Integrated Project

Priority 2.4.7

Semantic based knowledge systems

The Social Semantic Desktop

NEPOMUK

First version Backbone and Connector Infrastructure

Deliverable D6.1

Version 1.0

08.11.2006

Dissemination level: PU

Nature	Other
Due date	30.09.2006
Lead contractor	NUIG
Start date of project	01.01.2006
Duration	36 months

Authors

Tudor Groza, NUIG
Leo Sauermann, DFKI
Paul – Alexandru Chirita, L3S

Mentors

Prof. Dr. Manfred Hauswirth, NUIG
Prof. Dr. Harald C. Gall, UNIVERSITÄT ZÜRICH, INSTITUT FÜR INFORMATIK, ZÜRICH

Contributions

Dr. Siegfried Handschuh, NUIG
Knud Möller, NUIG
Dr. Gerald Reif, UNIVERSITÄT ZÜRICH, INSTITUT FÜR INFORMATIK, ZÜRICH
Pat Croke, HPGL
Aine Leddy, HEWLETT PACKARD GALWAY LTD
Michael Sintek,DFKI
Gunnar Grimnes, DFKI
Raluca Paiu, FZI
Stefania Costache, L3S

Project Co-ordinator

Dr. Ansgar Bernardi
German Research Center for Artificial Intelligence (DFKI) GmbH
Erwin-Schroedinger-Strasse (Building 57)
D 67663 Kaiserslautern
Germany
Email: bernardi@dfki.uni-kl.de, phone: +49 631 205 3582, fax: +49 631 205 4910

Partners

DEUTSCHES FORSCHUNGSZENTRUM FUER KUENSTLICHE INTELLIGENZ GMBH (DFKI)
IBM IRELAND PRODUCT DISTRIBUTION LIMITED (IBM)
SAP AG (SAP)
HEWLETT PACKARD GALWAY LTD (HPGL)
THALES S.A. (TRT)
PRC GROUP - THE MANAGEMENT HOUSE S.A. (PRC)
EDGE-IT S.A.R.L (EDG)
COGNIUM SYSTEMS S.A. (COG)
NATIONAL UNIVERSITY OF IRELAND, GALWAY (NTUA)
ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE (EPFL)
FORSCHUNGSZENTRUM INFORMATIK AN DER UNIVERSITAET KARLSRUHE (FZI)
UNIVERSITAET HANNOVER (L3S)
INSTITUTE OF COMMUNICATION AND COMPUTER SYSTEMS (ICCS)
KUNGLIGA TEKNISKA HOEGSKOLAN (KTH)
UNIVERSITA DELLA SVIZZERA ITALIANA (USI)
IRION MANAGEMENT CONSULTING GMBH (IMC)

Versions

Version	Date	Reason
0.1	24.05.2006	First draft
0.2	31.7.2006	Some text by leo
0.21	03.08.2006	Re-arrangement – Tudor
0.3	15.08.2006	Added content – Tudor
0.3	16.08.2006	Added content – Leo & Tudor
0.3.1	11.09.2006	Added content – Raluca & Stefania
0.4	13.09.2006	Added content and re-arrangements – Paul
0.4.3	25.09.2006	Refinements: Leo, Paul, Raluca, Tudor
0.5	27.09.2006	Changed template and further refinements – all
0.5.1	01.10.2006	Added content, refined content – Tudor, Paul, Siggi, Leo
0.6	20.10.2006	Added content (revised the Backbone chapter) – Tudor
0.7	29.10.2006	Rewritten Implementation. Refinement and addition of transition and explanation text. Executive Summary, Conclusion – Siggi
0.8	2.11.2006	New Introduction/Motivation, Adding of design rational into Nepomuk architecture – Siggi Nepomuk backbone APIs in Appendix - Tudor
1.0	08.11.2006	Released as final after final review/further smaller corrections; layout streamlined by IMC

Explanations of abbreviations on front page

Nature
R: Report
P: Prototype
R/P: Report and Prototype
O: Other

Dissemination level
PU: Public
PP: Restricted to other FP6 participants
RE: Restricted to specified group
CO: Confidential, only for NEPOMUK partners

Executive Summary

The objective of this deliverable is to document a first starting point of the NEPOMUK architecture as well as the first prototype of the backbone and connector infrastructure. The pre-condition for the architecture is that it should lead to a basis for standardization efforts.

In order to achieve this, we applied the following methodical steps:

- We defined the terminology and investigated the technological background necessary for the envisioned Service-Oriented Architecture (SOA) of the NEPOMUK system.
- We reviewed the state-of-the art for Semantic Desktop implementations with respect of the underlying architectural principles.
- We identified the need for a NEPOMUK specific software engineering lifecycle, starting with a bottom-up approach.
- We designed the NEPOMUK backbone and connector infrastructure in a way that supports a later standardization of the architecture, by imposing a language and component neutral API and framework on the components and their communication. The design follows the principle of abstraction, but foresees dedicated efficient native implementation as well as federations for cross-platform scenarios.

The work resulted in the following:

- A *starting point* NEPOMUK architecture, which describes a first harmonized view on the early software services and components
- A backbone and connector infrastructure framework and API.
- Implementations of the backbone and connector infrastructure in the form of libraries. Firstly, a reference architecture using platform neutral web services infrastructure. Secondly, native implementations for specific platforms.

In conclusion, we have a first version of the connector and backbone infrastructure which enables and facilitates the integration of the various services using open semantic web standards.

Table of contents

1. Introduction

In traditional desktop architectures, applications are isolated islands of data - each application has its own data, unaware of related and relavant data in other applications. Individual vendors may decide to allow their applications to interoperate, so that e.g. the email client knows about the address book. However, today there is no consistent approach for allowing interoperation and a system-wide exchange of data between applications. In a similar way, the desktops of different users are also isolated islands - there is no standardized architecture for interoperation and data exchange between desktops. Users may exchange data by sending emails or upload it to a server, but so far there is no way of seamless communication from an application used by one person on their desktop to an application used by another person on another desktop.

The problem on the desktop is similar to that on the Web. On the Web we are faced with isolated data islands, and also at the desktop there is not yet a standardized approach for finding and interacting between applications (viz. "Web Services"). The **Social Semantic Desktop paradigm** adopts the ideas of the **Semantic Web paradigm** [BernersLee2001], which offers a solution for the web. Formal ontologies capture both a shared conceptualization of desktop data and personal mental models. RDF (Resource Description Format)[1] serves as a common data representation format. Web Services - applications on the web - can describe their capabilities and interfaces in a standardized way and thus become Semantic Web Services. **On the desktop, applications** (or rather: their interfaces) **will therefore modeled in a similar fashion**. Together, these technologies provide a means to build the semantic bridges necessary for data exchange and application integration. The Social Semantic Desktop will transform the conventional desktop into a seamless, networked working environment, by loosening the borders between individual applications and the physical workspace of different users.

The aim of the NEPOMUK project is to provide a standardized description of a Semantic Desktop architecture, independent of any particular operating system or programming language. Reference implementations will show the feasibility of the standard. This deliverable describes in detail the so-called NEPOMUK backbone and connector infrastructure (or "backbone", for short). The backbone is the central piece of the NEPOMUK architecture. Considering the complete architecture as a set of services, the backbone's main responsibility is the publishing, discovering and invoking of each of these services. To elaborate on this and introduce a central concept in the design of NEPOMUK, we use the Service-Oriented Architecture (SOA) paradigm (see Section 3.3.).

While one pillar of the design of NEPOMUK is a clear architectural vision – data integration on the basis of RDF and ontologies, and the integration of application functionality in the form of a Web Service-like Service-Oriented Architecture –, another pillar are the already existing software components coming into the project from the various project partners.

[1] http://www.w3.org/RDF

This deliverable has therefore two important aspects. The first is a precise definition of the theoretical approach and grounding in proven technologies, while the second aspect is the generation of a harmonized integration of existing components in a starting point architecture. Over the course of the project, these two aspects will move closer and closer and eventually merge.

The deliverable is structured as follows: We start by outlining the requirements and objectives for designing a Social Semantic Desktop architecture in Section 2 and then in Section 3 continue with a thorough survey of existing technologies that we deem relevant for the design of the backbone. Then, in Section 4, we present other work aimed at developing a Semantic Desktop or similar systems. Section 5 introduces the concrete NEPOMUK architecture and details the role of the major types of the components present in it. Following, in Section 6 we focus on the actual realization of the NEPOMUK backbone and connector infrastructure by detailing a possible communication scenario on the desktop, describing the layered organization of the backbone, the inter-component communication and the NEPOMUK Registry. Furthermore, in Section 7, the deliverable shows how the prototypical implementation was developed and ends by stating our conclusions and future directions in Section 8.

2. Requirements and Objectives

In the following we shortly sketch the requirements and objectives of the Social Semantic Desktop to motivate the architecture and the backbone presented in this deliverable.

The **starting point architecture** of NEPOMUK (presented in Section 5) is the result of a reengineering process follows **the** design rationale and motivation which we aim to present here. The general idea of the Social Semantic Desktop is a wide use of Semantic Web technologies on personal computers. The use of ontologies, metadata annotations, and semantic web protocols on the desktop will allow the integration of desktop applications and the web, enabling a much more focused and integrated personal information management as well as focused information distribution and collaboration on the Web beyond sending emails. The goal is an open personal information management system and collaborative infrastructure based on Semantic Web technologies, built into current operating systems.

The Components of the Social Semantic Desktop can be classified in three areas: i) Personal Information Management, ii) Distributed Information Management, iii) Social Networks and Community Services.

The focal point of the initial architecture of NEPOMUK is Personal Information Management. However, some distributed and social aspects are integrated already now. In detail we see the following basic requirements for the semantic personal information management:

- **Knowledge Articulation and Visualization**. A means for articulation and visualization of structured information is needed. This is crucial both for presenting semantic data to the user, as well as for providing an editing environment for such data.
- **Standard Desktop Classification Structures**. The system has to provide a set of Standard vocabularies and ontologies for personal information management, which allows the user to structure and classify his everyday information. Examples are calendar data or task management. These ontologies are not static and form the basis for extension.
- **Mapping and Aligning of Information Schemes**. Information from similar domains might be expressed by different schemes (i.e. ontologies). While this is already true on the single desktop this problems gets bigger in a distributed environment. Hence, a Semantic Desktop needs means to align and map ontologies.
- **Wrapping of Legacy Information**. Current desktops contain a lot of both structured and unstructured data, which needs to be transformed into a standardised semantic representation (e.g. RDF/S). For structured data (e.g. existing file system metadata, email metadata), the transformation process will mean a mapping from one structured format to another. For unstructured data (mainly textual data such as emails or PDF documents), transformation will mean the application of Information Extraction and Language Processing technologies. This will enable interoperable applications using this information.
- **Metadata Storage and Querying**. The desktop information and the associated metadata and ontologies needs to be stored in a central place and to be made queryable.

- **Linking of Data Items and Relational Metadata**. Related information might be spread on the desktop. Hence, there must be support to link arbitrary information on the local desktop, across different media types, file formats, and applications. Semantic web data structures and techniques will be applied and adapted to achieve this goal.
- **Social Aspects**. Means for social relation building and knowledge exchange which support knowledge sharing within social communities. These means will provide semantically rich recommendations, which allow members of a community to not only exchange documents and other isolated information chunks, but all relevant information about their context and the participating community as well.
- **Open Architecture**. The Social Semantic Desktop has an open framework architecture with clearly defined interfaces which are published and possibly submitted for standardization. This will allow external adopters to integrate their proprietary desktop tools into the framework and offers ways for commercial support and extension activities. We aim to reach early dissemination of project results and to interact with the open-source developer community. This will allow for the gathering and inclusion of feedback and development contributions form interested third parties.

Section 5 shows how these requirements can be mapped to the various components of the NEPOMUK starting point architecture.

To tackle the requirement of an open architecture we aim for a Services Oriented Architecture (SOA). We further aim to ensure language and platform independence, and thus the publication of the implemented middleware as open-source software. Also, in the relevant cases, we intend to submit our results into a group review process of the NEPOMUK consortium.

The following Section 3 gives an overview of the underlying technology of a Service Oriented Architecture.

3. Technology Overview

This section introduces some of the technologies that will be used across this deliverable. We start by describing shortly the concepts behind the NEPOMUK backbone and connector infrastructure, and then we present a brief overview of the web services (WS) technologies, including the service oriented architecture (SOA) paradigm, and a particular Java alternative, i.e. the OSGi service platform. Finally, we discuss other technologies that we investigated for the inter-process communication across the NEPOMUK architecture.

3.1. Terminology

The NEPOMUK **architecture** represents a set of standardized, neutral and platform-independent services provided by the NEPOMUK framework, without including particular descriptions for a specific service. Specific services are realized through conformance to the NEPOMUK standard. For example: PIMO service – a Personal Information Model Ontology Service or Context Manager Service (see Section 5 for more details).

The **backbone and connector infrastructure**, supporting the NEPOMUK architecture, enables and facilitates the integration of different services providing the means for communication and interaction, and establishing a standard communication protocol. It takes care of the following processes:

- service publishing (and therefore registry)
- service discovery and match-making
- service invocation

While writing this deliverable we noticed that "**middleware**[2]" is also an appropriate term for that what we call "backbone and connector" infrastructure, since middleware denotes software that connects software components or applications. However, since all documentation and communication in the project so far is referring to "the backbone" we decided to stick to this name for the moment being.

3.2. Web Services

A *Web Service* represents a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered by XML artifacts and furthermore the Web Service supports direct interactions with other software applications using XML based messages via an Internet-based protocol.

[2] http://en.wikipedia.org/wiki/Middleware

3.3. The Service Oriented Architecture (SOA)

In Section 2 we argued that a Service Oriented Architecture is the appropriate mean to ensure the requirement of an **open architecture** for NEPOMUK. In the following we will introduce what **SOA** is and present a thorough survey of existing SOA related technologies such as WSDL, SOAP, UDDI, REST, OSGi. The investigation on these technologies has been important for us to fully understand the possibilities and to decide on a proper design for the backbone (see Section 6).

Overview. *Service-Oriented Architecture (SOA) expresses a perspective of software architecture that defines the use of services to support the requirements of software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation.* [3]

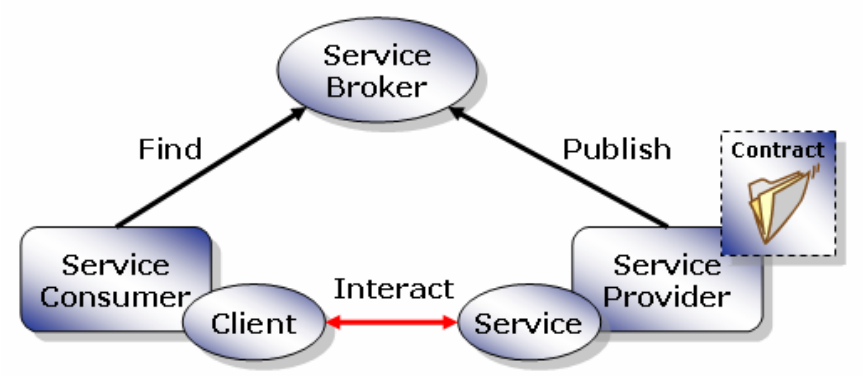

Figure 1. The Service Oriented Architecture

SOA represents a style of architecture, usually based on Web Services standards (e.g. SOAP – Simple Object Access Protocol or REST – Representational State Transfer) enabling the design of applications that combine loosely-coupled and interoperable services. A service is a unit of work published by a service provider, which is meant to produce results for a service consumer. The interoperation between the provider and consumer is based on a formal definition (or contract, e.g. WSDL – Web Services Description Language) independent of the programming language and underlying platform. Thus, the contract hides the implementation details, providing a method for different services implemented in different programming languages to be consumed by a common composite application.

The Service Broker is an optional component, depending on the architecture definition. Usually it is represented by a registry (e.g. UDDI –

[3] Wikipedia Contributors, ,“Service Oriented Architecture“, Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Service-oriented_architecture (accessed August 21 2006)

Universal Description, Discovery and Integration) having a known interface with the role of providing the means for the publishing and the discovery of services. In other words, the registry is a container for published contracts which can be inquired by potential service consumers.

3.3.1. SOA and Web Service protocols

3.3.1.1 WSDL, SOAP, UDDI family

Web Services Description Language (WSDL). WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate. [Christensen2001]

Simple Object Access Protocol (SOAP). *SOAP* is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.

Two major design goals for SOAP are simplicity and extensibility. SOAP attempts to meet these goals by omitting, from the messaging framework, features that are often found in distributed systems. Such features include but are not limited to: reliability, security, correlation, routing and Message Exchange Patterns (MEPs). [Gudgin2003]

Universal Description, Discovery and Integration (UDDI). The UDDI protocol is a central element of the group of related standards which comprises the Web Services stack. The UDDI specification defines a standard method for publishing and discovering the network-based software components of a service-oriented architecture (SOA). Its development is led by the OASIS consortium of enterprise software vendors and customers. [Oasis2004]

The functional purpose of a UDDI registry is the representation of data about services. It was designed to offer several benefits as part of a vision in which service-based applications would be linked through a public or private dynamic brokerage system. These benefits would increase the code re-use and improve the infrastructure management by:

- publishing information about Web Services
- inquiring about Web Services based on a certain criterion
- determining the security and transport protocols supported by a given Web Service

The directory usually contains the contracts published by the services' providers as WSDL descriptions. These define the protocol bindings and the required message formats in order to interact with that service provider. The inquiry is realized by an exchange of SOAP messages.

3.3.1.2 REST – Representational State Transfer

Representational State Transfer (REST) is a software architecture style for distributed hypermedia systems. It was first described in the doctoral thesis of Roy Fielding [Fielding2000], and has quickly passed into widespread use in the networking community.

The core idea is to use the existing parts of HTTP to describe services rather than to use a more detailed framework on top, like SOAP. The application state and the functionality are divided into *resources*, each resource being identified by its URI. The existing HTTP methods POST, GET, PUT, and DELETE can then be used to access or manipulate the resource.

Using REST, setting a list of participants of an event may result in an operation such as:

- Use the HTTP PUT method ...
- ... on the URI http://example.com/event/200/participants.
- PUT the list of participants using a defined XML syntax.

In contrast to SOAP and WSDL, REST does not enforce a formal description of the services.

3.3.1.3 The OSGi Service Platform

OSGi (Open Services Gateway initiative) technology is a dynamic module system for Java. The OSGi Service Platform provides functionality to Java that makes Java a good environment for software integration and thus for development. Java provides the portability that is required to support products on many different platforms. The OSGi technology provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. These components can be composed into an application and deployed.

The OSGi Service Platform provides the functions needed to dynamically change the service composition on a variety of devices and networks, without requiring restarts. To minimize, as well as to manage coupling, the OSGi technology provides a service-oriented architecture that enables these components to dynamically discover each other for collaboration. The OSGi Alliance has developed many standard component interfaces for common functions such as: HTTP service configuration, logging, security or user administration. Pluggable implementations of these components can be obtained from different vendors, which feature different optimizations and costs. However, service interfaces can also be developed on a proprietary basis. (See also [OSGI2006])

3.3.2. Analysis of platforms for Web Services

3.3.2.1 Web Services and OSGi

OSGI provides a platform that is similar to the intended backbone, but does not match the service-oriented paradigm exactly. The following table shows the main differences and similarities.

	Web Services	OSGi
Technology	Language/technology independent	Java world only
Components	A single component is an individual application (individual process)	A single component is a .jar or .zip file containing OSGi-specific meta-information
Services	A single service is an HTTP application exposing some functionalities through a protocol respecting a service contract defined in WSDL	A single service is a Java object implementing a Java interface defining the contract of this service
Communication between components	Defines how applications (independent processes) can communicate with each other. Does not specify the internal structure of individual applications	Does not specify inter-process communication. Defines communication between components inside one process.
Life-cycle of components	Does not control the life-cycle of applications (SOA components)	Defines and controls the whole life-cycle of bundles (OSGi components) - how they can be installed, activated, deactivated and removed
Consuming / providing services	One SOA component (application) can expose and use many different web services	One OSGi-bundle (OSGi-component) can expose/use many different java objects which are registered as services
Service discovery	UDDI defines how a service can be found by its WSDL descriptor	OSGi defines how components (services) can be registered/discovered based on java interfaces and version numbers

3.3.2.2 Web Services and REST

For the envisioned HTTP based service architecture, two architectural approaches are commonly used in industrial projects today, REST and SOAP. In order to decide for one of them, we need to evaluate which approach will provide a better foundation for the NEPOMUK standard. Please note that a real comparison between REST and SOAP would not be valid, firstly because REST is not a standard, and secondly, because REST is an architectural style, while SOAP is a protocol. REST does not define a formal protocol, but provides guidelines on how to build web services in a simple, clear and clever way. On the other hand, the triumvirate of SOAP, WSDL, and UDDI defines standards for communication between services and service lookup, but the programmers are completely free in defining their services.

For NEPOMUK, the main requirement is an approach that lets us **define a standard**. WSDL is a formal language that allows describing such a standard. In theory, WSDL 2.0 can be used to describe REST architectures, but this is not common practice and we could not verify tool support for this approach, whereas using WSDL in combination with SOAP allows automatic generation of client stubs and server skeletons for various programming environments. Therefore, the standardization requirement of NEPOMUK can be best met by using WSDL and SOAP in combination, because this standard can be easily adopted by companies and professional software developers.

This leads to the second requirement, namely **tool support**, which is closely coupled to the first requirement. For SOAP, various implementations exist and are used in industrial projects, for example to expose company services for business-to-business communication. For REST, also many implementations exist, especially in the Web 2.0 context. For REST services, clients to access these services are hand-written based on the documentation provided by the service provider or by the reverse engineering of the protocol. For popular services like Amazon[4] or Flickr[5], this work was done by various open source projects. So we see that time can be saved using SOAP and WSDL because clients stubs and skeletons can be auto-generated. Once such REST clients for various programming languages exist, the difference between SOAP and REST is negligible.

A third requirement to the standard would be to **fit the NEPOMUK Architecture context**. Some of the NEPOMUK services are usual RPC method invocations, like the task management service (see Section 5). Others, like the semantic wiki, are resource centered approaches, where a REST architecture may help. Both are realizable using SOAP or REST technologies.

As a result of this discussion, NEPOMUK will recommend the use of SOAP and WSDL in their stable versions for developing and documenting the majority of the NEPOMUK services. In parallel, for the RDF database interfaces, we intend to analyze what would be the advantages of using REST as communication mean.

3.4. Inter-Process Communication Infrastructures

Inter-Process Communication (IPC) happens when different applications communicate on one computer. An application ends at its memory space. For example a Java application calling a C-written DLL (Dynamic Linked Library) is still an in-process invocation. Another application contacting a database server via a pipe uses IPC for this. There are many examples of standards that describe IPC protocols: DBUS, DCOP, KParts, Bonobo, ActiveX, COM, D-COM, ActiveX, CORBA, RMI, plain pipes, shared memory, files or HTTP. On Microsoft Platforms, COM is the quasi-standard of IPC and found wide use in systems that are similar to

[4] www.amazon.com

[5] www.flickr.com

NEPOMUK. On Linux platforms, DBUS, KParts, and DCOP are popular counterparts.

Application of Inter-Process Communication can range from simple method calls (for example: open the system's web-browser) to more complex invocations (create a new contact in the address book, set the given name and family name).

In NEPOMUK, IPC happens between separate applications and services running on a single desktop. For example, any graphical user interface will use IPC to contact the NEPOMUK services.

Following, we will describe two of the most common approaches in more detail, together with their relation with the NEPOMUK Architecture:

- **DBus**, representing one of the most commonly used communication protocols on Linux platforms. The KDE community is supporting the efforts of the NEPOMUK consortium and will develop a KDE based prototype following the NEPOMUK standards. One direct consequence will be the integration between DBus and the NEPOMUK Backbone.
- **COM**, the “de facto” communication protocol standard on Windows platforms. We describe COM because, we envision that the future NEPOMUK standard will provide an unified support to semantically interconnect applications, and thus could be considered as a viable alternative even to the currently existing proprietary “de facto” standards.

DBus. DBus is a protocol for desktop inter-process communication. It is standardized under the umbrella of the *freedesktop.org* project. DBus allows other programs to register themselves as services for others. Clients can lookup existing services and send messages to these services. DBus is implemented as a daemon process, opening a separate communication channel for each participating user. Thus it provides a communication mean between applications of one user, also on multi-user platforms. It aims to replace DCOP [Brown2003] and Bonobo [Gnome] for simple inter-process communication. Finally, although DBus is platform and programming language independent, it is primarily useful for the Linux platform.

COM. The Microsoft Standard COM (Component Object Model) is a standard for inter-process-communication on the Microsoft Windows operating system. COM allows an efficient way for object-oriented communication across applications.

Applications can register as COM servers, publishing themselves to the Windows registry using the regsrv32 application. The registry will list all components by name, using a globally unique identifier (GUID) to avoid problems between incompatible versions of the same service. Clients can contact a COM server by calling operating system methods. The communication is object-oriented, clients receive a handle on an object and can call functions of it. The operating system will detect if the requested service is running, if not, it will be started.

Functions can return complex objects and data types. Interfaces to COM objects are described in “Type Libraries” (TBL files). Using these libraries, clients can automatically generate code and interfaces to interact with COM servers. All major programming languages include either such COM clients or other commercially available clients.

There exist two extensions for COM:

- D-COM: allowing the invocation of methods in a distributed scenario
- ActiveX: more convenient access to COM objects and the possibility to include graphical components.

The bus-system implemented by COM contains listener/sender daemons running in every COM-enabled application. COM is not a bus system, but more a client-server method of connection, which is mediated by the operating system.

COM is a major standard on the Windows platform, a multitude of vendors sell their software components as COM services.

4. Related Research Activities

This Section will review some of the most important research, open source and industrial projects targeting similar functionalities as NEPOMUK. We will start with the research oriented ones (i.e., Haystack, SEMEX, and IRIS), as they are the most ambitious in terms of innovation, then continue with the open source ones (i.e., Chandler, DeepaMehta and Apogee), as they represent community efforts towards creating a semantic desktop, and finally overview several other projects, including industrial ones such as PHLAT.

4.1. Research Oriented Systems

Gnowsis. Gnowsis[6] [Sauermann2006] is a semantic desktop with a strong focus on extensibility and inte- gration. The goal of gnowsis is to enhance existing desktop applications and the desktop operating system with Semantic Web features. The primary use for such a system is Personal Information Management (PIM), technically realized by representing the user's data in RDF. The gnowsis pro ject was created 2003 in Leo Sauermann's diploma thesis and continued in the DFKI research project EPOS[7].

The Gnowsis architecture can be split into two parts, the gnowsis-server which does all the data processing, storage and interaction with native applications; and a graphical user interface (GUI) part. The interface between the server and GUI is clearly specified, making it easy to develop alternative interfaces.

Gnowsis uses a Service Oriented Architecture (SOA), where each component defines a certain interface, after the server started the component the interface is available as XML/RPC service, to be used by other applications.

An important design goal of Gnowsis is to complement rather than replace existing desktop applicaiton and data from external applications like Microsoft Outlook or Mozilla Thunderbird are integrated via the extraction framework Aperture[8].

Haystack. The Haystack Project[9] [Karger2005] is investigating approaches for allowing people to manage their information in ways that make the most sense to them. By removing arbitrary application-created barriers, which handle only certain information “types” and relationships as defined by the developer, Haystack aims to allowing users to define their most effective arrangements and connections between views of information. Such personalization of information management is meant to improve everyone’s ability to find information located in the personal

[6] http://www.gnowsis.org/

[7] http://www3.dfki.uni-kl.de/epos

[8] http://aperture.sourceforge.net/

[9] http://haystack.csail.mit.edu/

space. The Haystack architecture can be described in two distinct parts, a Haystack Data Model (HDM) and a Haystack Service Model (HSM). The Data Model is the means by which the user's information space is represented and the services append to or process the data in some fashion. The abstract representation of the Haystack Data Model (HDM) is that of a directed graph. The vertices and edges are typed and the typing information provides semantic information about the structure. With the HDM it is then possible to represent both the metadata associations between objects (i.e. the URL of a document, the author of a thesis, the date the picture was deleted), as well as the associations between documents (i.e. all the documents that a paper is citing). The set of functionalities within Haystack is implemented by objects in the Haystack Service Model (HSM). Abstractly, Haystack can be viewed as standard three-tiered architecture consisting of three different layers, a user interface layer (the client), a server/service layer, and a database.

SEMEX. Another relevant personal information management tool is the Semex System (SEMantic EXplorer) [Dong2005], which organizes the data in a semantically meaningful way by providing a domain model consisting of classes and associations between the classes. Besides, Semex leverages the PIM environment to support on-the-fly integration of personal and public data. Users interact with Semex through a domain ontology that offers a set of meaningful domain objects and relationships between these objects. Information sources are related to the ontology through a set of mappings, share domain models with other users and import fragments of public domain models in order to increase the coverage of their information space. When users are faced with an information integration task, Semex aids them by trying to leverage from previous tasks performed by the user or by others with similar goals. Hence, the effort expended by one user later benefits others. Semex begins by extracting data from multiple sources and for these extractions it creates instances of classes in the domain model. Semex employs multiple modules for extracting associations, as well as allowing associations to be given by external sources or to be defined as views over other sets of associations. To combine all these associations seamlessly, SEMEX automatically reconciles multiple references to the same real-world object. The user browses and queries all this information through the domain model (Figure 2).

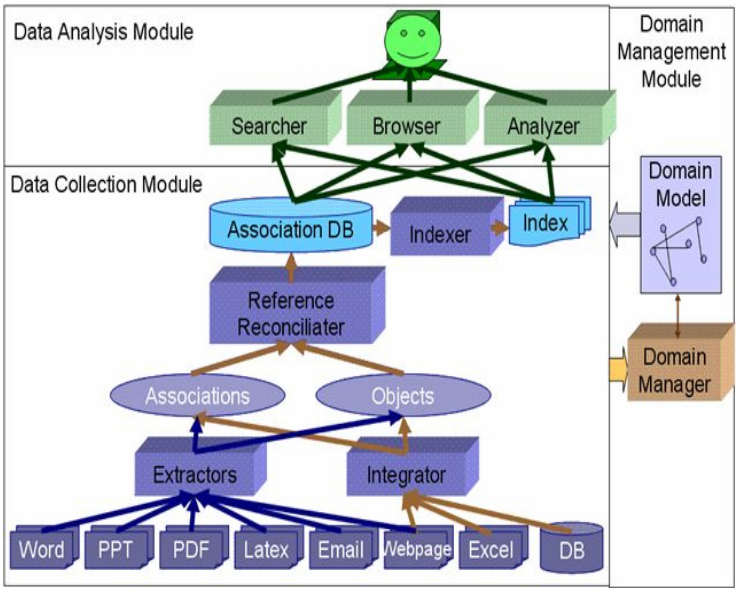

Figure 2. SEMEX Architecture [Dong2005]

IRIS. A similar idea is exploited by the IRIS Semantic Desktop [Cheyer2005], an application framework for enabling users to create a "personal map" across their office-related information objects. IRIS is an acronym for "Integrate. Relate. Infer. Share" and the framework offers integration services at three levels (Figure 3):

1. Information resources (e.g., email messages, calendar appointments) and applications that create and manipulate them, must be accessible to IRIS for instrumentation, automation and query. IRIS offers a plug-in framework, in the style of the Eclipse architecture, where "applications" and "services" can be defined and integrated within IRIS. Apart from a very small, lightweight kernel, all functionality within IRIS is defined using a plug-in framework, including user interface, applications, back end persistence store, learning modules, harvesters, etc.

2. A Knowledge Base (KB) provides the unified data model, persistence store, and query mechanisms across the information resources and semantic relations among them.

3. The IRIS user interface framework allows plug-in applications to embed their own interfaces within IRIS, and to interoperate with global UI services, such as notification pane, menu toolbar management, query interfaces, the link manager and suggestion pane.

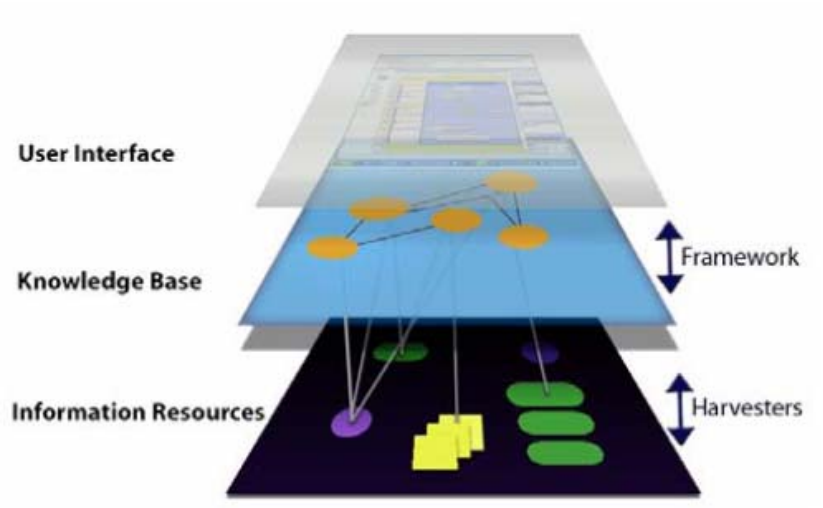

Figure 3. The three-layered IRIS integration framework [Cheyer2005]

4.2. Open Source Community Software

Chandler. An interpersonal information manager, adapting to the changing user needs, is the Chandler system[10]. Chandler delivers an integrated system for individuals and small workgroups and offers capabilities in knowledge sharing to support workgroup collaboration. Chandler sharing is server-based, works across platforms and supports multi-author editing. Chandler's architecture consists of several layers, as depicted in Figure 4.

[10] http://chandler.osafoundation.org/

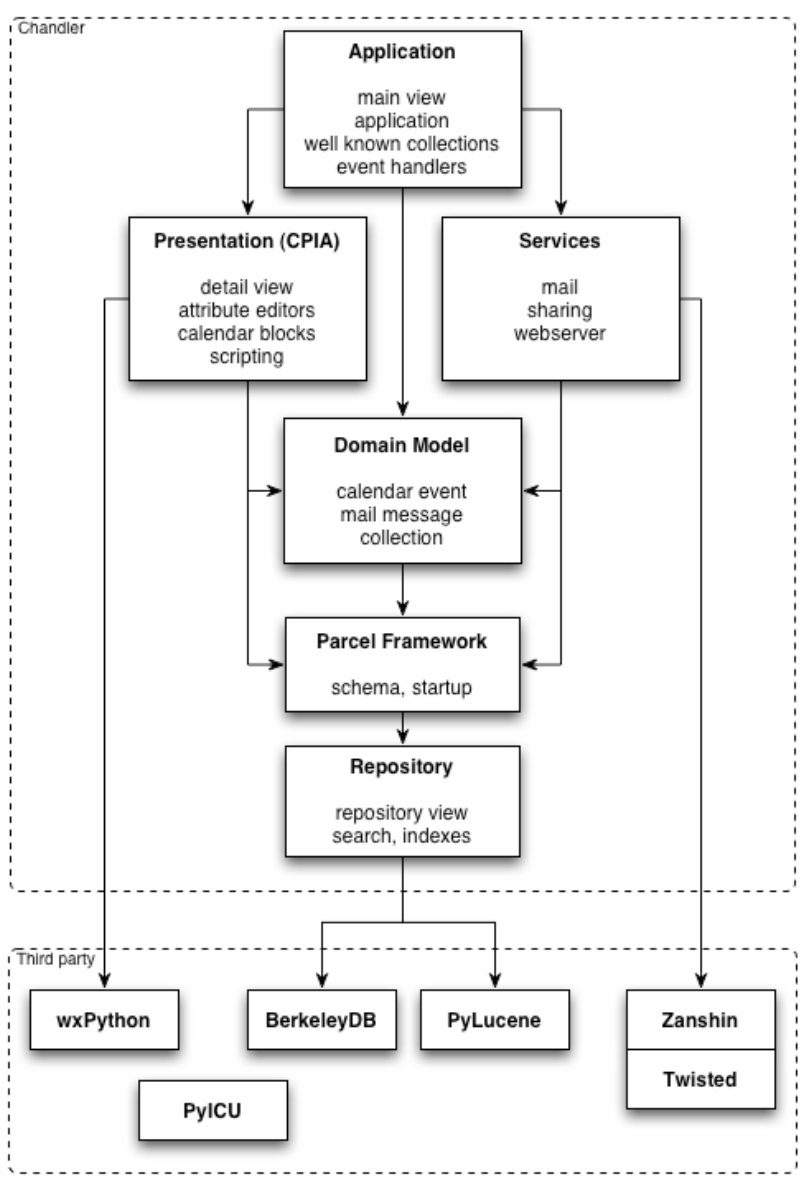

Figure 4. Chandler's Architecture

Generally, each layer or component is able to directly access the APIs of the layers below it. The lower level layers communicate with the layers above via system of notifications. The lower level layers have no specific knowledge of the layers above, and the higher level layers know only the APIs of the layers below. At the top of the diagram, the application layer is responsible for pulling all of the pieces together to present the interface to the user and for starting up the system. Chandler's Presentation layer is handled by the Chandler Presentation and Interaction Architecture (CPIA). Its role is to provide building blocks for Chandler's user interface, including some generic building blocks (e.g., Menus, Status Bar) as well as more specific building blocks (e.g., Sidebar, Calendar View, Detail View). The services layer allows communication with the outside world. Currently this includes sharing (via WebDAV and CalDAV), email (IMAP, POP and SMTP), and running Chandler as a local web server. Chandler allows users to share Collections by publishing them to a server and similarly, a Chandler user can subscribe to other users' Collections. The domain model defines all of the domain specific classes that represent application content such as Calendar Events, Mail

Messages, Tasks, etc. The Repository is the persistent store for Chandler's data driven architecture. It implements the core code for Items and sets of Items (the basis of Collections), as well as notifications. It also supports full text search, sorting and indexing of sets of Items.

DeepaMehta. DeepaMehta [Richter2005] is an open source semantic desktop application based on the Topic Maps standard[11]. It aims at evolving nowadays separated desktop applications into an integrated workspace, enabling the user to organize, describe and relate information objects like text notes, external documents and media, browse the web, search databases and create semantic networks – all these in one seamless, semantic-enabled desktop environment. DeepaMehta is a service oriented application framework with a data model based on topic maps and a UI that renders them as a graph, similar to concept maps. Information of any kind as well as relations between information items can be displayed and edited in the same space. The user is no longer confronted with files and programs. Topic Maps are individual views on interconnected content and they may evolve on their own, as the users continue to work with the system. DeepaMehta has a layered, service oriented architecture, the main layer being the application layer (Figure 5). It offers various ways for the presentation layer to communicate with it via the communication layer (API, XTM export, EJB, SOAP). The built-in server offers an out-of-the-box user interface which runs in almost any browser. The storage layer manages the corporate memory, which holds all topics and their data either in a relational database or simply in the file system.

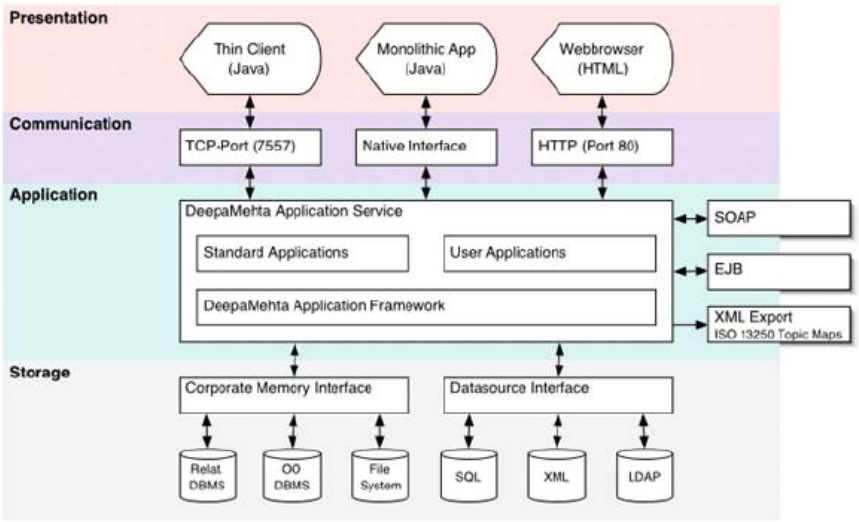

Figure 5. DeepaMehta Architecture [Richter2005]

Apogee. So far, we have looked at projects supporting mainly every-day desktop users. Apogee[12] is a project, which targets the Enterprise

[11] ISO/IEC 13250:2000 "Topic Maps" standard, http://www.topicmaps.org/xtm/1.0/#ref_iso13250

[12] http://apogee.nuxeo.org/

Development Process (ECM) market. More and more customers need specific applications related to ECM to properly handle their data and integrate in a seamless way all their digital assets and involved processes. These applications share a lot of features and need many common services. ECM application developers need a framework that would ease the creation of this kind of desktop application. Apogee aims at building a framework to create ECM-oriented desktop applications, independent from vendor or technologies. This framework could be used to create applications that will be integrated with any ECM platform. In the first phase of the project two ECM providers will be implemented: a local provider and a remote one based on the Nuxeo's CPS ECM platform (Figure 6).

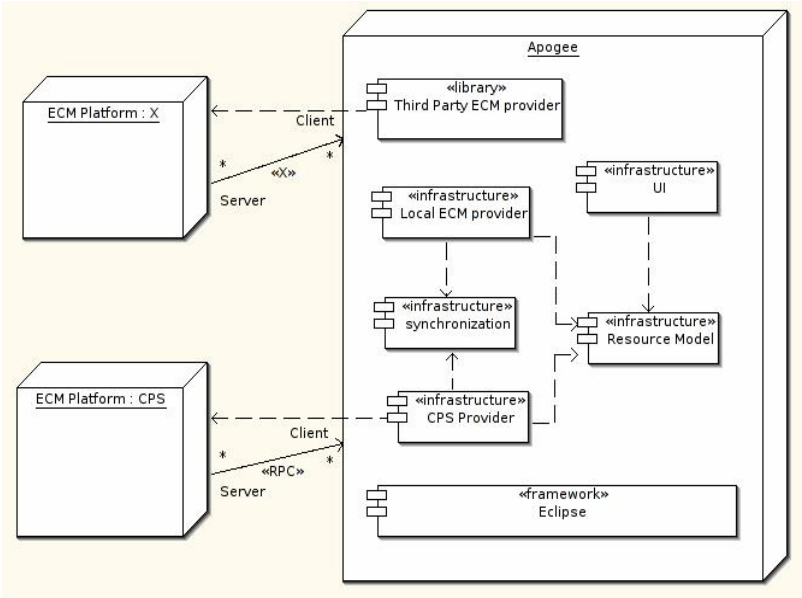

Figure 6. Apogee Deployment

Apogee objects are living inside a tree structure in the same manner as files and folders in a file system but the similarity ends here. Apogee resources are neither folders, nor files. They may point to physical objects (stored somewhere) or they may represents only logical nodes in the resource tree. Examples of possible objects are files, folders, user groups, users or any other objects that may be represented in a tree structure. Apogee runs as a plug-in inside the eclipse IDE.

4.3. Other Systems

Other Systems. Other relevant initiatives include (1) **DBIN** [Tummarello2005], which is similar to a file sharing client and connects directly to other peers, (2) **PHLAT** [Cutrell2006], a new interface for Windows, enabling users to easily specify queries and filters, attempting to integrate search and browse in one intuitive interface, or (3) **MindRaider**[13], a

[13] http://mindraider.sourceforge.net/

Semantic Web outliner, trying to connect the tradition of outline editors with emerging technologies. MindRaider aims to organize not only the content of your hard drive but also your cognitive base and social relationships in a way that enables quick navigation, concise representation and inferencing. Finally, starting from the idea that everything has to do with everything else, **Fenfire**[14] is a Free Software project developing a computing environment in which you can express such relationships and benefit from them.

[14] http://fenfire.org/

4.4. Conclusions

So far we have looked at a number of tools, today isolated and focusing on complementary fields. NEPOMUK will considerably advance the state-of-the-art and technology by creating a completely novel knowledge-worker support system, thus bringing together these today isolated and complementary aspects. This new technical and methodological platform, The Social Semantic Desktop, enables users to build, maintain and employ inter-workspace relations in large scale distributed scenarios. New knowledge can be articulated in semantic structures and can be connected with existing information items on the local and remote desktops, while knowledge information items and their metadata can be shared spontaneously without a central infrastructure. Moreover, NEPOMUK realizes a freely available open-source integration framework with a set of standardized interfaces, ontologies and applications. Also, the NEPOMUK's standardized plug-in architecture combined with usage experiences opens up manifold business opportunities for new generic or domain-specific products and services.

Although the systems we have looked at focus on isolated and complementary aspects, some of their architectural models were worth investigating. The NEPOMUK Architecture is also based on layers, same as Haystack, IRIS and DeepaMetha, having a User Interface Layer, a Service and a Data Storage Layer. The modular architecture, also identified within the Haystack, SEMEX and DeepaMetha systems, as well as the standardized APIs offer an easy way of plugging-in new components. The different components in NEPOMUK do not interact directly with each other, like in SEMEX, IRIS or DeepaMetha, but rather communicate through the Data Services and the Web Server Layers. In Chandler, however, the components can only communicate with the components in the layers below. Our approach guarantees that each component may be changed without affecting other components it interacts with. The interaction has to suffer only in the case in which the API of the component is modified. Similar to Chandler, the NEPOMUK Architecture also provides service discovery functionalities: the NEPOMUK Registry providing a proper support for publishing and discovering the existing NEPOMUK Services by using a standard interface.

5. NEPOMUK Architecture

The text book approach for **software design** is to start with the system requirements, then to identify the necessary functionalities and further to design the system, viz. to apply a **top-down approach** to the architecture. This classical waterfall model is argued by many to be a bad idea in practice, mainly because of their belief that it is impossible to get one phase of a software lifecycle "perfected" before moving on to the next phases and learning from them. In a complex project like NEPOMUK with many partners it is even harder to apply the top-down approach as such, since some partner already came with preliminary prototypes into the project, which has been developed further from the first day on of the project. Each of these prototypes is codifying the share and vision of an individual partner with respect to a semantic desktop. In conclusion, the first step towards a common architecture was **a reengineering** step of all existing -- and to some extend also envisioned -- software, viz. a bottom-up approach in order to gain a common and harmonized view of the architecture. As shown in the design rational Section 5.1.1 the reengineered components link to the requirements for a Semantic Desktop (Section 2). However, this architecture is to seen as first **starting point** and by no means as the final NEPOMUK architecture.

The following will describe this **harmonized high level view** on the starting point NEPOMUK architecture. As part of the architecture, the NEPOMUK components implement services for the general architecture and they communicate using web services and the NEPOMUK backbone. Please note that the component/service names are adopted from the existing software and will be refined and renamed accordingly in a later architecture phase.

5.1. Architecture and Components

This section provides an introduction to the current NEPOMUK Component Architecture, as well as a short description of its main parts. The current status does not cover the whole envisioned architecture, but only the part developed until this moment. Thus, for example, the components dealing with the collaboration between several Semantic Desktops are not yet present.

The Component Architecture consists of all the components and interfaces used in NEPOMUK. The diagram from Figure 7 depicts how these pieces are interconnected in practice. Each component has a description of its functionality, the interfaces it supports and the interfaces it requires from other components. WSDL was used as description format. The Backbone and connector Infrastructure constitutes the central part that connects all the other components and it ensures proper communication among all of them. In the remainder of this section we will provide short descriptions of each component present in the general architecture.

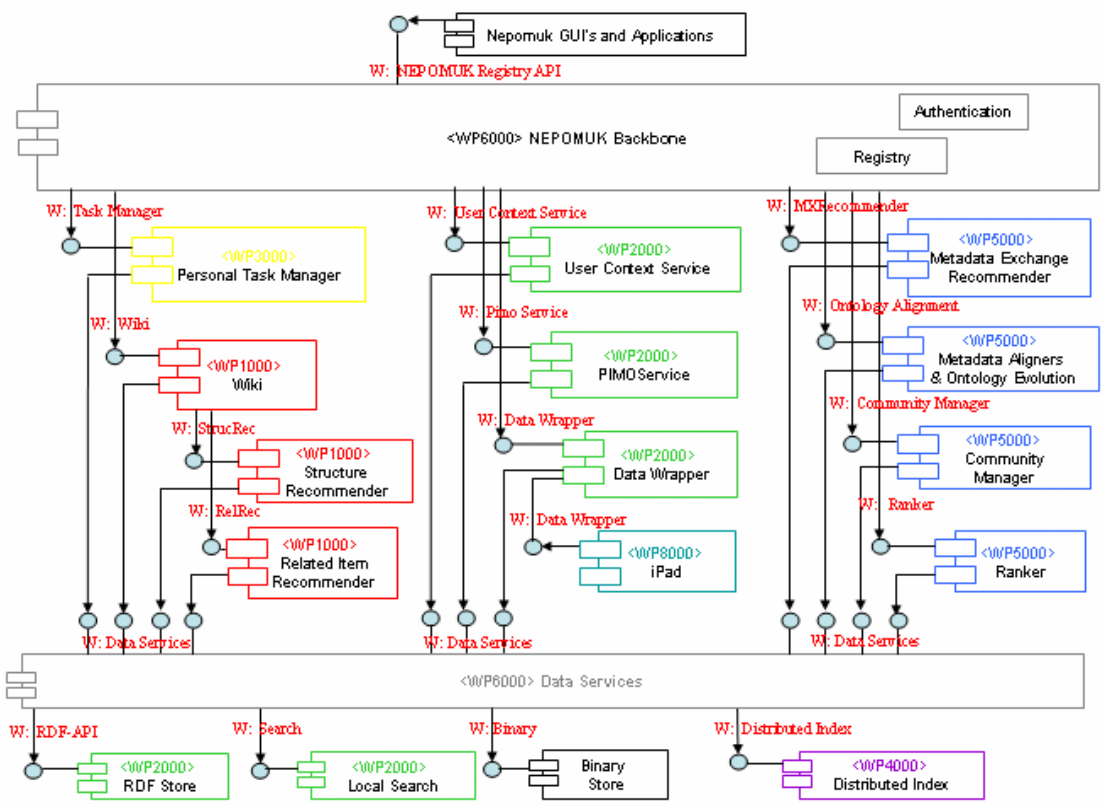

Figure 7. Current NEPOMUK Component Architecture Diagram

5.1.1. Design Rationale

The requirements and considerations from Section 2 feed into the design rational of the starting point NEPOMUK architecture. The design rational links the requirements to the NEPOMUK modules. This results in a N:M mapping (neither functional nor injective). An overview of the matrix is given in Table 1.

5.1.2. Components and Services

In the following we present a detailed description of each of the components[15].

Several Wiki types of components have been identified during the Bottom-Up analysis phase, among which TripleWiki, KaukoluWiki, RichWiki, SemWiki and ImapWiki, all resumed into the abstract ***Wiki*** component. The TripleWiki is a simple semantic Wiki with direct statements entries. The KaukoluWiki is a Wiki which allows semi-automatic metadata generation. The RichWiki is an Eclipse-based Wiki, which provides support for rich interactive clients. The result from the SemWiki will be a hypertext-based prototype for personal knowledge

[15] Please note that the naming of the components is originated by the reengineered prototypes and might be changed later to reflect a more proper design.

management. The ImapWiki is a Visual Knowledge Workbench with Wiki-abilities. It is a prototype for visual as well as textual editing of knowledge structures.

Requirement/ Component	Knowledge Articulation and Visualization	Standard Desktop Classification Structures	Mapping and Aligning	Wrapping of Legacy Information	Metadata Storage and Querying	Linking of Data Items and Relational Metadata	Social Aspects	Open Architecture
Wiki	X					X		
Structure Recommender				X				
Related Item Recommender				X		X		
User Context Service				X		X		
PIMO Service		X						
Data Wrapper				X				
Personal Task Manager		X						
RDF Store (Data Service)					X			
Local Search (Data Service)					X			
Distributed Index (Data Service)					X			
Metadata Exchange Recommender							X	
Ontology & Metadata Aligners			X				X	
Community Manager							X	
Ranker					X			
Backbone								X

Table 1. Design Rational - Linking Requirements with NEPOMUK components

5.1.3. Components and Services

In the following we present a detailed description of each of the components[16].

Several Wiki types of components have been identified during the Bottom-Up analysis phase, among which TripleWiki, KaukoluWiki, RichWiki, SemWiki and ImapWiki, all resumed into the abstract ***Wiki*** component. The TripleWiki is a simple semantic Wiki with direct statements entries. The KaukoluWiki is a Wiki which allows semi-automatic metadata generation. The RichWiki is an Eclipse-based Wiki, which provides support for rich interactive clients. The result from the SemWiki will be a hypertext-based prototype for personal knowledge management. The ImapWiki is a Visual Knowledge Workbench with Wiki-abilities. It is a prototype for visual as well as textual editing of knowledge structures.

The ***Structure Recommender*** Component refers to pro-active structure recommendation. Using Ontology-based Information Extraction (OBIE) the system will derive certain structural patterns and metadata automatically from text such as Wiki pages, emails, etc... To deal with user feedback the OBIE components will use mixed-initiative learning, in which the user first does the recommendation task alone, then the machine learns how to suggest answers, which the user corrects where necessary, and so on.

The ***Related Item Recommender*** Component makes related item recommendations based on content, structure and semantic data. As with the previous component, the system is able to learn from user feedback.

The ***User Context Service*** will realize a set of interfaces and services to allow the observation of and reasoning about a user's current work context. Interfaces and prototypes for context detection plug-ins will be realized. When integrated into the semantic desktop, these plug-ins will allow to elicitate knowledge about the current goals of the user which in turn is useful for tuning information structuring, storage, and retrieval services.

The ***PIMO Service*** hosts most ontologies that need to be accessed in an integrated way. It stores the personal information model/conceptual data structures. Also, it allows qualified access to the personal information model, to functions which create classes, instances, relate instances to tags, etc.

The ***Data Wrapper*** allows access to several data source types. The different adapters / wrappers are packaged as one big component (for ease of use and installation).

The ***RDF Store*** represents the local database to store all metadata of the user, including ontologies, metadata of files, etc. We would suggest providing four different storage areas, with distinction made on requirements and use. Each is a separate RDF repository with different features: (1) ontology store, containing all ontologies plus CDS

[16] Please note that the naming of the components is originated by the reengineered prototypes and might be changed later to reflect a more proper design.

(Conceptual Data Structures)/PIMO, (2) resource store, containing all crawled resources, files, e-mails, address book, calendar, web-sites, (3) configuration store, containing configuration data of the system, and (4) service data store, containing routing tables, context trails, observed user actions, statistic data, etc. These areas are still to be further discussed and agreed within the consortium.

The aim of the ***Local Search*** infrastructure is to extend traditional full text search with more sophisticated metadata based searching and ranking capabilities. The component builds upon an existing open source desktop search engine, such as Novell / SuSE Beagle[17]. The work comprises the integration of semantic metadata into the search paradigm and the realization of personalized and desktop-adapted ranking mechanisms for search results.

The ***Personal Task Manager*** is a tool for personal process management which allows for the ad-hoc definition of activities (including hierarchical refinement) and their interconnection to various documents in the personal workspace.

The ***Distributed Index*** component offers a distributed index of data shared by users of the system. The component guarantees that users can find data shared by other users by replicating the index information among participating computers. The data itself remains at the providing user and is not replicated, therefore not accessible if the users goes offline, even though other users would be able to query it. The distributed index offers the four basic operations required to insert, update, delete and retrieve data.

The ***Metadata Exchange Recommender***. Current social networking research is mostly targeted at analyzing the interactions between users and the communities they generate. Although this is a necessary step in building social software, it is just its beginning. In order to interact and find relevant material in a community, there must be a concrete mechanism for metadata exchange between users, possibly subject to several access models. This component will thus result in desktop metadata sharing solutions, as well as recommendation algorithms for our social semantic infrastructure.

The ***Ontology & Metadata Aligners*** component consists of several pieces: A Self Organized Metadata Aligner, a Semi-Automated Metadata Aligner, and a Social Metadata Aligner. The Metadata Aligner will implement one or more metadata alignment methods, as well as some means to automate these proposed solutions. The component uses name-based, structure-based, and content / instance-based similarity measures in order to compute the ontology mappings. Finally, it will use data gathered from the community to discover relationships between items in NEPOMUK's knowledge base.

The ***Community Manager*** consists of a Community Detector, a Community Labeller, as well as a Community Structure Analyzer. They are all applications of Semantic Social Network Analysis and analyze workflows and data available on the Social Semantic Desktop.

[17] http://beagle-project.org/Main_Page

The ***Ranker***. Along searching for metadata into our social circle, we will probably be overwhelmed by the amount of responses we obtain, especially considering the already very large amounts of data encompassed by current PCs. Thus, different ranking algorithms based on shared metadata and generic user ranking information will be brought in place, in order to facilitate a fast access to the high quality shared resources in the social environment.

The ***NEPOMUK Backbone*** is the main component responsible for the interconnection of the other components, providing the means for registration and authentication of the existing services.

The ***Data Services*** incorporate the data provider functions for data and metadata for the other components.

6. Backbone and connector infrastructure

In the context of the Semantic Desktop, the roles of the Backbone and Connector Infrastructure can be split into two main categories:

- Interoperability and invocation (delegation)
- Registry – publication and discovery

The first main role of the backbone is to assure a transparent interoperability between several applications, represented as services on the desktop. The interoperability issue will be solved by establishing a standard communication and interaction protocol. The backbone will describe the protocol to use for contacting the services and furthermore how services are / will be described. A service requiring a certain functionality will use the protocol and interfaces defined and will not be aware of the provider's underlying implementation details.

The second main role deals with the publication and discovery of the existing services on the desktop. The backbone aims to act as a *desktop service registry*, able to accept publication and discovery inquiries from local services. Thus, one service requiring a certain functionality does not need to know the provider's location a priori, but instead it needs only to ask the registry for it.

In the following, we will start by defining the interoperation problem through the description of a possible scenario in the context of the Semantic Desktop and afterwards, detail how we intend to solve the two afore mentioned issues, i.e. interoperability and registry.

6.1. Defining the problem

A good way to understand and define a possible problem is by modelling it in terms of a scenario. We will place our scenario in the context of a typical desktop and make a series of feasible assumptions in order to show the roots from which the application interoperation problem arises.

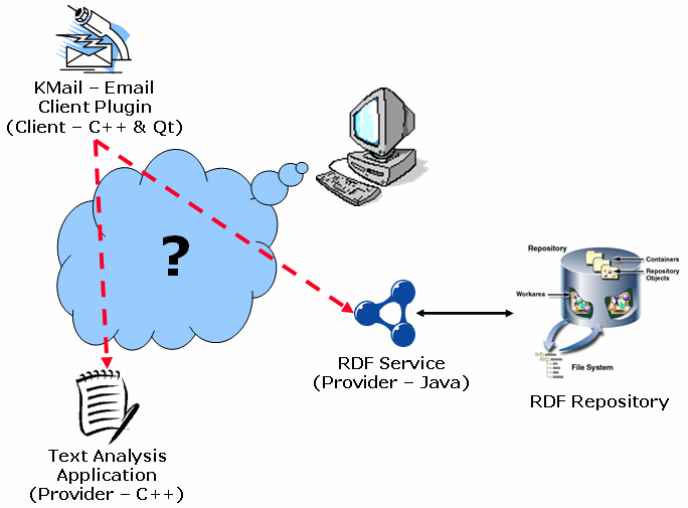

Figure 8. Example of a possible application inter-operation scenario on the desktop

Let us assume the existence of the following two applications on a desktop, which will rest on a Linux operating system (the scenario can be

equally applied to other operating systems, and it is depicted in Figure 8):

- An RDF Service Provider, implemented in Java and exposed as a Java Web Service (using SOAP/HTTP), offering the possibility to store and retrieve data in RDF format (for e.g. semantic annotations about documents), and also providing support for inference on the already existing data. It can be seen as a front-end for an RDF Repository.
- A plugin (or extension) inside the KMail email client, implemented in C++ & Qt, and having the goal of collecting particular information from the exchanged emails and contacts. The plugin will ideally act as a client for the RDF Service in order to be able to store the collected information and use the Provider's inference capabilities.

The problem that arises in this simple setting is the difficulty of communication due to the difference of the programming language in which the two applications are implemented. A quick and fast solution to this, would be the use of a particular intermediary application capable of understanding requests written in C++, translate them into SOAP/HTTP, delegate them to the RDF Service Provider, and then fetch the results and transform them back into C++. In this particular case, the above mentioned solution is sufficient.

At a later stage, we assume the existence of a Text Analysis Application, implemented also in C++ and having functionalities such as word detection and counting, semantic annotation, Named Entity Recognition or text summarization available for the desktop. The KMail Plugin could make use of this application to detect the context of the received emails, based on the body of the message and use this as additional information in its reasoning process. In this particular case, the communication could take place directly through method invocation, presuming that the client would know a priori the interface provided by the Text Analysis Application.

The problem increases in complexity, with the number of added applications and thus, combining direct method invocation with particular communication channels between each pair of applications does not represent a feasible solution any more. Therefore, the need for a unified general approach becomes crucial in order to properly solve the interoperability issue.

Furthermore, until now, we assumed that the KMail plugin is implicitly aware of the existence of both applications and knows exactly how to connect to them and use them. This would work in a static setting in which users always use the same application(s).

However in a typical desktop setting the situation is different. The variety of end-user applications is quite large and it is sufficient to consider only a certain category of applications in order to observe that the afore mentioned assumption does not stand. Thus, we have to be able to deal with different applications providing similar functionality and to allow a certain degree of flexibility in discovering existing applications which are open for collaboration. A possible solution to these two issues can be captured by the standardization of the interfaces provided by the applications which fall into a particular category and creating a discovery mechanism to be used by all applications which have functionalities to provide and by all probable clients of these functionality providers.

The NEPOMUK backbone and connector infrastructure aims to solve two of these three issues, i.e. the application interoperation and discovery,

while the NEPOMUK Semantic Desktop framework provides a solution to the third one, i.e. application interface standardization.

6.2. NEPOMUK Backbone architecture

This section will give an overview of the NEPOMUK backbone and connector infrastructure starting with the description of its internal organization, then detailing the NEPOMUK Services and Registry, and in the end how inter-component communication is achieved.

6.2.1. Backbone components

Internally the NEPOMUK Backbone is organized in three layers: the Client layer, the Core layer and the Provider layer. Each of these three layers (see Figure 9) is represented by a particular component, described as follows.

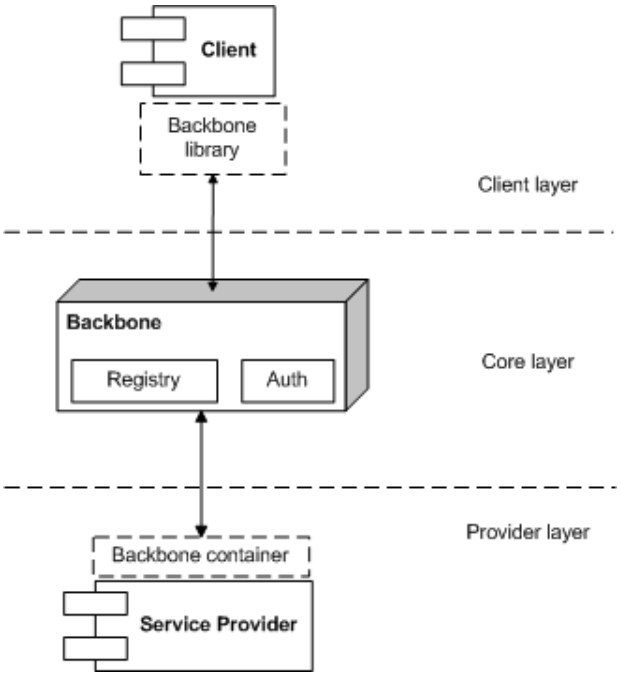

Figure 9. NEPOMUK Backbone layered organization

The Client layer is represented by the **Backbone library**. This library resides in the client component's environment and provides an abstract interface which the client uses in order to achieve communication with the Core layer. The role of the library is to hide the communication implementation details and detach the business logic part of the communication between clients and service providers from the clients. As a direct result, the client will always use the same abstract interface to communicate with the core layer and the service providers and thus will not be aware of the actual way in which the communication is achieved. This approach is similar to the abstraction layer present in Jade[18] (Java

[18] http://jade.tilab.com/

Agent DEvelopment Framework), or to other approaches present usually in agent environments.

From the implementation point of view, the library is implemented in the client's programming language and has a "dual-sided" organization. On one side it provides the abstract interface against which the clients are programming. On the other side, it realizes the actual communication with the core layer of the Backbone and with the service providers. One could see this library as a black box, having always the same two sides, but without knowing how and in which language is the interior realized. This was also our goal: to provide a uniform way of achieving the communication between clients and providers, without considering the underlying implementation details. The Backbone Library API can be found in Annex B.

The Provider layer is represented by the **Backbone container** which is the provider side alternative for the Backbone library. It assures the flexibility of moving a service provider from one environment to another without the necessity of changing the implementation. Both the library and the container represent logic blocks of the backbone, although physically they reside in the client's and respectively provider's environment.

The Core layer contains the **Core Backbone**, including two logical blocks: the Backbone Registry and the Backbone Authentication mechanism. The main roles of the Core Backbone are:

- to provide a way for service publishing and discovery, discussed in detail in Section 6.2.4,
- to route the exchange of messages between a client present in a particular environment and a service provider present in a different environment, detailed in the following section,
- and, to offer the necessary authentication mechanisms, which in this phase are only part of the design, but not yet realized.

6.2.2. Backbone federation

The way in which the actual communication between clients and service providers is realized, is mainly driven by the underlying implementation language of the Core Backbone. By following this approach we wanted to optimize to the maximum the communication mechanism and to eliminate any possible overhead. As an example, if the Core Backbone is implemented in C++ and uses the DBus[19] protocol, and the client and service provider are also implemented in C++, then the communication is realized using their native protocol, i.e. DBus. In this case, the role of the Core Backbone is only to provide the registry information, the real communication being done directly between client and service provider (to be more specific, between the backbone library and the backbone container).

The example presented the most often encountered scenario in a typical desktop setting, and also the perfect way in which communication can be

[19] http://dbus.freedesktop.org/

realized. But, in order to allow the freedom of deploying services implemented also in different programming languages than the one in which the Core Backbone or the clients are implemented, we introduced the notion of **Backbone Federation**.

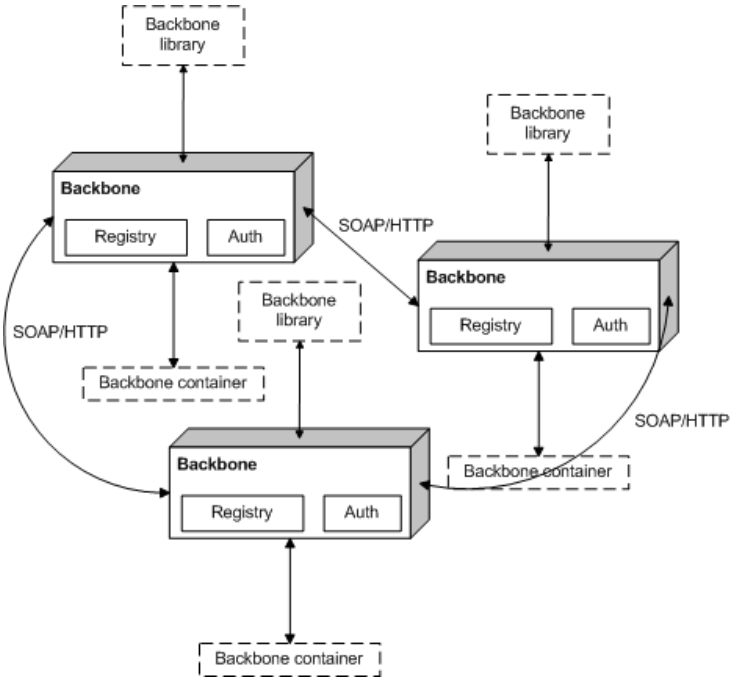

Figure 10. Backbone federation

The federation assumes the existence of multiple backbones on the same desktop, implemented in different programming language and having their own communication protocols. The reader has to keep in mind that, every implementation respects the imposed standards, i.e. every published interface of the logical blocks of the backbone is the standard interface.

The presence of multiple backbones assures the communication efficiency between the clients and service providers implemented in the same programming language and communicating via the same protocol. In order to provide a standard way to realize also the communication between clients and service provider implemented in different languages and having different communication protocols, the Core Backbones will create virtual paths between them and thus opening direct communication channels by exchanging standard messages using SOAP/HTTP. Figure 10 depicts the organization of a federation.

Each newly deployed Backbone will contact the other existing backbones and register itself with them. And thus, when a client will send a service discovery inquiry to its native backbone, the inquiry will be propagated to all the registered backbones and the complete result will be returned to the client. Also, all the communication between a client and a particular service will be routed through the two respective backbones to which they belong to. A detailed description of the possible communication scenarios is presented in Section 6.2.5.

6.2.3. NEPOMUK Services

A service is a desktop application deployed as a component in the NEPOMUK architecture, having an “end point”, through which it will be accessed by other components, and defining its interface as a WSDL description. The description will contain the operations available for the specified service. At deployment time, the level of access will also be defined in order to be able to use these operations.

Moreover, all the provider applications are exposed through standard interfaces allowing users to replace an application with another without making any modifications. Also, the communication takes place without considering the underlying programming language. The WSDL service description raises the definition to an abstract level, hiding the implementation details and therefore providing a uniform and transparent way for inter-component communication.

The overall goal of the NEPOMUK architecture is to define a set of services having standardized interfaces, with the remark that the services themselves do not represent part of the backbone. They represent the method through which the functionality of a component is defined and accessed.

6.2.4. NEPOMUK Registry

The NEPOMUK Registry represents the main container of the registered NEPOMUK services. It has three main roles: to register new NEPOMUK services, to un-register existing ones and to provide answers to the received discovery inquiries. It is important to point out that the NEPOMUK Registry is not involved in the service invocation.

The advantages brought by using such an approach are the following:

- All the services need to know only the NEPOMUK Registry’s end-point, as opposed to a static inter-service communication where the service would be obliged to know directly the other services’ end-points.
- Publishing and discovering of the existing NEPOMUK services is done by using a standard interface.
- The discovery inquiry support offers a high flexibility in finding a particular service based on the required functionality as opposed to its name.

We argue that the above mentioned set of advantages is enough to counter-balance the biggest disadvantage of this approach, i.e. the overheads introduced by the discovery, which could fluctuate depending on the actual implementation. A view over the Registry API can be found in Annex A.

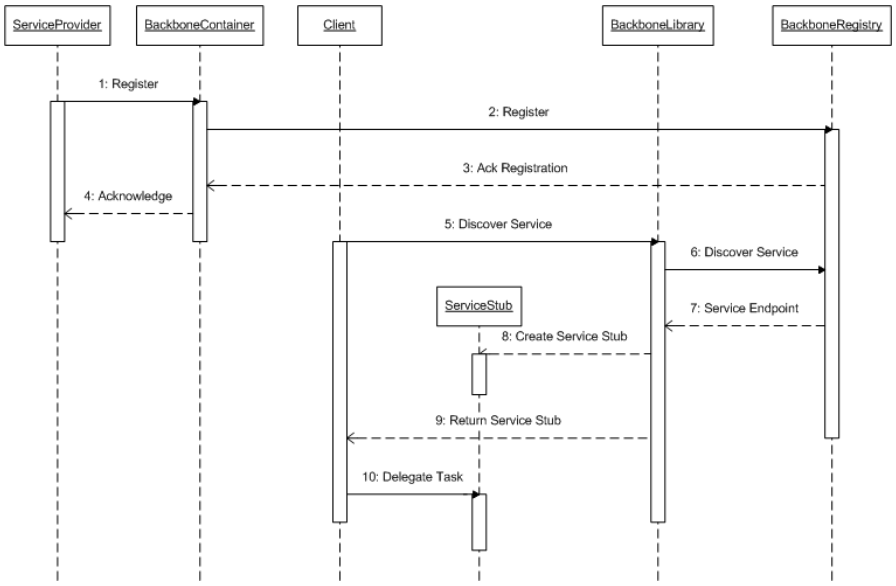

Figure 11. Sequence diagram showing the interaction steps between the services and the NEPOMUK Registry

Figure 11 details the steps needed to be performed in order to publish and to discover a particular service:

- The Service Provider communicates the intention of being published to the Backbone Container.
- The Backbone Container contacts the Backbone Registry and registers the Service
- The Backbone Registry responds to the Backbone Container with a registration acknowledgment that is afterwards propagated to the Service Provider. In this moment the service can be discovered by eventual client components.
- A client uses the standard interface of the Backbone Library to discover a message.
- The Backbone Library asks the Backbone Registry for the particular service.
- Presuming that the inquiry was successful, the Backbone Registry returns the service's endpoint.
- The Backbone Library creates a service stub encapsulating the service's endpoint and returns the stub to the Client.
- The Client uses the stub to delegate the communication tasks with the actual service.

The afore-mentioned discovery scenario described conceptually how the discovery mechanism is realized. The way in which the discovery is actually performed fits in the two possible communication scenarios and represents the subject of the following section.

6.2.5. Inter-component communication

As shown in the beginning of this section the communication between clients and service providers can be achieved in different ways depending on the deployment situation. In reality, there are two possibilities, which we will detail as follows.

6.2.5.1 Native communication

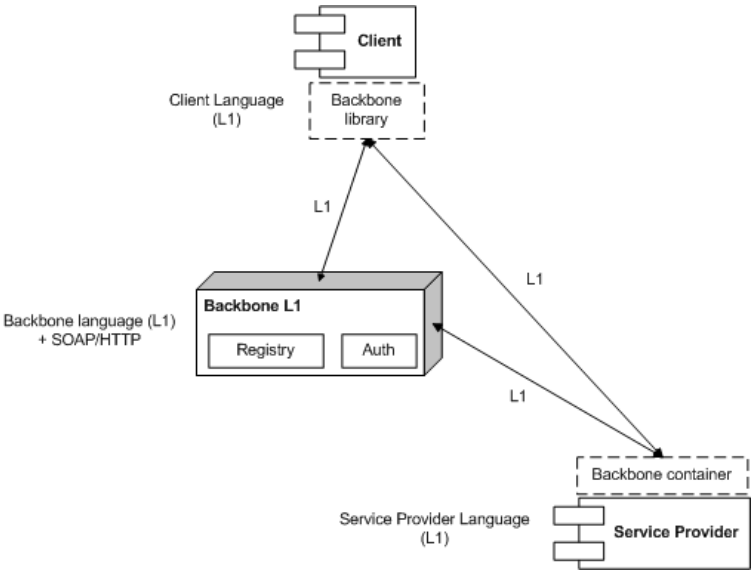

Figure 12. Native communication scenario

Native communication is achieved when all three participants are implemented in the same programming language and using the same communication protocol, i.e. the client, the service provider and the core backbone. In this setting, the backbone library will use the Core backbone only for its registry capabilities, the actual communication with the service provider being realized directly, as shown in Figure 12. The advantage is represented by the absence of any communication overhead, which may appear when the messages need to be transformed from one format to another. We could state that most of the communication realized on a usual desktop will profit of this scenario.

6.2.5.2 Non-native communication

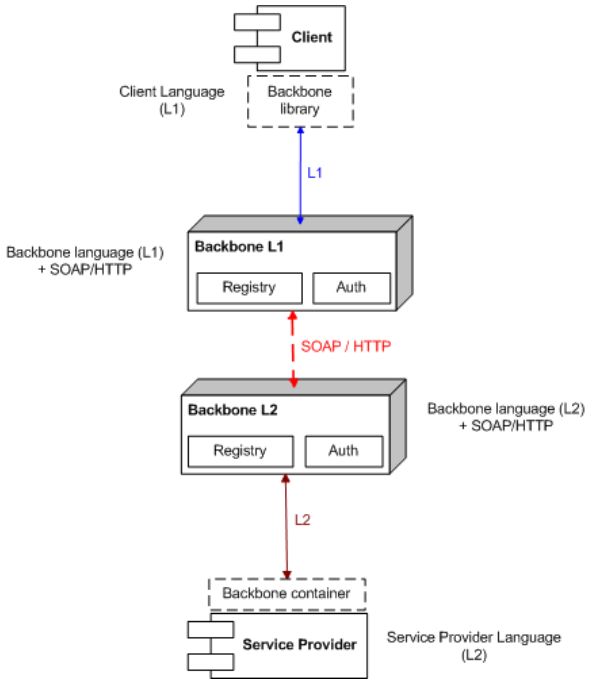

Figure 13. Non-native communication scenario

The non-native communication takes place when the desktop hosts a federation of backbones. In this case the backbone library uses the backbone core as a message router. As depicted in Figure 13, the backbone library communicates with the Backbone Core in its native protocol, the message is then being transformed into SOAP/HTTP by the core, routed to the proper backbone, which transforms it into its native protocol and passes it to the backbone container.

6.3. Scenario revisited

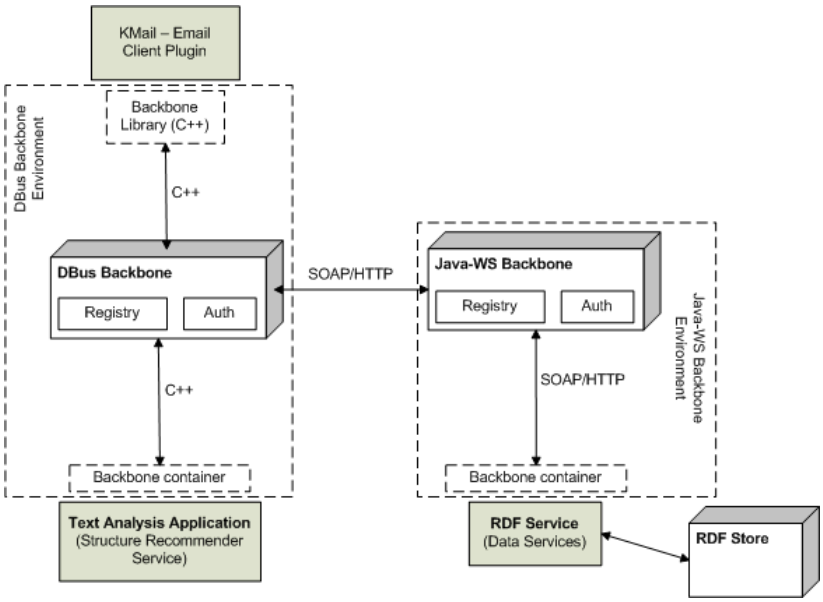

Figure 14. Application interoperation scenario in the NEPOMUK Architecture context

Coming back to the scenario presented in Section 6.1. , we will now present it in the context of the NEPOMUK Architecture and in the presence of the NEPOMUK Backbone (see Figure 14). The following transformations occurred:

- Both the Text Analysis Application and the RDF Service were modeled as standard NEPOMUK Components (see Section 5.1.), and therefore the Text Analysis Application is represented here by the Structure Recommender Service and the RDF Service by the Data Services.
- Due to the different underlying programming languages of the two services, each was deployed as part of a particular Backbone, able to provide direct communication between the Backbone Core and the respective Service.
- As a final step, in order to achieve the communication presented in the scenario, we created a backbone federation, as explained in Section 6.2.2.

The big advantage of this modeling approach is that one is free to deploy new C++ and Java-WS clients which will directly profit of the existing native communication possibilities and also of the already present federation of backbones.

The reader has to note that this solution represents is an optimal solution for deploying services implemented in different programming languages and using different communication protocols. It takes the advantage of

the presence of the backbone federation and profits to the maximum from the underlying native communication protocol. In reality there could be a simpler solution presuming the existence of a single backbone and all services deployed in it. In this case, the Backbone Container takes care of all communication issues that might arise as a consequence of the usage of different programming languages and communication protocols by the existing services and backbone.

7. Backbone and Connector infrastructure Implementation

The design of the backbone brings the following advantages:

- establishing a standard for NEPOMUK components to describe their services,
- establishing a standard for NEPOMUK components for their communication,
- freedom in the choices of the programming language and
- freedom in the choice of the native communication platform.

First prototypes of the NEPOMUK backbone have been implemented based on the previous defined interfaces. Currently, there are two undergoing prototype implementations: the reference implementation in Java, using Web Services Architecture[20] and a second one, in C++, using KDE and the native Linux DBus communication platform[21]. An implementation supporting OSGI process calls will follow.

The Java-Web Services implementation was realized according to the architecture described in the previous section.

- **The Core Backbone** is represented by a Web Service implemented in Java and deployed as an Axis2[22] service in the Apache Tomcat[23] web container. Due to its deployment type, the communication with the Registry can be achieved by using SOAP over HTTP and REST, the difference being given by the different endpoint.
    - The Registry implements the standard NEPOMUK Registry interface and already provides the support for the backbone federation. In addition, it creates a local store of registered NEPOMUK components which is loaded and verified at start-up and saved every time a modification appears.
- **The Client library** provides the standard interface as a Java interface which can be directly used during the implementation. It hides the underlying communication mechanisms, by transforming the local method invocations into SOAP/HTTP core backbone calls. The current status provides the support for the non-native communication (see Section 6.2.5.2), but it still needs proper testing before the actual deployment.
- **The Backbone container**. Due to the Web Services nature of the implementation, the service providers need to be deployed as Web Services, and thus providing the standard WSDL interface and the necessary SOAP/HTTP communication. Therefore, this

[20] http://svn.nepomuk.semanticdesktop.org/repos/trunk/component/Comp-Backbone/Java-WS-Backbone/

[21] svn://anonsvn.kde.org/home/kde/branches/work/nepomuk-kde/

[22] http://ws.apache.org/axis2/

[23] http://tomcat.apache.org/

component is represented in our case by Axis2. In the near future, we intend to implement a flexible Backbone container which will provide the ability of moving a NEPOMUK service from one container to another without the necessity of changing the underlying implementation.

In general, the beauty of the solution is given by the fact that all component owners program against the same standardised backbone interfaces independent from the underlying means of communication. As well as all services provider will describe their interface in a standardised way by means of WSDL description. These descriptions will be either used directly, in the case of the web service implementation or interpreted accordingly in the case of the native implementation such as OSGI or DBUS.

8. Discussion and Conclusions

Software architectures are not specifications. They are intended to convey a common understanding and make people understand what is going on in a system, viz. one of the main goals of architecture is to communicate a design. This is especially important in a project of the size of NEPOMUK where we are concerned about a common understanding between over a hundred individuals. Hence, the starting point architecture presented here and the first version of the connector and backbone infrastructure are the result of many discussions and outcomes of consensus processes. Both represent a sound basis for the continuation of our work. While we presume that the core design of the backbone presented here is quite stable, we are also quite aware that the NEPOMUK architecture must undergo additional clarification and refinement steps within the next few months.

The work presented here resulted into the following:

- An initial starting point for the NEPOMUK architecture, which describes an initial harmonized view of early software services and components
- A backbone and connector infrastructure framework and API.
- Implementations of the backbone and connector infrastructure in the form of libraries. Firstly, a reference architecture using platform neutral web services infrastructure. Secondly, native implementations for specific platforms.

In addition to the major contributions listed above the work provided several additional insights and remaining questions, such as:

- Throughout Nepomuk we defined methods that use RDF for transporting any complex parameters/return values for all types that are explicitly defined in our Ontologies. How to describe the RDF data (or ontological concepts) in the WSDL interface description?
- Using Ontologies in the Software Architecture shifts the engineering from the software design to ontology design. Significant behaviour within the system might be modelled by ontologies. What is an appropriate methodology and tool support for such an ontology driven architecture?

For the next months we envision the following steps:

- By extending the functionalities of the backbone and connector infrastructure it will begin to further resemble semantic desktop middleware. Hence, we need to adopt the name accordingly as well as to define the core set of services that such a semantic desktop middleware has to provide.
- Also for the NEPOMUK architecture we will define and standardise the core set of services which denotes a social semantic desktop.
- In general the design of the NEPOMUK architecture will be improved by applying an interwoven approach of top-down and bottom-up analysis. The top-down analysis will consider the feature request of the NEPOMUK cases studies while the bottom up analysis will consider further insights of the evolutionarily prototypes.

Ideally, the backbone and connector infrastructure should interact with operating system libraries. The Social Semantic Desktop services should start as services of the native operating systems, just as present day file systems and network services are started. This would also enable better integration of external data sources. Current implementations include Spotlight on Apple's OSX or the index services on Microsoft Windows. We anticipate semantic indexing will become a part of future operating systems, based on the NEPOMUK standards.

References

[BernersLee2001] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. In *Scientific American*, May 2001.

[Brown2003] P. Brown. DCOP: Desktop COmmunications Protocol. May 2003. http://developer.kde.org/documentation/other/dcop.html

[Cheyer2005] A. Cheyer, J. Park, R. Giuli. IRIS: Integrate. Relate. Infer. Share. In Proceedings of the 1st Workshop on The Semantic Desktop - Next Generation Personal Information Management and Collaboration Infrastructure at the International Semantic Web Conference, Galway, Ireland (2005)

[Christensen2001] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1, W3C Technical Report, March 2001, http://www.w3.org/TR/wsdl

[Cutrell2006] E. Cutrell, D. C. Robbins, S. T. Dumais, R. Sarin. Fast, flexible filtering with Phlat - Personal search and organization made easy. In Proceedings of the SIGCHI conference on Human Factors in computing systems. Montreal, Quebec, Canada (2006)

[Dong2005] X. (Luna) Dong, A. Halevy. A Platform for Personal Information Management and Integration. In Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, Canada (2005)

[Fielding2000] R. T. Fielding, "*Architectural styles and the design of network-based software architectures*", Dissertation, 2000

[Gudgin2003] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Technical Report, June 2003, http://www.w3.org/TR/soap12-part1/

[Karger2005] D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data. In Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, Canada (2005)

[Richter2005] J. Richter, M. Volkel, H. Haller. DeepaMehta - A Semantic Desktop. In Proceedings of the 1st Workshop on The Semantic Desktop - Next Generation Personal Information Management and Collaboration Infrastructure at the International Semantic Web Conference, Galway, Ireland (2005)

[Sauermann2006] L. Sauermann, G. Grimnes, M. Kiesel, C. Fluit, D. Heim, D. Nadeem, B. Horak, A. Dengel. Semantic Desktop 2.0: The Gnowsis Experience. Semantic Web In Use Track of the 5th International Semantic Web Conference (ISWC), Athens, Georgia, USA (2006)

[Tummarello2005] G. Tummarello, C. Morbidoni, P. Puliti, F. Piazza. The DBin Semantic Web platform: an overview. In Proceedings of WWW2005 Workshop on The Semantic Computing Initiative (SeC), Chiba, Japan (2005)

[Gnome] Gnome Bonobo. http://www.gnome.org/gnome-office/bonobo.shtml

[Oasis2004] OASIS. Introduction to UDDI: Important Features and Functional Concepts. October 2004. http://uddi.org/pubs/uddi-tech-wp.pdf

[OSGI2006] The OSGi Alliance. OSGi Technology. September 2006. http://www.osgi.org/osgi_technology/index.asp?section=2

Annex A – NEPOMUK Registry API

Function	Description
registerService(String uri, String typeDescription)	- registers a service based on the provided uri
registerServiceByName(String name, String uri, String typeDescription)	- registers a service based on the provided name and uri
unregisterService(String uri)	- unregisters a service based on the provided uri
unregisterServiceByName(String name)	- unregisters a service based on the provided name
discoverServiceByType(String typeDescription)	- returns the service descriptor of the discovered service based on the provided WSDL file URL
discoverServiceByName(String name)	- returns the list of service descriptors representing the discovered service based on the regex
discoverServiceByURI(String uri)	- returns the service descriptor of the service with the given uri
allServices()	- returns the list of all registered services
registerRegistry(String uri)	- registers a new Registry as part of the federation
unregisterRegistry(String uri)	- unregisters a Registry from federation

Annex B – NEPOMUK Backbone Library API

Object / Function	Description
BACKBONE – Singleton	
getRegistry()	- returns an instance to the NEPOMUK Registry
REGISTRY	
discoverService(String typeDescription)	- returns a NEPOMUKService discovered by the provided type
discoverServiceByName(String name)	- returns a list of NEPOMUKService(s) discovered by the provided name (regex)
registerService(ServiceDescriptor servDesc)	- registers a new NEPOMUKService in the Registry
unregisterService(ServiceDescriptor servDesc)	- unregisters the provided NEPOMUKService
unregisterService(String uri)	- unregisters a NEPOMUKService based on the provided uri
NEPOMUKService	
sendMessage(Message message)	- sends a message to the current NEPOMUKService and returns a Result
getServiceDescriptor()	- returns the ServiceDescriptor for the current NEPOMUKService
Message	
addParameter(Object value)	- adds a new Parameter to the list of parameters
setParameters(List<Object> values)	- sets the list of parameters to the one provided
setMethod(String name)	- sets the name of the method
Result	
getValue()	- returns the value of the result. Note: the type of the value is already known by the user, so one could cast directly the returned type to the expected one
getStatus()	- returns the status of the result
ServiceDescriptor	
getName()	- returns the name of the service
getTypeDescription()	- returns the type description (WSDL URL)
getURI()	- returns the service's uri