Reinhard Dietl

A Reference Architecture for Providing Latent Semantic Analysis Applications in Distributed Systems. Diploma Thesis

Working Paper (Published)

A Reference Architecture for Providing Latent Semantic Analysis Applications in Distributed Systems

Reinhard Dietl

WIRTSCHAFTSUNIVERSITÄT WIEN

DIPLOMARBEIT

Titel der Diplomarbeit:
Eine Referenzarchitektur für das Anbieten von Anwendungen der latenten semantischen Analyse in verteilten Systemen

Englischer Titel der Diplomarbeit:
A Reference Architecture for Providing Latent Semantic Analysis Applications in Distributed Systems

Verfasser:	Reinhard Dietl
Matrikel-Nr.:	0550627
Textsprache:	Englisch
Betreuer:	Univ. Prof. Dipl.-Ing. Dr. techn. Kurt Hornik
Universität / Institut:	Department of Statistics and Mathematics
	Wirtschaftsuniversität Wien
	Augasse 2-6
	A-1090 Wien
Jahr:	2010

Zusammenfassung

Mit zunehmender Verfügbarkeit von Speicherplatz und Rechenleistung hat latente semantische Analyse (LSA) im letzten Jahrzehnt eine starke Zunahme an Bedeutung in der Praxis erfahren. Diese Diplomarbeit strebt die Entwicklung einer Referenzarchitektur an, die genutzt werden kann, um LSA-basierte Anwendungen in verteilten Systemen zur Verfügung stellen zu können. Sie skizziert die zu Grunde liegenden Probleme der Erzeugung, Verarbeitung und Speicherung von großen Datenobjekten, die bei LSA Operationen entstehen, die Schwierigkeiten, die durch das Einbringen von LSA in ein verteiltes System auftreten, schlägt eine Architektur für die Softwarekomponenten, die für die Ausführung der Aufgaben benötigt werden, vor, und evaluiert die Anwendbarkeit auf reale Szenarien, inklusive der Implementierung einer Klassenraum-Anwendung als Proof-of-Concept.

Schlagworte
Latente Semantische Analyse, Service-orientierte Architektur, Framework, Verteiltes System, R, PHP

Abstract

With the increasing availability of storage and computing power, Latent Semantic Analysis (LSA) has gained more and more significance in practice over the last decade. This diploma thesis aims to develop a reference architecture which can be utilised to provide LSA based applications in a distributed system. It outlines the underlying problems of generation, processing and storage of large data objects resulting from LSA operations, the problems arising from bringing LSA into a distributed context, suggests an architecture for the software components necessary to perform the tasks, and evaluates the applicability to real world scenarios, including the implementation of a classroom scenario as a proof-of-concept.

Key Words

Latent Semantic Analysis, Service Oriented Architecture, Framework, Distributed System, R, PHP

Acknowledgements

Thanks go to my supervisor, Prof. Kurt Hornik, who suggested the topic of this diploma thesis, made a one semester internship at the Department of Statistics and Mathematics at the Vienna University of Economics and Business Administration, for intensive engagement in the subject matter, possible, and provided substantial feedback throughout the composition process of this thesis.

Thanks go to Mag. Fridolin Wild, who not only provided substantial input at the time we were colleagues at the Vienna University of Economics and Business Administration, but also far beyond this time frame and even after he moved to a different university.

Further thanks go to my colleagues from the Institute for Information Systems and New Media, especially MMag. Bernhard Hoisl and Mag. Robert Koblischke, as well as the colleagues from the LTfLL project, who contributed to this thesis with their input and feedback.

Contents

1 Introduction

The creation of this thesis was part of the author's contribution to the "Language Technology for Lifelong Learning" project (http://www.ltfll-project.org). The goal was to create a reference architecture which enables implementers of Latent Semantic Analysis applications to make their programs usable as part of a Service Oriented Architecture. As part of the thesis, the problems surrounding the exchange, storage and processing of large text corpora should be analysed, an infrastructure for hosting of LSA services on a server should be designed, and a prototype implementation of a complete LSA application example should be implemented. For this to work, the characteristics of distributed systems had to be evaluated and the specifics of LSA had to be brought into a context with them. After creating the reference model, a demo application should be implemented, and in the process, all technology decisions should be documented.

Note that parts of this diploma thesis text may be found, without quote, in internal and external documents used by the consortium of the "Language Technology for Lifelong Learning" project. However, all text passages used in this document have originated from the process of composition of this diploma thesis (unless stated otherwise), and some have been posted in the mentioned project documents unquoted, because at the time of creation of the documents this thesis was yet unpublished.

This thesis is structured as follows: Chapter 2 will give a short introduction into basics required to understand the concepts discussed in this thesis, chapter 3 shows the actual motivation of the thesis – how LSA can be used in a distributed system and what the benefits are – and chapter 4 will then show a reference model to implement LSA in a distributed system. Chapter 5 will then show how the reference model can be used to solve the problem of distributing LSA services in scenarios taken from actual contexts, leading to a proof-of-concept implementation outlined in chapter 6. A summary of the achievements and outlook to further challenges will be made in chapter 7.

1.1 Research Question

This thesis aims to give an answer to the question

> "How does a framework for LSA web services have to be designed to overcome the challenges of distributed systems?"

The following facets of the research question have to be investigated to reliably answer it:

1. Which algorithms make up the core functionality of typical LSA processes?
2. What are the challenges of distributed systems?
3. Which aspects of LSA algorithms make it difficult to use LSA in distributed systems, especially as part of web services?
4. Which best practices are utilised to solve similar problems in related application scenarios?
5. Which requirements must be demanded from a solution architecture?
6. Is the solution architecture applicable to real-world scenarios?

1.2 Definitions

This thesis makes use of some notions that have varying definitions in the relevant field of research. To ensure a correct understanding of the topics discussed, key terms will be defined in this section to avoid confusion or suspicion of incorrectness solely caused by different nomenclature.

1.2.1 Web Service

The Web Services Architecture Working Group (2004) defines that "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." Although others exist, this definition will be adhered to throughout this thesis, and notes will be placed if external works use a different understanding.

1.2.2 Distributed System

According to Coulouris et al. (2005), a distributed system is "one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages." This includes intranets, the internet, and mobile and ubiquitous computing solutions. Distributed systems are defined by their underlying fundamental and architectural models, one of which is the client-server-model used extensively throughout this thesis.

1.2.3 Service Oriented Architecture

"Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." (OASIS SOA Reference Model Technical Committee, 2006) So, the key idea is to distribute properly encapsulated functionality for use in logically separated units or even geographically different locations. The benefit of such an architecture is, among others, that each piece of functionality can be performed utilizing service providers (e.g., servers) that are specialised in executing such tasks.

"Though built on similar principles, SOA is not the same as Web services, which is a collection of technologies, such as SOAP and XML. SOA is more than a set of technologies and runs independent of any specific technologies." (Hentrich and Zdun, 2006) This means that the implementation of a service oriented architecture does not need to be, but may be based on web services. Sections 4 and 6 will describe a set of web service facilities for usage in an SOA of a natural language processing system using a reference architecture based on software patterns as suggested in Avgeriou and Zdun (2005) by looking at the architecture of the services and the logic behind them from different points of view.

1.2.4 Framework

A framework is a software library that provides basic functionality for a certain purpose, and allows it to extend this functionality, yet encourages adherence to an agreed upon structure. "When you use a framework, you reuse the main body and write the code it calls. You'll have to write operations with particular names and calling conventions, but that reduces the design decisions you have to make" (Gamma et al., 1994).

2 Required Basics

To fully comprehend the topics discussed in this thesis, basic knowledge of some related fields of science have to be known in beforehand. This chapter aims to give the reader a condensed introduction to these topics. To acquire a deeper understanding of the concepts, please refer to the quoted literature for details.

2.1 A Short Introduction to Latent Semantic Analysis

The fact that the human brain stores information in a different way than machines has led to significant shortages in the interaction between the former and the latter throughout the age of electronic data processing. Latent Semantic Analysis (LSA) emerged from the need for a more content-aware indexing method in information retrieval (Landauer and Dumais, 1997), which led to the invention of Latent Semantic Indexing (LSI) and later LSA.

2.1.1 Mathematical foundation

LSA is a mathematical model which tries to process textual data (which is the common data format for communication between human and machine) and extract from it what man perceives as a "meaning" of something stored in the mind. Concepts are "learned" by simulating a human brain acquiring knowledge about a language from experience. It relies on the basic assumption that "meaning" can be derived from the frequency of terms occurring in a certain context (e.g., a text about a certain topic) and mathematically condenses these semantics into orthogonal dimensions (Landauer, 2007).

The task of decomposition is tackled by counting the number of occurrences of words, phrases or similar identifiers (in LSA referred to as "terms") within a portion of text, which may be a sentence, a paragraph or a whole text ("document"). A "document-term-matrix" is generated, having a row for each term and a column for each scanned document, with the term count at their intersections (Landauer and Dumais, 1997).

2.1.2 The latent semantic space

Initially, the representation of the semantic information is one-dimensional: an integer representing the word count of every document-term-combination. After a process called "Singular Value Decomposition" (SVD, see Martin and Berry (2007)), the LSA space is an object consisting of three matrices U, Σ and V, representing the higher-dimensional abstraction of the document-term-matrix. U and V are the term- and document-matrix, respectively, and Σ is a diagonal matrix containing a set of dimensional coefficients, which can be used to locate elements of U and V in the respective dimensions using multiplication.

A very simple example of an LSA space can be seen in Figure 2.1 on page 7. As stated in Dietl (2009), the SVD has been generated from some paragraphs of text. Still, the resulting data object has a notable size. The amount of data produced from longer documents is much larger and requires adequate facilities for storage and processing.

The processes of generation and storage described above are basic operations performed with LSA spaces. Other such operations are, according to Berry et al. (1995), the update of the term-matrix U, the document-matrix V, as well as the weights used to normalise the occurrence counts of the terms in the documents. As noted in the introductory section, a goal during development of the framework was

```
         [,1]  [,2]  [,3]  [,4]  [,5]
[1,]     14.6     0     0     0     0
[2,]        0  14.2     0     0     0
[3,]        0     0  10.8     0     0
[4,]        0     0     0  9.20     0
[5,]        0     0     0     0  8.70
```

```
           1     10     11      12     13       14     15      2       3       4      5       6       7        8       9
[1,] -0.0435 -0.085 -0.274 -0.0379 -0.775 -1.4e-17 -0.538 -0.051 -0.0435 -0.0340 -0.138 -0.0081 -0.0091 -0.00055 -0.0072
[2,] -0.0085 -0.140  0.207 -0.0074  0.507  1.1e-16 -0.821 -0.010 -0.0085 -0.0556  0.051  0.0032  0.0036 -0.00103 -0.0126
[3,]  0.0413 -0.012 -0.927  0.0306  0.306  9.0e-17 -0.032  0.064  0.0413 -0.0046  0.191  0.0230  0.0290 -0.00033 -0.0022
[4,]  0.1794 -0.037  0.147  0.1075 -0.206 -1.4e-16 -0.048  0.541  0.1794 -0.0129  0.713  0.1364  0.1910 -0.00250 -0.0113
[5,]  0.0817 -0.016 -0.037  0.0431  0.064 -2.8e-16 -0.014  0.791  0.0817 -0.0052 -0.549 -0.1270 -0.1871 -0.00151 -0.0059
```

```
               [,1]     [,2]     [,3]     [,4]     [,5]
inform     -1.5e-02 -2.9e-03  1.9e-02  9.6e-02  4.6e-02
lsa        -2.3e-01 -4.4e-02  1.2e-01  3.4e-01  1.2e-01
retriev    -1.5e-02 -2.9e-03  1.9e-02  9.6e-02  4.6e-02
cite       -1.7e-01 -2.6e-01 -1.6e-02 -3.6e-02 -1.4e-02
dimension  -2.5e-02 -4.2e-02 -5.2e-03 -2.1e-02 -9.8e-03
model      -1.5e-01 -2.4e-01 -1.5e-02 -3.6e-02 -1.4e-02
match      -2.8e-01  2.0e-01 -2.2e-01 -2.5e-02  1.2e-02
sentenc    -9.2e-02  7.2e-02 -4.2e-01  7.9e-02 -2.1e-02
set        -9.2e-02  7.2e-02 -4.2e-01  7.9e-02 -2.1e-02
string     -9.2e-02  7.2e-02 -4.2e-01  7.9e-02 -2.1e-02
structur   -9.2e-02  7.2e-02 -4.2e-01  7.9e-02 -2.1e-02
word       -2.8e-01  2.0e-01 -2.2e-01 -2.5e-02  1.2e-02
combin     -1.3e-02 -2.5e-03  1.4e-02  5.8e-02  2.4e-02
context    -2.4e-01  1.5e-01  1.8e-01  2.2e-01 -2.2e-01
expect     -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
input      -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
relev      -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
select     -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
text       -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
user       -2.6e-01  1.8e-01  1.4e-01 -1.1e-01  3.6e-02
concept    -9.2e-19 -8.1e-17  7.7e-17 -9.7e-17 -2.7e-16
statement  -9.2e-19 -8.1e-17  7.7e-17 -9.7e-17 -2.7e-16
classic    -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
evalu      -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
hofmann01  -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
perform    -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
plsa       -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
polysemi   -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
ref        -1.8e-01 -2.8e-01 -1.4e-02 -2.6e-02 -8.1e-03
abl        -1.7e-02 -3.5e-03  2.9e-02  2.9e-01  4.5e-01
human      -1.7e-02 -3.5e-03  2.9e-02  2.9e-01  4.5e-01
landauer97 -1.7e-02 -3.5e-03  2.9e-02  2.9e-01  4.5e-01
approach   -1.5e-02 -2.9e-03  1.9e-02  9.6e-02  4.6e-02
short      -1.5e-02 -2.9e-03  1.9e-02  9.6e-02  4.6e-02
matrix     -3.5e-02  1.4e-02  7.5e-02  3.8e-01 -3.3e-01
svd        -4.7e-02  1.8e-02  8.7e-02  3.8e-01 -3.1e-01
type       -4.7e-02  1.8e-02  8.7e-02  3.8e-01 -3.1e-01
origin     -3.1e-03  1.2e-03  1.3e-02  1.0e-01 -1.1e-01
space      -2.1e-03 -3.7e-03 -9.2e-04 -5.9e-03 -3.3e-03
```

R Code

```
R> textmatrix(mydir='./', minDocFreq=2, stopwords=stopwords_en, stemming=TRUE) -> tem
R> lw_bintf(tem) * gw_idf(tem) -> tem_red
R> lsa(tem_red)
```

Legend

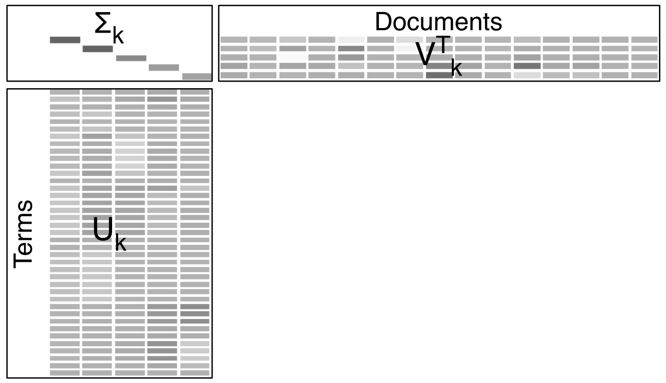


Figure 2.1: The components of an LSA space generated from a bachelor thesis text extract. This figure is taken from Dietl (2009).

that it should enable execution of such operations in a timely manner by providing a suitable infrastructure.

2.2 Considerations for Distributed Systems

This section aims to give an overview of the facets of distributed systems (for a definition, see 1.2.2) that are required to understand the basic idea behind the considerations and assumptions made in this thesis.

2.2.1 Advantages of Distributing

Adding distributed system capabilities to a computer system can add substantial overhead, potentially decreasing performance. Still, there are reasons to take this loss in favour of the advantages that can be gained. In Völter et al. (2005), the advantages of distributed systems over monolithic architectures are outlined:

- Performance and Scalability
  Often, a single system is not able to take the whole load of processing user requests alone in a time-effective way. In this case, a technique called "load balancing" can be employed to distribute workload between multiple machines.
- Fault Tolerance
  Since hardware and software are not free of faults, it is possible that due to such failure, a part of a process can not be executed, either on a particular machine, or not at all. A distributed system can provide remedy by allowing for redundancy of hardware and dynamic re-allocation of tasks in case of failure.
- Service and Client Location Independence
  Distributed systems allow for the planning of software systems without ever having to know where the participants of the system are located physically.

Advanced architectures even allow system components to join and leave the system arbitrarily.

- Maintainability and Deployment
  Centralisation of business logic is a key method to reduce maintenance cost: If clients are only used to gather input data and display results, changes to a system only have to be deployed to a relatively small number of system components.

- Security
  Access privileges to a system's services are data as well and therefore have to be maintained. The knowledge about user credentials may be used by different clients simultaneously, and may be required to be read or even administered from different locations. Furthermore, often computer systems contain sensible data or know-how, and therefore have to be locked down physically. A distributed system allows for this to happen without requiring a user to be physically present in the security zone, but still allow for usage of the system.

- Business Integration
  In a changing business world, interchange of data between companies is no longer limited to transmission over surface mail or telephone: with the rise of e-mail and the internet, a need for business systems to communicate with each other in realtime emerges. It allows for business processes that depend on information owned by different companies to be executed in real-time if this external information is provided on demand by an electronic system.

2.2.2 Challenges of Distributed Systems

According to Völter et al. (2005), "compared to traditional, non-distributed systems, additional challenges arise when engineering distributed systems". These problems are a key consideration when choosing to make a system distributed as their solution often "adds complexity, concurrency, inefficiency and other potential problems to your application".

- Network Latency:
  A remote invocation in a distributed system takes considerably more time than an invocation in a non-distributed system.
- Predictability:
  The time it takes to invoke an operation differs from invocation to invocation, because it depends on the network load and other parameters. These latencies also appear in non-distributed systems, but only in the form of rather constant and therefore predictable amounts.
- Concurrency:
  Problems might arise from the fact that there is real concurrency in a distributed system. Orchestration of parallel execution of tasks can require sophisticated solutions for things as simple as a common time reference.
- Scalability:
  Since different parts of a system are more or less independent systems themselves, it is not always possible to know in advance how high the communication load is going to be at a certain time.
- Partial Failure:
  In distributed systems, only a part of the overall system might fail, and the rest of the system should still fulfil the overall system task.

2.3 Computers' Roles in the Subprocesses of LSA

The algorithms involved in LSA processes are complex and require a lot of calculation, which, in most realistic scenarios, cannot be performed by humans. With the increasing amount of computation power available with advancing microprocessor development, computers are the tool of choice for calculation in LSA. This section is going to outline the individual sub-processes of LSA and show the state-of-the-art methods used in current computer-aided implementations.

2.3.1 Collection of Text Data

Text has been the form of knowledge storage for ages. With the rise of computers in the last decades, ways to store text in binary format have been developed, and in recent years, digital text has become ubiquitous. For use of text in distributed systems, standards like the Hypertext Transfer Protocol (HTTP) have been developed to allow for transport of hyptertext data over network.

To perform automatic processing of text, first, the data has to be collected. Computers collect text data in digital storage formats from local or remote data sources. For example, the `tm` R-package described in Feinerer (2008) has an abstract `source` class which abstracts the import process of gathering text data from a directory, CSV files, Gmane RSS feeds, PDF documents and test collections like the Reuters 21578. Depending on the input format, the text is preprocessed and converted to a text database, preserving as much data as possible while fitting it into a common format, which can in turn be stored as a text file.

2.3.2 Stemming and Stopwords

Many languages spoken throughout the world feature the characteristic of "inflection", which means that words in their "basic" forms are modified to reflect the context in which they are used. "Stemming" is a technique to reduce words modified in that way to a common basic form (which does not necessarily have to be the initial "basic" form). Computers are well suited for word stemming as in most cases, inflection is based on algorithmic rules which can be used to reconstruct the word stem. If inflection has caused a word to be modified beyond algorithmic reconstruction (e.g. irregular verbs), dictionaries can be used to restore word stems.

Stopwords are words that occur very frequently in a language. Compared to words with high frequency in a specific domain (which LSA can identify and eliminate if necessary), stopwords are numerous in all texts of a language. For this reason, so-called “stopword lists” help to strip those common words from texts to isolate terms that have potential do distinguish texts from each other. Stopword lists for common languages are often provided with text mining frameworks (Feinerer, 2008).

2.3.3 Decomposition to Text Vectors and Matrix Composition

Foundation for the LSA process is the presence of a term-document-matrix (see 2.1), which is an association of text vectors. Each vector is a “bag of words”, meaning that the number of words in the underlying document is calculated without regards to order.

The text mining framework described in Feinerer (2008) provides a method that takes a text database as input, and takes parameters about what level of granularity should be parsed. For example, instead of words, the user might want phrases or sentences as “terms” in the matrix. In that case, `tm` uses OpenNLP’s facilities for tokenisation. The computer then uses sophisticated algorithms to detect grammar and part-of-speech structures to determine the correct phrases (a task that, despite complex mathematical demands, is still performed in a fraction of time a human would need for extracting those elements manually).

After text vectors for each document have been collected, their vocabularies are merged and they are coerced into a term-document-matrix. Note that the vast amounts of data collected from text collections puts the computer at risk to run out of memory during performance of the composition process. Mechanisms must be implemented to handle such cases.

One approach to avoid this issue is to store matrices in “sparse” formats. Due to the fact that many words do not occur in documents, document-term-matrices often contain lots of zero values and are therefore called “sparse”. Instead of redundantly storing lots of zeros in a tabular format, only non-zero values are stored together with their positions within the matrix (e.g. as lists, keys or coordinates).

Another approach which many operating systems follow is to provide additional, higher-latency storage (like so called “swap” space often located on partitions on a hard disk) as a fallback medium if main memory is exceeded.

2.3.4 Weighting

As Martin and Berry (2007) outlines, importance of word occurrences is not a linear function. On the one hand, terms which occur only once in a document collection do not contribute to distinguishing concepts, and on the other hand, terms present abundantly across documents don’t either. As a result, the term counts in the document-term-matrix’s cells have to be transformed to represent the actual “importance” or “weight” (hence the term “weighting”) of a term in a document.

There are local and global weighting functions, the former transforming the value based on a pre-defined function $f(i, j)$ for term i in document j, the latter being a function $g(i)$ for term i across all documents. Computers are well suited for performing this task, because they are able to calculate these values for large matrices in a timely manner.

2.3.5 LSA Space Generation

After the underlying term-document-matrix has been generated and weighted (see above), a process called Singular Value Decomposition (SVD, see Martin and Berry (2007)) is performed to obtain matrices U, V and Σ. This mathematically complex process is well suited to be performed by computers, and as LSA is not the only field that SVD can be applied in, there are standard packages for many platforms to perform SVD, some of which are:

- LAPACK (http://www.netlib.org/lapack/) is a library written in Fotran90, licensed under BSD licence, which allows for a wide range of linear algebra operations, one of which is SVD. LAPACK packages are available for many programming languages and mathematical frameworks.
- GSL (http://www.gnu.org/software/gsl/) is a C library and part of the GNU project (and therefore published under GPL). It provides functions for a wide range of numerical calculations, one of which is SVD.
- SVDLIBC (http://tedlab.mit.edu/~dr/SVDLIBC/) is a C-rewrite of the Fortran77 implementation SVDPACK. It is a library specialised in performing SVD calculations, with support for sparse and dense matrices (see 2.3.3).

Resulting LSA spaces are often represented by objects or lists with entries for the respective matrices. *Sigma* may be stored as a vector representation of the diagonal matrix.

2.3.6 Performance of LSA Logic

After the foundations for LSA have been laid with the generation of a latent semantic space, users can start to perform LSA logic with it. Different levels of abstraction can be realised, either providing LSA as a library in a programming language (see for example Wild (2009), which references the "lsa" package for the scripting language R) for the user to write his own logic, or provide pre-defined logic as part of an application. Both approaches are backed with a vast landscape of programming facilities for computers and software development models to aid developers give the user the desired usage experience.

3 LSA in a Distributed System

As already mentioned in the introduction, the goal of this thesis is to develop a reference architecture to make LSA usable in a distributed system. In this chapter, the context of distributed system aspects outlined in the previous chapter will be brought into context with LSA, showing advantages, challenges, and current approaches of providing LSA as a service component.

3.1 How LSA Can Take Advantage of Being Distributed

The goal of this section is to outline reasons for introducing LSA into a distributed architecture, possibly introducing overhead as a result. The examination will be conducted with the advantages and challenges outlined in 2.2 as an orientation, showing the relevance of each of the aspects for LSA.

3.1.1 Performance and Scalability

LSA is facing a significant obstacle: when problems grow beyond a certain size, the memory consumed to calculate the SVD might exceed the memory available on the calculating computer system. Regarding the mere reduction of memory consumption during LSA calculation, Kontostathis et al. (2005) provides a methodology that enables significant sparsification of T_k and $S_k D_k$ matrices without significant loss of retrieval performance.

However, if reduction of initial complexity does not remedy the problem, a way must be found to split up calculations. In this context, Martin et al. (2007) suggest two different concepts for calculation of LSA problems (especially the SVD):

1. In-core calculation: calculations are performed with the whole data (the "core", i.e. parameters as well as generated intermediate results) required for the calculations held in a memory mechanism with every bit of data randomly accessible by the calculating machine at any time (in most current computer systems, RAM and swap space).
2. Out-of-core calculation: calculations are performed with only a pre-defined subset of the core available to each calculating machine. The approach allows for batch-processing of a large problem on a single machine, or, with appropriate orchestration, even the distributed calculation of the result.

To enable distributed calculations, some modifications have to be made to the monolithic LSA approach, especially in the process of SVD calculation. One such modification is outlined in Martin et al. (2007): it is shown how the often utilised Lanczos algorithm is modified to allow for out-of-core reorthogonalisation of each vector with only a subset of data available.

In a different approach, Vigna (2008) describes a technique called "index interpolation" and shows that this method enables calculation of problems with multiple billions of non-zero entries in the document-term-matrix. This technique enables the deconstruction of the problem into batches, which can then be stored in memory or secondary storage. Furthermore, the paper shows how these batches can be processed in a distributed computing environment, and how, in this fashion, problems of unprecedented size can be computed in finite time.

The modified algorithms mentioned above allow for LSA not only to be executed even if the problem as a whole exceeds the available memory by splitting it into batches, but also, given a suitably fast method of transfer of intermediate results between computer systems, may be used in a distributed system to process batches on multiply machines, and can therefore benefit from increased processing power and available memory.

3.1.2 Service and Client Location Independence

As with most computationally intensive tasks to be introduced to a distributed architecture, LSA can profit from the independence of physical location of participants (clients and servers) in a (wide area) network. Given that the underlying implementation is compatible with a wide range of platforms, computational power can be consumed wherever it is available at any given time, allowing for profiting from economies of scale and maximising degrees of utilisation.

3.1.3 Maintainability and Deployment

Running LSA in a distributed system normally requires an approach that endorses well-encapsulated units of program code. A distributed architecture is less prone to development of single-purpose solutions, and facilitates sustainable development. In such an environment, experts on certain fields can be provided with "sandboxes", which are basic software packages that allow an expert to concentrate on the desired task and not be distracted by outside logic required to make his work run in a different context.

Distributed systems allow for centralised handling of data, enabling the use of so-called "thin clients" which are only used to gather data and display results. Software developed in such an environment is also normally suitable for automated testing mechanisms, which fosters quality assurance, resulting in increased confidentiality and flexibility.

3.1.4 Security

Although the sensibility of data held in LSA processes is not always high, they may still contain a lot of effort and know-how which must be protected from unwanted access. Maintenance of credentials, roles and permissions from a centralised location can therefore be advantageous for LSA too. Furthermore, the vast amount of computational resources consumed upon invocation of some LSA-related processes requires reliable control of execution permissions for users of a distributed LSA system to avoid over-use of available resources, leading to failure of execution for all participating parties.

3.2 Tackling the Challenges of Distributed Systems

As already outlined in the previous chapter (and further discussed in Völter et al. (2005)), providing a software system as part of a distributed system introduces overhead and has the developer face a set of completely new challenges. This section is going to show what challenges LSA processes have to face when being distributed, and will outline solution approaches.

3.2.1 Network Latency

One approach to reduce the impact of network latency is the use of "web caching", which "consists of temporarily storing resources in a fast access location, for later retrieval. [...] if the cache is closer to the client than the origin server owning the resource, the route that that the resource must traverse to reach the client is shorter, which reduces bandwidth consumption and response time" (Ceri et al., 2003) In the context of LSA, this is especially desirable for large data objects like matrices resulting from calculation of spaces. An interesting approach would be the setup of Content Delivery Networks (CDNs) if an LSA logic, used by servers around the world, that always calculates its results based on the same, yet frequently changing, space, would be able to receive the current version of the space from a server located in the close vicinity.

Another way to reduce network latency is to simply reduce the amount of data sent over the network. This can be done using compression algorithms. Compression is a process to reduce the amount of data used by a data stream, among others, by utilising the full numeric space provided by the underlying binary container, finding recurring text passages and replacing them with (shorter) synonyms and making use of arithmetic properties. The downside is that compressing and deflating data consumes computation power and therefore takes time. The amount of time used depends mostly on data size, complexity of the compression algorithm and data handling overhead.

3.2.2 Predictability

One approach to overcome the issues of lacking predictability in distributed systems is to introduce Quality of Service capabilities to an LSA service. Serhani (2008) suggests a broker (Völter et al., 2005) based QoS mechanism for distributed systems, which could make decisions on who can perform actions on a service at what time (and which credentials have to be present), and tell a requester whether a request could, at any given time, be performed within the time frame requested.

3.2.3 Concurrency

Providing multiple users with processing power simultaneously might cause each user's actual task execution time to be prolonged beyond the acceptable duration in a productive environment. One way to avoid this is to provide a server suitably scaled to perform the desired amount of requests simultaneously in due time. Unfortunately, on the one hand, single computers can not be scaled arbitrarily, and on the other hand, the exact number of peak concurrent requests may not be known in advance.

One way to solve this issue is to provide multiple machines able to perform the same task, as concurrent service providers. According to Coulouris et al. (2005), with a technique called "load balancing", requests can be distributed among multiple servers depending on their current degree of utilisation (i.e. the "load" on the server is "balanced"). More sophisticated approaches suggest that after execution of a process, the next step or even the whole processes themselves, may be moved between servers if such procedure is possible.

Also, some application scenarios (e.g. classroom applications) produce a lot of redundant processing if not treated correctly. Management of calculation time and reuse of intermediate results may reduce the load on servers and enable a fast user interaction desirable for both the client and the server.

3.2.4 Partial Failure

Since distributed computing architectures are often comprised of a number of computers large enough that they can no longer be individually monitored by humans, mechanisms must be provided that each machine on a distributed system can be monitored automatically. In case of failure, routines must be present to dynamically reallocate tasks designated for the failing component to a different machine, restoring the state of the failing element e.g. from a backup and resuming operation from the last known point of operation.

3.3 Current Approaches to Providing LSA as a Service Component

There are lots of implementations for common LSA operations (including some with support for providing these applications as a service over a network like the internet). Anyway, most of these implementations lack standard procedures to handle the challenges outlined in the sections above, as will be shown in the (not necessarily comprehensive) following subsections about the implementations:

3.3.1 GTP and GUP

The “General Text Parser” (GTP) is a software package for Unix operating systems developed by S. Howard, H. Tang, M. Berry, and D. Martin, which provides most common LSA operations via command parameters. An extension of GTP is PGTP, which allows for the distributed computation of SVDs.

A web-interface called the “GTP usability prototype” (GUP) is available at http://sourceforge.net/projects/gup/. GUP is programmed in PHP (PHP Development Team, 2010) and uses the file system to store serialised LSA objects, as well as a MySQL database (MySQL Development Team, 2010) to manage the stored data.

The main problem about GUP is that is simply a PHP interface for the various command line parameters of GTP, and not a service as defined in 1.2.3. It does not provide any workflows to solve the problems outlined in 2.2.2. Rather, according to the introduction section of the GUP documentation, it is considered to be a web-based system to facilitate testing of and experimenting with GTP.

3.3.2 Cooper

In Giesbers et al. (2007), a service-oriented approach to providing LSA as a web component has been described, which is shown in figure 3.1.

Sparse matrix handling is performed in a separate layer, and a separate SVD library is called by the main calculation routines (the “LSA engine”). A presentation layer is set on top of the engine, which acts as the interface to other components. A coordination layer (“processing environment”) is set between the LSA engine and the presentation layer, having a separate “output” converter which receives LSA output and turns it into objects readable by the processing environment.

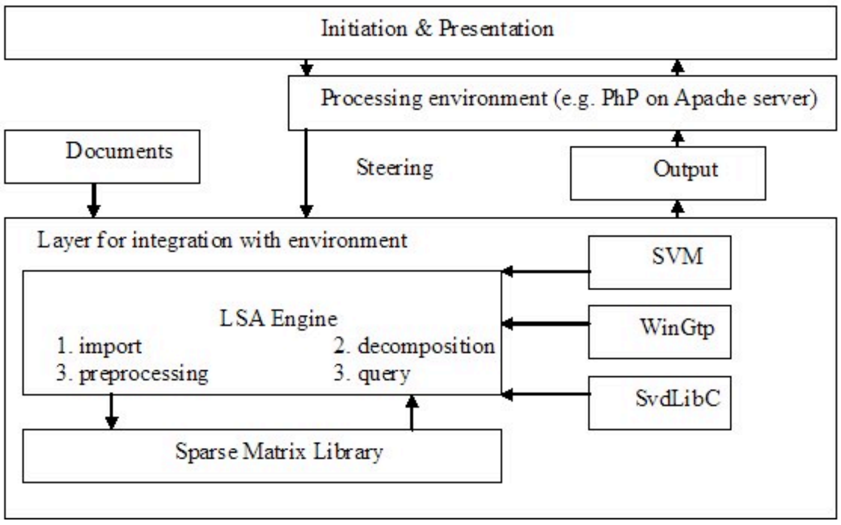


Figure 3.1: The service approach for LSA used in Cooper (Giesbers et al., 2007)

3.3.3 LSA PHP Extension

The PHPLSA software package is a function-level extension to PHP written in Object Pascal and can be found at the project's TRAC system at http://sourceforge.net/apps/trac/phplsa/. It can be compiled as a PHP extension library to provide basic LSA functionality directly within PHP code. According to the TRAC, SVD is done via AlgLib, SVDLIBC or wingtp, a decision that also depends on the operating system of the server which will host the PHP code executing LSA.

PHPLSA is a software component for the PHP scripting language, and therefore not a web service software at all. It can be considered a "processing environment" as it has been referred to in Giesbers et al. (2007). The availability of LSA functionality in a scripting language facilitates the development of web service logic, but does not provide any pre-defined workflows for handling the issues discussed in 2.2.2.

3.3.4 SOAP web service of TENCompetence Suite

A SOAP web service wrapper with similar architecture as Cooper (see 3.3.2) has been described in Kalz et al. (2009), which was used to perform placement experiments as part of the TENCompetence Suite (http://www.tencompetence.org/). However, no details about the implementation have been published concerning the criteria of decoupling and automation facilities provided by this web service implementation, as well as the concurrency behaviour and failure tolerance. Anyway, the document describes the SOAP web service as single-purpose implementation that interfaces between the low-level PHPLSA and a high-level, single-function SOAP-API, which hints that new API elements can be implemented by a web service provider. Process outputs are stored in the local file system in an "output in an easy readable non compressed format if the matrices are small enough", which hints that a serialisation process of some kind is employed to enable later retrieval of intermediary results of the process chain.

4 A Pattern-Based Reference Architecture for LSA Services

This chapter aims to find a way of how to organize the individual components of a potential LSA framework so the individual functionalities remain encapsulated properly, and can be coupled easily where appropriate.

One way to find "good" approaches for software design problems is to refer to tried-and-tested solutions called "patterns". There are multiple subclasses of patterns, depending on their level of granularity. In software design, design patterns are often used, which are a "mechanism for expressing design structures [and] identify, name, and abstract common themes in object-oriented design" (Gamma et al., 1993). More general solutions can be found in architectural patterns, which "refer to recurring solutions that solve problems at the architectural design level, and provide a common vocabulary in order to facilitate communication" (Avgeriou and Zdun, 2005). Architecture, in this context, means that after the definition of individual system components' function, the placement of the components across a network in search of "useful patterns for the distribution of data and workload" as well as "the interrelationships between the components - that is, their functional roles and the patterns of communication between them" are considered, according to Coulouris et al. (2005). The distinction between a "design pattern" and an "architectural pattern" is normally done utilizing the "size" of the patterns underlying context. Still, it is difficult to define the thresholds for such distinction (Avgeriou and Zdun, 2005).

A related method to look at the potential organisation of a software framework is to group desired functionality into a set of viewpoints. “An Architectural View is a representation of a system from the perspective of a related set of concerns” (IEEE, 2000). These concerns may be how to allocate the physical hardware, how to group services, or how interaction between components should be organised. The fact that this pattern considers a very “large” scope suggests that it is an “architectural” pattern (Avgeriou and Zdun, 2005). The following sections will show how the framework will solve the encapsulation of different functionalities by various concerns.

As a first step, a set of requirements will be derived that must be met for the architecture to be considered complete. Then, an architecture will be outlined utilising “patterns”. Finally, it will be evaluated whether the resulting architecture satisfies all the requirements.

4.1 Requirements for a Reference Architecture for LSA as a Service Component

Analysing requirements prior to implementation is an essential step to developing software. “The output of the requirements specification activity is a user-oriented, easy-to-understand, yet precise, specification, which is addressed both to the designers, who use it to understand what the application must do, and to the stakeholders, who use it to validate the adherence of the specifications to the business requirements, before proceeding with development” (Ceri et al., 2003).

Based on the knowledge of distributed systems application and its challenges up to this point, the following requirements are established for an architecture for a distributed LSA system:

4.1.1 Network Latency

1. Caching

   It must be possible to make use of caching technologies wherever the actual processes allow to do so. Especially, it must be possible to distributedly make large data objects available.

2. Compression

   All data interchanges must be shaped in a way that compression algorithms can be negotiated by the communication partners. Ideally, a message syntax that allows for effective compression should be favoured.

4.1.2 Security

The authors of Web Services Architecture Working Group (2004) state that "At this time, there are no broadly-adopted specifications for Web services security." This is not true any longer as there are standards like WS-Security which cover at least some of the aspects outlined in this section. This thesis will not attempt to create a new specification, but instead, will ensure that security-relevant aspects are shown and concrete treatment for them is suggested or at least the architecture does not pose an obstacle for implementing such.

According to Web Services Architecture Working Group (2004), the most important security mechanisms and associated required capabilities are:

1. "Security policies":

   Policies are machine-readable documents about constraints concerning a resource. These constraints can be split into those that entitle a requester to perform an action with the service ("permission policy") and those that require to perform an action in order to use the service ("obligatory policy").

2. "Message Level Security":

   Protection of data against alteration and unwanted disclosure must be provided at all levels of abstraction necessary. If the communication channels are point-to-point, transport level security might suffice, otherwise, the chosen communication protocols must allow for a layered application of security measures.

3. "Web Services Security":
   Security must be provided for data not only during transport, but also during storage at a partner's site. It must be possible to monitor usage levels of clients and make them responsible for misconduct.

4. "Privacy":
   Involved parties must be able to trust that their counterparts are revealing appropriate amounts of personal data for a trusting communication and execution of the service. Partners must be able to trust their data is handled only by those they actually intend to.

Remedy for the problems belonging to these areas must be found and applicable in scenarios utilising the developed architecture.

4.1.3 Service and Client Location Independence

1. Cross-platform compatibility:
   The architecture must not be designed in a way that certain hardware platforms, operating systems or applicable intermediaries are excluded from acting as part of a concrete implementation without at least providing for use of a reasonable alternative. Switching from one platform to another must also be possible with reasonable effort.

2. Communication standards:
   Interfaces within the architecture must be able to utilise a range of current messaging standards for transmission of data across wide area networks.

3. Internationalisation:
   The architecture must provide the possibility to adapt components for use by clients from different regions without engineering changes.

4.1.4 Maintainability and Deployment

1. Quality assurance:
   It must be possible to encapsulate logic in a way that different approaches of testing can be utilized for individual groups of functionalities, and that components can be tested with as few dependencies on other components as possible.
2. Collaborative development:
   It must be possible for developers to alter functionalities they have expertise with, with a minimum knowledge of the internals of other components of the system.
3. Versioning:
   The architecture must allow for usage of systems for version control in order to allow for dependency management as well as the planning and automatic execution of deployment plans.

4.1.5 Concurrency

1. Load balancing:
   If operations that can be executed on a single machine are requested by multiple users simultaneously, it must be possible to distribute calculation load of these operations between individual machines with only as few overhead as necessary. Since excess of available (or acceptable) calculation time, which may occur especially in multi-user environments, poses failure of system execution, the architecture must provide for mechanisms of load balancing to

allow for assignment of calculation tasks to individual machines or sub-arrays of clusters if such is available.

2. Support homogeneity:
   If required processing time of multiple requested operations exceed the capacity of a single machine only on a certain stage of processing (i.e. are homogenous regarding the infrastructure needed for execution), it must be possible to assign only sub-processes of this particular stage to a distributed architecture, allowing for a different level of distribution (if any is necessary) on the other stages where possible.

3. Sharing of common objects:
   Individual scenarios may require redundant execution of identical operations in multi-user environments. To enable highest possible re-use of calculation time spent, the architecture must provide for facilities that store intermediate calculation results as objects which can be made available to other processes to avoid recalculation.

4.1.6 Performance and Scalability

1. Execution time:
   Processing requirements that originate from a single, computationally expensive process must be executed in a timely manner. No concrete requirements concerning the execution times shall be made on the architecture level as these vary greatly between usage scenarios, available hardware and embodiment of application logic to be performed, and often, execution times increase proportionally with problem size (Martin and Berry, 2007). Just like excess of calculation time due to a too high amount of parallel load is a failure of execution, the exceeding of the time limit due to a single request taking too long is failure as well, and possibilities for distribution should also be possible on a sub-process level.

2. Memory usage:
   It must be demanded that memory is not exhausted during processing, or algorithms must provide for distribution of workload over multiple machines if memory might be exceeded. Especially, the resulting architecture must allow for utilisation of the approaches of tackling these problems discussed in 3.1.1.

3. Utilisation of existing libraries:
   It must be possible to effectively utilise the aid of computers in the processes of LSA as outlined in 2.3. Therefore, utilisation of technologies of various origins must be possible with minimum effort by encouraging a maximum amount of encapsulation, which further supports the demand to distribute LSA processes over physical machines.

4. Calculation time management:
   To enable execution of complex LSA procedures in a (close to) realtime execution context, workflows that allow for time-shifted execution of processes (especially calculations in advance) must be supported by the resulting architecture. In the course of that, facilities must be offered that enable the effective re-use of intermediary calculation results as well as the pre-calculation of data if the application scenario allows for such course of action (e.g. preparation of generic and domain spaces).

4.1.7 Stability and Handling of (Partial) Failure

1. Monitoring:
   The architecture must allow for monitoring of individual components' availability without requiring the unnecessary execution of computationally expensive overhead.

2. Handling of defects in components:
   The architecture must allow for implementation of services running on re-

dundant hardware (components or whole machines) on each stage of the execution process. This way, handling of hardware failure must be possible, as well as handling of software failure if it is caused e.g. by deficient deployment, by dynamic reallocation of (sub-)process execution to a different component/machine.

3. Handling of ultimate failure:
   In case of ultimate failure, e.g. due to a software bug promoted throughout the system as part of deployment of a new (faulty) version, error messages must be generated for both the developer and the user, offering each of them different levels of verbosity.

4.2 Layer View

According to Avgeriou and Zdun (2005), this view aims to decompose the whole framework into interacting parts, associated in the form of "layers". The aim is to find components that are well encapsulated, and have defined points where they can be coupled. Layer models normally assume that each layer needs compatible instances of the "lower" layers to operate, but not those that are "higher" in level (Fowler, 2003).

4.2.1 Determining the Optimal Number of Layers

The decision of how many layers to use for a distributed system is often difficult:

- According to Reese (2000), a **2-tier** architecture normally consists of a "fat-client" directly addressing a data storage, causing all business logic to take place in the client (hence the name "fat"), with only the security and validation routines in the database. Such practice has the downside that "Two-tier, fat-client systems are notorious for their inability to adapt to changing environments and scale with growing user and data volume". It also discourages reuse of the client, since it is single-purpose and depends on the database layout.

- According to Fowler (2003), a **3-tier** architecture is desirable because it separates tasks performed by processing units into more decoupled roles:

  1. Presentation
     "Provision of services, display of information", often the human-machine interface, but may also be a code interface which produces output to be interpreted by another machine.

  2. Domain logic
     Also known as "business logic", this layer contains the knowledge on how to perform certain tasks.

  3. Data Source
     Logic within this layer is used to "[communicate] with other systems that carry out tasks on behalf of the application", which can be "transaction monitors, other applications, messaging systems" and others like database management systems (DBMS) or storage facilities.

  Reese (2000) uses the names "Client", "Server" and "Data store" for these layers, respectively.

  This architecture decouples the logic used to present results from their actual calculation, and at the same time, makes the business logic independent of the mechanisms used to store data. This separation makes sense since the business objects receive a separate space for the rules used to act with them, and "the rules for how data should be processed rarely change" Reese (2000). On the other hand, without changing a single data object, the presentation layer is able to display the results of the process on various media.

- A modification of the 3-tier architecture is the "Network Application Architecture" Reese (2000), which inserts a service layer between clients and the business logic, actually resulting in a **3-tier with sub-levels** architecture with intermediate tiers transparent to the top-level tiers.

- Evans (2003) suggests to use one more tier, separating most of the application layer into a "domain" layer, turning the former into a thin layer used for internal orchestration of the subcomponents of the latter. This results in a **4-tier** architecture. A similar architecture is described in Ceri et al. (2003), where 4 layers are derived containing client, web server, application server and database server, respectively.
- If more granularity is needed, the system can be further split, resulting in an **n-tier** architecture. However, it has to be noted that with increasing numbers of tiers comes additional complexity and therefore workload during development and maintenance of the system. For this reason, a compromise between encapsulation on the one hand and complexity on the other hand has to be found.

The above considerations lead to the derivation of the appropriate n following this argumentation:

- Since most of the selection criteria of the 2-tier architecture mentioned in Reese (2000) can be denied, this architecture is not to be chosen. Also, good reasons have been mentioned above to choose $n \geq 3$, introducing a separate layer for business logic.
- A separate layer for LSA logic orchestration and result object serving should be introduced in order to encapsulate logic and hide complexity, speaking for the model suggested by Evans (2003), with $n = 4$, having the LSA logic in a "domain" layer and the distributed system communication logic in an "application" layer. This allows for the free reuse of LSA logic in different environments, using different service technologies (e.g. SOAP, XML, RMI, ...) in the "application" layer.
- No need seems to arise for $n \geq 4$ since the relevant granularity seems reached.

For these reasons, a **4-tier client-server architecture** has been chosen.

4.2.2 Fitting Common Components of LSA Processes into the Client-Server Architecture

Figure 4.1 shows how the components of a typical LSA system can be split into layers.

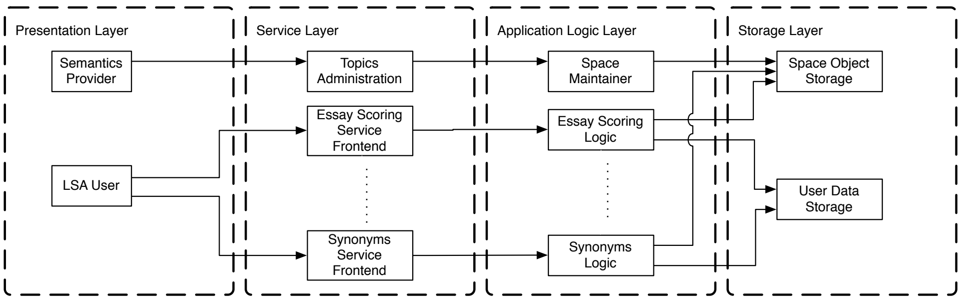


Figure 4.1: Layer decomposition of a possible LSA system

The suggested 4-tier architecture comprises the layers of the 3-tier with sub-levels architecture, but with the two separate parts of the "application" layer considered as two individual tiers. Note that the evolution underlying the development of this architecture has lead to a nomenclature more similar to that of Reese (2000) than to that of Evans (2003), which is outlined in Table 4.2.

LTfLL This thesis	*2-tier* Reese (2000)	*3-tier* Fowler (2003)	*3-tier w/ sub* Reese (2000)	*4-tier* Evans (2003)	*4-tier* Ceri et al. (2003)
Presentation	Client	Presentation	Client	User Interface	Client
Service	–	–	Web Services	Application	Web server
Application Logic	–	Domain	Business Logic	Domain	Application server
Storage	Data Store	Data Source	Data Storage	Infrastructure	Database

Figure 4.2: Comparison of nomenclature in distributed systems literature

Starting with the highest-level components on the left side, the presentation layer contains all software used to interface with LSA services. This may be a full-fledged application including a GUI, a web application opened in a browser, or another server handling the succeeding processing and display tasks for the retrieved data separately. In this context, a “semantics provider” is any client serving text data or other semantic information to the system that is used to build reusable objects like spaces; an “LSA user” is a client used to provide semantic data to the system that is used to perform semantic calculations utilizing the reusable objects on the server, using the LSA service.

The service layer exposes the key functionality of the system to the clients. It serves as an indirection layer, as it can expose the functionality from the application logic layer in a condensed form if necessary. The role of the service layer is also explained in detail in Web Services Architecture Working Group (2004): “A service is an abstract resource that represents a capability of performing tasks that represents a coherent functionality from the point of view of provider entities and requester entities. To be used, a service must be realized by a concrete provider agent.” The same document also outlines the required capabilities of a service, which shall be shown to be provided in this architecture below (optional attributes omitted):

- “a service is a resource”: an identifier is provided for each service functionality, and each functionality is owned by a person or organisation.
- “a service performs one or more tasks”: 4.1 shows some examples of tasks that may be performed in the application layer and addressed via the service layer.
- “a service has a service description”, “a service has service semantics” and “a service has a service interface”: service interfaces can be described e.g. using WSDL
- “a service has an identifier”: the actual format of the identifier depends on the chosen technique of addressing (see 4.5.1).
- “a service has one or more service roles in relation to the service’s owner”: 4.1 hints that functions that are homogenous in their functionality may have sim-

ilar component distribution in the architecture and may therefore be grouped to roles.

- “a service is owned by a person or organization”: Due to the distinction between the presentation layer and the other layers, the service itself can be located and owned by anyone without harming the client.
- “a service is provided by a person or organization”: see above.
- “a service is realized by a provider agent”: the application layer and the storage layer are“capable of and empowered to perform the actions associated with a service”.
- “a service is used by a requester agent”: the requester agent is every entity situated in on the presentation layer, accessing the service layer via defined interfaces.

Note that the “topics administration” functionality is used to create, modify and delete reusable objects that are stored on the server for later use (“topic” refers to the context of LSA, where a semantic space represents a specific topic). It is also responsible for handling any connection-related issues in the client-server communication process. In Figure 4.1, the dotted line between the two frontends implies that multiple LSA tasks can be wrapped this way, the essay scoring and the synonym search being only examples.

The application logic layer holds any infrastructure responsible for the actual calculations. The space maintainer is a routine capable of creating, modifying (“fold-in”) and dropping actual latent-semantic spaces, and therefore encapsulates the core LSA logic. Furthermore, this layer comprises any task-specific logic used to serve LSA user requests.

The storage layer represents a supporting sub-system, serving the application logic layer. The space object storage is able to hold generated spaces in a highly accessible way. If transfer of spaces is chosen to be avoided in favour of a reference-driven communication, the storage must be able to serve a space identification token ("space ID") for every space provided, and vice versa. The user data storage holds parameter data provided by the user, possibly on a per-session or a per-account basis.

The concerns of this view (Avgeriou and Zdun, 2005) have thus been solved:

- The parts of the framework have been defined.
- It has been shown which component interacts with which others.
- The adequate decoupling of the individual components, including their dependencies, have been set.

4.3 Data Flow View

In Avgeriou and Zdun (2005), this view is described as a way to look at the framework "as a number of subsequent transformations upon streams of input data". Again, the focus lies on the components being independent of each other, only defining communication protocols and data formats for their connectors.

Figure 4.3 shows how data is moved within an LSA system during the two key processes "space generation" and "task execution" as outlined in 2.3. It also shows the key input and output data types at each stage, which is important for realisation of a pipes-and-filters-architecture.

For the space generation, a text corpus is put into the process, where it is transformed into a computable object and then transformed into a space (Figure 4.3 shows this process for an LSA-based computation). Note that this architecture suggests that the actual space is not returned to the requester, but rather, a reference to the space's location. This is due to the fact that LSA spaces are large, complex, and non-sparse objects that actually have to be available on a fast medium, which suggests handling the actual space data internally and only exposing a "space ID" to other functions.

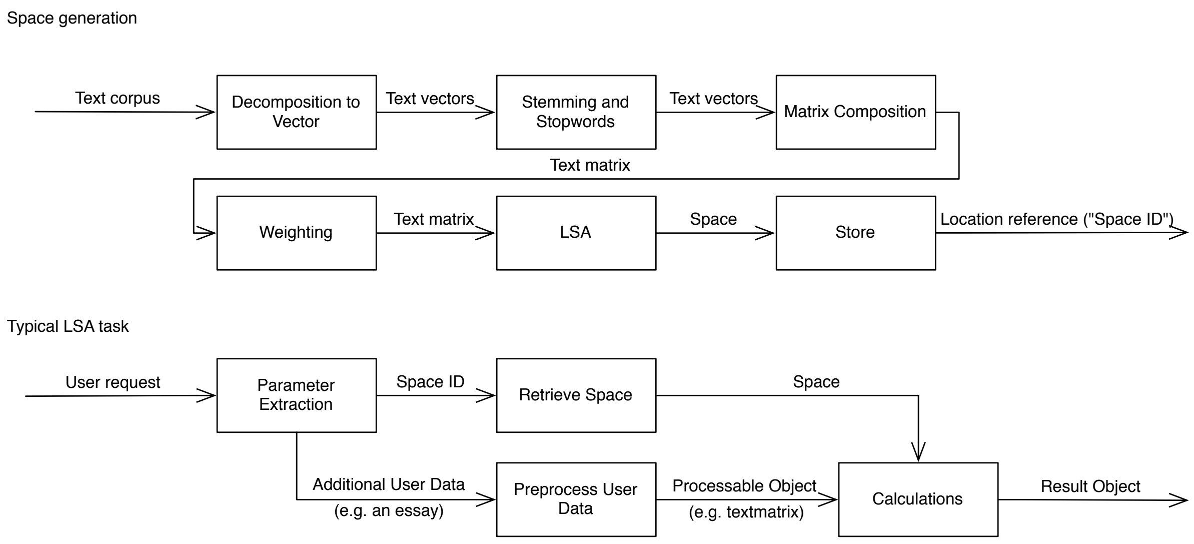


Figure 4.3: Data flow during a typical LSA process

During a typical LSA task, the service interface receives a user request holding the execution parameters, one of which must be the space (again, space handling via space locators is only a suggestion). After pre-processing the user parameters and data, an internal LSA logic is invoked, returning a result object to the communication controller.

Note that the typical LSA task allows a parallel execution of space retrieval and user data pre-processing, while the space generation is a pipelined operation.

The concerns of this view (Avgeriou and Zdun, 2005) have thus been solved:

- A set of transformation elements has been defined
- The carriers and data formats for the data streams are clear
- Connection points and order of execution have been set

4.4 Component Interaction and Distribution View

This view augments the layered view by looking closer at the interface structure as depicted in Figure 4.4.

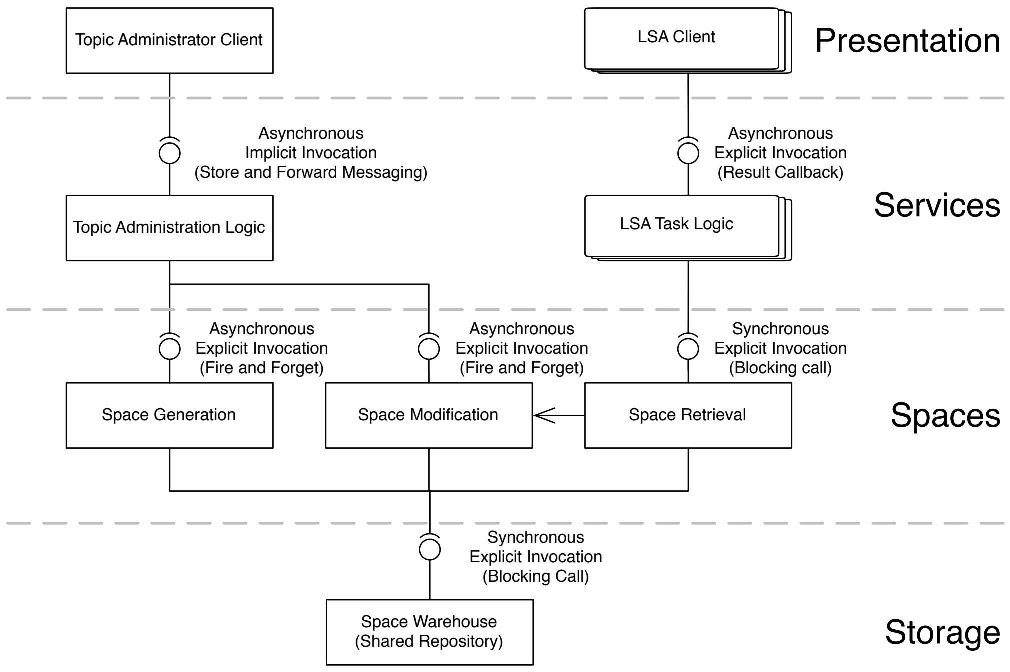

Topic Administrator Client
LSA Client
Presentation
Asynchronous Implicit Invocation (Store and Forward Messaging)
Asynchronous Explicit Invocation (Result Callback)
Topic Administration Logic
LSA Task Logic
Services
Asynchronous Explicit Invocation (Fire and Forget)
Asynchronous Explicit Invocation (Fire and Forget)
Synchronous Explicit Invocation (Blocking call)
Space Generation
Space Modification
Space Retrieval
Spaces
Synchronous Explicit Invocation (Blocking Call)
Space Warehouse (Shared Repository)
Storage

Figure 4.4: Component Interaction and Distribution View

Component interaction is depicted with the invocation types next to the interfaces. Starting at the least-granular layer, the space warehouse holds objects, which are essential for the execution of the accessing components' logic and therefore, retrieval of the space blocks the accessing component until completion.

Space generation and modification (the latter relying on the component of space retrieval) are a time consuming task, and most likely no client will want to wait for its completion. Therefore, together with the topic administration logic (addressed, e.g., using a web service) a store-and-forward-messaging architecture is suggested. The topic administrator client sends a request object (including parameters and data) to a queue managed by the topic administrator logic, and receives nothing but a confirmation of receipt at the queue. The topic administrator logic then retrieves the topmost element in the queue as soon as processing capacity is available and forwards it to the space generation/modification logic, using a "fire and forget" invocation. At any time of this process, the topic administrator client can access information about the progress by accessing the space retrieval logic via the modification logic.

Finally, the clients of the LSA tasks (depicted by a stack in Figure 4.4, as there can be many different tasks, addressed by different specific clients) access their underlying logic via their respective service interfaces, using arbitrary remote invocation methods, most likely, (a)synchronous explicit invocations. The respective logic components then access the space retrieval component using a blocking call, as again, the spaces are vital for the calculations.

From the distribution view, Figure 4.4 shows the different remoting approaches used for the topic administration on the one hand and the LSA task execution on the other. The topic administration uses a message queuing remoting pattern, for the reasons described in the component interaction view above. The invocation of task logic is realised using the remote procedure calls remoting pattern.

The concerns of these views (Avgeriou and Zdun, 2005) have thus been solved:

- The components' interaction has been shown
- The decoupling has been defined

4.5 Treating Prevalent Problems in Client-Server Architectures

The previous sections have shown how a 4-tier client-server architecture can be used to create the facilities to provide LSA as a service component and enabling use of LSA as a distributed system. This section aims to provide solution for some problems commonly observed in client-server architectures. It is not presumed to be a complete list, but enumerates the most obvious problems met during development of the model and evaluation of the requirements.

4.5.1 Addressing a Resource and Its Functionality Consistently

Often, while a service oriented architecture grows, developers face the problem that a lot of functionality is available for a single type of object, and the question arises how the functions should be addressed. There are multiple approaches to solving this issue:

Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) is a protocol which uses XML messages contained in the body of a HTTP message to provide an envelop for communication between machines with heterogeneous characteristics (operating system, object model, . . .).

A SOAP message consists of an envelop containing a header and a body. To draw an analogy to physical mail, the header contains the information put on a letter's envelop, like the the priority of the letter (QoS) or the stamp (billing information). The body is equivalent to the actual letter and contains a list of "body-childs". One of these body-childs may contain information about what function to call at the service interface.

The advantage of this approach is the versatility: arbitrary objects and functionalities can be exposed at a single web service URL, and responses by the server can contain another arbitrary set of information. There is much more functionality packed into SOAP than just the addressing of functionality at a service. For more details on SOAP, see Papazoglou (2008).

Disadvantages of SOAP include the added complexity and payload size of SOAP messages.

Representational State Transfer (REST) and CRUD

REST means “Representational State Transfer” and is a design guide for hypermedia system architectures that “emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components” (Fielding, 2000), with a focus on uniform handling of objects and associated operations in the context of the Hypertext Transfer Protocol (HTTP).

REST is actually an object-centred approach to addressing a resource: an object available in a network has an identifier and a set of operations defined by the REST-”standard” that can be performed on it. These operations are called by passing the respective HTTP request method with the request, along with any parameter data. Advanced usages allow for keeping state between more complex operations in a REST context.

Web services supporting REST requests are called “RESTful”. An advantage of RESTful web services it that they profit from a standardised way of addressing objects over a network. Especially, REST allows for a standardised way of performing Create, Read, Update and Delete (CRUD) operations on an object. Notice, however, the disadvantages of this approach outlined in Evdemon (2005):

- The operations performable with REST requests is very limited
- If a SOA is build solely upon REST, the developer might be tempt to expose too much internals about the contained object structure to the public interface, which in turn is considered an anti-pattern.
- Usage of REST for implementation of a CRUD interface is least recommended for usage in a service context because it violates the actual principles of services: not the actual behaviour of a business logic is exposed, but rather, a part of the internal data model.
- As soon as the object logic requires more advanced functionality (e.g. committing changes separately), CRUD is dangerous as it does not preserve state

and cannot perform actions as a transaction.

- If, however, the “service” is actually meant to be only an interface to a simple object, CRUDy REST implementations have a right for existence if the manipulation service is justified in the scenario context and manipulations do not require maintenance of state or transactions.

4.5.2 Transfer of Object Data Over Networks

To transport structured object data over networks (utilising unstructured data streams) requires either a transportable binary format or some form of serialisation. Various transfer data types shall be discussed in this section, providing an overview of options. Note that only standardised formats with reasonable usage in practice are mentioned here. Throughout this section, (S) shall be the sending machine, (R) the receiving machine.

Binary Data

The simplest form of data transfer between is to create a binary stream from the in-memory object held by (S) and directly sending a message to (R) holding the stream in its payload.

The advantage of this approach is that no serialisation has to take place, which reduces load on the processor of both machines.

The disadvantage is that both systems have to use the same internal data format and memory mechanisms, which is a requirement often unthinkable in the heterogeneous environment of a distributed system (especially across company borders).

Comma Separated Values (CSV)

Simple objects can often be represented by table data (i.e. 2-dimensional structured data consisting of “rows” and “columns”, having “cells” at the intersection of these). Such data can be converted to Comma Separated Values (CSV) data. The actual separator may be any symbol as long as escaping rules are defined if cells may hold values containing the separator.

The advantage of this format is that maintenance of the structure requires comparably few overhead (one symbol per cell, one line break per row) and the easy readability for humans.

The disadvantage is that, obviously, this format is only applicable to 2-dimensional data, i.e. no hierarchy can be represented. It also has no means of specifying the encoding standard used for the cells, so this has to be agreed on by both machines over a different channel.

Javascript Object Notation (JSON)

JSON is a serialisation format described in RFC 4627 which uses a pre-defined syntax of brackets, quotation marks and colons to store object data in a structured format.

The advantage of this format is that data remains (somewhat) human-readable, that overhead necessary to preserve object structure is comparably low, and that the structure of JSON is somewhat similar to many common programming languages, which is why converter functions are often present in such languages.

The disadvantage is that JSON requires the original object to be serialised, which consumes processing time. Another disadvantage is that JSON can not store object references, leading to redundancy and loss of consistency in more complex object structures.

eXtensible Markup Language (XML)

XML is a serialisation format defined in the XML 1.0 Specification and multiple other open standards primarily published by the W3C. It uses a notation for structure characterised by strings delimited by inequality symbols (so-called "tags") holding attributes and values to represent objects. Despite the basic rules for XML notation, various standards allow for specialised structure definitions, and XML documents can even be used to describe other XML documents.

The advantage of this format is that data remains human-readable, that rules about the structure can be defined to aid deserialisation and validation, that XML is currently very widely used as data communication format and has support in most common programming languages, and that references between objects can be set (e.g. using xPath).

The disadvantage is that XML produces a lot of string data for even simple data structures, that it does not have a specific notation to distinguish data types, and that it requires (S) to serialise and (R) to deserialise the data stream, often also requiring a schema definition to make the XML data machine processable.

YAML

YAML is a serialisation format which relies on indentation of strings as well as delimiting object collections with brackets to represent the structure of an object.

The advantage of this format is that, again, data remains human-readable, that it is less verbose than XML but easier to read than JSON, and that object references can be stored to reduce redundancy.

The disadvantage is that YAML can in some cases require a lot of whitespace to represent objects (especially deeply nested structures), that it is prone to errors as whitespaces (which are not visible for humans) are a crucial element of maintaining structure, and that (de-)serialisation has to be performed on the respective ends.

Also, YAML converters are not as widely spread among programming languages as they are for e.g. JSON and XML.

4.5.3 Maintaining State Between Accesses to the System

User interaction often involves production of parameter data the user defines during his interactive experience. In many processes involving LSA, this parameter data can include large objects that are the by-product of computationally expensive calculations. Avoiding repeated calculation of this parameter data throws up a problem often observed in the context of distributed systems: "client-dependent state must be maintained in the distributed object middleware between individual accesses of the same client" (Völter et al., 2005). A potential solution for this problem is the "Sessions" pattern (Sørensen, 2002): the user himself is given a session ID, and all data passed to the server (or produced there) is kept in the server's storage. This way, transfer of data between client and server has to be done at most once (except if the user loses his session ID). As a result, the user can re-use his parameter data (hence must not be bothered with entering it repeatedly), and the server has to do calculations less often if re-use of generated objects is supported by the application logic.

These considerations require a augmentation of Figure 4.1: another potential component to be contained in the service layer is the Session Retrieval Service, which is responsible for restoring the session object in the relevant invocation context (Völter et al., 2005); see the revised architecture in Figure 4.5. The application logic layer may contain an associated Session Object Restore Logic, which obtains all relevant variables from the Session Data Storage, which may be located on the storage layer. After the stored session data has been restored by the Session Object Restore Logic, it may use any space IDs found to retrieve the actual space objects from the space warehouse if the implementation scenario favours this approach, or might simply return the space IDs for the actual service' application logic to perform space retrieval.

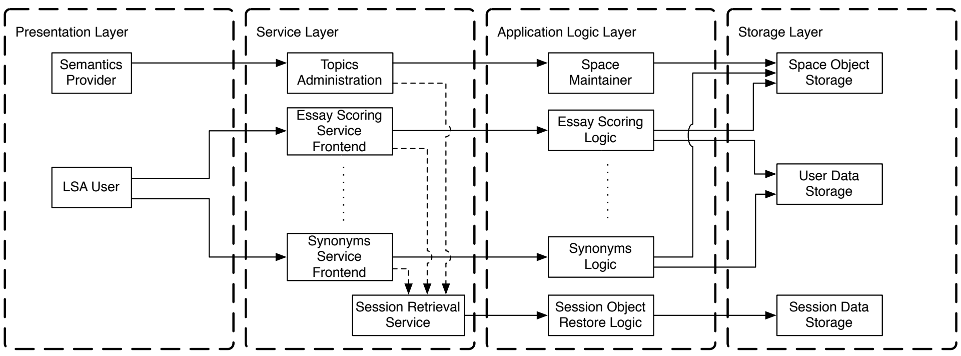


Figure 4.5: Layer decomposition of a possible LSA system featuring a session storage

Note that there is a difference between "session data" and "user data": the former's existence is limited to the duration of the session. The latter's is limited to the lifetime of the user. This means that, if there is data that must outlive the session (e.g. essays a user has written), this data must be stored in the user data storage (which is why it has not been removed in the revised architecture). Also note that session IDs are stored on a machine locally and are typically not easy to transfer (or transfer is barred by definition), meaning that all data that should be available after the user has changed the physical machine he uses to access the system must also be stored in the user data storage.

The introduction of the session mechanism shows an essential reason for a distinction between the presentation layer and the service layer: if the client is the GUI of a user frontend of an application, the developer of that GUI does not want to be bothered with restoring previous navigation parameters; he will happily accept if they are simply adhered to during calculation of user reports, as long as there is an interface to alter them if needed.

4.5.4 Security

As client-server architectures are systems open for anyone with a physical connection to the server (which, in case of the internet, is a lot of people), the maintainer of a service will have to consider restricting access to the server for various reasons:

- The service provides information not suitable for anyone to get.
- The service runs on scarce resources and must only be provided to people who are entitled to consume those resources (e.g. because they belong to the interest group hosting the service, or because they paid for its usage).
- Usage of the service allows for performing operations that might have consequences for other users and therefore, every such operation must be attributable to its initiator
- Users of the service are entitled to perform different tasks depending on e. g. credentials given to them as part of a information security strategy.

All these aspects require some way to restrict access to a system. With the rise of the internet age, the identity of a person in the web has become a matter of registering with a service by choosing a username and a password, which is then used to assign an account to a user. As the query of a service for the login data for the purpose of identity verification itself causes the user to disclose his credentials to the service. This causes multiple issues, most importantly:

- The risk of eavesdroppers intercepting the credentials and impersonating the client ("steal" his identity). As eavesdropping has been a problem on the internet since the inception of telecommunication, secure communication channels have been developed. Currently, network communications can easily be encrypted and made very difficult to decrypt, but providing a service with encryption facilities costs computing power, which is the reason why especially free services often don't provide such. Users who are not aware of this are at risk to lose their identities to other users of the network.
- In the case of re-use of the same login data for multiple services, the risk that the provider of the service himself takes the data to impersonate the

client at a different service. This can easily be avoided by using different passwords for every service, but this makes the login credentials difficult to maintain and remember. A potential solution to this problem is to lock away all credentials in a single “vault” which is the secured by a master password which makes the login data easier to manage.

One approach to provide a solution that eliminates these issues is OpenID, which is described at OpenID Foundation (2010): “OpenID is a decentralized authentication protocol that makes it easy for people to sign up and access web accounts.” An end-user (i.e. a person which wants to own an identity in the internet) claims a document somewhere in the internet (e.g. his personal website) and relays, within that document, the information needed to verify the identity of the document’s owner. This requires the end user to create one single account with an identity provider – that is a service that has the task to store the login credentials of the identity and hold it available. The provider does not actually check if the user is actually known under the name he pretends to be, but creates a “virtual person” that can then go on and register for services (“relying party”) that provide openID authentication. To authenticate, the user provides to URL of the document he has claimed, and is then asked for authentication with his providing party, and to grant usage permission to the relying party. The relying party simply stores the address of the claimed document (the virtual representation of the person) with the account. This way, the following access control issues have been solved:

The OpenID logo [1]

- The end-user only needs a single set of login credentials: the data to authenticate with his provider.
- The end-user can choose any identity provider that he finds to be trustworthy, preferably, one which supports encrypted verification communication,

[1]Source: http://openid.net/images/logo/openid-icon-500x500.png

and may swiftly change his provider by simply changing the verification information stored in his personal document.

- The relying party does not have to worry about password theft or storage of confidential data stored on its servers: it waits for the end-user to provide evidence that he has verified with his provider, which can be checked with the provider to ensure authenticity.
- Once the user is authenticated, he can be treated like any other user holding an account on the server, including access control, role management and usage quotas.

Identity management by OpenID should be the initial step performed on the service layer. This way, no resources other than the verification request are consumed by unauthenticated users, which provides less fertile grounds for denial-of-service attacks. In Figure 4.6, this modification has been made to the architecture outlined previously.

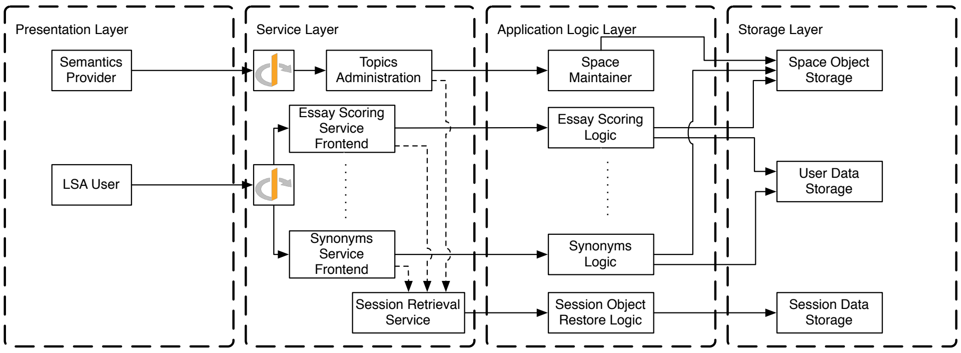


Figure 4.6: Layer decomposition of a possible LSA system augmented with OpenID

4.6 Requirements Met with This Architecture

Some requirements outlined in the previous section are met per se when a solution implementation adheres to the architecture specified above. This section is going to show for which requirements this is true, and why.

4.6.1 Network latency

1. Caching

   The separation of the application logic from the data used for processing (with the separation of application logic layer and storage layer) allows for data warehouses to be shaped as CDNs, and the separation from the clients requests from the actual logic (by having an intermediate service layer) allows for proxy servers and server accelerators to provide cached information if it is requested multiple times (e.g. in a search engine scenario).

2. Compression

   Many common data interchange formats are verbose and therefore do not have a good information/storage ratio (see 4.5.2). Compression algorithms allow for making a data representation of an object more dense. Separation of the most important layers allow for communication channels that are frequently seen in web service architectures and therefore, chances are that a compression algorithm is present for a particular communication channel implementation. Especially, XML technologies (especially SOAP) allow for excellent compression of messages (Augeri et al., 2007).

4.6.2 Security

When using OpenID with communication utilising an encryption mechanism that also uses digital certification mechanisms (e.g. SSL) on all channels, the following security requirements (see 4.1.2) are met:

1. Policies:

   Trust policies and cross-domain identities can be be met as SSL allows for identification of the communication partners by the certificates (service identification with service provider's certificate, client identification with identity provider's certificate), as long as a web of trust is established, defining which certificate issuers are trusted.

2. Message Level Security:
   Secure messaging is established as SSL allows for encryption of messages sent between the communication partners. This ensures confidentiality of the information and protection from alteration. Partners holding digital certificates can avoid man-in-the-middle and replay attacks and spoofing.

3. Web Services Security:
   OpenID provides for reliable authentication mechanisms, and an established identity of a user allows for authorisation mechanisms to be established on both the client and server side, as well as audit trails to identify users violating e.g. the terms of use. Integrity and confidentiality are ensured by the encryption mechanisms of SSL, even across the insecure channel of an internet connection.

4. Privacy:
   Message encryption ensures that information communicated between identified parties can be kept confident during transfer, and determination of identities after authentication ensures non-disclosure of stored information between accesses of the system.

Since SSL is a transport layer security mechanism, it does not allow for end-to-end security. If this is needed (e.g. because security intermediaries are necessary), a different approach for encryption must be chosen, which will probably feature encryption mechanisms anyway.

4.6.3 Service and Client Location Independence

1. Cross-platform compatibility:
   The proposed architecture does not obstruct implementation of the components in a portable language, and does not impose platform-specific constraints neither to the communication between nor to the implementation of the components themselves, hence the requirement is met.

2. Communication standards:
   The layer composition has been designed to have the points of interfacing located within the process so that data can be transferred using most standard communication protocols, including transfer using HTTP over TCP/IP, message management using SOAP and service discovery using UDDI.

3. Internationalisation:
   The presentation layer proposed in the architecture can be used to hold translation information for pre-defined messages provided by the service layer. If a thin client is desired, translation must take place in the service layer, allowing the client to send the desired response language with the request messages (e.g. with a HTTP accept-language header field).

4.6.4 Maintainability and Deployment

1. Quality assurance:
   Splitting the service architecture into layers allows for application of different test techniques for different demands. For example, client software can regularly be tested by human users, as it may be required to be tested not only for functional operation, but also, e.g. for acceptance after changes. Services can be tested for regression and integration using the techniques described in Baresi and Di Nitto (2007) as they have to perform correctly in conjunction with other services. Application logic can be developed and maintained using techniques like test driven development to make use of the various advantages of this technique outlined in Beck (2003). The storage layer may be tested for performance using benchmark utilities. Continuous integration can be used as an accompanying measure to ensure constant quality even in agile development environments.

2. Collaborative development:
   Other than the encapsulation fostered by the layered structure, the architecture does not provide facilities that make collaborative development more

attractive than it is for any other architecture. However, use of collaborative development techniques is not hindered either, hence the requirement is met.

3. Versioning:
   As with collaborative development, use of versioning systems is neither necessary nor hindered. The structure, which focuses on versatile components, however, allows for effective dependency management and enables planning and automatic execution of deployment.

4.6.5 Concurrency

1. Load balancing:
   The separation of application logic from the service layer enables for effective load balancing upon request by components of the presentation layer. Since the distribution of load is performed in the service layer, this process is transparent for the LSA programmer who acts on the application layer.
2. Support homogeneity:
   The 4-tier architecture splits the execution of LSA operations in a service context into the sub-elements which are most probable to be homogenous. Permanent loads on a layer can be supported by specialized hardware, varying loads between layers in a constantly changing application scenario can be supported by multi-purpose configurations hosting many or all components at once, changing their active roles depending on the current demands.
3. Sharing of common objects:
   Maintaining a separate storage layer allows for application logic to store intermediate results for re-use by other processes effectively. Determination of re-usable parts of calculation must be performed as part of the application logic, possibly depending on load information from the service layer.

4.6.6 Performance and Scalability

Adhering to the 4-tier architecture outlined in the previous section, the following performance and scalability requirements (see 4.1.2) are met:

1. Execution time:
   The approaches outlined in 3.1.1 turn out to be deeply embedded in the logic of LSA itself and therefore seem best situated on the application layer as part of the LSA libraries utilised to perform the application logic. Still, the suggested architecture does not hamper the inclusion of such advanced approaches (see next requirement).

2. Memory usage:
   Arguments outlined in the requirement above also apply to the use of distributed memory management approaches as outlined in 3.1.1.

3. Utilisation of existing libraries
   The suggested distribution of system components facilitates encapsulation and independent development of interchangeable solutions based on different libraries, but embedded in the same service context. Communication middleware can be exchanged without requiring adjustments to application logic (which is often beyond the scope of a service engineer).

4. Calculation time management:
   With the presence of the storage layer, holding components tailored specifically towards the goals of re-use and portability of data objects in an intertemporal context (specifically, by allowing fast retrieval and effective memory management), expectable calculations can be performed in advance, and intermediate results can be accessed in a timely manner for further processing. Utilisation of asynchronous request mechanisms between presentation and service layer allow for invocation of LSA procedures as non-blocking calls, facilitating usage of LSA in a support role running in parallel with other processes (e.g. writing a text and providing feedback as the user types).

4.6.7 Stability and Handling of (Partial) Failure

1. Monitoring:

   As the 4 layers offer interfaces appropriate for each respective purpose, attaching a monitoring software to there interfaces is easily possible. For example, availability of the service or storage layer can be monitored using check scripts triggered by CRON or Nagios.

2. Handling of defects in components:

   The suggested partitioning of the components into layers allow for specific redundancy solutions on each layer if specialized hardware is affordable, but does not hinder use of general-purpose redundant hardware. This way, backup solutions and reallocation procedures can be scaled for the individual application scenario.

3. Handling of ultimate failure:

   Although ultimate failure should be prevented quite effectively by the mechanisms outlined in 1, ultimate failure and resulting notification and fixing demands can be met if adequate error messages and exception handling are performed throughout the system. Problems may arise if error descriptions are contained in different message formats between layers, as conversion of these messages may no longer be possible upon failure of certain subroutines.

5 Evaluation of Architecture Applicability to Real World Applications

In order to support the design decisions made in 4, this chapter is going to show how the 4-tier client-server architecture can be applied to real-world examples. For this purpose, the general workflows of these examples outlined in the referenced literature will be taken as a basis for developing a solution concept that features the suggested component distribution among the layers, and details will be explicated where necessary.

First, this chapter shows how the various implementations outlined in 3.3 can be extended to fit into the architecture outlined in 4 to profit from meeting the distributed system requirements outlined in 4.1. Then, real world applications are taken from various sources to show benefits of implementing the suggested architecture.

5.1 Improving the Architecture of the General Text Parser Usability Prototype (GUP)

Figure 5.1 shows how the General Text Parser Usability Prototype (see 3.3.1) can be augmented to better utilise best practices. Instead of directly accessing GTP logic via proxy actions, GTP could implement a Model-View-Controller pattern (**?**), providing views for various application workflows executable with GTP, which are coordinated by controller implementations that utilise GTP to perform the required LSA operations. Storage of user data is performed using a separate model, which can be a MySQL model as suggested in GUP. The GUP MVC elements have access to session data retrieved by PHP's built-in session functionality. Using a controller independent of GTP library allows for parallel execution of LSA tasks if the application logic allows for such behaviour.

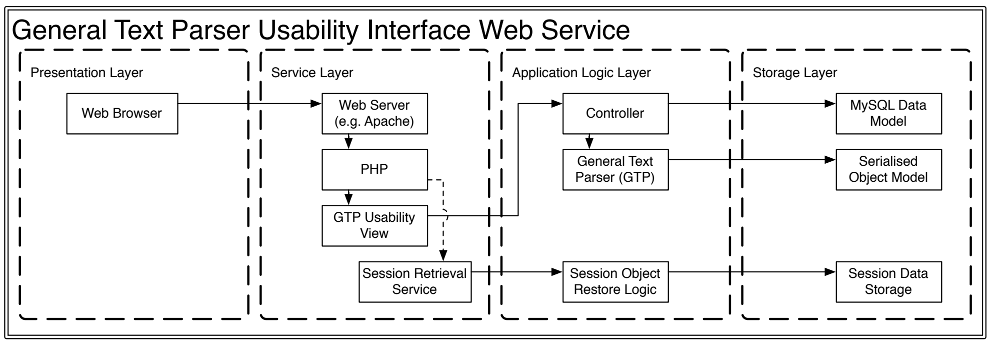


Figure 5.1: Layer decomposition of GUP Interface Service utilizing 4-tier architecture

5.2 The Cooper Architecture as a Distributed System

The LSA module implementation of Cooper (see 3.3.2) is quite similar to the architecture outlined in 4. Figure 5.2 shows how the initial structure (see Figure 3.1) is fit into the 4-tier architecture, preserving nomenclature where possible. The "layer of integration with environment" is now replaced with the "controller", which handles coordination of processing and assignment of result data to a result view, and a "format converter", which is used to perform conversion of data structures for the "very sensitive tools" (Giesbers et al., 2007). Support functions of SVM, WinGtp and SvdLibC have been omitted in the Figure, it is assumed that they are part of the LSA engine. In the backend, the "Sparse Matrix Library" is embedded into the storage layer, as well as the session storage and a model for user data storage.

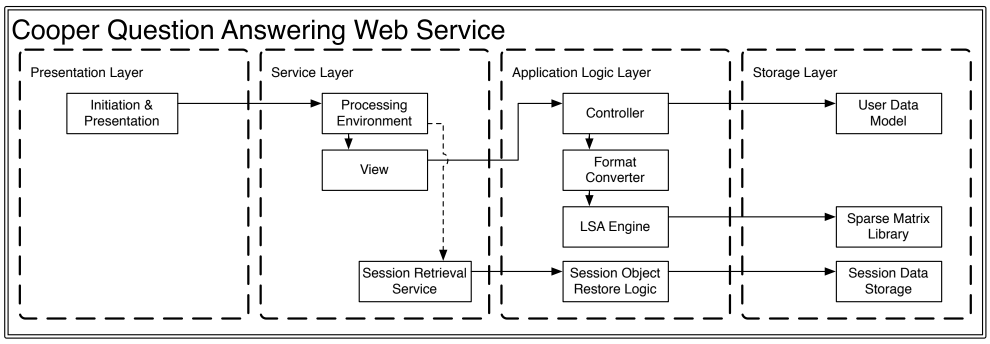


Figure 5.2: Layer decomposition of Cooper Question Answering Tool utilizing 4-tier architecture

5.3 LSA PHP and the TENCompetence Suite

Figure 5.3 shows how the TENCompetence Suite LSA SOAP service can be fit into the architecture outlined in 4. As already mentioned in 3.3.4, the architecture of the TENCompetence Suite's web service is quite similar to that of Cooper. The usage of LSAPHP, which has an integrated format converter interface, allows for removing the separate format converter from the architecture. Each functionality wrapped by the SOAP specification can be considered a view. Should the count of functions available from the web service outgrow the single-purpose API of the current implementation in the future, a controller will be necessary to assign processing facilities to the application logics executed with LSAPHP. Again, a session mechanism is added to the concept to allow for keeping state between user interactions.

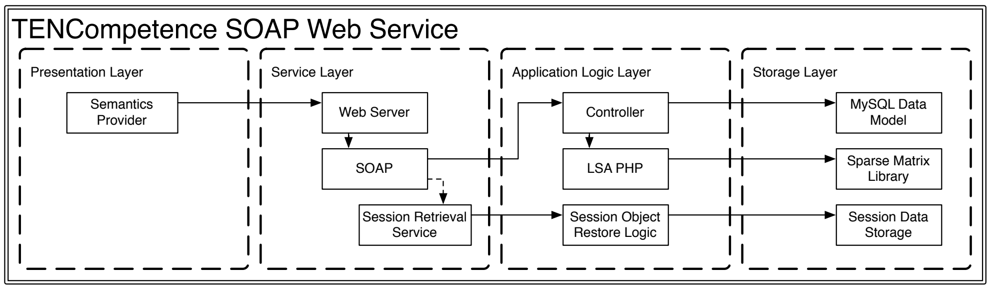


Figure 5.3: Layer decomposition of TENCompetence Suite LSA functions utilizing 4-tier architecture

5.4 Positioning a Learner as Part of Accreditation of Prior Learning

LTfLL deliverable 4.1 (Burek et al., 2008) outlines a pre-pilot scenario for a web service that assists assessment tasks that are part of Accreditation of Prior Learning (APL). Often, previous education and experience of a lifelong learner is not officially recorded by exams or certificates. To asses the prior learning experience of a learner, he can choose a collection of documents that represent his personal knowledge profile in an electronic medium – called an "ePortfolio", "holding items such as: courses completed; readings by the learner; products of the learners (here mostly written stuff)." From the documents provided, the learner is positioned within the space of relevant domain knowledge and courses or fields of proficiency that are already known by him are determined using LSA.

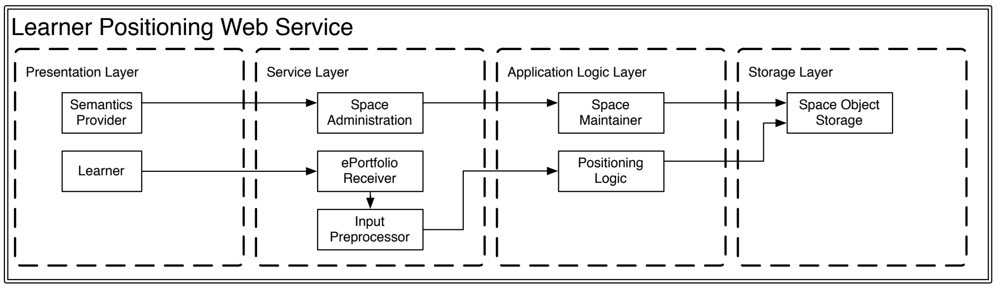


Figure 5.4: Layer decomposition of Learner Positioning Web Service utilizing 4-tier architecture

Figure 5.4 shows how the described tasks can be fit into the 4-tier architecture suggested in 4.2, using exactly the space maintenance logic suggested there.

The ePortfolio receiver might handle ePortfolio documents, or URLs to such, which might then be retrieved by the input preprocessor. Management of spaces is performed through a graphical interface, which acts as a proxy for semantic data provided by the user, hence, acts as a "semantics provider", and therefore, is part of the presentation layer.

This scenario, in a slightly altered version concerning the involved parties, is used for an implementation example in 6: assisted summary writing. This is outlined as a sample scenario in LTfLL deliverable 3.2 (Hensgens et al., 2009). In this scenario, too, a space is defined containing general language corpus data for filtering out "standard" language, as well as a set of domain specific documents. A student who decides to write a summary can quickly obtain feedback on whether the text produced (now, this is not an ePortfolio, but instead, a summary text of the chosen domain) fits into the target knowledge space.

5.5 Implicit Link Identification in the PolyCAFe Application

In LTfLL deliverable 5.3 (Trausan-Matu et al., 2010), a web application called PolyCAFe is outlined. The purpose of the application is to enable participants of courses aided by interactive text communication systems (namely, forums and chats, hence the name) to asses contribution behaviour of themselves and others.

In the course of this task, "threads" (chains of interconnected utterances) have to be identified within the logs of such communications. A vital input for the "Thread Identification" component of PolyCAFe is the provided by the "Implicit Links Identification" logic. Although the GUI and the log format allow for explicit linking of utterances by the users, in many cases such interconnections are not made explicit (and are therefore "implicit"), and have to mined out of the text.

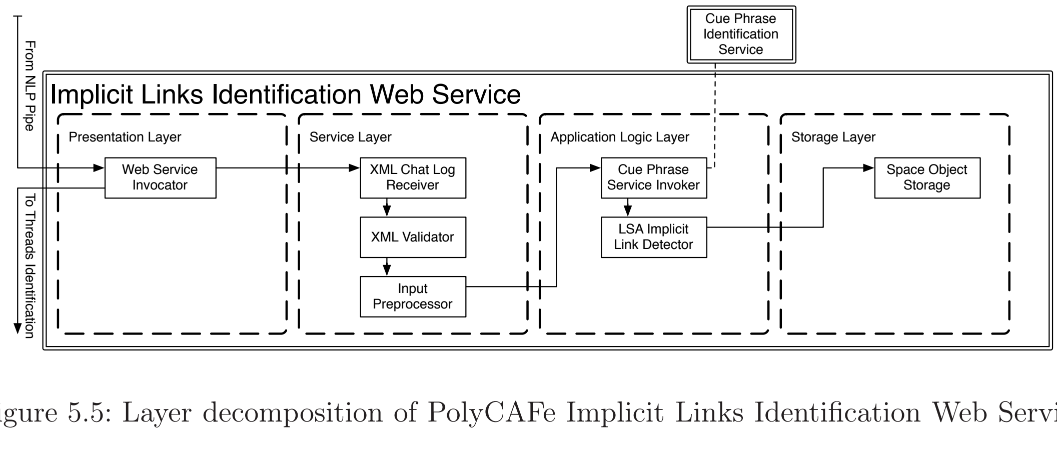


Figure 5.5: Layer decomposition of PolyCAFe Implicit Links Identification Web Service utilizing 4-tier architecture

Figure 5.5 shows how the Implicit Links Identification logic can be embedded into the 4-tier architecture suggested in 4.2 [1]. It assumes that the “Implicit Link Identification” is a self-contained web service rather than a component that is part of a broader application as outlined in the deliverable, as otherwise, the service architecture would not be applicable.

The suggested decomposition shows a great advantage of the 4-tier system over a 3-tier system: the person that implements the LSA logic may conduct this task without having to know anything about XMl validation. If desired, the XML data can be converted to other processable formats (e.g. an array of text strings) in the “Input preprocessor” component. In that case, the LSA logic implementer would not have to know anything about XML at all.

A “client” in this scenario is a web service invoker, most probably part of an orchestration process that is part of PolyCAFe. It feeds the previously obtained stemmed version of the XML into the web service through a logic defined during the implementation process and receives the result object (in the current PolyCAFe implementation, this is a text representation of the similarity vector obtained from a system call to LSA) from the service for further processing (or error handling). In most cases, this invoker will be a component of a web service orchestration middleware framework. Still, the decision about what specific client to use has to be made, which makes mentioning the presentation layer essential.

Note that LTfLL deliverable 5.2 suggests that “Cue Phrase Identification” is also a part of the Implicit Links Identification process. This logic is not based on LSA and may therefore be performed during a nested (web-) service call, which has been illustrated in the figure by positioning the logic outside the actual web service. Still, the invocation of the logic may be done on the same machine, and even within the same thread, if performance demands of the web service allow for this.

[1] A similar approach has been proposed in Trausan-Matu et al. (2009) already, based on the publication of the suggested 4-tier architecture during my participation in the LTfLL project. The approach outlined here tries to expand the idea and optimize the encapsulation.

5.6 Creation of Conceptograms in CONSPECT

LTfLL deliverable 4.2 (Burek et al., 2010) describes an application for visualisation of and interaction with so called "conceptograms" which are graphics produced as part of Meaningful Interaction Analysis (MIA), a process based on LSA combined with Social Network Analysis (SNA). The goal of the application scenario is to help the user discover the relative perceptional "position" of terms to others, and enables comparison of these "positions" with other concept spaces in a graphical process. Exploration can lead to discovery of new, yet related, fields of knowledge for the student, feedback on the factual correctness of assumed closeness, or visualisation of a learner's progress over time, to name only some scenarios.

Deliverable 4.2 outlines a demonstration process, during which a space is created that contains knowledge of "actual" (i.e. domain-specified) closeness of terms. Then, a user's knowledge represented by a text source (e.g. a blog of his) is folded into that space to show the deviation of the user's conceptual model of the contained terms from those calculated in the space.

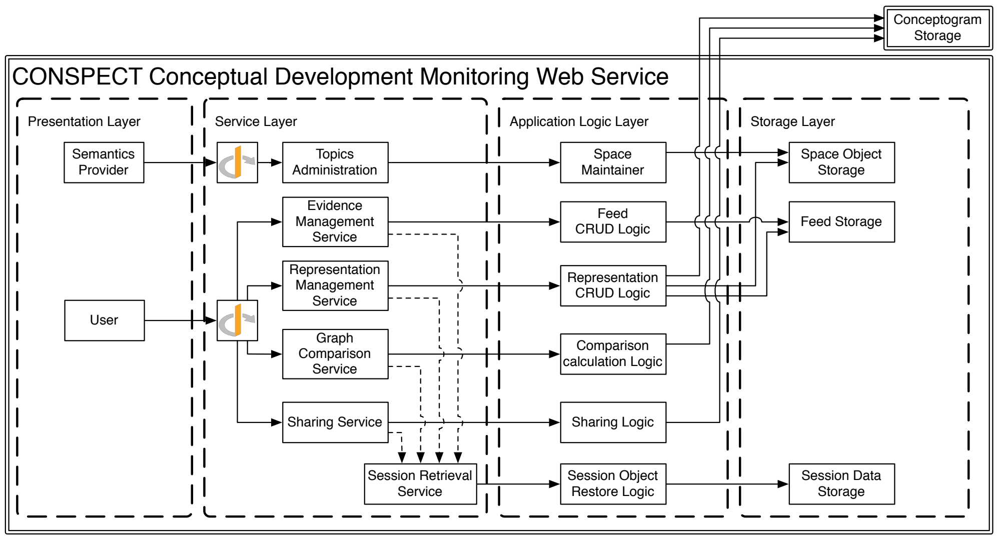


Figure 5.6: Layer decomposition of CONSPECT Conceptual Development Monitoring Web Service utilizing 4-tier architecture

Figure 5.6 shows how all applications outlined in the SUM diagram displayed in

Figure 3.9 of LTfLL deliverable 4.2 are fit into the 4-tier architecture. The task of defining, altering, computing and managing an LSA space has been outlined in 4.2. Note that, according to the deliverable, spaces are maintained through a graphical user interface, which itself becomes a "semantics provider" as it passes on the semantic information obtained from the user through the GUI to the service.

Management of identity has been chosen to be performed utilising OpenID technology (see 4.5.4) by the deliverable authors. A concrete implementation of identity checks with openID must be implemented for each process at the service layer. This, again, highlights the importance of distinguishing between the service layer and the application logic layer: a scientist implementing the conceptogram logic and the foundation LSA routines will probably have no idea of how to perform identity checks using openID – with two separate layers, however, integration of logic and security management can be done independently.

Managing feeds may be done through a typical REST-style web service interface for create, read, update and delete (CRUD) operations. See 4.5.1 on the potential disadvantages of that approach; the simplicity of the service in this context, however, justifies this solution. The web service (which may actually be as simple as an Apache module configured to call PHP files depending on the request method) may instruct the feed storage to perform CRUD operations via a simple object-relational mapping software situated in the application logic layer.

CRUD operations for space representations are conducted almost exactly the same way, with the exception that creation and update of the graph triggers a graph computation process, which itself requires the underlying LSA space from the storage layer. Calculation of the conceptogram involves, among others, the very time-consuming task of calculating term-to-term-similarities. Recalculation of the resulting matrix for every individual access of the term-specific detail view of a conceptogram would be too costly in terms of computation time. Therefore, the implementation of a user data storage system as outlined in 4.5.3 is necessary. It can be seen that the "Representation CRUD Logic" has connections to all the elements of the backends layer: the space object storage for obtaining LSA information of the domain, the user data service interface (which restores the relevant conceptogram objects as part of the user object according to the session ID) and the conceptogram storage (which holds the conceptograms and, if the process of adjusting a conceptogram requires it, the term-to-term-matrix).

Comparison of graphs needs some parameter validation in the web service logic, as invocation of the graph comparison logic is only necessary if the parameters are sane (especially, that the user does not try to compare the same two conceptograms). Also, the comparison calculation logic needs access to the user data (holding the list of and metadata about conceptograms) and the conceptogram storage. The latter is external to the LSA web service, but may of course be located on the same physical system for ease of communication.

Sharing may be implemented similarly to the CRUD logic outlined above, with the "read"-action being a list of available sharing modes, "create" actually sharing the graph, "update" providing facilities to alter comments or tags associated with the shared graph, and "delete" to revoke public sharing of a graph. Again, the graph storage (for the actual graphs) and the user data storage (e.g. for preferences regarding public sharing on social media sites) are required.

6 Proof of Concept: A Service-oriented Framework for LSA Applications based on R

Evaluation of the architecture applicability has been done in the previous chapter. To prove the practical feasibility of the approaches described, this chapter aims to convey concrete techniques to realise the concepts proposed in this thesis with the help of a real-world example. First, the software products used as part of the example framework implementation, as well as the hardware used for testing, are shown. Then the implementation itself is outlined. Finally, a frontend application scenario chosen to utilize the backend infrastructure is shown: the placement experiment, which has coarsely been outlined in 5.4.

6.1 Tools

This section outlines the tools used to realize the example implementation. All chosen software components are open source projects and are compatible with most common operating systems.

6.1.1 R

"R is a free software environment for statistical computing and graphics." (R Development Core Team, 2010). It is a scripting language, which means that the source code is parsed at runtime, just before execution. Syntax and interpreter are optimised for handling of vector- and matrix-type data objects. To date, the core application is supplemented with over 2200 task-specific add-on packages.

R has been chosen as the core programming language of the project for a full-fledged LSA package being already available, as well as substantial capabilities in the area of natural language processing (Wild, 2009). Also, it is obvious that the enormous size of the objects handled in LSA would require the number of cross-application data conversion operations to be kept at a minimum, so ideally, the chosen framework should be able to handle all tasks from the lowest (i.e., matrix operations) to the highest (web service) level of processing.

6.1.2 RServe

The key idea behind RServe is to be able to instantiate an R session with its own workspace and directory from inside a different program, without having to link against R (Urbanek, 2009). It also provides the facilities to create a persistent R session, which exists in memory independent of the application that created it, and waits for a connection. Such connection utilizes the TCP/IP protocol for data transmission and allows for asynchronous execution of R processes. This allows for one R session being present all the time, with other instances connecting to it, initiating an operation, and retrieving the results later.

6.1.3 RApache

RApache (Horner, 2009) is a loadable module for the Apache HTTP server (The Apache Software Foundation, 2010a). The purpose of the module is to enable the invocation of an R instance whenever an associated script is called by the Apache server. The created instance contains references to the request parameters (e.g., the GET and POST parameters), which enables the dynamic handling of data provided by the user.

6.1.4 GotoBLAS

"Basic Linear Algebra Subroutines" (BLAS) are a set of software routines intended to perform vector and matrix operations. It is optimised to utilize low-level operations of the hardware architecture it is compiled for to allow for high processing speeds during such operations. Kazushige Goto implemented a version of a BLAS which he claims to be "currently the fastest implementations" (Goto, 2010) available. The BLAS is made available in R by referencing libGoto at compile time of R.

6.1.5 Hardware

All benchmarks in this thesis have been conducted on a development server which had two virtual machines installed, sharing the following hardware specification:

- Hardware
  - 2x Quad-Core Xeon 2.8GHz: 1 Quad-Core for each VM
  - 32 GB RAM (8x4GB dual rank DIMMs): 16 GB for each VM
  - 1.8 TB HD (6x300 GB)
- Software
  - Operating system: Debian Lenny (64 bit)
  - Apache httpd 2.2.10
  - MySQL Community Server 5.0.51a-17 (Debian) (MySQL Development Team, 2010)
  - PHP 5.2.6 featuring Zend Engine v2.2.0 (PHP Development Team, 2010)
  - R 2.8.1 with GotoBLAS 1.26

The large memory of this system left plenty of room to store spaces of 1GB and more in size.

6.2 Implementation of the Four Layers of the LSA Client/Server Architecture

As outlined and thoroughly discussed in section 4, the web service architecture should be composed of four layers. Following the previous considerations, this section is going to outline an R-based LSA framework for web services, which is composed of these four layers. The implementation of the backends layer as well as the aspects to consider when implementing components for the other layers will be shown, giving example approaches where applicable. As a reference, the final architecture is displayed in advance of the actual technical description in Figure 6.1 below.

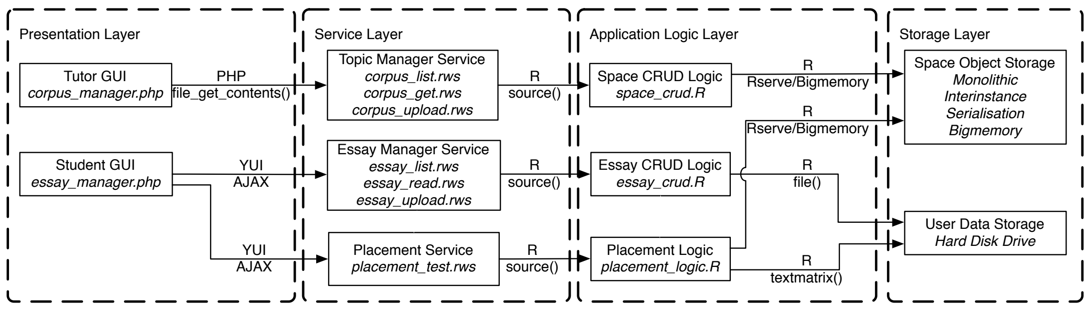


Figure 6.1: Layer decomposition of the proof of concept implementation

6.2.1 Backends Layer

This layer comprises the core functionality of the framework. It handles the generation, storage, retrieval, and processing of LSA spaces. Its input is data that is to be processed into a space object, as well as identification tokens used to locate spaces in the storage. Its output is process monitoring data as well as the space objects themselves.

Infrastructure for LSA Logic: The Space Warehouse

R is a script language and therefore allows for handling of application logic in a different way than most languages requiring the code to be compiled to binary instructions. This allows for logic being defined as “function”-object, which may be treated like a variable, but may also be executed. So instead of extending a basic LSA object with concrete implementations, the logic is passed to the warehouse as a such a “function”-object, which is defined in the application layer. This mechanism resembles the principle of a framework as defined in section 1.2.4.

Decoupling the LSA logic from the service layer framework has the following advantages:

- The logic can be performed within the warehouse as outlined in section 6.2.1. This can be advantageous in some cases (see section 6.2.1).
- A dedicated LSA logic developer can implement the logic function with neither knowing, nor caring about the actual implementation of the space warehouse on the one hand and the service layer on the other hand.
- The implementation of the logic might be sub-optimal in means of variable copying. The availability of the logic (which is a `language`-object in R) to the abstraction layer allows for adjustment the actual implementation on-the-fly using R’s language manipulation. For example, Oehlschlägel et al. (2008) show how R.ff is able to make such modifications to to program logic on the fly to make it compatible with a more efficient storage mechanism.

Implementers of LSA logics may face several problems concerning the retrieval of the space objects:

- Storage requirements for spaces: Depending on the application scenario of the LSA logic, spaces can be very large and might not fit into the main memory of the computer.

- Transfer bottlenecks: Depending on the storage medium, retrieval of a space from that medium may be slow.
- Retrieval overheads: storage mechanisms like compression or serialisation create a computational overhead when retrieving a space.
- Simultaneous access: Some applications scenarios (like web services) may require instant access to space objects for multiple clients at the same time.

The considerations above lead to the development of three types of space storage mechanisms, which shall be outlined now.

Monolithic Approach ("Mainframe-like Warehouse")

The monolithic warehouse approach keeps a central R instance permanently open, holding all space objects previously calculated in main memory. LSA application logic is passed into this R instance and executed there, locally. This approach eliminates all overhead created by copying large space objects between storage media as they are accessed directly from memory. It has to be kept in mind that the central R instance holding the spaces is unavailable to other requests until the LSA logic has finished.

This mechanism is especially useful when instructions are performed in a short period of time, but on very large corpora, because in this case the lock-up of the warehouse might be shorter than in inter-instance mode if the LSA logic can be performed in less time than creating a copy in memory would take, and deserialisation of the space from hard disk might be outperformed by both RAM-based approaches.

In case of multiple instances trying to access the warehouse, all but one will get a "temporarily unavailable"-signal, instructing them to try again. This mechanism is called "polling" (Papazoglou, 2008).

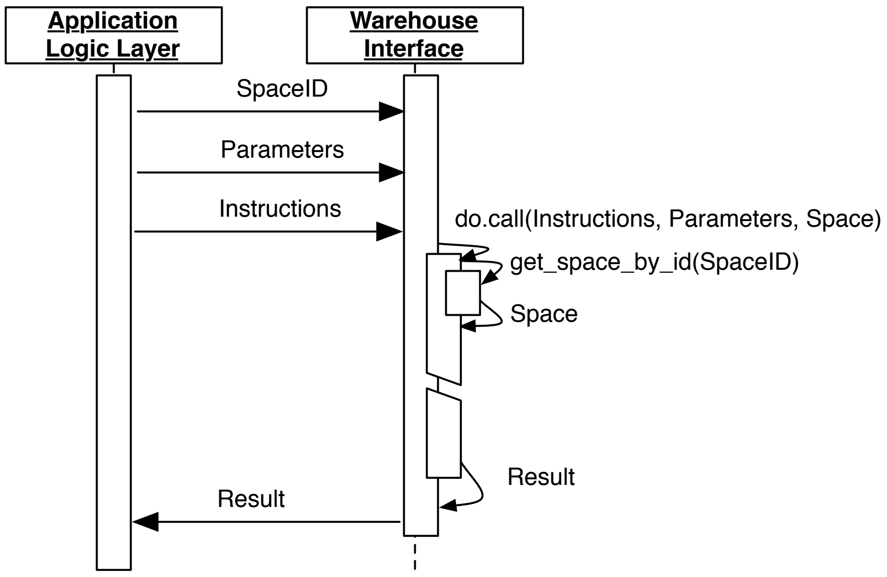


Figure 6.2: Sequence diagram of monolithic logic performance

Inter-Instance Copy ("RAM Storage")

The inter-instance copy approach keeps — like the monolithic approach — all spaces in main memory. Application logic is - in contrast - not executed in the storage R instance, but rather, a copy of the original object is passed to the RApache instance, freeing the central R instance's access interface again as soon as the copy has been generated.

Serialisation ("Hard Disk Storage")

The serialisation approach keeps all space objects in a binary file on the server's hard disk. This has the advantage that on most servers, HDD space will by far exceed main memory, and for that reason, storage should be less a problem. On the downside, (de-)serialisation of space objects may be - depending on the hard ware used - a time consuming task which may slow the LSA process. Note that, if the amount of data to be stored is not too large, and it turns out that the other warehouse types are superior in a given scenario only because of the slow transfer

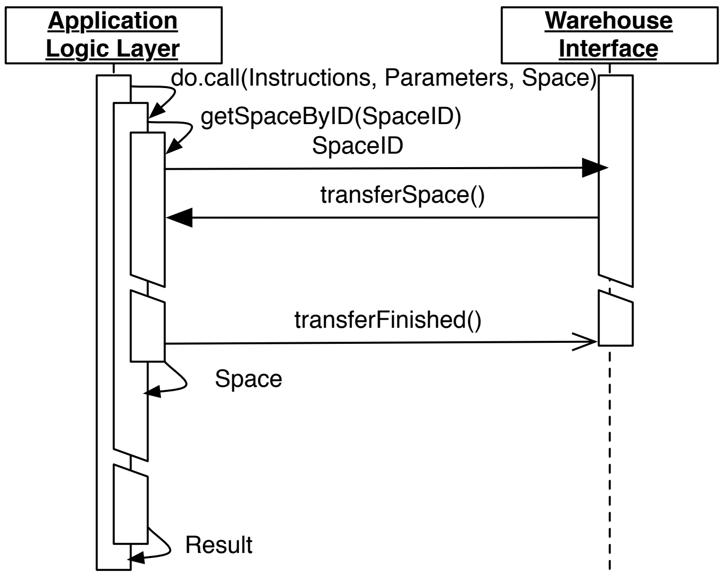


Figure 6.3: Sequence diagram of inter-instance copying space acquisition

rate of HDDs to main memory, a RAM-disc (virtual hard disc storing data in a partition of the main memory) would be an option.

Choosing the Appropriate Warehouse Type

Table 6.5 gives an overview of the available warehouse modes considering the issues of space object retrieval outlined in section 6.2.1

Alternative Implementation: bigmemory

Bigmemory (Kane and Emerson, 2010) is an attempt to bypass the limitations of classic in-memory handling of matrices in R, and the tiresome process of either (de-)serializing data repeatedly or using non-standard mechanisms to keep it in memory. Bigmemory allows for R to save so-called descriptor-files to disk which represent links to data stored in memory, and backing-files which are filled with overflow data if the RAM of a machine is exceeded.

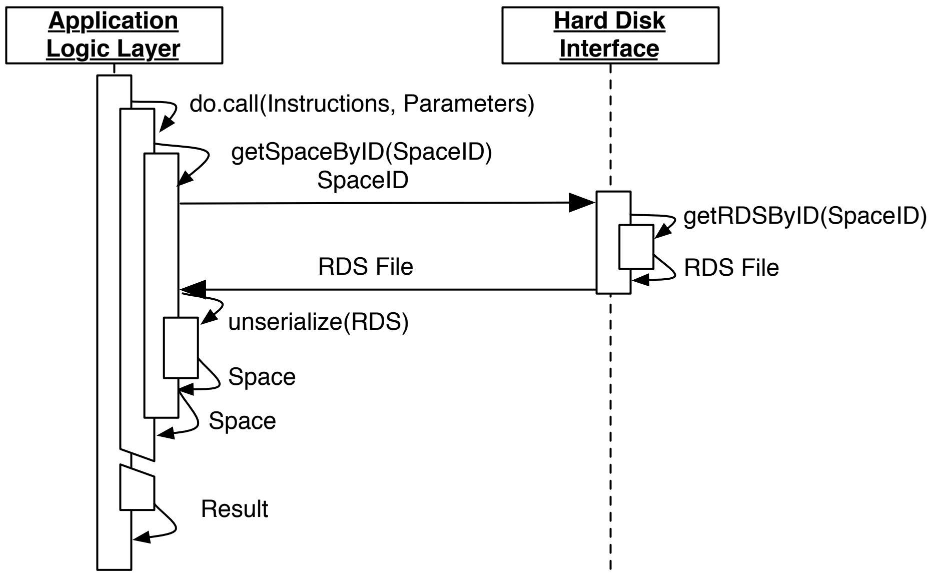


Figure 6.4: Sequence diagram of serialisation-based space acquisition

	Monolithic	*Inter-Instance Copy*	*Serialisation*
Storage Limit	RAM		HDD
Retrieval overhead	None	RAM-RAM-Copying	(De-)serialisation
Warehouse unavailable	Until logic performed	Until space copied	Until file retrieved
Access mode	Polling		Independent

Figure 6.5: Comparison of space warehouse modes

The advantage of the shared memory mechanics is that multiple processes can operate with data contained in the same matrix object held in memory. This allows for a reduction in redundancy as well as improved performance in space retrieval. Bigmemory's sister-package "synchronicity" even allows for parallel execution of "mutex" (mutually exclusive) matrix modification operations, which means that a single matrix can be modified by multiple processes as long as each process only works on a "section" of the matrix that no other process has access to. This is ensured using sophisticated locking mechanisms provided by the package.

Using a current version of bigmemory, which still lacks basic matrix operations, the package can actually just be used as a space storage that does not have a deserialisation-overhead: the space is stored as-is in shared memory and can be retrieved (and copied to memory) comparably fast. This process is outlined in Figure 6.6. The matrices contained in the initial LSA space (U_k, V_k and Σ_k, see 2.1.2) are each extracted from the space and stored in shared memory as bigmemory objects, the references to the matrices in the space are replaced with `null` values. Then, the space is stored as RDS. During retrieval, the RDS file is read back, the bigmemory-objects are read from shared memory (i.e. copied to the active process), converted back to R-matrices to enable matrix calculus, and references to them restored in the LSA space object.

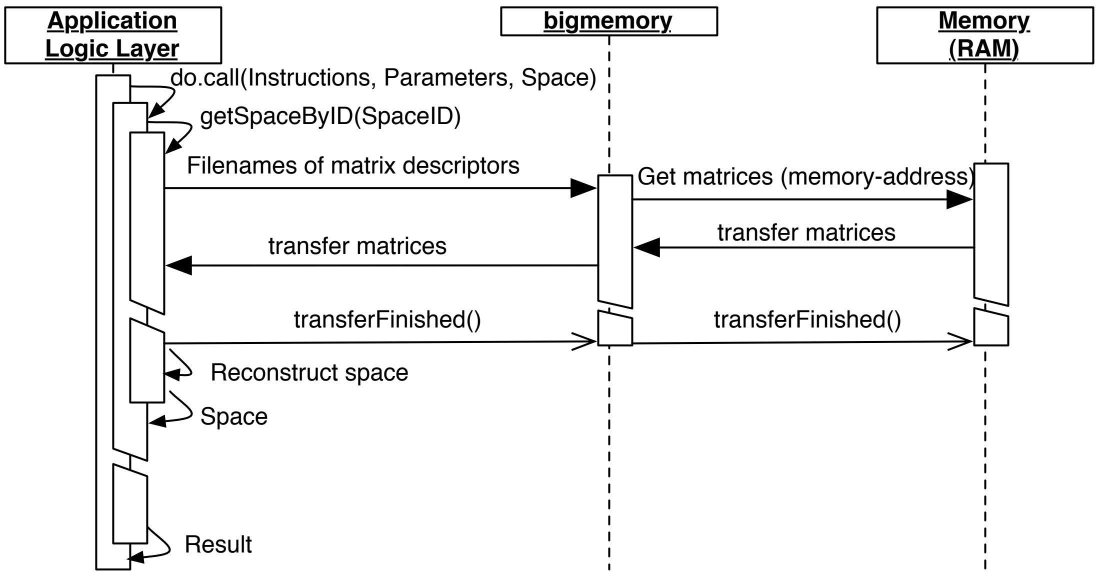


Figure 6.6: Sequence diagram of bigmemory space acquisition

Performance

On page 76, the performance of the warehouse modes is compared for various usage scenarios:

1. "Storage" is the duration it takes to store a space of given size to the respective storage medium.

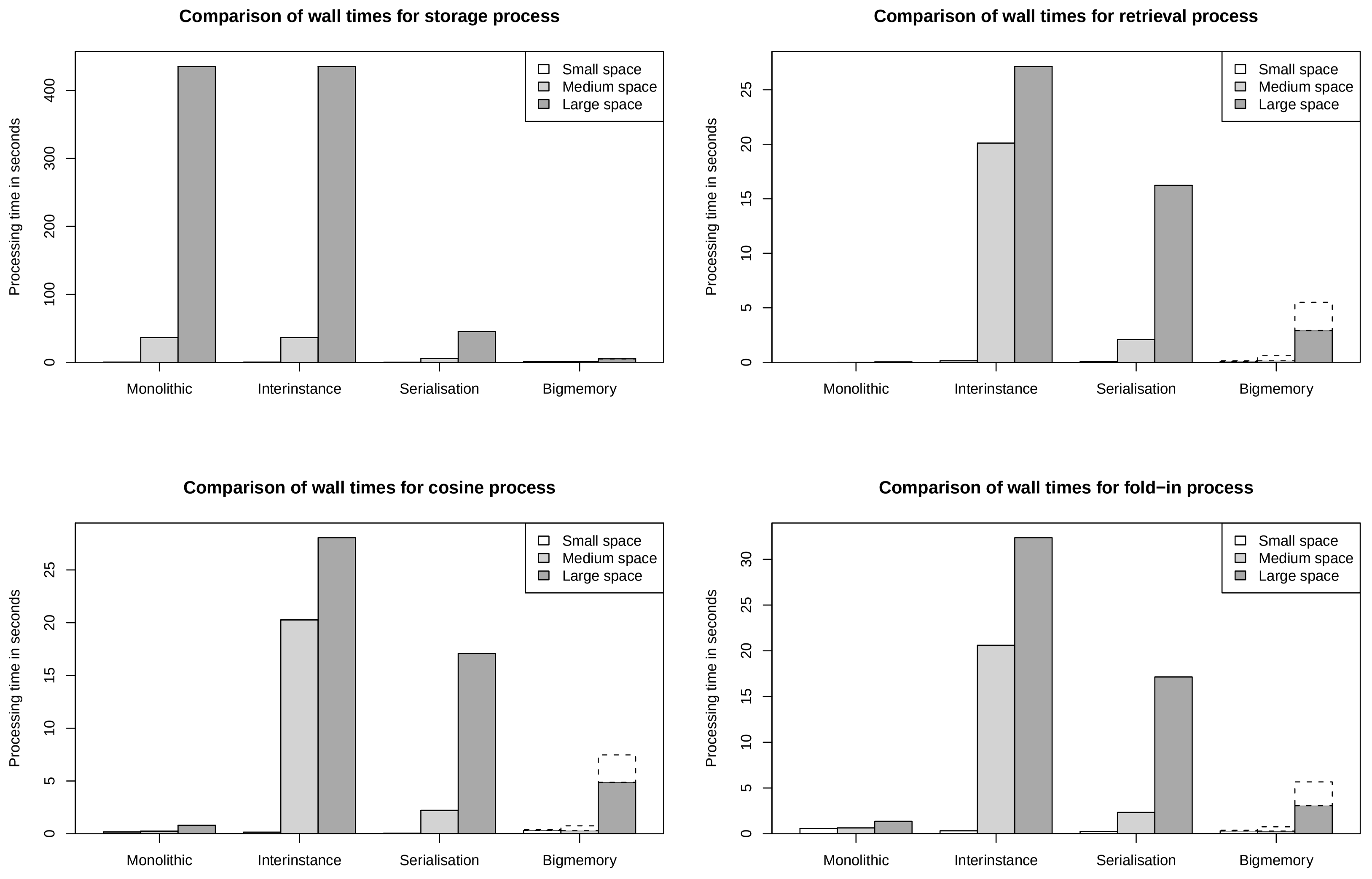


Figure 6.7: Comparison of warehouse performance on the basis of various usage scenarios.

2. “Retrieval” is the duration it takes for the space to be retrieved from the storage medium.

3. “Cosine” is the wall time of performance of a simple cosine calculation between two terms (“car” and “list”).

4. “Fold-in” is the wall time of folding a small text into a space.

The sizes of the spaces have been determined using the R function `object.size`:

1. “Small space”: 4.092.144 Byte

2. “Medium space”: 101.259.936 Byte

3. “Large space”: 627.039.944 Byte

Note that LSA logic is only performed, but no result is returned during this benchmark. Therefore, if the result objects are large in size, another level of overhead may arise from the fact that “monolithic” and “interinstance” warehouses have to copy the result objects from the R warehouse instance to the calling R client.

Connection overhead in “serialisation” mode consists solely of accessing the reference to the RDS file on the hard disk, which is in the area of milliseconds on the hardware of the test server. Retrieval overhead in monolithic mode consists solely of accessing the memory reference to the object, which is almost instant. “Bigmemory” currently lacks capabilities to perform basic linear algebra calculations – for this reasons, a severe overhead is created by the process of conversion from bigmatrix to R-matrix objects after a bigmatrix has been retrieved. This additional overhead is displayed in Figure 6.7 by the dotted boxes above the bigmatrix execution times. This is still quite inefficient compared to the abilities a space warehouse would provide that allows computation on shared-memory objects on a by-reference basis. Luckily, the authors of bigmemory have announced that the sister-package “bigalgebra” will be able to include all mathematical operators necessary for processing LSA objects containing bigmatrix-matrices (namely, vector- and matrix-calculus and SVDs powered by BLAS and LAPACK). At the current development stage, however, these operations are not handled transparently, which means that the code of the LSA package would have to be adjusted to make the correct calls if the input objects contain bigmatrix-matrices. The authors have, however, also announced more userfriendly wrappers for the functions, which might mean that one day a “generic” for R’s matrix multiplication function `%*%` will be implemented which is able to handle the bigmatrix objects.

Still, a direct comparison shows that the bigmemory approach is in every way superior to all RServe implementations except the “monolithic” mode, which does not have a retrieval overhead at all since the logic is transferred to the space, not the space to the logic. Therefore, if “monolithic” mode is not an option (e.g. due to severe multi-user parallel access requirements for the space warehouse), “bigmemory” is the correct choice. The R community has put significant efforts into areas of high-performance computing during the period of creation of this thesis. Bigmemory is better suited for the purpose or holding large objects in RAM than RServe is, as the communication between shared memory and the R instance does not require the overhead of a communication protocol between two R instances in the case of the former, which it does for the latter.

6.2.2 Application Logic Layer

This layer contains the functions tailored towards the specific applications of LSA as a service provider. Its input is a set of execution parameters passed by the service layer, and the spaces provided by the backends layer. Its output is an R object that represents the result of the LSA process.

During the setup of the LSA framework, a set of functions must be implemented. They must be able to perform the LSA logic with only the parameter list passed by the service layer, which is the gateway for client applications to access this functionality. Functions may serve utility purposes, such as the maintenance logic (including generation, deletion and modification of spaces), or task-specific logic such as the folding-in of a text into a pre-defined space for relevance checks. They can utilize the backends layer to retrieve the spaces needed to perform the algorithm, independent of the actual warehouse implementation in use.

The joint environment of functionality and spaces is created by the backends layer using dependency injection (Fowler, 2004). In such environment, spaces can be accessed by the applications layer using a common interface. This interface simply consists of two methods:

1. `get_space_by_id(space_id)`: Function to retrieve the space with identification (ID) *space_id*.
2. `set_space_by_id(space_id, space)`: Function to store a space passed as a parameter into the space warehouse. Depending on whether a space with the ID *space_id* is already present, a new item in the warehouse will be created, or an existing one will be overwritten.

These functions have different implementations depending on the chosen warehouse type, but their outside interface remains the same, enabling implementation of the LSA logic completely decoupled from the actual backend behaviour.

The LSA application logic then returns a result object, which may be any R object, including lists, arrays or even binary image data which may be generated by a graphics implementation, depending on the task. This data is then passed to the service layer for transformation and communication to the presentation layer.

6.2.3 Service Layer

This layer contains routines to handle user requests and responses to them. Its input is a set of user request parameters passed by the presentation layer. Its output is text, formatted in a way that the contained data is interchangeable with the client - normally, some form of XML string.

The service framework relies on code written in R which is used to transform, validate, and then communicate the parameters to the application logic layer.

Invocation of the underlying R-scripts is handled by an Apache server, which is equipped with the “RApache” module. On arrival of a request for an R web service at the Apache server, RApache invokes an instance of the R shared library, executes the required R web-service script and enables the R framework to access all information that has been passed to the Apache server by the client (via the HTTP protocol).

After computation of the application logic — successful or not — the service layer returns a custom XML structure representing the result object. The actual structure to be used can be freely chosen by the web service implementer. This individual choice is given to developers as it enables quick development of XML interactions for simple tasks.

Still, a more sophisticated communication solution is easily achievable by replacing the service layer with a toolkit capable of more comprehensive interaction. A possible solution would be a full-fledged communication architecture based on a standard XML protocol (e.g., SOAP), which would be the case if e.g., XML-RPC is chosen as the standard protocol. Another promising approach is the Biocep-R project (Biocep-R Team, 2010), which acts as an intermediate layer between R and a Java Virtual Machine. Such JVM could then be embedded in an Apache Tomcat server (The Apache Software Foundation, 2010b) to utilize its web service abilities.

6.2.4 Presentation Layer

This layer is used to create requests to the LSA system. It interacts with the application layer by creating request strings and sending them to the LSA server.

Most of the implementation effort is put into this layer when using the LSA web service framework. Due to the various applications of LSA, the client can be anything from a forum plug-in (Landauer, 2007) to a web-based learning tool (see section 6.3). The client may use the result data by displaying it as a graphical component in a UI, or process the data further in a service chain.

Regardless of the actual implementation scenario, clients can use any HTTP based communication mechanism to access web services via RESTful requests (see 4.5.1). The server hosting the service framework handles the request by invoking the applications on the service layer and passes on any parameters that have been sent to the server.

A concrete implementation of a web-based learning software utilizing the LSA web service framework is outlined in the following section.

6.3 Sample Application: The Placement Experiment

The following scenario has been created as a prototype, with the aim of discovering a first set of technologies that may be used to realise the service architecture concept of section 6.2. Essay scoring is a process in which a topic is defined by a tutor using text corpora specific to this topic, and essays written by students can then be rated using a scoring mechanism. This prototype uses LSA to generate a space for the topic and to fold in a student's essay, finding the score using Pearson correlation as a proximity measure. On the server machine, an Apache server is listening for REST-style requests for *.rws scripts, which are R scripts that can be executed by the `RApache` Apache module. These scripts execute the request and return custom XML data as a result.

6.3.1 Tutor's View

Corpus and topic administration is realised using a frontend based on PHP (PHP Development Team, 2010) to generate the requests. Using a PHP command as shown in Figure 6.8, PHP generates a request like in Figure 6.9 at runtime. The server will return a list of existing corpora as in Figure 6.10.

```
$requestURL = 'http://host.com/webservice/corpus_list.rws';
$xml_response = file_get_contents($requestURL);
```

Figure 6.8: PHP instructions used to generate a request

```
GET /web-service/corpus_list.rws HTTP/1.1
User-Agent: PHP/5.2.4-pl2-gentoo
Host: host.com
Accept: */*
```

Figure 6.9: HTTP request for a list of existing corpora

This XML data is then processed using PHP to generate a graphical user interface (GUI) for topic administration. Upload of a corpus is done using HTTP POST utilising the RFC 1867, which is commonly used by browser-based forms. The form itself has been generated by the PHP script and is then utilised by the client browser.

Using these technologies (REST-style requests for a small set of parameters, RFC 1867 style POST-upload for corpora), all functionality from the tutor's view is implemented, providing a GUI for the topic management.

Space generation jobs are passed to the server using the message queuing mechanism outlined earlier. A GET request states the IDs of the corpora to be put into the space, and an R script on the server generates the space as soon as computation capacity is available, utilising the package "lsa". The spaces are then stored in a persistent R instance (using Rserve) that acts as the space object storage outlined in section 6.2.1. Therefore, the spaces are held in RAM and are highly available. The status of generation can be monitored using the GUI (see the bottom of Figure 6.13).

```
HTTP/1.1 200 OK
Date: Wed, 29 Oct 2008 12:55:27 GMT
Server: Apache
Transfer-Encoding: chunked
Content-Type: text/xml

358
<WSR:webServiceResponse
   xmlns:WSR="http://www.w3c.org/2002/ws/"
   xmlns:ltfll="http://www.ltfll-project.org/">
<ltfll:corpus id="1">
<ltfll:title>Medical Texts</ltfll:title>
<ltfll:original_filename>med.all</ltfll:original_filename>
<ltfll:textsize>1114373</ltfll:textsize>
</ltfll:corpus>

<ltfll:corpus id="2">
<ltfll:title>CISI Test Texts</ltfll:title>
<ltfll:original_filename>cisi.all</ltfll:original_filename>
<ltfll:textsize>2561998</ltfll:textsize>
</ltfll:corpus>
</WSR:webServiceResponse>
```

Figure 6.10: XML response from the server

```
POST /web-service/corpus_upload.rws HTTP/1.1
Host: host.com
Content-Type: multipart/form-data; boundary=----------------------cc1b3257ba
Content-Length: 309

-----------------------cc1b3257ba
Content-Disposition: form-data; name="corpus[1]"; filename="test.txt"
Content-Type: text/plain

This is a simple text corpus.

-----------------------cc1b3257ba
Content-Disposition: form-data; name="title[1]"

Title of the test
-----------------------cc1b3257ba--
```

Figure 6.11: HTTP request for upload of a new corpus

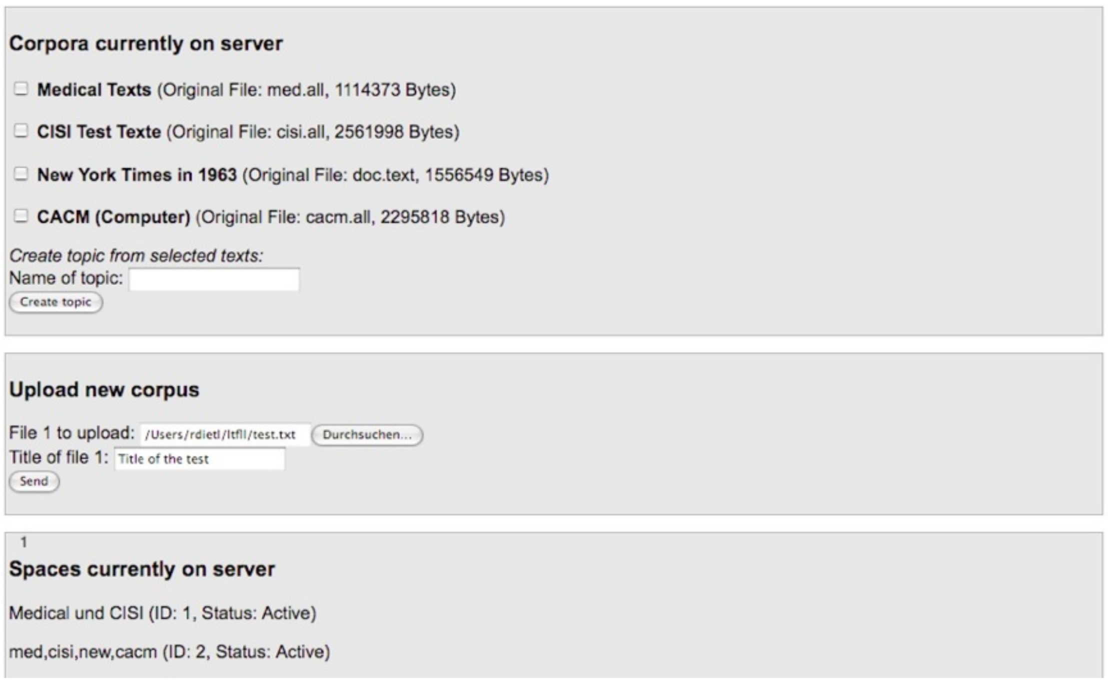
```
HTTP/1.1 200 OK
Date: Wed, 29 Oct 2008 13:10:22 GMT
Server: Apache
Transfer-Encoding: chunked
Content-Type: text/xml

9a
<webServiceResponse xmlns="WSR" xmlns:ltfll="LTfLL">
   <ltfll:success>
      The file test.txt has successfully been saved.
   </ltfll:success>
</webServiceResponse>
```

Figure 6.12: XML response upon upload

6.3.2 Server Application: Space Maintainer

As already outlined in section 2.1 and thoroughly discussed in Dietl (2009), the generation of a space from a text corpus can be a lengthy operation. Still, the Service must be available, even if, during peak times in a university environment, the generation of multiple spaces at once is triggered. Therefore, a solution has to be found that ensures that space warehouse maintenance tasks are not blocking calls, but can be performed simultaneously.

Figure 6.14 on page 87 shows an approach to the asynchronous generation of multiple spaces by utilizing multiple CPUs on a server. What happens is that multiple clients send text corpora and associated metadata to the space creation web service gateway via a carrier, e.g., the internet. The space maintenance logic stores the corpus data in a file on the file system, and parameters into a MySQL database (MySQL Development Team, 2010), including metadata that enables the list of jobs to be handled as a queue. Simultaneously, a CRON job (a process triggered for execution every minute on the server) is executed, checking whether the queue



Corpora currently on server

☐ **Medical Texts** (Original File: med.all, 1114373 Bytes)

☐ **CISI Test Texte** (Original File: cisi.all, 2561998 Bytes)

☐ **New York Times in 1963** (Original File: doc.text, 1556549 Bytes)

☐ **CACM (Computer)** (Original File: cacm.all, 2295818 Bytes)

Create topic from selected texts:
Name of topic:
Create topic

Upload new corpus

File 1 to upload: /Users/rdietl/ltfll/test.txt Durchsuchen...
Title of file 1: Title of the test
Send

1
Spaces currently on server

Medical und CISI (ID: 1, Status: Active)

med,cisi,new,cacm (ID: 2, Status: Active)

Figure 6.13: Screenshot of the PHP-based tutor GUI

manager contains new jobs to be performed and, if true, executing the topmost entry in the queue. This results in a separate R session being created, which is contained in an RServe instance. The job manager disconnects from the RServe session, making the space generation process asynchronous. The reference to the running process is stored in a instance monitor database. As soon as the space generation finishes, it notifies the instance monitor of the successful completion of the job. Furthermore, CRON regularly triggers the execution of the space warehouse maintainer script, which polls the instance monitor for finished processes. If one is found, the process attaches to the RServe instance that holds the — now finished — space, and to the space warehouse, so the space can be moved from the former to the latter.

6.3.3 Student's View: Essay Scoring

For the student's side, an "asynchronous JavaScript and XML" (AJAX) based GUI has been developed. It utilises the same technologies as the tutor's side. Creation

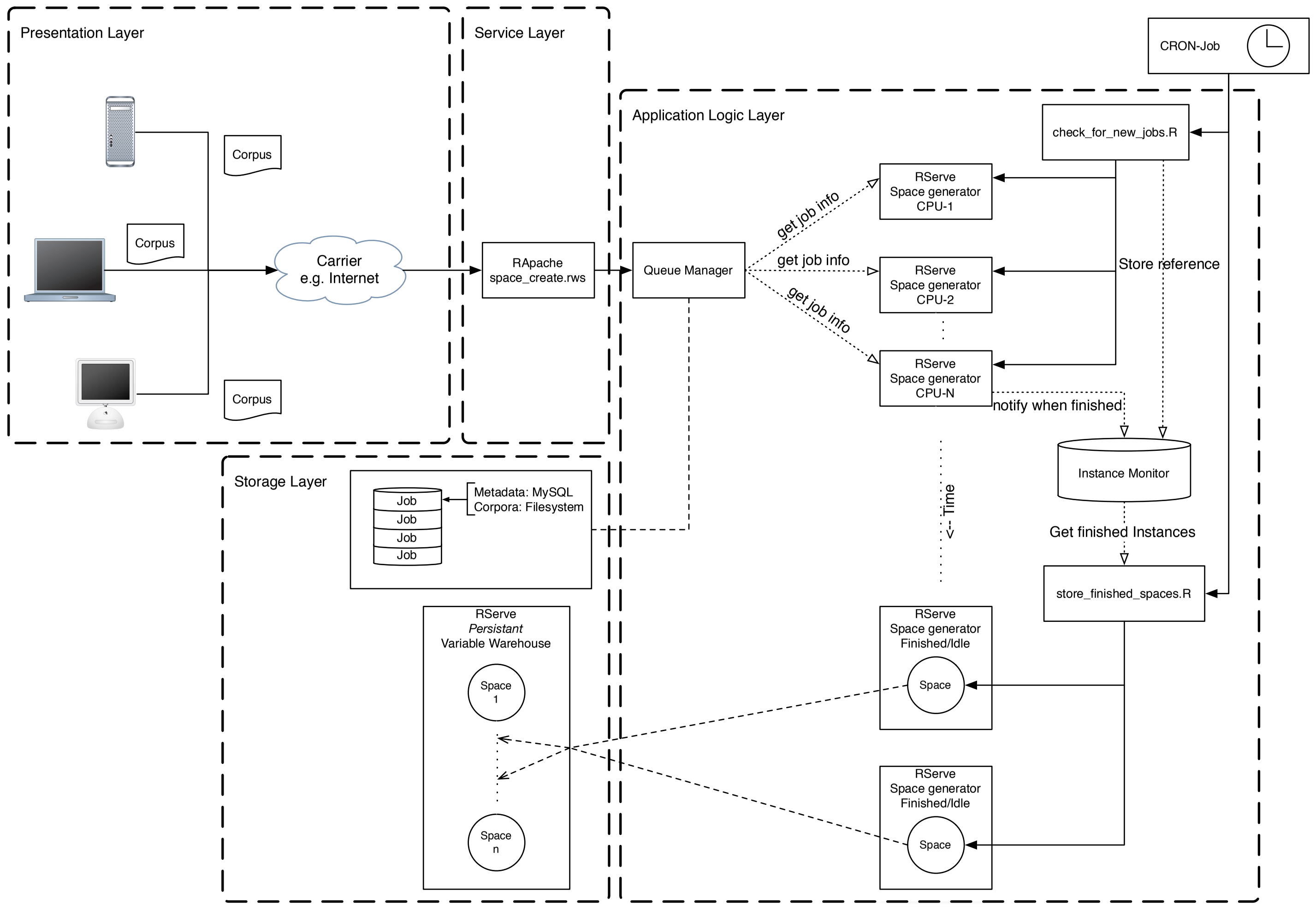


Figure 6.14: An architecture which enables the simultaneous processing of text corpora to LSA spaces

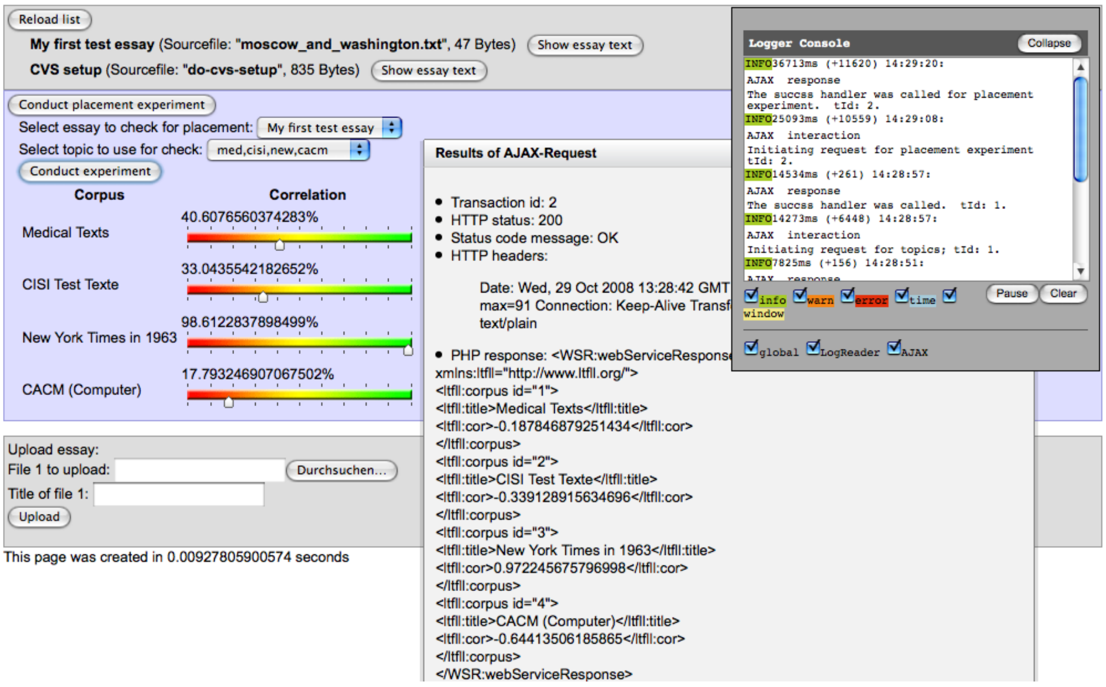
of GET and POST requests is handled using the “Yahoo! User Interface” (YUI) library (Yahoo! Developer Network, 2009) module “connection”, which allows for asynchronous invocation of the R services.

The R service itself first retrieves the text of the essay (which is kept in a database) and stores it in a text file. Afterwards, the code shown in Figure 6.15 is executed to retrieve the correlations of the essay file to the chosen topic space.

```
space_id <- as.integer(GET$space_id);

parameters<-list(essay_file=essay_file, space_id=space_id);

logic<-function(essay_file, space_id){
        space = get_space_by_id(space_id);
        trm_red = as.textmatrix(space);

        tem = textmatrix(essay_file, vocabulary = rownames(trm_red));
        tem = lw_bintf(tem) * gw_idf(trm_red);
        tem_red = fold_in(tem, space);

        cors = cor(tem_red[,basename(essay_file)], trm_red);
        cors;
}
cors<-lsa_perform_logic(logic, parameters);
```

Figure 6.15: Declaration of an R function holding the application logic for a placement experiment

Afterwards, on the service layer, the result object is converted into an XML response, which is then returned to the client. The client (here: the browser) then displays the result by creating visual indicators using JavaScript and the *slider* component of the YUI library, which is depicted in Figure 6.16.



My first test essay (Sourcefile: "**moscow_and_washington.txt**", 47 Bytes) Show essay text

CVS setup (Sourcefile: "**do-cvs-setup**", 835 Bytes) Show essay text

Select essay to check for placement: My first test essay

Select topic to use for check: med,cisi,new,cacm

Corpus	**Correlation**
Medical Texts	40.6076560374283%
CISI Test Texte	33.0435542182652%
New York Times in 1963	98.6122837898499%
CACM (Computer)	17.793246907067502%

Upload essay:

File 1 to upload:

Title of file 1:

This page was created in 0.00927805900574 seconds

Results of AJAX-Request

- Transaction id: 2
- HTTP status: 200
- Status code message: OK
- HTTP headers:

  Date: Wed, 29 Oct 2008 13:28:42 GMT
  max=91 Connection: Keep-Alive Transf
  text/plain

- PHP response: <WSR:webServiceRespons

```
xmlns:ltfll="http://www.ltfll.org/">
<ltfll:corpus id="1">
<ltfll:title>Medical Texts</ltfll:title>
<ltfll:cor>-0.187846879251434</ltfll:cor>
</ltfll:corpus>
<ltfll:corpus id="2">
<ltfll:title>CISI Test Texte</ltfll:title>
<ltfll:cor>-0.339128915634696</ltfll:cor>
</ltfll:corpus>
<ltfll:corpus id="3">
<ltfll:title>New York Times in 1963</ltfll:title>
<ltfll:cor>0.972245675796998</ltfll:cor>
</ltfll:corpus>
<ltfll:corpus id="4">
<ltfll:title>CACM (Computer)</ltfll:title>
<ltfll:cor>-0.64413506185865</ltfll:cor>
</ltfll:corpus>
</WSR:webServiceResponse>
```

Logger Console

```
INFO36713ms (+11620) 14:29:20:
AJAX  response
The success handler was called for placement
experiment.  tId: 2.
INFO25093ms (+10559) 14:29:08:
AJAX  interaction
Initiating request for placement experiment
tId: 2.
INFO14534ms (+261) 14:28:57:
AJAX  response
The success handler was called.  tId: 1.
INFO14273ms (+6448) 14:28:57:
AJAX  interaction
Initiating request for topics; tId: 1.
INFO7825ms (+156) 14:28:51:
AJAX  response
```

✓ info ✓ warn ✓ error ✓ time ✓ window

✓ global ✓ LogReader ✓ AJAX

Figure 6.16: Screenshot of student's GUI based on the Yahoo! User Interface library

7 Conclusion and Outlook

This thesis attempt to find a way to overcome the challenges of distributed systems when using LSA in a service oriented architecture. In the course of this, reasons for distributing LSA procedures have been found, and requirements for a reference architecture have been set up.

As a result, a 4-tier client-server architecture has been suggested. This architecture allows for decoupling of homogenous system components with interfaces that allow for distribution of sub-procedures. The introduction of a separate service layer aids the use of the LSA components in a service oriented architecture as a black-box application on the one hand, without bothering the implementers of LSA application logic with web service specifics on the other hand.

The suggested reference architecture successfully fulfills all requirements set up before. It utilises best-practice approaches that have been tried and tested for decades in distributed system contexts, and solutions to common problems resulting from these practices have been suggested. Furthermore, it provides the facilities to make use of advanced LSA algorithms which can operate in clustered environments.

The solution was compared to currently existing approaches to provide LSA as a web service, and the applicability of the reference architecture to scenarios from practical contexts has been shown. A demonstration implementation (a classroom demonstration of the "placement experiment") has been developed to prove the feasibility of an application utilising the reference architecture outlined in this thesis.

The suggested reference architecture, however, is no more than what its name says: a reference, aimed to determine a beneficial way of decoupling components. Although a set of possible middleware frameworks and open source components that can act as parts of the respective layers has been shown, this list is not comprehensive, and such a list would be beyond the scope of this thesis. Further studies must be conducted on the optimal embodiment of the components on each layer. The future might bring even better mechanisms for shared use of in-memory space objects and intermediate results, LSA implementations in different programming languages may rise demand for cross-platform interfaces to the storage layer, and the rapid development of SOAP and its competitors may open up completely new contexts and approaches to providing LSA as a component in a service oriented architecture.

Also, research concerning possibilities of distributing LSA calculation has only been conducted with consideration of existing technologies. Development of algorithms that enable distributed calculation of SVDs, techniques for efficient execution of linear algebra operations on large matrices (which may be distributable as well), and even the reduction of bottlenecks on the hardware level (e.g. access times of hard disks), are fields of research that shall remain potential subject matters for future works.

8 Appendix

List of Figures

Bibliography

Augeri, C. J., Bulutoglu, D. A., Mullins, B. E., Baldwin, R. O., and Baird, L. C. (2007). An analysis of XML compression efficiency. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, New York, NY, USA. ACM.

Avgeriou, P. and Zdun, U. (2005). Architectural Patterns Revisited - A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPloP 2005)*, Irsee, Germany.

Baresi, L. and Di Nitto, E., editors (2007). *Test and Analysis of Web Services*. Springer.

Beck, K. (2003). *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley.

Berry, M. W., Dumais, S. T., and O'Brien, G. W. (1995). Using Linear Algebra for intelligent information retrieval. *SIAM Review*, 37:573–595.

Biocep-R Team (2010). Biocep-R. http://biocep-distrib.r-forge.r-project.org/ (last checked on 2010-02-16).

Burek, G., Berlanga, A. J., Kalz, M., Braidman, I., Smithies, A., Wild, F., Osenova, P., Simov, K., Hoisl, B., Lemnitzer, L., Stoyanov, S., van Rosmalen, P., Hensgens, J., Regan, M., Bruggen, J. V., and Armitt, G. (2008). Language Technologies for Lifelong Learning: Project Deliverable Report D4.1. http://hdl.handle.net/1820/1761 (last checked 2010-09-09).

Burek, G., Gerdemann, D., Hoisl, B., Koblischke, R., Braidman, I., Smithies, A., Haley, D., Wild, F., Osenova, P., Simov, K., Berlanga, A., and Mauerhofer, C. (2010). Language Technologies for Lifelong Learning: Project Deliverable Report D4.2. http://hdl.handle.net/1820/2301 (last checked 2010-09-08).

Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., and Matera, M. (2003). *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition.

Dietl, R. (2009). Latent semantic analysis: Overview and applications. BSc thesis at the Vienna University of Economics and Business Administration. Available online at http://epub2.wu-wien.ac.at/dyn/openURL?id=oai:epub2.wu-wien.ac.at:epub-wu-01_10c2.

Evans, E. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Evdemon, J. (2005). Principles of service design: Service patterns and anti-patterns. http://msdn.microsoft.com/en-us/library/ms954638.aspx (last checked 2010-09-20).

Feinerer, I. (2008). *A Text Mining Framework in R and Its Applications*. PhD thesis, Department of Statistics and Mathematics, Vienna University of Economics and Business Administration.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Fowler, M. (2003). *Patterns of enterprise application architecture*. Pearson Education, Boston, MA, USA.

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html#InversionOfControl (last checked 2010-02-18).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the ECOOP'93 Conference, Kaiserslautern, Germany.*

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series, Reading, MA, USA.

Giesbers, B., Taddeo, A., van der Vegt, W., van Bruggen, J., and Koper, R. (2007). *A Question Answering service for information retrieval in Cooper.* Paper presented at the Cooper workshop, Crete, Greece.

Goto, K. (2010). GotoBLAS FAQ. http://www.tacc.utexas.edu/resources/software/gotoblasfaq/ (last checked 2010-02-11).

Hensgens, J., Rusman, E., Bruggen, J. V., Armitt, G., Osenova, P., and Simov, K. (2009). Language Technologies for Lifelong Learning: Project Deliverable Report D3.2. http://dspace.ou.nl/handle/1820/2038 (last checked 2010-09-13).

Hentrich, C. and Zdun, U. (2006). Patterns for Process-Oriented Integration in Service-Oriented Architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPloP 2006)*, pages 1–45, Irsee, Germany.

Horner, J. (2009). rapache: Web application development with R and Apache. http://biostat.mc.vanderbilt.edu/rapache/ (last checked 2010-01-21).

IEEE (2000). *Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000.*

Kalz, M., Drachsler, H., van der Vegt, W., van Bruggen, J., Glahn, C., and Koper, R. (2009). A placement web-service for lifelong learners. In Tochtermann, K. and

Maurer, H., editors, *Proceedings of the 9th International Conference on Knowledge Management and Knowledge Technologies*, pages 289–298, Graz, Austria. Verlag der Technischen Universität Graz.

Kane, M. J. and Emerson, J. W. (2010). *Documentation - The bigmemory Project*. Yale University, http://sites.google.com/site/bigmemoryorg/research/documentation.

Kontostathis, A., Pottenger, W. M., and Davison, B. D. (2005). Identification of critical values in latent semantic indexing. In *Foundations of Data Mining and Knowledge Discovery*, pages 333–346. Springer Verlag.

Landauer, T. K. (2007). LSA as a Theroy of Meaning. In Landauer, T. K., McNamara, D. S., Dennis, S., and Kintsch, W., editors, *Handbook of Latent Semantic Analysis*, chapter 1, pages 3–34. Lawrence Erlbaum Associates, Philadelphia, PA, USA.

Landauer, T. K. and Dumais, S. T. (1997). A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge. *Psychological Review*, 104(2):211–240.

Martin, D. I. and Berry, M. W. (2007). Mathematical Foundations Behind Latent Semantic Analysis. In Landauer, T. K., McNamara, D. S., Dennis, S., and Kintsch, W., editors, *Handbook of Latent Semantic Analysis*, chapter 2, pages 35–56. Lawrence Erlbaum Associates, Philadelphia, PA, USA.

Martin, D. I., Martin, J. C., Berry, M. W., and Browne, M. (2007). Out-of-core SVD performance for document indexing. *Applied Numerical Mathematics*, 57(11-12):1230–1239.

MySQL Development Team (2010). *MySQL :: The world's most popular open source database*. The Oracle Corporation, http://www.mysql.com (last checked 2010-07-03).

OASIS SOA Reference Model Technical Committee (2006). *Reference Model for Service Oriented Architecture 1.0.*

Oehlschlägel, J., Adler, D., Nenadic, O., and Zucchini, W. (2008). A first glimpse into 'R.ff', a package that virtually removes R's memory limit. http://www.statistik.uni-dortmund.de/useR-2008/slides/Oehlschlaegel+Adler+Nenadic+Zucchini.pdf (last checked 2010-02-12).

OpenID Foundation (2010). *OpenID Foundation Website.* http://openid.net/ (last checked 2010-09-08).

Papazoglou, M. P. (2008). *Web Services: Principles and Technology.* Pearson, Prentice Hall.

PHP Development Team (2010). *PHP: Hypertext Preprocessor.* The PHP Group, http://www.php.net/ (last checked 2010-01-17).

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Reese, G. (2000). *Database Programming with JDBC and Java, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Serhani, M. A. (2008). *A Framework and Methodology for Managing Quality of Web Services.* VDM Verlag, Saarbrücken, Germany.

Sørensen, K. E. (2002). Sessions. In *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuropPLoP 2002)*, Irsee, Germany.

The Apache Software Foundation (2010a). Apache HTTP Server Project. http://httpd.apache.org/ (last checked 2010-02-18).

The Apache Software Foundation (2010b). Apache Tomcat. http://tomcat.apache.org/ (last checked 2010-02-18).

Trausan-Matu, S., Dessus, P., Rebedea, T., Loiseau, M., Dascalu, M., Mihaila, D., Braidman, I., Armitt, G., Smithies, A., Regan, M., Lemaire, B., Stahl, J., Villiot-Leclercq, E., Zampa, V., Chiru, C., Pasov, I., and Dulceanu, A. (2010). Language Technologies for Lifelong Learning: Project Deliverable Report D5.3. http://dspace.ou.nl/handle/1820/2802 (last checked 2010-10-16).

Trausan-Matu, S., Dessus, P., Rebedea, T., Mandin, S., Villiot-Leclercq, E., Dascalu, M., Gartner, A., Chiru, C., Banica, D., Mihaila, D., Lemaire, B., Zampa, V., and Graziani, E. (2009). Language Technologies for Lifelong Learning: Project Deliverable Report D5.2. http://hdl.handle.net/1820/2251 (last checked 2010-09-08).

Urbanek, S. (2009). Rserve - Binary R server. http://www.rforge.net/Rserve/index.html (last checked 2010-02-18).

Vigna, S. (2008). Distributed, large-scale latent semantic analysis by index interpolation. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium.

Völter, M., Kircher, M., and Zdun, U. (2005). *Remoting patterns: foundations of enterprise, internet and realtime distributed object middleware*. Wiley and sons, Chichester, UK, 34. edition.

Web Services Architecture Working Group (2004). *Web Services Architecture*. W3C. http://www.w3.org/TR/wsa-reqs/.

Wild, F. (2009). CRAN Task View: Natural Language Processing. http://cran.at.r-project.org/web/views/NaturalLanguageProcessing.html (last checked 2010-02-15).

Yahoo! Developer Network (2009). The Yahoo! User Interface Library (YUI). http://developer.yahoo.com/yui/ (last checked 2010-02-10).