Triple Client

WonderWeb: Ontology Infrastructure for the Semantic Web

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

[1]University of Karlsruhe
Institute AIFB
D-76128 Karlsruhe
email: {lastname}@aifb.uni-karlsruhe.de

[2]FZI - Research Center for Information Technologies
Haid-und-Neu-Strasse 10-14
D-76131 Karlsruhe
email: {lastname}@fzi.de

Identifier	Del 8
Class	Deliverable
Version	1.0
Date	31-06-2003
Status	Final
Distribution	Internal
Lead Partner	AIFB

WonderWeb Project

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

For further information about WonderWeb, please contact the project co-ordinator:

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

Contents

Executive Summary

This deliverable describes the integration of the RDF Triple engine into KAON SERVER. We present the functionality offered by the demonstrator and the architectural considerations underlying the implementation.

1 Introduction

As showcase for the suitability of the component-based architecture of KAON SERVER, the objectives of the WonderWeb projects state, that we will connect and adapt several existing services. This deliverable shows such an adaption for an existing RDF triple engine - aidministrator Sesame.

One possible use case for an connected RDF triple engine component is providing storage and query facilites. For example, the foundational ontologies constructed within work package 3 or industrial ontologies constructed within work package 5 could be stored within the server. In order to achieve these objectives the server is a client to the triple store, who provides the above mentioned services.

Today, several RDF triple engines are available, e.g. aidministrator's Sesame[1], Hewlett Packards Jena Joseki engine[2], RDFDB[3], TRIPLE[4] or RSSDB[5]. For our purpose we selected aidministrator Sesame due to several reasons:

- it is actively maintained and well supported, for example its interpretation of the RDF model theory is up-to-date with the latest W3C recommendation.
- it is the basis for OntoDiff, the implementation of versioning framework as described in deliverable D20 of work package 4.1
- the system is scalable and robust
- offering a triple client to Sesame is expected to find a wide user area, for example within the EU IST project SWAP.

This deliverable is structured as follows. Section **??** gives a high-level overview over the Sesame system. Section 3 presents several integration scenarios and discusses their individual benefits and drawbacks. Section 4 briefly discusses how individual scenarios are implemented using the technical means offered by the hosting application server - KAON SERVER [**?**]. We conclude in Section 5 summarizing our results and giving an outlook to possible future work.

2 Description of Sesame

This Section illustrates which functionality is provided by Sesame to clients such as KAON SERVER.

2.1 Functionality offered to clients

The actions that can be performed by a Sesame server are:

[1]http://sesame.aidministrator.nl
[2]http://www.joseki.org
[3]http://www.guha.com/rdfdb/
[4]http://triple.semanticweb.org/
[5]http://139.91.183.30:9090/RDF/

- Requesting a list of repositories that are available to a specific user.
- Evaluating query is different query languages (currently RQL, RDQL, SeRQL) on a specific repository.
- Extracting the contents of a specific repository in some RDF format. Ontology and data can be extracted separately.
- Adding data to a repository.
- Removing specific statements from a repository.
- Clearing a repository.

Notably, users cannot create new repositories programmatically. This can only be done via the system configuration of a Sesame server. It is clearly an administrator's task, since it may either involve to create an appropriate database within a database server external to Sesame or the specification of a local filename to achieve the storage of data. Eventually clients, such as KAON SERVER, cannot take full control over Sesame.

2.2 Interacting with the system

Clients can perform the available actions using a variety of possibilities. On the one hand actions can be performed using HTTP calls to the server. Implementation-wise this interaction with clients is typically achieved via dedicated Servlets that take user requests and invoke actions within the Sesame server.

The results of the actions are exchanged using specialized XML vocabularies. Hence, the interpretation of the XML results has to be performed on the client side. To leverage users from this task a client-side library is available, which encapsules the interaction with the server, viz. Servlets ”living at a certain url”, into standard Java method calls and returns Java objects as the result of method invocations.

On the other hand, it is possible to interact with the system using two distributed object protocols (SOAP and RMI) which encapsule the results of the above mentioned actions into appropriate objects. However, this kind of interaction is only possible if these communication mechanisms have been configured explicitly by the Sesame administrator.

Notably, the RMI and SOAP interaction offers the same functionality as the HTTP-based interaction. Unfortunately clients cannot simply switch between the different communication means in practise, since no common set of interface structures exist. A change in communication protocol therefore necessitates to change the client source code itself. Notably, this situation could be improved via connecting to KAON SERVER.

All actions are performed relative to the privileges assigned to a user. In case of the HTTP interaction, user authentication data is transmitted with each requests. In case of using either the client-side library or object protocols the user authentication has to be set separately by previous calls to special methods dedicated to authentication.

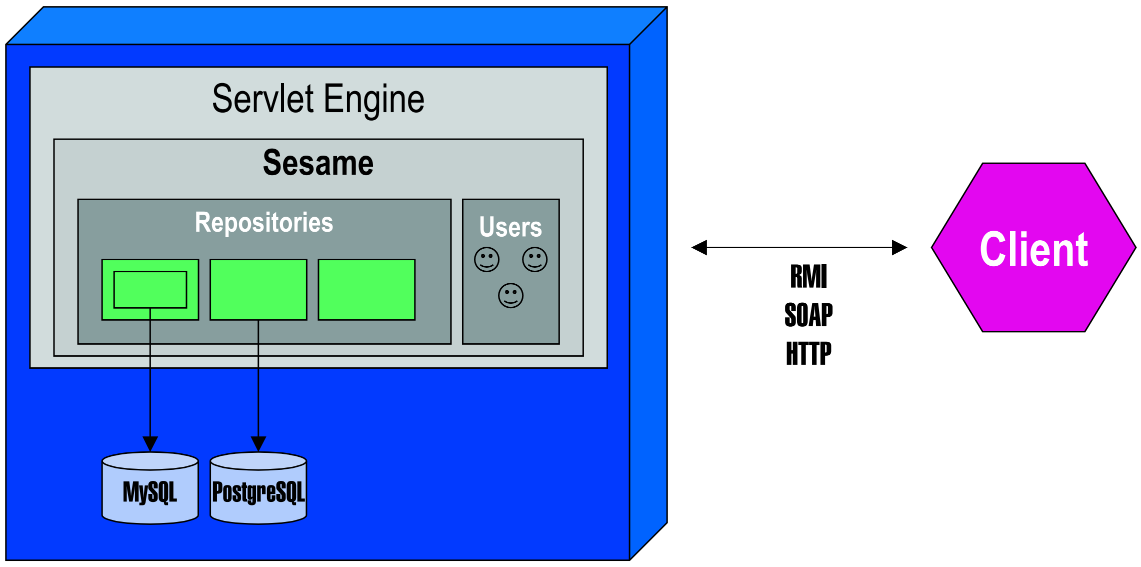

Figure 1: Sesame Architecture

2.3 Sesame Architecture

The Sesame system is structured as depicted in Figure 1. Sesame itself operates within a Servlet container that is provided by a Servlet engine such as Apache Tomcat[6]. The Servlet engine itself offers clients to connect to its hosted services via the HTTP protocol. Services are typically hosted at URLs that are relative to the URL of the Servlet engine itself. For example, Sesame can typically be reached at *servlet-engine-url*/Sesame.

2.3.1 Repositories

A running Sesame server can be configured to offer access to several RDF repositories. Currently, several types of RDF repositories are offered, each having individual distinguishing properties. Sesame can store data in main-memory (optionally with a dump to file), or it can store data in a relational database. In the latter case, a Sesame installation additionally requires the presence of a DBMS with a JDBC-driver. Currently, PostgreSQL, MySQL and Oracle 9i databases are supported.

Common features Each repository within a Sesame system declares an id and title. The id is how the repository will be accessed within Sesame. All client access to a repository is based on this identifier.

SAIL All repositories are so-called SAILs. This means that they implement certain interfaces which form a common level of abstraction between individual repositories. These interfaces constitute the fundamental part of the Sesame Storage And Inference Layer (SAIL).

[6]http://jakarta.apache.org/tomcat/

Repositories can be stacked, viz. certain repositories can act on top of other basic repositories and implement certain features for other repositories. For example, one can stack a SyncRdfSchemaRepository on top of a RdfSchemaRepository. Within this configuration the SyncRdfSchemaRepository handles concurrent access issues on behalf of the RdfSchemaRepository, which would otherwise behave unpredictably when several users access the repository simultaneously.

Sesame currently offers three types of basic repositories.

- Database RDF Repository a non-inferencing driver for relational database storage.
- VersioningSail an inferencing driver for relational database storage that supports change tracking.
- In-Memory RDF Repository: a non-inferencing driver for storage in main memory.

Database RDF Repositories SAILs working on relational databases require a number of parameters:

- jdbcDriver identifies the JDBC (Java Data Base Connectivity) driver that is to be used to access the database.
- jdbcUrl identifies the location of the database through a URI. The precise syntax of this URL is specific to each DBMS. Note that this identifier used for communication with a database and has nothing to do with the Sesame repository id.
- user identifies a username with which the Sesame system can access the database. This must therefore be a user which is known to the DBMS, and which has been granted access rights.
- password identifies a password with which Sesame can access the database. This must therefore be a password that matches the username configured in the user parameter .

In-Memory RDF Repository The in-memory RDF repository takes two *optional* parameters:

- file specifies a file in which the in-memory repository stores its contents on local disk. This file is automatically saved and reloaded on (re)start of the server.
- compressFile specifies whether the file used for storage should be compressed with gzip.

2.3.2 Transactions and Versioning

Certain variants of SAIL repositories support transactional processing of data with the usual ACID-properties. To achieve this transactions have to be started explicitly. To ensure that changes are actually performed transactions have to be commited explicitly as well. A dedicated method for aborting a transaction is not available, instead a new transaction is created. This way all changes specified for the previous transaction are lost.

Versioning of data is available in a certain variant of a SAIL repository, which add the possibility of handling multiple states of a repository. Different states of a repository can be compared and differences can be computed.

3 Integration Scenarios

This section discusses different possibilities to implement the triple client, viz. KAON SERVER being a client of Sesame. We refer to the KAON SERVER / Sesame interaction as integration, since it will appear to clients of KAON SERVER that Sesame is actually an integral part of KAON SERVER.

3.1 Integration Goal

The goal of the integration is offering clients of KAON SERVER seamless access to the functionality offered by Sesame. In principle users should be able to work with a Sesame component in the same way they are used to work with a dedicated Sesame server. This means, that the interfaces established within the Sesame system are used as the basis of interacting with the integrated Sesame system.

3.2 Integration Prerequisites

Naturally, the differences cannot be completely eliminated. While users of a Sesame server must at least know the URL of the server, users of a Sesame component that is embedded into a KAON SERVER must at least know the URL of the KAON SERVER and the name of the Sesame component. The latter can be acquired dynamically using the registry mechanisms available in KAON SERVER to enable the interaction with the component.

3.3 Interacting with the integrated system

Clients will be able to interact with the integrated system using a client API. This API tries to faithfully reflect one of the three different client APIs of Sesame. We selected to mirror the structure of the Sesame client API that wraps HTTP-based communication with the system. In the future, we will provide a wrapper of the API towards the RDF interfaces established within the KAON project.

Clients can search for deployed Sesame components via the registry of the KAON SERVER. The embedded component will offer the same functionality as a stand-alone Sesame system.

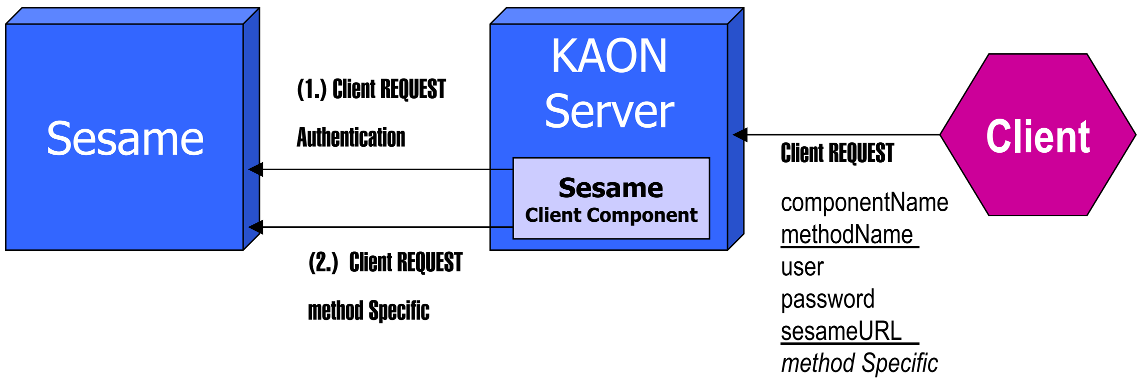

Figure 2: Generic Proxy Scenario

3.4 Generic Proxy

Figure 2, which shows how Clients (of the KAON SERVER), the KAON SERVER and a Sesame server interact if a generic proxy integration is sought[7]. The KAON SERVER offers a Sesame client in form of a proxy component in this scenario. This proxy component is able to connect to any stand alone Sesame server. All client requests issued to the KAON SERVER are forwarded to the Sesame server. The responses of a Sesame server are forwarded back to the client who initiated the request.

A generic proxy authenticates does not have any state. It connects to a Sesame server on a per request basis, hence with the URL of the Sesame server and user data as to be transmitted with every method invocation of a client. This allows to connect to different Sesame servers with the same component.

The simplicity of this solution has the drawback that the KAON SERVER has no control over the state of the Sesame server whatsoever. Additionally it requires the installation of a separate Sesame server. Another serious drawback is that we cannot faithfully reflect the Sesame client API and have to augment every function with the parameters required to capture the server URL and authentication data. In case of the RMI and SOAP based communication this solution also leads to increased network traffic, since every request issued to KAON SERVER is translated to two requests on the Sesame server. The first request will bind to a certain sesame server (and authenticate the client). The second request calls the requested functionality.

3.5 Multiple Proxies

A slight derivation of the generic proxy scenario is the multiple proxy scenario (cf. Figure 3). While the generic proxy solution requires to transmit the URL of the Sesame and user credentials with every request, the multiple proxy solution binds to an existing server using a fixed (but mutable) set of parameters at the time of the creation of the component. Hereby, a stateful proxy is generated, who could be able to offer additional services such as request pooling and/or caching.

[7]All requests are depicted by arrows, the labels of the arrows specify the parameters that have to be transmitted with each request

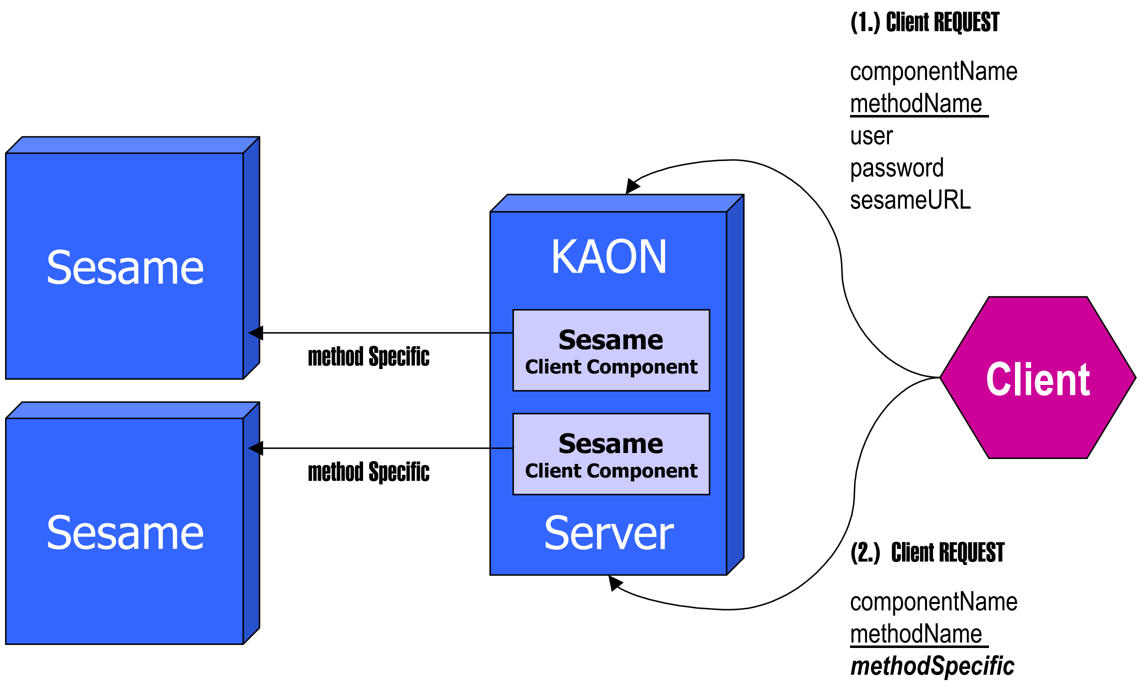

Figure 3: Multiple Proxies Scenario

Like the generic proxy this form of a Sesame client can connect to arbitrary existing Sesame servers. We can faithfully reflect the Sesame client API, since we adopt a similar stateful solution.

If a running Sesame server already exists[8], this scenario is the simplest migration path towards KAON SERVER-based solution for application clients, since no transfer of existing data and user accounts towards a new server is required. However, it is impossible for the KAON SERVER to control the creation, configuration, starting and stopping of the Sesame system itself.

Also, existing applications based on the Sesame server can be migrated incrementally and easily. Naturally, the access interface to the KAON SERVER Sesame client resembles the direct Sesame client API.

3.6 Embedded Servlet engine

A further possibility for integration is the integration of a Servlet engine into KAON SERVER (cf. Figure 4). This requires to turn the Servlet engine into a component, viz. a JMX wrapper for a Servlet engine has to be developed.

In consequence, Sesame itself could be deployed inside the Servlet engine of KAON SERVER. In turn, KAON SERVER could connect as a client to the Sesame server hosted by itself. This approach has the benefit that only users require one application server running on a machine for both Sesame and KAON SERVER.

The reader may note that Sesame could still require additional components such as

[8]ideally populated with data and user settings

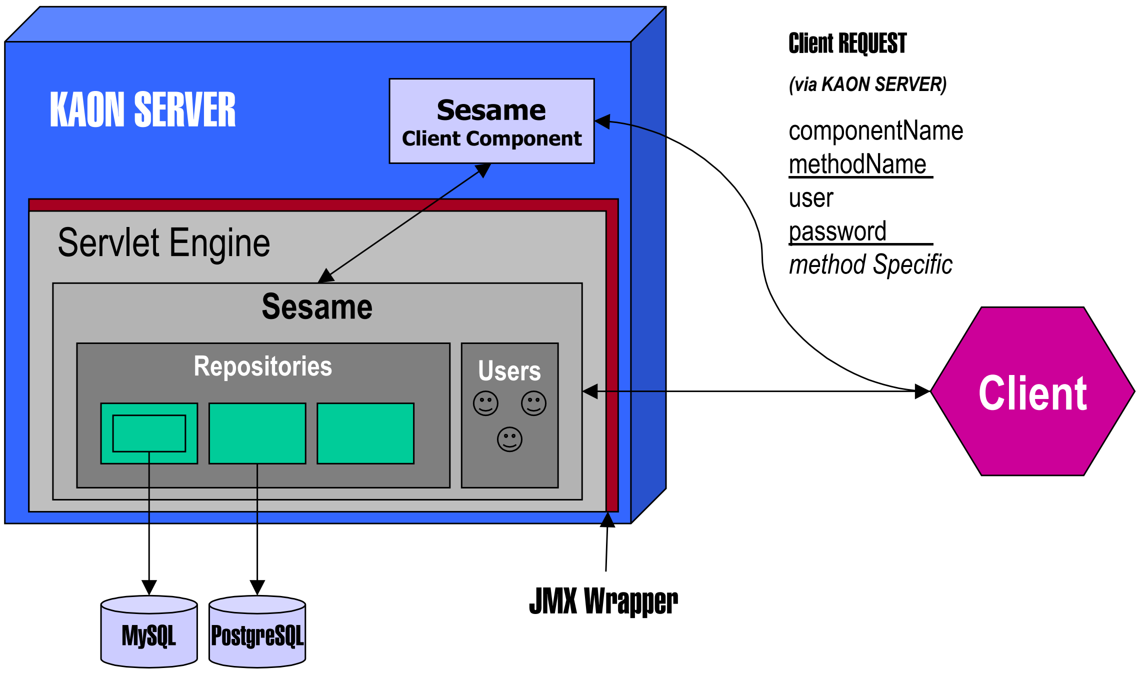

Figure 4: Embedded Servlet engine

databases which are themselves outside of the control of the KAON SERVER. A major drawback with this approach is that users searching for components of the KAON SERVER would not find Sesame (or any other web application hosted by the Servlet engine), but only the Servlet engine itself. Hence, it is impossible for the KAON SERVER to control the creation, starting and stopping of the Sesame system itself. It can only exert this control upon the Servlet engine. Secondly, this scenario must be combined together with the proxy scenario by hosting a proxy component within the KAON SERVER that forwards requests to the Sesame server living within the own Servlet engine.

However, a benefit of this approach is that the Sesame system itself does not require any modification. Clients can additionally connect to a running Sesame servlet within the KAON SERVER system as usual (via HTTP, RMI, SOAP). The major drawback of the approach is that a Sesame has to be installed from scratch and existing data has to be migrated to the new server.

3.7 Sesame component

To overcome the problem of not being able to control the Sesame server itself, we could implement a Sesame client by turning the existing Sesame code into a deployable component within KAON SERVER (cf. Figure 5). This involves the implementation of a JMX wrapper.

The main job of the wrapper beyond making the Sesame system a managable component within the KAON SERVER is to offer access to the Sesame component. The wrapper replaces the previously existing means of interaction with Sesame, which are primarily based on servlets. Since no servlet engine is available in the KAON SERVER the un-

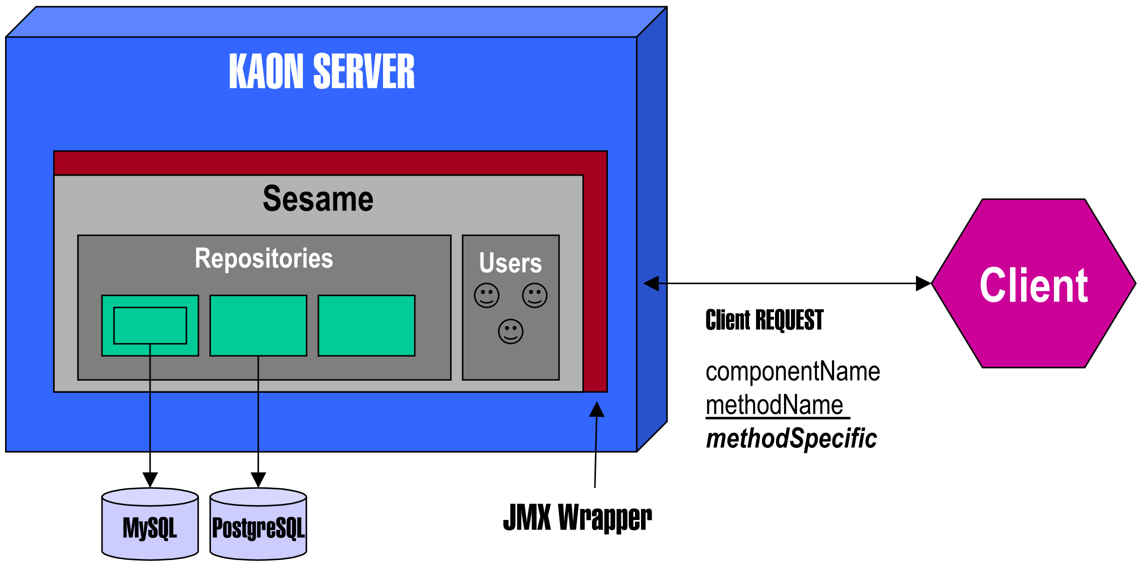

Figure 5: Embedded Sesame Component

derlying infrastructure for the HTTP-based communication is not available and has to be replaced.

The reader may note that a Sesame component may still rely on databases which are outside of the manageable realm of a KAON SERVER. This dependency arises if database-backed repositories are used within Sesame. The major drawback of this solution is that non-trivial amounts of development time has to be spent with no increased functionality.

3.8 Embedded SAIL repositories

A final option for the integration of Sesame into the KAON SERVER is integrating all Sesame repositories individually as separate components into the system. This has the benefit that individual repositories are not any longer managed by Sesame itself but by KAON SERVER instead. Naturally, picking out parts of a running system is quite difficult, since many dependencies to other parts may exist.

For example, the authentication to individual SAIL repositories relies on a specialized part of Sesame, which handles user management. The wrapper code that makes individual repositories manageable components within KAON SERVER has to implement methods for all actions that were previously offered by the Sesame system.

The reader may note, that a request for available repositories is now enabled by the KAON SERVER registry itself, instead of the Sesame system. As a result of this organization, the SAIL stack within Sesame is re-implemented within KAON SERVER using the means for expressing the dependency of components.

However, this scenario leaves our basic intention of developing a Sesame client, instead we would develop a client to different SAILS, i.e. Sesame subsystems. This solution additionally has the danger that our implementation results in a temporary solution. This is due to the fact that we would implement against interfaces which are not guaranteed to

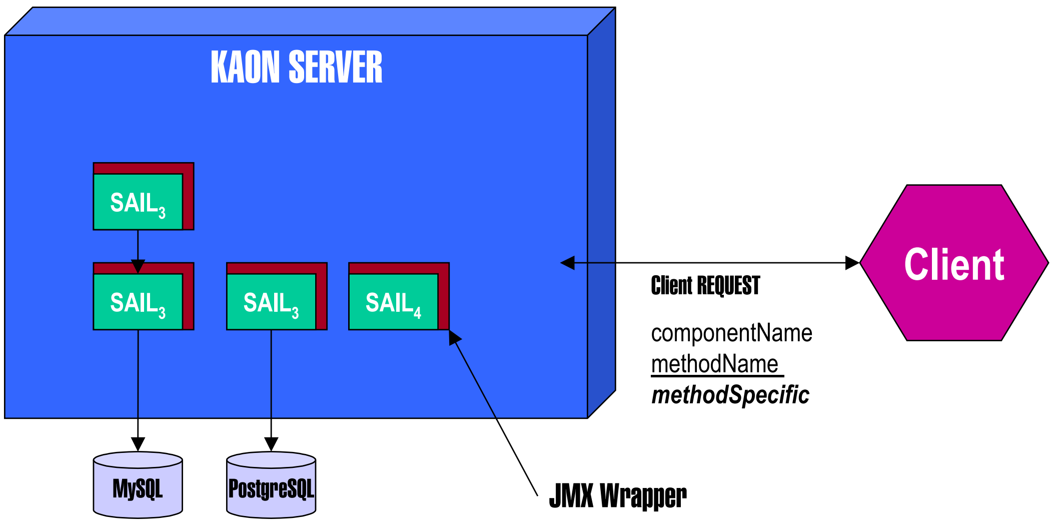

Figure 6: Embedded SAIL repositories components

evolve in a coordinated and stable manner.

4 Implementation

We decided to implement the multiple proxies scenario due to the limitations associated with the other scenarios. In a sense, this makes KAON SERVER a real client of a Sesame server. Our interface resembles the Sesame client API for HTTP-based communication as far as possible.

We could not implement several methods specified within this API since they required to pass non-serializable objects such as listeners for administrative actions and query results. Additionally we generalized the API a little bit to accept Strings as parameters where Sesame specific objects are expected, e.g. with the add Statements method which took Resources and Values as parameters.

In our ongoing work we are wrapping the Sesame client to the RDF API established in the KAON project. This allows the seamless usage of Sesame as a RDF component in KAON. It also demonstrates the usage of dependencies within components within the KAON SERVER.

5 Conclusion

We presented the triple client, which realizes a three-tier infrastructure, viz. it allows KAON SERVER to connect to remote Sesame servers on behalf of clients of the KAON SERVER. The implementation clearly shows the limitations of KAON SERVER when existing components have to be adapted.

- Usage of interfaces is quasi mandatory to achieve that components which are hosted by the server can be used instead of existing classes. If this is not done, clients will always have to be rewritten to deal with the updated API.
- A component has to be constructed for remote communication, viz. all parameters and results passed from and to the KAON SERVER have to implement the Serializable interface to allow remote communication. If existing libraries, such as in this case the Sesame client API, do not enforce this for the defined classes, appropriate wrappers have to be written.
- Call back methods from the server to the client are not possible, this disallows notification structures such as Listeners.
- The existing system must allow facilities for its management in the first place to be a fully manageable by KAON SERVER. For Sesame this is not the case, since they administration of Sesame is based on configuration files which are external (with respect to control) to the Sesame system itself.
- Dependencies to external components, such as databases can not be removed (or managed) by KAON SERVER without rewriting the code of the library itself.

Nevertheless, the triple client achieves a higher level of abstraction, in particular it removes the dependency of Sesame clients to the underlying communication protocol used. It enriches the spectrum of functionality offered by the server with several query languages for RDF data and gives persistence and versioning to RDF data. The major benefit of the triple client is flexibility. For example, it allows clients to chooses between different providers of RDF data, e.g. move from file-based persistence such as attained by the KAON RDF implementation to a database-based persistence with concurrency control.