Leuphana - University of Lüneburg
Faculty III Environmental Sciences and Engineering

Semester 2008/2009
Master's Thesis

Design and Implementation of a Social, Semantic Agent

Author: Jörg Unbehauen
Mat. Nr.: 3003758
Arthur-Hoffmann-Strasse 47
04107 Leipzig
joerg@unbehauen.net

Examiner: Prof. Dr. Hinrich Bonin
Dipl. Inf. Sebastian Dietzold

Date of submission: February 7, 2009

Abstract

Instant Messaging is in addition to Web and Email the most popular service on the Internet. With xOperator we demonstrate the implementation of a strategy which deeply integrates Instant Messaging networks with the Semantic Web. The xOperator concept is based on the idea of creating an overlay network of collaborative information agents on top of social IM networks. Queries can be issued using a controlled and easily extensible language based on AIML templates. Such a deep integration of semantic technologies and Instant Messaging bears a number of advantages and benefits for users when compared to the separated use of Semantic Web technologies and IM, the most important ones being context awareness as well as provenance and trust. Our demonstration showcases how the xOperator approach naturally facilitates enterprise and personal information management as well as access to large scale heterogeneous information sources.

Contents

1 Introduction

This chapter gives a concise overview of the project itself. We describe the motivation for the creation of this project, in what kind of environment it was created and the methodological aspects of this paper are further laid out. Also an overview of the remaining chapters is given.

1.1 Project Mission

With the xOperator project we want to explore how the user can benefit from the integration of two technologies: Instant Messaging (IM) and the Semantic Web (SW). We demonstrate a collaborative method of querying the SW and show how this synergistic approach addresses some of the problems of the SW. In this document we illustrate the idea, design, implementation and evaluation of the resulting prototype called the xOperator.

The xOperator is not only a program that performs the tasks we later define in the use cases but it is also a open platform. This is not only achieved by simply releasing the source code to the public but also by opening the adaption of other use cases by a scripting envioment. We design the xOperator to be domain agnostic and flexible in adapting different scenarios.

1.2 Project Environment

The xOperator project is embedded into the Agile Knowledge Engineering and Semantic Web (AKSW) working group at the Department of Computer Sciences of the University of Leipzig. This working group with its head, Dr. Sören Auer, participated and created various high profile projects.

The development of DBpedia[1] described by Auer et al. (2007) as an effort to extract structured information from Wikipedia[2] is an example here. This project is about to become one of the crystallization points of an interlinked SW, a concept called Linked Data[3].

[1] http://dbpedia.org/About
[2] http://www.wikipedia.org/
[3] http://linkeddata.org/

Another project is for example OntoWiki[4], a semantic data wiki that is designed for distributed, agile knowledge creation. Here SW technologies support the user by introducing flexibility and reusability into the process Auer et al. (2006).

Triplify[5] is a lean method of making structured data as found in webapplication available to the SW where it can be reused, for example, to create mashups on top of SW technologies.

The xOperator project is made available under an open source license through the working groups site on `http://www.aksw.org`.

The project was created under the management of Sebastian Dietzold, who managed the development process and formed the main ideas of this project. The design and implementation was done by the author of this thesis, Jörg Unbehauen. The evaluation was assisted by members of the AKSW working group who also contributed to the implementation of some use cases.

1.3 Methodology

As we do not want to tackle a specific problem of the semantic web but want to answer the question of how the integration of IM and SW technologies can be useful in information retrieval, the methodlogical approach differs from typical engineering approaches in some parts. Rather than presenting a problem statement we describe the xOperator's basic ideas and map them against current problems found in the SW. Also the use cases that usually are derived from the users' wishes, as described by Avison & Fitzgerald (2006), were created by anticipating potential usage scenarios.

We selected an iterative approach for creating the agent. Although not committing to a strict methodological approach, we followed the ideas of a prototype based, iterative development process as described by Avison & Fitzgerald (2006). The main phases of each iteration are depicted in Figure 1.

[4]http://ontowiki.net/Projects/OntoWiki
[5]http://triplify.org/Overview

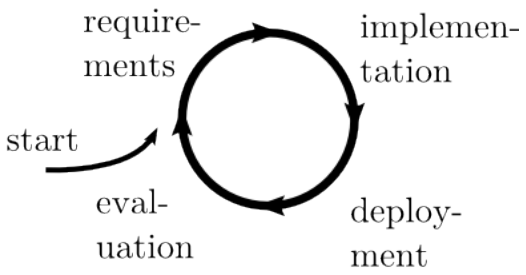

Figure 1: Iterative prototype implementation

We started the project with an initial brainstorming and discussion of the core ideas of the xOperator. Subsequently this a throw-away prototype was created and deployed. We decided that a further exploring of the idea would be necessary and based on the experiences made with the early prototype the requirements were refined and further functionality added and then implemented. This process was executed numerous times resulting in the mentioned early proof-of-concept, one major release (xOperator-0.1) and multiple prereleases. The next major release (xOperator-0.2) is as the time of writing, scheduled for a immidiate publication.

Feedback was generated by both the project members and the scientific community and is tracked via the Google code site of the xOperator [6] and is described in Chapter 8.4.

1.4 Structure of the Remaining Chapters

This document is structured into severals chapters representing the different steps taken to bring this project to a success. Starting with an introduction into the background technologies which are the Semantic Web and Instant Messaging in chapter 2, we present afterwards the core ideas of the xOperator in Chapter 3. In the subsequent chapter 4 we describe related work and competing concepts.

The successive chapters describe the engineering process in which the prototype was created. We start with the requirements definition in chapter

[6] http://code.google.com/p/xoperator/

5, where usage scenarios are created. In this Chapter we also identify the activities and the requirements in terms of functionality and quality.

Design, including architectural patterns and modularization is presented in chapter 6.

The implementation of this design is presented in chapter 7 where the modules that comprise the system are discussed.

In the evaluation chapter 8 we analyze how the xOperator matches the use cases, the performance is examined and the experience gathered while working and presenting the xOperator is shown.

The final chapter 9 concludes the results of this project and discusses the ways the xOperator will be developed further.

2 Background

In this Chapter we present a short intrdocution to the Semantic Web (SW) and Instant Mesaging (IM). These technologies play an integral part in the creation of the xOperator.

2.1 The Semantic Web

The web of today is the World Wide Web (WWW) which was created according to Conolly (2000) in 1989 at the CERN as a system to exchange results of scientific experiments. The creator, Tim Berners-Lee, used the concept of Hyperlinks to connect these results. To achieve this the Hypertext Markup Language (HTML), Hyper Text Transfer Protocol (HTTP) and Uniform Resource Locator (URL) technologies were created, standards that still form today's webs foundation.

With the growth of the web some limitations became more and more obvious. As described by Berners-Lee (1998) the web is mostly geared towards human consumption. Even structured information from a database is not understandable by computers as soon as it is presented in HTML. The drawback of this is illustrated by Antoniou & van Harmelen (2008). Web search engines like Google normally offer a high recall combined with a low

precision. They still depend on a human user to pick relevant matches from the display list. These applications are able to capture massive amounts of data, but as they are not able to understand its meaning, the queries can only be answered in an unprecise way.

The goal of the SW is to make the web more usable for both humans and machines. A description on how interaction using a machine-readable web could be like is shown in the visionary article of Berners-Lee et al. (2001). Here the authors describe how a human interacts with a personal agent which is able to precisely query data from a machine-readable web. This agent can therefore make for example an appointment negotiation over the web. While this is clearly a vision, the scenario demonstrates the potential of a machine-understandable web. In the same article Berners-Lee also explains that the SW is not a replacement for today's web but an extension of it. He describes the SW as a Web of Data inside the existing web.

The Semantic Web Coordination Group[7] of the World Wide Web Consortium plays a central role in developing the SW. It serves as an exchange platform for members of the scientific community and the industry and defines the most important standards of the SW.

The focus of the SW is on data integration. Applications like text processing tools or photo managing software keep their data in proprietary formats. The SW instead focuses in offering a platform for the integration of all kinds of data for maximum reusability. In order to achieve this without introducing monolithic or central data schemes we present here a set of tools and techniques geared towards decentralized data access. These can be arranged to a set of layers, each taking care of an aspect of the SW. This so called Layer Cake evolved during time, the latest iteration is depicted in Figure 2.1.

The bottommost layers define in which way data is represented in the Semantic Web. Besides pure serialization, Extensible Markup Language (XML) is presented here, this includes resource identification via Uniform Resource Identifier (URI) and Internationalized Resource Identifier (IRI) and resource description by means of the Resource Description Framework (RDF). These technologies are described in detail in Chapter 2.2. With these layers the

[7] http://www.w3.org/2001/sw/CG/

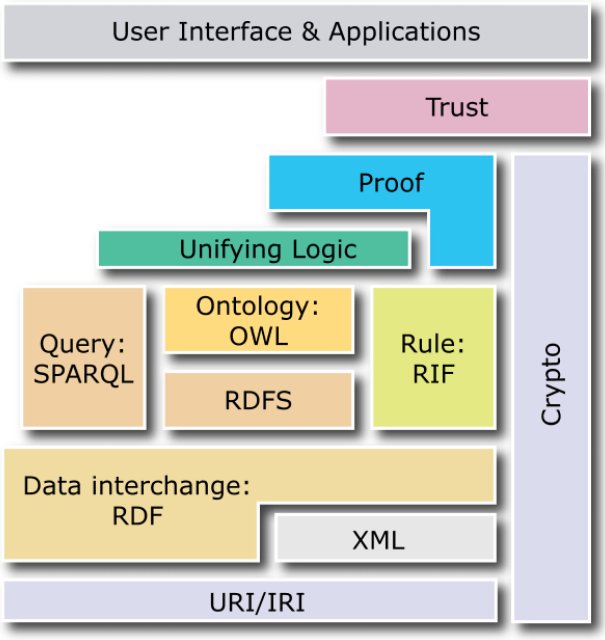

Figure 2: Layers of the Semantic Web according to Herman (2009)

foundation for a machine readable Web of Data is established.

Stacked on top of this are means of querying the data via the SPARQL Query Language for RDF (SPARQL). As this technology gets heavily used in the xOperator and is further described in Chapter 2.3.

Also building on top of RDF are means of modeling the data. RDF Schema (RDFS) allows the modeling of objects as classes and putting them into hierarchies. The Ontology Web Language (OWL) builds upon RDFS and allows a more sophisticated modeling of classes. These technologies get further described in Chapter 2.4.

The third technology that stacks directly on RDF is the Rule Interchange Format (RIF). Here rules in the sense of logic programming can be defined that further express constraints on relationships described in RDF. Herman (2008) describes the form of these rules to follow the *if* <*condition*> *then* <*consequence*> pattern and are base on Horn Logic. Rules allow expressions over tripels that are hard to define in for example OWL. Herman (2008) describes this using the example of merging multiple address books. Rules

can be made that express for example that upon the condition of an identical email address the entries consequently belong to the same person.

On top of these resides a logic layer that ties together the technologies of the layers below and provides declarative, application specific knowledge (Antoniou & van Harmelen 2008).

Via the trust layer an application is able to determine where information is coming from (provenance) and how confident it can be about its correctness (trust). Cryptography is employed here to ensure the authenticity of the information and its source.

On the very top of Figure 2.1 resides the application that makes use of the SW stack and provides functionality to a user or some other program.

According to Antoniou & van Harmelen (2008) two principles are to be followed whenever programs takes advantage of one of this layer. They should make full use of the layers that are below their own and should offer an at least partial understanding of the layers above. So by taking advantage of OWL also RDF as to be fully considered. The other way round means that when using RDF, OWL could be (partially) used.

The toolset introduced here is still under development. Because of the dependencies in the Semantic Web Layer Cake the technologies were standardized bottom up. While the technologies describing the bottom layers are published as standards or recommendations the upper layers are still under development.

2.2 Resources Description and Addressing

As Manola & Miller (2004) defines

the Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web.

By definition of the Semantic Web Coordination Group it is the default way of representing information on the SW.

In order to be able to express information about resources we have to first define the term and establish a way of identifying them. Antoniou

& van Harmelen (2008) describes resources to be *things of interests* like a person, book, website or concept. We use the URI or its internationalized form, the IRI, to address these things. In the case of a website this URI could be its URL. For a book it could be its ISBN or for a person its email address. While each of these identifiers has to be unique, a resource could have multiple identifiers, like a person identified by email address and its social security number.

For relating resources in RDF to each other a special type of resource is used, the property. Examples would be properties like *author-of* or *phone-number-of*. These special resources are, as any other resource, identified by a unique URI, enabling reuse of these properties.

Resources are related to each other by statements. A statement consists out of three elements: one subject, one predicate and one object and is therefore also called triple. The subject is always a resource, the predicate is always a property and the object is either a resource or a literal value. These two possibilities are presented in Figure 3.

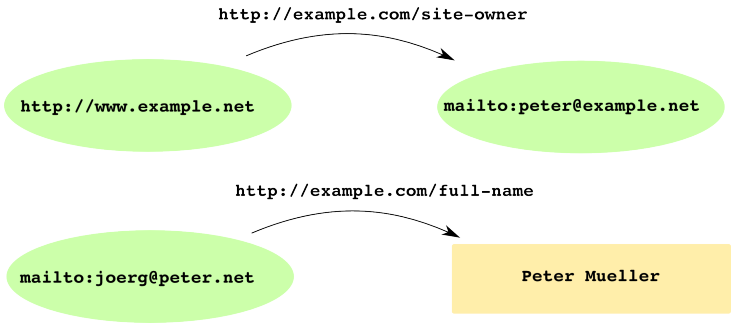

Figure 3: Graphical representation of two statements

The first statement describes the ownership of a web site. The subject, the resource `http://www.example.net` is connected via the predicate `http://example.com/site-owner` to the object, the resource identified by `mailto:peter@example.net`.

The second statement illustrates that the object can as well be a literal, here the resource `mailto:peter@example.net` is aligned by the property `http://example.com/full-name` to the literal `Peter Mueller`.

Statements in RDF construct a directed graph. A depiction of such a graph that includes the above mentioned examples can be found in Figure 4. As described in Chapter 2.3 this graph can be analyzed by using a query language.

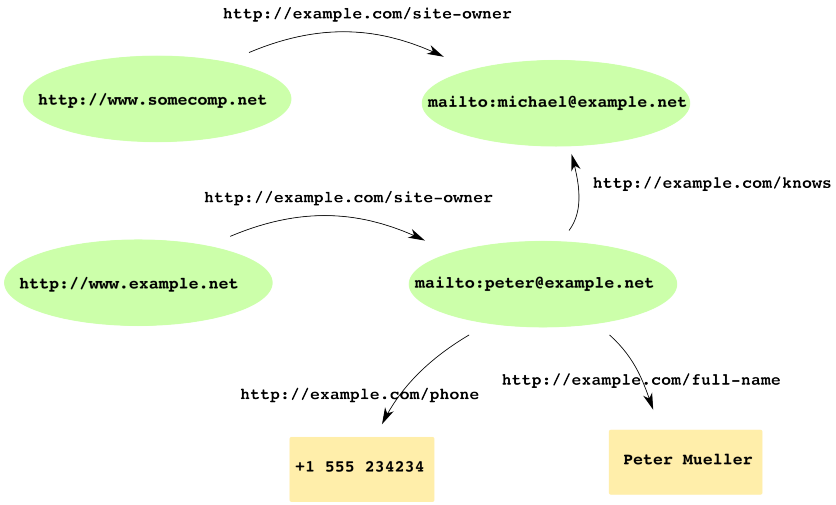

Figure 4: A graph of multiple Statements

2.3 Querying

In order to retrieve information from a graph query languages have been created. The SPARQL Protocol and RDF Query Language (SPARQL) has become the standard query language (see Prud'hommeaux & Seaborne 2008) and received the status of a recommendation by the W3C in January 2008.

The basic goal of SPARQL is finding triple patterns in a graph. Every arc of the graph and thus every triple is matched against these patterns. A pattern consists of a value for subject, predicate and object, the value can either be a literal, a resource identified by a URI or a variable. The query is executed in a SPARQL processor which is responsible for loading the graph into a triple store and finding the matching parts of the graph by traversing it.

A simple query that selects all owners of a web site described in the

graph of Figure 4 is shown in the following listing. We assume that an RDF representation of the graph is located at `http://example.com/graph.rdf`.

```
SELECT ?owner FROM <http://example.com/graph.rdf>
   WHERE {?site <http://example.com/site-owner> ?owner}
```

`SELECT` queries the graph for a table of results, in this case a single column identified as `?owner`. The `FROM` instructs the SPARQL processor to load this graph from this location and evaluate it against the triple patterns described in the `WHERE` clause. Here only a single pattern is defined. `?site <http://example.com/site-owner> ?owner` matches triples, that have an arbitrary subject (`?site`) and are connected via the property identified by `http://example.com/site-owner` to an arbitrary object (`?owner`). Whenever a matching triple is found, the variables are bound and the result is returned as a table.

```
-------------------------
-      ?owner           -
-------------------------
- <peter@example.com>   -
- <michael@example.com> -
-------------------------
```

This table is normally serialized in XML using the format defined by the W3C in SPARQL Query Results XML Format [8].

2.4 Ontologies

Ontologies play an integral role in the SW. Gruber (1992) defined that,

an ontology is a specification of a conceptualization.

Ontologies can thus also be seen as a vocabulary to describe a certain domain. So sharing an ontology enables others to use the same specification for description and create a common understanding. In this way numerous ontologies have been created and shared.

[8] http://www.w3.org/TR/rdf-sparql-XMLres/

The way ontologies are used in the SW does not impose a certain ontology on a domain. For the same domain multiple ontologies can co-exist. In order to ensure interoperability the technologies of the SW provide mechanisms to ensure compatibility.

The Semantic Web Coordination Group has standardized two technologies for creating ontologies. These are RDF Schema (RDFS)[9] and the Ontology Web Language (OWL)[10]. These were used to create for example the Friend Of A Friend (FOAF)[11] or the Semantically-Interlinked Online Communities (SIOC)[12] ontologies. These OWL ontologies belong to the most widespread used and define vocabularies describing personal information such as names, phone numbers or internet community related content, like which person contributed to a forum and the like. By using these vocabularies to describe data, users can achieve compatibility with other applications that make use of these or related ontologies. Ontologies use a namespace for identification. These namespace first identifies the ontology and second instead of writing the full URI of a concept defined there, it can be abbreviated. The namespace `http://xmlns.com/foaf/0.1/` that identifies the FOAF ontology has a concept person `http://xmlns.com/foaf/0.1/Person`, which can be for be abreviated to `foaf:Person`.

We further describe the concepts of ontologies by applying them to the example of Chapter 2.2 which results in the graph presented in Figure 5.

Ontologies use classes for describing concepts. In our example we use two classes `foaf:Agent` and `foaf:Person`. A `foaf:Agent` is described by Brickley & Miller (2004) as being something that is able to interact with some other thing. Its subclass `foaf:Person` is a specialized version of `foaf:Agent`. It represents a human being, while the latter could also be a computer program or similar.

These classes have instances similar to the concept of object oriented programming languages. The instance identified by `mailto:peter@example.com` is associated to a class with the `type` property. The phone number property

[9]http://www.w3.org/TR/rdf-schema/
[10]http://www.w3.org/TR/2004/REC-owl-features-20040210/
[11]http://xmlns.com/foaf/spec/
[12]http://rdfs.org/sioc/spec/

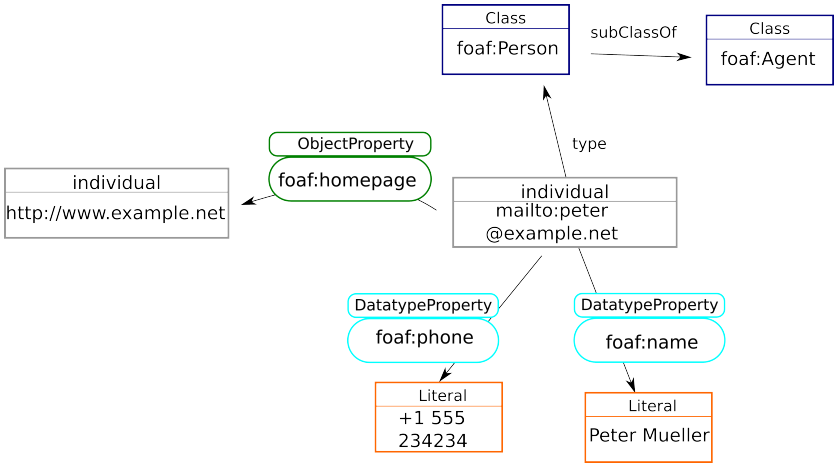

Figure 5: Graph using the FOAF ontology

of `mailto:peter@example.com` is using the FOAF ontology represented by a DatatypeProperty. This is a special form of a property and is also defined by OWL. It links an Individual to a literal value in this case the `foaf:phone` to the phone number literal and the `foaf:name` to the name literal.

The last concept that is important for the xOperator is ObjectProperty, which is similar to DatatypeProperty. It links two Individuals together, here the ObjectProperty `foaf:homepage` is used to indicate that a person has a homepage.

2.5 Instant Messaging

Instant Messaging (IM) is one of the most popular services on the internet. A quick survey on the most popular services indicates a massive user base, in detail:

Skype 370 million users (Skype Limited 2008)

Tencent QQ 800 million users (Tencent Holdings Limited 2009)

MSN Messenger 300 million users (Leskovec & Horvitz 2008)

The purpose of IM is to enable users to exchange short text messages synchronously over a network. While mainly geared towards private use it

is, as shown by Nardi et al. (2000), becoming more and more popular in corporate use. It allows quick information exchange in an informal way. A typical informal interaction is displayed in Figure 6.

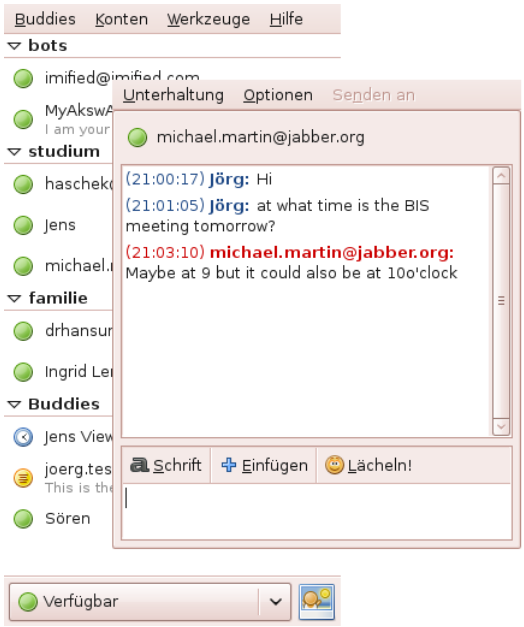

Figure 6: A short chat using the Pidgin IM client

One other important element of IM is that a list of friends is provided and maintained by the user. This so called roster shows the online presence of the user's peers and allows contacting them but it also allows a user to block certain entries. An example roster is shown in the background of Figure 6.

Clients that make it possible to participate in an IM network are available for virtually all platforms, ranging from implementation for mobiles to desktop computers. The transport of the text messages does not require a high bandwith so participation over a mobile network is possible.

2.6 The Extensible Message and Presence Protocol

The Jabber protocol was initially drafted in 1999, was later renamed to Extensible Message and Presence Protocol (XMPP) and is still under constant development. In contrast to the IM services mentioned in Chapter 2.5 it was

created as an open architecture with a publically available specification. The protocol has matured so far that it is published as a standard by the IETF[13] in Saint-Andre (2004). Here a core set of functionality is defined, furthermore the Jabber Foundation[14] maintains a large collection of extensions.

This IM network is based upon a client-server architecture, which is shown in Figure 7. Messages between the users are first sent by their clients to the server they are logged on to. Then these relayed to other servers, a mechanism similar to email delivery. Each user in the XMPP network is identified by an ID the Jabber Identifier (JID). It consists out of a username and the servername, aligned like an email address, an example JID is `user1@example.com`. Whenever users log into a server not only their availability is displayed in the roster of friends, but they are assigned a resource. This resource is appended to the JID, resulting in an address like `user1@example.com/Home` and allows addressing of a specific client.

XMPP defines two communication types. These are depicted in Figure 7 where also an exemplary infrastructure of the XMPP network is shown.

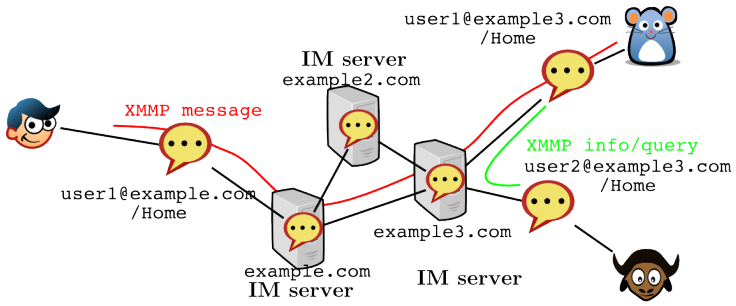

Figure 7: Communication in an XMPP network

Communication between users is of type `XMPP message`, indicated by the red line. It is used for transporting the chat messages between the users. With the type `XMPP info/query` the protocol defines a way for programs to exchange messages. In Figure 7 this is indicated by the green line. This form of message exchange allows for example a client to query the feature set of a

[13] http://www.ietf.org/
[14] http://xmpp.org/

server.

All transport between nodes of an XMPP network is encoded in XML. This allows an easy adoption of the protocol for own uses. Furthermore all transport between a server and a client can be and is usually encrypted.

3 Idea

With the design and implementation of the xOperator agent, the combination of the Semantic Web with Instant Messaging is under evaluation. We can extract two main concepts from this combination, first, using IM as a query interface, second, using the social network of the messaging system for enhanceing querying of the SW. These two ideas are presented and discussed in the following chapters.

3.1 Instant Messaging as a Query Interface

As we already described in the background chapter 2.5, IM is a well established, ubiquitous interface for informal communication. We want to explore how the user can use this interface for querying the SW in a similar manner.

As the way of communication on an IM network is via textual messages, a method for processing this input has to be found. Here especially the limitations of using a IM messaging client have to be considered. The first limitation is that communication is restricted to text as graphical elements cannot be displayed by the clients. Also, as these messages are pushed into the interface and instantly displayed, there is no way of for example cleaning the screening or mimicking a graphical interface, as done for example by the ncurses[15] library.

The solution for finding information in structured data like the SW is the use of some form of query language. In the chapter 2.3 we presented SPARQL as the most established way to do so. There are already existing approaches for this type conversion which are discussed in detail in chapter

[15]http://www.gnu.org/software/ncurses/

4, but as shown there, with the limitations of the IM technology these are not applicable to the xOperator.

So we want to create a method for associating natural language with the output of a SPARQL query. For making efficient use of the idea presented in Chapter 3.2 and for presenting the results in a user friendly form we decided to use small scripts to manage the processing. Scripts enable the user steer the querying process and perform operations that cannot be executed in SPARQL and transform the results in a form that can be displayed by an IM client in efficient way. The interface of the xOperator can thus be described as an mapping from natural language input to a script and uses only a minimal set of technologies that full grown natural language processing tools use.

3.2 Querying in a Social Network

The other central idea is that executing a query using the social context of the user can improve the results. This context is provided by the social network inherent in instant messaging networks.

3.2.1 Finding Information in Social Networks

Based on the fundamental research by Travers & Milgram (1969) on how humans are connected to each other Leskovec & Horvitz (2008) describes the graph that is modelling such a social network as being a small world graph. In his study, the author also points out some interesting properties of such a network. One is robustness, which is interesting as we want to use this network as an overlay network for communication. In result our overlay network preserves its basic structure upon removal of nodes so the unavailability of nodes does not cause the network to collapse. The other interesting aspects is the high clustering coefficient. This means that when a cluster is identified in this network these nodes are highly interconnected. This in combination with the findings of McPherson et al. (2001) about similarities between connected users makes it an interesting question how this connectedness and similarity can be exploited.

Social networks can also be found in IM networks. One way of analyzing

the social network is protocolling the flow of messages in the network, as done by Leskovec & Horvitz (2008). Another way is by examining the buddy lists of the users. These buddy lists, or rosters, contain the list of users that are able to see the users online status, personal information are able to communicate with him. The relations between the users can thus be organized to form a graph that describes the whole communication network.

These networks are not only found in these communication networks. In the last years more and more dedicated social networking sites like facebook [16] or LinkedIn[17] gained huge popularity Lampe et al. (2006). These websites allow users to search and find other users, most commonly based upon persons they already know, as shown by Lerman (2007). The author also shows that there are other social networking sites that are centred around specific types of content, like flickr[18] for photos or Digg[19] for bookmarking. These social media sites exploit the connectedness and similarity of their users in order to help users to find interesting content by analyzing their favorables and making recommendations.

The idea of the xOperator is to exploit the context provided by a social network in a more generic way. Instead of enriching information with social information, the whole information retrieval system is enriched with social data.

A real world example would look like this: Mike is searching the phone number of Peter, a fellow student he met on some social event. Unfortunately he does not really know who exactly Peter is, so he cannot look for the number in a directory like phone book. So what he does is asking all of his friends for the number of Peter and he finally gets two different numbers, of which the first one immediately connects him to Peter.

This scenario showcases that the precision of a query can be enhanced by executing it in a social context. If Mike would have asked a central directory he would have retrieved hundreds of phone numbers, by making some clever assumptions he might have been able to cut down the results to a some

[16] http://www.facebook.com
[17] http://www.linkedin.com
[18] http://flickr.com
[19] http://digg.com

dozens, but nevertheless too many to give them all a call. By using the social context he can retrieve the desired result in a reasonable amount of time.

3.2.2 Social Network as an Overlay Network

This human typical way of information retrieval can be adopted. We first create an infrastructure that allows a computer to execute a query in a social network. This structure is presented in Figure 8. In here three layers can be distinguished.

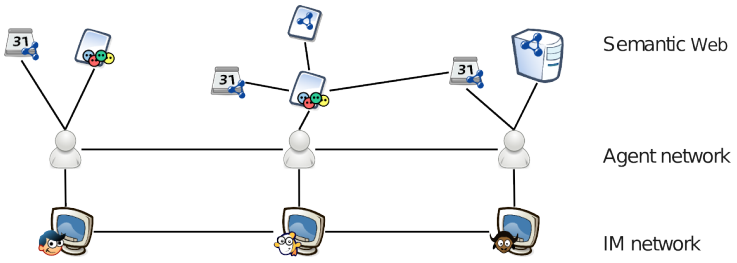

Figure 8: Relations between user, agent and resources

The bottommost layer represents the users of the system. They are connected to each other by means of their rosters and for example may chat with each other on a regular basis.

In the middle layer the agents can be found. They use the same network and the same accounts as their users. In an ideal case with everybody having an agent online, each agent can communicate with another agent for every contact in his users buddy list. All agents are equipped with facilities to answer queries from both the user and the peering agents.

On the upper layer the resources are represented. Each user assigns to his agent a number of resources they want to be published and that they trust. These resources form the knowledge that is queryable by its user and by the agents peering agents. Resources are for example FOAF profiles exported from a social networking site but also include SPARQL endpoints, like a semantic wiki used for personal information management.

On this infrastructure we are able to execute queries that mimic the human querying behavior as presented in Chapter 3.2.1. This querying process is described in Chapter 3.3.

3.3 Querying in the Instant Messaging Network

Combining the idea of receiving textual input via instant messenger and of querying on the infrastructure created in Chapter 3.2.2 results in a cooperative query answering process depicted in Figure 9.

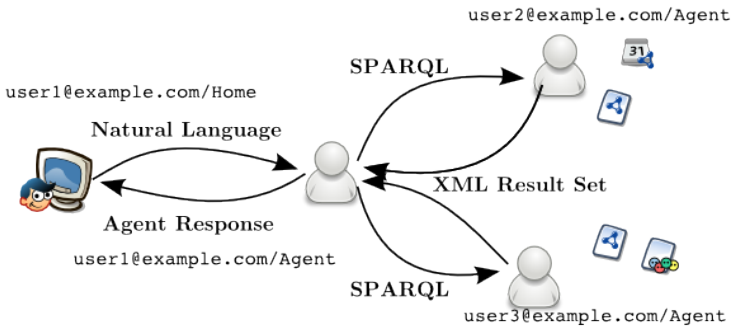

Figure 9: Executing a query

This query answering process is divided into five steps.

1. The process is initiated with the user asking the agent a question.
2. The question is mapped to a script.
3. Multiple queries are sent to neighboring agents by the script.
4. Answers to the questions are sent back to the agent.
5. A human readable answer is sent back to the user.

This models the exemplary human retrieval model presented in Chapter 3.2.1.

3.4 Trust and Privacy

By querying in a social context we can also implicitly address the problem of trust in the SW. As in the world wide web, the SW may as well contain contradictory information. Shadbolt et al. (2006) points out that determining the origin and propagation of data on the web is a complicated process. It important to know where certain data is coming from to be able to resolve those contradictions.

With the idea of equipping an agent with semantic resources the problem of trusting information is addressed, as the provenance of information is always known, as a answer to a query can always be associated via the agent to a user.

Privacy can both be improved and be endangered by allowing this kind of social querying. As shown in Gross & Acquisiti (2005), information gathered from social networking sites can endanger one's privacy and security. In the agent topology described earlier only directly connected agents are able to query each other, which results that an anonymous access on personal information published by the agent is blocked. This of course does not rule out dangers from authorized friends, in Chapter 9.2 we discuss how this can be addressed in the near future.

4 Related work

The idea of using natural language to query semantic data was the target of numerous research projects. One approach is using a controlled language, as for example described by Bernstein et al. (2005). The author showcases an approach of generating rules from ontologies. These rules convert questions posed in a controlled language into queries. These are then used to query this data from the ontology. As we want to be able to query peering agents we do not have any knowledge about the ontologies they use, so this approach will not work in the xOperator scenario.

A similar approach is described by Freese (2007). The author created AIML bots, a technology also employed in xOperator, by generating AIML

categories from ontologies. Here again, this approach cannot be used in our concept of querying, as again the ontologies are not known.

In Kaufmann & Bernstein (2007) the author compares various natural language interfaces (NLI) for querying semantic data. While good results are achieved and the interfaces do not require prior domain knowledge these tools rely on interaction with the user in order to clarify ambiguities. Again the interfaces available in IM are not suitable to provide these mechanism as no interactive menus can be presented.

These limitations led to the decision of using a simple, template matching approach.

Bringing semantic applications to the user's desktop and sharing this information is the aim of several more projects. Probably most prominent is the Nepomuk[20] project. Groza et al. (2007) introduces an architecture for semantic desktop applications and builds a framework for service oriented application integration and interaction. For example the Nepomuk-KDE[21] project demonstrates a deep integration into the K Desktop Environment, enabling the collection of data directly from the user. Projects related to Nepomuk, like Nepomuk-KDE or gnowsis[22], the reference implementation of the Nepomuk interfaces, are focused on the management of the user's data by using rich client data presentation. This in contrasts with xOperator since we use a minimal user interface.

The DBin[23] project from the Institute of Applied Informatics and Formal Description Methods of the University of Karlsruhe is, like the xOperator, a domain agnostic information sharing tool. It uses domain specific presentation widgets, although these rely on a Graphical User Interface (GUI) this is similar to the domain specific templates of the xOperator. A difference to the xOperator is that the data of DBin resides on servers and is managed in a newsgroup-like way. Provenance of the data is ensured in DBin by cryptographic signatures. It thus offers a distributed, extensible information sharing system, in contrast to the xOperator it does not make use of a social

[20]http://nepomuk.semanticdesktop.org
[21]http://nepomuk-kde.semanticdesktop.org/
[22]http://www.gnowsis.org/
[23]http://dbin.org/

network.

An approach to establish a network of trust in the SW is described by Golbeck et al. (2003). Here a web of trust is created by analyzing the social network of user and assigning a trust value to each person. The values ranging from 1 (absolute distrust) to 9 (absolute trust) are used to compute a trust value for the path information traverses trough this web of trust. Information is annotated by that with a trust value. While the xOperator does not explicitly addresses trust, the concept of sharing information on a social networks brings in an implicit trust system. This trust system however allows the user only to express trust or distrust. A further differneciation is not possbile. Also trust cannot be expressed beyond on edge in the social network.

Proposals and first prototypes which are closely related to xOperator and inspired its development are Dan Brickley's JQbus[24] and Chris Schmidt's SPARQL over XMPP[25]. However, both works are limited to the pure transportation of SPARQL queries over XMPP.

Quite different but nicely complementing the xOperator approach are works regarding the semantic annotation of IM messages. In Osterfeld et al. (2005) for example the authors present a semantic archive for XMPP instant messaging which facilitates searches in IM message archives. Franz & Staab (2005) suggests ways to make IM more semantics aware by facilitating the classification of IM messages, the exploitation of semantically represented context information and adding of semantic meta-data to messages. Comprehensive collaboration frameworks which include semantic annotations of messages and people, topics are, for example, CoAKTinG Shum et al. (2002) and Haystack Karger et al. (2005). The latter is a general purpose information management tool for end users and includes an instant messaging component, which allows to semantically annotate messages according to a unified abstraction for messaging on the Semantic WebQuan et al. (2003).

[24] `http://svn.foaf-project.org/foaftown/jqbus/intro.html`
[25] `http://crschmidt.net/semweb/sparqlxmpp`

5 Requirements

In this chapter we describe the requirements the system needs to fullfill. As earlier noted, no initial requirements or use cases were given, therefore an informal brainstorming with the project members as described by **??** was undertaken and use case scenarios were defined. These were then refined during the incremental progress of the project. Upon the description of the steps of each use case we identify common activities and common functionalities. Later these were used to create the functional building blocks of the application. During the iterations of the process we also refined how the application should interact with the user. We present these results as the quality attributes of the system.

5.1 Use cases

For creating the use cases we first selected an application domain. We focused on two domains, first knowledge modeled in the FOAF ontology representing personal information and second data from the DBpedia project which is a broad, multidomain knowledge base.

The use cases are presented here in form of tables. A use case is identified by a sample input. This sample input stands exemplary for all the possible variations of the input. The short description sums up the activities of the long description. Further, the template, which is used for matching the input is named. In this template the parameters that are passed on to the executing script are presented by an asterisk and a number. In the description is explained how the script makes use of them. The output describes what the users have to expect as a result for their input. Last the actors are identified. They represent what kind of endpoints are involved in the information retrieval process.

5.1.1 Scenario 1: FOAF data

Scenario 1 is the core use case scenario of the xOperator agent. As shown on semanticweb.org[26], the FOAF vocabulary is one of the most widespread ontologies, so the use cases described here can be tested by a broad audience. We created these use cases in anticipation of possible usage patterns for querying personal information. As depicted in Figure 5.1.1 this scenario consists out of four uses cases.

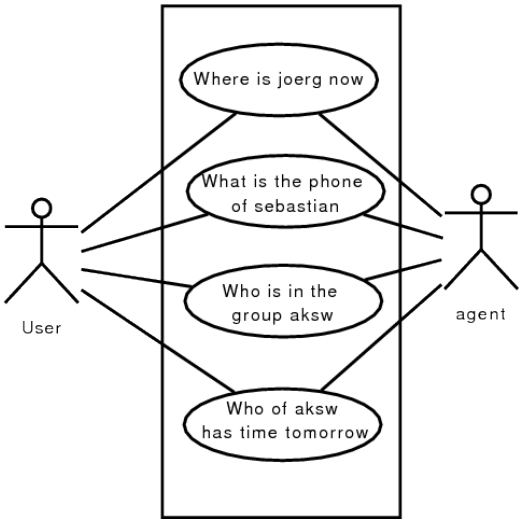

Figure 10: Scenario 1: FOAF data

Use case scenario 1.1 locates a user by examining associated calendar information.

[26]http://semanticweb.org/wiki/Ontology

Use case 1.1: Where is joerg now	
Short Description	Identifies the location of a person.
Template	Where is *1 now
Long Description	- The user enters the search string. - Calendar information belonging to a person matching *1 is searched globally. - The found calendar information is searched for entries pointing to now. - Location information associated with the user is printed out.
Output	The location of the user as a human readable text or as a link.
Actors	User, User Agent, P2P Agents

Use case 1.2 can be used for querying any kind of attribute associated with a certain person. This could be phone number, street name, birthday, etc.

Use case 1.2: What is the phone of sebastian	
Short Description	Searches for an attribute associated with a person
Template	What is the *1 of *2
Long Description	- The user enters the search string. - Local ontologies are searched for an attribute with a description that contains *1. - Persons, that have this attibute and match *2 are searched globally. - The result his shown to the user.
Output	The value of the property associated with the person.
Actors	User, User Agent, P2P Agents

In use case 1.3 we describe the search for members belonging to a certain group, in this example of the AKSW working group.

Use case 1.3: Who is in the group aksw	
Short Description	Lists persons associated with a certain group.
Template	Who is in the group *1
Long Description	- The user enters the search string. - A global search for persons associated with a group labeled *1 are searched. - The results are merged and displayed to the user.
Output	The location of the user as a human readable text or as a link.
Actors	User, User Agent, P2P Agents

The last use case of this scenario retrieves the members of a group and examines calendar information associated with them. The calendars are then queried in order to find out if an entry exists for a certain point in time.

Use case 1.4: Who of aksw has time tomorrow	
Short Description	Searches for members of a group who have no entry in their calendar at a certain time.
Template	Who of *1 has time *2
Long Description	- The user enters the search string. - Members of the group *1 are searched globally. - Calendar data is fetched for each found person. - The calendar data is searched for entries on *2 - The members are printed out, each with a list of entries on that day.
Output	The name of the person plus any calendar entries of that day.
Actors	User, User Agent, P2P Agents.

5.1.2 Scenario 2: DBpedia data

In this use case scenario we show how the xOperator can be used to query a large dataset like DBpedia. We created two sample interactions, as shown in Figure 11.

The first use case 2.1 queries DBpedia for concepts of the yago[27] hierachy that

[27]http://www.mpi-inf.mpg.de/ suchanek/downloads/yago/

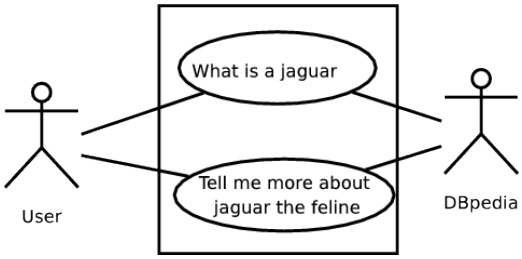

Figure 11: Scenario 2: DBpedia data

are associated with a certain term.

Use case 2.1: What is a jaguar	
Short Description	Searches for classes in the yago hierachy that contains a specific string in their label.
Template	What is a *1
Long Description	- The user enters the search string. - DBpedia is queried for a list of concepts containing *1. - The list is presented to the user.
Output	A list of the concepts.
Actors	User, User Agent, DBpedia.

The second use case 2.2 queries for more information about this concept.

Use case 2.2: Tell me more about jaguar the feline.	
Short Description	Diplays information about a specific concept.
Template	Tell me more about *1 the *2
Long description	- The user enters the search string. - The concept matching *1 in the label and *2 in the name is queried from DBpedia. - A short description of the concept is displayed.
Output	A short description of the concept.
Actors	User, User Agent, DBpedia.

5.2 Requirements

With the description of the use cases the requirements can be determined. These are differentiated into technical and quality requirements, the technical requirements describe the functions the agent needs in order to fullfill the use cases while the quality requirements address cross cutting concerns of the use cases.

5.2.1 Technical Requirements

The requirements defined here describe the minimum functional aspects of the xOperator. These requirements are directly linked to the functional view on the system, which is described in Chapter 6.

Interaction

Means of interacting with the agent have to be created. This includes the ability to receive the users input and respond with useful output by associating the input to a use case specific logic.

Data querying

Data is queried in multiple ways. These ways are local, remote and peer to peer querying.

- Local querying allows the user to query the semantic resources he previously has specified, like FOAF profiles or ontologies.
- Remote query endpoints like DBpedia in use case scenario 2 need to be accessible.
- Queries need to be passed on to neighboring agents and thus each agent needs to be able to answer these queries. Also the section of the social network the agent resides in has to be made available for querying.

Configuration

Numerous functionalities of the agent need precise configuration information. These include the information on how to log into the users IM account in order to make use of the social overlay network. Further, the location of the resources that the users trust and they want to be published in their network need to be stated. Also information which users should be able to access the agent needs to be configured.

5.2.2 Quality Requirements

The quality attributes presented here relate to the list presented in International Organization for Standarization (2001) in a complementary way, complete coverage is not intented but an overview of attributes that we deem as central to the xOperator project.

Changeability

The ability to adapt the xOperator to other use cases is central for further development. The usage scenarios presented in Chapter 5.1 are intended as a demonstration. bein able to define own use cases is essential to the xOperator and thus it has to provide facilities for a fast and efficient implementation of these.

Security

As one of the central ideas of the xOperator is to preserve the users privacy, measures have to be taken to prevent the misuse of the agent. A system is required to authorize every user or inter-agent information exchange. Also, the use of secure transmission lines for message transport is required to prevent privacy intrusion.

Usability

When using IM as the way of interacting with the user, usability has an especially high account. Presentation of results has to comply with the restricted user interface of the IM clients and as well has to work with a variety of clients. Clients to be considered range from desktop IM clients like Pidgin[28] to mobile clients as fring[29]. This further includes that the users should be able to modify the system to their own use cases in an effcient way.

Portability

The xOperator should be working on as many platforms as possible in order to increase the target audience. Also, deploying the software on a server is to be considered.

[28] http://www.pidgin.im/
[29] http://www.fring.com/

5.3 Activities

Based upon the use cases a general workflow was created and is depicted in Figure 12. Differentiating the activities into generic and specific part helps achieving extensionability. The use case specific logic is thus contained in the script execution activity.

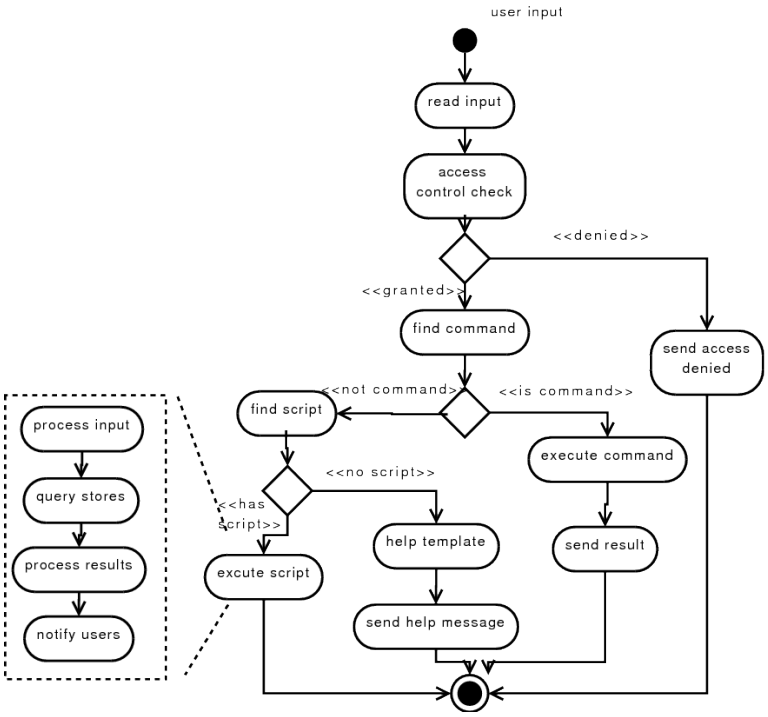

Figure 12: Activity of user input

First, the generic user input activity consists of the reception and processing of the user input. The input is then checked against a control mechanism that verifies that the input is coming from a legitimate user of the agent. Input received from not authorized users is reject. If this validation is passed, the input is then checked if it is a command used for configuration or testing the agent. If a matching command is found, the appropriate command is called for execution and the output of the command is handed down to the user.

If no matching command is found the input is handed down to the template matching engine. This engine is configured to respond to all user input by first trying to determine whether a script can be associated with the input. If a script

is associated it gets called with the wildcards determined by the engine. The script is now in charge to execute the necessary steps needed to fullfill a certain use case and to send the result to the user. If no script can be found, the help template is called and the user is presented a list of exiting templates and further help options.

The second activity the agent performs is the answering of incoming queries from peering agents. In the use cases this ability is implicitly described, as there is stated that the agent can query neighboring agents. The corresponding activities are described in Figure 13.

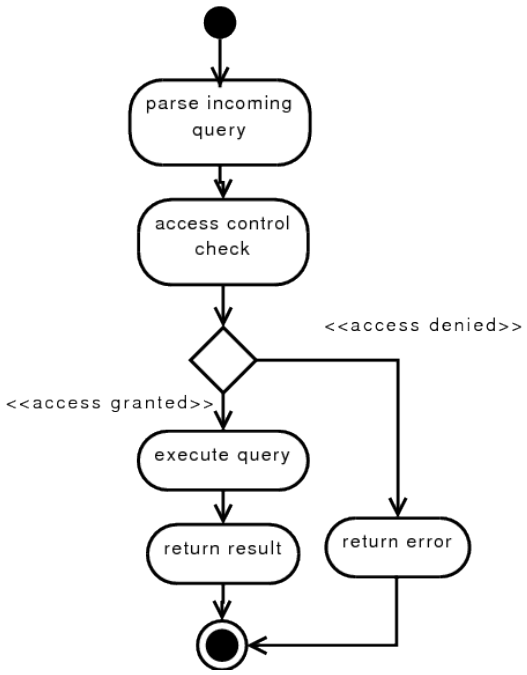

Figure 13: Activity of p2p input

Incoming queries are first parsed and then checked against a control mechanism that allows only authorized agents to perform queries. If a querying agent does not fall into this category, an error is reported back. Otherwise the query is executed using the resources the user first assigned to this agent. The result is then sent back to the querying agent.

6 Architecture

In his article about the IEEE 1471 standard Maier et al. (2001) defines software architecture as

the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

According to this definition we will first look on how the xOperator system is integrated in its environment. Later we will describe the internal working of the system. According to Perry & Wolf (1992) we will first give an introdcution about the style in which we arranged the components and later describe the modules that are arranged according this style. The description of this style is made from the functional point of view as proposed by the author.

6.1 Integration

We decided to implement the xOperator as a standalone application with no user interface. As a result any chat client is able to communicate with the xOperator. There were two alternatives to that, an integration into the IM client and an integration into the IM server.

Regarding message exchange which is depicted in Figure 14 the selected solution produces some overhead compared to the other proposals as every interaction between the user and the agent is transported over the network.

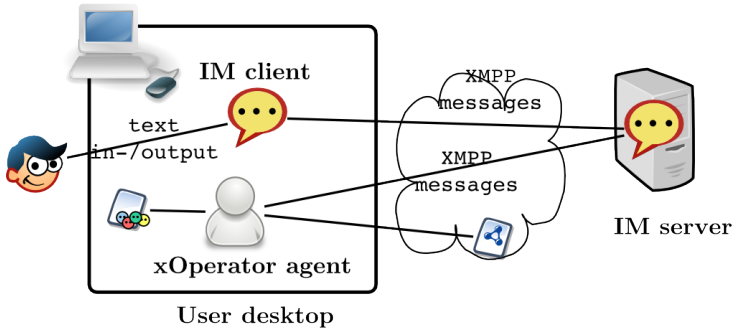

Figure 14: User interaction as standalone application

An implementation directly into the IM client or server would either eliminate or reduce this overhead. Integrating the agent into other software would impose

other drawbacks.

- Compared to the IM client integration, the agent is not tied to a desktop computer. As a result the xOperator can both be deployed on a desktop or on a server, leaving the choice to the user or operator. This makes the scenario of managed hosting possible which means that the agent could run in a managed environment with a guaranteed availability. So future business model could be offering the xOperator as a service.

- Integration into a server would remove the xOperator from the control of the user which is a core idea of the xOperator.

- Both scenarios of integration would tie the agent to a specific implementation of a client or server. This would reduce portability and make it harder for a user to evaluate the agent.

6.2 Architectural Style

For the first proof of concept, no architectural style was employed as it was intended to be a throw-away prototype with a small code base. In the following iteration the Blackboard pattern was selected because of its advantages in extendability, as described by Harrison & Avgeriou (2007), but was later refactored to the Model-View-Controller (MVC) pattern. We describe the reasons for this change in Chapter 6.2.2.

6.2.1 Blackboard Pattern

The Blackboard pattern was originally designed for non-deterministic operations as are encountered in speech recognition systems (Nii 1989). As Stegemann et al. (2007) points out, this pattern can also be used for managing a workflow and arranging business logic accordingly. The basic elements of the blackboard pattern are shown in Figure 15.

The interaction between the elements can best be described using the analogy the name of this pattern suggests. A system built according to this pattern consists of the following elements.

- The blackboard. A single instance that keeps track of the data. Here the data represents all information that is gathered and processed in order to

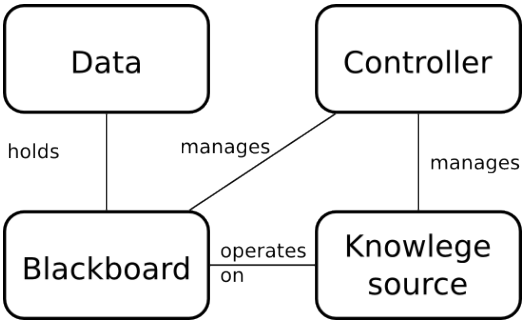

Figure 15: The Blackboard pattern according to Garcia et al. (2003)

solve a business case or process some input. For isolating different business cases this blackboard is segmented, so the data belonging to one business case is contained in one segment.

- Knowledge sources. Knowledge sources encapsulate the logic to process one specific task such as validating user input. This encourages a strong separation of concerns. Knowledge sources read from a segment of the blackboard and post the results of their processing back onto it.
- The controller. Contains an execution plan that determines the knowledge source that should contribute to a business case and which knowledge source should subsequently be called.

The whole process is started by placing a problem on the blackboard and opening a new segment on it. The Controller will then call the knowledge sources according to its plan until the process is considered finished.

6.2.2 Change of the Architectural Pattern

As Stegemann et al. (2007) points out, this architectural paradigm offers in combination with the reference implementation openBBS[30] advantages like extensibility, control flow management and enforces some implementation practices like programming against interfaces. As Harrison & Avgeriou (2007) shows in his comprehensive comparsion of various system architectural patterns, the Blackboard pattern has drawbacks some of which were encountered while creating the application and were

[30]http://openbbs.sourceforge.net/

found so grave that the architectural paradigm was changed. In particular the following problems that appeared during the implementation finally led to a change in the architectural paradigm:

Oversynchronization in the framework A lot of parallel calls, like SPARQL queries are expected, so it is crucial to have a performant support for parallelism. In the openBBS implemention 0.8 parallel access on the blackboard is serialized which has strong impact on the performance and blocks the application.

Unfamiliarity with the pattern None of the interested developer were familiar with the pattern. As the project is intended to be further developed as an open-source project with multiple developers this could hinder developers in contributing to the project.

Overhead in workflow management Beside the blackboard implementation, the openBBS framework offers numerous classes and interfaces required for implementing the workflow. The workflows, like answering a remote query as shown in Chapter 5.3, do not branch often and cannot take advantage of the implementation while producing overhead in the implementation.

6.3 Model-View-Controller Pattern

In its latest iteration the MVC pattern is used to orchestrate the interaction of the systems modules. This pattern is one of the most established architectural paradigms and is described for example by Reenskaug (1979). In this pattern, user input is forwarded to the controller for evaluation. Then the controller calls the appropriate methods in the model to process the data. Finally, the view is notified and the updated data gets displayed.

The refactoring from the blackboard pattern to MVC can be regarded as mapping. The controller component of the blackboard performs the same task in the MVC. The knowledge sources of the blackboard were transformed into the model with the exception of the knowledge sources formerly taking care of the user input, which were transformed into the view component. The architecture of the latest iteration can be seen in the next chapter with a description of its building blocks.

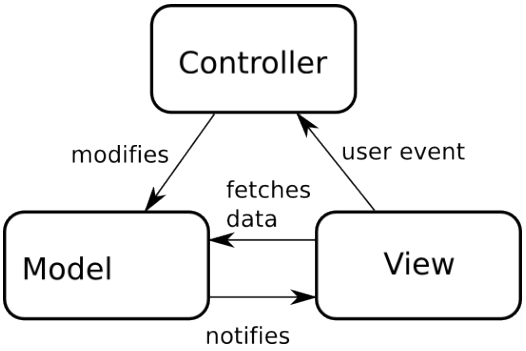

Figure 16: The Model-View-Controller pattern

6.3.1 Modularization

One of the fundamental principles of system architecture is the modularization as for example illustrated by Perry & Wolf (1992). The functionality of the information system gets broken into functional segments that do not overlap. These segments form the modules or components of the application. The modules depicted in Figure 17 each deal with the requirements previously defined and are arranged according to the MVC pattern. The modules are described in short regarding the functionalities they cover and which requirements they address.

Controller Receives incoming user interaction and selects the appropriate modules from the model to process the input. The two activities described in section 5.3 define the behavior.
Input: The parsed user or peer agent input.
Output: The parsed answer determined by evaluating the appropriate modules of the model or an error condition.
Requirements: Changeability, as this allows an easier adoption to different workflows.

View: User interaction This module manages the interaction with the user.
Input: User input from the messaging network.
Output: The processed results according to the input.
Requirements: Usability (non-interference with human-to-human communication), Interaction (textual user-agent communication).

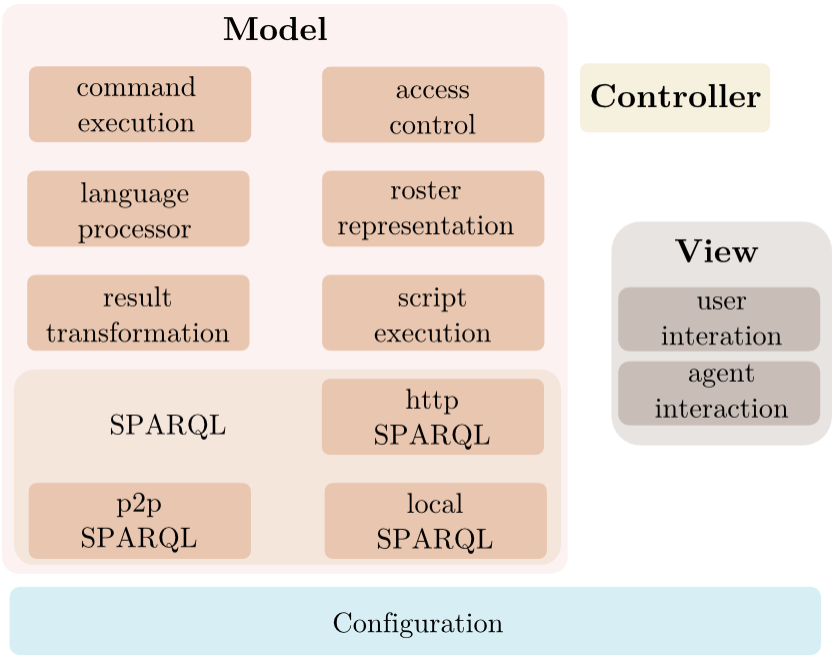

Figure 17: System architecture

View: Agent interaction Manages the interaction between the agents in the p2p network.
Input: SPARQL queries received from peering agents.
Output: The results of these queries.
Requirements: Data querying (agent-to-agent communication), usability (non-interference with human-to-human interaction).

Model: language processor Maps the non-command input to scripts via templating and provides help functionality when no matching scripts are found.
Input: The parsed user input.
Output: The name of the script associated with a template and extracted parameters or a help message.
Requirements: Interaction (language-to-script mapping), usability.

Model: command execution Performs commands necessary to test and configure the agent.

Input: The parsed user input.
Output: The result of the commands, configuration modification.
Requirements: Configurability.

Model: roster representation Transforms the roster and thus the state of the social network into a RDF representation.
Input: Information about peering agents.
Output: A corresponding RDF representation.
Requirements: Data querying (agent-to-agent communication).

Model: script execution Executes the use case specific logic. Includes querying of data sources, processing of the results and presentation to the user.
Input: The name of the script and the extracted parameters.
Output: The results of the script.
Requirements: Extensionability, usability.

Model: access control Validates if a message is authorized to be processed, based upon configured rules and the roster representation.
Input: The sender of the message.
Output: An authorization or a reject message.
Requirements: Security.

Model: result transformation Transforms the result of a SPARQL query into a human readable form.
Input: A SPARQL result set.
Output: The textual, human readable representation.
Requirements: Usability.

Model: SPARQL Provides an efficient way to execute queries against various SPARQL end points. Is separated into submodules that share a common behavior.

SPARQL local Loads configured resources and makes them available as

background graphs. Provides a generic SPARQL endpoint that is able to dynamically load resources identified in scripts and queries. Answers queries received through the View: agent communication module.

SPARQL remote Queries remote SPARQL endpoints and maintains a list of SPARQL endpoints that the user wants to use.

SPARQL p2p Distributes SPARQL queries among it the agents peers and collects the responses.

Input: A SPARQL query, configuration information.
Output: A SPARQL result set.
Requirements: Data querying.

Configuration Manages the configuration of all the other modules.
Input: A serialization of the systems configuration.
Output: The configuration of the system as objects.
Requirements: Configurability.

7 Prototype Implementation

The implementation of the prototype was executed in several iterations, according to the iterative approach described in 1.3. While the first iteration was a proof of concept, the second and third iterations were structured approaches to satisfy the defined use case scenarios and the functional and quality requirements. In this chapter the most relevant aspects of the implementation are described and discussed.

First, the basic implementation environment is described. Then we discuss how the design created in Chapter 6 is used to model the system. The following chapters describe the implementation of the modules of the design, followed by the chapters covering the implementation of the modules described in Chapter 6.3.1.

All descriptions and depictions of the components are reduced in such a way that their core functionality can be described but further details not essential for understanding the implementation is left out.

7.1 Implementation Environment

We chose to implement the xOperator agent as a Java[31] application. Since programs written in Java are executed in a virtual machine the demanded portability is given. While this requirement could be satisfied by many languages such as Python or .NET, mature libraries covering IM, SW and language processing can be found in no other programming language. Further, Java is widely spread among the scientific community and the knowledge to use its tools exists in the AKSW working group.

In order to be compliant with the selected libraries and to encourage further development the GNU General Public License 3.0[32] was selected. The source code and the build process is managed by Apache Maven[33], the code resides publicly available as a Google Code project[34] where in addition the development process is supported by bug and feature tracking tools.

7.2 Design implementation

While the first architectural pattern was implemented using the openBBS framework we chose to implement the MVC ourselves. In terms of implementation the key difference between the Blackboard pattern and the later used MVC approach is that the openBBS requires the implementation of the control flow as defined by the activities defined in Chapter 5.3 to be done by a control plan. The control flow in the MVC implementation resides in the controller component.

The modules are connected via the PicoContainer[35]. This container makes use of the principle of Dependency Injection which, according to Fowler (2004), means that dependencies between modules are resolved by the container. An implementation using this principle requires the dependent module not to directly reference an implementation but rather an abstract description, an interface. The container of the application is then in charge to find an implementation of this interface. Fowler (2004) describes that this behavior encourages the developer to program against interfaces and separate the modules according to their concerns. As a further advantage the author mentions that testing is easier using Dependency

[31] http://java.com/
[32] http://gplv3.fsf.org/
[33] http://maven.apache.org/
[34] http://code.google.com/p/xoperator/
[35] http://www.picocontainer.org/

Injection. The PicoContainer uses a constructor-based injection approach. This requires all classes to have constructors that demand for references to all required classes. As a result circular dependencies of the modules are prevented.

Modules are implemented as a set of classes and interfaces organized in one package for each module. This ranges from the one class implementation of the controller to the SPARQL modules with multiple subpackages and numerous interfaces and classes.

7.3 Controller Implementation

The controller is the module that connects and orchestrates the other modules and is implemented as a single class. Input from the user or neighboring agents is received from the view by registering itself as a listener into the view component according to the pattern described by Gamma et al. (1995). The class diagram in Figure 18 shows the layout of this class.

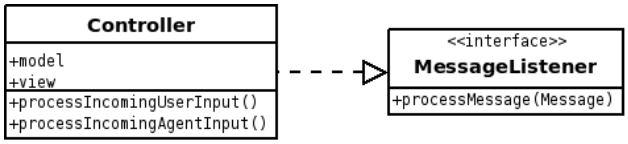

Figure 18: Controller implementation

The two functions depicted here refer to the two main activities, as described in Chapter 5.3. The implementation of the activities is a sequence of calls to the previously injected classes.

7.4 View Implementation

For implementation of the view components a suitable IM protocol has to be found first. As Adams (2002) points out, XMPP is the only viable option here. It is not tied to a certain vendor and is not restricted to messaging but also supports machine-to-machine communication. From the comprehensive listing by XMPP Standards Foundation (2009) the SMACK framework was chosen, as it offers the best documented extension facilities.

The view addresses two different aspects, user and agent interaction. As a lot of functionality is shared these are implemented in one package offering a separate

interface for each type of communication.

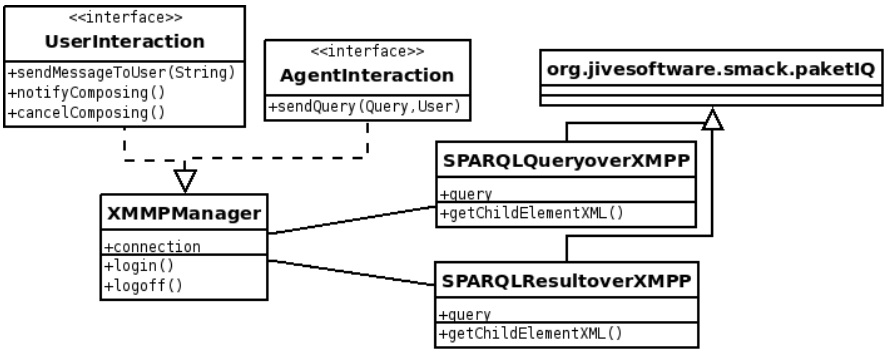

Figure 19: View implementation

In Figure 19 the core classes of the view implementation are depicted. The two already mentioned interfaces that are implemented by the class `XMPPManager`, which is encapsulates all activities that require communication over its XMPP `connection`. This is for example logging into a server or receiving messages from other users. Also wrapper classes for sending and receiving queries are part of this package, these classes are directly derived from the info/query implementations of the SMACK framework and contain the functionality to transform a SPARQL query or result into a serialization that is suitable for transport over the IM network.

In the following chapters we cover details of further interesting aspects of the view implementation.

7.4.1 Basic Interaction Functionalities

The xOperator logs into the user account using the credentials provided by the configuration module. This account is used for agent-to-agent communication. For the user-agent interaction a separate, special account is defined as shown in the roster in Figure 20.

This account, the agent account, was created as most IM clients do not allow the user to contact himself. This would be necessary as the agent mainly logs into the users account. The agent account can be added to the user's roster easily and relays from there the users input further to the controller.

Using an agent account is optional and the deployment without such an account is possible. This mode can be used if a shared access to one xOperator instance is

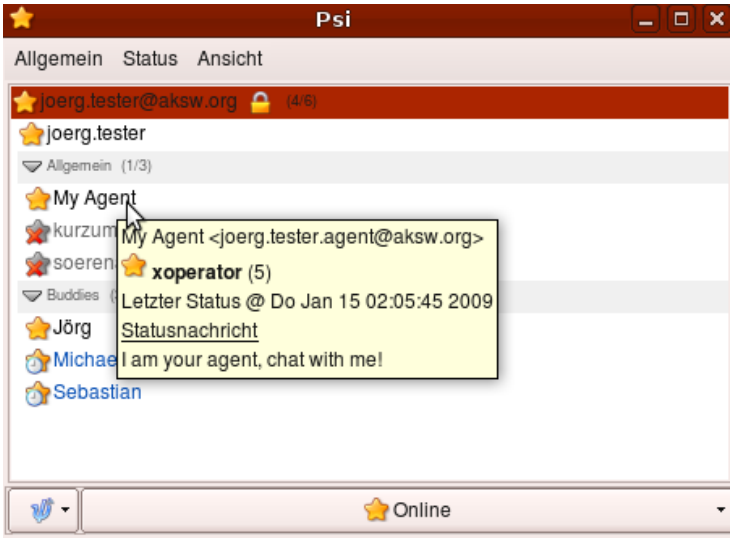

Figure 20: The agent account in the users contact list

desired. This can be the case if a data store should be made available via chat for a group of users.

7.4.2 Presence

Presence information is generated whenever the user is logged into the server. It indicates other users whether or not the user is willing to communicate. Presence information is used by most IM networks and usually consists of a status and a status message. While the status is a predefined value, the message usually can be set arbitrarily. Presence information is a substantial element of an IM network and as we require the agent not to disturb human activity we took measure no to disturb user interaction.

When the xOperator agent is logged into the account of the user this will affect the presence of the user. At the time there is no way to hide an agent from a user, as explained in Saint-Andre (2005), so the agent is configured by default to appear as an absent, low priority user, with a status message indicating that the agent is not able to communicate with the user. The screenshot in Figure 21 illustrates this behavior.

Here the Psi[36] client shows the presences associated with the user in a hovering

[36]http://psi-im.org/

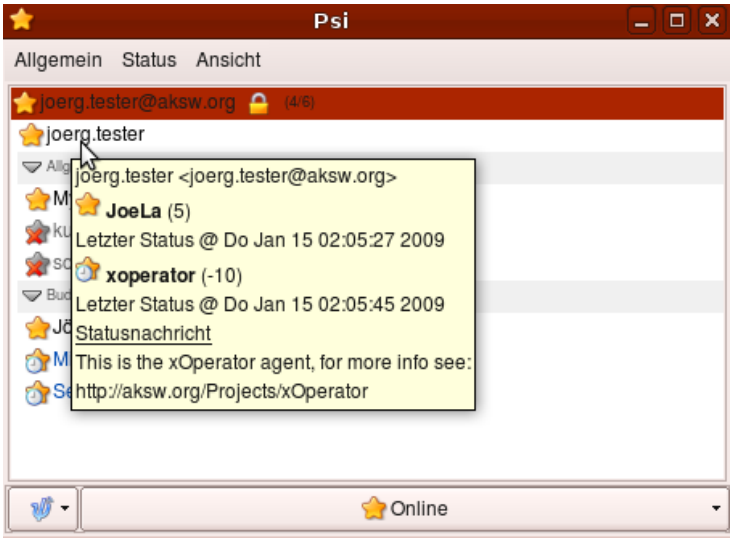

Figure 21: Multiple presences of a user

box over the users contact. It consists of two entries, one belongs to the chat client identified by the resource `JoeLa` and one that belongs to the xOperator. Other users will see this presence information in exactly the same way. The bottommost icons of Figure 21 illustrate the appearance of the roster if only the xOperator is logged in. Here an icon indicating the absence is shown. In case the agent gets contacted nevertheless, an automatic reply is returned, instructing the communication partner to try again later.

7.4.3 Agent Autodiscovery

Goal of the autodiscovery is the identification of agents among each other. For that purpose each agent sends a special message of type info/query (`iq`) to all known and currently active peers. Info/query messages are intended for internal communication and queries among IM clients without being displayed to the human users. An autodiscovery query from one client of Figure 9 to another for example, would look as follows:

```
<iq from="user1@example.com/Agent" type='get'
    to="user2@example.com/Agent" id='...'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

A positive response to this feature discovery message from an xOperator agent would contain a feature with resource ID *http://www.w3.org/2005/09/xmpp-sparql-binding*. This experimental identifier/namespace is defined by Brickley (2005). The response message to the previous request would look as follows:

```
<iq from='user2@example.com/Agent' type='result'
    to='user1@example.com/Agent' id='...' />
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity
    <foaf:Person
     category='client' name='xOperator' type='bot'/>
    <feature
     var='http://www.w3.org/2005/09/xmpp-sparql-binding'/>
    <!-- ... more here -->
  </query>
</iq>
```

The autodiscovery is triggered upon the reception of new presence information. Should the presence information to change, this process is triggered and if neceesary the roster representation described in Chapter 7.5.2 is notified.

7.4.4 Peer-to-Peer Query Transport

The serialization of incoming and outgoing SPARQL queries is managed by the `SPARQLQueryOverXMPP` and `SPARQLResultOverXMPP` classes depicted in Figure 19. We implemented this transport mechanism according to the proposal of Brickley (2005). This proposal defines a mode to embed the protocol part of the SPARQL definition in the info/query packets of XMPP.

In this example we illustrate the transport of a sample SPARQL query.

```
<iq from="user1@example.com/Agent" type='get'
    to="user2@example.com/Agent" id='...'>
  <query
   xmlns="http://www.w3.org/2005/09/xmpp-sparql-binding">
    SELECT DISTINCT ?Concept WHERE {[] a ?Concept} LIMIT 5
  </query>
</iq>
```

The corresponding answer is shown in the subsequent listing.

```
<iq from='user2@example.com/Agent' type='result'
 to='user1@example.com/Agent' id='...' />
 <query-result xmlns=
   "http://www.w3.org/2005/09/xmpp-sparql-binding">
  <sparql
   xmlns="http://www.w3.org/2005/sparql-results#">
   <head> <variable name="Concept"/> </head>
   <results distinct="false" ordered="true">
    <result> <binding name="Concept"> <uri>
 http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
      </uri> </binding> </result>
   </results>
  </sparql>
 </query-result>
</iq>
```

7.5 Model implementation

The model section contains the instructions on how to access and process the data of the xOperator and contains the modules as described in Chapter 6.3.1.

7.5.1 Security and Access Control

In order to fullfill the security requirements defined in 5.2.2, in addition to the facilities built into XMPP access control has to be implemented.

Basic security in terms of encryption is already provided through the Simple Authentication and Security Layer (SASL) authentication mechanisms defined in Saint-Andre (2004). This guarantees an encrypted connection between the server and the client. Communication from client to client can furthermore be encrypted as proposed in Muldowney (2005), a behavior not yet implemented. Even without this, authenticity of the sender is guaranteed by the authentication mechanism defined in Saint-Andre (2004).

Access control could be defined in two ways. First, the server maintains a list of blocked contacts. This list can be adjusted using the mechanisms provided by

XMPP, which leaves the filtering to the server. Second the messages reach the agent but get rejected or accepted according to ruleset. Implementing the filter list in the agent has the benefit of being more flexible with the technology used. Instead of string matching more sophisticated techniques like regular expressions or coupling with an Lightweight Directory Access Protocol (LDAP) server are possible.

The current implementation of the access control relies on a list of regular expressions provided by the configuration module. A sample configuration is shown in the following listing.

```
<security>
    <allowGroupsToChat>
      <pattern>^friends$</pattern>
    </allowGroupsToChat>
    <allowUsersToChat>
      <pattern>^.*@example.edu$</pattern>
    </allowUsersToChat>
    <allowGroupsP2PQuery>
      <pattern>
  ^(friends|colleagues|younameit)$
      </pattern>
    </allowGroupsP2PQuery>
    <allowUsersP2PQuery>
      <pattern>^.*@example.edu$</pattern>
    </allowUsersP2PQuery>
    <accessDeniedChatMessage>Sorry, i am a bot
      and unable to chat with you.
    </accessDeniedChatMessage>
</security>
```

Whenever the sender address of an incoming message does not match the appropriate pattern or the sender does not belong to a certain group on the users roster access is blocked. If the message was from a human-agent interaction, the `accessDeniedChatMessage` is returned. In the case of an inter-agent comunication an error is returned.

7.5.2 Roster Representation

The agent does not only share information that was previously added by the user. In the requirements we defined that the social network visible to the agent should be included in the local store of the agent and that it should be shared among its peers. The XMPP roster that is associated with every account in the XMPP network contains this information and is transformed into an RDF representation by the roster representation module. The information added includes details about name and email addresses but also about online presences and whether the presence is associated to an xOperator. In effect, existing data, for example in the FOAF format gets interlinked with the presence information of the agent allowing richer data retrieval.

The representation of the roster is modeled with the idea of reusing well established vocabularies for a maximum of integration into existing platforms and concepts.

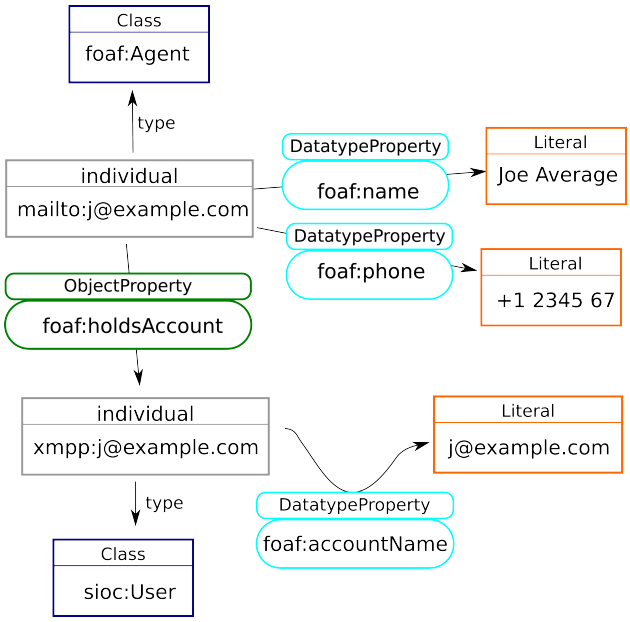

Figure 22: Static roster model

The representation is depicted in Figure 22. Each contact of the roster including the user of the xOperator is represented using a `<foaf:Agent>`. This class is intended to represent human beings as well as chat bots or other interaction enabled

computer programs.

Further information about the person, like nickname and email addresses, are added using classes and properties defined in the FOAF namespace. Information that can be used as URI is also employed as identifiers for the `<foaf:Agent>` for an easier linkage with other information about the identifiers. The information that this `<foaf:Agent>` has an XMPP account is modeled using the class `<sioc:User>`. This class is a subclass of `<foaf:OnlineAccount>`. The service provider is identified by the `<sioc:chatAccountHomepage>` property (not depicted here), the user is identified by the `<foaf:accountName>` property.

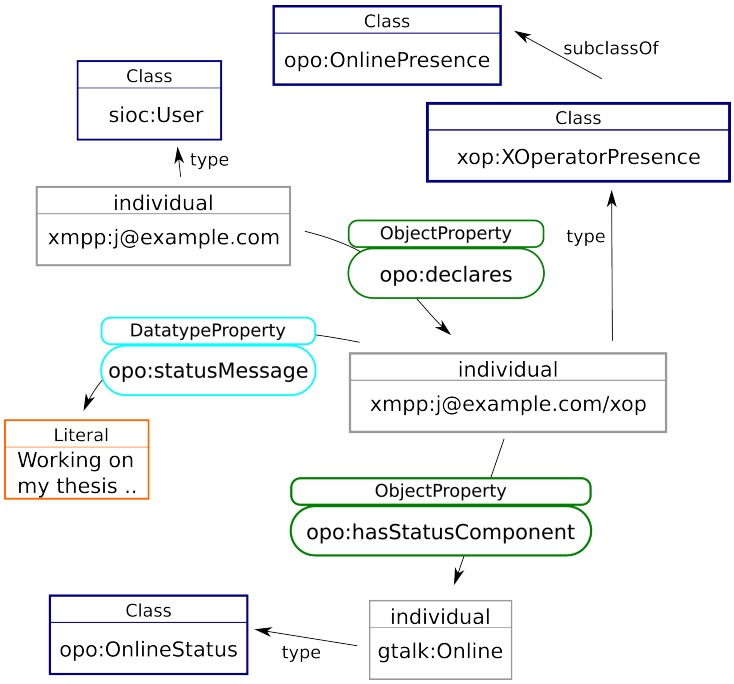

Figure 23: Presence roster model

The information presented in Figure 22 is unlikely to change during runtime. Information with frequent changes are the presences of the contacts. The presence is modeled using the vocabulary of the Online Presence Ontology (OPO) and is depicted in Figure 23. The fact, that a certain user with an online account is online is modeled by adding an instance of `<opo:OnlinePresence>` via the `<opo:declaresOn>` property to a `<sioc:User>`. This `<opo:OnlinePresence>` references status information, modeling the momentary desire of the user to com-

municate.

We extended the vocabulary of the OPO for the use in the xOperator by introducing a new subclass of `<opo:OnlinePresence>`, the `<opo:XOperatorPresence>`. We use this specialization in order to signal that a peering xOperator was found through the autodiscovery process described in Chapter 7.4.3. This information can be used in scripts in order to query for neighboring xOperator agents.

7.5.3 Language Processing

While the view module of the xOperator manages the interception of messages from the communication channel, the input is, as long it is not a command, not yet associated with any action or response. This is the task of the language a fully-fledged Natural Language Processing (NLP) tool is not feasible for this task. We instead opted in for a template based matching approach. For this we selected the Artificial Intelligence Markup Language (AIML). This language allows the definition of templates and offers basic NLP techniques such as stopword removal or stemming, as described by Wallace (2005).

We selected chatterbean[37] as AIML interpreter as it offers full support of AIML and an efficient integration into the xOperator.

An AIML template file is a collection of categories. A `category` consists of a `pattern` and a `template`. The interpreter validates input by searching for the best matching `pattern`. In the following example we defined the most unspecific `pattern` consisting only of `*`. The asterisk is a special character that matches everything. It is also the character that extracts parts of the input and makes it available for the scripts.

```
<category>
  <pattern>*</pattern>
  <template>I am sorry but I did not understand your
      command. For some instructions type in HELP,
      to hear some rumors type in GOSSIP.</template>
</category>
```

Whenever this `pattern` is matched the associated template is executed by the AIML interpreter. In this case, only plain text is included in the `template` so this

[37] http://chatterbean.bitoflife.cjb.net/

text is sent back to the user. As the * is the most unspecific `pattern` we use this template in the xOperator to signal the user that no other match was possible.

The `pattern` of the `category` of the following listing is more specific and refers to use case 1.3.

```
<category>
  <pattern>WHO IS IN THE GROUP *</pattern>
  <template>
<external
 name="groovy" param="foafGroupMember.groovy"/>
  </template>
</category>
```

This `template` contains no text but the tag `external` parameterized with the name of the associated script. The tag `external` does not belong to the AIML standard, the chatterbean interpreter was extended to handle such a tag. Here the mapping between the input and the script takes place. With the identification of the script name the scripting enviroment can be called. The text matched by the asterisk (*) character is passed on to the script in form of a parameter in the script context, as described in Chapter 7.5.4

7.5.4 Scripting Environment

We designed a scripting environment for an effective execution of the scripts which are implementing the use cases. As scripting language Groovy[38] was selected since it offers a seamless integration into the existing Java application. We selected this scripting approach as the scripts can be changed during runtime and further offer the ability to integrate use case implementations without modification to the system. The whole application can thus be seen as a platform for effective script execution.

An example implementation of such a script is described in detail in Chapter 7.7. When this script is called the executing scripting environment provides access to some functionalities of the xOperator. This context is injected via the script execution context and is modeled in the `GroovyContext` class, which is depicted in Figure 24.

[38]http://groovy.codehaus.org/

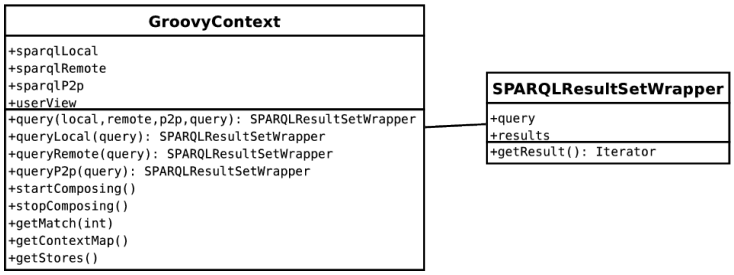

Figure 24: Groovy context classes

The `GroovyContext` is connected to the SPARQL components of the xOperator system and to the user view components, as it provides means of interacting with the SPARQL endpoints and the user of the system. These functionalities are encapsulated in the query methods. They allow a script to query stores in an efficient way. As a query will usually result in more than one answer, these results are wrapped in a `SPARQLResultSetWrapper`. This class allows access to the results in an object oriented way and supports iteration over the results, which is demonstrated in Chapter 7.7.

For interaction with the user it is not only possible to send text messages via the `sendMessage(text)` method, but also via the `startComposing()` and `stopComposing()` functions. These signal the user that the agent is reacting to the input and is currently processing. In case of queries with a long runtime this increases usability as the user knows that something is happening. Basically, this is an imitation of interacting with a human.

Further the `getMatch(int)` method is essential for the scripts. Whenever a script processes the user's input, the `getMatch(int)` method provides access to the wildcards defined in the templates. Thus, this methods connects the language processing component described in Chapter 7.5.3 with the script execution.

The other methods provide access to the configuration of the agent. With `listStore()` the stores defined in the configuration are accessible. The method `getContextMap()` allows a script to read or write values that are set by previously executed scripts.

7.5.5 Result Transformation

The result of a SPARQL query is returned as XML. Instead of processing these results using the wrapper class provided by the context, as described in Chapter 7.5.4, the results can as well be presented to the user by XSL Transformation (XSLT). For a basic transformation into a human consumptional form this component was created. The basic principle is depicted in Figure 25.

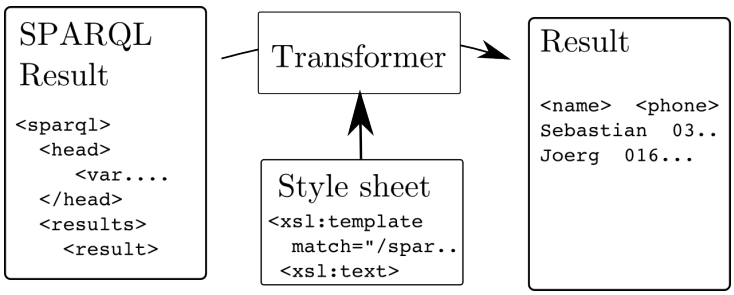

Figure 25: XSL Transformation

We used the standard `javax.xml.transform.Transform` provided by the virtual machine for applying the transformation to the result set. The transformation is defined in a user-customizable stylesheet and is a variation of the widely used result-to-html.xsl[39] style sheet provided by the W3C[40]. The clean text result can then be sent straight to the user.

7.5.6 Command Interface

The command interface was created in order to execute shell-like commands. During the first iterations all configuration was done using commands. In the latest version the xOperator solely relies on the configuration module.

Still included is the `query` command that executes a query as show in the listing below.

```
<joerg.tester> query SELECT ?x WHERE {?x a ?y} LIMIT 2

<My Agent> RDF-Store ontowiki (Remote) answered:
        x
```

[39] http://www.w3.org/TR/rdf-sparql-XMLres/result-to-html.xsl
[40] http://www.w3.org/

```
<http://bis.ontowiki.net/AlexanderGross>
<http://bis.ontowiki.net/AntoniusvanHoof>

<My Agent> RDF-Store local (Local) answered:
       x
<http://www.unbehauen.net/
  nejdl-w-2002-604-a_Decker_Stefan>
<http://xmlns.com/foaf/0.1/Image>
```

This command uses the default transformation as described in Chapter 7.5.5 to produce human readable output.

7.5.7 Semantic Web Framework

The SPARQL component encapsulates the facilities required for answering the queries created in the scripts. The implementation tallows a parallel, multi-threaded execution of these queries so that the execution time is determined by the longest running query. The component is structured into subcomponents that take care of the different execution environments of the queries. These are local, remote and p2p targeted queries and the structure incorporating this is depicted in Figure 26.

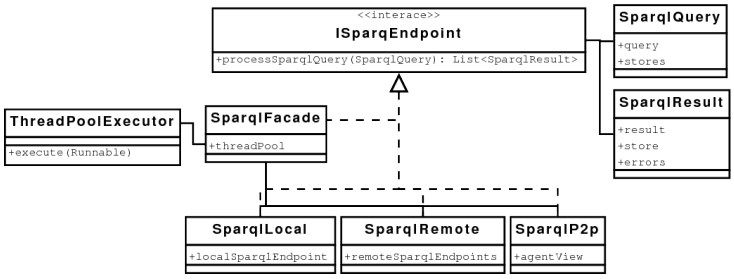

Figure 26: Sparql implementation classes

We created the `SparqlQuery` and `SparqlResult` classes to wrap the textual queries and results. Theses classes also contain information about which stores are to query and which store a result is from and wether an error was encountered during execution.

The parallel execution is governed by the `SparqlFacade`, which distributes the queries according to the targets defined in the `SparqlQuery`. The idea of this class

is to provide a facade, as described by Gamma et al. (1995), for a convenient access to the whole querying mechanism and to hide its complexity.

Parallelism is implemented via classes in the `java.util.concurrent` package. We used a predefined pool of threads that performs the actual querying. For canceling queries with a long execution time we implemented a time-out, after which the results of a query are discarded. This guarantees a maximum reaction time and prevents blocking.

The functionality for the communication with the different types of endpoints is modeled in separate classes.

`LocalSparqlEndpoint`

This class implements the requirement for being able to answer queries on the user's own resources like FOAF profiles or calendar information. We use the Jena/ARQ[41] framework for creating an in-memory SPARQL endpoint. During startup, all configured resources are loaded into this store and the roster representation, as described in Chapter 7.5.2, is added and kept up to date. By calling the `execute(query)` method defined in `ISparqlEndpoint` results from this store can be retrieved.

`RemoteSparqlEndpoint`

For communication with remote SPARQL endpoints this class was created. Receiving a query by the `execute(query)` method causes this component to either query all or just the endpoints configured in the query to be asked for a result. The communication with the endpoint uses the HTTP using the HttpClient[42] library of the Jakarta Commons project.

`P2pSparqlEndpoint`

Queries that require a forwarding to peering clients are managed by this class. It wraps the queries into the transport wrapper `SPARQLQueryOverXMPP` which is described in Chapter 7.4 and sends them using the agent view functionality. Also the collection of the results is managed by the `P2pSparqlEndpoint`, so the asynchronous nature of these calls are hidden and the normal synchronous calls can be made.

[41] http://jena.sourceforge.net/ARQ/
[42] http://hc.apache.org/httpclient-3.x/

7.6 Configuration

The configuration module is in charge of supplying information to the other modules with information about the resources, XMPP credentials and security definitions. The configuration is stored as an XML representation of the configuration objects. Figure 27 depicts a selection of these objects in combination with the class responsible for writing to and reading them from the configuration file, the `ConfigurationSerializer`, which makes use of the XStream[43] library for reading and writing.

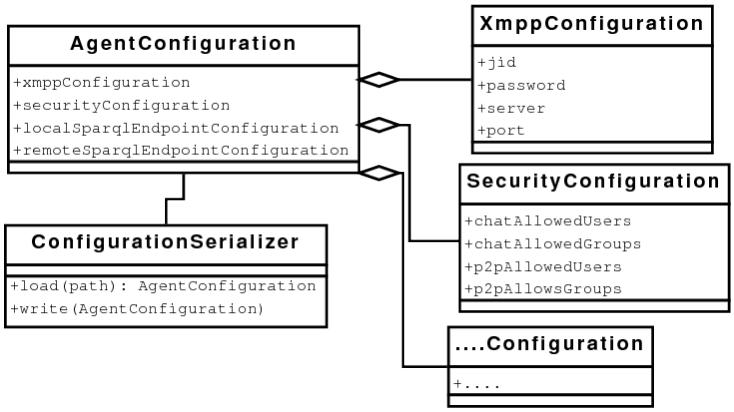

Figure 27: Configuration classes

In the earlier releases of the xOperator this information was managed by commands of the command module (see Chapter 7.5.6). In the current release the user modifies the configuration file directly. This has the advantage of being able to add comments in order to support the user supplying the right information. A snippet from a configuration file is shown in the following listing.

```
<agent>
  <localEndpoint>
    <trustedDocuments>
      <!--enter the URLs to the resources
          you want to add to the local store here-->
  <string>
  http://unbehauen.net/joerg-foaf.rdf
```

[43] http://xstream.codehaus.org/

```
  </string>
    </trustedDocuments>
  </localEndpoint>
  <jabber>
    <mainAccount>
      <!--enter your user name(jid)
  for your XMPP account here-->
      <jid>main@example.com</jid>
      <!--enter the password for the XMPP account here-->
      <password>example</password>
      <!--enter the server address here->
      <server>jabber.example.com</server>
.....
```

7.7 Use case implementation

In this chapter we present the implementation of use case 1.3 *Who is in the group aksw*. This simple use case consists out of one SPARQL query to determine all the users associated with a certain group, which is send to all configured SPARQL endpoints. After successful execution a list of all matching users is presented to the user.

When this script is called all the activities as described in Chapter 5.3 have been executed, so the user is authorized. For this example we assume that the input is *Who is in the group aksw?*.

First of all the query is constructed. It is a static text string in which a parameter read out by the template engine is inserted. This parameter is read out by calling the `context.getWildcard(1)` function. The parameter indicates which wildcard is to be read out.

```
memberQuery =
"PREFIX foaf: <http://xmlns.com/foaf/0.1/> " +
"SELECT DISTINCT ?name WHERE {"+
  "?s ?p1 ?subjectpattern. "+
  "?s foaf:member ?person. " +
  "?person foaf:name ?name."+
"FILTER regex(" +
```

```
      "?subjectpattern, "+
  " '.*"+context.getWildcard(1)+".*', 'i'" +
" ).}"
```

In detail the SPARQL query consists first out of the `PREFIX` declaration, in which is stated that we want to use `foaf` as an abbreviation for the namespace URI identifying the FOAF vocabulary. In the second line we declare that we want the variable `?name` to contain the result. The `WHERE` clause described the graph pattern that describes the pattern parts of the graph have to match.

The first graph pattern presented there searches for a subject `?s` which is interlinked via an attribute `?p1` with an object that matches `?subjectpattern`. This pattern matches all triples in the graph but is defined here, since we need it, as shown later, for finding the group name. Secondly, this subject `?s` has to be linked by the attribute `foaf:member` to an object `?person`. Here it is implicitly stated, that `?s` is a `foaf:group` and `?person` is a `foaf:Person`, as the attribute `foaf:member` defines a relation between these two classes. Furthermore we define that a `?person` also has to have a `foaf:name` relation which binds to the variable `?name`, which is the variable we defined in the `SELECT` part as the variable to be presented as a result.

The following section of the graph pattern starting with `FILTER` defines the conditions the variable `?subjectpattern` has to fullfill. Here the function `regex` is used and is parametrized with the wildcard match from the AIML engine. In this example this would be *aksw*. This, in combination with the first line requires that `?s`, a `foaf:group` as shown earlier, needs to be tied to a string containing the text *aksw*. The predicate `?p1` therefore was not specified in order to be as generic and farfetching as possible.

Having defined the query now the function provided by the xOperator can be used to execute the query. The function herefore is `context.query()`. This passes the previously defined query and three following parameters in which we declare that the query should be executed against the local store, all configured remote stores and all peering agents, as shown in the following listing.

```
members = context.query(memberQuery,true,null,null)
```

In `members` the results for each store are saved in form of a list. The next step is to pass the results to the user. The following listing shows the iteration over the result set of each store.

```
members.each(){res->
  if (res.getResultCount() > 0)
        context.sendMessageToUser(
      "From " + res.getStoreName() + ":")
  res.getResultRows().each(){
    context.sendMessageToUser(" * "+it["name"])
  }
}
```

To indicate which store and thus where the data is coming from, for each result set the name of the store is displayed. This followed by the list of names determined by the query, as shown in Figure 28.

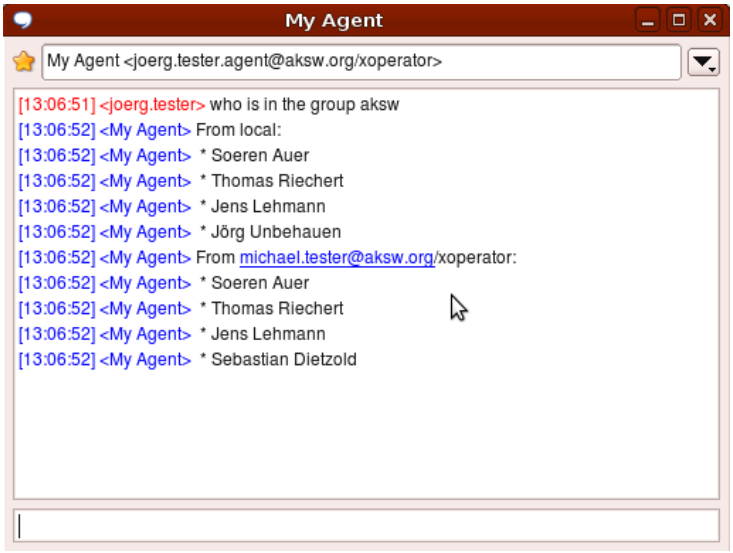

Figure 28: Screenshot of the use case 1.3 execution

The listings here show only the core functionality of this particular script. A full listing of this script can be found in the Appendix A.2. In addition to this, error handling and logging is included there. Also in the Appendix a listing of the script that implements use case 1.2 can be found(Appendix A.1).

8 Evaluation

The evaluation was performed on a deployment of four agents. As information sources we used FOAF profiles (20 documents, describing 50 people), the SPARQL endpoint of our semantic Wiki OntoWiki (Auer et al. 2006) (containing information about publications and projects), information stored in the LDAP directory service of our department, iCal calendars of group members from Google calendar (which are accessed using iCal2RDF[44]) and publicly available SPARQL endpoints such as DBpedia (Auer et al. 2007). Hence the resulting information space contains information about people, groups, organizations, relationships, events, locations and all information contained in the multidomain ontology DBpedia. This setup is depicted in Figure 29.

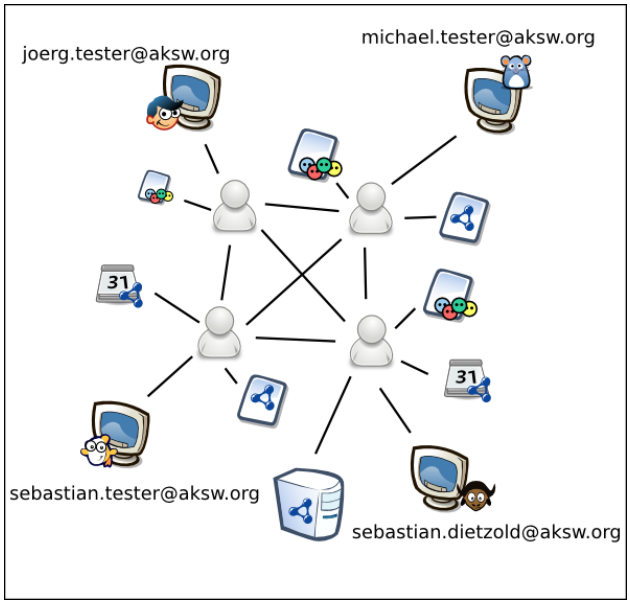

Figure 29: Test environment deployment

8.1 Qualitative Evaluation

The benefit of perfoming queries in a social context can best be described by an example. We assume that `joerg.test@aksw.org` is searching similar to the

[44]http://www.kanzaki.com/courier/ical2rdf

description in Chapter 3.2.1 for the phone number of somebody he encountered on some social event, in this case we know that this person is called *sebastian*. He uses the xOperator to search for this number by posing the question *what is the phone of sebastian*. In Figure 30 this interaction which corresponds to use case 1.2 is depicted.

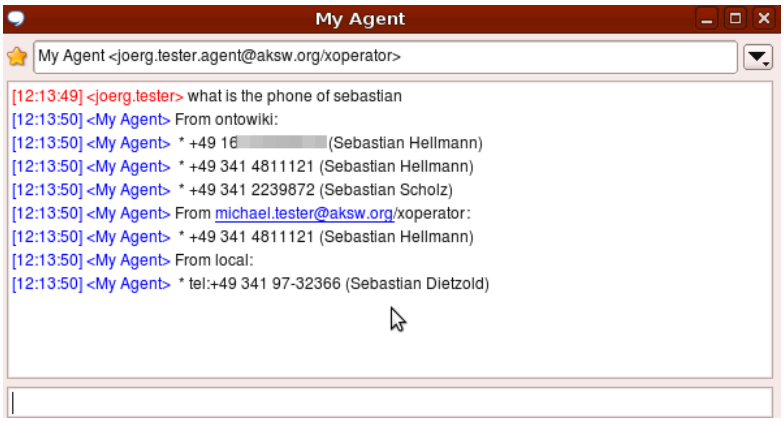

Figure 30: Screenshot of use case 1.2

Altough this question is rather unspecific it resulted in only 5 results, so the social context of the query limits the number of results to a reasonable amount. A similar query performed on a directory like a phone book would have retsulted in thousands of matches.

The provenance of the data is provided in this use case with the presentation of the names of the stores. We can use this information to select one of the results. For example, if we know that this *sebastian* also knows `michael.tester@aksw.org` we can assume that the number provided by his agent is likely to be the right number. Also in the case of contradictory information the end user could use the provenance information presented here to select the right one.

Furthermore privacy is kept. Altough `joerg.tester@aksw.org` is able to query sensitive information like an address book, he has previously been granted the privilege to do so by being added to the roster of `michael.tester@aksw.org`. Others that do not maintain such a connection to `michael.tester@aksw.org` are not able to query.

While in this use case we can clearly show what benefits querying in a social context can have, we also evaluated how well questions can be answered by the

xOperator. As long as questions are posed along the patterns defined in the templates, the results can be considered good. If data for a query is available, it is returned in a senseful manner. Whenever input is entered that cannot be related to a script however, nothing can be returned. So before the xOperator is able to answer a question like *Who has a homepage about squirrels* the appropriate pattern has to be created.

This becomes even more apparent in the use cases of scenario 2. The huge amount of knowledge provided by DBpedia needs a flexible way of querying which unfortunately cannot be provided by a template matching approach. This is illustrated in Figure 31.

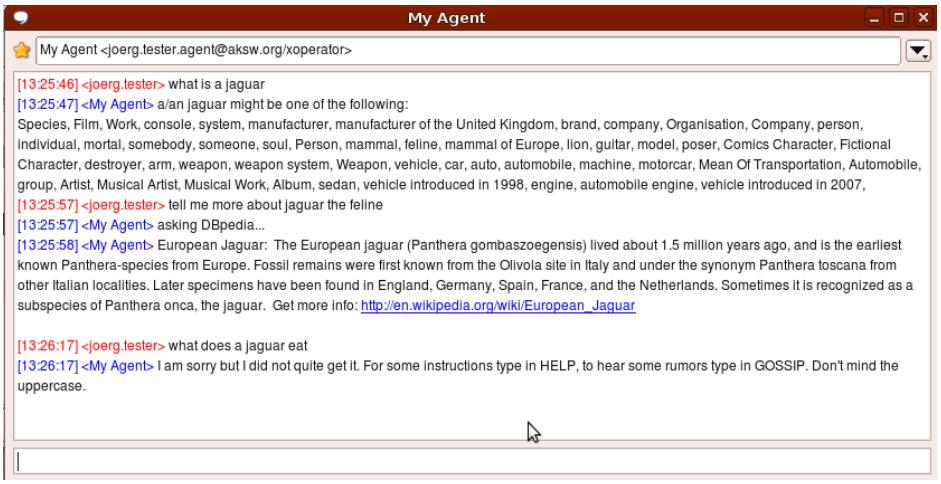

Figure 31: Screenshot of use case scenario 2

So whenever a template matches, good results are presented, but for questions not covered by the templates no results can be retrieved. While the use case scenario 1 with its limitation to the domain of FOAF can be quite well covered with templates as presented in use case 1.2, the multi-domain knowledge is harder to cover. These findings cover well with the results of Freese (2007) who also experienced problems with large knowledge bases when using AIML as an interface.

8.2 Performance

The performance of this setup is shown in table 1. Here some exemplary uses cases and some other scripts not belonging to the use cases are measured in terms

	Template	Scenario 1	Scenario 2	Scenario 3
1	What is / Tell me (the) * of *	2.3	3.9	1.5
2	Who is in the group *	3.5	4.3	1.6
3	Where is * now	5.1	6.7	4.2
4	Tell me more about *	–	–	3.5
5	Free dates * between * and *	5.1	6.8	4.7
6	Which airports are near *	–	–	3.4

Table 1: Average response time in seconds (client to client) of some AIML patterns used in three scenarios: (1) 20 documents linked from one FOAF profile, 1 personal agent with no neighborhood (2) 20 documents linked from different FOAF profiles and spread over a neighborhood of 4 agents (3) one SPARQL endpoint as an interface to a Semantic Wiki or DBpedia store

of execution time. The numbers were gathered by analyzing the log files over multiple runs. The xOperator release 0.1 was used for this evaluation.

The response timings indicate that the major factor are latency times for retrieving RDF documents or querying SPARQL endpoints. The impact of the number of agents in the agent network as well as the overhead required by the xOperator algorithm is rather small. This results in intuitive perception that xOperator is a very responsive and efficient way for query answering.

8.3 Query design

Experiences during the evaluation have led to the following rules for creating patterns and queries in xOperator.

(1) Query as fuzzy as possible: Instant Messaging is a very quick means of communication. Users usually do not capitalize words and use many abbreviations. This should be considered, when designing suitable AIML patterns. If information about the person 'Sören Auer' should be retrieved, this can be achieved using the following graph pattern: `?subject foaf:name "Auer".` However, information can be represented in multiple ways and often we have to deal with minor misrepresentations (such as trailing whitespace or wrong capitalizations), which would result for the above query to fail. Hence, less strict query clauses should be used instead. For the mentioned example the following relaxed SPARQL clause, which matches also substrings and is case insensitive, could be used:

```
?subject foaf:name ?name.
FILTER regex(?name,'.*Auer.*','i')
```

(2) Use patterns instead of qualified identifiers for properties: Similar, as for the identification of objects, properties should be matched flexible. When searching for the `homepage` of 'Sören Auer' we can add an additional property matching clause to the SPARQL query instead of directly using, for example, the property identifier `foaf:homepage`:

```
?subject ?slabel ?spattern.
?subject ?property ?value.
?property ?plabel ?ppattern.
FILTER regex(?spattern,'.*Auer.*','i')
FILTER regex(?ppattern,'.*homepage.*','i')
```

This also enables multilingual querying if the vocabulary contains the respective multilingual descriptions. Creating fuzzy queries, of course, significantly increases the complexity of queries and will result in slower query answering by the respective SPARQL endpoint. However, since we deal with a distributed network of endpoints, where each one only stores relatively small documents this effect is often negligible.

(3) Use sub queries for additional documents: In order to avoid situations where multiple agents retrieve the same documents (which is very probable in a small worlds scenario with a high degree of interconnectedness) it is reasonable to create query scripts, which only distribute certain tasks to the agent network (such as the retrieval of prospective information sources or document locations), but perform the actual querying just once locally.

(4) Be cautious with user output: As many queries produce a large number of results measures have to be taken to reduce the output of the scripts. This includes for example merging results or present error messages that demand the user to be more specific. An example for this can be seen in the query script for use case 1.1, which can be found in the Appendix A.1.

8.4 Exposure to the Scientific Community

As the xOperator is available as an open-source project and thus was exposed to the public. The releases were announced on mailing lists and on the AKSW

blog[45]. Further Yves Raimond[46] and Dan Brickley[47] wrote in their blogs about the xOperator releases.

Feedback was also collected using the tracking tool on the Google Code project site[48] and the xOperator mailing list[49].

This included the wish for support for different IM networks and getting help setting up xOperator. As the xOperator requires the XMPP network for query traversal only the latter problem could be addressed. Upon the requests the setup process was simplified by a better documentation of the configuration file and further a Getting Started Guide[50] was created.

Also help and guidelines for adopting AIML templates to custom use cases were requested. For an easier start into the creation of own scripts an overview of the existing templates and a documentation of their functionality we are about to set up a wiki[51].

The presentation of the xOperator at the European Semantic Web Conference (ESWC) 2008[52] and the International Semantic Web Conference (ISWC) 2008[53], both high profiled Semantic Web conferences, also generated feedback. On the ESWC the project was presented as a full paper (Dietzold, Unbehauen & Auer 2008*b*)) and as a demonstration paper (Dietzold, Unbehauen & Auer 2008*a*). The xOperator participated in the Semantic Web Challenge (SWC) of the ISWC 2008 where our entry (Unbehauen, Martin, Hellmann, Dietzold & Auer 2008) made it into the top five submissions. The entry presented by the author can be viewed on videolectures.net[54]. For the Semantic Web Challenge a demonstration page[55] was created in order to allow an easy evaluation of the project.

The feedback gathered on the ISWC was mostly centered around the language processing, where the limited expressability of the AIML approach was criticized, as described in Chapter 8.1 The ability of the xOperator to transport SPARQL queries over XMPP was received well. One of the chairs of ISWC, Tom Heath, encouraged

[45]http://blog.aksw.org
[46]http://blog.dbtune.org/
[47]http://danbri.org/words/
[48]http://code.google.com/p/xoperator/issues/list
[49]http://groups.google.com/group/xoperator
[50]http://aksw.org/Projects/xOperator/FirstSteps
[51]http://aksw.org/Projects/xOperator/Templates
[52]http://www.eswc2008.org/
[53]http://iswc2008.semanticweb.org/
[54]http://videolectures.net/iswc08_swcbtc/
[55]http://aksw.org/Projects/xOperator/SWC2008

to better specify this feature in order to have a reference implementation of the standard proposed by Brickley (2005).

9 Conclusions and Future Work

9.1 Conclusions

With the xOperator concept and its implementation, we have showed how a deeply and synergistic coupling of Semantic Web technology and Instant Messaging networks can be achieved. The approach naturally combines the well-balanced trust and provenance characteristics of IM networks with semantic representations and query answering of the Semantic Web. The xOperator approach goes significantly beyond existing work which mainly focused either on the semantic annotation of IM messages or on using IM networks solely as transport layers for SPARQL queries. xOperator on the other hand overlays the IM network with a network agents, which have access to knowledge bases and Web resources of their respective owners. The neighborhood of a user in the network can be easily queried by asking questions in a subset of natural language. By that xOperator resembles knowledge sharing and exchange in offline communities, such as a group of co-workers or friends. We have showcased how the xOperator approach naturally facilitates contacts and calendar management as well as access to large scale heterogeneous information sources. In addition to that, its extensible design allows a straightforward and effortless adoption to many other application scenarios such as, for example, sharing of experiment results in Biomedicine or sharing of account information in Customer Relationship Management. In addition to adopting xOperator to new domain application we view the xOperator architecture as a solid basis for further technological integration of IM networks and the Semantic Web.

9.2 Future Work

While the xOperator reached a state able to showcase the core ideas there are still plenty of feature requests we want to satisfy in the near future.

During the presentation of the xOperator as a participant in the Semantic Web Challenge of the ISWC 2008 most feedback received was related to the way of processing language. While the idea of mapping natural language to scripts was

generally accepted, the lack of adaptibility was criticized and a venturing into more dynamic mapping approaches was encouraged. Although alternatives have already been investigated, more research could possibly go into this direction.

Also during the ISWC we were encouraged build a spearate library for this purpose and further refine the proposal of Brickley (2005), which would allow other applications to exploit the idea of a social overlay network for querying. Factoring out a library that is one of the goals for the near future.

In the two usescase scenarios we restricted ourselves to easily available data. For a better demonstration on how the xOperator works and how the user can benefit from its use, more data needs to be made available. The incorporation of some kind of desktop crawler based upon, for example, the Aperture Framework[56] could address this problem. This framework allows the extraction metadata from digital content like media files which then can be stored inside the triple store of the agent. In that way the xOperator can be extended to new application domains, for example by examining the music preferences of my social network. Beside possible dangers to privacy this is nevertheless an interesting application scenario and will be evaluated in the near future.

Altough we tried to reduce the complexity of writing scripts to a minimum, still a basic understanding of pattern matching, scripting and SPARQL are required. By implementing a platform were scripts can be presented, requested and exchanged this issues can be addressed. Thus we plan to create such a platform on top of the semantic applications of our AKSW working group.

A further idea to explore is to compine the implicit conept about trust in the xOperator concept with an explicit. The work of Golbeck et al. (2003) in which users are assigned trust values, is an interesting approach. This would for example allow a ranking of responses according to the assigned trust level. The work of Olaf Harting[57] on an trust enabled SPARQL extension is promising and will be evaluated on the xOperator.

An idea for the far future is to research the implementation of a more sophisticated routing protocol, that allows query traversal beyond directly connected nodes without flooding the whole network.

[56]http://aperture.sourceforge.net/
[57]http://www2.informatik.hu-berlin.de/ hartig/

10 Acronyms

AKSW Agile Knowledge Engineering and Semantic Web

AIML Artificial Intelligence Markup Language

ESWC European Semantic Web Conference

FOAF Friend Of A Friend

GUI Graphical User Interface

HTTP Hyper Text Transfer Protocol

HTML Hypertext Markup Language

IM Instant Messaging

IRI Internationalized Resource Identifier

ISWC International Semantic Web Conference

JID Jabber Identifier

LDAP Lightweight Directory Access Protocol

MVC Model-View-Controller

NLP Natural Language Processing

OPO Online Presence Ontology

OWL Ontology Web Language

RDF Resource Description Framework

RDFS RDF Schema

RIF Rule Interchange Format

SASL Simple Authentication and Security Layer

SIOC Semantically-Interlinked Online Communities

SPARQL SPARQL Query Language for RDF

SW Semantic Web

RDF Resource Description Framework

URI Uniform Resource Identifier

URL Uniform Resource Locator

WWW World Wide Web

XML Extensible Markup Language

XMPP Extensible Message and Presence Protocol

XSLT XSL Transformation

References

Adams, D. J. (2002), *Programming Jabber: Extending Xml Messaging*, first edn, O'Reilly Associates.

Antoniou, G. & van Harmelen, F. (2008), *A semantic web primer*, 2. ed. edn, MIT Press, Cambridge, USA.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. & Ives, Z. G. (2007), DBpedia: A Nucleus for a Web of Open Data, *in* 'Proc. of ISWC/ASWC', pp. 722–735.

Auer, S., Dietzold, S. & Riechert, T. (2006), OntoWiki - A Tool for Social, Semantic Collaboration., *in* 'Proc. of ISWC', pp. 736–749.

Avison, D. & Fitzgerald, G. (2006), *Information Systems Development: Methodologies, Techniques and Tools*, 4 edn, McGraw-Hill Higher Education.

Berners-Lee, T. (1998), 'Semantic web roadmap', http://www.w3.org/DesignIssues/Semantic.html. accessed: December 2, 2008.

Berners-Lee, T., Hendler, J. A. & Lassila, O. (2001), 'The Semantic Web', *Scientific American* **284**(5), 34–43.

Bernstein, A., Kaufmann, E., Göhring, A. & Kiefer, C. (2005), Querying ontologies: A controlled english interface for end-users, *in* 'In 4th International Semantic Web Conference', pp. 112–126.

Brickley, D. (2005), 'http://www.w3.org/2005/09/xmpp-sparql-binding', http://www.w3.org/2005/09/xmpp-sparql-binding. accessed: 10 January, 2009.

Brickley, D. & Miller, L. (2004), FOAF Vocabulary Specification, Namespace Document 2 Sept 2004, FOAF Project. http://xmlns.com/foaf/0.1/.

Conolly, D. (2000), 'A Little History of the World Wide Web', http://www.w3.org/History.html. accessed: January 14, 2009.

Dietzold, S., Unbehauen, J. & Auer, S. (2008*a*), xoperator – an extensible semantic agent for instant messaging networks, *in* 'Proceedings of 5th European Semantic Web Conference (ESWC 2008)', pp. 787–791.

Dietzold, S., Unbehauen, J. & Auer, S. (2008*b*), xoperator – interconnecting the semantic web and instant messaging networks, *in* 'Proceedings of 5th European Semantic Web Conference (ESWC 2008)', pp. 19–33.

Fowler, M. (2004), 'Inversion of control containers and the dependency injection pattern', http://www.martinfowler.com/articles/injection.html. accessed: January, 22 2009.

Franz, T. & Staab, S. (2005), SAM: Semantics Aware Instant Messaging for the Networked Semantic Desktop, *in* 'Semantic Desktop Workshop at the ISWC'.

Freese, E. (2007), Enhancing AIML Bots using Semantic Web Technologies, *in* 'Proc. of Extreme Markup Languages'.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Massachusetts.

Garcia, A. F., de Lucena, C. J. P., Zambonelli, F., Omicini, A. & Castro, J., eds (2003), *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications [the book is a result of SELMAS 2002]*, Vol. 2603 of *Lecture Notes in Computer Science*, Springer.

Golbeck, J., Parsia, B. & Hendler, J. (2003), Trust networks on the semantic web, *in* 'In Proceedings of Cooperative Intelligent Agents', pp. 238–249.

Gross, R. & Acquisiti, A. (2005), 'Information revelation and privacy in online social networks (the Facebook case)', *Proceedings of the Workshop on Privacy in the Electronic Society* .

Groza, T., Handschuh, S., Moeller, K., Grimnes, G., Sauermann, L., Minack, E., Mesnage, C., Jazayeri, M., Reif, G. & Gudjonsdottir, R. (2007), The nepomuk project - on the way to the social semantic desktop, *in* T. Pellegrini & S. Schaffert, eds, 'Proceedings of I-Semantics' 07', JUCS, pp. pp. 201–211.

Gruber, T. (1992), 'What is an Ontology?', http://www-ksl.stanford.edu/kst/what-is-an-ontology.html. accessed: January 16, 2009.

Harrison, N. B. & Avgeriou, P. (2007), Leveraging architecture patterns to satisfy quality attributes, *in* F. Oquendo, ed., 'ECSA', Vol. 4758 of *Lecture Notes in Computer Science*, Springer, pp. 263–270.

Herman, I. (2008), 'W3C Semantic Web Frequently Asked Questions', http://www.w3.org/2001/sw/SW-FAQ. accessed: January 13, 2009.

Herman, I. (2009), 'W3C Semantic Web Activity', http://www.w3.org/2001/sw/. accessed: January 13, 2009.

International Organization for Standarization (2001), ISO/IEC Standard 9126: Software Engineering – Product Quality, part 1, Technical report.

Karger, D. R., Bakshi, K., Huynh, D., Quan, D. & Sinha, V. (2005), Haystack: A general-purpose information management tool for end users based on semistructured data, *in* 'Proc. of CIDR', pp. 13–26.

Kaufmann, E. & Bernstein, A. (2007), How useful are natural language interfaces to the semantic web for casual end-users?, *in* '6th International Semantic Web Conference (ISWC 2007)', pp. 281–294.

Lampe, C., Ellison, N. & Steinfield, C. (2006), A face(book) in the crowd: social searching vs. social browsing, *in* 'CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer Supported Cooperative Work', ACM, New York, NY, USA, pp. 167–170.

Lerman, K. (2007), 'Social browsing - information filtering in social media', http://arxiv.org/abs/0710.5697. accessed: December 12, 2008.

Leskovec, J. & Horvitz, E. (2008), Planetary-scale views on a large instant-messaging network, *in* 'WWW '08: Proceeding of the 17th international conference on World Wide Web', ACM, New York, NY, USA, pp. 915–924.

Maier, M. W., Emery, D. & Hilliard, R. (2001), 'Software architecture: Introducing ieee standard 1471', *Computer* **34**(4), 107–109.

Manola, F. & Miller, E. (2004), 'RDF Primer', http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. accessed: December 20, 2008.

McPherson, M., Lovin, L. S. & Cook, J. M. (2001), 'Birds of a feather: Homophily in social networks', *Annual Review of Sociology* **27**(1), 415–444.

Muldowney, T. (2005), 'XEP-0027: Current Jabber OpenPGP Usage', http://xmpp.org/extensions/xep-0027.html. accessed: January 12, 2009.

Nardi, B. A., Whittaker, S. & Bradner, E. (2000), Interaction and outeraction: instant messaging in action, *in* 'CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work', ACM, New York, NY, USA, pp. 79–88.

Nii, H. P. (1989), Blackboard systems, *in* A. Barr, P. R. Cohen & E. A. Feigenbaum, eds, 'The Handbook of Artificial Intelligence (Volume IV)', Addison-Wesley, Reading, MA, pp. 1–82.

Osterfeld, F., Kiesel, M. & Schwarz, S. (2005), Nabu - A Semantic Archive for XMPP Instant Messaging, *in* 'Semantic Desktop Workshop at the ISWC'.

Perry, D. E. & Wolf, A. L. (1992), 'Foundations for the study of software architecture', *SIGSOFT Softw. Eng. Notes* **17**(4), 40–52.

Prud'hommeaux, E. & Seaborne, A. (2008), 'SPARQL query language for RDF', http://www.w3.org/TR/rdf-sparql-query/. accessed: December 1, 2008.

Quan, D., Bakshi, K. & Karger, D. R. (2003), A Unified Abstraction for Messaging on the Semantic Web, *in* 'WWW (Posters)'.

Reenskaug, T. (1979), 'Models - views - controllers', http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html. accessed: January 9, 2009.

Saint-Andre, P. (2004), Extensible Messaging and Presence Protocol (XMPP): Core, RFC 3920, The Internet Engineering Task Force (IETF). accessed: January 16, 2009.

Saint-Andre, P. (2005), 'XEP-0126 Invisbility', http://xmpp.org/extensions/xep-0126.html. accessed: January 12, 2009.

Shadbolt, N., Berners-Lee, T. & Hall, W. (2006), 'The semantic web revisited', *IEEE Intelligent Systems* **21**(3), 96–101.

Shum, S. B., Roure, D. D., Eisenstadt, M., Shadbolt, N. & Tate, A. (2002), 'CoAKTinG: Collaborative advanced knowledge technologies in the grid', http://www.bib.ecs.soton.ac.uk/data/7480/pdf/CoAKTinG-WACE2002.pdf. accessed: December 7, 2008.

Skype Limited (2008), 'Skype appoints lead roles for technology and strategy', http://about.skype.com/2008/12/. accessed: January 15, 2009.

Stegemann, S. K., Funk, B. & Slotos, T. (2007), A Blackboard Architecture for Workflows, *in* J. Eder, S. L. Tomassen, A. L. Opdahl & G. Sindre, eds, 'CAiSE Forum', Vol. 247 of *CEUR Workshop Proceedings*, CEUR-WS.org.

Tencent Holdings Limited (2009), 'Tencent qq homepage', http://www.imqq.com/. accessed: January 15, 2009.

Travers, J. & Milgram, S. (1969), 'An Experimental Study of the Small World Problem', *Sociometry* **32**, 425–443.

Unbehauen, J., Martin, M., Hellmann, S., Dietzold, S. & Auer, S. (2008), 'xoperator - chat with the sematic web', http://www.cs.vu.nl/ pmika/swc-2008/xOperator-xOperator.pdf. Accessed: January 19, 2009.

Wallace, R. (2005), 'Artificial Intelligence Markup Language (AIML)'. accessed: January 17, 2009.

XMPP Standards Foundation (2009), 'Xmpp software: Libraries', http://xmpp.org/software/libraries.shtml. Accessed: January 15, 2009.

List of Figures

A Use Case Implementations

A.1 Use Case 1.2

The implementation of use case 1.2, taken from the file *foafPersonAttributes.groovy*.

```
//////////////////////////////////////////////
/// Implementation of the template "What is the * of *"
//////////////////////////////////////////////

// fetching the log
log = context.getLog();

attributeMatch = context.getMatch(1);
subjectMatch = context.getMatch(2);


//first query for the attributes that could match
attributeList = queryAttributes(attributeMatch);

switch(attributeList.size()){

  case 0 :
    //none nothing more to do
    context.sendMessage("Sorry, no attributes
       for$attributeMatch found");
    break;

  case 1..5:
    //reasonable amount found,
    querying for values
    resultSetList = queryValues(subjectMatch,
       attributeList);
    displayResults(resultSetList);
    break;
```

```
  default :
    //too many attributes found
    context.sendMessageToUser("To many
    attributes found, please be more specific.
    Found ${attributeList.each()} ");
    break;
}



//here the query for the attributes is defined
def queryAttributes (attibuteMatch){
  attributeQuery = "PREFIX rdf:
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  SELECT DISTINCT ?z
  WHERE  {?z a rdf:Property . ?z rdfs:label ?label .
  FILTER( regex(?label , '.*"+attibuteMatch+".*','i'))}";

  def attributeList = [];
  context.queryLocally(attributeQuery).each(){res->
    res.resultRows.each(){prop->
      attributeList.add(prop["z"]);
    }
  }
  attributeList;
}

//construct the value query
def queryValues(subjectMatch, attributes){
  subjectQuery = "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT DISTINCT ?value ?name ?firstname ?familyname ?s
WHERE {"
  attributes.eachWithIndex(){attribute,i->
```

```
    if(i!=0){
      subjectQuery += " UNION ";
    }
    subjectQuery += "{
    ?s ?p1 ?subjectpattern.
    ?s <"+attribute+ "> ?value .
    FILTER regex(?subjectpattern, '.*" + subjectMatch+".*', 'i').
    OPTIONAL {?s foaf:name ?name}.
    OPTIONAL {?s foaf:firstName ?firstname}.
    OPTIONAL { ?s foaf:surname ?familyname}} ";
  }
  subjectQuery += " }";
  context.queryForTable(subjectQuery,true,null,null)
}


//display the result
def displayResults(resultSetList){
  counter = 0;
  resultSetList.each(){p2pResult->
    p2pResult.each(){res->
      if (res.getResultCount() > 0) {
  context.sendMessageToUser(
  "From store " + res.getStoreName() + ":")

  resRows = res.getResultRows();
  log.info(resRows);
  log.info(resRows.class);

  for(row in resRows){
    log.info(row["value"]);
    context.sendMessageToUser(
    " * "+row["value"] + " (" + row["name"] + ")")
    counter++;
  }
```

```
      }
    }
  }
  if(counter == 0) {
    context.sendMessageToUser(
    "Sorry, I found no " + attributeList +
    " related to " + subjectMatch);
  }
}
```

A.2 Use Case 1.3

Implementation of use case 1.3, taken from the file *foafGrouMember.groovy*.

```
////////////////////////////////////////////
/// Who is member of *
////////////////////////////////////////////
// fetching the log
log = context.getLog()



// query for member names
memberQuery =   "PREFIX foaf: <http://xmlns.com/foaf/0.1/> " +
                "SELECT DISTINCT ?name WHERE {"+
      "?s ?p1 ?subjectpattern. "+
      "?s foaf:member ?person. " +
      "?person foaf:name ?name."+
    "FILTER regex(" +
      "?subjectpattern, "+
      " '.*"+context.getWildcard(1)+".*', 'i'" +
      " ).}"
log.info(memberQuery)
//context.sendMessageToUser(documentQuery)

counter = 0;
```

```
members = context.query(memberQuery,true,null,null)
members.each(){res->
  if(res.errors==null||res.errors.empty){
    counter = counter + res.getResultCount()
    if (res.getResultCount() > 0)
      context.sendMessageToUser(
        "From " + res.getStoreName() + ":")
    res.getResultRows().each(){
      context.sendMessageToUser(" * "+it["name"])
    }
  }else{
    context.sendMessageToUser(
      "Some error occured: " + res.errors);
  }
}
if(counter == 0) context.sendMessageToUser(
  "Sorry, I can't answer your question.")
```