KAON SERVER Architecture

Boris Motik[2], Daniel Oberle[1], Steffen Staab[1], Rudi Studer[1,2], Raphael Volz[1,2]

[1]University of Karlsruhe

Institute AIFB

D-76128 Karlsruhe

email: {lastname}@aifb.uni-karlsruhe.de

[2]FZI - Research Center for Information Technologies

Haid-und-Neu-Strasse 10-14

D-76131 Karlsruhe

email: {lastname}@fzi.de

AIFB

Technical Report No. 421

Identifier	Del 5
Class	Deliverable
Version	2.0
Date	09-05-2002
Status	Final
Distribution	Internal
Lead Partner	AIFB

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project coordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Executive Summary

This deliverable describes the architecture of KAON SERVER, which realizes the core technical infrastructure for the WonderWeb project. The development is carried out within workpackage 2. Since KAON SERVER is the infrastructure kernel of the project, its architecture must support and is influenced by the activities described in the other work packages. Developing KAON SERVER comprises means for component management making the architecture dynamically extensible, an API allowing transparent access to ontologies and finally a multi-user capable transactional system for storage of ontologies and instances. Besides, it is tightly connected to several external services, e.g. inference engines, to offer additional functionality to clients. The challenge of this task is the provision of capable, but open interfaces that allow the welding together of ontology-aware components that already exist, are to be developed in WonderWeb, or provided by the wider World Wide Web community. For this purpose, we exploit current state-of-the-art. If such technologies exhibit severe limits hindering the envisioned objectives, new solutions are sought and implemented. The architecture presented in this deliverable has to be viewed as an initial design which will be evaluated and incrementally improved. The latest versions can be accessed via the Karlsruhe Semantic Web and Ontology Tool Suite (KAON) website http://kaon.semanticweb.org.

Contents

Chapter 1

Introduction

The main organizational unit and infrastructure kernel of workpackage 2 is represented by KAON SERVER. It provides functionality for managing components allowing the dynamic extension of the server by new functionalities and offers access to ontologies and data. Besides, KAON SERVER delivers a comprehensive solution for the storing of ontologies and data. Existing clients have to be connected and adapted, new clients have to be developed that hook up to KAON SERVER to provide a range of components including support for ontology development and maintenance, migration, sharing and integration. In general, the aim of WP2 is the development of the technical infrastructure and tool support that are required for real world applications in the Semantic Web. The tools and components developed in this workpackage will also offer support to the activities described in the other workpackages. The design takes into account language development efforts (WP1), ontological engineering methodologies (WP4), experiences gained when connecting clients (WP2), and input from members of the industrial advisory board (WP5). Several Semantic Web related projects also pose requirements on a Semantic Web infrastructure that are taken into account as far as possible. Among those are the European Union funded projects OntoWeb, Onto-Logging, Semantic Web Enabled Web Services and SWAP. A detailed list of requirements and projects is given

in section 2.1. KAON SERVER is made up of several modules which can be dynamically connected. Its core is the component management[1] allowing allocation and dynamic loading of components (cf. chapter 4). Besides, there is a module that offers access to ontologies and data (cf. chapter 5) as well as an optional module providing data persistence (cf. chapter 6). Our basic methodology for the development is to rely on available state-of-the-art solutions - as far as possible - in order to provide a comprehensive architecture with features that could not possibly be built within WonderWeb itself and in order to have early versions of the KAON SERVER early in the project. For instance, an implementation of Java Management Extensions (JMX) is used for Component Management within KAON SERVER (cf. chapter 4). Only if the usage of such technologies exhibits severe limits hindering the envisioned goals, we will take on development by ourselves. This has been done for the Ontology API (cf. chapter 5) and the transactional RDF Server (cf. chapter 6). Even then, we try to reduce development efforts by reusing existing technologies. For instance, we exploit a relational DBMS for the aforementioned RDF Server. The architecture presented in this deliverable has to be seen as initial design which is evaluated and incrementally improved. The KAON SERVER will become part of the open-source Karlsruhe Semantic Web and Ontology Toolsuite (KAON) described in chapter 2. This report takes the following structure: Chapter 2 briefly describes our comprehensive KAON Framework [2] which can be seen as the technical environment in which the architecture lives. We give a survey on the conceptual functionality and technical architecture of KAON SERVER in chapter 3. Its core modules, i.e. component management, data access and data persistence, are highlighted in chapters 4, 5 and 6 respectively.

[1] Instead of component management one could also speak of service management. We decided in favor of the first, because "service" is awkwardly overloaded.

Chapter 2

Framework

This chapter gives a brief introduction to the Karlsruhe Ontology and Semantic Web Tool Suite (KAON). KAON provides the structural environment for the development of KAON SERVER and contributes several components to the development of KAON SERVER. We will present those parts of its architecture that are concerned with this deliverable.

The KAON Ontology and Semantic Web Tool Suite builds on experiences from previous developments and projects dealing with semantics-based applications in the areas of E-Commerce, Knowledge Management and Web Portals. From these experiences, we have collected requirements that we now fulfill for the construction of a Semantic Web infrastructure.

Section 2.1 provides a detailed list of requirements for KAON and section 2.2 gives a brief survey of the KAON architecture.

The KAON SERVER components are developed in context of the KAON Ontology and Semantic Web tool suite [2]. KAON is an open-source project and joint effort by both the Institute AIFB[1] and the Research Center for Information Technologies (FZI)[2].

[1] AIFB, www.aifb.uni-karlsruhe.de
[2] FZI, http://www.fzi.de

The development of KAON is carried out in and sponsored by several European Union projects:

WonderWeb

Ontology Infrastructure for the Semantic Web

SWAP

Semantic Web and Peer-to-Peer

OntoWeb

Ontology-based information exchange for knowledge management and electronic commerce

OntoLogging

Corporate Ontology Modelling and Management System

SWWS

Semantic Web Enabled Web Services

2.1 Requirements

An extensive requirement gathering process was undertaken to identify what must be met by the KAON Framework [2]. For KAON SERVER, there are further requirements stemming from other workpackages (language development efforts WP1, ontological engineering methodologies WP4, connection of clients WP2, and input from members of the industrial advisory board WP5) as well as from the other projects. All this led to following list of requirements:

Accessability

A framework should enable loose coupling, allowing access through standard web protocols, as well as close coupling by embedding it into other applications. This should be done by offering sophisticated standard APIs.

Consistency

Consistency of information is a critical requirement of any enterprise system. Each update of a consistent ontology must result in an ontology that is also consistent. In order to achieve that goal, precise rules must be defined for ontology evolution and components updating ontologies must implement and adhere to these rules. Also, all updates to the ontology must be done within

transactions assuring the common properties of atomicity, consistency, isolation and durability (ACID).

Concurrency

It must be possible to concurrently access and modify information. This may be achieved using transactional processing, where objects can be modified at most by one transaction at the time.

Durability

Durability is an almost trivial requirement easily accomplished by reusing existing database technology. A sophisticated storage system must offer facilities for replication: for often used ontologies redundant copies must be maintained to address scalability and availability problems.

Security

Guaranteeing information security means protecting information against unauthorized disclosure, transfer, modification, or destruction, whether accidental or intentional. To realize it, any operation should only be accessible by properly authorized agents. Proper identity of the agent must be reliably established, by employing authentication techniques. Confidential data must be encrypted for network communication and persistent storage. Finally, means for monitoring (logging) of confidential operations should be present. Those logs may be used further for the evolution of data sets that adhere to former versions of an ontology[20].

Reasoning

Reasoning engines are central components of semantics-based applications and can be used for several tasks like semantic validation and deduction of otherwise implicit information. KAON tools must have access to such engines, which can provide the reasoning services required to fulfill certain tasks.

Mapping

Often, there is a need for handling multiple ontologies. Complete support for handling multiple ontologies shall be given. This includes means for mapping and mediating between heterogeneous ontologies.

Distribution

We assume that data in the Semantic Web will be distributed. Therefore capabilities for accessing and aggregation of distributed information is required. Means for detecting and working with duplicate data sets are required. For example the same data accesible through with different reasoning services, which may provide distinct and unique services for that data. KAON should provide means to detect that the same data set is used. Also means for querying and composition of distributed data should be provided.

Localization

Since we assume that data in the Semantic Web will be distributed and the Web is a large and unsupervised information space, means for ontology-focused and intelligent localization of RDF and ontology-based data are required. Based on a semantic description of the search target, the system should be able to discover novel relevant information on the Web.

Internationalization

The framework should allow users to create ontologies and their instances in different languages and should support non-Latin character sets.

Formal semantics

The formal semantics specified by an ontology must be unambiguous and clear.

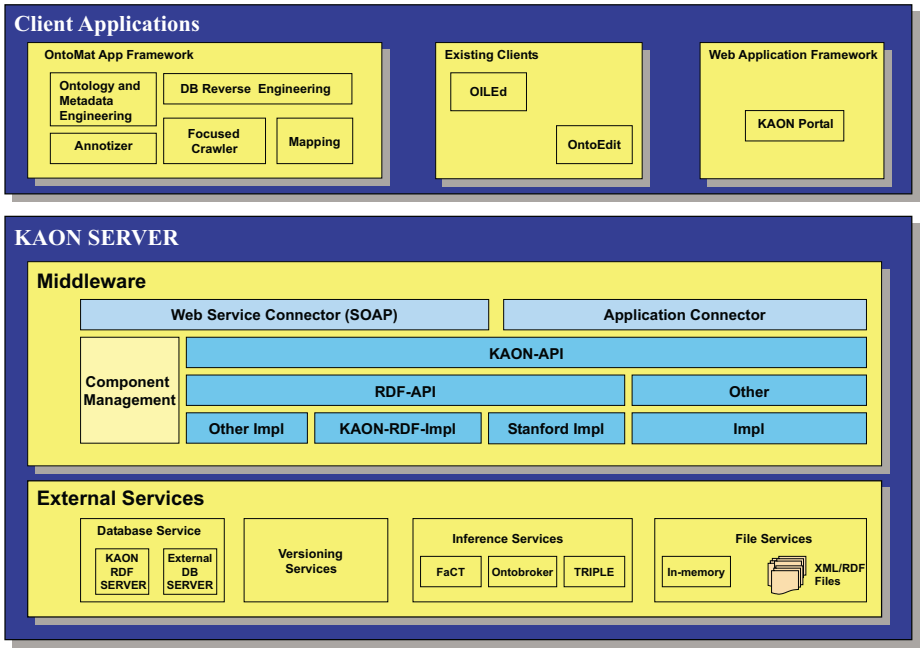

Figure 2.1: KAON Architecture

2.2 KAON Architecture

KAON builds on RDF [9] and provides specialized tools for the engineering, discovery, management, and presentation of ontologies and metadata. It is currently growing to a family of tools [2]. As Figure 2.1 depicts, three layers are distinguished, namely the client layer, the middleware layer and the external services layer.

Client Applications

Two separate lines of client applications are distinguished:

- **OntoMat** - desktop application framework
- **KAON PORTAL** - Web Portal framework [1]

Both application clients may either connect or embed components residing in the middleware layer. This may be the KAON API component, which provides

programmatic access to KAON ontologies and knowledge bases or the RDF API to access RDF data sets.

Application clients typically provide views and controllers for models realized by the KAON API.

Middleware Layer

The primary role of the middleware layer is to provide an abstraction for ontology and data access. It also allows for composition of components with functionality provided by external services. The latter is provided by the component management module which must support the dynamic instantiation and localization of appropriate functional components and delegate requests to such functional components. Functionalities may be provided by external services, thus it must be possible to delegate requests to those services. The interested reader may refer to chapter 4 for more details.

Data access is realized by two programmatic APIs:

- KAON API - offering access to ontologies and knowledge bases
- RDF API - offering access to RDF data

The KAON API provides programmatic access to ontologies and instances independent of the physical storage mechanism. It supports the aforementioned requirements by ensuring ontology consistency, adhering to the formal semantics and supporting composition of distributed data by means of modularization. Concurrent ontology access and transactional processing may be ensured by particular implementations.

Since the quality of performance depends on the specific application, the ontology API shields users from particular storage mechanisms. It supports operating on RDF files and on proprietary relational databases, which are used to make ontological persistent.

The RDF API provides programmatic access to RDF models. It features proprietary means for modularization, fast and efficient RDF parser and serializer as well as transactional access.

Data access is described in further detail in chapter 5.

External Services Layer

This layer has several roles. First it offers access to physical data stores such as databases, file systems or the network. Second it groups separate, external software entities such as reasoning engines. These services may not be managable by the middleware, unless proprietary management functionality is provided for individual external services.

For the demands of the WonderWeb project, this layer comprises versioning and inference components. The usage of several existing inference engines, like the description logics based FaCT [5], Ontobroker [19] or Triple [13], is possible and must be supported.

KAON itself provides two external services, namely an RDF SERVER, which is able to store RDF data and allows concurrent, transactional access to RDF data, and an ENGINEERING SERVER, which is able to store KAON ontologies and allows performant concurrent and transactional access to KAON ontologies. The latter uses an optimized database schema to increase performance in settings where an abstraction from RDF can be made, e.g. in ontology engineering scenarios.

Both services are intended to handle database content and may be used transparently through the KAON and RDF API. Non-RDF data sources may be accessed using other implementations of the KAON API, for example creating an ontology-compatible view of relational data, such as intended in OntoLift (cf. WonderWeb Deliverable 11 and 30).

Future versions of the KAON Tool Suite may provide further components. Additionally, mechanisms like views on ontologies, and an ontology query language are envisioned research prototypes developed by KAON participants.

2.3 KAON SERVER

In order to integrate already existing clients like OILEd [18] or OntoEdit [21], which were not developed within the KAON framework, further components will have to be provided. These components must adopt the KAON SERVER APIs towards their predefined structures.

From a client perspective, a KAON SERVER (or "OntoServer" as it was called in the WonderWeb project proposal) may be composed of different components provided by the KAON middleware layer and by third party external services. This composition is done using a dedicated component management module which registers and manages all functional components intended for a particular server instance.

There will be several KAON SERVER configurations, which are capable of providing different functionality depending on the selection of hosted components.

We decided to realize KAON SERVER in such a modular way to maximize the dissemination and use of individual functional components and to be able to benefit from external developments.

Chapter 3

Architecture

As described in chapter 2, a particular KAON SERVER configuration may be composed of central components of the KAON Framework. In the following we try to abstract from those individual components and take a functional approach for describing the architecture. First, intended functionalities are distinguished into five core layers, then individual components are classified into those layers. Later we sketch several simple usage scenarios and introduce the fundamental architectural pattern for the design.

3.1 Core functionalities

We can distinguish three core modules, which should be provided by every KAON SERVER configuration.

The first module provides the functionality for managing other components to allow the dynamic extension of the server by new functionalities. The second module has to offer access to data, e.g. the KAON API or RDF API. The third module has to provide data persistence, e.g. KAON RDF SERVER or ENGINEERING SERVER. These three core modules are discussed in detail in chapters 4 to 6, respectively.

3.2 Extended Functionalities

Besides, KAON SERVER should be extensible with further components, that provide more functionalities. Such as external services that facilitate the access to inference engines like FaCT [5] or Triple [13], other RDF databases such as Sesame [7] or arbitrary databases (cf. [8] and Wonderweb Deliverable 11 and 30: OntoLift) and many more.

It is important to notice, that extended functionality cannot be distinguished from core functionality from a client point of view. From the client perspective, a KAON SERVER offers all functionalities.

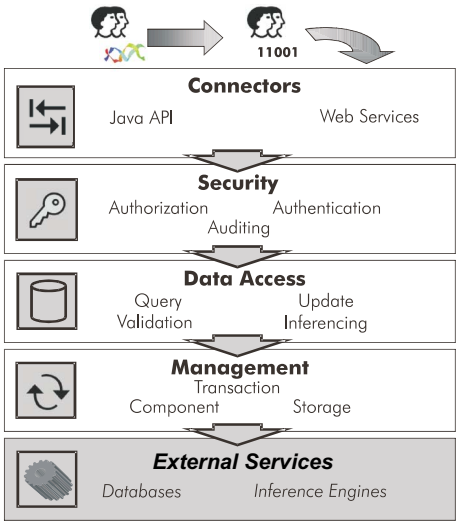

Figure 3.1: KAON SERVER Layers

3.3 Functional layers

We follow the well-known principle of layering functionalities such as known from the TCP/IP stack or from database architectures. The functional layers are depicted in Figure 3.1. Client requests are processed in a top down manner, first a request is made using some connector, then the properness of the request is checked by some security module. The request is encapsulated into a transaction

before being executed by delegation to the appropriate component, that is able to compute the request. Most of the time this will involve access to data. Eventually this is done by delegation of the request to a external service.

Connector Layer

Two APIs are provided to connect to a KAON SERVER: First, a Java API offers access to the hosted functionality. This API can be used to embed the KAON SERVER into a client application. Then, no remote calls are made to the KAON SERVER itself.

A Second API offers access to the server using some network protocol allowing to run the KAON SERVER as a separate process, which may be called by several clients using a network protocol.

Security Layer

The Security Layer introduces several interceptors which guarantee that operations offered by other components in the server are only available to appropriately authenticated and authorized clients. These interceptors could also provide auditing, i.e. the logging of clients' activities to log files. These log files may be further used, for example by a versioning or evolution component or to operationalize data roll-back.

Management Layer

This layer encapsulates all "basic" components commonly found in todays application servers. The transaction management system is responsible for ensuring the commonly known ACID transaction properties and wraps requests into transactional boundaries. Component Management is required to deal with the discovery, allocation and loading of functional components, that are eventually able to compute the request. Requests may also be replicated and coordinated between a set of components, if the required functionality can only be provided jointly.

Data Access Layer

Most requests will include some form of data access. Hence, this layer comprises of data-related functionality. This includes accessing and updating data. Additionally inferencing is logically situated in this layer since it manipulates data. However, we expect that inference itself may be provided by external services, to which appropriate interfaces must be made available. The integration of these services has to be seamless concerning the initial user request.

Additional data functionality provided by components located in this layer may be envisioned, e.g. syntactic and semantic validation of created data.

External Services

This layer comprises all external services that are used in a particular KAON SERVER configuration. These services are used via proxy components, which are managed components within the KAON SERVER and handle the communication and delegation of requests to the particular external service. If those proxy components implement functionality that allows management of the external service, the external service itself may become manageable through KAON SERVER, otherwise this is not possible. Examples for external services may be databases that are used for providing data persistence or inference engines, e.g. FaCT [5].

3.4 Functional components

The functional layers may be implemented by several functional components. Figure 3.2 lists all components that are required to be part of KAON SERVER due to the project proposal as well as several functional components that would complement the architecture and may possibly be delivered, i.e. the components of the security layer.

Since the KAON Framework is jointly developed within several projects, further yet unspecified functional components may be developed and non-WonderWeb

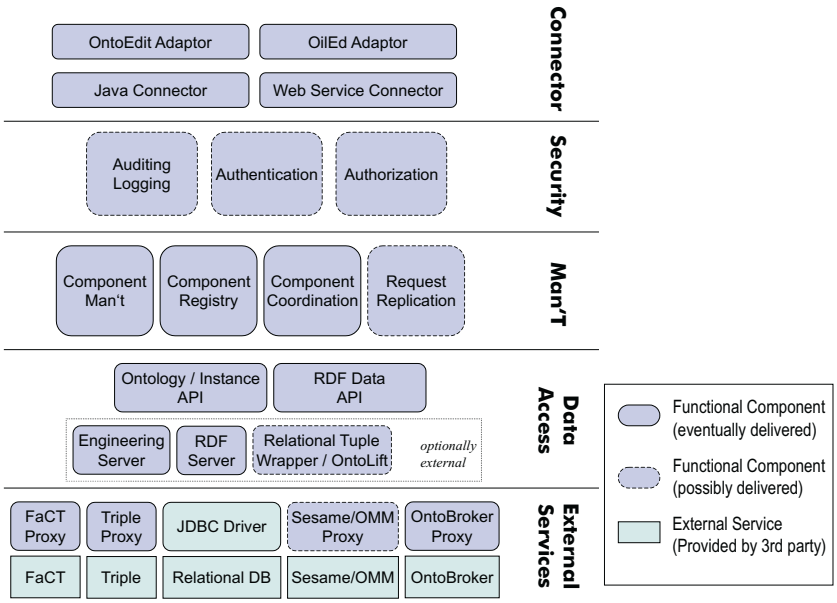

Figure 3.2: KAON SERVER Functional Components

requirements will be additionally considered, e.g. evolution aspects of ontologies from the EU-funded OntoLogging project. Also the responsibilities for realizing some functional components are delivered by project partners, e.g. the FaCT and OilEd clients.

The functional components which realize the aforementioned three core modules are discussed in detail in chapters 4 to 6. Here all components in the data and management layer are described further.

The remaining functional components are left unspecified in this deliverable, since separate deliverables will provide more detail on the requirements and objectives of these modules beyond the description given in the previous section. The interested reader is referred to the WonderWeb project proposal for a description of further requirements and a timelime on the realization of these components.

3.5 Technical Architecture

3.5.1 Component Management

Speaking in technical terms, extensibility is the strongest requirement for KAON SERVER. Therefore, the fundamental architecture for the server follows the well-established Microkernel design pattern [4]. This pattern basically separates minimal core functionality from extended functionality. Core to this approach is a so-called Microkernel[1], which only hosts the a minimal set of functional components. These components provide functionality

- required for bootstrapping the system itself
- allowing the dynamic extension of the system with further functionality.

In our setting, the Microkernel is realized by the components in the management layer. They allow to locate, instantiate, configure and restart further functional components dynamically, which is detailed in the following chapter.

3.5.2 Data Access and Data Persistence

Besides component management, further modules dealing with data access and persistence issues are likely to be part of virtually all KAON SERVER configurations. To maximize dissemination and re-usability, components belonging to those modules have been designed to be used separately on their own without other functional components and without requiring the component management infrastructure.

We can envision five abstract stand-alone usage scenarios for those components which lead to particular configurations:

Collaborative Editing of RDF Figure 3.3 illustrates this possiblity. Here, multiple clients edit a given RDF data set cooperatively, possibly requests are

[1] The pattern was adopted from a typical design approach for operating systems. Hence the term "kernel".

issued through the ontology API. An example application for this scenario would be the collaborative editing of RSS news.

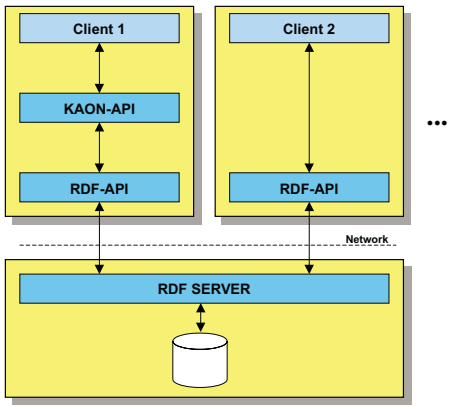

Figure 3.3: Collaborative Editing on RDF

Stand-alone Client Figure 3.4 illustrates this possibility. Here, the provided data access components are used to build a stand-alone application that does not involve any remote servers. An example application for this scenario would be a stand-alone ontology editor that operates on RDF files.

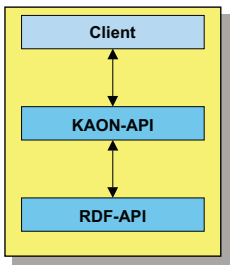

Figure 3.4: Stand-alone client

Persistent Web-application Figure 3.5 illustrates this scenario. Here, the provided data access components are used to access ontologies persisted in a database in a non-collaborative and two-tiered manner. For example, this sce-

nario could be applied to build web applications (since they provide request based processing of data).

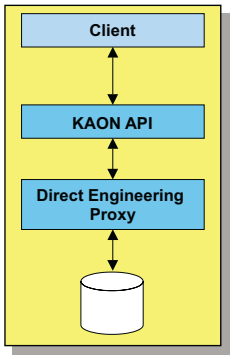

Figure 3.5: Persistent Web-application

Collaborative Engineering Figure 3.6 illustrates this possibility. Here, the provided data access components are used to collaboratively work on a shared server that hosts a given ontology. An example application for this scenario would be a cooperative ontology editor.

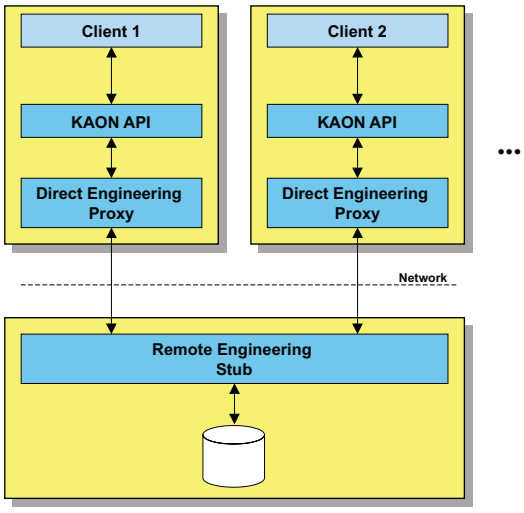

Figure 3.6: Collaborative Engineering

Shared Web and data server in particular for web applications, since the web application and the server can be deployed in the same JVM, and thus increase performance (since the remote call overhead is eliminated).

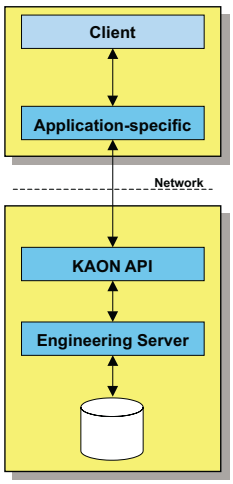

Figure 3.7: Shared Web

Chapter 4

Component Management

This chapter provides a detailed description of the Component Management infrastructure that is to be developed for KAON SERVER. We start with stating the goals for this module, resulting requirements and objectives and illustrate basic design decisions such as technology selection and usage scenarios.

4.1 Goals

Since extensibility is the strongest requirement for KAON SERVER, a component-oriented approach for the development of the server is chosen. The necessary component management must allow for

- *Reuse of functionality* from other, existing software,
- *Maintainability* due to the factorization of complexity into different components
- *Flexibility* due to the dynamic configuration of the infrastructure
- *Adherence to open standards* to benefit from external developments and maximum dissemination and use by third parties,

- *Understandability* due to factorization and by adhering to well-known software design patterns

4.2 Requirements

To achieve those goals a sophisticated component management infrastructure must support

- *component life cycle management*, this refers to (re)starting and stopping hosted components
- *run-time configuration*, it must be possible to (re)configure hosted components to avoid unnecessary starts and stops, which would otherwise lead to invocation overheads
- *component localization*, components and clients must be able to acquire references to components
- *component interaction*, components may rely on the existence of other components and interact with each other,
- *interceptable request*, this allows intercepting, queuing, or redirecting of requests and is needed for sharing generic functionality such as access control, logging, or concurrency control.

4.3 Objectives

Besides serving the requirements the component management infrastructure must facilitate adherence towards the following objectives:

- *Well-defined contracts* must ensure that components are accessed and managed in a unified and consistent manner
- *Component dependency management* that defines exactly how components interact with each other and the server.

- *User interface software* that allows developers to dynamically configure, maintain, and deploy components.
- *Minimal complexity*, the component management infrastructure should aim at minimizing the complexity of a component as seen by another component

4.4 Design

4.4.1 Technology Selection

Following the established requirements, a detailed research on readily available components that could suit our needs for the implementation has resulted in the following technology selection.

Component Management

The Java Dynamic Management Kit is a established solution for component, application and device management for the Java platform offered by Sun Microsystems. A Java technology specification has been derived from this commercial tool, called Java Management Extensions (JMX).

JMX is intended to provide tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications and service-driven networks. JMX defines a universal, open API for management, and monitoring. It has already been deployed across many commercial settings, where management or monitoring were needed and is therefore a promising platform for the construction of the required component management infrastructure.

Technically, the JMX specification defines instrumentation of components as MBeans, a so-called agent architecture and standard components. The contract for MBeans is simple, easy to implement, and unobtrusive for managed resources, making the adoption of JMX possible also for external services.

Additionally indirection and non-typed invocation make the JMX architecture resilient to changing requirements and evolving interfaces. Components constructed as MBeans may register or unregister from the server in accordance with their respective lifecycles, and their interfaces may evolve without having to disconnect the clients. The latter would allow for 24x7 uptime of the server.

JMX also enables the dynamic management of remote applications running on a variety of platforms, this could be used to manage external services. Hence, JMX is suitable for adopting existing systems by implementing new management and monitoring solutions on top of existing systems.

However, JMX only provides a specification, which may be implemented by vendors. Several freely and readily implementations exist, e.g. JBOSS MX[1]

Component Localization

Component localization may be realized by using existing or instantiation of naming and directory services. The latter play a vital role in today's distributed systems by providing network-wide sharing of various information about users, machines, networks, components, and applications. Hence, multiple servers could share components across the network.

For Java, the Java Naming and Directory Interface (JNDI) is an API that provides naming and directory functionality to applications written in Java.

Using JNDI, the implementation could build on functionality like the ability to store and retrieve named Java objects. For example, this may be used to store a given component temporarily to regain memory or for later use without startup overhead, i.e. unparsing of XML. In addition, JNDI provides methods for performing standard directory operations, such as associating attributes with components and searching for components using their attributes. This functionality may be extended by associating components with an ontology and using the ontology to search for suitable components. JNDI offers additional flexibility, since different naming and directory service providers may be seam-

[1] http://www.jboss.org

lessly integrated behind the provided common API. Thus, the implementation may build on different existing naming and directory services, such as LDAP, NDS, DNS, and NIS(YP), and would allow the server to coexist with further legacy applications and systems.

Component Management Framework

In order to take advantage of both component management and localization at the same time, one has to merge both technologies. For this purpose Hewlett-Packard already defined and implemented its so-called Core Service Framework (CSF) doing exactly this job[2].

This free-downloadable and -usable framework serves as a basis for the component management module and will be extended in several aspects. E.g. its component registry has to be changed in order to be Web-aware, dynamic instantiation of KAON-components must be added, localization may be ontology-enhanced and so on.

4.4.2 Interceptors

Intercepting, queuing, or redirecting requests becomes useful when a component is being restarted or reconfigured as part of its maintenance life-cycle.

Also, sharing generic functionality such as security, logging, or concurrency control lessens the work required to develop individual component implementations when realized via interceptors.

Therefore each component is internally registered with an invoker and a stack of interceptors that the request is passed through. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors toward the managed functional component.

For example, a logging interceptor could be inserted to implement auditing of operation requests. A security interceptor could be put in place to check that the requesting client has sufficient access rights for the component or one of its

[2] http://www.hpmiddleware.com

attributes or operations.

The invoker itself may additionally support the component lifecycle by controlling the entrance to the interceptor stack. When a component is being restarted, an invoker could block and queue incoming requests until the component is once again available (or the received requests time out), or redirect the incoming requests to another component that is able to service the request.

Readers familiar with the J2EE EJB specification may be aware that this functionality corresponds the EJB container contract which is usually implemented by setting a stack of interceptors in front of the EJB object. The request is passed through the interceptors that enforce the EJB specification functionality.

This is a specific instance of the more generic design outlined here.

4.5 Abstract usage scenarios

We can distinguish two abstract scenarios for the usage of the component management by clients[3]:

- The usage of available components
- The usage of components that are not available at the time of the request and have to be instantiated by the component management module

Several parties are involved to compute user requests in those scenarios:

- *Clients*, issue requests
- *Connector*, is the interface of the Server to the client
- *Component Registry*, Part of the module where all components are registered.
- *Component Management*, Part of the module that provides component management functionality.

[3] Please note, that any component could also be a client to another component as well

- *Functional component*, that provides the requested functionality
- *Invocator*, that proxies the functional component allowing interceptors to act in front of method invocations.

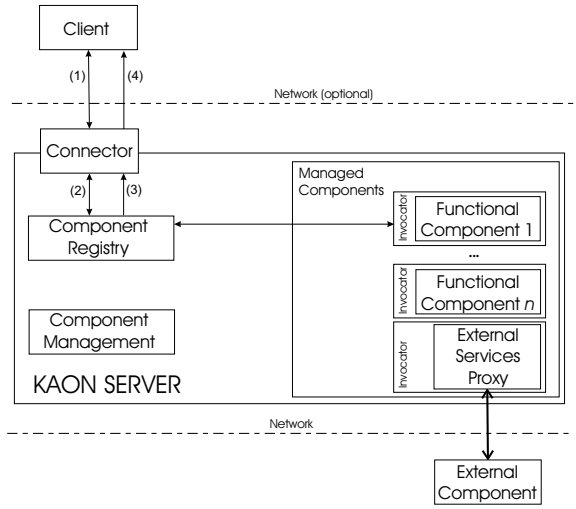

Figure 4.1: Abstract usage scenario with available functional component

Figure 4.1 illustrates the first scenario. Here, a client sends a request to the connector (1) which then accesses the component registry (2) to resolve a component available to compute the request. The reference to the invocator of the functional component is then returned to the Connector (3), which then returns the reference to the client (4). The client can now issue its request, which passes through the interceptors and should receive the appropriate response.

However, several additional steps have to be performed if a client requests a functionality that could be provided by some known functional component, but this component is not instantiated at the time of the request. This situation is illustrated in Figure 4.2. Here, the registry calls the component management functionality (A) to dynamically instantiate the appropriate functional component (B). The instantiated component is then available to the management component (C) and added to the registry (D). Notably, the instantiation of a

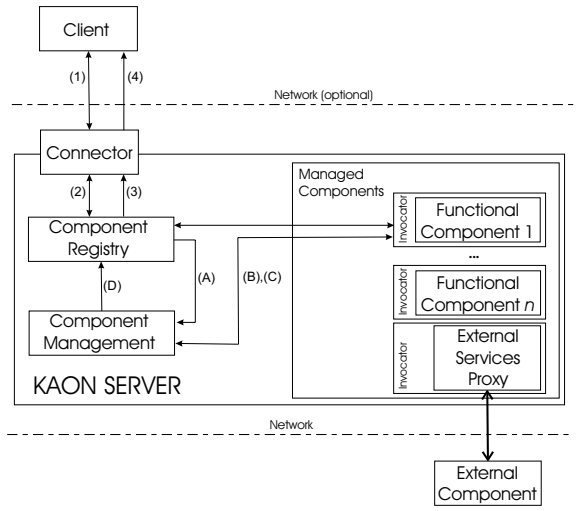

Figure 4.2: Abstract usage scenario without available functional component

component may lead to the cascading instantiation and registration of other components, if dependencies between components exist.

4.6 Additional benefits

- *Modular and maintainable software development*, since there should be no need to hard-code invocations to other components, rather a query should be made to the component registry. This removes "hard dependencies" which usually complicate maintenance.
- *Small memory footprint*, since you can configure the server to start only the necessary components required for your application. This avoids using the system resources by hosting a component no t needed.
- *Manageable setup*, since configuration and corresponding components could be loaded from across the network to the target system. This allows for a centralized configuration of a farm of servers and helps the system administrators to manage such setups.

Chapter 5

Data Access

This chapter discusses the module that provides access to data within KAON Server. We introduce two components that realize access to data. The first component allows access to RDF data. The second component allows access to ontologies and associated knowledge bases.

5.1 Goals

The data access module should primarily provide functionality that allows

- access to ontologies and knowledge bases, independent of the particular data representation, i.e. RDF,
- access to RDF data sets,
- multiple users to access data simultaneously in a safe and consistent manner,

Besides those goals, it is intended to meet the requirements stated for the KAON framework (see chapter 2).

5.2 Requirements

The achieve those goals the data access module must support

- *Data Modularity*, by allowing to compose distributed data,
- *Transactional changes*, by ensuring the commonly-known ACID properties,
- *Separation of concerns* by realizing separate interfaces for the manipulated entities,
- *Adhering to formal semantics* by being able to ensure that a given data set is valid,
- *Adhering to the intention of change* by ensuring that changes have the intended effect
- *Change notification* by providing mechanisms that notify other users of changes
- *Human-readable presentation* by providing a lexical layer on top on abstract URIs

5.3 Objectives

Additionally to the stated requirements the following objectives should be provided

- *Satisfying performance*, all operations should be performed in a satisfying time frame
- *Ability for undos*, all operations should be revokable.

5.4 Design

Our fundamental design decision was to separate concern on the level of accessing ontological and RDF data. Therefore two separate components are developed. The second design decision was to separate contract from implementation. Hence, two APIs are established, which could have multiple implementations. A particular implementation can offer to implement parts of the contract separately. For example, it could decide to implement modularization in a way that only allows to compose data sets which are hosted by a certain implementation.

5.4.1 Technology selection

Unfortunately no readily available components exist that fulfill the aforementioned requirements. The Jena project [10] provides both an ontology API and an RDF API, which are separate software entities. However, neither means for transactional processing, nor modularity nor consistent change operations are provided.

For RDF, a second API written (but no longer maintained) by Sergej Melnik at Stanford exists [12], however it fails to meet the requirements as well. To get started with an RDF access component, this implementation can be used as a guideline for crafting the interfaces used in our implementation.

5.4.2 RDF API

The RDF API is a set of interfaces that can be used to manipulate RDF models. Concern is separated from contract, thereby alternative implementations of the contract, i.e. the interfaces, are possible. Hence, it is possible to use the functionality provided by the data persistence module (cf. chapter 6).

Therefore clients of the RDF API component are isolated from the actual medium for RDF storage. However, the RDF API is primarily useful for in-memory manipulation of RDF models. For example, it can be used by the ontology API to read and access RDF-based ontologies.

In accordance with the requirements the interfaces must offer transactional manipulation of RDF models with the possiblity of modularization. Additionally, a streaming-mode RDF parser has to be provided to read from file and socket streams. Eventually, also a RDF serializer for writing RDF models is required. The implementation of the RDF API should be able to successfully read large RDF models like WordNet [14].

Further usability issues arise with RDF, which should be tackled by augmenting the RDF model. For example, URIs of RDF elements change when stated relatively to the base URI of the RDF model. For example, when an ontology is moved on the web or downloaded to the filesystems all relative identifiers change and make data operation nearly impossible since there is no way to tell that elements with different URIs are referring to the same thing.

The API should therefore differ between physical and a logical URIs of RDF models. The physical URI reflects the place from which the model has been loaded, e.g. the file system. While the logical URI is typically unique among all models. For example, a model may have the logical URI http://kaon.semanticweb.org/myModel.kaon, but may be stored at file:/c:/temp/model.kaon. We propose to generate unique resource names (URN) to ensure the uniqueness of logical URIs.

Object-Oriented Interfaces

The interfaces used to access and manipulate RDF data are generally very simple, since RDF itself is a simple data model. Hence, each RDF model is represented through the Model interface, which defines methods for model manipulation. A Model is viewed as a set of Statement objects. Each Statement has a subject, predicate and a object. Subject and predicate are Resource objects, while the object may be either a Resource of a Literal object.

Statement, Resource and Literal objects are immutable. Clients must use a dedicated Factory object to create statements, resources and literals. This factory ensures that there is at most one physical Statement, Resource or Literal

object with a given URI (singleton objects). By using this technique the memory consumption can be greatly minimized, however the fact that statements may have been multiply defined in the RDF source is lost.

Creating a Statement should not be equivalent to inserting it in the model. Rather statements are added to the model using the Model.add() method which has to be executed within transactional boundaries. Similarly, a statement may be removed from the model using Model.remove() method.

According to the requirements the RDF API supports modularization of models. Any RDF model can be included in any other RDF model. Including a model should not copy statements from the included model. Rather, it creates a virtual union.

This way requests, e.g. queries, on the outer model may be forwarded to included models. Each model can be represented as a managed component, hence the inclusion dependency could be expressed via component dependencies. This way it is possible to join models of different API implementations (for example, in-memory models may be joined with the RDF server-based models).

5.4.3 Ontology and Knowledge Base API

This component is realized by the KAON API. It isolates clients from different API implementations and provides a unified interface. It deals with objects representing various pieces of an ontology, such as Concepts, Relations, Attributes or Instances. Besides, there are objects for creating and applying changes to ontology entities as well as objects providing query facilities. The KAON API itself doesn't implement persistence, concurrency or security. Rather, it relies on lower layers to provide these features.

Implementations of the KAON API are responsible for providing consistency of the underlying ontology. Access to the API is performed through a dedicated evolution strategy whose purpose is to define and implement a set of change rules.

For example, when a concept is removed from an ontology, it must be decided

what to do with its subconcepts - they may be deleted, attached to the parent of the deleted concept or attached to ontology's root concept.

Several evolution strategies may be implemented for each of these policies, allowing the user to choose an appropriate policy when the ontology is instantiated. Finally, in order to improve performance, the KAON API allows for using a caching scheme. In that way many costly requests to remote data servers may be avoided and the overall application performance increased.

The Observable design pattern is used for notifications about model changes, thus achieving low coupling between model and associated views. All changes to the application model, whether local or remote, are propagated to registered listeners allowing them to display model updates immediately as they happen. The Java Messaging Service (JMS) is used to propagate change notifications in distributed environments. The API is entirely based on interfaces, allowing users to choose the appropriate implementation, depending on their needs.

In the following sections we discuss the ontology languages supported by the KAON API, its object-oriented interface and the different implementations of the KAON API.

Ontology Languages

RDFS (Resource Description Framework Schema) builds on top of RDF and is a language for defining light-weight ontologies in the Semantic Web. Its possibilities are rather basic allowing only the definition of classes, properties with their respective hierarchies as well as domains and ranges.

The Web Ontology Language (OWL) [11], which is currently developed by the W3C, will play a major role in the Semantic Web as standard ontology modeling language. OWL attempts to capture many of the commonly used features of DAML+OIL. It also adds functionality beyond RDFS in order to come up with a powerful language useful for semantic web applications.

OWL will consist of several layers starting with the so-called OWL Lite that attempts to choose features that do not impose too many restrictions on

toolbuilders. Compared to RDFS, OWL Lite additionally features

- Equality and Inequality
  - sameClassAs
  - samePropertyAs
  - sameIndividualAs
  - differentIndividualFrom
- Property characteristics
  - inverseOf
  - transitive
  - symmetric
  - functionality of properties
    (properties with min cardinality 0 and max cardinality 1. The same DAML+OIL side conditions hold that a transitive property (nor its superproperties) may not be declared functional.) [6]
- Datatypes
  Datatypes will be included. Thus, for example a range could be stated to be XSD:decimal. The exact details of this is dependent upon the RDF core group's decisions on datatypes for RDF.
- Additions
  - universal local range restrictions
  - existential local range restrictions
  - minimum and maximum cardinality

The KAON API will support OWL Lite after its standardization. In order to cope with the envisioned and more powerful OWL Full layer, additional inference components will be needed. In lack of standardization and current

dispute on the provided features, the KAON API currently realizes the ontology language described in [15]. The implementation adheres to the formal semantics given there. The language is based on RDF(S) and contains some proprietary extensions that may be available in OWL Lite. Additionally it provides:

- *Meta-modeling* a concept can be treated as an instance of some other meta-concept. Precise semantics is associated with such cases.
- *Lexical layer* lexical information about ontology entities is explicitly stored in the ontology and can be manipulated using the usual constructs.

Object-oriented Interface

Consistently with the terminology established in [15], ontologies and associated knowledge bases are referred to as OI-models (ontology-instance-model). The object-oriented representation consists of two objects representing an ontology and an instance pool, thereby separating concerns. We refer to an instance pool as being a set of instances and their actual relations[1].

An ontology consists of concepts (represented as Concept objects) and properties (represented as Property objects). A property has domain and range restrictions, as well as cardinality constraints. It may be an inverse of some other property, and may be marked as being transitive or symmetric. An instance pool consists of instances (represented through Instance objects) that may be linked with other instances. Each concept or property may have an instance associated with it through a spanning object. An UML-diagram of the API is shown in Figure 5.1.

Model Change

The object-oriented interface does not provide methods for performing changes. Instead Change is modelled by an event model. This allows to compile a list

[1] Since OI-Models allow means for modularization, the reflexive edge in the UML diagram (cf. [15]) is used. There will also be means for expressing equivalence between classes as well as properties because this will be required by OWL.

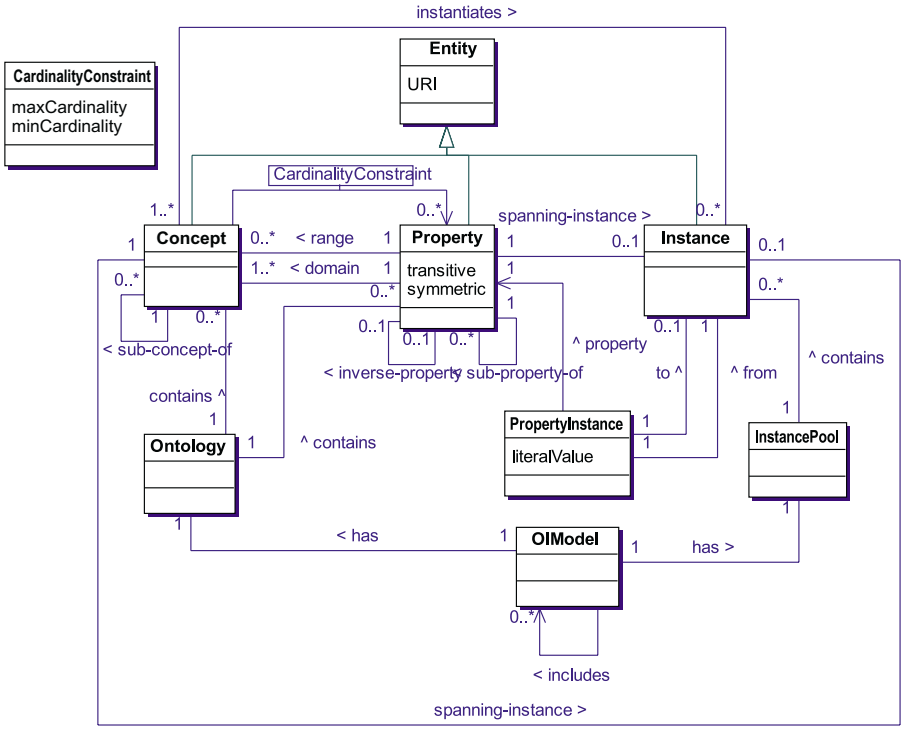

Figure 5.1: UML-View of KAON-API

of changes, which demark a natural transaction boundary, since changes should either be applied altogether and or none at all. Hereby Isolation and Atomicity of Changes are guaranteed. Additionally Changes and groups of changes can be revoked easily be performing compensating actions for each change in reverted order.

Single changes will often leave the model in an inconsistent state. Rather, additional changes are necessary to fix other parts of the model. For example, the user may request deleting a concept. To perform this operation, the respective concept must first be detached from its parents and children; children need to be reconnected to some other node etc.

Thus, a single user-initiated change may cause additional changes to be executed. Therefore the API realizes so-called Evolution Strategies whose task is to compute the additional changes required to be consistent with the intention of the change. The user may express her intention by choosing an appropriate strategy implementation.

The decision to model changes by sets of events instead of offering appropriate operations for manipulation has additional benefits of when accessing remote servers. Here, a set of change events can be packed into a list and be sent to the server all at once. This reduces network communication overhead. However, it requires local caching of information.

The Event-based design further enables evolution strategies that optimize updates. For example, elements are often moved in the ontology and attached to another place. If implemented strictly sequential, the move operation would first remove an element together with all ancillary data, and add the elements afterwards at the new location.

By representing all requested changes as objects, the evolution strategy knows in advance which changes need to be performed, and can prevent unnecessary deletion and later addition of information.

5.4.4 Implementations

RDF API

Selection of the implementation To allow the usage of several implementations, each implementation is separately registered with the component management module. Each implementation has to provide a factory object, that can be used to instantiate RDF models that use the particular implementation.

Additionally, implementations could be selected by assigning URI-prefixes to an appropriate implementation. E.g. http or file for instantiation of the in-memory implementation or database for the instantiation of a database implementation, etc. Then the component registry can be used to query a component that is able to handle URIs with a given prefix.

Proprietary extensions The RDF parser realizes proprietary extensions for model inclusion and assignment of logical and physical URIs by means of XML processing instructions. Note, that XML parsers are required to ignore unrecognized processing instructions. Hereby, serialized RDF models are guaranteed to be interoperable with other RDF parsers.

KAON API

Some implementations of the APIs are realized on top of the data persistence module which is responsible for common requirements, such as persistence, reliability, transaction and concurrency support.

Apart from providing abstractions for accessing ontologies, the KAON API also integrates different sources of ontology and data by offering different API implementations for various data sources. E.g. an implementation for relational databases or one working with RDF files by using the RDF API.

The following KAON API implementations may be used:

Implementation for RDF repository access An implementation of the KAON API based on the RDF API may be used for accessing RDF repositories.

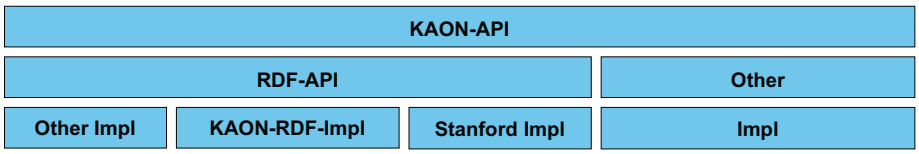

Figure 5.2: KAON API Implementations

The so-called Stanford implementation of the RDF-API [12] is primarily useful for accessing in-memory RDF models. However, it may be used for accessing any RDF repository for which RDF API implementations exist. KAON RDF SERVER is such a repository that enables persistence and management of RDF models and is described in more detail in chapter 6.

Implementation for accessing any database Another implementation of the KAON API may be used to lift existing databases to the ontology level. To achieve this, one must specify a set of mappings from some relational schema to the chosen ontology, according to principles described in [16]. E.g. it is possible to say that tuples of some relation make up a set of instances of some concept, and to map foreign key relationships into instance relationships. After translations are specified, an OI-model is generated. When accessed, the model will translate the request into native database queries, thus fulfilling most requests directly within the database itself. Similarly, the OI-model will translate ontology update requests to a series of updates to the underlying database. In such a way, the persistence of ontology information is obtained, while reusing well-known database mechanisms such as transactional integrity.

Implementation optimized for ontology engineering A separate implementation of the KAON API may be used for ontology engineering. This implementation provides efficient implementation of operations that are common during ontology engineering, such as concept adding and removal by applying transactions.

5.5 Additional benefits

The implementations of the KAON API fulfill the stated requirements. Additionally, we consider it suitable for enterprise-wide application because of the following reasons:

- Well-known technologies are used for ontology persistence and management, such as Enterprise JavaBeans (EJB)[2] and relational databases.
- The same technologies already realize many of the needed requirements: transactions, concurrent processing and data integrity come "for free".
- Because of the structure of conceptual models, a majority of requests is executed by the underlying databases, thus ensuring scalability in case of large information quantities. By choosing an appropriate API implementation it is possible to tune the system's performance for the given usage scenario.

[2] http://java.sun.com/products/ejb/

Chapter 6

Data Persistence

This chapter provides a detailed description of the data persistence module. We follow the established pattern of defining goals, deriving requirements and establishing further objectives. These are considered in the following discussion of our fundamental design. Along these lines an extensive survey of related systems is given.

6.1 Goals

The goal of the data persistence module is to provide a set of components that allow to store RDF and ontological data in order to achieve durability.

6.2 Requirements

The goal itself imposes few requirements, however the provided components must deal with requirements that arise from the basic design of the server (cf. chapter 3). Hence, it must enable

- Definition of a data basis
- Access operations for data

- Update operations for data
- Persistence of data
- Concurrent transactional access
- Data security, e.g. in case of a system crash, a consistent recovery of data is required
- Change notification mechanisms to provide collaboratively working users with information about changes made by others
- standalone deployability, to maximize dissemination and allow for usage scenarios, such as sketched in the basic architecture 3.

6.3 Objectives

Complementing the requirements we intend to

- simplify the installation of the data persistence modules as far as possible
- avoid requiring proprietary products by adhering to open standards

6.4 Design

To achieve the intended goal and fulfill the requirements, extensive use of available third party technologies, i.e. relational databases and application servers can be made and leads our design considerations. However, it is particularly important to acknowledge the objectives to maximize dissemination, i.e. through easy installation, and to avoid being bound to proprietary solutions.

We can follow fundamental architectural patterns for implementing data management solutions such as the ISO 5-layer architecture for database systems (cf. Figure 6.1).

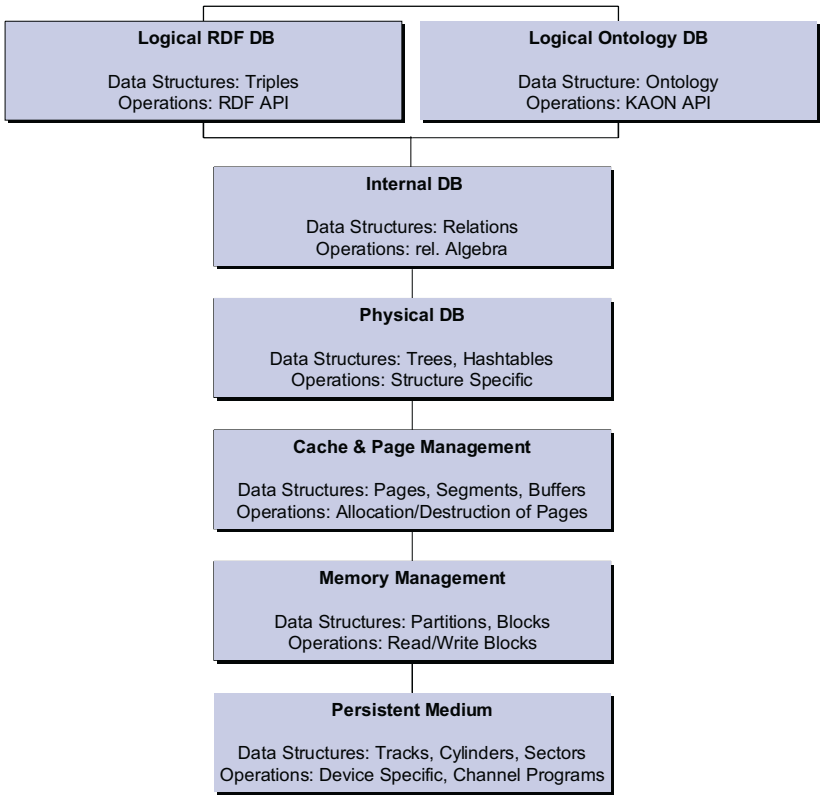

Figure 6.1: ISO database management architecture

Additionally the developed components should be able to be deployed as stand alone servers that are possible to use without other, e.g. RDF and ontology repositories.

6.4.1 Technology Selection

BerkeleyDB-based RDF databases

BerkeleyDB is a popular embedded database used in other products for data persistence. It provides transactions and a XA interface for transactional synchronization via transaction monitors. It provides data structures, i.e. B-Trees and Hashtables, and access methods for manipulating those structures. Hence,

it provides persistence functionality up to level 3 of the standard database architecture. Several RDF databases are built using BerkleyDB or similar products like DBM-variants.

RDFDB RDFDB[1] is the oldest RDF database and provides a simple, scalable (tested to 20 Mio. triples), open-source database for RDF data. It provides a textual query language in style of SQL to interface with the system. It may be used as a stand alone server and can be connected to via a socket interface. RDFDB provides limited and non-transactional updating of data. This restriction is a knock-out criteria due to the requirement of concurrent transactional access. It could be used for our purposes only by implementing transactions ourselves, however this appears to be more complex than implementing RDF data persistence.

Redland Redland[2] provides several RDF-related functionalities, among those are means for storing and querying RDF. It is developed at the University of Bristol. Storage mechanisms for models are provided via Berkeley DB, a query API is provided via Statement matching. Again, no transactional updates are possible.

RDFStore RDFStore[3] is a set of Perl modules targeted at RDF processing, including persistence using Berkley DB and DBM low-level database routines. It provides a query language which implements the SquishQL language. Additional features are RDF Schema inference on triples is provided and free-text searchs on literal values.

RDBMS-based RDF databases

The implementation of those databases relies on a full relational database to provide persistence. Hence, users are required to install a SQL-compatible database

[1] http://www.guha.com/rdfdb/
[2] http://www.redland.opensource.ac.uk/
[3] http://rdfstore.sourceforge.net/

to use the particular product. Conceptually the provided solutions operate on the fifth level of the standard DBMS architecture. Data access is realized by transformation of requests to appropriate SQL queries, which are then issued to the logical database level of the used relational database.

EOR - Extensible Open RDF Store The Extensible Open RDF store (EOR)[4] constitutes an open source project provided by the OCLC Office of Research and the Dublin Core Metadata Initiative. The current release provides services designed to validate RDF, to infuse RDF instance data into RDF databases. Querying is possible via triple-matching with wildcards.

EOR provides only basic, non-transactional update support. Hence, the same restrictions apply that were stated for RDFDB. An additional risk factor to relying on EOR is the fact, that the project appears to be extended rather sporadically leading to slow progress. The latest release is 15 months old.

RDFSuite RDFSuite[5] is provided by ICS-Forth, Greece with a suite of tools for RDF management, among those is the so-call RDF Schema Specific Database (RSSDB) that allows storing and querying RDF using a RDF Query Language (RQL) [17]. For the implementation of persistence an object-relational DBMS is exploited. It uses a storage scheme that has been optimized for querying instances of RDFS-based ontologies. The database structure is tuned towards a particular ontology structure. While this leads to a very efficient and scalable querying of instances, updates on the ontology lead to reorganization of the database structure and are necessarily non-transactional, since relational databases do not embed the removal and creation of relations into transactional boundaries. Hence, no transactions are provided and cannot be provided by custom implementation due to the initial design.

Sesame Sesame [7] is a RDF Schema-based Repository and querying facility developed by Aidministrator Nederland bv as part of the European IST project

[4] http://eor.dublincore.org/
[5] http://www.ics.forth.gr/proj/isst/RDF/

On-To-Knowledge[6]. The system provides a repository and query engine for RDF data and RDFS-based ontologies. It uses a variant of RQL that captures further functionality from RDF Schema specification when compared to the RDFSuite RQL language. Sesame shares its fundamental storage design with RDFSuite. Hence, the same restrictions apply that fail to meet our requirements.

Alternatively BerkeleyDB / RDBMS-based RDF databases

Two solutions abstract allow configuration of the used persistence solution and offer means to store data via SQL-based relational databases and the BerkeleyDB embedded database.

4Suite Server - Versa 4Suite Server is a platform for XML and RDF processing provided by Fourthought[7]. Among the provided tools is a RDF data repository. The server supports data access through a dedicated API and offers a query and inference language called Versa. It provides the data infrastructure of a full database management system, including transactions and concurrency support, access control and a variety of management tools. For purposes of integration with other tools, it supports remote, cross-platform and cross-language access through HTTP (including native SOAP and WebDAV), RPC FTP and CORBA exists. It offers APIs in Python, HTTP, SOAP and XSLT. It is the only component offering transactions. However, first tests of the software showed several problems. It was not able to process non-latin character sets (like UTF-16) and failed to import large amounts of RDF data. However, this appears to be the most promising option for our implementation.

Jena - RDQL Developed by the Hewlett-Packard Research, UK, Jena[8] is a collection of RDF tools including a persistent storage component and a RDF query language (RDQL[9]). For persistence the Berkley DB embedded database is

[6] EU-IST-1999-10132
[7] http://www.fourthought.com/4SuiteServer/
[8] http://www.hpl.hp.com/semweb/jena-top.html
[9] http://www.hpl.hp.com/semweb/rdql.htm

used. Alternatively any JDBC-compliant database may be used. Jena abstracts from storage in a similar way as our design. However no transactional updating facilities are provided. Additionally the scalability of the provided solution could not be verified. Processing large sets of RDF data (such as WordNet) lead to a system break-down when tested in April 2002. Again, the requirement for concurrent transactional access makes the solution less attractive for our purposes.

Other RDF databases

IntelliDimension RDF Gateway - RDFQL RDF Gateway is a distributed data semantic query service and inference infrastructure developed by Intellidimension Company. It comprises of two basic modules:

- *Data Services* that translate structured data into a knowledge base of RDF triples,
- *Query Service* that takes queries and inference rules, processed through a query Language called RDFQL on the knowledge base through a logic layer.

The Data Service module is designed to interface with arbitrary data sources, e.g. XML files, relational databases, or email accounts, which are mapped into RDF triples. Hence, it cannot be considered as a data repository, which provides a persistence component, but a dynamic knowledge representation framework, and fails to meet our purpose.

6.4.2 Persistence strategies

Looking at the provided solutions shows several possible design variants for implementing the persistence modules. Additionally, two further solutions are possible, which have not been considered for RDF databases before.

Obviously, we do not intend to operate on a operation system level, such as working on pages etc.

Reusing Physical Databases

The first realistic option for implementing persistence is translation of requests to access methods on the structure of the third, namely physical database layer.

Several example systems, i.e. the systems reported in section 6.4.1, use this approach. Hence, the prerequisite for the implementation would be the use of an embeddable database, such as BerkeleyDB.

While this solution allows for very fast processing and does not exclude transactions, several difficulties arise. First, we do not follow the standard architecture, which introduces a further layer, between the logical database and the actual physical data structures. Usually requests are made by users in a declarative query language, such as SQL, which is later translated into an executable program on the internal database, such as an expression in relational algebra. Query optimization techniques are applied during this translation. Hence, such optimization techniques, which are essential for complex query languages, may be more difficult to implement.

This also motivates why this solution was only taken by RDF databases, which do not consider ontologies (such as RDFS) at all. Due to the simplicity of the data model, the provided query languages are very simple. Hence, the implementation of complex operations on ontologies would be very time consuming.

Reusing Internal Databases

A second possibility is to allow for query optimization by relying on a more abstract data representation, such as provided by internal databases. Requests on the logical database level can be translated into operations on this level using established optimization techniques.

While this has not been done for any RDF database, the upcoming IBM DB2 version ("XPeranto")[10] offers such possibilities. Here a declarative query on XML and relational data formulated in the XQuery language [3] is translated

[10] http://xperanto.dfw.ibm.com/demo/

directly into the internal, executable query plan formulated in the DB2 Query Graph Model (QGM) - which is a patented variant of relational algebra.

Obviously, access to the internal representation of a database is required to implement this persistence strategy. While the implementation is tremendously simplified for complex operations on ontologies, the benefits for RDF would be rather low. Additionally a tremendous limitation is imposed by the fact, that the implementation is bound to a particular database such as DB2.

Reusing Logical Databases

This leads to a strategy used by several RDF databases, which consider RDFS in query processing, such as most systems presented in section 6.4.1.

Here logical data access operations on RDF and ontologies are served by exploiting the logical DB structure of a relational database. Hence, all requests are translated into declarative SQL queries.

Using SQL imposes the prerequistive of having to relay on an existing relational database. However, the dependency is no hard constraint, since SQL is an open standard, that is more or less followed across database vendors. Also, acceptable speed may be achieved, however it is tremendosly lower than the other databases.

Reusing XML databases

An alternative strategy could be to reuse existing XML databases such as Tamino[11] and reuse the fact that RDF has a XML-based syntax. Then RDF must be normalized into one of its many representations, stored as XML and requests may be transformed to an appropriate declarative XML query language that operates on the logical database.

However, we may not be able to gain large performance, since XML query languages usually rely on indexing structures for achieving efficiency. These indexing structures are usually created on an XML element level and not on an

[11] http://www.softwareag.com/tamino/

attribute level, which would be required for RDF (since objects are identified using rdf:id / rdf:about attributes).

This option has not yet been explored by existing systems.

Reusing Object-Relational Mapping Specification

Another alternative, which is actually outside of the standard database architecture and employed by modern application servers is to serve requests like invoking methods and accessing attributes of objects by automated translation of those requests to relational representations of objects, which have been created automatically as well.

Such services are offered by object-relational mapping services, e.g. Java Data e.g. Java Data Objects (JDO)[12], container-managed Enterprise Java Beans (EJB)[13] etc.

Using such services require existing relational databases, an appropriate EJB/JDO implementation and (for EJB) an application server.

Usually using such services has tremendous benefits like automatism, easy installation, portability across database vendors (due to specific implementations). For EJB this also provides the necessary remote interfaces and standalone deployability on J2EE-certified application servers.

However, a major restriction is largely imposed by the automatism itself. It is hard to control and hard to optimize. This imposes serious restrictions with simple data models such as RDF, where large quantities of information are represented in an uniform manner.

6.4.3 Storage Structures

Considering our requirements and objectives naturally leads to the third solution of implementing persistence via translation to the logical level of existing relational databases. Several alternatives for physically storing ontologies exist

[12] http://access1.sun.com/jdo/

[13] http://java.sun.com/products/ejb/

after choosing this particular persistence strategy. The physical organization of data is of central importance concerning the efficiency of data access methods.

However, there is no single, winning solution. Rather, the performance of a particular physical representation depends highly on the particular application setting. Hence, the below mentioned variants do not constitute exclusive alternatives but alternative proposals.

In the end, we envision to realize two separate persistence components with dedicated storage structures:

- RDF SERVER utilizing the Triple structure, and
- ENGINEERING SERVER utilizing the Metamodel structure

Per class and property structure

This is the optimal physical schema for instance manipulation and has been applied in several systems, such as RDFSuite and Sesame (cf. Figure 6.2). Here data is stored in a way, that two kinds of relations are created:

- unary relations for each class
- binary relations for each property

Given RDF data is then sorted into such tables. Apparently it is ideal for efficient querying of instance data. This is due to the fact that only few additional but unnecessary information is fetched from the physical data store by the relational database. The latter applies a technique called pre-fetching that loads additional blocks form the persistent medium when touching a particular storage area. This enhances performance due to the fact that swapping data in and out of memory can be minimized.

However, as discussed before, the fact that new relations have to be created for new classes and properties and relations are deleted when a class or property is no longer populated with instances prevents transactional processing if such operations are requested.

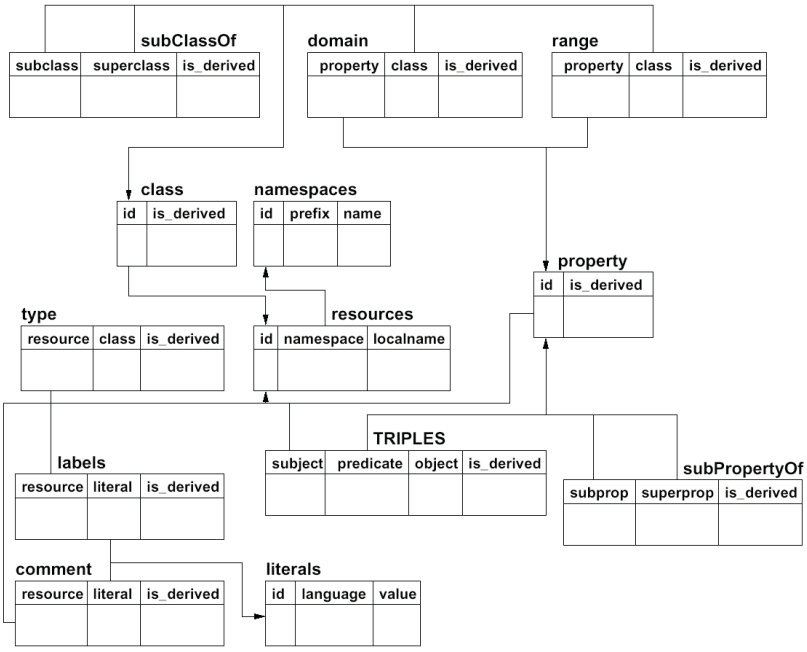

Figure 6.2: Per class and property storage

Per metaclass and -property structure

Alternatively, a storage structure that is based on storing information on a meta-model level is possible. Here a fixed set of relations is used, which corresponds to the structure of the used ontology language. Then individual concepts and properties are represented via tuples in the appropriate relation created for the respective meta-model element (cf. Figure 6.3).

This structure was not chosen before by any other RDF database, however it appears to be ideal for ontology engineering, where the number of instances (all represented in one table) is rather small, but the number of classes and properties dominate. Here, creation and deletion of classes and properties can be realized within transactional boundaries.

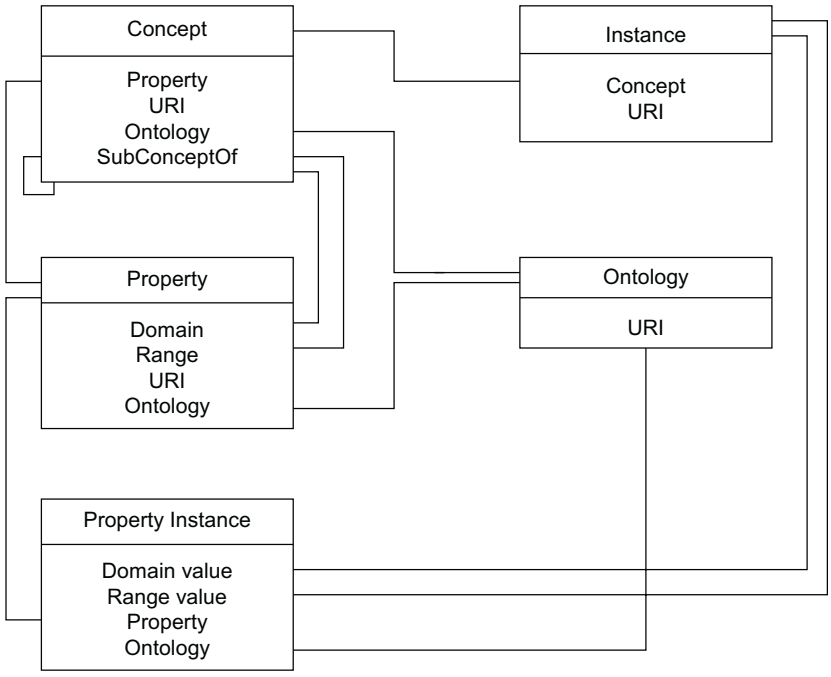

Figure 6.3: Per metaclass and -property storage

Triple structure

Another solution is to rely on a physical structure that corresponds to the RDF model. Hence, data is represented using minimally two relations, one represents models and the other one represents statements contained in the model (cf. Figure 6.4).

However, such simplistic structures are highly suboptimal for RDF, since usually a large number of joins are made to traverse object links. Hence indexing structures are needed on all elements of a statement and efficient data lookup is required. Usually this could be achieved using hashed indexes, which could provide lookup of a particular data element in constant time. However hashtables need a good distribution on keys for their values. This is not the fact for URIs (which have mostly similar prefixes) and also not for automatically incremented ids.

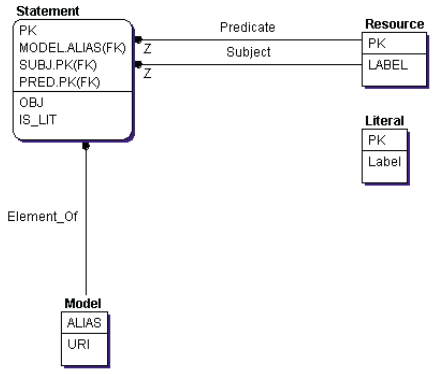

Figure 6.4: Triple structure

Consequently, this representation needs further relations to represent resources and literals and must provide an internal identifier that exhibits good indexing properties to provide efficient indexes on the statement relation.

Chapter 7

Conclusion

This deliverable presented KAON SERVER, the main organisational unit and infrastructure kernel of Workpackage 2. It will become part of the open-source Karlsruhe Semantic Web and Ontology Toolsuite (KAON). The server consists of the three modules Component Management, Data Access and the optional Data Persistence. Each module has been discussed in detail describing goals, requirements, objectives and design issues. Yet, the architecture presented in this deliverable has to be viewed as an initial design which will be evaluated and incrementally improved. The latest versions can be accessed via the Karlsruhe Semantic Web and Ontology Tool Suite (KAON) website http://kaon.semanticweb.org.

Appendix A

Glossary

Component: Managed software entity providing certain functionality.

Component Management: Module of the KAON SERVER concerned with initializing, creating, starting, stopping, monitoring of components.

Deployment: Process of registering a component to an appropriate management system.

Data Access: Module of the KAON SERVER consisting of APIs allowing access to ontologies and associated knowledge bases (i.e. KAON-API) as well as an API allowing access to RDF data (i.e. RDF-API). Both APIs make up the components of this module and may have multiple implementations (cf. Data Persistence).

Data Persistence: Module of the KAON SERVER consisting of implementations of the APIs defined for Data Access. There are two for the RDF-API, both a transient and a persistent called KAON RDF SERVER. The KAON-API, on the other hand, features an implementation on top of the RDF-API as well as the so-called Engineering server directly connecting to a database-management-system.

KAON RDF SERVER: Component belonging to the Data Persistence module. It implements the

RDF-API by utilizing a database-management-system instead of XML-files.

KAON SERVER: Consists of three core modules, viz. Data Access, Data Persistence and the optional Component Management. Each module consists of several components.

Module: Functional group of components.

Service: Independent software entity, non-managed, usually readily provided by a third party. Also referred to as External Service.

Bibliography

[1] R. Studer Y. Sure R. Volz A. Maedche, S. Staab, *Seal - tying up information integration and web site management by ontologies*, IEEE Data Engineering Bulletin (2002), 10–17.

[2] E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias, *Kaon - towards a large scale semantic web*, Proceedings of EC-Web 2002, Springer, 2002.

[3] J. Robie J. Simeon D. Chamberlin, D. Florescu and M. Stefanescu, *Xquery: A query language for xml*, Working Draft, W3C, June 2001.

[4] Hans Rohnert Peter Sommerlad Michael Stal Frank Buschmann, Regine Meunier, *Pattern-oriented software architecture, volume 1: A system of patterns*, vol. 1, John Wiley and Son Ltd, 1996.

[5] I. Horrocks, *The fact system*, Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98, Springer, 1998.

[6] U. Sattler I. Horrocks and S. Tobies, *Practical reasoning for description logics with functional restrictions, inverse and transitive roles, and role hierarchies*, Proceedings of the first workshop on Methods for Modalities (M4M-1), 1999.

[7] Frank van Harmelen Jeen Broekstra, Arjohn Kampman, *Sesame: A generic architecture for storing and querying rdf and rdf schema*, Proceedings International Semantic Web Conference 2002, Springer, 2002.

[8] R. Volz L. Stojanovic, N. Stojanovic, *Migrating data-intensive web sites into the semantic web*, ACM Symposium on Applied Computing SAC 2002, 2002.

[9] O. Lassila and R. Swick, *Resource description framework (rdf) model and syntax specification.*

[10] Brian McBride, *Jena: Implementing the rdf model and syntax specification*, Proceedings of SemWeb 2001, 2001.

[11] Deborah L. McGuinness, *Proposed compliance level 1 for webont's ontology language owl*, Knowledge Systems Laboratory, Stanford University.

[12] Sergej Melnik, *Rdf api*, Current revision 2001-01-19.

[13] Stefan Decker Michael Sintek, *Triple - an rdf query, inference, and transformation language*, In proceedings ISWC'2002, Springer, 2002.

[14] Richard Beckwith Christiane Fellbaum Derek Gross Miller, George A. and Katherine A. Miller, *Introduction to wordnet: An on-line lexical database*, International Journal of Lexicography **3** (1990), no. 4, 235–244.

[15] B. Motik, A. Maedche, and R. Volz, *A conceptual modeling approach for building semantics-driven enterprise applications*, Proceedings of the First International Conference on Ontologies, Databases and Application of Semantics (ODBASE-2002), November 2002.

[16] L. Stojanovic N. Stojanovic, R. Volz, *A reverse engineering approach for migrating data-intensive web sites to the semantic web*, IIP-2002, August 25-30, 2002, Montreal, Canada (Part of the IFIP World Computer Congress WCC2002).

[17] G. Karvounarakis D. Plexousakis S. Alexaki, V. Christophides, *On storing voluminous rdf descriptions: The case of web portal catalogs*, In Proceedings of the 4th International Workshop on the Web and Databases (WebDB'01) - In conjunction with ACM SIGMOD/PODS, Santa Barbara, CA, pages 43-48, May 24-25, 2001.

[18] C. Goble S. Bechhofer, I. Horrocks and R. Stevens, *Oiled: a reasonable ontology editor for the semantic web*, Proc. of the Joint German Austrian Conference on AI, number 2174 in Lecture Notes In Artificial Intelligence, pages 396-408, Springer, 2001.

[19] Dieter Fensel Rudi Studer Stefan Decker, Michael Erdmann, *Ontobroker: Ontology based access to distributed and semi-structured information*, DS-8, 1999, pp. 351–369.

[20] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic, *User-driven ontology evolution management*, Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW, October 2002.

[21] J. Angele S. Staab R. Studer D. Wenke Y. Sure, M. Erdmann, *Ontoedit: Collaborative ontology development for the semantic web*, Proceedings of the 1st International Semantic Web Conference (ISWC2002), June 9-12th, 2002, Sardinia, Italia, Springer, 2002.