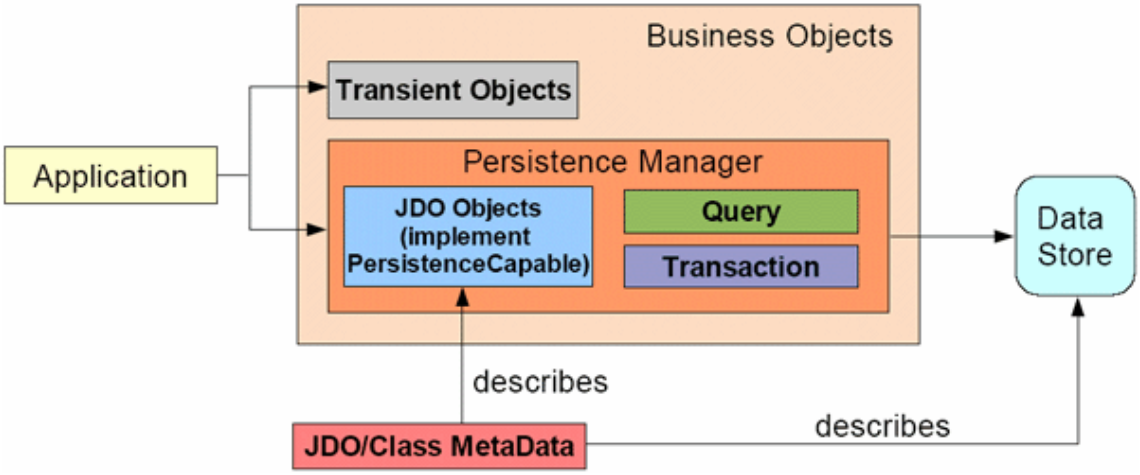

Figure 7: JDO Architecture

Interfaces are defined for the user's view of persistence:

- PersistenceManager: the component responsible for the life cycle of persistent instances, Query factory, and Transaction access
- Query: the component responsible for querying the data store and returning persistent instances or values
- Transaction: the component responsible for initiating and completing transactions

It is important to note that JDO does not define the type of data store: You can use the same interface to persist your Java technology objects to a relational database, an object database, XML, or any data store.

[10] http://www.jcp.org/en/jsr/detail?id=12

[11] http://www.jcp.org/en/jsr/detail?id=243

The benefits of using JDO are the following:

- Portability. Applications written using the JDO API can be run on multiple implementations available from different vendors without changing a single line of code or even recompiling.
- Transparent database access. Application developers write code to access the underlying data store without any database-specific code.
- Ease of use. The JDO API allows application developers to focus on their domain object model (DOM) and leave the details of the persistence to the JDO implementation.
- High performance. Java application developers do not need to worry about performance optimization for data access because this task is delegated to JDO implementations that can improve data access patterns for best performance.
- Integration with EJB. Applications can take advantage of EJB features such as remote message processing, automatic distributed transaction coordination, and security using the same DOMs throughout the enterprise.

3.1.2.5 Hibernate

Introduction

Hibernate[12] is an **object-relational mapping** (ORM) **solution** for the Java language: it provides an easy to use framework for mapping an object-oriented domain model to a traditional relational database. Its purpose is to relieve the developer from a significant amount of common data persistence-related programming tasks.

It also provides the data query and retrieval facilities that significantly reduce the development time.

Features

1. Hibernate provides three full-featured query facilities: Hibernate Query Language (HQL), the newly enhanced Hibernate Criteria Query API, and enhanced support for queries expressed in the native SQL dialect of the database.
2. Filters for working with temporal (historical), regional or permissioned data.
3. Enhanced Criteria query API: with full support for projection/aggregation and subselects.
4. Runtime performance monitoring: via JMX or local Java API, including a second-level cache browser.
5. Hibernate is Scalable: Hibernate is very performant and due to its dual-layer architecture can be used in the clustered environments.
6. Less Development Time: Hibernate reduces the development timings as it supports inheritance, polymorphism, composition and the Java Collection framework.
7. Hibernate XML binding enables data to be represented as XML and POJOs interchangeably.
8. Hibernate is Free under LGPL: Hibernate can be used to develop/package and distribute the applications for free.

Architecture

The Figure 8 shows that Hibernate is using the database and configuration data to provide persistence services (and persistent objects) to the application.

[12] http://www.hibernate.org/

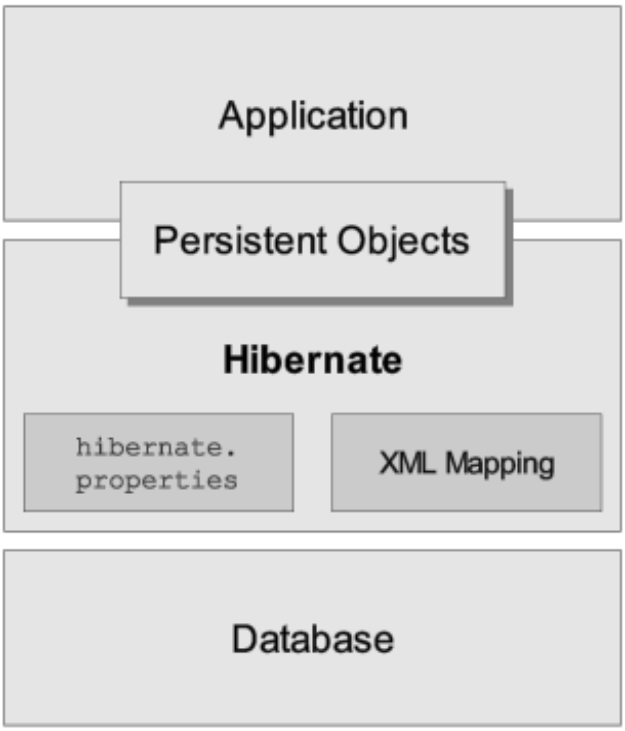

Figure 8: Hibernate architecture

To use Hibernate, it is required to create Java classes that represents the table in the database and then map the instance variable in the class with the columns in the database. Then Hibernate can be used to perform operations on the database like select, insert, update and delete the records in the table. Hibernate automatically creates the query to perform these operations.

Hibernate architecture has three main components:

- **Hibernate Connection** management service provides efficient management of the database connections. Database connection is the most expensive part of interacting with the database as it requires a lot of resources of open and close the database connection.
- **Transaction management** service provides the ability to the user to execute more than one database statements at a time.
- **Object relational mapping** is technique of mapping the data representation from an object model to a relational data model. This part of hibernate is used to select, insert, update and delete the records form the underlying table.

Hibernate is very good tool as far as object relational mapping is concern, but in terms of connection management and transaction management, it is lacking in performance and capabilities. So usually hibernate is being used with other connection management and transaction management tools. For example apache DBCP is used for connection pooling with the Hibernate.

Hibernate provides a lot of flexibility in use. It is called "Lite" architecture when we only use the object relational mapping component. While in "Full Cream" architecture the entire three components (Object Relational mapping, Connection Management and Transaction Management) are used.

3.1.3 Access XML Sources

XML is a markup language for documents containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure. A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents[13].

Several technologies for accessing and managing the XML content are described in this section.

3.1.3.1 WS-DAIX

The *"Web Services Data Access and Integration – The XML Realization (WS-DAIX) Specification, Version 1.0"* (Hastings, et al., 2006) is a realization of the WS-DAI base specification focused at providing data access to XML data resources using standard XML query languages.

Organization

The specification provides two sets of functionalities:

- **Collection management:** provides mechanisms for managing collections of XML documents (schemas and instance documents). It is possible to organize these collections hierarchically.

  These collections of documents can be used as sources in the functionalities provided by the query-based access area.

- **Query-based access:** WS-DAIX defines a set of interfaces that provide functionalities which support the evaluation of XPath, XQuery and XUpdate queries across an XML resource or a collection of resources.

  On the one hand, the XUpdateAccess, XQueryAccess and XPathAccess interfaces provide direct data access to collections of XML documents, by means of XUpdate, XQuery and XPath queries respectively. On the other hand, the realization also provides indirect data access to the very XML resources: the XUpdateFactory, XQueryFactory and XPathFactory make available the results of XUpdate, XQuery and XPath queries by means of the services which implement the XMLSequenceAccess interface. This latter interface provides finer-grainer direct data access to those results.

Figure 9 shows the interfaces defined in the WS-DAIX extension, and the messages and properties provided by each one of them.

[13] O'Reilly XML from the inside out: http://www.xml.com/pub/a/98/10/guide0.html?page=2#AEN58

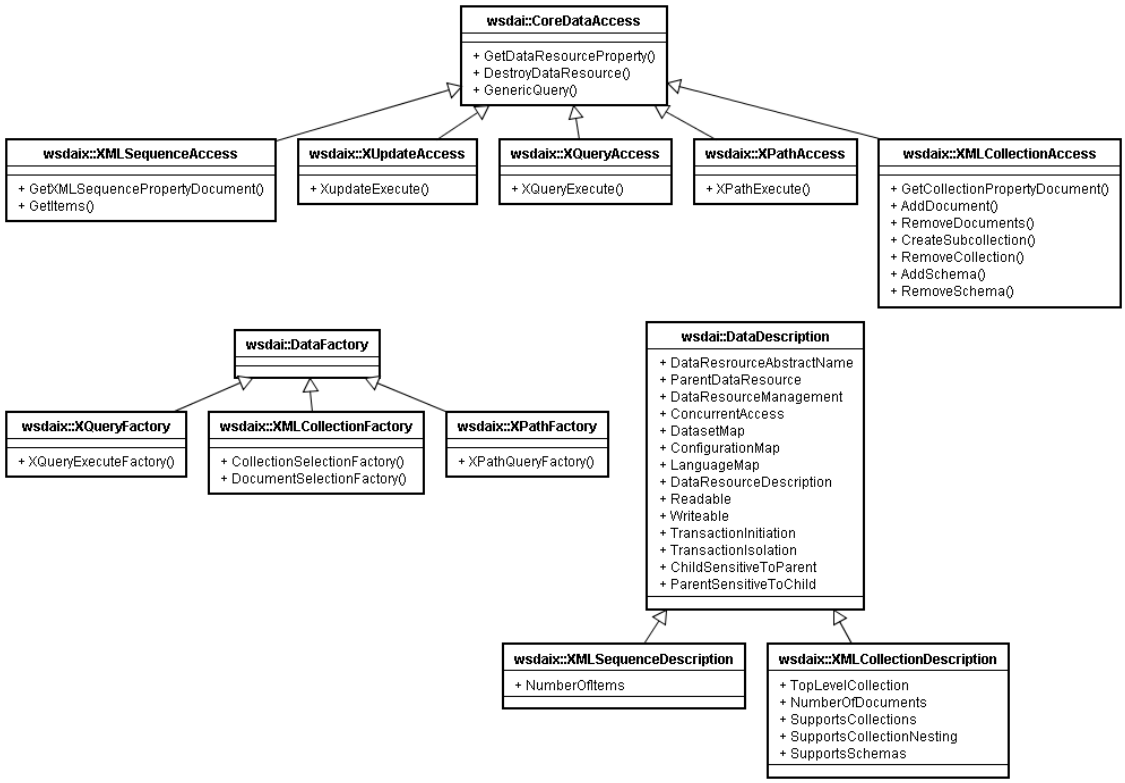

Figure 9: WS-DAIX interfaces, taken from (Berry, et al., 2006)

3.1.3.2 XML-Schema

In the context of NeOn where loosely coupled services interact as web services the definition of these services plays an important role. One part of these definitions is the structure of the exchanged content. As the content is typically passed as XML, the corresponding definition language XML Schema is important.

Also for many ontology languages there is a serialization, i.e. storage format, for ontologies in XML. To exchange ontologies that are stored in such a format, it is essential to understand the format. Again, XML schema is used to describe the format.

Thus, XML schema is a core technology for the exchange of ontologies as well as for the exchange of message in a SOA implemented by web services.

XML Schema is one of the core XML standards from W3C. It consists of 2 parts:

1. XML Schema Structures[14], which describes the schema language to construct own definitions.
2. XML Schema data types[15], which describes the basic data types, which are used in XML schema but also in other XML specifications like XQuery.

XML schema offers opposite to its predecessor DTD powerful modelling concepts. It contains a powerful type system, the possibility to define local and global definitions and several subtyping mechanisms.

This allows the usage of several modelling styles ranging from grammar-oriented DTD style to programming language data structures.

[14] XML Schema Structures: http://www.w3.org/TR/xmlschema-1/

[15] XML Schema data types: http://www.w3.org/TR/xmlschema-2/

XML itself with DTD was originally very much oriented towards document-oriented information, which means very large instances, irregular structures, relatively small number of instances with same schema and mostly textual data.

With XML schema it is now possible to additionally model data-oriented information. This means small instances, regular structures, a large number of instances with same schema and typed data.

The consequence is a quite high complexity of the language, which allows several equivalent formulations for the same validation capabilities.

The XML schema data types define a rather complete framework for the definition of basic data types. It includes more than 40 data types, which can be parameterized via so called facettes. Thus it is possible to represent nearly all occurring data values from most programming languages, data bases and other infrastructures.

The XML schema for a concrete XML document is enforced by a so called validator, which checks whether the document conforms to the schema.

Additionally XML schema contains features like assigning default values for not available elements or attributes. Thus the validation extends the values of XML document and adds typing information to the nodes of the XML document. Therefore the result of a validation process is a so called post-schema-validation info set (PSVI).

XML schema has been broadly adopted. It is probably the most important part of the web service description language (WSDL) and is also used in many other XML standards like XQuery, XSLT.

Most of the standard schemas in vertical areas are available in XML schema.

3.1.3.3 XQuery

As mentioned in the previous section about XML schema, NeOn needs to support the exchange of messages and ontologies in a service oriented architecture. To achieve this, the messages and ontologies will be transmitted as XML documents. Now, as one of the core standards XQuery is the most important tool for querying and transforming XML documents. As such XQuery automatically plays an important role within NeOn.

The main intention of XQuery is to be a query language for XML data sources. It can be characterized as a functional language with almost no side effects. It has been recently published as a W3C standard[16].

It is based on a conceptual data model for XML – the XQuery data model. This is an extension of the usual XML data model as the result of XQuery is usually set of nodes or values, which is not a valid XML document.

Thus the notion of sequence as a list of nodes or values is the central concept of the XQuery data model.

Additionally the data model contains nodes like elements, attributes, documents and text nodes. Nodes have an identity, which distinguishes them from values. Values can be of any of the XML schema primitive types like xs:string, xs:decimal or xs:float.

[16] XQuery: http://www.w3.org/TR/xquery/

One can distinguish up to four sub languages within XQuery:

- XML: the first language is XML itself as it is possible to construct arbitrary instances of the XQuery data model, which are a subset of the XML model. This includes the construction of nodes with identity, which has a side effect and thus represents a deviation from a pure functional language.
- XPath: one of the origins of XQuery was XPath. XPath expressions allow traversing the hierarchical paths of the XML data model. They resemble the directory notation of operating systems. For the traversion XPath distinguishes between so called axis like:
  - Child
  - Parent
  - Ancestor
  - Predecessor
  - Siblings

  Additionally filters are possible during the traversal of the hierarchical nodes.
- FLWOR expressions: XPath expressions are not powerful enough for a query language – e.g. they do not allow expressing joins. Therefore the so called FLWOR (For Let Where Order Return) expressions have been introduced. They are an adoption of the SQL select-from-where expressions.

```
for     $b in collection("bib")/books
where   $b/price lt 39.99
return
 <bookresult>
        <author>{$b/author/name}</author>
         <info>  {$b/title, $b/price} </info>
 </bookresult>
```

  In the above example a for-clause defines an immutable variable $b, which is bound to an iterator on a sequence resulting from a XPath expression. The where-clause allows defining filter on those variables. The return-clause allows defining the result of the whole expression by delivering constructed nodes with the variables.
- Programming language constructs: The already mentioned functionality qualifies XQuery as a query language for XML. Nevertheless XQuery has a lot of additional programming language functionality like:
  - Recursive functions
  - Functional if then else
  - Modules

  These constructs beyond a query language gives XQuery the power of a programming language. Thus it can be for example used for tasks, for which conventionally XSLT is used.

3.1.3.4 XQJ

XQJ is the acronym of XQuery for Java. For the moment the XQJ specification is in the status of draft and it is expected to be finished after the XQuery Recommendation.

The main goals of XQJ are:

- To provide a Java interface for XQuery. Using this interface is possible to use XQuery as a query language to XML databases, and also as a XML programming language.
- To be the JDBC for XML databases.
- To have a good integration with other Java XML infrastructures.

XQJ support directly XML objects in Java and with XQJ is possible to use compiled queries. The XML models that XQJ use are DOM, SAX and StaX.

For more information read the XQJ specification (Michels, 2006).

3.1.4 Access Ontological Resources

The key concept of the NeOn project is “networked ontologies”. Since these “networked ontologies” are distributed in the network, we need some mechanism for accessing and managing them.

3.1.4.1 WS-DAI-RDF(S)

Recently, the DAIS WG has updated its charter in order to include the RDF(S) data access initiative as part of its developments. This initiative is focused at developing a set of specifications for providing specific RDF(S) data access mechanisms for the Grid building upon the WS-DAI specification (Esteban Gutiérrez, et al., 2006) The work is being developed by the AIST and the UPM (as part of the OntoGrid project[17])

At the time being, there are two specifications under development:

- *Query-based access:* defines a set of porttypes and messages for dealing with RDF(S) resources using the SPARQL query language.
- *Ontology-based access:* defines a set of porttypes that provide ontology access primitives for dealing with the RDF(S) data model, including creation, retrieval, update and deletion of data. Figure 10 shows a snapshot of the current interface organization. This work is based on the WS-DAIOnt-RDF(S) (Gómez Pérez, et al., 2006) specification which is being developed in the OntoGrid project.

[17] http://www.ontogrid.eu

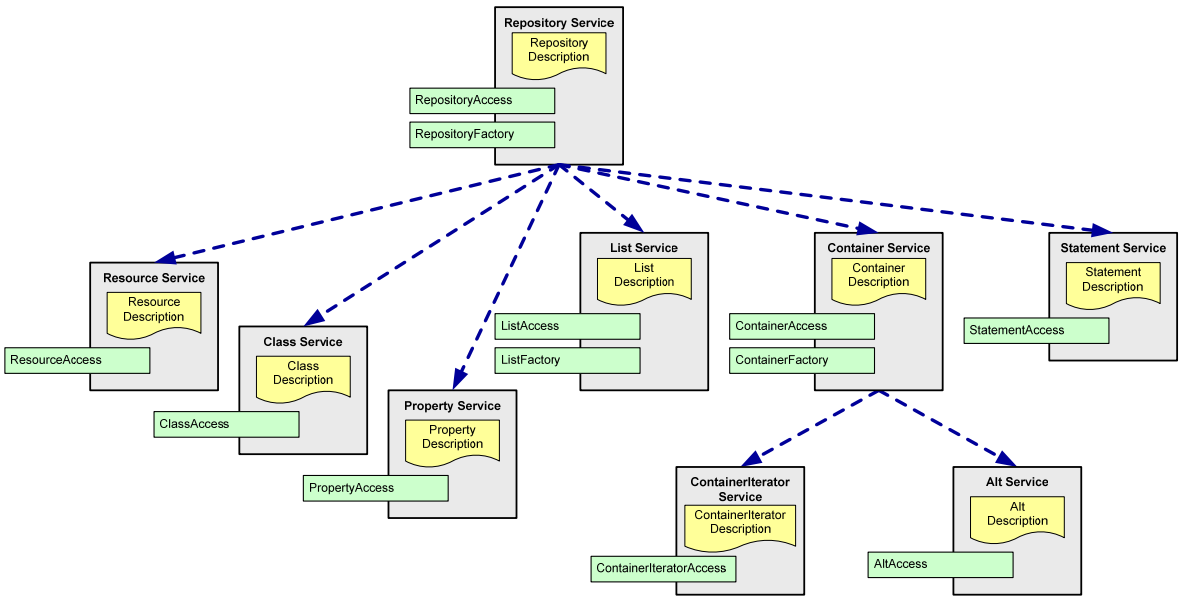

Figure 10: WS-DAI-RDF(S) ontology access interfaces.

3.1.4.2 DIG

The DIG interface[18] provides uniform access to Description Logic Reasoners. The interface defines a simple protocol (based on HTTP PUT/GET) along with an XML Schema that describes a concept language and accompanying operations.

The interface is not intended as a heavyweight specification of a reasoning service. Rather, it provides a minimal set of operations (e.g. satisfiability and subsumption checking and classification reasoning) that have been shown to be useful in applications.

A number of reasoners including FaCT++, Pellet, RacerPro, and KAON2 provide support for DIG. A number of ontology editors (including OilEd, Protege and SWOOP) can use the DIG interface to communicate with such reasoners.

The DIG interface makes a number of assumptions.

- The specification is agnostic as to multiple client connections. Multi-threaded implementations of a reasoner may be provided, but as DIG does not specify transaction models including isolation levels etc., no guarantees are made as to the semantics when clients attempt to simultaneously update and query.
- The connection to the reasoner is effectively stateless. Clients are not identified to the reasoner, thus the reasoner will not distinguish between clients and maintain any kind of consistency checking or record of which client is adding information or making requests. Conversely, a client has no way of ensuring that the reasoner has not been given additional information (such as additional axioms) since its last communication.
- There is no explicit classification request. The reasoner will decide when it is appropriate to, for example, build a classification hierarchy of concepts. This may happen after each TELL request, alternatively the reasoner may choose to defer the classification until absolutely necessary, or even when there is a lull in traffic.

The current version of DIG - DIG 1.1 – has a number of drawbacks: For example, it is not sufficient to capture general OWL-DL ontologies — in particular datatype support is lacking in DIG 1.1 and there is a poor fit between DIG's notion of relations and OWL properties.

[18] DIG Interface: http://dl.kr.org/dig/interface.html

DIG is currently undergoing a major redesign. Version 2.0 will provide the following new features[19]:

- It will be compliant with OWL1.1 (specifically, it will rely on the same XML Schema for the representation of ontologies.
- Well-defined mechanisms for extension to the basic interface: Non-standard Inferences are increasingly recognised as a useful means to realise applications. For example, Least Common Subsumer (LCS) provides a concept description that subsumes all input concepts and is the least specific (w.r.t. subsumption) to do so. A DIG 2.0 extension provides a proposal for an extension supporting NSIs.

3.1.5 Query Languages

In NeOn project, OWL and F-Logic are the two major languages for describing ontologies. Therefore SPARQL as a query language for OWL ontologies and F-Logic's query possibilities are introduced here.

3.1.5.1 SPARQL

SPARQL is designed as a query language for RDF (Resource Description Framework) (Lassila, et al., 1999) and now is widely used to query ontologies that are encoded based on RDF, for example, OWL web ontology language (McGuinness, et al., 2003). Let's firstly look at some brief introduction of the relations between RDF and SPARQL.

An RDF graph is a set of triples and each triple consists of a subject, a predicate and an object. RDF graphs are defined in RDF Concepts and Abstract Syntax[20]. These triples can come from a variety of sources. For instance, they may come directly from an RDF document; they may be inferred from other RDF triples; or they may be the RDF expression of data stored in other formats, such as XML or relational databases. The RDF graph may be virtual, in that it is not fully materialized, only doing the work needed for each query to execute. SPARQL is a query language for getting information from such RDF graphs. It provides facilities to:

- extract information in the form of URIs, blank nodes and literals.
- extract RDF subgraphs.
- construct new RDF graphs based on information in the queried graphs.

As a data access language, it is suitable for both local and remote use. The companion SPARQL Protocol for RDF document[21] describes the remote access protocol.
The current technical document of SPARQL can be found at: http://www.w3.org/TR/rdf-sparql-query/. The computational complexity of SPARQL is NP-Space, which is introduced in (Pérez, et al., 2006) as well as the semantics.

OWL can be encoded based on the RDF language; therefore it is possible to query OWL ontologies using an RDF query such as SPARQL due to the missing of a standardized OWL query language. In the NeOn project, as one of the major ontology language, OWL is widely used in many work packages, particular in case studies work packages as an essential modelling language for knowledge bases in variety of information systems.

[19] DIG 2.0: The DIG Description Logic Interface Overview. http://dig.cs.manchester.ac.uk/overview.html

[20] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[21] http://www.w3.org/TR/rdf-sparql-protocol/

3.1.5.2 Queries in F-Logic

The language of F-Logic and its elements have been covered in D5.1.1 of NeOn. In this deliverable we will focus on the possibilities to pose queries in F-Logic. Being a rule language, query functionality is literally “built in”.

From the rules and facts in an F-Logic Program, a model is computed on which **queries** can be asked. Queries consist of a rulebody with empty rulehead. The result of a query is a list of substitutions for the query’s variables, that can be derived from the facts in the knowledge base.

For example, the following query:

FORALL X, *Y, Z* ← *X*: Person [nationality → spanish, isEmployed → *Y*] AND *Y* [isMember → *Z*]

Could get as a result: X = Filipe, Y = AECE, Z = Pharmainnova.

Note that not only instances and their values but also concept and attribute names can be provided as answers via variable substitutions.

3.2 Version management protocols

The main task of a CVS[22] system is to keep track of different versions of files and folders, typically of software projects and allowing different users to collaborate. Important features thus are multi-user access, merging (in case different users worked on the same file) and versioning including diffs between files (all versions of a file are stored and differences between any version can be presented to the user).

A main motivation for subversion is to have a more comprehensive version management than the currently popular versioning cvs. In addition to cvs functionality subversion provides:

- Versioning of directories: it is possible to version not only simple files but also directories. Thus whole directory trees can be versioned.
- Complete version history: subversion is capable of tracking all operations on files and directories.
- Transaction support: the repository treats a set of modification as one transaction, which is executed completely or not at all.
- Versioning of metadata: for each file or directory metadata in the form of key value pairs can be associated. This metadata will be also versioned.
- Flexible usage of http based network protocols. It actually uses a subset of the webDAV based DeltaV protocol.

[22] http://www.nongnu.org/cvs is a good starting place for information about CVS

3.2.1 Concurrent Versions System (CVS)

CVS works by storing all data in a central repository that users can get access to (this can be either on a server or on a local machine in case it is just used for local versioning). Users then "check out" any given version of the project (in case there are multiple) and work with local copies. On the client side, CVS stores the local copy along with metadata that enable the client to track which version of a file was checked out. The client also monitors which of the files were changed locally. During working on a project locally, users can get updates from the repository to get the latest version of all files, or commit their work to the repository. In case there is a conflict (two users worked on the same file), either versions are automatically merged (in case users worked on different parts of a file) or have to be merged by hand. As another protection, changes can only be committed to the latest possible version of a file. So users are forced to work on the latest possible data. In case of a successful update, the name of the user is stored in along with a new version number per changed file and optional comments. In order to save space, CVS uses delta compression, thus only saving differences between versions and not the complete versions themselves.

For many (open source) software projects it is also common to allow anonymous read-only access to the repository, for example to allow interested users to always work with the latest builds.

While for a long time only command line tools existed, nowadays there are graphical clients available for the most common operating systems. Listing all of the around 50 CVS commands would be out of the scope of this deliverable, especially since the complexity is well hidden by the graphical tools.

The interested reader should take a look at the official manual (Cederqvist, 2002).

3.2.2 Subversion (SVN)

Subversion (Nagel, 2005) was specifically designed to replace CVS as a versioning or revision control application. It therefore offers essentially the same functionality as CVS but offers some improvements. Because elaborating the technical details of the implementation and different commands would be out of scope of this document, we will focus on the functionality and especially on the advantages of SVN in comparison to CVS. Similar to CVS, a number of graphical SVN interfaces exist, so users should not have to bother with learning the command line syntax if they do not choose to do so.

As already mentioned, SVN provides most current CVS features and follows CVS interface except when the developers think there is a compelling reason not to do so. In contrast to CVS, and maybe one of the most requested features, directories, renames and file meta-data are versioned. This means that Subversion does not only file contents and file existence, but also directories, copies and renames as well as arbitrary metadata and file permissions. Commits are truly atomic, meaning no part of a commit takes effect until the entire commit has succeeded. For the ease of retrieving a specific state of the repository, revision numbers are assigned per-commit, not individually per-file. Log messages are attached to the revision and not stored redundantly as in CVS. This allows the retrieval of a specific state of the repository by just checking out the state indicated by a single global revision number in contrast to different version numbers per file. By supporting the HTTP-based WebDAV/DeltaV protocol and Apache web sever for repository-side networked service, key features like authentication, wire compression and basic repository browsing are automatically provided. Alternatively, Subversion can be run as a standalone server using a custom protocol instead of Apache, which then can be run as inetd service or in daemon mode as well as tunnelled over ssh.

Branching and tagging are cheap operations because they are both implemented in terms of an underlying copy operation. The copy takes up a small, constant amount of space. Any copy is a tag and if copies are committed on, this will be considered as a branch.

In contrast to CVS, it was natively designed to be client/server which avoids the maintenance problems plaguing CVS. Also diffs are sent in both directions, not only server to client. This enables efficient use of bandwidth. Binary files can also be efficiently transmitted and stored using a binary diffing algorithm.

The time a commit takes is only proportional to the size of changes resulting from that operation, not the absolute size of the project itself. Also, Subversion is not depending on a BerkeleyDB database back-end. Any normal flat-file back-end using a custom format can be used for the repository.

In contrast to CVS, symbolic links can be placed under the version control. They are recreated in Unix working copies, but not in win32 copies. The command-line output is designed to be both human readable and automatically parseable, to enable scriptability.

Based on local settings, Subversion can use localized messages. It also enables file locking for unmergeable files—so called “reserved checkouts”– and has a file MIME support. Whether to use SVN or CVS is up to the user and the kind of features needed. The user base of SVN is growing steadily while CVS still has a bigger user base due to the long time it has been in use.

Parts of this section have been taken from the official SVN homepage[23].

3.3 Service access protocols, formats and frameworks

3.3.1 Access remote services

In this section we include a list of protocols and techniques that allow accessing remote services that are potentially relevant for NeOn. There are a lot of protocols and techniques that let programmers to develop applications under the client-server paradigm.

The protocol RPC (Remote Procedure Call) was proposed initially by Sun Microsystems as a great advance in comparison with the sockets used until the moment. According to the use of this protocol the programmer did not have to take care on the communications, being them embedded inside the RPC. The RPC is very used in the client-server paradigm. There are several incompatible kinds of RPC but most of them use the Interface Description Language (IDL) that defines the exported methods by the server.

CORBA (Common Object Request Broker Architecture) is a standard that establish a development platform of distributed systems making easier the invocation of remote methods in the object oriented programming paradigm. CORBA uses IDL too for specifying the interfaces with the services that the objects will offer. CORBA was defined and is managed by the Object Management Group (OMG) that defines the APIs, the communications protocol and the mechanism needed for allow the interoperability between different applications written in different languages and executed in different platforms, thing that is fundamental in distributed computing. But CORBA is more than a multiplatform specification, because CORBA defines services usually necessary like security and transactions.

Using RMI (Java Remote Method Invocation) a Java program can export an object. Then a client can connect to this program and invoke methods. Java RMI is very friendly to the programmers and can do things that cannot do other standards like SOAP or CORBA (passage by reference of objects, distributed garbage collection and passage of arbitrary types).

The Java EE platform lets to do multi-tier applications with low development cost. The implementation from Sun Microsystems can be downloaded for free, and there are a lot of open source tools available to extend the platform or to simplify development.

[23] http://subversion.tigris.org/ is a good place to find initial information about SVN

A Web service as a software system designed to support interoperable machine-to-machine interaction over a network. This definition includes many different systems, but in common usage the term refers to those services that use SOAP-formatted XML envelopes and have their interfaces described by WSDL.

The name “SOAP” was originally an acronym for “Simple Object Access Protocol”, but the full name was dropped because the focus shifted from object access to object inter-operability.

SOAP (Gudgin, 2003) is a protocol which sends XML-based messages. It is mostly bound to HTTP as the transport layer, but other bindings are as well supported. SOAP supports synchronous and asynchronous messaging patterns. The most common one is the synchronous one, which reflects the roots of SOAP, when it began as an XML variant of an RPC call.

SOAP itself is an extensible protocol or more precisely a messaging framework. It has already been extended in various ways, e.g. by WS Encryption or WS Security. Thus SOAP plays the role of the foundation layer for the web service stack.

WSDL is the abbreviation for “Web Service Description Language” pronounced “wiz-del”. WSDL (Chinnici, 2006), as its name says, describes the interface of a Web Service in an XML-based way. It deals with the description of services and their messages. The format of the messages is formulated using XML Schema, thus supporting rich and hierarchical data-types. But WSDL supports as well the description of a binding to a specific transport protocol.

The key concept of the WSDL is that this interface definition is independent of the implementation; in the WSDL there is only a reference to the service endpoint defined. In conjunction, the WSDL provides an openness which allows building extensions into it like it's done for BPEL, or WS-Policy.

Web services related technologies

To support reliable, large-scale interconnectivity of Web services by software, computer processible semantics are needed, which include the properties, capabilities, interfaces, and effects of the service (Aalst, 2003). The semantic web services come to give us the solution.

The *“Web Services Data Access and Integration – The Core (WS-DAI) Specification, Version 1.0”* (Atkinson, et al., 2006) defines a collection of generic data interfaces which can be extended in order to provide access to specific types of data resources, i.e. databases, XML documents, etc.

Based on extensions to SOAP messages the Web Service Security specification[24] aims at secure transmission of such messages. The standard covers authentication, integrity and confidentiality issues. It is applicable to various security models and encryption technologies.

While the Web Service Description Language (WSDL) describes functional properties of interacting services, it does not say anything about non-functional properties. But in a world of non trivial Web services communication is not that easy because often non-functional properties have to be considered. The Web Service Policy Framework fills this gap in WSDL.

There are several competing approaches for the definition of such semantic web services. The Semantic Annotations for WSDL and XML Schema standard[25] is the first standardization activity in this area. It does however cover only a relatively small part of the definition of semantic web services. SA-WSDL defines how the important modeling parts of a WSDL can be annotated with semantic definitions. The semantic definitions itself however are not part of the SA-WSDL standard.

Service Data Objects (SDO) and Service Component Architecture (SCA) are projects of the so-called “Open SOA Collaboration“[26]. They aim at powerful and flexible means to build applications in a service-oriented architecture using a wide range of programming languages.

[24] WS-Security Specification. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[25] SA-WSDL. http://www.w3.org/TR/sawsdl/

[26] Open SOA. http://www.osoa.org/

3.3.1.1 RPC (Remote Procedure Call)

Up until now RPC is the most used protocol that allow access to remote services and the first one that appeared. RPC is a protocol that allows a program to invoke a service that is located in a remote machine without the programmer explicitly coding the details for this interaction. RPC spans the application layer and the transport layer in the OSI (Open System Interconnection) model of network communication.

RPC uses the client-server model of distributed computing. The client must know what features does the server provide which are indicated by the service definition, written in IDL (interface description language). A RPC call is a synchronous operation that suspends the calling program until the results of the call are returned. The RPC call flow is explained in the Figure 11. When an RPC is compiled a stub is included in the compiled code that represents the remote service. When the program runs it calls the stub tat knows where the operation is and how to reach the service. The stub will send the message through the network to the server. The result of the procedure will return to the client in the same way.

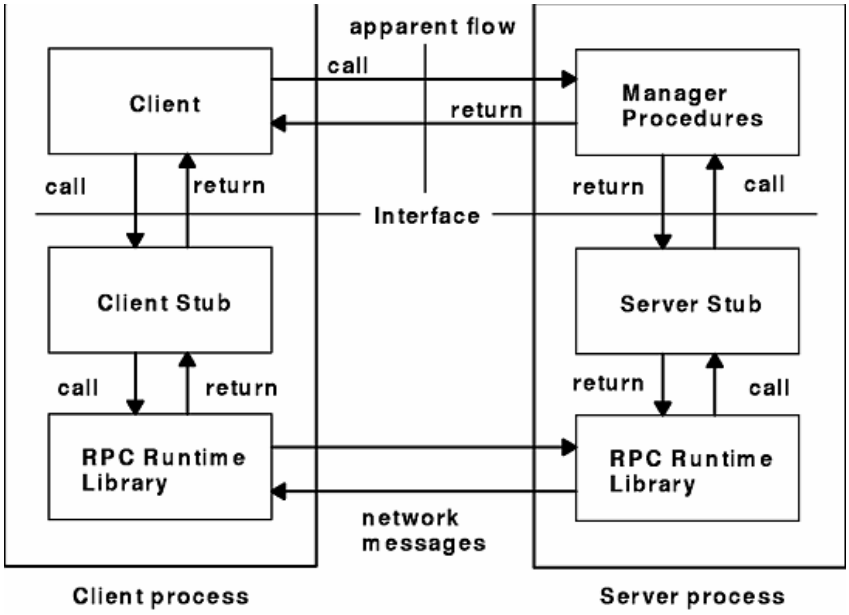

Figure 11: RPC call flow[27]

One of the problems of RPC is t hat RPC implementations use to be incompatible between them. To use one of the possible implementations of RPC will result in a highly dependence of the RPC development. This compatibility problem can have a very negative impact in the global system. A system that implements one concrete RPC solution can have problems of flexibility, maintainability, portability and interoperability.

3.3.1.2 Interface Definition Language (IDL)

The interface definition language is a language used to describe the services that are available from a server. When a client wants to invoke a service in a remote server it needs to know what to invoke and the parameters needed in that invocation. IDL defines how this operation can be called. IDL is a component one of the key elements in the remote procedure call paradigm. As limitation of

[27] Javvin. Network management and security. Available from: http://www.javvin.com/pics/rpc.jpg

IDL it is necessary to mention that the IDL code has to be mapped to a concrete language. The procedures specified in IDL will be implemented in this language and a mapping between the language and the IDL is necessary. A high amount of work is required in order to map the IDL and the language.

3.3.1.3 RMI (Remote Method Invocation)

RMI or Remote Method Invocation is a java application interface that performs the object equivalent of Remote Procedure Calls (RPC). A program can run a remote service that is offered by a service without knowing anything from the implementation. The client and the server need to be running in a Java Virtual Machine and the communication is only possible from one JVM to another. In order to a Java program running in a VM call code developed in other languages it exist the Java Native Interface. The JNI is a solution implemented for those programs that cannot be entirely developed in Java and need other programming languages.

One of the limitations of RMI is that it tied to platforms with Java support. Currently this is not a big issue due to it exist JVM for most of the platforms but it is still an open issue. Security threads are also an important limitation for RMI. There exist security threats with remote code execution, and limitations on functionality enforced by security restrictions.

3.3.1.4 CORBA (Common Object Request Broker)

The Common Object Request Broker (CORBA) was developed in order to communicate between clients and servers developed in different code languages.

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to interoperate. The OMG defines the API of CORBA, the communication protocols and the mechanisms needed to allow communication between clients requesting services and servers offering those services that have been developed with different programming languages. CORBA provides the interoperability needed for these applications, which is basic in distributed computing. The services that are available to the clients are defined by the Interface Definition Language (IDL) which specifies the parameters of the operations that are accessible from the server. Figure 12 depicts the reference model architecture of CORBA defined by the OMG.

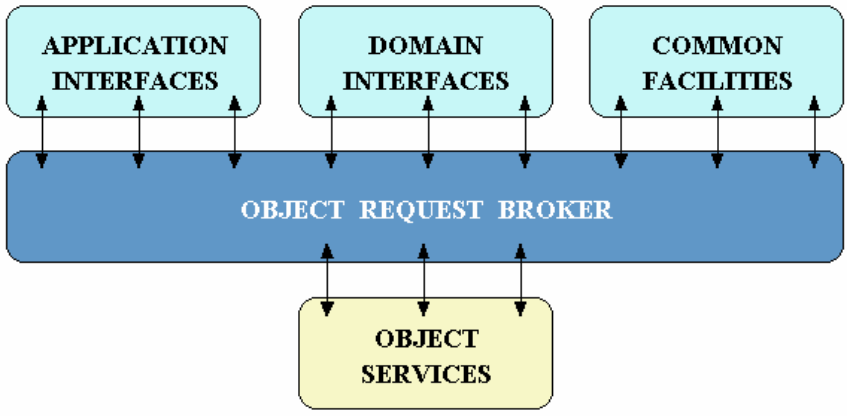

Figure 12: Reference model architecture of CORBA (Schmidt's)

Figure 13 show the how CORBA works. First the client access to the object reference in order to be able to invoke this object. The user also has to know what type of object and the operation to be invoked. The user knows all these things by accessing the interface definition. The client does the petition through an IDL stub. The CORBA ORB will find the implementation of the defined operation through the IDL skeletons. These IDL skeletons are specific for each interface and are in charge of sending the parameters and transfer the control to the implementation of the procedure. Once the operation has been done the results are returned to the user through the same channels used for the invocation.

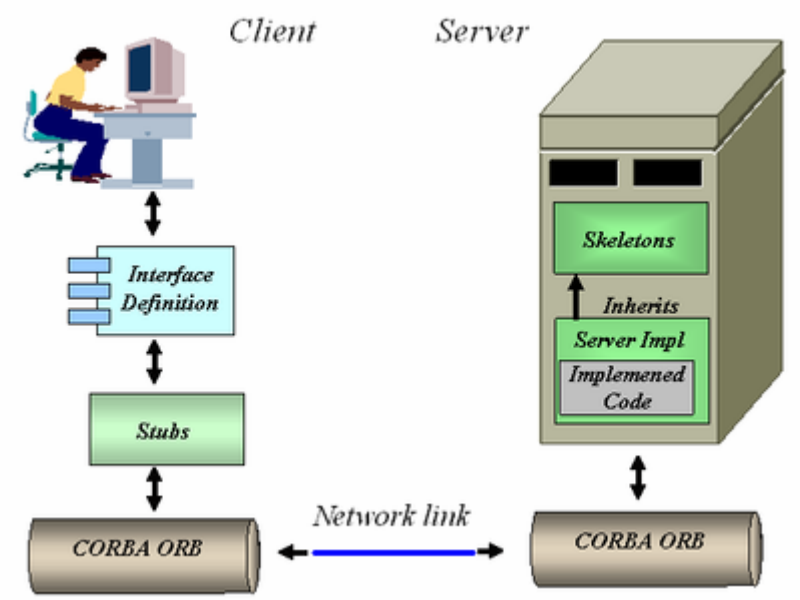

Figure 13: How CORBA works

CORBA also has some limitations. The user has to learn the IDL specification in order to be able to invoke the operations that are available in the server. To implement a service which uses IDL definitions require a mapping between the IDL definition and the language. If the language does not support writing a mapping between this IDL and the language requires a high amount of work.

3.3.1.5 J2EE

J2EE (Java 2 Enterprise Edition)[28] is the enterprise edition of the Java platform created and distributed by Sun Microsystems. It is a standard of development, building and deploying of applications and contains a set of specification, functionalities oriented to the development of corporative and distributed enterprise applications and web services.

Several essential requirements appear in the development of enterprise applications that they must accomplish. For example:

- Reliability and scalability.
- Ease of maintenance.
- Security.

[28] Java 2 Enterprise Edition. Available from: http://java.sun.com/javaee/

- Performance and high availability.
- Extensibility

Also these kind of applications use to have a tier based architecture: one client tier or presentation tier that provides a user interface, one or more intermediate tiers that provides the business logic and a final tier that interacts with the applications and corporative databases.

The J2EE technology integrates a set of APIs, frameworks and design patterns that cover all of these needs. The division on tiers that J2EE propose is:

- **Client tier:** includes the components that are executed in the client side. There are a lot of client types (desktop applications, web navigators, portable devices, etc.).
- **Presentation tier:** is in charge of presenting the system to the user and receiving the information that the user sends. It is a bridge between the client tier and the business tier.
- **Business tier:** this tier contains all the business objects and the service for processing the business logic. It receives the user request and generates the user responses after the execution of the needed processes.
- **Integration tier:** is in charge of the services that provides access to the external resources, for example databases.
- **Resources tier:** contains the final information systems that will be accessed; for example databases, non J2EE applications or other final data sources.

J2EE covers the presentation, business and integration tiers offering for example the following solutions:

- Presentation tier: JSP pages, Servlets and mobile devices programming with J2ME[29].
- Business tier: Enterprise Java Beans Components[30].
- Integration tier: access to databases using JDBC[31], connection to corporative applications using connectors, JMS messages[32] and CORBA interfaces[33].

Finally, J2EE is a platform that sets solutions for development, effective use and multi-tier handling in server centred applications.

3.3.1.6 Web Services

One of the major achievements that web services deliver to applications is that they can provide implementations of almost arbitrary functionality with a location and transportation format independency, i.e. the involved parties only need to know the interface definitions. As such, web services are an ideal technology to implement functionally distributed systems like Neon. Therefore we describe some of most important and not previously mentioned standards and implementation technology around web service technology in more detail here.

Besides its spectacular growth, the Web becomes more dynamic with the advent of the Web service technology. A Web service (WS) is a (self-contained) software component that allows access to its functionality via a Web interface. WSs communicate by employing established protocols for message transport and encoding. Indeed, the W3C Web Services Architecture Working Group defines a Web service as:

[29] Java 2 Micro Edition. Available from: http://java.sun.com/javame/index.jsp

[30] Enterprise Java Beans Technology. Available from: http://java.sun.com/products/ejb/

[31] JDBC Technology. Available from: http://java.sun.com/javase/technologies/database/index.jsp

[32] Java Message Service. Available from: http://java.sun.com/javame/index.jsp

[33] CORBA/IIOP Specification. Available from: http://www.omg.org/docs/formal/04-03-01.pdf

"*a software application identified by an URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols.*" (W3C, 2002)

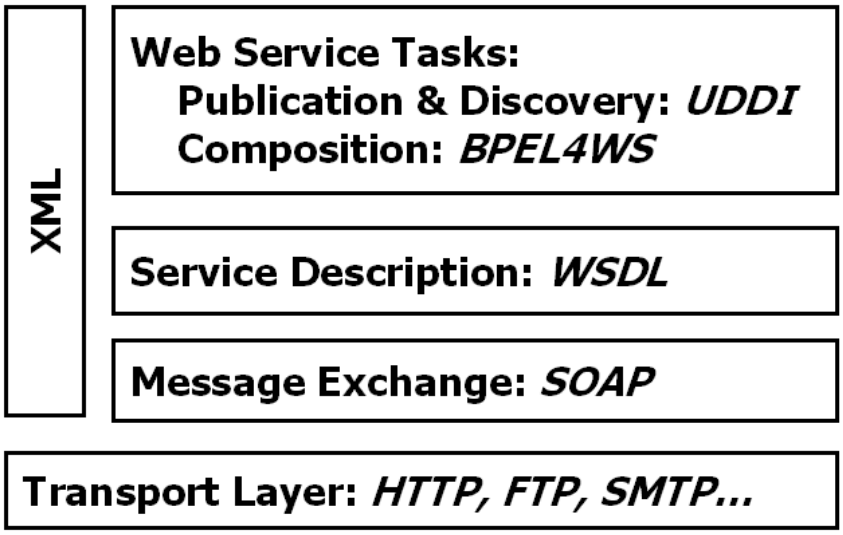

Figure 14: Overview of Web Service Standards.

Web service technology has introduced a new abstraction layer over and a radically new architecture for software. Indeed, the innovative vision is that by employing a set of XML standards to define and describe Web service functionalities, several tasks such as discovery and composition of these services can be facilitated (or even automated) to some extent. Web service technology also aims to facilitate the interaction between different Web services (i.e., software programs) by enforcing the use of XML standards for data exchange. Note, that any kind of data can be exchanged between Web services (e.g., semi-structured, textual, structured) as long as it is embedded in an XML based messaging protocol.

Figure 14 (adapted from (Aalst, 2003)) shows the main Web service technology standards, all based on XML. A Web service interface is described using the Web Service Description Language11[34] (WSDL). Web services exchange messages encoded in the SOAP[35] (Simple Object Access Protocol) messaging framework and transported over HTTP or other Internet protocols. Several tasks can be performed with Web services. A typical Web service life-cycle envisions the following scenario. A service provider *publishes* the WSDL description of his service in UDDI[36], a registry that permits Universal Description Discovery and Integration of Web services. Subsequently, service requesters can inspect UDDI and *locate/discover* Web services that are of interest. Using the information provided by the WSDL description they can directly *invoke* the corresponding Web service. Further, several Web services can be composed to achieve a more complex functionality. Such compositions of services can be specified using BPEL4WS[37] (Business Process Execution Language for Web Services). By relying on these standards, Web services hide any implementation details therefore increasing cross-language and cross-platform interoperability.

[34] http://www.w3.org/TR/wsdl

[35] http://www.w3.org/TR/soap/

[36] http://www.uddi.org/

[37] ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

Example Scenario. Many Web services allow access to large databases permitting controlled access to information that might not be explicitly stated on Web pages. Imagine, for example, a scenario in which a user needs to find *Medicare certified pharmacy which is in a mile range from a location identified by zip code 19901*. Using Web services, the process of finding such a pharmacy could be repeated for *any* zip code (or for any health care provider) relying on the output of one or more Web services and not on data provided by static Web pages. A good Web service to generalize this task is the MedicareSupplier Web service which can retrieve details of Medicare suppliers given a zip code, a city name or the types of supplies provided.

Service(
PortType:MedicareSupplierSoap (
op:GetSupplierByZipCode(
IMsg(zip), OMsg(SupplierDataLists))
op:GetSupplierByCity(
IMsg(City), OMsg(SupplierDataLists))
op:GetSupplierBySupplyType(
IMsg(description), OMsg(SupplierDataLists))
)
)

The example above shows a schematic representation of the WSDL file associated to the MedicareSupplier service. WSDL has considerable support from industry and increasing tool-support (WSDL generators, editors). As an XML-based language, it is machine processable, being a structured and standardized way to describe web-interfaces of services. In WSDL a service is seen as a collection of network endpoints which operate on messages. The example service provides one **port**: *MedicareSupplierSoap*. This port groups together three **operations** that return lists of Medicare suppliers and their details given a zip code (for *GetSupplierByZipCode*), a city name (for *GetSupplierByCity*) or the description of the supplied material (for *GetSupplierBySupplyType*). Each operation has an input (IMsg) and an output (OMsg) message. A **message** has a name and a set of **parts** of certain type. Parts represent input/output parameters depending if they are declared in the input or the output message. For brevity, the example above does not state the name of the message only the name of its parts. The type of the parts can be any XMLSchema data type or a previously defined complex type (the type of the parts is also omitted by the schematic description). A WSDL document has two major parts. First, the ***abstract interface*** of the service specifies the data types, messages and portTypes with the corresponding operations (which refer to previously defined messages). Second, an ***implementation part*** binds the abstract interface to concrete network protocols and message formats (SOAP, HTTP).

Limitations of the Web Service Technology. SOAP, WSDL, UDDI, and BPEL4WS are the standard combination of technology to build a Web service application. However, they fail to achieve the goals of automation and interoperability because they require humans in the loop (Lassila, 2002). Indeed, WSDL specifies the functionality of the service only at a syntactic level. While these descriptions can be automatically parsed and invoked by machines, the interpretation of their meaning is left for a human programmer.

3.3.1.6.1 Semantic Web Services

SOAP, WSDL, UDDI, and BPEL4WS are the standard combination of technology to build a Web service application. However, they fail to achieve the goals of automation and interoperability because they require humans in the loop (Lassila, 2002). Indeed, WSDL specifies the functionality of the service only at a syntactic level. While these descriptions can be automatically parsed and invoked by machines, the interpretation of their meaning is left for a human programmer.

The Semantic Web community addressed the limitations of current Web service technology by augmenting the service descriptions with a semantic layer in order to achieve their automatic discovery, composition, monitoring and execution (McIlraith, et al., 2001) (Martin, et al., 2004) (Stuckenschmidt, et al., 2004). The automation of these tasks is highly desirable and, as a result, several research projects adopted semantic Web service technology in different application domains (e.g., bioinformatics grid (Wroe, et al., 2004), Problem Solving Methods (Motta, et al., 2003)). Semantic Web Service technologies could be useful in the development of the NeOn toolkit; therefore we include them in our analysis.

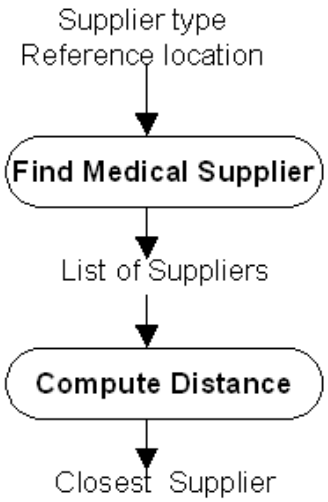

Figure 15: Workflow for binding the closest medical supplier

Example scenario. The example task is a specialization of the more generic task of finding the closest medical provider (or, of the even more generic task of finding a provider in any domain). One of the strategies for performing this generic task is to 1) retrieve the details of all medical providers (of a certain type) and then selecting the closest by 2) computing the distance between the location of the provider and a reference location. This workflow can be schematically represented as in Figure 15. For our example task it is enough if we populate this workflow with the MedicareSupplier service and a Web service that calculates distance between zip codes.

Semantic Web service technology aims to automate performing such tasks based on the semantic description of Web services. Using these descriptions the right services can be selected and combined in a way that would solve the task at hand. There are two major approaches to Web service composition (ten Teije, et al., 2004). First, given the specification of a start and final state, pre/post condition reasoning is performed to select and combine the right services. Second, using the parametric design paradigm, generic task workflows are formally specified and then populated with the right Web services depending on the task at hand.

A common characteristic of all emerging frameworks for semantic Web service descriptions (OWL-S (Martin, et al., 2003), WSMO[38], IRS (Motta, et al., 2003) - see overview and comparison in (Cabral, et al., 2004)) is that they combine two kinds of ontologies to obtain a service description. First, a *generic Web service ontology*, such as OWL-S, specifies generic Web service concepts (e.g., *Input*, *Output*) and prescribes the backbone of the semantic Web service description. Second, a *domain ontology* specifies knowledge in the domain of the Web service, such as types of service parameters (e.g., *City*) and functionalities (e.g., *FindMedicalSupplier*), that fills in this generic framework. We discuss these two kinds of ontologies in the next two subsections.

Generic Web Service Ontologies

Two generic ontologies for Web service descriptions are under development. First, DAML-S is an ontology that permits describing several aspects of a Web service. DAML-S was translated from DAML to OWL and renamed to OWL-S. The second, more recent, initiative is WSMO (Web Service Modelling Ontology) which, even if has overlaps with OWL-S, is based on different principles and brings several additions to OWL-S (see (Lara, et al., 2005) for a detailed analysis of these two ontologies). In this deliverable we only consider OWL-S because NeOn partners have the most expertise in this language. Also, note that the goal of this deliverable is to give an overview of existing techniques that could be possibly useful, but it does not aim at providing a complete analysis. We believe that this section should be enough to demonstrate the idea of SWS and if needed in the project, further in depth analysis can be performed for other related techniques.

The OWL-S ontology is conceptually divided into four sub-ontologies for specifying *what a service does* (Profile[39]), *how the service works* (Process[40]) and *how the service is implemented* (Grounding20[41]). A fourth ontology (Service[42]) contains the *Service* concept which links together a *ServiceProfile*, a *ServiceModel* and a *ServiceGrounding* concept (see Figure 16). The *Service presents* a *ServiceProfile*, is *described by* a *ServiceModel* and *supports* a *ServiceGrounding*. These three concepts are all further specialized in the Profile, Process and Grounding ontologies respectively. In the rest of this subsection, we explain all the three parts of OWL-S by exemplifying their use for describing our example service. We also introduce the schematic service representations that will be used through this subsection.

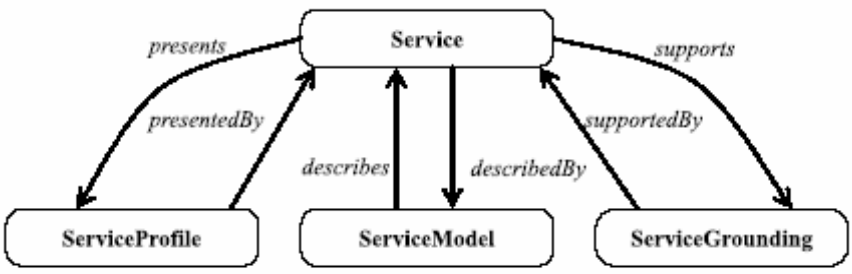

Figure 16: The OWL-S Service Ontology. (Note that the arrows in this picture are directed according to the OWL-S model even if their direction might seam counterintuitive.)

[38] http://www.wsmo.org

[39] http://www.daml.org/services/owl-s/1.0/Profile.owl

[40] http://www.daml.org/services/owl-s/1.0/Process.owl

[41] http://www.daml.org/services/owl-s/1.0/Grounding.owl

[42] http://www.daml.org/services/owl-s/1.0/Service.owl

1. The Profile Ontology specifies the functionality offered by the service (e.g., *GetMedicalSupplier*), the semantic type of the inputs and outputs (e.g., *City, Medicare-Supplier*), the details of the service provider and several service parameters, such as quality rating or geographic radius. This description is used for discovering the service. The central concept of this ontology, *Profile*, is a subclass of *ServiceProfile*.

In the schematic representation of semantic Web service descriptions used throughout this subsection, for each *Profile* instance we depict the process it describes (indicated by the *hasProc* relation) and its functional characteristics (Inputs, Outputs, Preconditions, Effects - from now referred to as IOPE's) together with their type. In the example below, the *MedicareSupplier* service presents three profiles (i.e., it offers three distinct functionalities). Each Profile has a semantic type described by one of the functionality concepts *FindMedicareSupplierByZip*, *FindMedicareSupplierByCity* or *FindMedicareSupplierBy-Supply*. Each Profile describes a Process (later specified in the Process Model - *P1, P2, P3*). Finally, all Profiles return an output which was described with the *SupplierDetails* concept. The input type varies for each Profile: *ZipCode*, *City* or *SupplyType*. Note that this description was constructed using concepts defined in the Web service domain ontology presented bellow.

Service MedicareSupplier:

**Profile:FindMedicareSupplierByZip (hasProc P1)*

(I(ZipCode), O(SupplierDetails))

**Profile:FindMedicareSupplierByCity (hasProc P2)*

(I(City), O(SupplierDetails))

**Profile:FindMedicareSupplierBySupply (hasProc P3)*

(I(SupplyType), O(SupplierDetails))

**ProcessModel: ...*

**WSDLGrounding: ...*

2. The Process ontology. Many complex services consist of smaller services executed in a certain order. For example, buying a book at Amazon.com involves using a browsing service (which selects the book) and a paying service. OWL-S allows describing such internal process models. These are useful for several purposes. First, one can check that the business process of the offering service is appropriate (e.g., product selection should always happen before payment). Second, one can monitor the execution stage of a service. Third, these process models can be used to automatically compose Web services. A *ServiceModel* concept describes the internal working of the service and it is further specialized as a *ProcessModel* concept in the Process ontology. A *Process-Model* has a single *Process* which can be atomic, simple or composite (composed from atomic processes through various control constructs). Each Process has a set of IOPE's.

In our notation, for each service we represent its *ProcessModel* with its *Process*. For each *Process* we depict its type, the involved control constructs, the IOPE's and their types. The *MedicareSupplier* service allows a choice from its three *AtomicProcesses* (corresponding to the three Profiles), therefore its *ProcessModel* consists of a *CompositeProcess* modelled with the *Choice* control construct.

Service MedicareSupplier:

**Profile:...*

**ProcessModel:*

CompositeProcess: MedicareProcess:Choice

{

AtomicProcess:P1 (I(ZipCode),O(SupplierDetails))

AtomicProcess:P2 (I(City),O(SupplierDetails))

AtomicProcess:P3 (I(SupplyType),O(SupplierDetails))

}

**WSDLGrounding: ...*

Profile to Process Bridge. A *Profile* contains several links to the *Process*. Figure 17 shows these links, where terms in bold-face belong to the *Profile* ontology and the rest to the *Process* ontology. Firstly, a *Profile* states the *Process* it describes through the unique property has_process. Secondly, the *Input, Outputs, Preconditions* and *Effects* (from now on IOPE) of the *Profile* correspond (in some degree) to the IOPEs of the *Process*.

Understanding this correspondence is not so trivial given the fact that the IOPE's play different roles for the *Profile* and for the *Process*. In the *Profile* ontology they are treated equally as parameters of the *Profile* (they are subproperties of the *profile:parameter* property). In the *Process* ontology only *Inputs* and *Outputs* are regarded as subproperties of the *process:parameter* property. The Preconditions and Effects are just simple properties of the *Process*. While technically the IOPEs are properties both for *Profile* and *Process*, the fact that they are treated differently at a conceptual level is misleading. The link between the IOPE's in the Profile and Process part of the DAML-S descriptions is created by the *refersTo* property which has as domain *ParameterDescription* and ranges over the *process:paramater*.

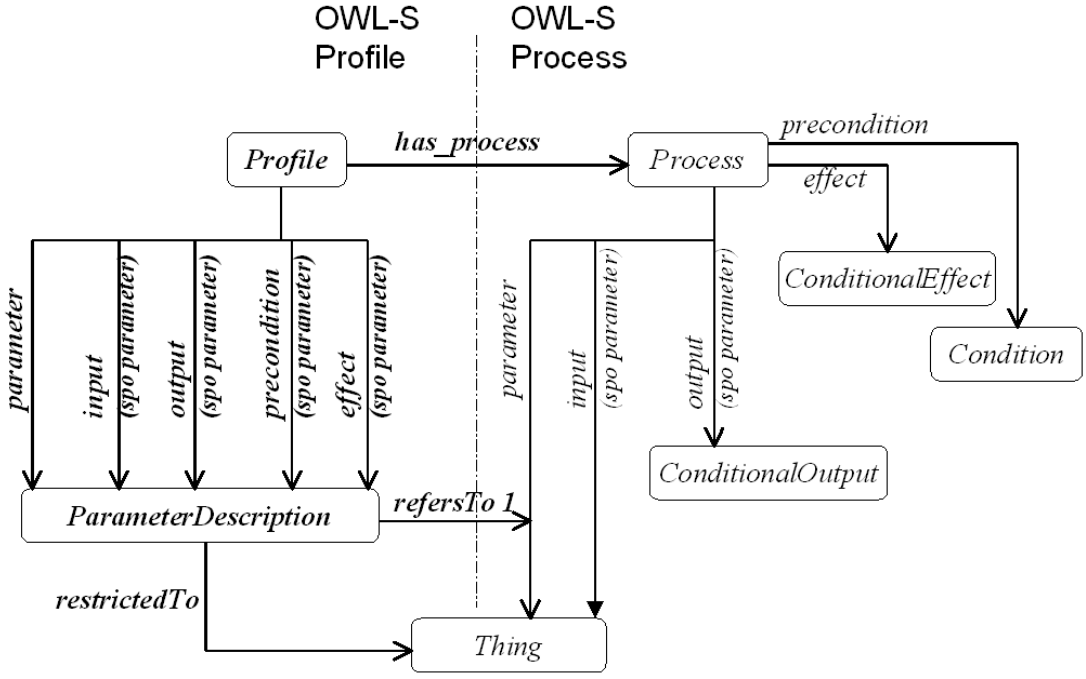

Figure 17: Profile to Process bridge.

3. The Grounding ontology provides the vocabulary to link the conceptual description of the service, specified by the Profile and Process, to actual implementation details, such as message exchange formats and network protocols. The grounding to a WSDL description is performed according to three rules:

R1. Each *AtomicProcess* corresponds to one WSDL operation.

R2. As a consequence of the first rule, each input of an *AtomicProcess* is mapped to a corresponding message-part in the input message of the WSDL operation. Similarly for outputs, each output of an *AtomicProcess* is mapped to a corresponding message-part in the output message of the WSDL operation.

R3. The type of each WSDL message part can be specified in terms of a OWL-S parameter (i.e., an XML Schema data type or a OWL concept).

The Grounding ontology specializes the *ServiceGrounding* as a *WSDLGrounding* which contains a set of *WsdlAtomicProcessGrounding* elements, each grounding one of the atomic processes specified in the *ProcessModel*. In our abstract notation, we depict each atomic process grounding by showing the link between the atomic process and the corresponding WSDL element. The *MedicareSupplier* service has three atomic process groundings for each processes of the *ProcessModel*.

Service MedicareSupplier:

**Profile:...*

**ProcessModel:...*

**WSDLGrounding:*

WsdlAtomicProcessGrounding: Gr1 (P1->op:GetSupplierByZipCode)

WsdlAtomicProcessGrounding: Gr2 (P2->op:GetSupplierByCity)

WsdlAtomicProcessGrounding: Gr3 (P3->op:GetSupplierBySupplyType)

We finish this subsection by describing a set of **design principles** underlying OWL-S that we identified during our use of this ontology.

1. Semantic vs. Syntactic descriptions. OWL-S differentiates between the semantic and syntactic aspects of the described entity. The *Profile* and *Process* ontologies allow for a semantic description of the Web service while the WSDL description encodes its syntactic aspects (such as the names of the operations and their parameters). The *Grounding* ontology provides a mapping between the semantic and the syntactic parts of a description facilitating flexible associations between them. For example, a certain semantic description can be mapped to several syntactic descriptions if the same semantic functionality is accessible in different ways. The other way around, a certain syntactic description can be mapped to different conceptual interpretations offering different views of the same service.

2. Generic vs. Domain knowledge. OWL-S offers a core set of primitives to specify any type of Web service. These descriptions can be enriched with domain knowledge specified in a separate domain ontology. This modelling choice allows using the core set of primitives across several domains.

3. Modularity. Another feature of OWL-S is the partitioning of the description over several concepts. The best demonstration for this is the way the different aspects of a description are partitioned in three concepts. As a result a *Service* instance will relate to three instances each of

them containing a particular aspect of the service. There are several advantages of this modular modelling. First, since the description is split up over several instances it is easy to reuse certain parts. For example, one can reuse the *Profile* description of a certain service. Second, service specification becomes flexible as it is possible to specify only the part that is relevant for the service (e.g., if it has no implementation one does not need the *ServiceModel* and the *ServiceGrounding*). Finally, any OWL-S description is easy to extend by specializing the OWL-S concepts.

Web Service Domain Ontologies

Externally defined knowledge plays a major role in each OWL-S description. OWL-S offers a generic framework to describe a service, but to make it truly useful, domain knowledge is required. For example, domain knowledge is used to define the type of functionality the service offers as well as the types of its parameters.

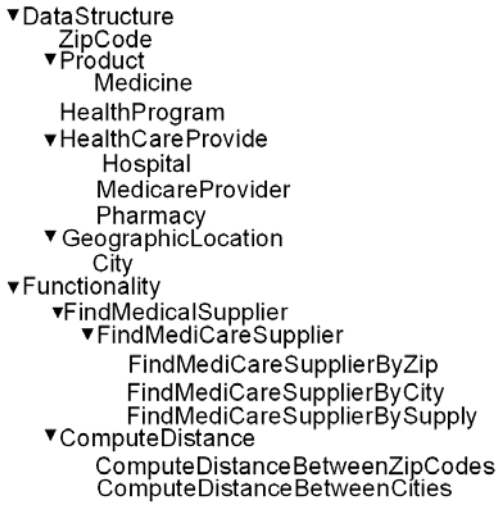

Figure 18: Web service domain ontology.

Figure 18 depicts the hierarchical structure of the domain ontology used to describe the example Web service. Note that it specifies a DataStructure hierarchy and a Functionality hierarchy. The Functionality hierarchy contains a classification of service capabilities. Two generic classes of service capabilities are shown here, one for finding a medical supplier and the other for calculating distances between two locations. Each of these generic categories has more specialized capabilities either by restricting the type of the output parameters (e.g., find Medicare providers) or the input parameters (e.g., *ZipCode, City, SupplyType*).

The complexity of the reasoning tasks that can be performed with semantic Web service descriptions is conditioned by several factors. First, ideally, all Web services in a domain should use concepts from the same (or a small number of coupled/networked) domain ontology in their descriptions. Otherwise the issue of ontology mapping has to be solved which is a difficult problem in itself. While NeOn will provide matching techniques to create ontology networks, the ideal case is still when a single ontology can be used by all Web services. This requires that domain ontologies should be *broad* enough to provide the needed concepts by any Web service in a certain domain. Second, the richness of the available knowledge is crucial for performing complex reasoning.

Example scenario. By using the semanticWeb service descriptions presented above, the example task can be generalized and automated. The right services needed to perform the task can be selected automatically from a collection of services. Semantic metadata allows a flexible selection that can retrieve services that partially match a request but are still potentially interesting. For example, a service which finds details of medical suppliers will be considered a match for a request for services that retrieve details of Medicare suppliers, if the used Web service domain ontology specifies that a *MedicareSupplier* is a type of *MedicalSupplier*. Note that this matchmaking is superior to the keyword based search offered by UDDI. The composition of several services in a more complex service can also be automated. Finally, after being discovered and composed based on their semantic descriptions, the services can be invoked to solve the task at hand.

3.3.1.6.2 WS-DAI (Web Services Database Access and Integration)

This base specification provides the basic means needed for defining systematically and in a uniform way specific data access mechanisms. The means provided consist of a set of message definition patterns and a predefined built-in core interfaces, messages and properties.

Even though, this specification does not mandate how the data access mechanisms for specific data resources must be. These data access mechanisms are defined as WS-DAI “realizations”, which are WS-DAI extensions that define how specific data resources and systems can be described and manipulated.

The WS-DAI specification, and underlying realizations, can be used in traditional web services environments or be included as part of a grid fabric.

Organization

The specification is organized around the notions of *data access services* and *data resources*. A data access services is a web service that implements one or more WS-DAI interfaces to provide access to data resources. A data resource is a software system that can act as a sink or source of data. Figure 19 shows how these elements are related within the WS-DAI model.

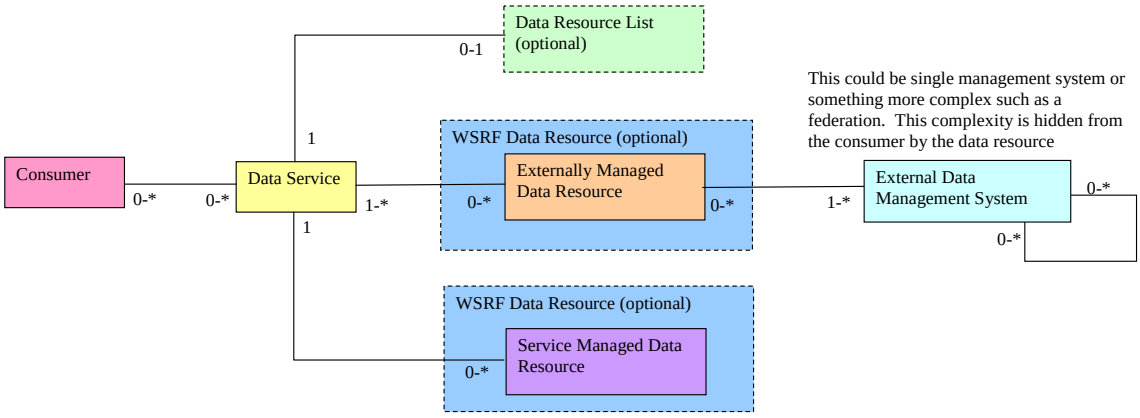

Figure 19: Elements of the WS-DAI model, taken from (Atkinson, et al., 2006)

WS-DAI distinguishes between two types of data resources: *externally managed*, which are those which exist out the data service (i.e. a relational database); and *service managed*, those which are part of the data service itself.

In the end, data access services offer to the consumer a set of *interfaces* which provide specific data access to a concrete type of data resource. The term *interface* refers to the set of *messages* or *properties* that a service offers. *Properties* are used to describe the characteristics of a data resource as well as the data access service’s relationship with that data resource.

However WS-DAI do not mandate how interfaces are composed into data access services.

Message Patterns

The WS-DAI specification defines two types of data access patterns:

Direct data access: a consumer receives a direct response, containing the requested data, following a request made to a data access service. An operation that directly inserts, updates or deletes data through a data access service is also involved in direct data access.

Indirect data access: a consumer does not receive the results in the response to a request made to a data access service. The request to access data will be processed by the data access service and data resource, with the results being made available to the consumer indirectly as a new data resource, often through a different data access service that may support a different set of interfaces.

3.3.1.6.3 WS-Security

One of the most important challenges in the development of distributed systems is to provide security when accessing remote services. This applies not only for the services themselves but also the reliable and unchangeable transport of message. WS-Security provides an extensible general-purpose mechanism for associating security tokens of multiple security token formats with messages. The extensibility allows applications to choose their favourite authentication and authorization mechanisms. The WS-Security profile specification describes how to encode Username Tokens, X.509 Tokens, SAML Tokens, REL Tokens and Kerberos Tokens and attach them to SOAP messages. WS-Security is a flexible standard which allows applications to describe the credentials that are included with a message. To address all the security requirements of application WS-Security can be used together with other Web service protocols.

Usage of XML Signature and security tokens ensure that messages have originated from the appropriate sender and were not modified (messages integrity). XML Encryption and security tokens are used to keep portions of a SOAP message unreadable for externals (message confidentiality).

3.3.1.6.4 WS-Policy

WS-Policy provides an extensible and flexible mechanism to describe requirements and properties of entities in a Web Service based application landscape. Within this framework WS-Policy offers syntax to describe primitive and composite requirements and properties. While service documentation could describe these requirements and properties as well, WS Policy aims at extensibility with other WS specifications. Further building blocks in this framework are WS-PolicyAttachments and WS-PolicyAssertions. The former one describes general-purpose mechanisms for associating policies with the subject to which they apply, the latter one defines building blocks to accommodate a wide variety of policy exchange models.

3.3.1.6.5 SCA and SDO

Service Data Objects (SDO) and Service Component Architecture (SCA) are projects of the so-called “Open SOA Collaboration“[43]. They aim at powerful and flexible means to build applications in a service-oriented architecture using a wide range of programming languages. It is not yet clear whether these technologies will reach the state of a standard or de-facto standards. However, due to the fact that many important drivers of the service-oriented paradigm collaborate in these projects they should at least be considered when defining an implementation architecture for a service-oriented application landscape.

SCA covers the service components level using SOA principles. One could also say it covers the “A” in SOA. It defines a programming model for service-based systems to construct, assembly and deploy composite applications. It also models creation of new service components.

[43] Open SOA. http://www.osoa.org/

SDO covers the data aspects within applications. It aims at a simplified, unified and consistent handling of such data independent of their source and format. In the end application programmers shall uniformly access and modify data from distinct and heterogeneous data sources.

SDO supports different kind of data sources like RDBMS, XML and web services. It decouples the data source and the application by offering access to the data objects disconnected from the connection of the data source. Manipulations on the disconnected data objects are however also supported. All gathered manipulations are propagated to the data source. Details of the changes are available in the form of change summaries.

It defines both a dynamic and static API to access and manipulate a graph of data objects. The static API is generated for a specific schema of data like XML schema or relational schema. Both APIs allows traversing the graph of data objects. Additionally simple queries in the form of XPath expression are possible for more complex traversal of objects.

The dynamic API has a rather complete support for accessing the meta model, which goes beyond Java reflection.

Due to its disconnected mode and complete support of XML it is especially suited to access web services in a very comfortable way in Java and other programming languages.

3.3.2 Access service directories and registries

A directory service is needed to publish and access to the resources (ontological and non-ontological) that are spread among the Internet. We will need two directory services as the NeOn architecture is defined. One directory is for the resources in the level of the distributed repository and the other is needed for describing and accessing the services in the middle-ware layer of the NeOn architecture.

3.3.2.1 UDDI

UDDI (Universal Description, Discovery and Integration) defines an interface of a service registry. UDDI forms together with the SOAP protocol and the WSDL web service description the basic functionality for a SOA.

The goal is to enable business over the Internet to publish and look up services and to describe, how they interact. The functionality can be categorized by an analogy to white and yellow pages according to the following aspects:

- White Pages, which provides address, contact and known identifiers
- Yellow Pages, which lists industrial categories based on standard taxonomies and
- Green Pages, containing technical information about services exposed by the business

It has a fixed data model for services consisting of:

- businessEntity, which usually describes a company or an organization
- businessService, the description of the service itself
- bindingTemplate, the mechanism how to invoke the service
- tModel, for the classification of services
- categoryBags

Their relationship is sketched by the following figure from the UDDI standard:

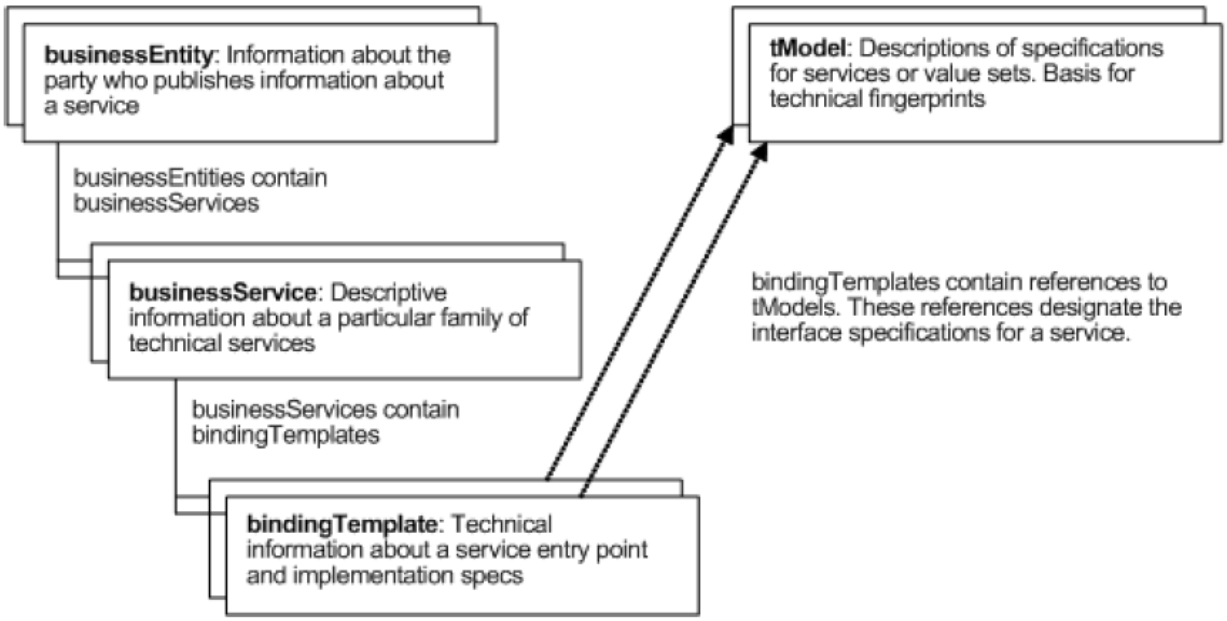

Figure 20: UDDI datamodel for services relationship diagram

The indirection of discovering a web service at runtime via an UDDI registry gives applications high degree of flexibility. Thus applications are isolated from movement of services, from the availability of a single service and can utilize information about the quality of the service.

UDDI is an OASIS standard. The current version is V3[44].

3.3.2.2 ebXML registry

An ebXML registry securely manages arbitrary artefacts of SOA applications together with its associated metadata. The artefacts can have XML representations like XML schemas or web service descriptions (WSDL) but also non-XML representations are possible.

Thus is extends an UDDI registry in two main aspects:

- Registry functionality for arbitrary artefacts not just services.
- Integration of repository functionality with the registry functionality.

The standard consists of two parts:

- **ebXML registry information model**. The information model has predefined classes and associations but allows the user also to define its own one.
- **ebXML registry services**. The functionality is available as set of web services differentiating mainly between query services and life cycle services

Additionally to the service interface the functionality is available via the java interface Java API for XML Registries[45].

From a functionality point of view it contains the UDDI functionality but offers additional features.

[44] UDDI Specification: http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm

[45] JAXR: http://java.sun.com/webservices/jaxr

3.4 Notification and syndication protocols

In the collaborative design process, a way to notify the people involved is needed. In this section we include the protocols used in the communication via e-mail (SMTP, POP and IMAP), one protocol for instant messaging (JABER) and a protocol that come from the Grid community used for the purpose of notification (INFOD).

3.4.1 Notification protocols

3.4.1.1 Simple Mail Transfer Protocol (SMTP)

The Simple Mail Transfer Protocol (SMTP) is used for communication between a mail (SMTP-) client and a mail (SMTP-) server. Its objective is to transfer mails in a reliable and efficient way. It is a higher level protocol and is independent of specific network layer protocols (although, most of the times, TCP/IP is used). Transporting mail across networks (SMTP mail relaying) is an important feature since most of the times sender and recipient are not on the same server or subnet.

Because the basic SMTP protocol was considered to be not complete enough, service extensions were proposed. Nowadays most of the SMTP servers support the extensions while being 100% backward compatible to the original SMTP protocol. Timeouts are used for every SMTP command, however, the duration can be determined by the SMTP server administrator. Recommendations are given in the SMTP RFC.

Using SMTP commands, different functions and the mail transfer itself can be requested by the user.

It should be noted that SMTP alone does not offer authentication or security. It is important to realize that information like the sender's email address can be easily forged and the message content altered by any of the relaying servers. The only way to ensure security is to use encryption or digital signatures (e.g. by using PGP) at the message level. To protect users against spammers crawling for email addresses, the VRFY and EXPN command can also be disabled. For additional information see RFC2821 (Klensin, 2001)

Information used in this section is taken directly from the SMTP specification—RFC2821 (Klensin, 2001).

3.4.1.2 Post Office Protocol (POP)

The Post Office Protocol (POP) is used to retrieve emails from a remote server given a standard TCP/IP connection on port 110. The latest version –POP3– has made earlier versions obsolete. The main advantage of POP is the possibility of just connecting to the server for mail fetching when needed and not having to keep a connection established. Since emails are downloaded to the client, they can easily be processed offline, as opposed to emails processed using the IMAP protocol (in case the user has not enabled complete caching of the IMAP folders).

The communication between client and server can be distinguished into three different states: **AUTHORIZATION**, **TRANSACTION** and **UPDATE**.

After the TCP connection between client and mail-server is established and the server has issued a on line greeting (e.g. +OK POP3 server at your service) the POP3 session is in the **AUTHORIZATION** state. During that state, only the commands *USER*, *PASS*, *QUIT* and *APOP* (if implemented) are valid[46]. After the user has identified himself through either a *USER*, *PASS*

[46] For a complete list of commands and description of corresponding actions see list below

combination or the *APOP* command, the specific maildrop is locked with an exclusive-access lock to prevent modification or deletion of messages before the **UPDATE** state is entered. If this could successfully be applied, the session enters the **TRANSACTION** state with no messages marked as deleted, otherwise the user is presented an error message, a potential lock is released and the connection potentially closed.

After a maildrop has been opened, every message is assigned a message-number and the size of each message in octets is noted. During **TRANSACTION** state, *STAT*, *LIST*, *RETR*, *DELE*, *NOOP*, *RSET*, *QUIT*, and *TOP*, *UIDL* (if implemented) are valid. Once the client issues the *QUIT* command during **TRANSACTION** state the session switches into the **UPDATE** state (in case *QUIT* is issued during **AUTORIZATION** state the session terminates without entering **UPDATE** state). During **UPDATE** state all messages marked as deleted are removed from the maildrop.

Apart from the responses to *STAT*, *LIST* and *UIDL* commands, the server's reply is significant only to "+OK" and "-ERR".

The following commands are used to communicate with the server:

- *USER* xxx Tells the server which account to access
- *PASS* xxx Sends the password to that account
- *STAT* Returns the status of the mailbox, this is for example number of messages and new messages
- *LIST* (n) Returns the number and size of emails (or, if the optional parameter n is also submitted, of the n^{th} email)
- *RETR* n Is used to retrieve the n^{th} email from the server
- *DELE* n Deletes the n^{th} email from the server
- *NOOP* No function, server responds with +OK (can be used to avoid time-out in case the server has one)
- *RSET* resets all *DELE* commands
- *QUIT* quits the current POP3 session and executes all *DELE* commands

Optional commands (depending on the server used or commands implemented):

- *APOP* secure authorization
- *TOP* n x retrieves header and first x lines from the n^{th} email
- *UIDL* n displays the unique ID of the n^{th} email

Closing remarks: Since password and user information is transmitted in clear text when not using the *APOP* command, one way to ensure privacy is encrypting the whole communication with the server using SSL/TLS.

Information used in this section is taken directly from the POP3 specification—RFC1939 (Myers, et al., 1996).

3.4.1.3 Internet Message Access Protocol- Version 4rev1 (IMAP)

The main idea of IMAP is that clients can manipulate and use mailboxes on servers as if they were local. Caching strategies can be used to enable offline clients to process their email. The protocol operates on port 143 TCP. All interactions between client and server are in the form of lines (strings that end with a CRLF). There are different states of client server communication. After the initial connection has been established and the server has sent a greeting, the communication is in the **not authenticated** state (in case the connection had no **pre-authentication**). Once the client has authenticated himself (or the connection has been established using pre-authentication), the

connection is in the **authenticated** state. After a successful *SELECT* or *EXAMINE* (see below) command, the connection is in the **selected** state. From here, using the *CLOSE* command the connection can go back to the **authenticated** or the **logout** state in case the *LOGOUT* command is used.

There are also experimental commands. They are prefixed with an *X* and are not defined in the RFC (Crispin, 2003).

For each of the commands, different server answers exist. They can be found in the official specification (Crispin, 2003). Concluding it can be said that using IMAP to access emails is beneficial for users who want to access a mailbox from different machines or want to share mailboxes with other users. The administration possibilities for the mailboxes are far superior to POP3. Different access rights can be given, a folder structure can be created on the server, and, if needed, messages can be cached locally and changes synched once a connection has been re-established. Also security is improved because encrypted authentication is supported natively. Messages are not downloaded automatically, which keeps down the traffic and therefore is crucial when the mailbox is accessed from mobile devices that have high bandwidth costs.

Downsides include a higher load on the mail server and the fact that the protocol is very complicated to implement correctly. Also in most cases sent messages have to be transferred twice, once to the SMTP server and once for the copy in the sent-items folder on the IMAP server.

Information used in this section is taken directly from the IMAP4 specification—RFC3501 (Crispin, 2003).

3.4.1.4 Jabber

Jabber is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time. Jabber is built on the Extensible Messaging and Presence Protocol (XMPP) that is an open, XML-based protocol for near-real-time, extensible instant messaging and presence information. The jabber network is a decentralized server-based network in which users can not talk directly one to another and there is no central login server like other instant messaging servers. To communicate to other people in different Jabber servers the client sends the message to its server. This server sends the message to the other client's server (if it is not blocked) and the other side server sends the message to the final client.

3.4.1.5 INFOD

The *"Information Dissemination in the Grid Environment – Base Specifications"* (Davey, et al., 2006) provides mechanisms for subscription-based data access.

The specification defines the infrastructure needed to allow data sources to make data available to consumers by a notification mechanism. Whenever data is changed in a data resource, an event is generated and messages created and passed from the *publishers* to the *consumers*, using a shared registry where publishers publish their messages, and consumers consume those messages to which they have subscribed, see Figure 21.

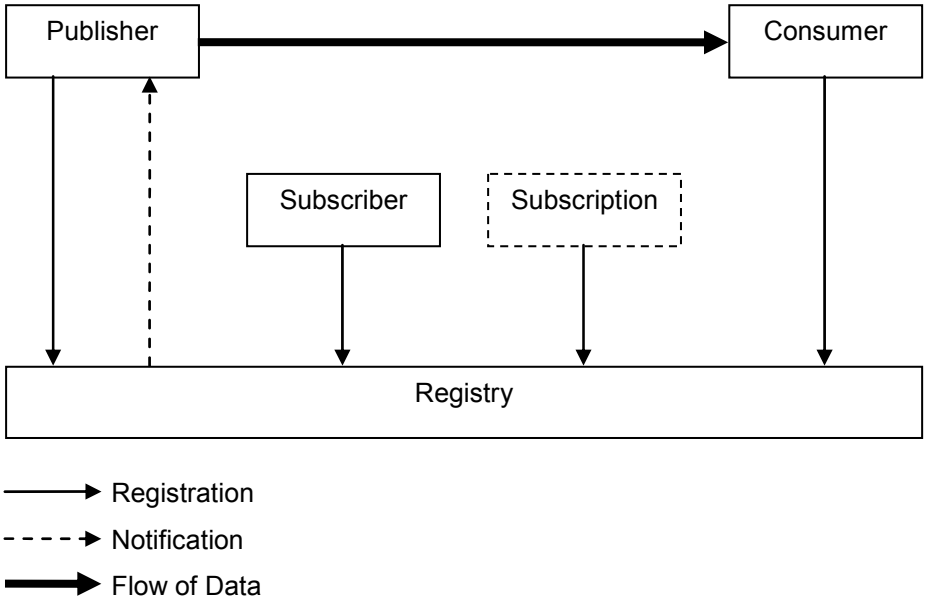

Figure 21: INFOD subscription-based data access, taken from (Davey, et al., 2006)

Figure 22 shows the interfaces defined by (Davey, et al., 2006).

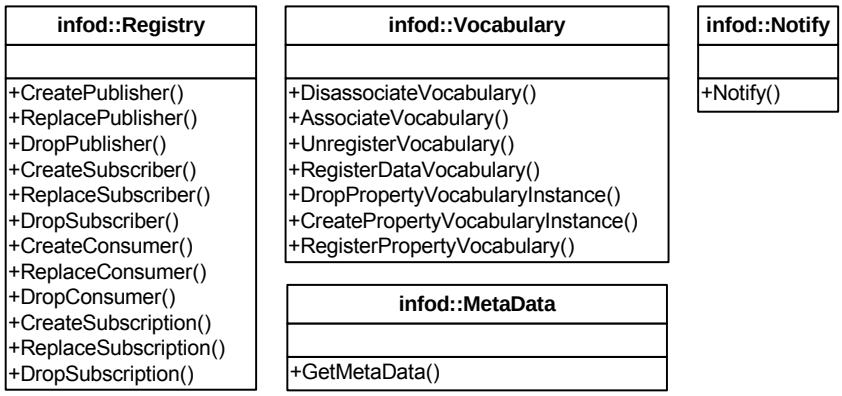

Figure 22: INFOD Interfaces

3.4.2 Syndication protocols and formats

Web syndication is a form of syndication in which a section of a website is made available for other sites to use. This could be simply by licensing the content so that other people can use it; however, in general, web syndication refers to making Web feeds available from a site in order to provide other people with a summary of the website's recently added content (for example, the latest news or forum posts). ICE, RSS and Atom are protocols that allow web syndication.

3.4.2.1 ICE

This section presents an overview of the Information and Content Exchange (ICE) protocol in its 2.0 version, summarized from (Brodsky, et al., 2004).

ICE[47] is a protocol developed by the International Digital Enterprise Alliance (IDEAlliance) to be used by content producers (syndicators) and their consumers (subscribers).

The ICE protocol defines the roles and responsibilities of Syndicators and Subscribers, defines the format and method of content exchange, and provides support for management and control of syndication relationships.

The ICE specification presented in this chapter is last one that exists by the time of writing this deliverable, ICE 2.0, and this chapter refers every time to this version where not stated explicitly.

ICE Overview

The ICE working scenario is the following. Two entities are involved in forming a business relationship where ICE is used. The Syndicator produces content that is consumed by Subscribers.

An important point to understand is that ICE operations start after the two parties have already agreed to have a relationship, and have already worked out the contractual, monetary, and business implications of that relationship.

The ICE protocol covers three general types of operations:

- Messaging
- Delivery / Transport / Packaging
- Subscription

ICE uses a package concept as a container mechanism for generic data items. ICE defines a sequenced package model allowing Syndicators to support both incremental and full update models. ICE also defines push and pull data transfer models as well as out-of-band transfer.

Managing exceptional conditions and being able to diagnose problems is an important part of syndication management; accordingly, ICE defines a mechanism by which faults can be exchanged in a standardized manner between (consenting) Subscribers and Syndicators.

Finally, ICE provides a number of mechanisms for supporting miscellaneous operations, such as the ability to query and ascertain the status of the subscription.

ICE conformance levels

ICE 2.0 defines two levels of conformance. These levels of conformance spell out the features of ICE that must be supported for that level of conformance. The definition of levels of conformance enables software vendors to develop ICE applications that are interoperable:

- Basic ICE software can be expected to interoperate with other software that supports Basic ICE.
- Full ICE software can be expected to interoperate with other software that supports Full ICE.
- Full ICE software can be expected to interoperate with other software that supports Basic ICE.

[47] http://www.icestandard.org/

Basic ICE

The Basic ICE level of conformance provides for very simple syndication functionality, as Figure 23 shows. In fact, all that Basic ICE enables is for the Syndicator to post messages to a URL where the Subscriber can “get” them.

In Basic ICE, the Subscriber initiates all messages with `HTTP GET` to URLs on the Syndicator. Basic ICE does not allow for subscription management capabilities. The Syndicator sends no messages to the Subscriber in Basic ICE. Basic ICE has no requirement for either the Syndicator or the Subscriber to establish a “listener” for push messages.

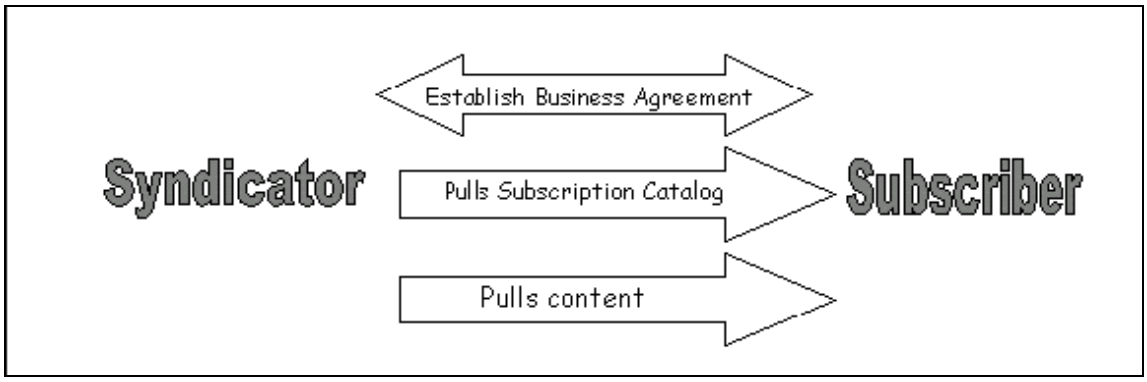

Figure 23: Basic ICE capabilities

Full ICE

A Full ICE implementation implements all the features of the ICE 2.0 specification (Figure 24). Full ICE implementations must support SOAP (Mitra, 2003) transport bindings and adds messages to support subscription management.

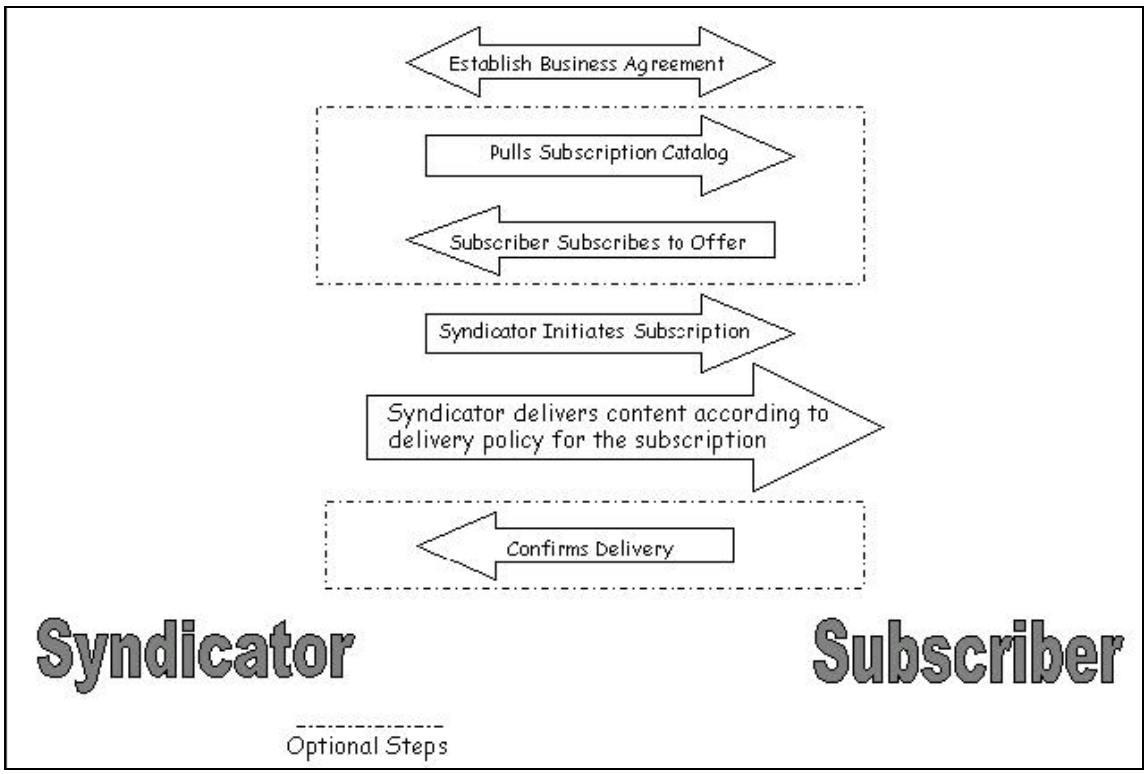

Figure 24: Full ICE capabilities

Protocol overview

The ICE protocol is primarily a request/response protocol that allows for fully symmetric implementations, where both the Syndicator and Subscriber can initiate requests. This fully symmetric implementation is known as Full ICE. The ICE protocol also allows for a Basic ICE implementation where only the Subscriber can initiate requests.

ICE uses message exchange as its fundamental protocol model, where a message is defined for the purposes of this specification to be a SOAP payload.

ICE messages contain header information along with requests and responses. A request asks for the performance of an operation.

Every logical operation in ICE is described by a request/response pair. All operations are forced to fit this model; thus, a valid ICE protocol session always comprises an even number of messages when it is in the idle state.

Bindings of ICE

Because one of the design goals of ICE 2.0 is to enable ICE as a Web service, the capability of ICE to function over SOAP is critical. While ICE 2.0 will remain a transport independent protocol, thus the ICE 2.0 Specification will explicitly discuss binding of the generic ICE protocol over the SOAP transport mechanism and term that ICE/SOAP. In addition to the specification of ICE 2.0 with an explicit binding to SOAP, ease of implementation also dictates that ICE 2.0 also enable the use of `HTTP:GET` ICE/HTTP as a transport mechanism:

- **ICE/HTTP**. In Basic ICE all messages are initiated with a SOAP Response Message Pattern (`HTTP:GET`). The `HTTP:GET` retrieves whatever data is identified by a URL. The body of the response will be an XML message with a SOAP envelope and ICE messages.
- **ICE/SOAP**. For Full ICE, an explicit binding to SOAP is provided. The ICE message header was designed to be carried in the SOAP header and the ICE fault, delivery and subscription mechanisms were designed to be enclosed in the SOAP body.

For either Full ICE or Basic ICE, content is encoded in an ICE/SOAP format. This is simply a XML file where ICE messages are wrapped in a SOAP envelope.

Identifiers

ICE defines a number of identifiers that control the access to content and enable content management throughout the syndication process.

- **Subscriber and Syndicator Identifiers**. ICE uses globally unique identifiers for identifying Subscribers and Syndicators. The globally unique identifier for the Subscriber and Syndicator should conform to the Universal Unique Identifier defined by the Open Group (The Open Group, 1997).
- **Other identifiers**. As distinct from the Subscriber UUID and the Syndicator UUID as outlined by the Open Group, ICE does not define the format of other identifiers it specifies except for uniqueness constraints. All other identifiers function as being unique only within a certain scope.

ICE syntax, datatypes and namespaces

ICE 2.0 uses XML as the format for all ICE messages. XML schemas are used to define simple datatypes, the ICE message header and status codes and ICE delivery and subscription elements.

ICE simple datatypes are defined in a ICE simple datatypes schema, and all ICE-defined elements are defined into one of three ICE namespaces to enable ICE to function as a Web service and utilize SOAP messaging.

The ICE 2.0 namespaces are:

- xmlns:icemes = “http://icestandard.org/ICE/Spec/V20/message”
- xmlns:icedel = “http://icestandard.org/ICE/V20/delivery”
- xmlns:icesub = “http://icestandard.org/ICE/V20/subscribe”
- xmlns:icesdt = “http://icestandard.org/ICE/V20/simpledatatypes”

ICE schemas

ICE defines three schemas for sending messages, for delivery of content, and for establishing and cancelling subscriptions.

ICE Message schema

The ICE message schema is defined within `ice-message.xsd` as the *icemes* namespace. This schema defines structures relating to the ICE message itself. This includes message header information and ICE status codes. ICE uses the SOAP envelope and SOAP header. The ICE message is carried within the SOAP header.

The ICE message uses the simple datatypes defined within the simple datatype module.

ICE Delivery schema

ICE delivery is defined in a schema module with the *icedel* namespace. This module defines the elements that support the delivery of syndicated content and is carried within the SOAP body.

ICE delivery is most often made up of packages. Two kinds of ICE packages include those bearing or point to content and those containing a catalog of subscription offers.

ICE Subscribe schema

The ICE subscription module is used to establish and cancel subscriptions for syndicated content. It is defined within the *icesub* namespace. Each ICE subscription contains one offer that will be subscribed to. Attributes on the offer identify it uniquely. Each ICE subscription offer must contain a delivery policy. The delivery rule within the delivery policy defines how and when content will be delivered.

In addition, the ICE subscription allows for the subscriber to cancel a subscription, for the syndicator to verify cancellation and for the subscriber to get the status of a subscription.

3.4.2.2 RSS

RSS is a simple XML-based system that allows users to subscribe to a content source. Using RSS, an organization can put its content into a standardised format.

A program known as a feed reader or aggregator can check a list of feeds on behalf of a user and display any updated articles that it finds.

There are also search engines for content published via web feeds.

RSS Overview[48]

There are several different versions of RSS, falling into two major branches (RDF based versions and non RDF based versions). 1.* versions are based on RDF and include the following versions:

- RSS 0.90 was the original Netscape RSS version. This RSS was called RDF Site Summary, but was based on an early working draft of the RDF standard, and was not compatible with the final RDF Recommendation.
- RSS 1.0 is an open format by the RSS-DEV Working Group, again standing for RDF Site Summary. RSS 1.0 is an RDF format like RSS 0.90, but not fully compatible with it, since 1.0 is based on the final RDF 1.0 Recommendation.
- RSS 1.1 is also an open format and is intended to update and replace RSS 1.0. The specification is an independent draft not supported or endorsed in any way by the RSS-DEV Working Group or any other organization.

The RSS 2.* branch includes the following versions:

- RSS 0.91 is the simplified RSS version released by Netscape called Rich Site Summary. This was no longer an RDF format, but was relatively easy to use. It remains the most common RSS variant.
- RSS 0.92 through 0.94 are expansions of the RSS 0.91 format, which are mostly compatible with each other and with RSS 0.91, but are not compatible with RSS 0.90.
- RSS 2.0.1 has the internal version number 2.0. RSS 2.0.1 was proclaimed to be “frozen”, but still updated shortly after release without changing the version number. RSS now stood for Really Simple Syndication. The major change in this version is an explicit extension mechanism using XML Namespaces.

For the most part, later versions in each branch are backward-compatible with earlier versions (aside from non-conformant RDF syntax in 0.90), and both versions include properly documented extension mechanisms using XML Namespaces, either (in the 2.* branch) or through RDF (in the 1.* branch). Most syndication software support both branches.

Syntax and Semantics

RDF based format (1.0)

RSS 1.0 has an RDF Schema available. Dublin Core module is an official module of the RSS-DEV working group. According to the RSS 1.0 Specifications[49] a graph representing this schema is the following:

[48] From Apache Labs – RSS (file format): http://www.apachelabs.org/rssfileformat.htm

[49] RSS 1.0 Specifications: http://web.resource.org/rss/1.0/

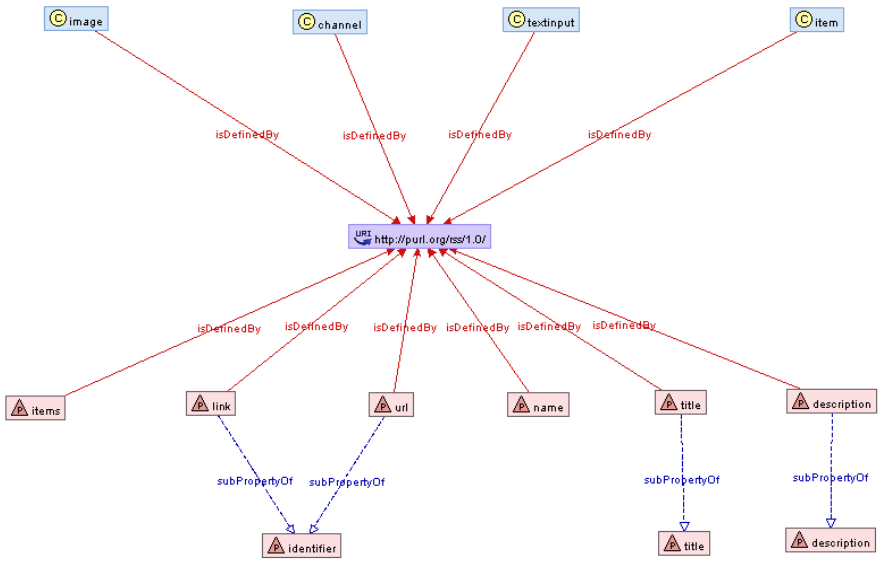

Figure 25: RSS 1.0 Graph

Description of the classes:

- image: an RSS image.
- channel: an RSS information channel.
- textinput: an RSS text input.
- item: an RSS item.

Description of the properties:

- items: points to a list of rss:item elements that are members of the subject channel.
- link: the URL to which an HTML rendering of the subject will link. It is a sub property of `http://purl.org/dc/elements/1.1/identifier`.
- url: the URL of the image to used in the 'src' attribute of the channel's image tag when rendered as HTML. It is a sub property of `http://purl.org/dc/elements/1.1/identifier`.
- name: the text input field's (variable) name.
- title: a descriptive title for the channel. It is a sub property of `http://purl.org/dc/elements/1.1/title`.
- description: a short text description of the subject. It is a sub property of `http://purl.org/dc/elements/1.1/description`.

Non RDF based format (2.0)

According to the RSS 2.0 Specifications[50] the required channel elements are:

- title: the name of the channel. It's how people refer to the service.
- link: the URL of the HTML website corresponding to the channel.
- description: phrase or sentence describing the channel.

[50] RSS 2.0 Specifications: http://www.rssboard.org/rss-specification

According to the RSS 2.0 Specifications the optional channel elements are:

- language: the language the channel is written in. copyright: copyright notice for the content in the channel.
- managingEditor: email address for person responsible for editorial content.
- webMaster: email address for person responsible for technical issues relating to channel.
- pubDate: the publication date for the content in the channel.
- lastBuildDate: the last time the content of the channel changed.
- category: specify one or more categories that the channel belongs to.
- generator: a string indicating the program used to generate the channel.
- docs: a URL that points to the documentation for the format used in the RSS file.
- cloud: allows processed to register with a cloud to be notified of updates to the channel.
- ttl: number of minutes that indicates how long a channel can be cached before refresing from the source.
- image: specifies a GIF, JPEG or PNG image that can be displayed with the channel.
- rating: the PICS rating for the channel.
- textInput: specifies a text input box that can be displayed with the channel.
- skipHours: a hint for aggregators telling them which hours they can skip.
- skipDays: a hint for aggregators telling them wich days they can skip.

The following is an example of an RSS 2.0 file:

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Liftoff News</title>
    <link>http://liftoff.msfc.nasa.gov/</link>
    <description>Liftoff to Space Exploration.</description>
    <language>en-us</language>
    <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
    <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>Weblog Editor 2.0</generator>
    <managingEditor>editor@example.com</managingEditor>
    <webMaster>webmaster@example.com</webMaster>

    <item>
      <title>Star City</title>
      <link>http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp</link>
      <description>How do Americans get ready to work with Russians aboard the
        International Space Station? They take a crash course in culture, language
        and protocol at Russia's Star City.</description>
      <pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>
      <guid>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</guid>
    </item>

    <item>
      <title>Space Exploration</title>
      <link>http://liftoff.msfc.nasa.gov/</link>
      <description>Sky watchers in Europe, Asia, and parts of Alaska and Canada
        will experience a partial eclipse of the Sun on Saturday, May 31st.</description>
      <pubDate>Fri, 30 May 2003 11:06:42 GMT</pubDate>
      <guid>http://liftoff.msfc.nasa.gov/2003/05/30.html#item572</guid>
    </item>

  </channel>
</rss>
```

3.4.2.3 Atom

The information in this document is partially extracted from RFC 4287 about the Atom Syndication Format[51].

Differences with RSS

The development of Atom was motivated by the existence of many incompatible versions of the RSS syndication format and the poor interoperability of XML-RPC-based publishing protocols.

A brief description of the ways Atom 1.0 seeks to differentiate itself from RSS 2.0 follows (Schollmeier, 2002):

- RSS 2.0 may contain either plain text or escaped HTML as a payload, with no way to indicate which of the two is provided. Atom in contrast uses an explicitly labeled (i.e. typed) "entry" (payload) container. It allows for a wider variety of payload types including plain text, escaped HTML, XHTML, XML, Base64-encoded binary, and references to external content such as documents, video and audio streams, as so forth.
- RSS 2.0 has a "description" element which can contain either a full entry or just a description. Atom has separate “summary” and “content” elements. Atom thus allows the inclusion of non-textual content that can be described by the summary.
- Atom standardizes autodiscovery in contrast to the many non-standard variants used with RSS 2.0.
- Atom is defined within an XML namespace whereas RSS 2.0 is not.
- Atom specifies use of the XML's built-in xml:base for relative URIs. RSS 2.0 does not have a means of differentiating between relative and non-relative URIs.
- Atom uses XML's built-in xml:lang attribute as opposed to RSS 2.0's use of its own "language" element.
- In Atom, it is mandatory that each entry have a globally unique ID, which is important for reliable updating of entries.
- Atom 1.0 allows standalone Atom Entry documents whereas with RSS 2.0 only full feed documents are supported.
- Atom specifies that dates be in the format described in RFC 3339 (which is a subset of ISO 8601). The date format in RSS 2.0 was underspecified and has led to many different formats being used.
- Atom 1.0 has IANA-registered MIME-type. RSS 2.0 feeds are often sent as application/rss+xml, although it is not a registered MIME-type.
- Atom 1.0 includes an XML schema. RSS 2.0 does not.
- Atom is an open and evolvable standard developed through the IETF standardization process. RSS 2.0 is not standardized by any standards body. Furthermore according to its copyright it may not be modified.
- Atom 1.0 elements can be used as extensions to other XML vocabularies, including RSS 2.0 as illustrated in a weblog post by Tim Bray entitled "Atomic RSS".
- Atom 1.0 describes how feeds and entries may be digitally signed using the XML Digital Signatures specification such that entries can be copied across multiple Feed Documents without breaking the signature.

[51] RFC 4287. The Atom Syndication Format: http://tools.ietf.org/html/rfc4287

Despite the emergence of Atom as an IETF Proposed Standard and the decision by major companies such as Google to embrace Atom, use of the older and more widely known RSS 1.0 and RSS 2.0 formats has continued.

- Many sites choose to publish their feeds in only a single format. For example CNN, the New York Times, and the BBC offer their web feeds only in RSS 2.0 format.
- News articles about web syndication feeds have increasingly used the term "RSS" to refer generically to any of the several variants of the RSS format such as RSS 2.0 and RSS 1.0 as well as the Atom format. (For example, "There's a Popular New Code for Deals: RSS" (NYT January 29, 2006)
- RSS 2.0 support for enclosures led directly to the development of podcasting. While many podcasting applications, such as iTunes, support the use of Atom 1.0, RSS 2.0 remains the preferred format.
- Each of the various web syndication feed formats has attracted large groups of supporters who remain satisfied by the specification and capabilities of their respective formats.

Atom overview

Atom is an XML-based document format that describes lists of related information known as “feeds”. Feeds are composed of a number of items, knows as “entries”, each with an extensible set of attached metadata.

The primary use case that Atom addresses is the syndication of Web content such as weblogs and news headlines to Web sites as well as directly to user agents.

Atom is an extendible format. There are two kinds of Atom Documents specified in terms of XML:

- Feed Documents: representation and metadata of a feed and some entries associated with it.
- Entry Documents: represents one entry, outside the context of a feed.

Syntax and Semantics

Common Atom constructs

Many elements of Atom share a few common structures. When an element is identified as a particular kind of construct, it inherits the corresponding requirements from the definition of that construct. The common constructs are:

- Text Constructs: contains human-readable text in a Language-Sensitive context. Text Constructs have a “type” attribute that can be “text” (default), “html”, or “xhtml”.
- Person Constructs: describes a person, corporation or similar entity. It could be a human-readable name, an IRI associated with the person or the email of a person.
- Date Constructs: represents date and time.

Container elements

- feed: the document (i.e., top-level) element of an Atom Feed Document, acting as a container for metadata and data associated with the feed. Its element children consist of metadata elements followed by zero or more atom:entry child elements.
- entry: represents an individual entry, acting as a container for metadata and data associated with the entry. This element can appear as a child of the atom:feed element, or it can appear as the document (i.e., top-level) element of a stand-alone Atom Entry Document.
- content element: either contains or links to the content of the entry. The content of atom:content is Language-Sensitive. On the atom:content element, the value of the "type"

attribute may be one of "text", "html", or "xhtml". atom:content may have a "src" attribute, whose value must be an IRI.

Metadata elements

- author: the author of the entry or feed.
- category: information about a category associated with an entry or feed. Its attributes are:
  - term: identifies the category to which the entry or feed belongs.
  - scheme: IRI that identifies a categorization scheme.
  - label: human-readable label for display in end-user applications.
- contributor: Person Construct that indicates a person or other entity who contributed to the entry or feed.
- generator: identifies the agent used to generate a feed.
- icon: an IRI that identifies an image that provides iconic visual identification for a feed.
- id: universally unique identifier for an entry or feed.
- link: defines a reference for an entry or feed to a Web resource.
  - href: the link's IRI.
  - rel: indicates the link relation type with the values "alternate", "related", "self", "enclosure", or "via". A description of the semantics of this values is in RFC 4287.
  - type: an advisory MIME media type.
  - hreflang: the language of the resource pointed to by the href attribute. When used together with the rel="alternate", it implies a translated version of the entry.
  - title: human-readable information about the link.
  - length: advisory length of the linked content in octets.
- logo: an image that provides visual identification for a feed.
- published: Date Construct indicating an in the life cycle of the entry.
- rights: Text Construct that conveys information about rights held in and over an entry or feed.
- source: designed to allow the aggregation of entries from different feeds while retaining information about an source feed of an entry.
- subtitle: Text Construct that conveys a human-readable description for a feed.
- summary: Text Construct that conveys a short abstract of an entry.
- title: text construct that conveys a human-readable title for an entry or feed.
- updated: Date Construct indicating the most recent instant when an entry or feed was modified in a way the publisher considers significant.

Example

Next, we have an example atom feed document:

```
<?xml version="1.0" encoding="utf-8"?>
   <feed xmlns="http://www.w3.org/2005/Atom">

     <title>Example Feed</title>
     <link href="http://example.org/"/>
     <updated>2003-12-13T18:30:02Z</updated>
     <author>
       <name>John Doe</name>
     </author>
     <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

     <entry>
       <title>Atom-Powered Robots Run Amok</title>
       <link href="http://example.org/2003/12/13/atom03"/>
       <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
       <updated>2003-12-13T18:30:02Z</updated>
       <summary>Some text.</summary>
     </entry>

   </feed>
```

3.5 Network communication protocols

In computer science, an intelligent agent (IA) is a software agent that exhibits some form of artificial intelligence that assists the user and will act on their behalf, in performing non-repetitive computer-related tasks. While the working of software agents used for operator assistance or data mining (sometimes referred to as bots) is often based on fixed pre-programmed rules, "intelligent" here implies the ability to adapt and learn.

In this section we provide standards that can be use for communicating agents (ACL) and a language and protocol for exchanging information and knowledge (KQML). As the reader will see in the following description of these languages, they can be used to exchange ontological elements or whole ontologies, thus being already existing protocols for exchanging and sharing ontologies.

3.5.1 Languages for working with agents

In computer science, an intelligent agent (IA) is a software agent that exhibits some form of artificial intelligence that assists the user and will act on their behalf, in performing non-repetitive computer-related tasks. While the working of software agents used for operator assistance or data mining (sometimes referred to as bots) is often based on fixed pre-programmed rules, "intelligent" here implies the ability to adapt and learn.

In this section we provide standards that can be use for communicating agents (ACL) and a language and protocol for exchanging information and knowledge (KQML).

3.5.1.1 Agent Communication Language (ACL)

Introduction / Aims / Objectives

In order to communicate, agents, which might be implemented in different programming languages, need a common language, i.e. common protocols. Apart from that they also have to be able to locate each other autonomously.

The Foundation for Intelligent Physical Agents[52] (FIPA) is an IEEE Computer Society standards organization promoting standards in agent-based communication and agent-based technology in general. They provide the basic models enabling agents to communicate also on a higher abstraction layer, not only on the layer of procedural calls. One important aspect in that context is the distinction between “belief” (statements that are assumed to be true) and “uncertainty” (statements the agent does not know to be true but assumes to be true with high probability). Within their possibility, agents perform different actions to reach their intentions. When different agents interact, they communicate by exchanging messages.

Protocol Summary

The communication is partitioned into different sub-problems that can be solved using standardized protocols, languages or mechanisms. On the lowest layer of communication a message has to be exchanged using a transport-protocol for which it is embedded into a message envelope (Message Transport Reference Model) (FIPA00067). The communication mechanism is not standardized by FIPA, yet every FIPA compliant agent platform has to support the Internet Inter ORB (IIOP) standard developed by OMG as communication protocol. The message itself constitutes an action and therefore one of the 22 defined communicative acts (FIPA00037). The statement the communicative act is about is called “message content”. It is a “tuple consisting of an action expression denoting the action to be done and a proposition giving the conditions of the

[52] http://www.fipa.org/

agreement." (FIPA00037). An ontology (as domain knowledge) can be used so that the content can be interpreted in a certain way

Summary of Messages, syntax and Format

On the level of the messages the ACL is defined by the Message Structure Specification (FIPA00061) including syntax and grammatical structure of a message.

Components of a message are (FIPA00061):

- performative: type of communicative act (e.g. request, inform, agree etc.)[53]
- sender: denotes identity of sender
- receiver: denotes identity of intended recipients, specified by a number of agent-IDs
- reply-to: responses to the message should be send to this agent
- content: content of message; object of the action
- language: one of the content languages (e.g. FIPA-SL, CCL, etc)[54]
- encoding: specific encoding of the content language expression[55]
- ontology: denotes the ontology(s) used to give meaning to the symbols in the context expression
- protocol: denotes interaction protocol the sending agent is employing with ACL message[56]
- conversation identifier: introduces an expression used to identify the ongoing sequence of communicative acts that together form a conversation
- reply-with: introduces expression used by responding agent to identify this message
- in-reply-to: denotes expression reference an earlier action to which this message is a reply
- reply-by: ultimatum for answer

For each of the 22 communicative acts the FIPA Communicative Act Library Specification (FIPA00037) defines a formal syntax and semantics. For example if agent i informs agent j that (it is true that) it is raining today it could look like this:

```
(inform
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "weather (today, raining)"
  :language Prolog)
```

An interaction protocol models a typical "conversation" between agents for which possible sequences of messages of certain types have been specified. Examples for interaction protocols are auctions, brokering services or requests. To enable the creation of additional protocols, the FIPA Interaction Protocol Library Specification (FIPA00025) defines how a protocol has to look like. The specification of protocols is done using protocol diagrams, an extension of UML sequence diagrams.

[53] for reserved values see (FIPA00037)

[54] for reserved values see (FIPA00007)

[55] for reserved values see (FIPA00007)

[56] for reserved values see (FIPA00025)

As mentioned above, the language used to formulate the content is specified in the attribute "language" in the message. FIPA defined the properties of such a content language in the FIPA Content Language Library Specification (FIPA00007). At the moment four languages have been specified:

- FIPA Semantic Language (SL) (FIPA00008)
- Constraint Choice Language (CCL) (FIPA00009)
- Knowledge Interchange Format (KIF) (FIPA00010)
- Resource Description Framework (RDF) (FIPA00011)

An agent does not have to support a certain language. Therefore two agents have to find out whether they share at least one common language before they first communicate.

For the transport of a message in a common language between agents FIPA provides several specifications to provide standardization. In compliance with the standards an ACL-message is embedded in a message envelope for transport, which itself contains all information necessary for transport. Therefore the message content itself is not important for transportation and can be encrypted. Information necessary for decryption is then provided in the message envelope. (FIPA00067).

For transportation between different platforms a Message Transport Protocol (MTP) (FIPA00067) has to be used. FIPA specifications describe the use of CORBA Internet Inter-ORB Protocol (IIOP) (FIPA00075), the Hypertext Transfer Protocol (http) (FIPA00084) and the Wireless Application Protocol (WAP) (FIPA00076). The representation of the message envelope is chosen according to the MTP using CORBA Interface Definition Language (IDL) for IIOP, XML for http (SOAP) and bit-efficient for WAP.

3.5.1.2 Knowledge Query and Manipulation Language (KQML)

The following document relies on the different KQML specifications and publications (Finin, et al., 1994) (ARPA Knowledge Sharing Initiative, 1993) (Labrou, et al., 1997) (Fritson, et al., 1994).

Introduction / Aims / Objectives

For an interaction to take place between two agents, they have to agree on the following points:

- Transport: How are the messages interchanged?
- Language: What is the meaning of a message?
- Specification: How is the communication between agents structured?
- Architecture: How to connect systems within the realm of different protocols?

KQML provides a syntactic grounding for communication between software agents. Certain transport assumptions are made for underlying network communication protocols (Labrou, et al., 1997):

- agents are connected by a unidirectional connection carrying discrete messages
- connections might have delays
- when an agent receives a message, it knows from where it arrived
- when an agent sends a message, it can decide where to send it
- messages sent to a single recipient arrive in the order in which they were sent
- message transmission is reliable

Protocol Summary

KQML is conceptually a layered language, consisting of the content layer, the message layer, and the communication layer. Because the content (which is in the content layer) can carry any representation language, including languages expressed as ASCII strings or binary notation, KQML is independent of the representation language. Features of the message describing details of lower level communication are encoded in the communication layer. The message layer determines the kinds of interaction one can have with a KQML-speaking agent. It identifies the protocol to be used to deliver the message and supplies a “speech act” or “performative” which the sender attached to the content. Performatives tell the receiver whether the content is an assertion, a query, a command, or any other of a set of known performatives. The message layer also includes optional features to describe the content like its language or the ontology it assumes.

Summary of Messages, syntax and Format

Following the ASCII-representation of common lisp prefix notation, performatives are placed in round brackets and consist of parameters divided by blanks. Parameters start with a colon followed by the parameter keyword. The values of the parameters follow after a blank. The parameter value can be of type word or expression, or can itself be a whole performative. Performatives that are used frequently, like sender or content, are reserved in the specification of KQML. While some parameters are mandatory for certain performatives, others are optional. Discussing all different performatives would be out of scope for this document, detailed information can be found in the KQML specifications (ARPA Knowledge Sharing Initiative, 1993) (Finin, et al., 1994).

Important are the following reserved performatives:

- **discourse performatives** which deal with queries to databases and manipulation of data (ask-if, tell, deny, insert, subscribe, ...)
- **intervention and mechanics performatives** that take care of communication control like error messages or status messages (error, sorry, standby, discard, ...)
- **facilitation and networking performatives** are used to find other agents to perform tasks (register, forward, broadcast, transport-address, ...)

The content of a message can be described using the following optional arguments:

- **:sender** <word> and **:receiver** <word> denote the sender and receiver of a performative
- **:reply-with** <word> and **:in-reply-to** <word> are necessary for messages, that expect responses, like queries. The sender of the first message provides an expression in the “reply-with” parameter that the agent responding places in the “in-reply-to” parameter
- **:content** <expression> contains the actual content of the performative
- **:language** <word> defines the language of the content like PROLOG or KIF
- **:ontology** <word> provides the ontology used in the content field

An exemplary KQML-message (Fritson, et al., 1994) illustrates how language and ontology are included in the message to ensure a common understanding of the message:

```
(tell
        :language KIF
          :ontology motors
          :in-reply-to s1
        :content (= (val (torque motor1) (sim-time 5))    (scalar 12 kgf))
```

So concluding, KQML provides a flexible basic envelope for messages exchanged between agents while it does not require a certain KR language in its content section.

3.5.2 P2P Protocols

Peer-to-peer (P2P) computing is not a new concept. One can argue that when two computers were first connected, they formed a P2P network. The Internet as originally conceived in the late 1960s was a peer-to-peer system. The goal of the original ARPANET was to share computing resources around the U.S. The challenge for this effort was to integrate different kinds of existing networks as well as future technologies with one common network architecture that would allow every host to be an equal player (Anderson, 2001). As another example, Mail servers, network news servers (NNTP), and domain name servers (DNS) operate in peer-to-peer networks (i.e. e-mail servers interact directly with each other to send, route, and receive e-mail messages and can be considered a P2P network) (Brookshier, et al., 2002).

Coming up with a concise definition of P2P, however, is not so simple. There are not only problems with what makes up a P2P application, many competitive P2P protocols and implementations that operate in very different ways (Brookshier, et al., 2002).

Among the literature we find many definitions:

A peer-to-peer (or P2P) computer network is a network that relies primarily on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. Such networks are useful for many purposes. Sharing content files containing audio, video, data or anything in digital format is very common, and real-time data, such as telephony traffic, is also passed using P2P technology. A pure peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server (i.e. FTP server).

Another definition of P2P is by describing what it's not rather than trying to pin down what it is (Brookshier, et al., 2002):

P2P is not about eliminating servers. It is not a single technology, application, or business model. Perhaps most controversial is that it should be not characterized strictly by degree of centralization versus decentralization.

Centralization in a P2P network can consist of a central catalog, such as Napster [see section 3.5.2.3]. Napster acted as a traditional client server when users were looking for music and it acted as a P2P network when users transferred files. In a completely decentralized P2P network, such as Gnutella [see section 3.5.2.2], no one peer is different than another except in the content that it shares. Some other approaches mix centralization with decentralization, such as JXTA [see section 3.5.2.3], looking for a happy medium.

In General, P2P is more a style of computing that makes the network interactions more symmetrical

Classification of peer-to-peer networks:

One possible classification of P2P networks is *according to their degree of centralization* in order to distinguish P2P networks with a central entity from those without any central entities. It is general practice to split the Peer-to-Peer networking definition into two sub-definitions (Schollmeier, 2002):

Pure peer-to-peer:

- Peers act as equals, merging the roles of clients and server
- There is no central server managing the network
- There is no central router

Hybrid peer-to-peer:

- Has a central server that keeps information on peers and responds to requests for that information.
- Peers are responsible for hosting available resources (as the central server does not have them), for letting the central server know what resources they want to share, and for making its shareable resources available to peers that request it.
- Route terminals are used addresses, which are referenced by a set of indices to obtain an absolute address.

Another possible classification is *based on how the nodes in the overlay network*[57] *are linked to each other*. According to this criterion we can classify P2P networks as structured and unstructured:

An *unstructured* P2P network is formed when the overlay links are established arbitrarily. Such networks can be easily constructed as a new peer that wants to join the network can copy existing links of another node and then form its own links over time. In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query has to be flooded through the network in order to find as many peers as possible that share the data. The main disadvantage with such networks is that the queries may not always be resolved. A popular content is likely to be available at several peers and any peer searching for it is likely to find the same, but, if a peer is looking for a rare or not-so-popular data shared by only a few other peers, then it is highly unlikely that search be successful. Since there is no correlation between a peer and the content managed by it, there is no guarantee that flooding will find a peer that has the desired data. Flooding also causes a high amount of signaling traffic in the network and hence such networks typically have very poor search efficiency. Most of the popular P2P networks such as Napster, Gnutella and KaZaA are unstructured.

Structured P2P networks overcome the limitations of unstructured networks by maintaining a Distributed Hash Table (DHT) and by allowing each peer to be responsible for a specific part of the content in the network. These networks use hash functions and assign values to every content and every peer in the network and then follow a global protocol in determining which peer is responsible for which content. This way, whenever a peer wants to search for some data, it uses the global protocol to determine the peer(s) responsible for the data and then directs the search towards the responsible peer(s). Some well known structured P2P networks are: Chord, Pastry, Tapestry, CAN, Tulip.

3.5.2.1 JXTA 2.0[58]

The JXTA Protocols comprise an open network computing platform designed for peer-to-peer (P2P) computing. The set of generalized JXTA protocols enable all connected devices on the network -- including cell phones, PDAs, PCs and servers -- to communicate and collaborate as peers. The JXTA protocols enable developers to build and deploy interoperable services and applications, further spring-boarding the peer-to-peer revolution on the Internet.

The JXTA protocols are a set of six protocols that have been specifically designed for ad hoc, pervasive, and multi-hop peer-to-peer (P2P) network computing. Using the JXTA protocols, peers can cooperate to form self-organized and self-configured peer groups independent of their positions in the network (edges, firewalls, network address translators, public vs. private address spaces), and without the need of a centralized management infrastructure.

The JXTA protocols are designed to have very low overhead, to make few assumptions about the underlying network transport and impose few requirements on the peer environment, and yet are

[57] An overlay network is a computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network

[58] JXTA v2.0 Protocols Specification http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html

able to be used to deploy a wide variety of P2P applications and services in a highly unreliable and changing network environment.

Peers use the JXTA protocols to advertise their resources and to discover network resources (services, pipes, etc.) available from other peers. Peers form and join peer groups to create special relationships. Peers cooperate to route messages allowing for full peer connectivity. The JXTA protocols allow peers to communicate without the need to understand or manage the potentially complex and dynamic network topologies which are increasingly common.

The JXTA protocols allow peers to dynamically route messages across multiple network hops to any destination in the network (potentially traversing firewalls). Each message carries with it either a complete or partially ordered list of gateway peers through which the message might be routed. Intermediate peers in the route may assist the routing by using routes they know of to shorten or optimize the route a message is set to follow.

The JXTA protocols work together to allow the discovery, organization, monitoring and communication between peers:

- *Peer Resolver Protocol* (PRP) is the mechanism by which a peer can send a query to one or more peers, and receive a response (or multiple responses) to the query. The PRP implements a query/response protocol. The response message is matched to the query via a unique id included in the message body. Queries can be directed to the whole group or to specific peers within the group.
- *Peer Discovery Protocol* (PDP) is the mechanism by which a peer can advertise its own resources, and discover the resources from other peers (peer groups, services, pipes and additional peers). Every peer resource is described and published using an advertisement. Advertisements are programming language-neutral metadata structures that describe network resources. Advertisements are represented as XML documents.
- *Peer Information Protocol* (PIP) is the mechanism by which a peer may obtain status information about other peers. This can include state, uptime, traffic load, capabilities, and other information.
- *Pipe Binding Protocol* (PBP) is the mechanism by which a peer can establish a virtual communication channel or pipe between one or more peers. The PBP is used by a peer to bind two or more ends of the connection (pipe endpoints). Pipes provide the foundation communication mechanism between peers.
- *Endpoint Routing Protocol* (ERP) is the mechanism by which a peer can discover a route (sequence of hops) used to send a message to another peer. If a peer “A” wants to send a message to peer “C”, and there is no known direct route between “A” and “C”, then peer “A” needs to find intermediary peer(s) who will route the message to “C”. ERP is used to determine the route information. If the network topology changes and makes a previously used route unavailable, peers can use ERP to find an alternate route.
- *Rendezvous Protocol* (RVP) is the mechanism by which peers can subscribe or be a subscriber to a propagation service. Within a peer group, peers can be either rendezvous peers or peers that are listening to rendezvous peers. The Rendezvous Protocol allows a peer to send messages to all the listening instances of the service. The RVP is used by the Peer Resolver Protocol and by the Pipe Binding Protocol in order to propagate messages.

All of these protocols are implemented using a common messaging layer. This messaging layer is what binds the JXTA protocols to various network transports.

Each of the JXTA protocols is independent of the others. A peer is not required to implement all of the JXTA protocols to be a JXTA peer. A peer only implements the protocols that it needs to use (i.e. a device may have all the necessary advertisements it uses pre-stored in memory, and therefore not need to implement the Peer Discovery Protocol).

Each peer must implement two protocols in order to be addressable as a peer: the Peer Resolver Protocol and the Endpoint Routing Protocol. These two protocols and the advertisements, services

and definitions they depend upon are known as the JXTA Core Specification. The JXTA Core Specification establishes the base infrastructure used by other services and applications.

The remaining JXTA protocols, services and advertisements are optional. JXTA implementations are not required to provide these services, but are strongly recommended to do so. Implementing these services provides greater interoperability with other implementations and broader functionality. These common JXTA services are known as the JXTA Standard Services

3.5.2.2 Gnutella 0.6[59]

Gnutella is a decentralized peer-to-peer system. It allows the participants to share resources from their system for others to see and get, and locate resources shared by others on the network. Resources can be anything: mappings to other resources, cryptographic keys, files of any type, meta-information on keyable resources, etc.

Each participant launches a Gnutella program, which will seek out other Gnutella nodes to connect to. This set of connected nodes carries the Gnutella traffic, which is essentially made of queries, replies to those queries, and also other control messages to facilitate the discovery of other nodes.

Users interact with the nodes by supplying them with the list of resources they wish to share on the network, can enter searches for other's resources, will hopefully get results from those searches, and can then select those resources amongst the results: if those resources are files, for instance, they can download them. But one can imagine other types of resources that, once fetched, will bring more than their content value.

Resource data exchanges between nodes are negotiated using the standard HTTP protocol. The Gnutella network is only used to locate the nodes sharing those resources.

The Gnutella protocol defines the way in which servents [60]communicate over the network. It consists of a set of messages used for communicating data between servents and a set of rules governing the inter-servent exchange of messages. Currently, the following messages are defined:

- *Ping*: Used to actively discover hosts on the network. A servent receiving a Ping message is expected to respond with one or more Pong messages.
- *Pong*: The response to a Ping. Includes the address of a connected Gnutella servent, the listening port of that servent, and information regarding the amount of data it is making available to the network.
- *Query*: The primary mechanism for searching the distributed network. A servent receiving a Query message will respond with a Query Hit if a match is found against its local data set.
- *QueryHit*: The response to a Query. This message provides the recipient with enough information to acquire the data matching the corresponding Query.
- *Push*: A mechanism that allows a firewalled servent to contribute file-based data to the network.
- *Bye*: An optional message used to inform the remote host that you are closing the connection, and your reason for doing so.

[59]Gnutella Protocol Development http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html

[60]A program participating in the Gnutella network is called a servent. The words "peer", "node" and "host" have similar meanings, but refers to a network participant rather than a program. When a servent has a clear client or server role the words "client" or "server" may be used. The word "client" is sometimes used as a synonym for servent. This is a contraction of "SERVer" and "cliENT"; some other documents use the word "servant" instead of servent.

3.5.2.3. Napster[61] [62]

Napster is based on a client - server architecture. The role of the server is to hold a searchable index that contains entries of mp3s that all the currently connected clients contain. The server is actually multiple very hi-spec machines load balancing the requests from clients. This makes scaling the service simply a matter of adding machines into the server pool and ensures redundancy in the fact that servers can fail and be replaced without significant disruption to the service they are providing. Redundancy needs to be implemented for the connection between client and server as well so the servers are placed on multiple connections to different large ISPs.

The clients have the functionality of being able to index and associate meta-data with shared mp3s on their own machine. This information is then sent to the Napster servers when connecting. At this point the client may search all clients connected on Napster by sending search queries to the Napster server. The server will search its internal indexes of currently shared files and return results to match. The results contain the meta-data about the file, the location of the file and speed of the clients that are sharing the files. If the client wishes to download one of the files contained in the search results then it connects directly to the other client sharing the file and begins the download. The file itself never passes through or is stored on the Napster server. This is the peer-to-peer aspect of the protocol.

Client-Server protocol

Napster uses TCP for client to server communication. Each message to/from the server is in the form of <length><type><data> where <length> and <type> are 2 bytes each. <length> specifies the length in bytes of the <data> portion of the message. Be aware that <length> and <type> appear to be in little-endian format (least significant byte goes first).

Client-Client Protocol

File transfer occur directly between clients without passing through the server. There are four transfer modes, upload, download, firewalled upload, and firewalled download. The normal method of transfer is that the client wishing to download a file makes a TCP connection to the client holding the file on their data port. However, in the case where the client sharing the file is behind a firewall, it is necessary for them to "push" the data by making a TCP connection to the downloader's data port.

3.5.2.4. Bittorrent[63]

BitTorrent is the name of a peer-to-peer (P2P) file distribution protocol, and is the name of a free software implementation of that protocol. The protocol identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over plain HTTP is that when multiple downloads of the same file happens concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load.

[61] Peer-to-Peer Technologies and Protocols http://ntrg.cs.tcd.ie/undergrad/4ba2.02/p2p/index.html

[62] Napster Messages http://opennap.sourceforge.net/napster.txt

[63] BitTorrent.org for Developers: http://www.bittorrent.org/protocol.html

A BitTorrent file distribution consists of these entities:

- An ordinary web server
- A static 'metainfo' file
- A BitTorrent tracker
- An 'original' downloader
- The end user web browsers
- The end user downloaders

To start serving, a host goes through the following steps:

- Start running a tracker (or, more likely, have one running already).
- Start running an ordinary web server, such as apache, or have one already.
- Associate the extension .torrent with mimetype application/x-bittorrent on their web server (or have done so already).
- Generate a metainfo (.torrent) file using the complete file to be served and the URL of the tracker.
- Put the metainfo file on the web server.
- Link to the metainfo (.torrent) file from some other web page.
- Start a downloader which already has the complete file (the 'origin').

To start downloading, a user does the following:

- Install BitTorrent (or have done so already).
- Surf the web.
- Click on a link to a .torrent file.
- Select where to save the file locally, or select a partial download to resume.
- Wait for download to complete.
- Tell downloader to exit (it keeps uploading until this happens).

BitTorrent's peer protocol operates over TCP. It performs efficiently without setting any socket options.

Peer connections are symmetrical. Messages sent in both directions look the same, and data can flow in either direction.

The peer protocol refers to pieces of the file by index as described in the metainfo file, starting at zero. When a peer finishes downloading a piece and checks that the hash matches, it announces that it has that piece to all of its peers.

Connections contain two bits of state on either end: choked or not, and interested or not. Choking is a notification that no data will be sent until unchoking happens.

Data transfer takes place whenever one side is interested and the other side is not choking. Interest state must be kept up to date at all times - whenever a downloader doesn't have something they currently would ask a peer for in unchoked, they must express lack of interest, despite being choked. Implementing this properly is tricky, but makes it possible for downloaders to know which peers will start downloading immediately if unchoked.

Connections start out choked and not interested.

When data is being transferred, downloaders should keep several piece requests queued up at once in order to get good TCP performance (this is called 'pipelining'.) On the other side, requests which can't be written out to the TCP buffer immediately should be queued up in memory rather

than kept in an application-level network buffer, so they can all be thrown out when a choke happens.

The peer wire protocol consists of a handshake followed by a never-ending stream of length-prefixed messages. The handshake starts with character nineteen (decimal) followed by the string 'BitTorrent protocol'. The leading character is a length prefix, put there in the hope that other new protocols may do the same and thus be trivially distinguishable from each other.

After handshaking, next comes an alternating stream of length prefixes and messages. Messages of length zero are keepalives, and ignored. Keepalives are generally sent once every two minutes, but note that timeouts can be done much more quickly when data is expected.

Downloaders generally download pieces in random order, which does a reasonably good job of keeping them from having a strict subset or superset of the pieces of any of their peers.

3.5.2.5 Kademlia

(From (Maymounkov, et al., 2002)).

Kademlia is a distributed hash table for decentralized peer to peer computer networks. It specifies the structure of the network, regulates communication between nodes and how the exchange of information has to take place. Kademlia nodes communicate among themselves using the transport protocol UDP. Kademlia nodes store data by implementing a distributed hash table. Over an existing LAN/WAN (like the Internet), a new virtual or overlay network is created in which each node is identified by a number ("Node ID"). This number serves not only as its identification, but the Kademlia algorithm uses it for further purposes.

A node that would like to join the net must first go through a bootstrap process. In this phase, the node needs to know the IP address of another node (obtained from the user, or from a stored list) that is already participating in the Kademlia network. If the bootstrapping node has not yet participated in the network, it computes a random ID number that is not already assigned to any other node. It uses this ID until leaving the network.

The Kademlia algorithm is based on the calculation of the "distance" between two nodes. This distance is computed as the exclusive or of the two node IDs, taking the result as an integer number.

This "distance" does not have anything to do with geographical conditions, but designates the distance within the ID range. Thus it can and does happen that, for example, a node from Germany and one from Australia are "neighbours".

Information within Kademlia is stored in so called "values", every value being attached to a "key".

When searching for some key, the algorithm explores the network in several steps, each step approaching closer to the searched-for key, until the contacted node returns the value, or no more closer nodes are found. The number of nodes contacted during the search is only marginally dependent on the size of the network: If the number of participants in the net doubles in number, then a user's node must query only one more node per search, not twice as many.

The Kademlia protocol consists of four RPCs: PING, STORE, FIND NODE, and FIND VALUE. The PING RPC probes a node to see if it is online. STORE instructs a node to store a <key; value> pair for later retrieval. FIND NODE takes a 160-bit ID as an argument. The recipient of a RPC returns <IP address; UDP port; Node ID> triples for the k-nodes it knows about closest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about).

FIND VALUE behaves like FIND NODE—returning <IP address; UDP port; Node ID> triples—with one exception. If the RPC recipient has received a STORE RPC for the key, it just returns the stored value. The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID.

The Kad Network (supported by eMule, MLDonkey and aMule) is a peer-to-peer network which implements the Kademlia P2P overlay protocol

3.5.2.6 FastTrack[64]

FastTrack is another distributed file sharing protocol used by a number of clients. Unfortunately the protocol is a well kept company secret and details are not available to the public. Some of the applications using the FastTrack protocol are KaZaA, KaZaA Lite, Grokster, iMesh.

FastTrack uses a semi-distributed and hierarchical architecture to achieve performance greater than that of Gnutella. Even though Gnutella introduced ultrapeers, FastTrack based clients perform better on most occasions. Since the protocol is a company secret, where a license needs to be obtained through the company Sherman Networks, only minor details have been available, meaning that the architecture described could have been changed recently.

The FastTrack protocol classifies some nodes as super nodes. These nodes act as directory servers for other clients and are elected without centralized control. It is certainly possible that more roles exist. There is probably some kind of aggregation between the super nodes as well, but this has not been proven. The supernode functionality is built into the client; if a powerful computer with a fast network connection runs the client software, it will automatically become a supernode, effectively acting as a temporary indexing server for other, slower clients.

In order to be able to initially connect to the network, a list of supernode IP numbers is stored in the program. The client attempts to contact these, and as soon as it finds a working supernode, it requests a list of currently active supernodes, to be used for future connection attempts. The client picks one supernode as its "upstream" and uploads a list of files it intends to share to that supernode. It also sends search requests to this supernode. The supernode communicates with other supernodes in order to satisfy search requests. The client then connects directly to a peer to download the file.

Some FastTrack clients also uses a reputation system which encourages users to share files and allow uploads. For example, KaZaA Lite users' reputation is reflected by their participation level, which is a number that is well encapsulated by encryption. A user starts at participation level 10 and can get a participation level between 0 and 1000. A high participation level means that the client has been connected for long periods of time and allowed many users to benefit from it. Users with higher participation level are favored in queuing policies and should receive better quality of service (QOS).

FastTrack runs on top of both UDP and TCP. Clients receive fewer packets per minute compared to Gnutella clients. FastTrack does not maintain TCP connections for longer periods of time, unless it is a download or upload. FastTrack uses a simplified version of HTTP to perform the actual downloads. This makes it possible for users to bypass the regulations set by the client on the maximum number of simultaneous downloads. Earlier versions of FastTrack clients even allowed a user to download files from itself using a web browser and thereby fooling the reputation system to believe that the client had contributed a lot to the network.

To allow downloading from multiple sources, FastTrack employs the UUHash hashing algorithm. While UUHash allows very large files to be checksummed in a short time, even on slow computers, it also allows for massive corruption of a file to go unnoticed. Many people, as well as the RIAA, have exploited this vulnerability to spread corrupt and fake files on the network.

[64] FastTrack: http://www.cs.umu.se/~bergner/thesis/html/node62.html

3.5.2.7 Chord

(From (Stocia, et al., 2001)).

Chord is a distributed lookup protocol that addresses the problem of discovering efficiently the location of a node that stores a particular data item in a peer-to-peer application. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing.

The Chord protocol specifies how to find the locations of keys, w new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes.

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses consistent hashing (Karger, et al., May 1997) (Lewin, 1998). With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an N^{th} node joins (or leaves) the network, only an *O(1/N)* fraction of the keys are moved to a different location - this is clearly the minimum necessary to maintain a balanced load.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about *O(log N) other* nodes, and a lookup requires *O(log N)* messages.

Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

3.6 Remote plug-in installation protocols, standards and platforms

The NeOn toolkit will provide and architecture that will be extended using plug-ins. The technology that will be behind for allowing this extension is OSGI.

3.6.1 OSGI

OSGI (Open Service Gateway Initiative)[65] was originally developed for embedded systems. OSGI specifies an open, java-based platform for services. It goes back to the OSGI alliance, which was founded in 1999. The OSGI alliance provides the following definition:

> The OSGi™ specifications define a *standardized*, *component oriented*, computing environment for *networked services* that is the foundation of an enhanced *service oriented architecture*[66].

OSGI defines a lifecycle model and a registry for services. Services are specified by Java™-interfaces. It supports server-based administration and provides a number of standard services. Those cover administrative tasks, security, protocols and I/O (and more).

[65] Not to be mistaken for OGSI, the Open Grid Services Infrastructure

[66] OSGI Technical Whitepaper published by OSGI alliance: http://www.osgi.org

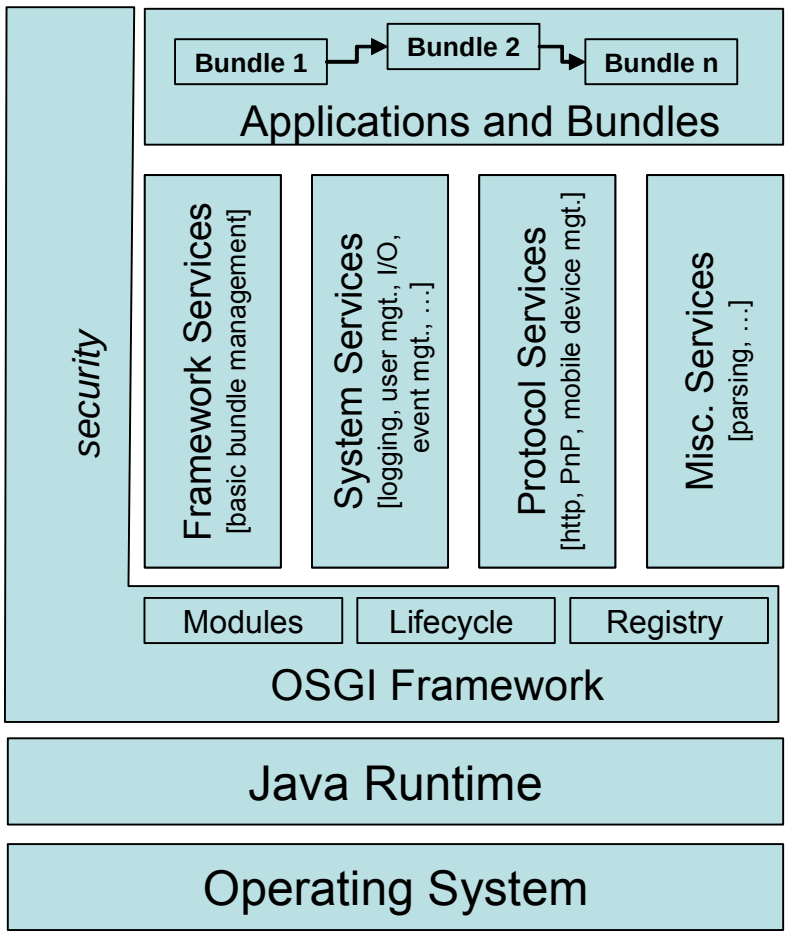

Figure 26: OSGI Architecture

Figure 26 shows the OSGI architecture. The basic building blocks are the OSGI framework and a set of standard services. The framework itself provides modularization of applications or bundles[67] (using class-loading policies), lifecycle management (e.g. starting and stopping applications) and a service registry, that allows the dynamic interaction of services (sharing of objects). Security features are provided orthogonal to those functions ("ubiquitous security"). It uses security mechanisms of the Java platform (VM security features, language features) and adds security mechanisms for bundles on top of this.

Being designed for embedded devices, OSGI is designed for remote management. OSGI itself does not enforce a protocol but allows encapsulating it by API calls and providing this API through the management system.

OSGI supports the sharing of code as well as services. Code sharing is useful if different (dynamically managed) applications use the same library. OSGI provides means of dealing with dependencies on the same library but in different version.

Services allow sharing objects, which are specified by interfaces. This is supported by a registry, which enables bundles to register objects with the registry, search the Service Registry for matching objects and receive notifications when services become registered or unregistered.

The dynamic support for the modularization of applications is one the characteristics that has made OSGI attractive not only in the context of embedded devices. OSGI implementations (see below) cover a range of applications proving this fact.

[67] Usually modularized applications are referred to as "bundles" in the OSGI context

OSGI Implementations

There are a number of OSGI implementations, including (but not limited to):

- Knopferfish OSGI[68].
- Apache Felix[69].
- Eclipse Equinox[70].

The last one has gained some popularity due to the fact that it's part of the eclipse platform since release 3.2. It supports OSGI R4, including *declarative services*. The latter are a certain form of dependency injection, which allows the configuration of dependencies and the declaration of services based on XML-files. This includes an "intelligent" class loading strategy, resolving dependencies dynamically ("lazy loading"). However, eclipse 3.2 does not yet support declarative services in such a way that tool support is available in a similar way as for the "traditional" eclipse extension mechanism.

OSGI for Server-Side Applications

Recently there have been several approaches to combine OSGI implementations with server-based technology. This is seen as a possibility to combine the advantages of OSGI (modularization, dynamic management of modules, versioning of modules) with the capabilities of application server technology. This can be achieved by either embedding OSGI on an application server (leaving start-up control to that server) or embedding web-server technology in the OSGI runtime. The latter option just means to create a specific web-server bundle and integrate it.

[68] Knopferfish OSGI http://www.knopflerfish.org/

[69] Felix: http://cwiki.apache.org/FELIX/index.html

[70] Eclipse Equinox: http://eclipse.org/equinox/

4. Social protocols through the literature on collaboration

4.1 Introduction

As shown in Section 2 of (Catenacci, et al.) social protocols, in terms of the literature on collaboration in cooperative knowledge communities may be presented by focussing on requirements, on tools or on their matching.

In general, the issues that an overview on such literature makes clear are two:

1. *The notion of collaboration is not univocal. Moreover, despite having been widely treated in the literature, none of the existing treatments provides a sufficiently general definition for it*. The tables below, 2.1 (Example of comparison between collaboration requirements) and 2.2 (Example of comparison between collaboration tools), provide a synthetic view on this claim. As a matter of fact, each proposal (i.e. each column) defines specific requirements or functionalities for given groups of users. But the proposals do not describe their intended groups of users, of requirements or of functionalities in terms of a common conceptualisation. In other words, the headings of each row in the two tables are by no means an ontology, but simple abstractions. Furthermore even by sticking to such simple abstractions, the comparison between the different proposals does not become necessarily easier. This is a consequence of the fact that the each proposal evaluates the same row in a different way or at a different level of detail. For instance, group definition in Table 2.1 is achieved either by enabling mutual recognition, or by defining group boundaries, or by promoting continuity; but it remains unclear how these three different ways of coping with group definition relate to one another. Furthermore, again in Table 2.1, requirements on information management and group management in DILIGENT are much more detailed than, for instance, in the proposal of (Axelrod, 1984); and the consequence of this are the gaps in the first and the second column. The situation just described makes it impossible to adopt any of the existing proposals as a basis for the definition of a language to talk about collaboration.
2. *The lack of generality of existing proposals mainly is a consequence of the social-technical gap* between social requirements and technical feasibility: “the divide between what we know we *must* support socially and what we *can* support technically” (Ackerman, 2001). Most existing proposals approach this divide from the technical side. None of them attempts at providing a conceptual framework of the collaborative use of existing or forthcoming technology.

The situation described in 1 and 2 above is the reason behind the attempt made in (Catenacci, et al.) at providing NeOn with a general enough understanding of the notion of collaboration (and of collaborative ontology design) by means of a new proposal: the Collaborative Ontology Design Ontology (C-ODO).

Section 2, 3, and 4 below present a summary of the literature on requirements, tools and their matching. Section 5 gives a summary of C-ODO. For extended versions of all this material refer to (Catenacci, et al.).

Collaboration requirements	Evolution of Cooperation (Axelrod, 1984)	Evolution of Institutions (Ostrom, 1990)	How virtual communities work (Godwin, 1994)	DILIGENT (Vrandecic, 2005), (Tempich, 2006)
Group definition	Enable mutual recognition	Define group boundaries	Promote continuity	
Information management	Circulate information	-	Use good discussion software	Communicate changes in the ontology
	-	-	Provide institutional memory	Integrate control and argumentation support
	-	-	-	Graphic representations of ontology
Group management	Arrange meetings	Rule use of collective goods	Users resolve their own disputes	Use argumentation
	-	Monitor members	Confront users with a crisis	Use a clear methodology
		Sanction members		Use clear decisions processes

Table 2.1 Example of comparison between collaboration requirements

Collaboration tools	**OBOS** (Das, et al., 2001)	**HACM (Compendium-based)** (Shum, et al., 2002)	**Claimspotter** (Sereno, et al., 2004)
Types of users	Ontologists	Ontologist	Scholars
	Domain experts	Scientists	-
	Business analysts	-	-
Functionalities	Ontologies development	Structure collaborative sense making	Support annotation of scholarly documents
	Ontologies maintenance	Aid group memory	Create triples (source destination, relation between them (e.g. source 'proves', destination)
	Multiple access	Dialog mapping	-
	Discussion rooms	Direct formalization of conceptual proposals	-
	-	Direct display of proposals on screen	-

Table 2.2 Example of comparison between collaboration tools

4.2 Requirements for Collaboration

This section presents schematic overviews of proposals of requirements for collaboration.

Requirements for the possibility of cooperation (Axelrod, 1984):

- Arrange that individuals will meet each other again.
- They must be able to recognize each other.
- They must have information about how the other has behaved until now.

Design principles of successful communities (Ostrom, 1990):

- Group boundaries are clearly defined.
- Rules governing the use of collective goods are well matched to local needs and conditions.
- Most individuals affected by these rules can participate in modifying the rules.
- The right of community members to devise their own rules is respected by external authorities.
- A system for monitoring members' behavior exists; this monitoring is undertaken by the community members themselves.
- A graduated system of sanctions is used.
- Community members have access to low-cost conflict resolution mechanisms.

Principles for making virtual communities work (Godwin, 1994):

- Use software that promotes good discussion.
- Don't impose a length limitation on postings.
- Front-load your system with talkative, diverse people.
- Let the users resolve their own disputes.
- Provide institutional memory.
- Promote continuity.
- Be host to a particular interest group.
- Provide places for children.
- Confront the users with a crisis.

Requirements in DILIGENT (Vrandecic, 2005) (Tempich, 2006):

- Engineering tools should have support for communicating changes in the collaboratively developed ontology.
- All other required tools like version control and argumentation support should be strongly integrated into the engineering tool.
- It is important to have a graphical visualization of the ontology.
- Use a concrete methodology for the collaborative development process to clarify the objectives and to have a list of things to do next. The methodology should cover the whole ontology lifecycle including the maintenance phase and not only the initial creation of the ontology. With this respect, it proved to be useful to have a well-defined process for feeding back change requests and to document the argumentation process which led to certain design decisions.

- Find good evaluation measures which show whether one reached the goals which were originally set for the ontology.
- Use an argumentation framework. Discussions are often inefficient and time consuming if a clear structure is missing. For example, it is useful to restrict the users to certain argument types so that they didn't get lost in the discussion.
- Use a clear decision process has to be defined which of the proposed solutions should be included into the ontology. Many of the requirements for ontology engineering tools also apply for the argumentation tool.
- The user needs a possibility to monitor a discussion so that she/he is automatically informed of changes to discussions of interest.
- The argumentation tool should be integrated with the ontology engineering tool so that one can access the argumentation data from the engineering tool and vice versa.

Requirements for efficient version management and control system (Noy, et al., 2006):

- Use software that promotes good discussion.
- It is very important for the users that they can attach annotations to their changes which explain the rationale and/or which refer to citations and documents on which the change is based.
- Do not compute textual differences between e.g. two OWL ontologies but that instead a list of changed ontology elements, including information about e.g. which concepts were split or merged, an information typically not available if the changes were computed based on textual differences.
- The description of changes is needed in such a granularity that it is possible to go back to earlier versions of an ontology at any time.
- Having fine grained access rights. It is especially not sufficient to define the access on the level of an ontology. Instead it should be possible to define it on the level of ontology elements. This helps to avoid conflicts between the different versions of editors. Nevertheless, before checking in a new version it is necessary to identify direct and indirect conflicts between two versions. Indirect conflicts may e.g. occur in subclasses that depend on one of the changed classes. In case that a conflict occurs, a kind of negotiation process between editors is needed which helps to resolve the conflict. In some collaborative scenarios, there exists a central authority or curator, which decides which local changes of editors will be included in the shared version of an ontology. In this case, the curator needs the possibility to accept a whole set of changes. This set of changes may be identified structurally or based on who performed the change and when. Otherwise, accepting each single change separately would be very tedious. In this scenario, the automatic detection of conflicts as well as the annotations made by the authors are very useful for the curator as they explain why a change was necessary.

4.3 Tools for Collaboration Support

This section presents overviews of tools for collaboration.

Ontology Builder and Ontology Server (OBOS) (Das, et al., 2001). An application suite proposed in and developed for supporting the creation and maintenance of ontologies used in e-commerce and B2B applications. There is not a precise definition of collaboration, the issue is approached focusing mainly on tool functional requirements identification. OBOS has been built with the aim of supporting a distributed and collaborative team of users (ontologists, domain experts, and business analysts) developing and maintaining shared ontologies. Basing on an informal evaluation of four existing tools (i.e., Ontolingua/Chimaera, Protégé/PROMPT, OntoWeb/Tadzebao, Ontosaurus/Loom), a set of requirements is identified. Among them the following are very important for collaborative ontology creation: scalability, availability, reliability, performance, ease of use, distributed multi-user collaboration support, security management, difference and merging support, internationalization, and versioning. OBOS uses a frame-based representation based on OKBC knowledge model and its implementation is based on J2EE. The tool provides a collaborative environment for the development and maintenance of shared ontologies. Multiple access is managed using a role-based policy, and users are provided with discussion rooms where they can communicate about their work on the ontology/ies. OBOS implements a pessimistic locking strategy for editing and changes to the ontologies are immediatly notified so as the user can refresh the information. Multilinguality is supported by means of so called locales, versioning is not supported. The tool resulted to be sufficiently easy to use (as claimed by the authors), nevertheless the different types of users (ontologist, domain expert, and business analyst) are not provided with specific interfaces and/or methods.

Hypertext-Augmented Collaborative Modelling (HACM) (Shum, et al., 2002). An application proposed in is based on a Compendium approach. Within the Advanced Knowledge Technology (AKT) consortium the AKTive Portal was designed to be a next generation portal infrastructure that supports the capture, indexing, dissemination and querying of information. The first application of the portal was to the AKT project itself. Mifflin, a hypertext tool for Compendium, was used to facilitate ontology-based scientific knowledge creation and management in collaborative settings and, interestingly, the case-studies were AKT meetings. Mifflin's main functions in the project were to provide:

1. Structure to collaborative sense making;
2. The rationale for an ontology engineer when implementing the agreed specification;
3. A memory aid in and between meetings for both the group and for the group coordinator;
4. Multiple on screen visualizations of both the existing ontology structure and of the ongoing discussion about it.

Mifflin succeeded in supporting the collaborative creation of an ontology of scientific knowledge, mainly in terms of:

1. Dialog mapping,
2. Direct formalization of conceptual proposals,
3. Direct display of proposals on screen,
4. Compatibility with existing software tools.

The achievement of these four results was not cost-free, though. Mifflin imposed on (even expert) users the development of some literacy and, at the beginning, some cognitive overhead.

Claimspotter The application presented in (Sereno, et al., 2004) is an open architecture based on the ScholOnto ontology. Claimspotter supports the semiformal (collaborative) annotation of scholarly documents. It is based on a simple paradigm: triples. The text of a document is represented by couples of concepts (source and destination concept) plus a relation between them (for instance: 'is an example of', 'is enabled by', 'proves', 'supports', 'is similar to', etc.). Such triples allow building a network of claims about the internal structure of the document, or about its relations with other documents. The resulting network, or parts of it, can be shared and incremented by different users over time. The final result is a commentary to the original text, which is usually dialectic, as the network can host logically contradictory claims about the contents of the document.

Claimspotter supports the creation of triples by means of suggestions given to the user. On the one hand, suggestions have to do with the structure of the document and the scientific rhetoric the keeps it together. Two main families of rhetorical roles are considered: what in the document refers to the work being described (background, aim, textual structure) and what refers to the the work of other researchers (contrast, basis). On the other hand, suggestions have to do with so-called information bricks, i.e. parts of the document, which may be used as concepts in the network (keywords, the instances of ScholOnto relations found in the text - i.e. verbal expressions -, cited documents).

Claimspotter's architecture as well as the presentation of the suggestions is highly modular. A toolbar gathers all different suggestions on the following aspects: the concepts made by the current annotator, the instances of ScholOnto relations found in the text, the documents important sentences (where importance is defined in terms of keyword-matching with title, headers, and abstract), the document's rhetorically-consistent zones, the sentences matching a particular user-defined query expressions.

A very limited user study has been done for evaluation of Claimspotter. This has revealed two main points. On the hand, an annotation system should be very flexible with respect to the quantity and the quality of suggestions provided to the user. Users want to be able to switch back and forth from a very structured configuration (where to get support and inspiration from what other annotators have done) to a lightweight configuration (where to "think outside the box"). On the other, it has become clear that gaining expertise with the system corresponds for annotators to move from a "concepts to relations approach" (which tends to produce idiosyncratic networks) to a "relations to concepts approach" (which facilitates standardization).

Ontology of the Academic Field presented in (Benn, et al., 2005) generalizes that approach (in terms of an, rather than of scholarly comment only) and it makes one step towards automation (in terms of a number of functionalities that allow to derive knowledge from a model based on the ontology).

The Ontology of the Academic Field comprises three main components:

1. the Community of Practice (with concepts, attributes and relations like Publication, Title, author-of, researcher-at, etc.);
2. the Lexicon (with concepts, attributes and relations like Lexical-Term, Gloss, broader-term, etc.);
3. the Argumentative Discourse (with concepts, attributes and relations like Statement, Question, Issue, Premise, Conclusion, Postulates, supports, coheres).

Services are provided for:

1. The usual bibliographic database functions provided by tools like CiteSeer or Google Scholar;
2. Finding key statements made by an author on a particular issue;
3. Assisting navigation around a complex argumentation network, which renders an ontology as an interactive map;

4. Flexible visualization of the network;
5. Inferences on the network, by creating paths, like for instance, (in)coherence paths that connect the opinions of a given author with a scholarly position that is typical of the reference field.

ClaiMaker An application introduced in (Mancini, et al., 2006) designed to represent discourse in a semiotic way within the scholarly domain. More generally, the paper discusses the representational requirements for collaborative systems that support sensemaking and argumentation over contested topics. Sensemaking is intended as expressing and contesting explicit, possibly competing views of the world. Supporting sensemaking therefore means supporting a way of annotating different interpretations of the same object or issue. This is what ClaiMaker does, with a theoretical backbone consisting of semiotic (in a Saussurian fashion) and coherence relations, as in Mann and Thompson's Rhetorical Structure Theory (RST) (Mann, et al., 1988).

ClaiMaker is a hypertext system that makes use of constrained base relational classes, but imposing no constraints on how such classes are rendered, or on how nodes are expressed/classified. ClaiMaker's ontology allows users to establish as many referential relations between concepts (e.g. the summary message of a document) and sources (e.g. a document) and also connective relations between sources. Claims of the first kind are called "primary claims", whereas those of the second type are called "secondary claims".

1. *Primary claim*: users can associate documents with concepts: this consists in establishing a referential relation between a concept and a referent. In other words: a primary claim is the creation of a sign (=the concept) that refers to a particular referent (=the document) in the virtual reality (=the ClaiMaker repository), in some respect (=a context). From an ontological point of view, it is interesting to note that concepts linked to sources can (optionally) be classified. The classification is not rigid, however, so that the same concepts can be assigned different classes by two different persons, or even by the same one in different contexts. Like in natural language, in ClaiMaker meaning is continually negotiated by means of establishing referential relations between referents and concepts, and by means of defining concepts according to different classes (different from ontology-based systems).
2. *Secondary claim*: A secondary claim establishes a discourse connection between two concepts. The authors borrow plenty of terminology and insights from linguistic theories, such as RST and Sanders et al.'s theory of connectives (Sanders, et al., 1993) to model what they term an "upper level discourse relations ontology". Grounding on Sanders et al's approach, they treat coherence relations as psychological constructs and take a small number of cognitively basic concepts. The relational scheme is based on four parameters (Cognitive Coherence Relations [CCR]): Basic operation [additive, causal], Source of Coherence [semantic, pragmatic], Order [basic, non-basic], Polarity [positive, negative]

A relational hierarchy is then derived from these four parameters. By also incorporating insights from Louwerse's (2001) description of coherence relations (Louwerse, 2001), the authors obtain the final ClaiMaker's relational ontology, which can be used for annotation of secondary claims. Since it is based on cognitive primitives, it has the main advantage of being applicable in different disciplines and domains.

Co-OPR project (Bunemann, et al., 2006) presents the integration of two existing tools (i.e., Compendium, and I-X) for the, the simulation of a personnel recovery mission. The experiment presented deals with decision-making support for a team collaborating on the same mission. In particular, Compendium has been used in order to support the collaboration between members of the team who were geographically distributed; I-X has been involved as a tool supporting a team whose members were physically in the same place. The paper underlines the effectiveness and usability of the two tools when used together by giving a very pragmatic evaluation. The focus is mainly on the usability and utility

of provided functionalities.

4.4 Matching requirements and tools

Tools that support collaboration are obviously developed on the basis of user requirements. Some valuable insights come from comparing such requirements and available tools, as to identify not only the technical gaps, but rather determine which gaps can be bridged by advancing technology and which are instead unavoidable (i.e. which requirements are unsupportable).

An important contribution is Mark Ackerman's work on the gap existing between social requirements and technical feasibility (Ackerman, 2001). What Ackerman terms "the *social-technical gap*" is "the divide between what we know we *must* support socially and what we *can* support technically", and is likely to be the highest challenge of Computer-Supported Cooperative Work (CSCW). Within NeOn, it is not only relevant the description of social requirements (in collective work), but also the attention to what kind of support is difficult to achieve technically, and therefore a definition of an upper bound with respect to tool development for supporting collaborative activities.

The following are some social aspects of communication that need to be considered when building any tool that supports collaborative activities

- *Nuances of social activity*: social activities are fine-grained and flexible, thus making systems technically difficult to build.
- *Multiplicity and Diversity of Goals*: members of a given organisation might have different goals and different organisations may not have shared goals, knowledge, and meanings. Conflict is as important as cooperation in issue resolution. Meanings must be negotiated, for example (see also (Mancini, et al., 2006) about the importance of building tools that support negotiation of "sense making", such as ClaiMaker (Mancini, et al., 2006)).
- *Exceptions* in work processes are normal. And roles can often be informal and fluid CSCW approaches to workflow should deal with exceptions and fluidity.
- *Visibility of communication exchanges and information* facilitates learning but might inhibit for fear of criticism. Ways must be found to manage the trade-offs in sharing.
- *Norms for using CSCW*: they are set (negotiated) by the users and can change while using the system. The system must therefore allow for renegotiation, changes and flexibility (see again (Mancini, et al., 2006)).
- *Critical Mass*: with an insufficient number of users, people will not use a CSCW system.
- *Adaptation*: people adapt their systems to their needs, so not everything can be forseen when developing a system; however, systems are often too rigid to allow for such changes.
- *Incentives*: using a tool might be time consuming, also from a learning-to-use-it point of view. So it must be rewarding, i.e. benefits must be evident.

Additionally, (Porzel, et al., 2004) claim how one of the main failures of human-computer interaction (HCI) is the treatment of *turn taking*. Experiments are presented that show that HCI systems are not equipped with means for dealing with natural turn-taking issues, such as pauses, overlaps, and similar behaviour. Although this applies to human-machine interaction, in the spoken dialogue domain there might be similar problems in collaborative activities between humans conducted over the Web, especially if done in a synchronous manner (see also (Chandler, 2001)).

In the specifics of collaboration towards ontology development, (Lu, 2003) is a source of interesting points with respect to requirements and tool support. According to (Lu, 2003), since modern ontologies are characterized by their huge size and high complexity, ontology engineering is to be considered an inherently collaborative activity, involving the effort of many domain experts and software developers which are often not co-located. This is especially prominent when not only the initial design stage, but the whole ontology life cycle is considered. Throughout this work, 'collaboration' seems to be defined as a reiterated process, the output of which, at each of the

involved stages, is the obtaining of a 'convergence of views'.

Based on a survey of five authoring tools (Ontolingua Server, OntoEdit, APECKS, CO4, and Protégé-2000), which were widely used at the time of the inquiry (updated in 2000), the conclusion is reached that collaborative ontology development was – again, at that time – far from being well supported by said tools.

The identified inefficiencies with respect to collaboration support were the following:

- Coordinated group work (e.g. collaborative editing, discussion or annotation) was not well supported, mainly because the systems lack functions for keeping developers informed of each other's activities (compare, by contrast, with current wikis)
- Contextual communication support was not considered in these tools, i.e. no way was provided for easily keeping track of a whole coherent discussion, which is e.g. scattered in many people's mailboxes (compare, by contrast, with Compendium, Claimspotter, and ClaimMaker below)

Since collaborative work across distance in software engineering and ontology engineering share many similar characteristics, three dimensions of collaborative ontology engineering are identified based on documents and experiences in the first field:

- *Distance and communication*: Co-located team members communicate informally anytime during the work day, while this cannot happen to geographically distributed team members (A study at Carnegie Mellon University showed that the rate at which scientists collaborated spontaneously with one another was a function of distance between offices). Informal and unplanned communication has been proved to have a direct impact on development processes, in particular on
- *Coordination* ("the act of integrating each task with each organizational unit, so each unit contributes to the overall objective"), and
- *Control* ("the process of adhering to project goals, specifications, and standards").

Coordination and control are necessary not only to manage interdependencies within the tasks, but also for the development and maintenance of *shared mental models* (compare with (Fleck, 1986) on *thought-styles*), which are considered to be the most effective support for explicit coordination in team work. Communication affects shared mental models in two ways: a) during task execution, it refines team members' mental models with contextual cues; b) it keeps the models up-to-date, especially in dynamic or novel situations. It has been proved that weak shared mental models in asynchronous tasks can lead to productivity losses. Designing tools to support and enhance informal communication is then a key step toward bridging the missing link in distributed development work.

- *Documentation and knowledge management*:

Information and knowledge obtained during meetings, email correspondences, and instant messaging need to be captured easily, stored and shared effectively. The distribution of resources and developers in space and time combined with the dynamic evolution of knowledge make the use of tools for knowledge management a necessity. Moreover, the documentation must be kept up-to-date.

- *Version control and change tracking*:

Tools for version control and change history tracking are crucial when development resources are not co-located, in order to make sure that two developers do not work on the same part of the ontology and to avoid the complication of resolving conflicts.

Finally, a range of tools and groupware technologies in the Computer Supported Collaborative Work (CSCW) domain are investigated, in order to determine how they can be used in the ontology development domain.

- *Experiences in the Global Software Development (GSD) field are examined in order to understand the correlation between distance and collaboration:*

1. instant messaging techniques (support spontaneous and informal communication)
2. web portals (support tasks in the area of group knowledge management)
3. Peer-to-Peer (P2P) network technologies (but poor reliability and security)

- *Report of an experiment where the possibility of adding collaborative support to a knowledge engineering tool based on a P2P network was evaluated*
- *Long term vision: to combine these two fields and create a collaborative ontology engineering environment that provides collaboration support in multiple dimensions:*

Finally, as far as coordination in collaborative environments is concerned, interesting suggestions may be provided by the coordination models and workflow patterns presented respectively in (Perry, et al., 1991) and (van der Aalst, et al., 2003).

4.5 C-ODO

C-ODO (Catenacci, et al.) provides a model of the collaboration in ontology design in terms of the following classes:

- *Network of ontologies*: the semantics of the relations between ontologies. Please note that because of the networked perspective we take here, design is not to be intended as limited to creation time, i.e. to an *initial* phase of an ontology lifecycle, but as an aspect of the entire ontology lifecycle.
- *Ontology element:* an (identified) element of an ontology, like a concept, a relation, an instance, an axiom, etc.
- *Knowledge resource:* any piece of knowledge that is used while working on the design of an ontology, including modules, workspaces, sources, libraries, networks, etc.
- *Ontology design rationale:* the reasons why an ontology is designed the way it is. Reasons can be grounded on *content, task, or sustainability* data. The application of ontology design rationales typically produces a choice space for a set of ontology elements.
- *Ontology project:* a project having the goal of influencing the lifecycle of a networked ontology.
- *Epistemic workflow:* a generalization over the possible relations holding between two ontology elements, as they are created, discussed, used or modified by ontology designers.
- *Collaborative workflow:* a special case of epistemic workflow, which is characterized by the ultimate goal of designing networked ontologies, and by specific relations between designers, ontology elements, and collaborative tasks.
- *Argumentation:* a structure for discussing possible design solutions, based on rationales and dialectic rules.
- *Design solution:* a state of an ontology or a part of it at time *t*.
- *Design making:* a situation in which ontology design rationales are implemented in order to obtain certain design solutions. Design making is unfolded by executing design operations that accomplish a functionality by following a method.
- *Choice space:* a space of design solutions allowed by an ontology design rationale on a set of ontology elements.
- *Design pattern:* a configuration of ontology elements that is relevant from the logical, architectural, or conceptual viewpoint.
- *Functionality:* a task to be accomplished by a design operation according to a method.

Section 3.2.3 of (Catenacci, et al.) provides a number of examples of C-ODO-based modelling of argumentation models like (Vrandecic, 2005) (Tempich, 2006) or frameworks like (van Eemeren, et al., 2003), and of collaboration patterns like (Perry, et al., 1991) and workflows like (Busi, et al., 2001).

5. Conclusions

As we stated in the introduction, the role of this document is to analyse the state of the art of technical and social protocols and techniques and related issues (such as formats and standards) for exchanging and sharing information according to the needs identified in D6.1.1.

At the time of writing this report, some candidate technologies were pre-selected from partners to be part of the NeOn platform and new ones to be developed during the project life were also identified. WP6 provided an initial list of protocols and techniques to be used in the project. The list of **technical protocols and techniques** is:

Data access protocols	
Access remote files	WebDAV
Access relational databases	JDBC
Access XML sources	XML-Schema, XQuery, XQJ
Access ontological resources	DIG
Query languages	SPARQL
Version management protocols	SVN
Service access protocols, formats and frameworks	
Access remote services	J2EE
Web services	SOAP, WSDL
Access service directories and registries	UDDI, ebXML
Notification and syndication protocols	
Notification protocols	*Not defined yet*
Syndication protocols and formats	*Not defined yet*
Network communication protocols	
Languages for working with agents	*Not defined yet*
P2P protocols	JXTA
Remote plug-in installation protocols, standards and platforms	OSGI

It is a fact that the subset of standards referred before are well established in the industry (e.g. J2EE) or they are in the way to be settled as industry standards.

As shown in Section 2 of (Catenacci, et al.) **social protocols**, in terms of the literature on collaboration in cooperative knowledge communities may be presented by focussing on requirements, tools or their matching. In general, two issues are clear after over viewing such literature, and these are:

1. The notion of collaboration is not univocal. Moreover, although this notion has been widely dealt with in the literature, none of the existing treatments provides a sufficiently general definition for it.
2. The lack of generality of existing proposals is mainly a consequence of the social-technical gap between social requirements and technical feasibility: “the divide between what we know we must support socially and what we can support technically” (Ackerman, 2001).

Most existing proposals approach this divide from the technical side. None of them attempts at providing a conceptual framework of the collaborative use of existing or forthcoming technology.

The situation described in 1 and 2 above is the reason behind the attempt made in (Catenacci, et al.) at providing NeOn with a general enough understanding of the notion of collaboration (and of collaborative ontology design) by means of a new proposal: the Collaborative Ontology Design Ontology (C-ODO).

Defining C-ODO-based social protocols would mean modelling collaborative workflows to the desired level of specificity so to constrain the design operations carried out by a knowledge collective while working on an ontology project.

References

Aalst, W. V. 2003. Don't Go with the Flow: Web Services Composition Standards Exposed. s.l. : IEEE Intelligent Systems, 2003, 18(1), pp. 72-76.

Ackerman, M.S. 2001. *HCI in the New Millennium.* [ed.] John Carroll. 2001.

Anderson, David. 2001. *Peer to Peer.* s.l. : O'Reilly, 2001.

Antonioletti, Mario, et al. 2006. *Web Services Data Access and Integration - The Relational Realization (WS-DAIR) Specification, Version 1.0.* s.l. : OGF, DAIS-WG, 21 June 2006.

ARPA Knowledge Sharing Initiative. 1993. *Specification of the KQML agent-communication language.* s.l. : External Interfaces Working Group working paper, July de 1993.

Atkinson, Malcom, et al. 2006. *Web Services Data Acces and Integration - The Core (WSDAI) Specification. Version 1.0.* s.l. : OGF, DAIS-WG, 21 June 2006.

Axelrod, R. 1984. *The Evolution of Cooperation.* New York : Basic Books, 1984.

Benn, N., Shum, S.B. and Domingue, J. 2005. Integrating Scholarly Argumentation. Text and Community: Towards and Ontology and Servicies. *Proceedings of the 5th International Workshop on Computational Models of Natural Argument (CMNA), IJCAI-05, also publicated as Technical Report KMI-05-5.* Edinburgh : s.n., 2005.

Berry, Dave, et al. 2006. *OGSA(TM) Data Architecture Version 0.6.2.* s.l. : OGF, OGSA DATA WG, 29 June 2006.

Bester, J., et al. 2003. *GridFTP: Protocol Extensions to FTP for the Grid.* [ed.] W. Allcock. s.l. : Open Grid Forum, GridFTP WG, April 2003.

Brodsky, J., et al. 2004. ICE: Information and Content Exchange Protocol. Primer: Introduction and overview. Version 2.0. [Online] 2004. http://www.icestandard.org/Spec/SPEC-ICE-2.0Primer.pdf.

Brookshier, Daniel, et al. 2002. *JXTA: Java(TM) P2P Programming.* s.l. : Sams, 2002.

Bunemann, P., et al. 2006. A Provenance Model for Manually Curated Data. *Proceedings of the International Provenance and Annotation Workshop.* 2006.

Busi, Nadia, et al. 2001. Coordination Models: A Guided Tour. [ed.] Andrea Omicini, et al. *Coordination of Internet Agents: Models, Technologies and Applications.* s.l. : Springer, 2001, pp. 6-25.

Cabral, L., et al. 2004. Approaches to Semantic Web Services: An Overview and Comparisons. [ed.] C. Bussler, et al. *Proceedings of the First European Semantic Web Symposium (ESWS2004).* Heraklion : Springer-Verlag, 2004, Vol. 3053 of LNCS, pp. 225-239.

Catenacci, C., et al. *Design rationales for collaborative development of networked ontologies - State of the art and the Collaborative Ontology Design Ontology.* Deliverable D2.1.1 of the NeOn Project.

Cederqvist, P. 2002. *Version Management CVS.* Bristol, U.K. : Network Theory Ltd., 2002.

Chandler, H.E. 2001. The complexity of online groups: a case study of asynchronous collaboration. *ACM Journal of Computer Documentation.* 2001, Vol. 25(1), pp. 17-24.

Chinnici. 2006. Web Services Description Language (WSDL) Version 2.0. 2006, Part 1: Core Language.

Crispin, M. 2003. *Internet Message Access Protocol - Version 4rev1.* March de 2003. RFC 3501.

Das, A., Wand, W. and McGuinness, D. L. 2001. Industrial Strength Ontology Management. *Proceedings ot the International Semantic Web Working Symposium.* 2001.

Davey, Stephen, et al. 2006. *Information Dissemination in the Grid Enviroment - Base Specifications.* s.l. : OGF, INFOD WG, 28 June 2006.

Esteban Gutiérrez, Miguel, et al. 2006. *DAIS RDF(S) Realization: Background and Motivational Scenarios.* [ed.] OGF. s.l. : DAIS WG, 28 July 2006.

Finin, T., et al. 1994. KQML as an agent communication language. *Proceedings of the third international conference on Information and knowledge mangement.* New York : ACM Press, 1994, pp. 456-453.

FIPA00001. FIPA Abstract Architecture Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00001/

FIPA00007. FIPA Content Language Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00007/

FIPA00008. FIPA SL Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00008/

FIPA00009. FIPA CCL Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00009/

FIPA00010. FIPA KIF Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00010/

FIPA00011. FIPA RDF Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00011/

FIPA00023. FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000. http://www.ipa.org/specs/fipa00023/

FIPA00025. FIPA Interaction Protocol Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00025/

FIPA00037. FIPA Communicative Act Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00037/

FIPA00061. FIPA ACL Message Structure Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00061/

FIPA00067. FIPA Agent Message Transport Service Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00067/

FIPA00075. FIPA Agent Message Transport Protocol for IIOP Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00075/

FIPA00076. FIPA Agent Message Transport Protocol for WAP Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00076/

FIPA00084. FIPA Agent Message Transport Protocol for HTTP Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00084/

Fleck, L. 1986. The problem of epistemology [1936]. [ed.] R.S. Cohen and T. Schnelle. *Cognition and Fact - Materials on Ludwik Fleck.* Dordrecht : s.n., 1986, pp. 81-82.

Fritson, R. and Finin, T. 1994. KQML - A Language and Protocol for Knowledge and Information Exchange. *Proceedings of the 13th Intl.* s.l. : Distributed Artificial Intelligence Workshop, 1994, pp. 127-136.

Godwin, M. 1994. Nine principles for making virtual communities work. s.l. : Wired, 1994, 2(6), págs. 72-73.

Gómez Pérez, Asunción, et al. 2006. *Ontology Access in Grids with WS-DAIOnt and the RDF(S) Realization.* 16, Athens : GGF, 3rd Semantic Grid Workshop, February 2006.

Gruber, T.R. 1995. *Collaborating around Shared Content on the WWW.* Cambridge, MA : W3C Workshop on WWW and Collaboration, 11 Sep 1995.

Gruber, T.R. 1993. Knowledge Acquisition. *A Translation Approach to Portable Ontology Specifications.* 1993, 5(2), pp. 199-200.

Gudgin. 2003. *SOAP Version 1.2 Part1: Messaging Framework.* s.l. : World Wide Web Consortium, Conference Proceedings, June 2003. W3C Recommendation.

Hastings, Shannon, et al. 2006. *Web Services Data Access and Integration - The XML Realization (WS-DAIX) Specification, Version 1.0.* s.l. : OGF, DAIS-WG, 21 June 2006.

Hethmon, P. and Elz, R. 1998. *Feature negotiation mechanism for the File Transfer Protocol.* s.l. : IETF, Network Working Group, August 1998. RFC 2389.

Horowitz, M. and Lunt, S. 1997. *FTP Security Extensions.* [prod.] Network Working Group. s.l. : IETF, October 1997. RFC 2228.

Karger, D., et al. May 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *Proceedings of the 29th Annual ACM Symposium of Theory of Computing.* El Paso : s.n., May 1997, pp. 654-663.

Klensin, J. 2001. *Simple Mail Transfer Protocol.* April 2001. RFC 2821.

Labrou, Y. and Finin, T. 1997. *A Proposal for a new KQML Specification.* Baltimore, MD 21250 : University of Maryland, Baltimore County, Computer Science and Electrical Engineering Department, February 1997. TR CS-97-03.

Lara, R., et al. 2005. *A Caonceptual Comparison between WSMO and OWL-S. WSMO Deliverable, D.4.1.v.0.1.* 2005.

Lassila, O. and Swick, R. 1999. Resource Description Framewotk (RDF) Model and Syntax Specification. [Online] 1999. http://www.w3.org/TR/REC-rdf-syntax. W3C Recommendation.

Lassila, O. 2002. The Semantic Web Kick-Off in Finland - Vision, Technologies, Research, and Applications. [ed.] E. Hyvonen. University of Helsinki : HIIT Publications, 2002, Serendipitous Interoperability.

Lewin, D. 1998. *Consisteng hashing and random trees: Algorithms for caching in distributed networks. Master's thesis.* [Available at the MIT Library] s.l. : MIT, Department of EECS, 1998.

Louwerse, M. 2001. An analytic and cognitive parametrization of coherence relations. s.l. : Cognitive Linguistics, 2001, Vol. 12(3), pp. 291-315.

Lu, Y. 2003. Roadmap for tool support for collaborative ontology engineering. Master the-sis. [Online] 2003. http://www.cs.uvic.ca/_chisel/thesis/YilingLu.pdf.

Mancini, C. and Shum, S.B. 2006. *Modelling discourse in contested domains: A semiotic and cognitive framework.* s.l. : Open University, Final version submitted to International Journal of Human-Computer Studies, 2006. Technical report kmi-06-14.

Mann, W.C. and Thompson, S.A. 1988. Rhetorical structure theory: Toward a functional theory of textorganisation. 1988, Vol. 8(3), pp. 243-281.

Martin, D., et al. 2004. *Bringing Semantics to Web Services: The OWL-S Approach.* San Diego, California, USA : Proceedings ot the First International Workshop on Semantic Web and Web Process Composition (SWSWPC 2004), 2004.

Martin, D., et al. 2003. OWL-S 1.0 white paper. [Online] 2003. http://www.daml.org/services/owl-s/1.0/.

Maymounkov, Petar and Mazieres, David. 2002. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* Cambridge, MA, USA : MIT Faculty Club, Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), March 2002.

McGuinness, D. L. and van Harmelen, F. 2003. OWL Web Ontology Language Overview. [Online] 18 August 2003. http://www.w3.org/TR/owl-features/. W3C Candidate Recommendation.

McIlraith, S., Son, S. and Zeng, H. 2001. Semantic Web Services. s.l. : IEEE Intelligent Systems, 2001, Special Issue on the Semantic Web, 16(2), pp. 46-53.

Michels, Jan-Eike. 2006. *XQuery API for Java(TM) (XQJ) 1.0 Specification Version 0.5.0 (EDR) (JSR 225 EG draft specification).* s.l. : IBM, 3 de March de 2006.

Mitra, N. 2003. SOAP Version 1.2 Part 0:Primer. [Online] 24 June 2003. http://www.w3.org/TR/soap12-part0.

Morgan, M. and Chue Hong, N. 1995. *ByteIO Specification, Version 1.0.* s.l. : OGSA-ByteIO-WG, October 1995.

Motta, E., et al. 2003. IRSII: A Framework and Infrastructure for Semantic Web Services. [ed.] D. Fensel, K. Sycara and J. Mylopoulos. *The SemanticWeb - ISWC2003, Second International Semantic Web Conference, Proceedings.* Sanibel Island : Springer-Verlag, 2003, Vol. 2870 of LNCS, pp. 306-318.

Myers, J. and Rose, M. 1996. *Post Office Protocol - Version 3.* May 1996. STD 53, RFC 1039.

Nagel, W. 2005. *Subversion Version Control: Using the Subversion Version Control System in Development Projects.* NJ : Prentince Hall PTR Upper Saddle River, 2005.

Noy, N.F., et al. 2006. A Framework for Ontology Evolution in Collaborative Enviroments. *Proceedings of the Semantic Web - ISWC.* s.l. : Springer-LNCS, 2006, Vol. 4273, pp. 544-55.

Ostrom, E. 1990. *Governing the Commons: The Evolution of Institutions for Collective Action.* New York : Cambridge University Press, 1990.

Pérez, Jorge, Arenas, Marcelo and Gutiérrez, Claudio. 2006. Semantics and Complexity of SPARQL. s.l. : International Semantic Web Conference , 2006, pp. 30-43.

Perry, D.E. and Kaiser, G.E. 1991. Models of softeware development enviroments. *IEEE Transactions On Software Enginering.* 1991, 17(3), pp. 283-295.

Porzel, R. and Malaka, R. 2004. A Task-based Approach for Ontology Evaluation. *Proceedings od ECAI04.* 2004.

Postel, J. and Reynolds, J. 1985. *File Transfer Protocol (FTP).* s.l. : IETF, Network Working Group, October 1985. RFC 959.

Sabou, M. 2006. *Building Web Service Ontologies - PhD Thesis.* Amsterdam : Vrije Universiteit, 2006.

Sanders, T.J.M., Spooren, W.P.M. and Noordman, L.G.M. 1993. Coherence relations in a cognitive theory of discourse representation. s.l. : Cognitive Linguistics, 1993, Vol. 4(2), pp. 93-133.

Schmidt's, Douglas C. Distributed Object Computing with CORBA Middleware. [Online] http://www.cs.wustl.edu/~schmidt/corba.html.

Schollmeier, Rüdiger. 2002. *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01).* s.l. : IEEE, 2002.

Sereno, B., Shum, S.B. and Motta, E. 2004. Clainspotter: An Enviroment to support Sesemaking with Knowledge Triples. *Proccedings of the Intelligent User Interfaces Conference, IUI205.* San Diego : s.n., 2004.

Shum, S.B., Motta, E. and Domingue, J. 2002. Augmenting Design Deliberation with Compendium: The Case of Collaborative Ontology Design. *Proceedings of the Workshop on Facilitating Hypertext Collaborative Modelling in conjunction with ACM Hypertext Conference.* Maryland : s.n., 2002.

Stocia, Ian, et al. 2001. *Chord. A Scalable Peer-to-peer Lookup Service for Internet Applications.* s.l. : Proceedings of the 2001 ACM IGCOMM Conference, 2001.

Stuckenschmidt, H., Sabou, M. and Klein, M. 2004. *Semantic Web Technology - Bringing Maning to Distributed Systems.* s.l. : IEEE Distributed Systems Online, 2004.

Tempich, C. 2006. *Ontology Engineering and Routing in Distributed Knowledge Management Aplications. PhD thesis.* s.l. : University of Karlsruhe, 2006.

ten Teije, A., van Harmelen, F. and Wielinga, B. 2004. Configuration of Web Services as Parametric Design. [ed.] E. Motta, et al. *Proceedings of the 14th International Conference on Knowledge Engineering and Knowledge Management, (EKAW-2004).* Whittleburry Hall : Springer-Verlag, 2004, Vol. number 3257 in LNAI, pp. 321-336.

The Open Group. 1997. Universal Unique Identifier Format. DCE 1.1:Remote Procedure Call. [En línea] 1997. http://www.opengroup.org/onlinepubs/9629399/apdxa.htm.

van der Aalst, W.M.P., et al. 2003. Workflow patterns. Distributed and Parallel Databases. 2003, 14, pp. 5-51.

van Eemeren, F.H. and Grootendorst, R. 2003. *A Systematic Theory of Argumentation: The Prahma-Dialectical Approach.* Cambridge : Cambridge University Press, 2003.

Vrandecic, D. 2005. Explicit knowledge engineering patterns with macros. [ed.] C. Welty and A. Gangemi. *Proceedings of the Ontology Patterns for the Semantic Web Workshop at the ISWC 2005.* Galway : s.n., 2005.

W3C. 2002. *Web services architecture requirements.* 2002. W3C Web Services Architecture Working Draft.

Wroe, C., et al. 2004. Automating Experimens Using Semantic Data on a Bioinfrmatics Grid. s.l. : IEEE Intelligent Systems, 2004, 19(1), pp. 48-55.