Supporting application development in the Semantic Web

DANIEL ÖBERLE, STEFFEN STAAB, RUDI STUDER, RAPHAEL VÖLZ
University of Karlsruhe, Germany
Institute for Applied Computer Science and Formal Description Methods (AIFB)

The Semantic Web augments the current WWW by giving information a well-defined meaning, better enabling computers and people to work in cooperation. This is done by adding machine understandable content to web resources. Such added content is called metadata whose semantics is provided by referring to an ontology - a domain’s conceptualization agreed upon by a community. The Semantic Web relies on the complex interaction of several technologies involving ontologies. Therefore, sophisticated Semantic Web applications typically comprise more than one software module. Instead of coming up with proprietary solutions, developers should be able to rely on a generic infrastructure for application development in this context. We call such an infrastructure Application Server for the Semantic Web. The article discusses requirements and design issues of such a server. We also present our implementation KAON SERVER and demonstrate its usefulness by a detailed scenario.

Categories and Subject Descriptors: D.2.6 [Software Engineering]: Programming Environments; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Extensibility; D.2.11 [Software Engineering]: Software Architectures; H.3.5 [Information Storage and Retrieval]: Online Information Services—Web-based Services

General Terms: Design, Languages, Management
Additional Key Words and Phrases: Application Server, Extensibility, Interoperability, KAON, KAON SERVER, Ontology, Reuse, Semantic Web, WonderWeb

1. INTRODUCTION

The Internet was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help. One of the major obstacles to this is the fact that most information on the WWW is designed for human consumption, and even if it was derived from a database with well defined meanings (in at least some terms), the meaning of the data is not evident to a web application system[1].

The way out of this shortcoming is the Semantic Web which augments the current

---


Authors' addresses: University of Karlsruhe, Institute AIFB, 76128 Karlsruhe, Germany, email: [lastname]@aifb.uni-karlsruhe.de, http://www.aifb.uni-karlsruhe.de/WBS

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2003 ACM 1528-8785/00/0300/0147-0177 $5.00

WWW by giving information a well-defined meaning, better enabling computers and people to work in cooperation [Fensel et al. 2003]. This is done by adding machine understandable content to web resources. The result of this process are metadata, usually circumscribed as data about data, that can be a simple statement like “site x’s author is Daniel Oberle”. Descriptions like this are given their semantics by referring to an ontology, a domain’s conceptualization agreed upon by a community [Gruber 1993]. In the statement above, we could express that “Daniel Oberle” is a “PhD-Student” and “PhD-Student” is a specialization of “Graduate-Student”.

Ontologies serve various needs in the Semantic Web, like storage or exchange of data corresponding to an ontology, ontology-based reasoning or ontology-based navigation. Building a complex Semantic Web application, one may not rely on a single software module to deliver all these different services. The developer of such a system would rather want to easily combine different — preferably existing — software modules.

E.g. the domain ontology for a research and academia application can be easily expressed by Semantic Web languages and constructed by a corresponding editor (cf. Figure 1). There will be properties of concepts that require structured XML Schema data types [Biron and Malhotra 2001] whose correctness can be checked by a validator. A description logic reasoner is usually applied for semantic validation of the ontology. An ontology store saves the ontology and can be reused by a research and academia portal. The latter may exploit a rule-based inference engine that is capable of handling large amounts of instances and deduction of additional information by rules.

So far, such integration of ontology-based modules had to be done ad-hoc, generating a one-off endeavour, with little possibilities for re-use and future extensibility of individual modules or the overall system.

Fig. 1. Information flow in the research and academia example.

This paper is about an infrastructure that facilitates plug’n’play engineering of ontology-based modules and, thus, the development and maintenance of comprehensive Semantic Web applications, an infrastructure which we call Application...
Supporting application development in the Semantic Web

Server for the Semantic Web (ASSW). It facilitates re-use of existing modules, e.g. ontology stores, editors, and inference engines. It combines means to coordinate the information flow between such modules, to define dependencies, to broadcast events between different modules and to translate between ontology-based data formats.

The article is structured as follows: First, we provide a brief overview about the Semantic Web, in particular about all its languages, in Section 2. We list requirements for an Application Server for the Semantic Web in Section 3. Sections 4 and 5 describe the design decisions that immediately respond to important requirements, namely extensibility and discovery. The conceptual architecture is then provided in Section 6. Section 7 presents the KAON SERVER, a particular Application Server for the Semantic Web which is currently developed in the EU funded WonderWeb project. In Section 8 we will elaborate more on the scenario depicted in Figure 1, i.e. how it can be solved by making use of the KAON SERVER. Related work and conclusions are given in Sections 9 and 10, respectively.

2. THE SEMANTIC WEB

In this section we want to introduce the reader to the architecture and languages of the Semantic Web that we use in our Application Server. The left hand side of Figure 2 shows the static part of the Semantic Web, i.e. its language layers. Unicode, the URI and namespaces (NS) syntax and XML are used as a basis. XML’s role is limited to that of a syntax carrier for data exchange. XML Schema [Biron and Malhotra 2001] defines simple data types like string, date or integer.

![Fig. 2. Static and dynamic aspects of the Semantic Web layer cake](image)

The Resource Description Framework (RDF) may be used to make simple assertions about web resources or any other entity that can be named. A simple assertion is a statement that an entity has a property with a particular value, for example, that this article has a title property with value “Supporting application development in the Semantic Web”. RDF Schema extends RDF by class and property hierarchies that enable the creation of simple ontologies.


RDF and RDFS are already standardized by the World Wide Web Consortium (W3C) [Lassila and Swick 1999]. Figure 3 depicts an example for ontology-based metadata in the domain of research and academia. The ontology features a concept Person, along with specializations like Graduate-Student, PhD-Student as well as AcademicStaff and AssistantProfessor. RDFS’ modeling primitives formalize the domain description as RDF statements, e.g., “PhD-Student rdfs:subClassOf Graduate-Student”. CooperatesWith is a symmetric property defined on Person by using the rdfs:domain and rdfs:range primitives.

Fig. 3. Semantic Web example

XML serializations of RDF statements can be added to web resources like the homepages of PhD-Student “Daniel Oberle” and AssistantProfessor “Steffen Staab”. The metadata formally define both as instances of the ontology’s concepts through the rdf:type primitive. Relationships are provided with formal semantics by referring to the ontology. A search engine could later infer that also “Steffen Staab” cooperates with “Daniel Oberle”, if the property is defined to be symmetric.

The Ontology layer features OWL (Web Ontology Language [van Harmelen et al. 2003]) which is a family of richer ontology languages that augment RDF Schema and are based on the description logics paradigm [Baader et al. 2003]. OWL Lite is the simplest of these. It is a limited version of OWL Full enabling simple and efficient implementation. OWL DL is a richer subset of OWL Full for which reasoning is known to be decidable so complete reasoners may be constructed, though they will
be less efficient than an OWL Lite reasoner. OWL Full is the full ontology language which is however undecidable.

The Logic layer\(^4\) will provide an interoperable language for describing the sets of deductions one can make from a collection of data — how, given a ontology-based information base, one can derive new information from existing data.

The Proof language will provide a way of describing the steps taken to reach a conclusion from the facts. These proofs can then be passed around and verified, providing short cuts to new facts in the system without having each node conduct the deductions themselves.

The Semantic Web's vision is that once all these layers are in place, we will have an environment in which we can place trust that the data we are seeing, the deductions we are making, and the claims we are receiving have some value. The goal is to make a user's life easier by the aggregation and creation of new, trusted information over the Web.\(^4\) The standardization process has currently reached the Ontology layer, i.e. Logic, Proof and Trust layers aren't specified yet.

The right hand side of Figure 2 depicts the Semantic Web's dynamic aspects that apply to data across all layers. Often, the dynamic aspects are neglected by the Semantic Web community; however, from our point of view, they are an inevitable part for putting the Semantic Web into practice. Transactions and rollbacks of Semantic Web data operations should be possible, following the well-known ACID properties (atomicity, consistency, independence, durability) of Database Management Systems (DBMS). Evolution and versioning of ontologies are an important aspect, because ontologies usually are subject to change (cf. L. Stoianovic et al. [2002]). Like in all distributed environments, monitoring of data operations becomes necessary for security reasons. Finally, reasoning engines are to be applied for the deduction of additional facts\(^5\) as well as for semantic validation.

3. REQUIREMENTS

The requirements on an Application Server for the Semantic Web can be divided in four groups. First, such a server should respond to the static aspects of the Semantic Web layer cake. It has to be aware of all Semantic Web languages, in particular. The need to translate between the different languages also belongs to the static aspects. Translations would increase interoperability between existing software modules that mostly focus on one language only. Second, the dynamic aspects result in another group of requirements, viz. finding, accessing, modifying and storing of data, transactions and rollbacks, evolution and versioning, monitoring as well as inferencing and verification. Third, clients may want to connect remotely to the server and must be properly authorized. Hence, a distributed system like the Semantic Web needs connectivity and security. Finally, the system

\(^{4}\)A better description of this layer would be “Rule layer”, as the Ontology layer already features a logic calculus with reasoning capabilities. We here use the naming given by Tim Berners-Lee in his roadmap.


\(^{5}\)E.g. if “cooperatesWith” is defined as a symmetric property in OWL DL between persons, a reasoner should be able to deduce that B cooperatesWith A, given the fact that A cooperatesWith B.
is expected to facilitate an extensible and reconfigurable infrastructure. The last group of requirements therefore deals with flexible handling of modules. In the following paragraphs we will investigate the requirements.

Requirements stemming from the Semantic Web’s static part

—Language support A trivial requirement is the support of all the Semantic Web’s ontology and metadata standards. An Application Server for the Semantic Web has to be aware of RDF, RDFS, OWL as well as future languages that will be used to specify the logic, proof and trust layers.

—Semantic Interoperability We use the term semantic interoperability in the sense of translating between different ontology languages with different semantics. At the moment, several ontology languages populate the Semantic Web. We have already mentioned RDFS, OWL Lite, OWL DL and OWL Full apart from proprietary ones. Usually, ontology editors and stores focus on one particular language and are not able to work with others. Hence, an Application Server for the Semantic Web should allow to translate between different languages and semantics [Grosof et al. 2003; Bennett et al. 2002].

—Ontology Mapping In contrast to Semantic Interoperability, ontology mapping translates between different ontologies of the same language. Mapping may become necessary as web communities usually have their own ontology and could use Ontology Mapping to facilitate data exchange [Noy and Musen 2000; Handschuh et al. 2003].

—Ontology Modularization Modularization is an established principle in software engineering. It has to be considered also for ontology engineering as the development of large domain ontologies often includes the reuse of several existing ontologies. E.g., top-level ontologies might be used as a starting point. Hence, an Application Server for the Semantic Web should provide means to fulfill that requirement [Borgida and Serafini 2002; Maedche et al. 2003].

Requirements stemming from the Semantic Web’s dynamic part

—Finding, Accessing, Modifying and Storing of ontologies Semantic Web applications like editors or portals have to access, modify and finally store ontological data. In addition, the development of domain ontologies often requires other ontologies as starting point. Examples are Wordnet [Miller et al. 1990] or top-level ontologies for the Semantic Web [Oltramari et al. 2002]. Those could be stored and offered by the server to editors.

—Transactions and Rollback The dynamic aspects transactions and rollbacks (cf. Figure 2) lead to further requirements. All updates to the Semantic Web data must be done within transactions assuring the properties of atomicity, consistency, isolation (concurrency) and durability (ACID) [Ullman 1988].

—Atomicity of operations. An Application Server for the Semantic Web has to ensure the atomicity of each transaction. That means, a transaction is either run completely or not at all.

—Consistency. Consistency of information is a requirement in any application. 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 (cf. L. Stojanovic et al. [2002]). Modules updating ontologies must implement and adhere to these rules.

—Concurrency. It must be possible to concurrently access and modify Semantic Web data [Sure et al. 2002]. This may be achieved using transactional processing, where objects can be modified at most by one transaction at a time.

—Durability. Like consistency, durability is a requirement that holds in any data-intensive application area. It may be accomplished by reusing existing database technology.

—Evolution and Versioning Ontologies are applied in dynamic environments with changing application requirements (cf. L. Stojanovic et al. [2002]). To fulfill the changes, often the underlying ontology must be evolved as well. Database researchers distinguish between schema evolution and schema versioning [Noy and Klein 2002]. Schema evolution is the ability to change a schema of a populated database without loss of data (i.e. providing access to both old and new data through the new schema). Schema versioning is the ability to access all the data (both old and new) through different version interfaces. For ontologies, however, a distinction between evolution, which allows access to all data only through the newest schema, and versioning, which allows access to data through different versions of the schema, cannot be made. Hence, ontology evolution and versioning can be combined into a single concept defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology.

—Monitoring See “Connectivity and Security” below.

—Inference and Verification Reasoning engines are core components of semantics-based applications and can be used for several tasks like semantic validation and deduction. An Application Server for the Semantic Web should provide access to such engines, which can deliver the reasoning services required.

Connectivity and Security

—Connectivity An Application Server for the Semantic Web should enable loose coupling, allowing access through standard web protocols, as well as close coupling by embedding it into other applications. In other words, a client should be able to use the system locally and connect to it remotely via web services, for instance.

—Security Guaranteeing information security means protection against unauthorized disclosure, transfer, modification, or destruction, whether accidental or intentional. To realize it, any operation should only be accessible by properly authorized clients. Proper identity 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.

Flexible handling of modules

—Extensibility The need for extensibility applies to most software systems. Principles of software engineering avoid system changes when additional functionality
is needed in the future. Hence, extensibility is also desirable for an Application Server for the Semantic Web. In addition, such a server has to deal with the multitude of layers and data models in the Semantic Web that lead to a multitude of software modules, e.g. XML parsers or validators that support the XML Schema datatypes, RDF stores, tools that map relational databases to RDFS ontologies, ontology stores and OWL reasoners. Therefore, extensibility regarding new data APIs and corresponding software modules is an important requirement for such a server.

—**Discovery of software modules** For a client, there should be the possibility to state precisely what it wants to work with, e.g. an RDF store that holds a certain RDF model and allows for transactions. Hence, means for intelligent discovery of software modules are required. Based on a semantic description of the search target, the server should be able to discover what a client is looking for.

—**Dependencies** The server should allow to express dependencies between different software modules. For instance, that could be the setting up of event listeners between modules. Another example would be the management of a dependency like “module A is needed for module B”.

In the following Sections 4 to 6, we develop an architecture that is a result from the requirements put forward in this section. Thereafter we present the implementation details of our Application Server for the Semantic Web called KAON SERVER.

4. **COMPONENT MANAGEMENT**

Due to the requirement for extensibility, we decided to use the Microkernel design pattern. The pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core, i.e. the Microkernel, from extended functionality and application-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration [Buschmann et al. 1996].

In our setting, the Microkernel's minimal functionality offers simple management operations, i.e. starting, initializing, monitoring, combining and stopping of software modules as well as dispatching of messages between them. This approach requires software modules to conform to a management interface so that they can be managed by the Microkernel. Conformity is accomplished by **making existing software deployable**, i.e. bringing existing software into the particular infrastructure of the Application Server for the Semantic Web. This means that existing software is wrapped such that it can be managed by the Microkernel. Thus, a software module becomes a **deployed component**. The word **deployment** stems from service management and service oriented architectures where it is a term used by technicians [Bishop 2002]. We adopt and apply it in our setting. It describes the process of registering a component to the Microkernel with possible initialization and start.

Apart from the cost of making existing software deployable, the only drawback of this approach is that performance will suffer slightly in comparison to stand alone use, as a request has to pass through the Microkernel first (and possibly the network). A client that wants to make use of a deployed component's functionality talks to the Microkernel, which in turn dispatches requests.
But besides the drawbacks mentioned above, the Microkernel and component
approach delivers several benefits. By making existing functionality, like RDF
stores, inference engines etc., deployable, one is able to manage them in a centralized
infrastructure. As a result, we are able to deploy and undeploy components ad hoc,
reconfigure, monitor and possibly distribute them dynamically. Proxy components
can be developed for software that cannot be made deployable, e.g. because it has
been developed for a particular operating system. Throughout the paper, we will
show further advantages, among them
— enabling a client to discover the component it is in need of (cf. Section 5)
— definition of dependencies between components (cf. Section 6)
— easy integration of security aspects by interceptors (cf. Section 6)
— incorporation of quality criteria as attributes of a component in the registry (cf.
Section 10)

Thus, we have responded to the requirement for extensibility. In the following,
we discuss how the discovery of software modules is achieved.

5. COMPONENT DESCRIPTION
This section responds to the requirement “discovery of software modules”. As
pointed out in Section 4, all components are managed in a central infrastructure,
viz. the Microkernel. In order to allow a client to discover the components it needs,
we have to distinguish between them. Thus, there is a need for a registry that stores
descriptions of all deployed components. In this section we show how a description
of a component may look like. We start with the definition of a component and then
refine it. The definitions result in a taxonomy that is primarily used to facilitate
component discovery for the client.

Component. Software entity that is deployed to the Microkernel.

System Component. Component providing functionality for the Application
Server for the Semantic Web itself, e.g. a connector.

Functional Component. Component that is of interest to the client and can
be discovered. Ontology-related software modules become functional components
by making them deployable, e.g. RDF stores.

External Service. An external service cannot be deployed directly as it may be
programmed in a different language, live on a different computing platform, etc. It
equals a functional component from a client perspective. This is achieved by having
a proxy component deployed that relays communication to the external service.

Proxy Component. Special type of component that manages the communica-
tion to an external service. Examples are proxy components for inference engines,
like FaCT [Horrocks 1998].

Each component can have attributes like the name of the interface it implements,
connection parameters or several other low-level properties. Besides, we express
associations between components. Associations can be dependencies between com-
ponents, e.g. an ontology store component can rely on an RDF store for actual
storage, or event listeners. Associations will later be put in action by an associa-
tion management system component (cf. Section 6).
We formalize component taxonomy, attributes and associations in a management ontology outlined in Figure 4 and Table 1. The ontology formally defines which attributes a certain component may have and places components in a taxonomy. Eventually, functional components like KAON’s RDF Server and the Engineering Server (cf. Subsection 7.4) are instantiations of RDF Store and Ontology Store, respectively.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Property</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>receivingEventsFrom</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>sendingEventsTo</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>dependsOn</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Table 1. Attributes and associations of Component

The Microrkernel and component approach allows us to implement the registry itself as a component. As explained in Section 4, the Microrkernel manages any functionality if it conforms to the contract. The registry is not of direct interest to the client — it is only used to facilitate the discovery of functional components. Hence, it is an instance of system component.

Building on an ontology instead of a fixed data schema for component description allows us to retain flexibility and extensibility. Component providers can extend

---

6The table shows some exemplary properties of the concept “Component”. We use the term property as generalization for attribute and association. An attribute’s range is always string, whereas associations relate two concepts.

the ontology, for example by introducing new subconcepts or additional attributes stemming from similar approaches like DAML-S [Burstein et al. 2002].

So far we have discussed two requirements, viz. extensibility and discovery of software components, which led to fundamental design decisions. The next section continues with an overall view of the conceptual architecture.

6. Conceptual Architecture

When a client connects to the Application Server for the Semantic Web it needs to discover the required functional components. The system tries to find a deployed functional component in the registry fulfilling the client’s prescriptions and returns a reference.

From then on, the client can seamlessly work with the functional component. Similar to CORBA, a surrogate for the functional component on the client side can handle the communication over the network. The counterpart to the surrogate on the server side is a connector component. It maps requests to the Microkernel’s methods. All requests pass the Microkernel, which dispatches them to the appropriate functional component. While dispatching, the properness of a request can be checked by interceptors that may deal with authentication, authorization or auditing. An interceptor is a software entity which monitors a request and modifies it before the request is sent to the component. Finally, the response passes the Microkernel again and finds its way to the client through the connector.

After this brief procedural overview, the following paragraphs will explain the architecture depicted in Figure 5. Note that there will be only three types of software entities: components, interceptors and the Microkernel. Components are specialized into functional, system and proxy components to facilitate the discovery for the application developer.

Connectors. Connectors are system components. They send and receive requests and responses over the network. Aside from the option to connect locally, further possibilities exist for remote connection; e.g. ones that offer access via Java Remote Method Invocation (RMI), ones that offer access via Web Service protocols or ones that offer asynchronous communication. Counterparts to a connector on the client side are surrogates for functional components that relieve the application developer of the communication details similar to stubs in CORBA.

Management Core. The Management Core comprises the Microkernel (also called management kernel or simply kernel in the following) as well as several system components. The Management Core is required to deal with the discovery, allocation and loading of components. The registry, a system component, manages descriptions of the components and facilitates the discovery of a functional component for a client like explained in Section 5. Another system component called association management allows to express and manage relations between components. Event listeners can be put in charge so that a component A is notified when B issues an event or a component may only be undeployed if others don’t rely on it. The component loader facilitates the deployment process for a client, e.g. by

---

Fig. 5. Conceptual Architecture
adding a description of the newly created component to the registry. System components can be deployed and undeployed ad hoc, so extensibility is also given for the Management Core. Further components are possible, e.g., ones that replicate and coordinate requests between a set of components, if the required functionality can only be provided jointly, or a cascading component that offers seamless access to the components deployed in another instantiation of an Application Server for the Semantic Web.

**Interceptors.** Interceptors are software entities that monitor a request and modify it before the request is sent to the component. Security aspects are met by interceptors that guarantee that operations offered by functional components (including data update and query operations) in the server are only available to appropriately authenticated and authorized clients. Sharing generic functionality such as security, logging, or concurrency control requires less work than developing individual component implementations. E.g., when a component is being restarted, an interceptor can block and queue incoming requests until the component is available again. Transactions, modularization and evolution spanning several ontology stores may also be realized by interceptors.

**Functional Components.** RDF stores, ontology stores etc., are finally deployed to the management kernel as functional components (cf. Section 4). In combination with the component loader, the registry can start functional components dynamically on client requests.

<table>
<thead>
<tr>
<th>Requirement / Design Element</th>
<th>Connections</th>
<th>Kernel</th>
<th>Registry</th>
<th>Component</th>
<th>Association Management</th>
<th>Interceptors</th>
<th>Functional Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Support</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semantic Interoperability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Ontology Mapping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Ontology Modularization</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finding, Accessing, Storing of ontologies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transactions and Rollback</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Evolution and Versioning</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Monitoring</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Inference and Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectivity</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Extensibility</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Discovery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table II. Dependencies between requirements and architecture.

Table II shows where the requirements put forward in Section 3 are reflected in the architecture. Note that all requirements specific to the Semantic Web are met by functional components. That is because the conceptual architecture presented here is generic, i.e., we could make almost any existing software deployable and use the Application Server in any domain, not just in the Semantic Web. Apart from
security, every requirement met by an interceptor can also be met by functional components. E.g., transactions may be implemented by an ontology store. However, interceptors may be put it charge to implement transactions spanning several ontology stores, akin to what a transaction monitor does with several database systems. Modularization and evolution issues are similar. In the following section we discuss a particular implementation, KAON SERVER, that implements functional components specific for Semantic Web standards.

7. IMPLEMENTATION

This section presents our implementation of an Application Server for the Semantic Web, called KAON SERVER, which offers an infrastructure to host functional components, in particular ones provided by the KAON\textsuperscript{8} Tool suite [Bozsa\kak{} et al. 2002]. The latter includes tools allowing easy ontology creation and management, as well as building ontology-based applications in Java.

The KAON SERVER architecture reflects the conceptual architecture presented in the previous section. In the following, an in-depth description is given. We will start with the Management Core in 7.1 as it is necessary to understand Connectors in 7.2, Interceptors in 7.3 and Functional Components in 7.5. Several of the latter are implementations of the two Data APIS defined in the KAON Tool suite, which are discussed before in Subsection 7.4.

7.1 Management Core

The Management Core of an Application Server for the Semantic Web consists of the management kernel, the component loader, registry and association management system components. We will outline all of their implementations in the subsections below.

7.1.1 Kernel. In the case of the KAON SERVER, we use the Java Management Extensions (JMX [Lindfors and Fleury 2002]) as it is an open technology and currently the state-of-the-art for component management.

Java Management Extensions represent a universal, open technology for management and monitoring. By design, it is suitable for adapting legacy systems and implementing management solutions. Basically, JMX defines interfaces of managed beans, or MBeans for short, which are JavaBeans\textsuperscript{9} suited for management purposes. MBeans are hosted by an MBeanServer which allows their manipulation. All management operations performed on the MBeans are done through interfaces on the MBeanServer like depicted in Figure 6. We would like to point out two important methods of the MBeanServer:

\begin{verbatim}
registerMBean(Object object, ObjectName name)
\end{verbatim}

which, as the name suggests, registers an object as MBean to the MBeanServer; the object has to fulfill a certain contract implementing a prescribed interface, and

\begin{verbatim}
ObjectName invoke(ObjectName name, String operationName,
    Object[]params, String[] signature)
\end{verbatim}

\textsuperscript{8}Karlsruhe Ontology and Semantic Web Tool suite, http://kaon.semanticweb.org
\textsuperscript{9}http://java.sun.com/products/javabeans/

All method invocations are tunnelled through the MBeanServer to the actual MBean by this method. The corresponding MBean is specified by name, whereas operation-name, params and signature provide the rest of the information needed. Type checking has to be done by the developer and method calls are centralized. Hence, the architecture becomes resilient to changing requirements and evolving interfaces. Due to this technique, it becomes easy to incorporate the mechanism of interceptors (cf. Subsection 7.3).

Fig. 6. JMX Management Architecture

An MBean must be a concrete and public Java object with at least one public constructor. An MBean must have a statically typed Java interface that explicitly declares the management attributes and operations. The naming conventions used in the MBean interface closely follow the rules set by the JavaBeans component model. To expose the management attributes, one has to declare get and set methods, similar to JavaBean component properties. The MBeanServer uses introspection on the MBean class to determine which interfaces the class implements. In order to be recognized as a Standard MBean, a class x has to implement an interface javax.management.Attribute. Defining the methods get(x, attr) and set(x, attr) will automatically make attr a management attribute, in this case with read and write access. Only management attributes can be accessed and modified by a client. All the other public methods will be exposed as management operations. Each MBean is accessible by its identifying name that follows a special syntax.

JMX only provides a specification. There are several implementations available. For the KAON Server, we have chosen JBossMX, a JMX implementation which is part of the comprehensive application server JBoss. The reason for this decision is (a) it perfectly fits KAON SERVER requirements and (b) JBoss is open-source software.

In our setting, the MBeanServer implements the kernel and MBeans implement components. Speaking in terms of JMX, there is no difference between a system component and a functional component. Both are MBeans that are only distinguished by the registry.

<sup>http://www.jboss.org</sup>
7.1.2 Registry. The registry stores component specifications and constitutes a database about components that can be instantiated by the server. This information source is built around a management ontology, which specifies the functional aspects of a component, e.g., the libraries required by a component, its name, the class that implements the component itself etc. (cf. Section 5).

Building on an ontology instead of a fixed data schema, allows us to retain flexibility and extensibility. Component providers can locally extend the ontology, for example by introducing a new subcategory “InferenceEngine” to functional components. The use of expressive ontology languages allows to restrict globally defined component associations. E.g., an RDFStore may restrict the sendingEventsTo association to RDFEventListeners. The inference services offered by engines capable of dealing with expressive ontology languages additionally allow subsumption reasoning. Thus, it is possible to integrate multiple local extensions to the management ontology into a concise taxonomy.

We implemented the registry as MBean and re-used one of the KAON modules which have all been made deployable (cf. Subsection 7.5). The main-memory implementation of the KAON Application Programmer’s Interface (API) holds the management ontology. When a component is deployed, its description (usually stored in an XML file) is represented as instance of a concept. A client can use the KAON API’s query methods to discover the component it is in need of.

7.1.3 Association Management. The management ontology also allows to express associations between components. E.g., dependencies that state that a given component requires the existence of another component. Therefore, the server has to load all required components and be aware of the dependencies when unloading components. This essentially requires to maintain the number of clients to a component. A component can only be unloaded, if it does not have any further clients.

The JMX specification does not define any type of association management aspect for MBeans. That is the reason why we had to implement this functionality separately as another MBean. Apart from dependencies, it is able to register and manage event listeners between two MBeans A and B, so that B is notified whenever A issues an event.

7.1.4 Component Loader. The MBeanServer offers methods to deploy any MBean at runtime. However, the client application of an MBeanServer must explicitly create the MBeans it needs, it must maintain the list of required libraries and it must add newly created MBeans to the registry by itself.

To lift these responsibilities from the individual client, we have developed a special component loader MBean that facilitates the deployment process. MBeans are described by XML documents that contain RDF(S) serializations according to the management ontology mentioned in Section 5. The component loader uses this description to deploy the MBean, to add the MBean in the registry and to put associations into action by applying the association management. E.g., it deals with the transitive loading of required components. The component loader is able to deploy an MBean from arbitrary URLs, hence users of the server are not required to install any libraries on the server machine before instantiating a component. The
component loader also ensures that shared libraries which are part of the component implementation are only loaded once if multiple components share the same library.

7.2 Connectors

The KAON SERVER comes with four MBeans that handle communication. First, there is the HTTP Adapter from Sun that exposes all of the kernel's methods to a Web frontend. Second and third, we have developed Web Service (using the Simple Object Access Protocol) and RMI (Java Remote Method Invocation) connector MBeans. Both export the kernel's methods for remote access. Finally, a local connector embeds the KAON SERVER locally into the client application.

For the client there is a surrogate object called RemoteMBeanServer that implements the MBeanServer interface. It is the counterpart to one of the four connector MBeans mentioned above. Similar to stubs in CORBA, the application uses this object to interact with the MBeanServer and is relieved of all communication details. The developer can choose which of the four options (HTTP, RMI, Web Service, Local) shall be used by RemoteMBeanServer.

To facilitate all of the above for the client, we have built a ConnectorFactory the methods of which return surrogate objects for the registry, association management and component loader. In addition, we have developed surrogate objects also for functional components. E.g. there exists a RemoteRDFServer, relaying communication to one of the KAON tools (cf. Subsection 7.5). Every surrogate has to be provided with the MBean's identifying name which can be discovered in the registry.

7.3 Interceptors

As explained in Section 6, interceptors are software entities that monitor a request and modify it before the request is sent to the component.

In the kernel, each MBean can be registered with an invoker and a stack of interceptors. A request received from the client is then delegated to the invoker first before it is relayed to the MBean. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors towards the MBean. For example, a logging interceptor can be activated to implement auditing of operation requests. An authorization interceptor can be used to check that the requesting client has sufficient access rights for the MBean.

Apart from security, interceptors and interceptors are useful to achieve other goals. E.g., when a component is being restarted, an invoker could block and queue incoming requests until the component is available again or the received requests time out. Alternatively, it could redirect the incoming requests to another MBean which is able to fulfill them.

7.4 Data APIs

The functionality described so far, i.e. the Management Core, Connectors and Interceptors could be used in any domain not just in the Semantic Web. In the remaining subsections we want to highlight what makes the KAON SERVER particularly suitable for Semantic Web applications.

First, the KAON Tool suite has been made deployable. Two Semantic Web Data...
APIs for updates and queries are defined in the KAON Tool suite — an RDF API and an API for querying and updating ontologies and instances (KAON API). Their implementations are functional components that are discussed in Subsection 7.5. Furthermore, we are currently developing functional components that enable semantic interoperability of Semantic Web ontologies (cf. Section 3) as well as an ontology repository. Several proxy components have been developed for external services (inference engines in particular). All of them are discussed in the remaining Subsections. Before talking about the implementations and other functional components, the following paragraphs describe the APIs briefly.

**RDF API.** The RDF API consists of interfaces for the transactional manipulation of RDF models with the possibility of modularization, a streaming-mode RDF parser and an RDF serializer for writing RDF models. The API features object oriented representations of the entities defined in the RDF specification [Lassila and Swick 1999] as interfaces. An RDF model consists of a set of statements. In turn, each statement is represented as a triple (subject, predicate, object) with the elements either being resources or literals. The corresponding interfaces feature methods for querying and updating those entities, respectively.

**KAON API.** The KAON API currently realizes the ontology language described in [Motik et al. 2002]. It offers means for updating and querying an ontology and its instances. We have integrated means for ontology evolution, a transaction mechanism as well as ontology modularization. The interface offers access to KAON ontologies and contains classes such as Concept, Property and Instance. The API decouples the client from actual ontology storage mechanisms.

### 7.5 Functional Components

The KAON API is implemented in different ways like depicted in Figure 7. All of the implementations have been made deployable and are discussed subsequently in more detail. We also included descriptions of additional functional components, i.e. ontology repository, OntoLiFT, semantic interoperability and external services.

**RDF Mainmemory Implementation.** This implementation of the RDF API is primarily useful for accessing in-memory RDF models. That means, an RDF model is loaded into memory from an XML serialization on startup. After that, statements can be added, changed and deleted, all encapsulated in a transaction if preferred. Finally, the in-memory RDF model has to be serialized again in order to make changes persistent. The implementation allows for transactions and modularization of RDF models.

**RDF Server.** The RDF Server is an implementation of the RDF API that enables persistent storage and management of RDF models. It uses a relational database whose physical structure corresponds to the RDF model. Data is represented using four tables, one represents models and the other one represents statements contained in the model. The RDF Server uses a relational DBMS and relies on the JBoss Application Server\footnote{http://www.jboss.org} that handles the communication between client and DBMS. RDF Server also allows for transactions and modularization.

KAON API on RDF API. As depicted in Figure 7, implementations of the ontological KAON API may use implementations of the RDF API. E.g. the KAON API can be realized using the mainmemory implementation of the RDF API for transient access and modification of a KAON ontology (cf. also [Risch et al. 2003]). The implementation allows for transactions, modularization of OIModels as well as different evolution strategies.

Engineering Server. A separate implementation of the KAON API can be used for ontology engineering. This implementation, called Engineering Server [Motik et al. 2002], provides efficient implementation of operations that are common during ontology engineering, such as concept adding and removal. A storage structure that is based on storing information at a metamodel level is applied here. A fixed set of relations is used, which corresponds to the structure of the ontology language. Then individual concepts and properties are represented via tuples in the appropriate relation created for the respective meta-model element. This structure is suitable 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 is encapsulated within transactions. The Engineering Server also allows for modularization of OIModels as well as different evolution strategies (cf. L. Stojačević et al. [2002]).

Integration Engine. An additional implementation of the KAON API is currently under development that maps database schema 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 N. Stojačević et al. [2002]. E.g. it is possible to specify that tuples of some relation make up a set of instances of some concept, and to map foreign key relationships into instance relationships.

---

**Fig. 7.** KAON API Implementations

Ontology Repository. One optional component currently developed is an Ontology Repository [Maechele et al. 2003], allowing access and reuse of ontologies that are used throughout the Semantic Web, such as WordNet [Miller et al. 1990] or ontologies developed in the context of the WonderWeb project [Oltramari et al. 2002].

OntoLiFT. The OntoLiFT component leverages existing schema structures as a starting point for developing ontologies for the Semantic Web. Methods have been developed for deriving ontology structures for existing information systems, such as XML-DTD, XML-Schema, relational database schemata or UML specifications.
of object-oriented software systems. The LiFT tool semi-automatically extracts ontologies from such legacy resources.

**Semantic Interoperability.** A functional component already developed, realizes the OWL Lite language on top of a SQL-99 compliant database system [Grosf et al. 2003]. In addition, several others will later allow Semantic Interoperability between different types of ontology languages as a response to the requirement put forward in Section 3. In the introduction, we have already mentioned RDFS, OWL Lite, OWL DL and OWL Full. Besides, there are other languages, like DAML+OIL and also proprietary ones like KAON [Motik et al. 2002]. It should be possible to load KAON ontologies into other editors, like OntoEdit [Sure et al. 2002] or OiEd [Bechhofer et al. 2001], for instance. In general, information will be lost during transformation as the semantic expressiveness of the languages differ.

**External Services.** External services live outside the KAON SERVER. Proxy components are deployed and relay communication. Thus, from a client perspective, an external service cannot be distinguished from an actual functional component. Currently we are adapting several external services: Sesame [Broekstra et al. 2002], Ontobroker [Döcker et al. 1998] as well as a proxy component for DL classifiers that conform to the DIG interface\(^\text{12}\), like FaCT [Horrocks 1998] or Racer [Haarslev and Möller 2001].

8. **BUILDING A PORTAL FOR ACADEMIA AND RESEARCH**

This section shows the usefulness of an Application Server for the Semantic Web by a detailed scenario (cf. also [Staab et al. 2000]). The scenario shows the reader how the different parts of the Application Server, which so far have only been described in isolation from each other, interact.

We refer to the scenario depicted in Figure 1 that involved concise modelling of the research and academia domain in description logics. The ontology thus created can be used in several research and academia applications. In our scenario, we want to set up a comprehensive portal which exploits a rule-based system capable of handling large amounts of instances and deduction of additional information by rules. Basically, there are three types of rules:

1. **Schema integration** Rules that put concepts into a taxonomy, like *Graduate and AcademicStaff* are specializations of *Person*. Such constraints are resolved by subsumption reasoning in description logics.

2. **Rules involving instances and one concept** E.g. *If Person A cooperatesWith Person B then B also cooperatesWith A*. Description logics are capable of handling such rules in theory. However, no performant reasoner exists that can handle large amounts of instances.

3. **Rules that involve instances and several concepts** E.g. *If a Person A works in Project X and X's topic is Z, then Person A is familiar with the topic Z*. Decidable description logics are not able to express such rules\(^\text{13}\).

\(^\text{12}\)Description Logic Implementation Group, http://dl.kr.org/dig/

\(^\text{13}\)Also they don’t intend to, as they mostly focus on reasoning at the schema level.
In the following subsections, we want to show how the scenario can be solved with the KAON SERVER using existing clients and several components. The modeling of a domain ontology is initially independent of a particular application. It has to be as concise as possible and agreed upon by the community. The preferred choice for concise domain modeling are description logics, in our case OWL DL (cf. Section 2). Therefore one may use OIIEd [Bechhofer et al. 2001] for the construction of a domain ontology that takes a DOLCE [Oltramari et al. 2002] top-level ontology as starting point.

For the portal application, OntoEdit [Sure et al. 2002] and its corresponding ontology store Ontobroker [Decker et al. 1998] are well-suited because they are based on frame logics [Kifer et al. 1995] that, in contrast to OWL DL, allow the definition of and reasoning with rules of type (2) and (3).

We assume that an instance of the KAON SERVER is up and running, deployed with RMI and Web Service connectors, component loader, registry, association management as well as semantic interoperability, ontology repository functional components and proxy components for Ontobroker and FaCT [Horrocks 1998] (cf. Figure 8). The RDF Server will later be deployed by one of the editors.

![Diagram](image)

**Fig. 8.** An instance of KAON SERVER

OILed’s and OntoEdit’s interactions with the server are discussed by UML-like sequence diagrams [Booch et al. 1998] in the following. Note, that these diagrams do not show the exact Java method calls for the sake of brevity. For the same reason, we have omitted all the details involving connectors.

8.1 Modelling the Ontology

For ontology engineering we use OILed, an editor that supports the OWL DL language among others. It connects to the KAON SERVER through Java Remote Method Invocation (RMI). As depicted in Figure 9, OILed uses the ConnectorFactory to get surrogate objects for the MBeanServer itself, the component loader and the registry in the acquisition phase (1). What follows in step (2) is a successful discovery of the ontology repository functional component.

Interactions from surrogate objects (Remote*) to the KAON SERVER are not shown in the diagrams. Each surrogate translates its method calls to a connector’s
invoke() method which finally calls the MBeanServer’s invoke() on the server side.

A reference to the repository MBean is returned to OilEd which in turn loads a DOLCE top-level ontology as starting point for the domain ontology. This method invocation is directly routed through the MBeanServer without using a surrogate object. This is achieved by the invoke() method which takes an MBean reference, the name of the operation and its parameters as arguments (cf. Section 7.1).

After that, the editor looks up the MBean reference for the semantic interoperability functional component. OilEd uses it to transform the DOLCE ontology into the OWL DL language. This method invocation is also routed through the MBeanServer without any surrogate objects. At this point, the user is able to start editing the research and academia ontology (3). When finished, a verification on the ontology is usually done by applying the FaCT reasoner [Horrocks 1998]. OilEd tries to find such an inference engine in the registry. In our scenario, there is a proxy component deployed and thus a reference is returned. The editor creates a RemoteFaCT object that hides the communication details. In our case, the ontology is consistent, the user therefore proceeds with saving. For storing the ontology, an instance of KAON’s RDF Server along an authentication interceptor is created by using the component loader (4). OilEd is relieved from starting and initializing. It retrieves a reference to the newly created MBean from the component loader. Only then it is able to create an instance of RemoteRDFServer, which, like all other surrogates, hides the communication details as well as handling possible interceptors. For the latter, RemoteRDFServer has to be provided with the credentials first. After serializing the ontology into RDF, it is finally saved by the persistent RDF Server.

8.2 Definition of Rules

In the envisioned portal we want to apply reasoning based on logic programming [Das 1992] in order to deduce additional information. OWL DL does not allow the definition of rules, but we want to reuse the domain ontology. KAON SERVER’s capabilities for semantic interoperability allow the translation from OWL DL into frame logic and thus the usage of OntoEdit which provides a graphical user interface for editing ontologies and rules.

Figure 10 depicts the sequence diagram for OntoEdit’s communication with the server. RemoteMBeanServer and RemoteRegistry objects are created in phase (1), similar to OilEd’s interactions. We assume that the user is aware of the RDF Server and the ontology just created. He/she can provide enough information to perform a successful discovery for the store as well as the required credentials (2). An instance of RemoteRDFServer is responsible for communication and handling the authentication interceptor on the server side. Invocation of getOntology(...) on RemoteRDFServer delivers an RDF-stream which is to be transformed into frame logic, OntoEdit’s ontology language, by the semantic interoperability functional component. OntoEdit discovers the latter and calls the respective method directly, i.e. without creating any special surrogate object, through RemoteMBeanServer. The user is now able to add rules, instances and to perform adaptations on the ontology, as some information might have been lost during the translation from OWL DL (3).

OntoEdit uses Ontobroker for ontology storage and reasoning as well as semantic
validation of the ontology (analogous to OilEd and FaCT). Ontobroker exploits a relational database system for persistence. We have already assumed that a proxy component for Ontobroker is deployed to the KAON SERVER. Instead of loading a new one, OntoEdit tries to discover such a component and retrieves a reference to the respective MBean (4). Before loading the frame logic ontology into Ontobroker, the editor ensures that the proxy component is not unloaded by other clients, or due to server performance reasons. It therefore retrieves a reference to the association management via the registry and invokes a corresponding method. Frame logic ontology, instances and rules can now be loaded into Ontobroker.

8.3 Setting up the Portal

After translation into frame logic, possible adaptations and addition of rules with OntoEdit, the portal application just needs to reuse the deployed Ontobroker residing within the KAON SERVER. It already holds the required ontology together with the rules. The application has to connect to the KAON SERVER, in this scenario.
by a Web Service connector, discover Ontobroker and start displaying and changing
the ontology's instances by a web-fronted. Without the KAON SERVER, all of
the above would lead to a one-off effort of combining software modules without the
possibility for much reuse and extensibility.

9. RELATED WORK
We consider three distinct topic areas as related work. First, several systems ap-
proach some ideas relevant to an Application Server for the Semantic Web. Second,
onology development environments as well as knowledge base interoperability
approaches share some similarities to our approach. Finally, there is the work done
for middleware in general.

9.1 RDF Data Management Systems
All of the following data management systems focus on RDF(S) only. Hence, they
are not built with the aspect of extensibility in mind. However, they provide more
specialized components than our implementation does and offer more extensive
functionality with respect to RDF.

Sesame [Broekstra et al. 2002] is a scalable, modular architecture for persistent
storage and querying of RDF and RDF Schema. It supports two query languages
(RQL and RDQL), and can use main memory or PostgreSQL, MySQL and Oracle
9i databases for storage. The Sesame system has been successfully made deployable
as a functional component for RDF support in KAON SERVER.

RDFSuite [Alexaki et al. 2001] is a suite of tools for RDF management provided
by the ICS-FORTH institute, Greece. Among those tools is an RDF Schema specific
Database (RSSDB) that allows to query RDF using the RQL query language. The
implementation of the system exploits the PostgreSQL object-relational DBMS.
It uses a storage scheme that has been optimized for querying instances of RDFS-
based ontologies. The database content itself can only be updated in a batch manner
(dropping a database and uploading a file), hence it cannot cope with transactional
updates, such as KAON's RDF Server.

Developed by the Hewlett-Packard Research, UK, Jena [McBride 2001] is a col-
lection of Semantic Web tools including a persistent storage component, an RDF
query language (RDQL) and a DAML+OIL API. For persistence, the Berkeley DB
embedded database or any JDBC-compliant database may be used. Jena abstracts
from storage in a similar way as the KAON APIs. However, no transactional up-
dating facilities are provided.

9.2 Ontology Development Environments and Knowledge Base Interoperability
The Ontolingua ontology development environment [Fikes et al. 1997] provides a
suite of ontology authoring tools and a library of modular, reusable ontologies. The
tools in Ontolingua are oriented toward the authoring of ontologies by assembling
and extending ontologies obtained from a library. However, Ontolingua's tools do
not support the Semantic Web languages.

Stanford Research Institute's OKBC (Open Knowledge Base Connectivity) is a
protocol for accessing knowledge bases (KBS) stored in Knowledge Representation
Systems (KRSs) [Chaudhri et al. 1998]. The goal of OKBC is to serve as an
interface to many different KRSs, for example, an object-oriented database. OKBC

provides a set of operations for a generic interface to underlying KRSs. The interface layer separates an application from the idiosyncrasies of specific KRS software and enables the development of generic tools (e.g., graphical browsers and editors) that operate on many KRSs. OKBC abstracts from storage similar to the KAON API but is not limited to Semantic Web languages.

9.3 Middleware

Much research on middleware recently circles around so-called service oriented architectures (SOA)\(^\text{14}\), which are similar to our architecture, since functionality is broken into components — referred to as Web Services — and their localization is realized via a centralized replicating registry (UDDI)\(^\text{15}\). However, all components are stand-alone processes and are not manageable by a centralized kernel. The statements for SOAs also holds for previously proposed distributed object architectures with registries such as CORBA Trading Services\(^\text{16}\) or Jini\(^\text{17}\).

Several of today's application servers share our design of constructing a server instance via separately manageable components, e.g., the Hewlett Packard Application Server\(^\text{18}\) or JBoss\(^\text{19}\). Both have the Microkernel in common but follow their own architecture which is different from the one presented in our paper. JBoss wraps services like databases, Servlet and Enterprise JavaBeans containers or Java Messaging as components. Hewlett Packard applies its CSF (Core Services Framework) that provides registry, logging, security, loader, configuration facilities. It also relies on JMX but adds proprietary extensions. Neither JBoss nor the Hewlett Packard Application Server deliver ontology-based registries and association management. Both have not been designed for the support of Semantic Web nor are they suitable for the Semantic Web, in particular.

10. CONCLUSION

The article discussed the requirements, design and conceptual architecture of an Application Server for the Semantic Web. The article also presented a particular implementation — the KAON SERVER. The latter is part of the open-source Karlsruhe Ontology and Semantic Web Tool suite (KAON). From our perspective, the KAON SERVER is an important step in putting the Semantic Web into practice. Based on our experiences with building Semantic Web applications we conclude that such a system will be crucial for reuse and extensibility. This conclusion is consolidated by a detailed scenario that shows how the KAON SERVER can be used to facilitate the development of portal applications.

The KAON Tool suite is still work in progress. We are currently developing the aforementioned functional components like the ontology repository, semantic interoperability, the integration engine as well as OntoLiFT. The Web Service connector will be enhanced by semantic descriptions whose source is the registry. Interceptors

\(^{14}\text{http://archive.dewx.com/xml/articles/sm100001/sidebar1.asp}\)
\(^{15}\text{http://www.uddi.org/}\)
\(^{16}\text{http://www.omg.org/technology/documents/formal/tradingobjectservice.htm}\)
\(^{17}\text{http://www.jini.org}\)
\(^{18}\text{http://www.bluestone.com}\)
\(^{19}\text{http://www.jbossl.org}\)
for transactions, modularization and evolution across several ontology stores have to be developed.

In the future, we envision to integrate means for information quality — a field of research that deals with the specification and computation of quality criteria. Users will then be able to query information based on criteria like “fitness for use”, “meets information consumers needs”, or “previous user satisfaction” [Naumann 2002]. We will also support aggregated quality values, which can be composed of multiple criteria. Such criteria of a component will be reflected in the registry as attributes.

Acknowledgements. The KAON project is a meta-project carried out at the Institute AIFB, University of Karlsruhe and the Research Center for Information Technologies (FZI), Karlsruhe (cf. http://kaon.semanticweb.org). KAON consolidates the results obtained in several government-funded projects and fulfills requirements imposed by industry projects carried out at FZI. Individual software within KAON is developed within different projects.

The KAON SERVER is developed in the context of WonderWeb, an EU Information Society Technologies (IST) funded project (EU IST 2001-33052, http://wonderweb.semanticweb.org). Participants are the University of Manchester, the Vrije Universiteit Amsterdam, LADSEB-CNR, Padova, Italy and the University of Karlsruhe, Germany.

We would like to thank our colleagues, in particular Boris Motik, Alexander Maedche, Brian McBride and all WonderWeb researchers for fruitful discussions that helped to shape this contribution.

REFERENCES


