TECHNISCHE UNIVERSITÄT DRESDEN

Optimized Web Service for Audio Data Analysis

Xi Luo

Professor:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Supervisor:	M.Sc. Kateryna Rybina Dr.-Ing. Daniel Esser Dr.-Ing. Klaus Reindl (Citrix Online GmbH) Dipl.-Inf. (FH) Holger Kaden (Citrix Online GmbH)

Master thesis submitted in fulfilment of the requirements for the degree of *M.Sc. Distributed Systems Engineering* in the Chair of Computer Networks, Faculty of Computer Science.

April 22, 2016

Abstract

Speech quality is one of the essential factors in today's online communication applications, and therefore becomes a significant selling point for relevant commercial products. In order to improve the speech quality, specific signal processing techniques have been applied to speech signals. But to evaluate the effects of the processing, the statistics from previous recordings need to be tracked, analyzed and visualized in an efficient way. Thus audio experts are able to conduct further evaluation and improvement.

There is a web application prototype developed with a backend service and a user interface. It is used to automate the post-processing of the signal processing related statistics, and visualize requested data via a browser. It allows users from different locations to access the service at any time. However, the main shortcoming of this prototype is that the backend service implements a repeated and redundant computation in processing of each user request. This ends up with high latency and poor performance of the application.

An optimized web service based on the current application, named *AirAnalyzerService*, is introduced in this thesis. On the one hand, it serves data in a simple format and applies several modern libraries, frameworks and web technologies such as caching strategies; on the other hand, it integrates a modular design applying useful design patterns to provide better usability, maintainability and extensibility. The new web service is designed and built with Jersey framework, which is selected from evaluation and comparison of popular web service frameworks. According to the tests and evaluation, *AirAnalyzerService* is proven to have good performance with high throughput and low latency.

Declaration

I, Xi Luo, hereby declare that this thesis is my original and independent work. It has never been published and/or submitted for any award of degree to any other institution. All sources of information have been acknowledged by references.

Xi Luo
Dresden, 22.04.2016

Contents

Chapter 1

Introduction

During the last decade, the world has become flatter and people's working style has been dramatically changed due to the development of modern technologies. One example is the fast growing remote meetings with people at a different location that are supported by the real-time online communication techniques and tools. This on the one hand adds the convenience for communications since the web is location and platform independent, and on the other hand it also avoids frequent travelling thus significantly reduces the business cost and resource consumption. The remaining problem is how the online meeting can provide similar or even the same user experience and communication effectiveness, such that it is able to replace the traditional in-person meeting in the future.

1.1 Motivation

For a real-time online communication application that supports web-based meetings and conferences, speech quality is with no doubt one of the most important criteria to facilitate effective communications[1]. Nowadays, various signal processing techniques and algorithms have been applied in modern applications to enhance speech quality. Audio recordings of online conferences are made to gather as much useful information as possible to give feedbacks and realize evaluation of the signal processing and data transmission for further improvements. Furthermore, the statistics collected from the audio recordings need to be automatically evaluated and properly visualized.

For this purpose, a web application prototype for a specific use case has previously been developed. In this use case, the data of online conference recordings are stored in local disk. They are consisting of the statistics of *Conference* and *Channel* which will be explained in the next section. The main information that need to be analyzed and visualized is the score of the conferences and channels, which indicates the speech quality and can be calculated from their statistics. The application is designed

[1]`http://www.computerworld.com/article/2485215/enterprise-applications/web-conferencing-shootout-webex-vs-gotomeeting-vs-mytruecloud.html`

to involve a frontend UI (User Interface) and a backend service, as shown in the architecture in Figure 1.1. The UI is based on browsers for data visualization and user interaction. It retrieves data by making HTTP (Hypertext Transfer Protocol) requests to the backend service. The backend then fetches the statistics from the disk, based on which it calculates the scores. After that the result data is wrapped and sent back to the UI as response for visualization.

The existing application is able to fulfill the fundamental functional requirements of the audio data evaluation and visualization. However, a major concern is the performance of the backend service while it involves a repeated and redundant computation of the statistics for each request from the frontend. For example, when two users ask for the information of the same conference or one user makes the same request twice, the backend service has to doubly execute the score calculation. Apparently this is time and resource consuming, and causes a significant negative impact on the user experience if the processing takes a long time especially in a request-intensive environment. For the specified use case in reality, there will be large amount of data served to multiple users simultaneously. This drives the motivation to build an efficient and flexible web application for audio data analysis and visualization. In particular, the current backend service needs to be replaced by an optimized web service.

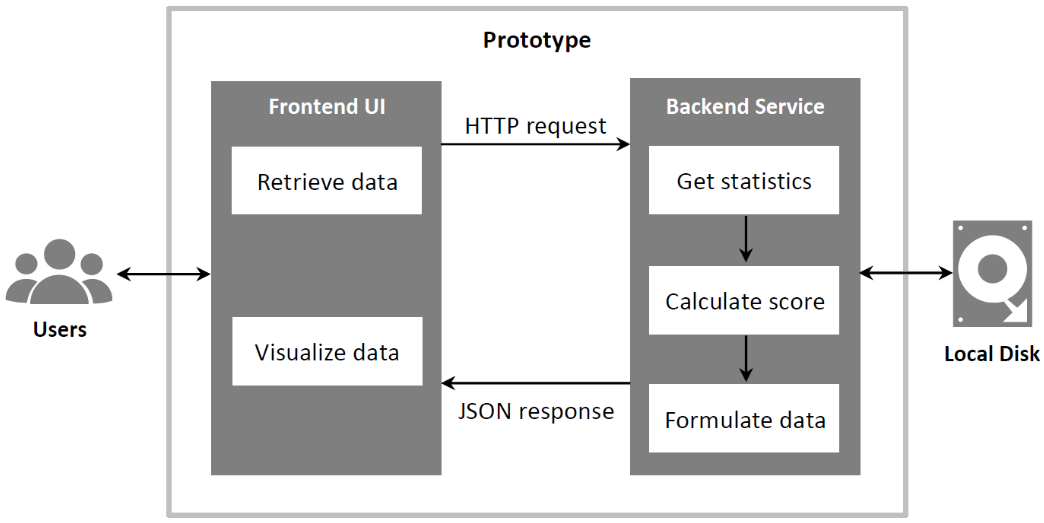


Figure 1.1: Architecture of Web Application Prototype

1.2 Main Definitions

In the specified use case, two main definitions need to be addressed:

- **Conference**

  A conference is an online meeting that is conducted among users and automatically recorded by the real-time communication application. It has its own statistics as well as the statistics of all its participants.

- **Channel**

  Each channel represents a participant in the conference and therefore is part of a conference. Normally there are two or more channels in a conference, depending on the number of users. Every channel has its own statistics.

1.3 Goals and Research Questions

Due to the limitations stated in Section 1.1, the goal of this thesis work is to design and implement an optimized web service with better usability, maintainability, extensibility, and performance which can:

- Gather data of speech quality, i.e. statistics of conferences and channels;
- Aggregate common metrics and calculate scores of conferences and channels;
- Provide results with other general information (uuid, timestamp, etc.) to consumers, e.g. frontend UI.

Based on this goal, four main research questions can be addressed and shall be answered within this thesis:

1. What kind of tool and technique can be used in the optimized web service for better maintainability and performance?
2. How is the optimized web service designed in order to ensure better flexibility and extensibility?
3. How is the optimized web service implemented with integration of the proposed tool and technique?
4. How does the optimized web service perform in the use case of audio data analysis?

1.4 Contributions

An optimized web service, named *AirAnalyzerService*, is proposed in this thesis to replace the current backend service in the web application for audio data analysis. It satisfies all the functional requirements of the existing prototype: gather statistics of conferences and channels, calculate their scores and provide the results together with other general information to consumers, i.e. in this use case the frontend UI.

AirAnalyzerService is designed as a modular web service based on REST (Representational State Transfer) architecture with clear interfaces for data retrieval. It applies a new design of the system architecture and resource model to enhance efficiency and flexibility. A few more tools and techniques such as Jersey framework, caching mechanism etc. are introduced into the system such that better usability, extensibility and maintainability are provided. More importantly, the optimization applies modern tools and techniques that speed up the backend processing and thus improve the overall performance of the service.

An experimental study on the performance evaluation of the optimized web service is conducted in the environment of multiple and concurrent client requests. According to the load testing results, *AirAnalyzerService* is proved to have a good performance, high throughput and low latency on request handling.

1.5 Structure

Following this chapter of introduction, the rest of this thesis is constructed as follows:

Chapter 2 introduces the background knowledge of the thesis with detailed explanations of the concepts and techniques that are used in *AirAnalyzerService*.

Chapter 3 presents a few examples from related work that are either relevant to the field or have applied the tools and techniques mentioned above. It also includes discussions of the limitations of the work and the gap left for this thesis.

Chapter 4 analyzes in details the requirements of the optimized web service and shows the design of system architecture, processing workflows and data structure.

Chapter 5 describes the process of the web service framework selection and gives the implementation details of *AirAnalyzerService*. It also demonstrates the use of caching in the optimization of web service by illustrating experimental results on the performance test and evaluation regarding to throughput and latency.

Chapter 6 summarizes the achievements as well as the limitations of *AirAnalyzerService*. It also outlines possible future work of this thesis.

Chapter 7 is the conclusion of the thesis.

Chapter 2

Background

This chapter is devoted to the basic concepts of the technologies that will be applied in this thesis. The first section introduces the term web service as a general concept, explains its architectures and two different types of it and lists some of the frameworks that are built for ease of development. The second section discusses the caching mechanism which is a widely used technique in modern web or desktop applications.

2.1 Web Service

2.1.1 Web Service in General

Web service, as its name suggests, is a service based on web. More specifically, web service provides a standardized communication between two machines or two applications over an Internet protocol backbone, where one machine or application acts as a service provider and the other acts as a service consumer. Unlike web application that is web-based software for human beings accessed from a web browser, web service is open to programmable clients: an application or even a simple script. It does not need to involve any user interface, hence can be considered as a component of an application. But it actually enhances an application in a way that it allows it to run on different platforms to support heterogeneous environment.

The way to sets up communication for web services is through HTTP, a transfer-independent and firewall friendly protocol. A running web service listens to HTTP requests, parses and handles them, and sends back the response. If the response includes requested data, it can encode it in XML (eXtensible Markup Language) or more light-weight JSON (JavaScript Object Notation) format. XML is also used to form the messages in SOAP (Simple Object Access Protocol[1]) envelope, to exchange information over HTTP. The other two terms in traditional web services are WSDL

[1]`http://www.service-architecture.com/articles/web-services/soap.html`

(Web Service Definition Language[2]) and UDDI (Universal Description, Discovery and Integration[3]), used to describe capabilities and issue identities of a web service, respectively. These components can be put together in Figure 2.1 [1].

As shown in Figure 2.1, each service provider registers with UDDI to publish its WSDL to describe all the services it provides. UDDI manages a list of service providers with available services in a repository. When a service consumer requests a service for the first time, it queries UDDI to get the corresponding service provider and the way to communicate with it (described in WSDL). Then the consumer can directly contact this service provider to send requests and receive response. All the messages among these three components are wrapped in SOAP format [2].

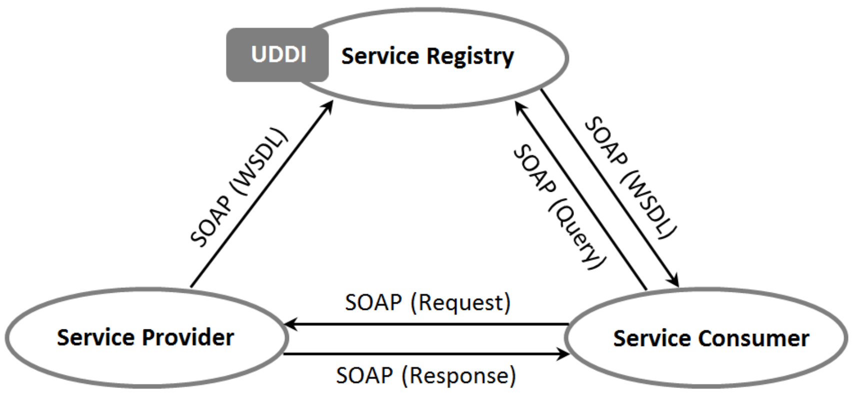


Figure 2.1: Basic Structure of Web Service [1]

2.1.2 Web Service Architectures

A web service involves two types of systems: request/response system (also known as call based distributed system) and message passing system (also known as message based distributed system)[4]. The former one usually provides a synchronous communication with input parameters, output values and the operations in between acting as the correlation. The latter one, however, focuses on the data marshaling and encapsulation specified by certain protocol and format, and the transmission between the source and destination which can be asynchronous.

The request/response system has three main architectures that are commonly used: object-oriented, service-oriented and resource-oriented [3]:

- **Object-oriented architecture (OOA)**

  OOA enables interaction with object instances which are passed by references. Thus, it mainly focuses on marshaling the parameter values. The client side

[2]http://www.service-architecture.com/articles/web-services/web_services_description_language_wsdl.html

[3]http://www.service-architecture.com/articles/web-services/universal_description_discovery_and_integration_uddi.html

[4]https://sites.google.com/site/suryalearnings/home/ooa-vs-soa-vs-roa

and server side are tightly coupled that a stateful communication is required such that the physical objects and state information can be stored at the server side. OOA normally collaborates with middleware protocols like RMI (Remote Method Invocation[5]) and supports middleware specific data formats. One well-known example is EJB (Enterprise Java Beans[6]) in J2EE (Java 2 Platform, Enterprise Edition[7]).

- **Service-oriented architecture (SOA)**

  Unlike OOA, SOA involves a stateless communication with specific application service. It focuses on the creation of request payloads with which client can request for a service via typical HTTP verbs (GET, POST etc.). The server offers services according to service descriptions defined in, e.g. a WSDL file, with a unique endpoint address. It firstly inspects the message data contents, and then based on the service descriptions it decides how to process the request. Client and server sides in this architecture are loosely coupled, and normally communicate with SOAP messages.

- **Resource-oriented architecture (ROA)**

  ROA also supports stateless communication that client and server sides are decoupled. All the data is defined as a resource uniquely defined by an address, or so-called URI (Uniform Resource Identifier). The server keeps a master copy of the resource and returns its up-to-date state as a snapshot to the client when it has made an HTTP request. The communication efficiency in this case is highly improved since no serialization is required and the server replies are cacheable at the client side.

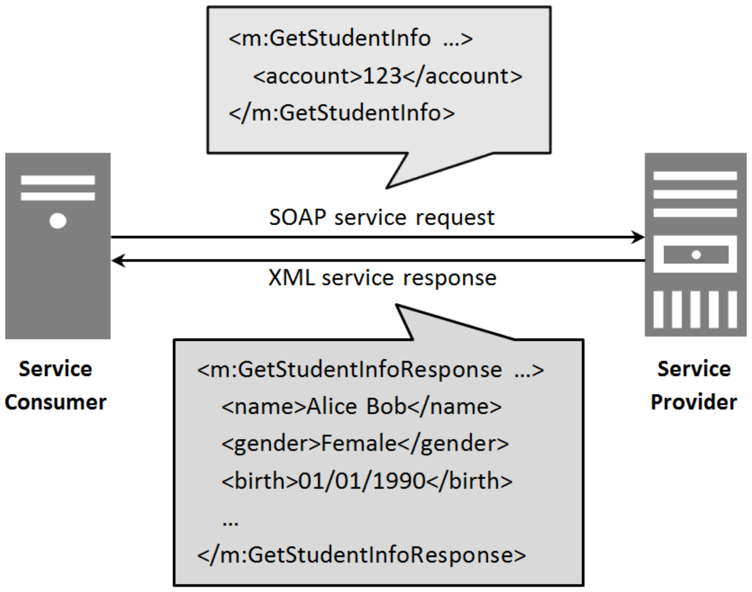
The stateless client-server communication allows simpler implementation, higher efficiency and scalability. Due to these characteristics the service-oriented and resource-oriented architectures are more suitable for systems across organization boundaries. But how to choose architecture highly depends on the requirements and specifications of the system. The choice of the architecture style can have implications on scalability, re-usability and ease of interconnection with other systems.

2.1.3 SOAP and RESTful Web Service

Today it is not necessary to involve UDDI and WSDL directly in a SOAP-based web service infrastructure. Instead of using a repository, SOAP messages for requests and responses can be hard-coded or generated, as illustrated in Figure 2.2 [4].

Although SOAP is the fundamental technique in web service, it has some limitations [5]. First of all, it exposes only one endpoint of API (Application Program Interface)

[5]`http://www.oracle.com/technetwork/articles/javaee/index-jsp-136424.html`
[6]`http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[7]`http://searchsoa.techtarget.com/definition/J2EE`



```
<m:GetStudentInfo ...>
  <account>123</account>
</m:GetStudentInfo>
```

```
<m:GetStudentInfoResponse ...>
  <name>Alice Bob</name>
  <gender>Female</gender>
  <birth>01/01/1990</birth>
  ...
</m:GetStudentInfoResponse>
```

Figure 2.2: SOAP example [4]

and hides the useful information such as operation details and data. Secondly, the SOAP message structure is too heavy and complex to support efficient transmission. Thirdly, the WS-* specifications[8] (WS stands for Web Service) defined in SOAP are inconvenient to develop and deploy. Moreover, clients are bound to WSDL which describes the service interface, and thus, have to follow all of its changes. Last but not least, its performance is limited by the redundant information in SOAP and WSDL, and lack of information in URI and HTTP to use proxy and cache servers.

These limitations result in wider space for a special architectural style called REST. REST has the most important advantages, much easier to use and lighter in communication, and therefore becomes more popular among developers and also in industry. In fact, according to a recent survey[9], RESTful web service nowadays has dominated over 70% of the market.

Although REST is usually considered to be a competitor or even opponent to SOAP, as a matter of fact, it is inappropriate to compare SOAP and REST directly, since REST is neither a protocol nor a standard. But their architectures and the resulting characteristics, performance and usage are comparable.

SOAP defines its own message format based on XML, including message header and message body. The extensibility of its header makes it possible to extend several Internet standards, for instance the well-known WS-* series specifications containing WS-Addressing, WS-Policy, WS-Security etc. This, on the one hand, forms a complete specification and mature standards; but on the other hand, also increases the complexity and difficulty of development. Besides, since SOAP message is self-defined and built out of HTTP messages it does not apply their parameters such as

[8]https://msdn.microsoft.com/en-us/library/ms951274.aspx
[9]https://stormpath.com/blog/rest-vs-soap

encoding, error codes etc. This certainly offers the possibility to customize parameters, but meanwhile also abandons the use of light-weight HTTP messages. Last but not least, SOAP supports both stateful and stateless protocols.

According to [5] and [6], RESTful web service which applies a resource-oriented architecture has the following characteristics:

- **Interface design which focuses on system resources and resource states instead of services**

  Leonard and Sam defined in their book [6] that system resource refers to anything on the server that a client is interested in, e.g. a file, a piece of information, a physical object, etc. Each resource has a unique URI that is accessible from a client. Specific APIs are designed, implemented and exposed to clients to get the representation of a resource or transfer resource states.

- **Straightforward communication mechanism**

  Not like the traditional web services that require finding, binding and then invoking services, RESTful web services can be invoked straightly. It also does not require serializing and deserializing messages as in SOAP-based services.

- **Usage of (but not limited to) HTTP protocol**

  Instead of XML in SOAP, REST uses HTTP protocol to produce and consume web services and takes full advantage of the abstract CRUD (Create, Read, Update, and Delete) operations specified in HTTP. This is less verbose in communication but able to satisfy most of the needs for resource fetching and manipulation. It can also delegate some security functionalities to natural HTTP message, for example, authentication and authorization. Thus, it is easier to understand and implement.

- **Stateless protocol support in client-server communication**

  REST architecture naturally only supports stateless protocol and hence requires self-contained message design. This benefits in better performance and scalability since the server does not need to store the conversation context.

- **Multiple data format support**

  Since resources are decoupled from their representations, they can be formed in various data formats. Apart from XML, RESTful web service also supports other data format such as text, RSS (Rich Site Summary), ATOM etc. Some lightweight formats such as JSON, when compared to XML, can help reduce the message payload supplied by the server and thus reduce the overall network traffic. This is especially helpful in a high volume communication network.

Figure 2.3 [4] shows an example of RESTful web service usage.

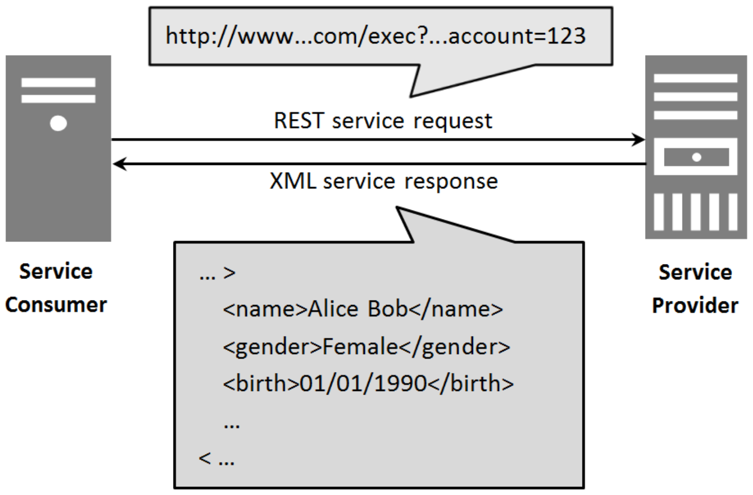


Figure 2.3: REST example [4]

Another factor to be considered is maintainability. Based on the T-test experiment for maintainability evaluation conducted by Ricardo Ramos [7], the statistics show that RESTful web service is more maintainable in terms of both evolutional and adaptive maintenance than SOAP web service on the server-side. However, on the client-side an opposite result is achieved. Furthermore, it is found that lots of business unit systems today are built from SOAP. Hence due to legacy support reason, it is highly probable to adapt the web service to use SOAP when many large company systems are using it for their APIs[10].

So far there is not a final conclusion to say which solution is better, SOAP or REST. Both of them have pros and cons and should be applied in different situations. For example, for simple applications that requires simplicity and efficiency, REST is the best choice. But if the application has high requirements on security and reliability, then SOAP is more suitable.

2.1.4 Web Service Frameworks

According to the document from W3C (World Wide Web Consortium) workshop[11], a framework helps identify specific functions to enable interoperability and avoid overlaps or conflicts in functionality. It allows decomposition of functional features but also defines the relationships in between, so that different components can be coordinated to have parallel development and better integration. Frameworks do not limit the range of facilities of applications, instead they set up standards for expressions, data formats and protocols for specific facilities to ease implementation complexity.

There exist numbers of modern frameworks that make convenience of building reliable and efficient web services. These frameworks are developed and maintained

[10]`https://stormpath.com/blog/rest-vs-soap`
[11]`http://www.w3.org/2001/03/WSWS-popa/paper51`

by independent teams or organizations. Each of them has advantages and disadvantages. Though in general, web service is supposed to be language independent (this is the reason why none of the programming languages has been mentioned till now), web service frameworks as realizations are built upon one specific languages, and some of them may have various versions to support multiple languages. The most widely applied languages are Java, C/C++, .NET, and PHP. A few of them are listed in Table 2.1 and brief introductions of some typical examples will follow.

Apache introduces Axis2 framework as an evolution of Axis that only supports SOAP. Axis2 is based on the Axis SOAP stack with full support of WS-* specifications, and adds REST style interface in the architecture to support RESTful web services. As one of the most popular web projects it provides implementations on C and Java. Besides, Apache CXF is another framework in the family written in Java. It naturally has a compliant REST implementation also with SOAP support.

Oracle has an application server project GlassFish which also can act as a framework for web services. Actually it provides much more than a web service framework as it integrates some extra projects as components, such as Metro for web service stack, OpenMQ for messaging, Grizzly for server scalability, and Jersey, also known as a popular framework, for providing RESTful web service support. Note that Jersey is only for REST web service and does not support SOAP.

Java	Apache Axis[12], Apache Axis2[13], Apache CXF[14], GlassFish[15], Jersey[16], Web Services Interoperability Technology, Web Services Invocation Framework[17], WSO2 WSF[18]
C/C++	Apache Axis, Apache Axis2, gSOAP[19], Restbed[20], Simple-Web-Server[21], WSO2 WSF
.NET	.NET Framework[22], Windows Communication Foundation[23]
PHP	Codelgniter[24], WSO2 WSF, Zend Framework[25]
Other	WSO2 WSF (Perl, Ruby, Python, etc.), .NET Framework (C#), Gugamarket[26] (Node.js)

Table 2.1: List of Web Service Frameworks[27]

[12]https://axis.apache.org/axis
[13]https://axis.apache.org/axis2/java/core
[14]https://cxf.apache.org
[15]https://glassfish.java.net
[16]https://jersey.java.net
[17]http://ws.apache.org/wsif
[18]http://wso2.com/projects/wsf
[19]http://www.cs.fsu.edu/~engelen/soap.html
[20]https://github.com/Corvusoft/restbed
[21]https://github.com/eidheim/Simple-Web-Server
[22]https://www.microsoft.com/net
[23]https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx
[24]https://www.codeigniter.com
[25]http://framework.zend.com
[26]http://www.gugamarket.com
[27]https://en.wikipedia.org/wiki/List_of_web_service_frameworks

gSOAP is an Software Development Kit (SDK) based on C and C++. It not only supports SOAP and REST, but also is one of the frameworks that are friendly with the old-fashioned RPC (Remote Procedure Call) with XML and JSON formats. For those systems built with these technologies gSOAP can intermediate the adaptation to the new architecture. On the contrary, other two C/C++ frameworks Restbed and Simple-Web-Server only have REST implementation.

There are a few frameworks that have not been adopted to REST, only providing SOAP based services, e.g. .NET Framework, Web Services Interoperability Technology, Web Services Invocation Framework, WSO2 WSF. However, WSO2 WSF do provide the most language implementations for runtime platforms based on its core in Java and C.

2.2 Caching

2.2.1 Caching in General

Caching is one of the most used mechanisms in software and applications today. It is extremely helpful in web applications since it can significantly reduce network traffic, transmission delays and server loads, which therefore enhance performance and scale the Internet [8]. Because of this it has even been named as web caching technique.

The idea of caching is to create a component to store some data that are, for instance latest or most popular, so that these data can be served faster in the next time. The data can be the result from previous processing or simply a copy of master data at a different location[28]. In a web-based multi-tier system, caching can be implemented and located in three places: the client-side, the server-side and the intermediate proxy layer. This can be depicted in Figure 2.4, taken from [9]. In this thesis only the server cache is considered. The main advantage of using server cache is to avoid the duplicate of data processing and database I/O (Input/Output). This enables the server loads to be reduced.

There are two main types of caching: file caching and in-memory caching[29]. File caching, namely, caching in files needs to interact with disks. It supports a variety of file formats, XML, DAT, TXT etc. and allows to cache large data. But as a trade-off it takes longer than memory for processing. In-memory caching is the opposite it is much faster but has a limited space for data. It can be realized by using a static map and performing regular operations on the objects such as adding, accessing, updating and deleting.

[28]https://en.wikipedia.org/wiki/Cache_(computing)
[29]http://developer.51cto.com/art/201411/456219.htm

2.2.2 Caching Strategies

The space for caching is always limited that cannot store all the data. Therefore some kind of replacement mechanism is needed when a great number of different requests are made causing the maximal cache capacity is reached, i.e. some old data are to be deleted from cache to create space for newly requested data.

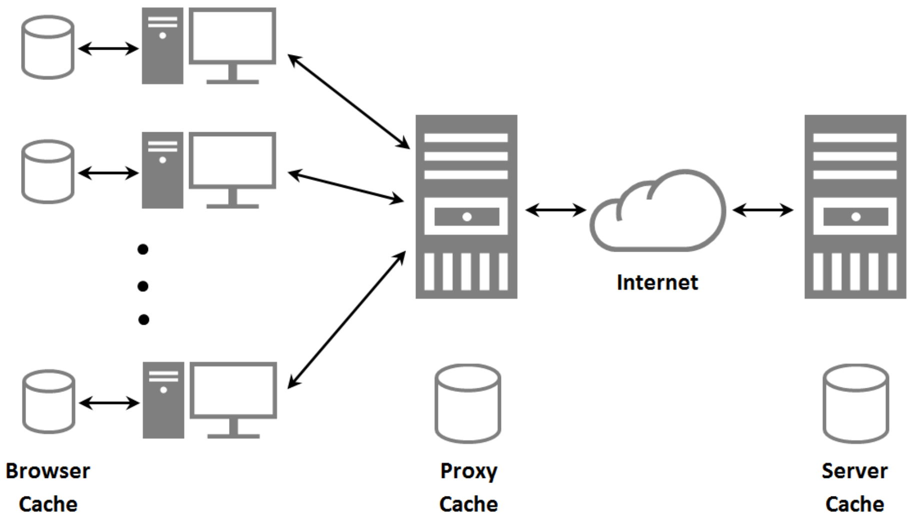


Figure 2.4: Location of Web Cache Placement [9]

Before introducing the concrete strategies, it is important to firstly see the factors that are considered in designing the strategies. As for [10], six important characteristics are listed below:

1. **Recency**: time of (since) the last reference to the object
2. **Frequency**: number of requests to the object
3. **Size**: size of the object
4. **Cost of fetching object**: cost to fetch the object from origin server
5. **Modification time**: time of (since) last modification of the object
6. **Expiration time**: time when the object is automatically set out of date

These factors can be used as a base for designing a variety of replacement strategies:

- **Recency-based strategy**

  This strategy is based on a pre-specified algorithm for recency. For instance, the most well-known strategy of this category LRU (Least Recently Used) replaces the object which is least recently used. This can be realized by assigning a timestamp to each access of an object. The object with the earliest timestamp will be removed when the mechanism is applied.

- **Frequency-based strategy**

  This strategy uses frequency as the primary factor for replacement algorithm. For instance, another most popular strategy LFU (Least Frequently Used) replaces the object which is least frequently used. The realization is to introduce a counter for each object. This counter increments every time the object is accessed. The object with the smallest counter will be removed when the mechanism is applied.

- **Size-based strategy**

  Instead of recency and frequency, object size is the most significant parameter in this strategy. This is straightforward as the object with the largest size will be removed when the mechanism is applied.

- **Function-based strategy**

  This strategy basically combines the consideration of most of the factors mentioned above. It gives different weighting parameters to those factors and removes the object with the smallest value when the mechanism is applied.

There exist more strategy categories with a number of variants. More information and detailed explanations of these strategies can be found in [10].

One of the reasons that there exist so many types of strategies is that essentially no 'silver bullet' for caching is found, i.e. there is no one perfect solution covering all the strategies to support a cache that works well in all cases. What generally happens is that in a specific environment or use case, one factor is more important than another, and thus, one strategy is preferred to the other. Certainly it is possible to combine a few strategies if needed, but things can become complicated when evaluating different factors. In a word, how to select one or more strategies and which strategy to select depends on the requirements and situations of an application, sometimes also relates to user behaviors.

In this thesis, the LRU and LFU strategies combined with the expiration time are used in the optimized web service.

2.3 Summary

Section 2.1 introduces web service with regards to its concept, infrastructure, architecture types and frameworks. In later chapters, the selected REST architecture and Jersey framework in *AirAnalyzerService* will be further explained.

Section 2.2 firstly gives a brief introduction of caching, and then describes several basic caching strategies that can be used in web applications or services based on recency, frequency, size etc. A few of them, LRU and LFU in particular, will be applied and further introduced in later chapters.

Chapter 3

Related Work

In this chapter, some related work of this thesis are presented. The first section shows analysis and evaluation of work on the web service frameworks. The second section gives a few examples of state-of-the-art applications that have used web technologies with specific mechanisms such as caching which are investigated and applied in this thesis.

3.1 Web Service Frameworks Comparison

As listed in Section 2.1.4, there exist a number of web service frameworks with different features developed in various programming languages. To help choose the most proper framework for their applications, a lot of evaluation and comparison work has been conducted, discussed and published.

		Axis	Axis2	CXF	Metro[1]	WSO2 WSF	gSOAP
General Features	REST support		✓	✓	✓	✓	✓
	SOAP 1.1	✓	✓	✓	✓	✓	✓
	SOAP 1.2		✓	✓	✓	✓	✓
	WSDL 1.1	✓	✓	✓	✓	✓	✓
	WSDL 1.2		✓	✓	✓	✓	✓
	Code generation		✓	✓		✓	✓
WS-* Specification	Addressing	✓	✓	✓	✓		✓
	Atomic Transaction	✓	✓		✓	✓	
	Notification	✓	✓	✓			
	ReliableMessaging	✓	✓	✓	✓	✓	✓
	Policy		✓	✓	✓	✓	✓
	Security	✓	✓	✓	✓	✓	✓
Encoding	XML textual	✓	✓	✓	✓	✓	✓
	JSON		✓	✓	✓	✓	✓

Table 3.1: Web Service Framework Feature Comparison

[1]`https://metro.java.net`

This section presents a few work mainly focusing on the feature sets and performance among some popular frameworks written in C/C++ and Java. Table 3.1 summarizes some relevant information and work of feature set comparison from multiple individuals or organizations [11–14]. Note that those not mentioned in the last chapter are excluded from the table as they are out of concern in this thesis, for instance, XFire is deprecated and now has become Apache CXF.

According to the table, all the frameworks (except for Axis as it is an old version that has been rebuilt and replaced by Axis2) support most of the basic features, i.e. fully supporting different versions of SOAP and WSDL, REST and enabling automatic code generation. WS-* specifications come with SOAP and do not need to be analyzed in details as REST is the most concerned feature in this thesis, so is encoding method as REST supports multiple data formats as mentioned in Section 2.1.3.

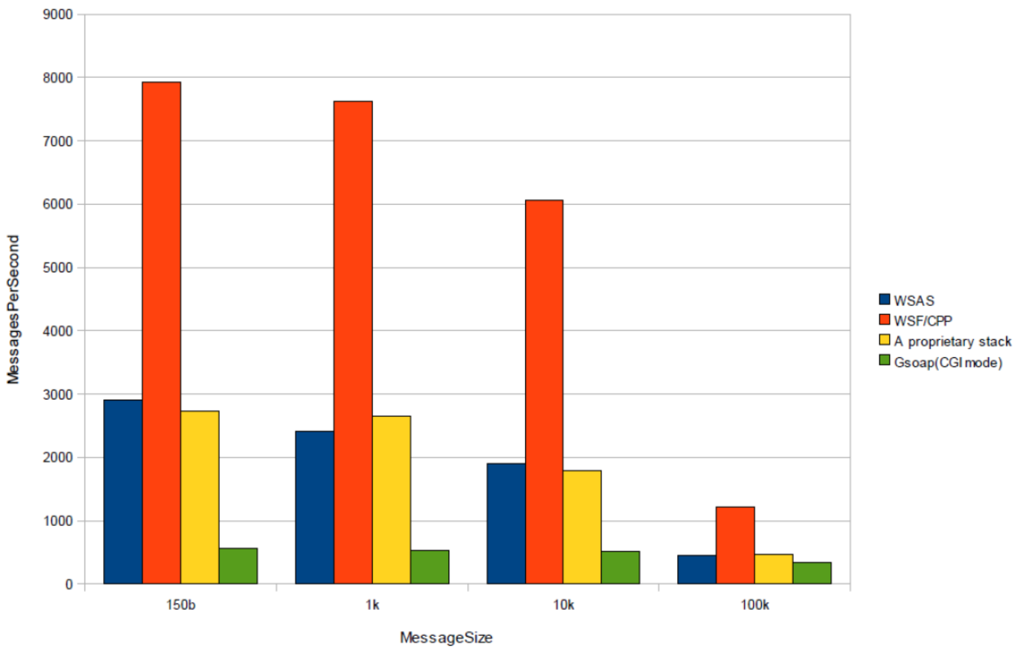


Figure 3.1: Web Service Framework Performance Comparison [15]

Except for features, a performance test on four frameworks is conducted and a figure of result comparison is published in [14], as shown in Figure 3.1 taken from [15]. The related and interesting part is the comparison between the C++ version of WSO2 WSF and gSOAP. The test is based on a simple echo service with only one echo operation for plain text messages running on a Linux system. It tests for a varying size of data load (150 bytes, 1KB, 10KB and 100KB) with the measurement of the throughput of the service. The experiment has indicated that the C++ version of WSO2 WSF has the best performance in all cases, even performs more than double faster than the others. But as the author has already pointed out, this figure only gives a rough result for gSOAP as its CGI (Common Gateway Interface[2]) mode is

[2]http://searchsoa.techtarget.com/definition/common-gateway-interface

used in the test. This has larger overhead in system resources consumption than others even though it has been configured in a most optimized way.

The comparison work shown above, however, mainly focuses on web service frameworks originally designed and built for SOAP, while in this thesis REST is the preferred architecture because of its simplicity and efficiency. Besides, it has not covered all the available state-of-the-art frameworks such as Jersey. This will be mentioned later and explained why it is selected to be the framework used for the optimized web service. Thereby, further evaluation work needs to be included and will be presented in Chapter 5.

3.2 State of the Art

In recent years, lots of state-of-the-art techniques have been successfully exploited in web applications and web services in research institutes and industries across variety of areas. These applications and services have turned out to be very helpful and valuable tools in scientific researches as well as commercial products. The following sections pick up some of the interesting examples that have either been published in papers or registered as patents.

3.2.1 Web-based Tool for Audio Data Analysis

Lately a new web audio evaluation tool comes out as a browser-based application [16]. The authors intend to build a platform-friendly, location-independent and multi-user enabled tool to provide perceptual evaluation tests in audio and music research for those who have little or no programming skills. Therefore a web application becomes their first choice because of its ease of development, deployment and usage.

This tool is based on browsers, so that it can run across different devices and platforms. Users can use this tool anywhere around the world, and anytime even at the same time. It supports a full range of functionalities and is able to automatically set up a test. The tool is well designed and implemented for listening tests that it is integrated with comprehensive processing algorithms. Thus no proprietary software such as Matlab is required. All the processing is completed at the backend and transparent to the user. All of these features are defined to maximally reduce the knowledge and work from the users to allow a wide range of them to use easily. The commonly used XML format is used for setup and result files. Setup file has default settings, but also supports customized configurations from the users. This for sure enhances the usability and user experience.

Earlier another team in the same department has introduced Sonic Visualiser [17], an end-user application to analyze, visualize and annotate music audio files. This application is not only able to load an audio file and display both its waveform and spectrogram, but also allows user interaction such as annotating audios by time instants, time-value plots etc. Users can also install plugins of external functions for additional analysis. The main concern of this tool is that it is a desktop application. As a result users have to download and install the software locally before using it. In addition, to broaden their target users, the providers have to provide different binary distributions for various operating systems. This causes the difficulty of development, testing and maintenance.

The main difference between those two mentioned above and *AirAnalyzerService* is the target data. The audio source files are tested and evaluated in the first tool, read and visualized in the second, while only the meta-file of audio recordings are analyzed and processed in *AirAnalyzerService*. What is more, instead of being an application directly faced to users as those two mentioned above, *AirAnalyzerService* itself acts as a web service and therefore does not involve user interface.

3.2.2 Web Service for Data Analysis in Other Fields

Apart from audio processing and analyzing, there exist a large number of applications applying web techniques in other research or industrial areas. These applications have been proved to be helpful and handful tools for data analysis, visualization, processing and other purposes. Below three examples are provided where the first two applications, in particular, have obtained great success and reputation in the scientific field.

CARMAweb is a comprehensive web application used for microarray data analysis in biology field [18]. It has been designed to be compatible with various types of raw data from different software tools and platforms. Multiple bio-data processing techniques are integrated as an all-in-one application, and these techniques are grouped into several modules which form an analytical pipeline in the usage of the application. This modular design provides the flexibility such that CARMAweb can be easily extended with new functionalities. Another useful feature related to this thesis is that it is able to automatically generate a result file in the local directory of users. The file contains all the detailed information in the analyzing process, including the raw data files, a report of all command lines executed in the processing, the descriptions of applied methods, and the result tables and figures. All the information can be permanently stored, reviewed, and even reused as input into next time analysis. Since it has been modularly designed, the results of each module can also be separately extracted for the next step analysis in another module. This technique enhances the data exploration, transparency and reproducibility.

In the version when the paper is published CARMAweb has been implemented as a multi-tier application using the latest J2EE technology. The authors claim that they would have a later release applying a complete SOAP interface in the facilities to allow external applications to use this service on web. Unfortunately there is no further paper following.

Another data visualization and analysis system, named Giovanni, was developed in 2008 [19]. The main focus of this system is to offer a web-based interface for Earth science data analysis, comparison and visualization, which is designed specifically for internal use in NASA (National Aeronautics and Space Administration) of America. It solves the problem of data downloading, data format and structure parsing, algorithm applying, and has resulted in a reduced cost in scientific investigation, general analysis and comparison service upon high-quality data. The target data includes the meta-data and the data extracted from documentations, papers or even information in informal communications that is not well documented.

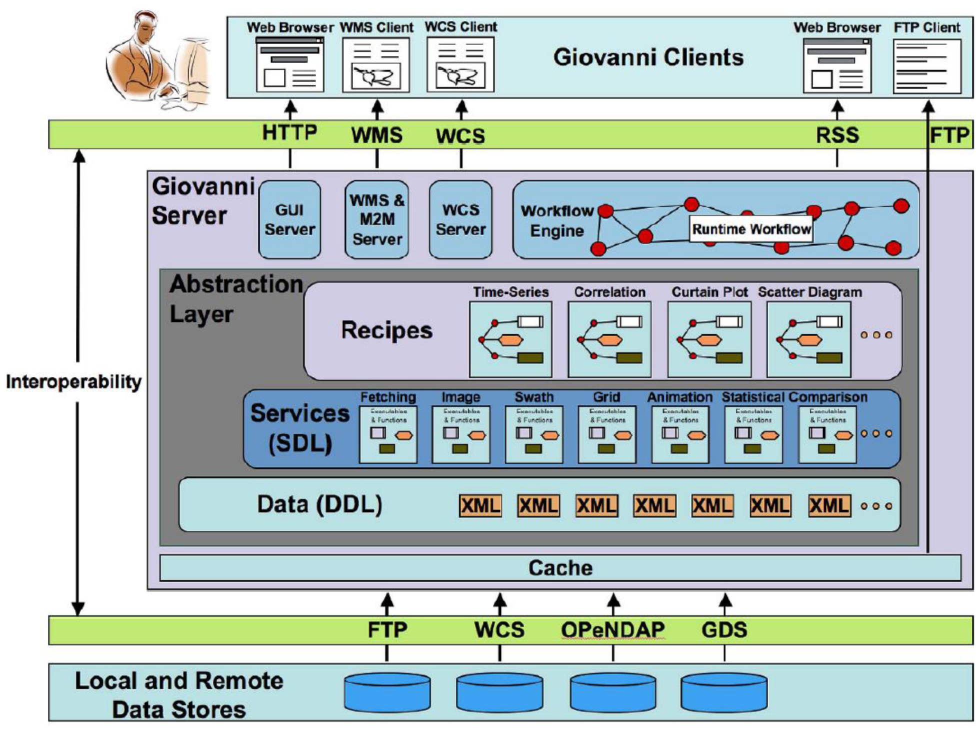


Figure 3.2: Giovanni Functional Architecture [19]

As depicted in Figure 3.2 taken from [19], the core of Giovanni is the server part which supports various types of clients. The relevant part is the abstraction layer and cache component contained in the Giovanni server. The abstraction layer consists of three parts (from bottom layer to the top): data sets that can be interpreted and processed by the services in this system are in XML schema, described by DDL (Data Definition Language[3]); services that can be provided are described by SDL (Services Description Language[4]), which extends the WSDL by involving some command-line programs and Perl/Python functions; recipes that simply gives a list of composite public APIs that can be called by Giovanni clients with references to the individual services described in SDL. The cache component is included to improve performance

[3]`http://databases.about.com/od/sql/a/sqlfundamentals_2.htm`
[4]`http://www.cs.odu.edu/~ibl/450/pdf/view-on-line/dbarch/dbarch4.pdf`

from the server-side since the services that are already performed and stored in cache do need to be executed again.

Giovanni has been considered to meet the up-front requirements such as serving data analysis and visualization with good performance, providing clean and intuitive web APIs, preparing for easy extension and maintenance, supporting for asynchronous processing etc. However, as explained in the last chapter, using WSDL, even with an extension version as SDL in this case requires extra knowledge and work in the developers perspective. Besides, as the authors have mentioned, there still exist some limitations in the system, for instance, intelligent caching solutions are needed when running the service in distributed systems, and customized user configuration need to be supported as the default settings cannot always meet the requirements.

The last example comes from the industry, more specifically, a patent registered by Samsung [20]. The idea was to design a web service method and system for a digital video and/or audio processing device that can be applied to digital televisions.

A classic web service architecture and workflow were used in this system in the following steps:

1. The backend module receives a web service address from an external device.
2. The web service engine checks the description file whether it contains the functionality mapping the address.
3. If the function exists, the engine outputs the address to the engine unit which parses the address and generates a web language corresponding to the function; otherwise returns error.
4. The function setting unit sets the function based on the generated web language.
5. The control unit performs the function as requested.

This invention meets two purposes: firstly, it allows users to remotely control and perform specific functionalities to their TVs at a different location. For example, users can turn on and off the TV, or subscribe and record a favorite program before they arrive home. Secondly, it provides a web service based on SOAP and UDDI for TVs and other external devices. Hence these electronic devices can communicate with each other within a network and specified protocols.

3.2.3 Cache-based Applications

As introduced in the previous chapter, caching is one of the most useful and thus popular techniques for web applications. In 2007, Dominguez, Jr. et al. obtained a patent in America for inventing a Java object cache server [21]. This server is designed to sit between the application servers and the database, serving the servers the data retrieved from database as displayed in Figure 3.3 taken from [21].

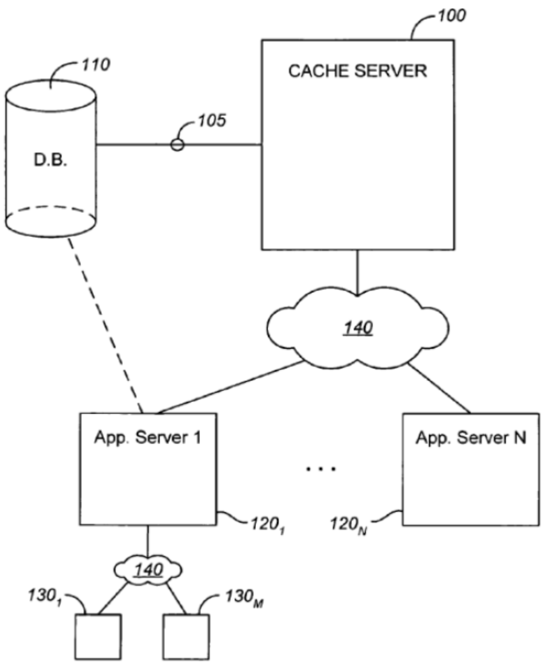


Figure 3.3: Cache Server Application Network [21]

Its workflow is like this: For each HTTP request coming from an application server, the cache server looks into its own directory and sees if the requested data has been stored before. If so, it returns the data directly; if not, it creates a call to database to fetch corresponding data, stores it in its directory and then replies to the application server. This newly stored data will be available immediately for the next time request from the same or any other application server. Note that the cache server has more functionality such as handling the serialization and deserialization for data transmission, which will be ignored in the following since this is irrelevant in this thesis.

With this cache server, the workload of the database can be significantly reduced as a great number of requests are already handled. This contributes to a great improvement in performance because database usually requires file I/O in disk while cache server only needs I/O in memory, and there is no doubt that the latter one is much faster than the former. To maintain the consistency of data, an invalidation policy is applied. The cached objects are valid in a certain period and will be invalidated after a certain amount of time. Within this period, if a data object is being modified by an application server, an update policy is applied. This means instead of invalidating the whole object, the original version of object in the cache server's memory can still be served. But once the update is completed, the new version of the object will replace the old one in cache so that no stale data is returned. This additional update policy is able to balance the consistency and availability of the data.

However, this cache server only applies one replacement mechanism: LRU. As discussed before, this will abandon other factors like frequency and size which may have strong influences on specific infrastructures and applications.

An existing semantic web application is implemented with a specific caching approach to test the effect this mechanism brings [22]. The application publishes information, e.g. general description, location, opening hours etc., of thousands of place-of-interest in Netherland under requests of users. In the proposed approach both client queries and concrete data objects are cached at the first time so that they can be quickly responded or transmitted for the following access. It uses fine-grained caches for the data objects, i.e. for one specific place-of-interest its description, location and opening hours are cached separately. This contributes to an optimized cache invalidation policy that when one of these attributes changes, it does not affect other attributes in the cache.

The performance testing experiments show a great difference between the web application with and without caching implementation: The one with caching approach executes approximately 10$\times$ times faster than before when no caching approach is applied. This result provides a direct proof that caching mechanism is able to improve the overall performance of a system.

3.2.4 REST in Mobile Applications and Services

Last but not least, it is also worth mentioning the web technology, REST in particular, in mobile wireless field since nowadays the mobile market has become one of the largest in the digital world. Although being rapidly and significantly improved, it is still commonly admitted that mobile devices have limited and less computing capability and storage capacity than desktop computers. This makes a perfect position for RESTful web services to fit in mobile applications that are based on web, sitting between the cloud and mobile platforms. The main advantages of RESTful web service are, on the one hand its simplicity to invoke and interpret; on the other hand, more importantly, its succinct and stateless design which is less memory intensive and lightweight. Especially while working with HTTP, making a request and parsing a response are much easier and convenient in a volatile network environments. Even the security concerns can be delegated to the basic HTTP authentication and authorization such that a simpler and cleaner REST architecture can be applied.

Jason H Christensen has done such an investigation in 2009 and elaborated the possibility and benefits to use RESTful web services replacing traditional web services to create next generation mobile applications [23]. He meanwhile points out some issues that need to be considered in this context, for example, test the connection existence and transmission rate, balance the data payload and request frequency etc. Fahad Aijaz et al. have recommended in [24] the REST as a suitable framework for mobile web communications based on their finding that mobile web server using REST performs better than traditional SOAP in all tested metrics such as server utilization, waiting-time, throughput and queue-length in their architectural performance experiments. Then this has been further proved in the study of Kishor Wagh [25] that REST is more suitable for wireless environment as well as mobile applications regarding to resource consumption including battery power.

There are application examples as well. The REST architecture has been applied in a mobile application named AppJoy [26], which is used for personalized mobile app recommendation based on user behavior analysis. It is implemented as a core module at the server side. It has received a good result in the evaluation of energy consumption, latency and popularity. Another instance is BeWell [27], a smartphone application to help people maintain healthy lifestyle and monitor daily activities. RESTful interfaces are offered also at the server side in the cloud infrastructure to store user uploaded files and inputs, and handle queries for data. Experiment results indicate BeWell is competitive with other applications in terms of resource efficiency and performs well in battery life tests. Though none of these researchers has conducted such a study that, and there is also no evidence showing, these results are directly related to REST, it is believed that the success of these mobile applications has its contribution.

One of the latest examples is a mobile application framework using RESTful web services with JSON parser [28]. The system aims to improve the interactions between users (in its scenario this includes students, staffs, librarians etc. in the university) and the university library due the nature of good usability and interoperability of RESTful web services. It consists of two application modules at the moment: One is for library management, e.g. e-book access and updates; the other one is for user management, e.g. account registration and permission control. Both applications support the basic CRUD operations specified in the HTTP protocol. They are easily integrated within the university wireless network. Currently the framework is built on android systems and is successfully tested. It can be extended to other mobile operating systems like iOS and Windows, and support many other mobile applications.

3.3 Summary

Section 3.1 collects some comparison works from previous researches on a few popular web service frameworks focusing on feature sets and performance evaluation. Nonetheless, they are not optimized for REST architecture and have not covered all the frameworks that have been used in modern applications, e.g. Spring.

Section 3.2 gives a considerable number of example applications for various usages in the data processing area in recent years, including audio signals, biological data, spatial images etc. In these applications the modern web techniques such as REST and/or caching have been applied. These techniques will also be used later. However, there still exists a blank space for a reliable, efficient and easy to use web service in the field of audio recordings in real-time systems, highlighting on audio data analysis, processing and visualization. This is the main focus of this thesis.

Chapter 4

Requirements Analysis and System Design

This chapter describes the functional and non-functional requirements with detailed specifications of *AirAnalyzerService* in the first section. Based on the analysis of these requirements and specifications, the use cases, design of the system architecture, processing workflows, data structure and resource model in the web service are presented in the second section.

4.1 Requirements Analysis

According to the introduction in Chapter 1, an optimized web service called *AirAnalyzerService* acting as the backend component in the audio data analysis tool, namely *AirAnalyzer*, needs to be designed and developed. It is placed between the frontend UI and the disk and interacts with these two components as shown in Figure 4.1. It is used to receive HTTP requests from the UI, implement file I/O with the disk, and reply to the UI with requested data in JSON format.

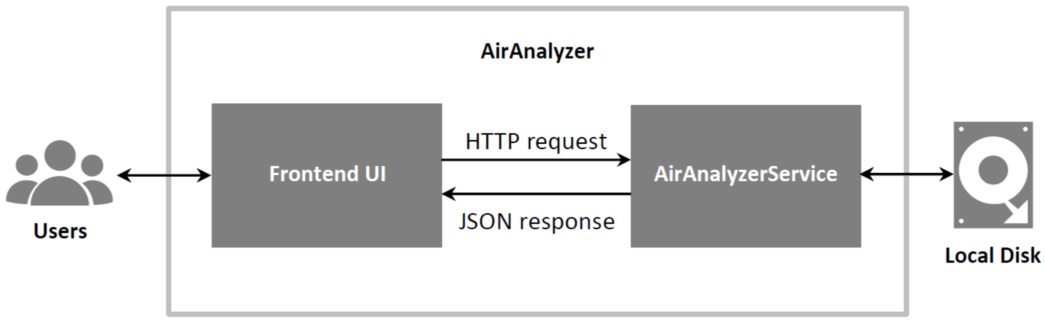


Figure 4.1: Architecture of AirAnalyzer

The fundamental requirements of *AirAnalyzerService* can be extracted and analyzed in further details. Following the convention in software engineering, the requirements can be specified in two perspectives: the functional requirements and the non-functional requirements. Ulf defines that functional requirements essentially specify what the system should do, and non-functional requirements specify how the system should behave[1]. The two types of requirements for this web service are described in the following sections.

4.1.1 Functional Requirements

AirAnalyzerService shall be able to fulfill three basic functionalities: gather data of speech quality, aggregate into metrics, provide the requested data to consumers. Thus the functional requirements are listed and explained below:

- **Read information of conferences and channels from data files**

  In the specified use case, the data files of conferences and channels are stored in a hierarchy in the same server machine where *AirAnalyzerService* is running. An example with one conference containing one channel can be illustrated below, while in reality there will be numbers of conferences and each of them may contain multiple channels.

  ```
  -  root (folder)
      -  conference (folder)
          -  conference1_uuid_timestamp (folder)
              -  source recordings (.wav files)
              -  mixSum (.txt file)
              -  channel0_uuid_timestamp_strProcessor (.txt file)
              -  channel0_uuid_timestamp_strEnhancer (.txt file)
              -  channel0_uuid_timestamp_mixOut (.txt file)
  ```

  The root directory contains conference folders, i.e. each conference has its own folder. The folder name indicates the uuid and timestamp of the conference. Each folder contains a list of files for this conference: source recording files, statistics files of the conference (mixSum) and its channels. Each channel has three files to store different statistics (strProcessor, strEnhancer and mixOut). Note that the meaning of these statistics are not concerned in this thesis as they only need to be properly read and wrapped for transmission. Each statistics file of a channel indicates its uuid and timestamp.

  Within a statistics file, data are stored like a database table, but with comma as dividers as shown in the following layout:

  ```
  [header_1], [header_2], [header_3], [header_4], [header_5] ...
  1, 2, 3, 4, 5 ...
  1, 2, 3, 4, 5 ...
  1, 2, 3, 4, 5 ...
  ...
  ```

[1]http://reqtest.com/requirements-blog/functional-vs-non-functional-requirements

The first line is the header line indicating the name of each column. From the second line all the statistic values of columns are followed. Different files have different headers, number of columns and rows. Normally the number of columns does not exceed 20 but the number of rows can be thousands.

- **Process statistics and calculate scores for conferences and channels**

  Score calculation is based on the statistics data read from statistics files of conferences and channels. Each channel has a score, and each conference has an overall score based on the scores of its channels. The concrete algorithm of the score calculation is already provided by external organization and will not be further explained in this thesis. However the conference score and channel score have a structure which does matter because each metric in the structure needs to be explicitly served to consumers. A conference score consists of two metrics: score of 'avgPLI' and 'avgLI', while a channel score contains three metrics: the two in conference score plus an 'avgPL' score. Again what these scores mean are not relevant.

- **Serve data via exposed APIs**

  The data provided to the consumers should be both coarse-grained and fine-grained available. This means consumers are able to query either all the conference data or only the data for one specific conference or channel in a single call. The data include the calculated score of conferences and channels as well as their general information: uuid and timestamp extracted from the folder or file names, and statistics read from the text files. This involves all the statistic data stored in column layout as shown above. For conferences, the number of their channels should also be displayed.

  All these data shall be properly constructed and have a clean readable layout in the return object.

4.1.2 Non-functional Requirements

Apart from the functionalities defined above, *AirAnalyzerService* shall be configurable and platform independent so that it is able to run under different OS environments. Besides, a few more characteristics are required in the system: usability, maintainability, extensibility, testability and performance. More details are followed:

1. **Configuration**

   Due to the fact that *AirAnalyzerService* will be running on various operation systems, it should enable the following configurations:

   - File directory: where the data files are stored.
   - Server host and port: where the data is accessible.

- Maximal line number to read a file: the file may contain unrelated lines of data.

If extra techniques and mechanisms are used in the web service they should also be configurable.

2. **Usability**

   AirAnalyzerService should allow an easy access to the requested data for consumer, i.e. the frontend UI. On the one hand, the consumer is able to get the desired data by efficiently making simple HTTP requests; on the other hand, the response data is in a common format that is easy to parse and use. JSON format is preferred for data transmission in this case because of its lightweight, compactness and ease of use in the JavaScript program which is written in the frontend.

3. **Maintainability**

   Since high maintainability of a system requires less working time the quality of *AirAnalyzerService* should be easily corrected and improved. This can be realized by having a good programming habit, e.g. frequent use of comments, considerable variable name definition, proper use of spaces etc. But on the other hand, it is also helpful to use state-of-the-art libraries and frameworks in the system which are stable, easily integrated and actively maintained by programming experts.

4. **Extensibility**

   Extensibility is an important characteristic for *AirAnalyzerService*. This not only means feature enhancement, but also increases flexibility and agility of the system when, for example, the location of data files changes. Extensibility also requires reducing the dependencies among system components so that changes in one component have no or less influence on others. This loose coupling structure can be realized by designing a modular architecture. As a result components can be reused, exchanged and extended.

5. **Testability**

   In order to ensure the quality and stability of the system, and realize the evaluation steps, *AirAnalyzerService* should be able to conduct automatic testing and therefore unit testing needs to be implemented.

6. **Performance**

   Runtime performance is the key factor to *AirAnalyzerService* as this is the main shortcoming of the current prototype. It is required to provide fast and efficient data fetching service in a request-intensive environment with high throughput and low latency, even in an intensive environment where multiple consumers make simultaneous and concurrent requests.

In addition, as already widely used in modern server applications, logging should be involved in *AirAnalyzerService* as well. This is a useful tool to achieve better usability, maintainability and testability because it provides all the details at system runtime and records exactly what has happened in the background. Certainly the resource and overhead need to be considered when introducing logging into the system.

4.2 System Design

The previous section describes and analyzes in details the requirements with respect to functional and non-functional aspects. Based on that the system design of *AirAnalyzerService* can be made and is explained in this section.

4.2.1 General Concepts

Before digging into the design details, a number of concepts and ideas that are introduced in the optimized web service can be firstly presented.

1. **REST interfaces**

   AirAnalyzerService is designed to be a RESTful web service since the REST architecture for web services, as introduced in Section 2.1.3, is simple, efficient and easy to use. It completely satisfies the functional needs of *AirAnalyzerService* and provides very good usability and maintainability.

   - All the information of conferences and channels can be treated as resources in the system and it is convenient to identify and expose their representations by REST APIs. The interfaces are easily extended if more resources are available.
   - The frontend UI is programmed in JavaScript and therefore can simply make an HTTP request to the backend service to get the data. The data will be wrapped in JSON format as desired in JavaScript.
   - A basic communication is needed between the frontend UI and backend web service. *AirAnalyzerService* is not required to keep any conversation state. But it needs to be scalable to handle multiple and concurrent client requests. As a result the stateless communication mechanism in REST is the best option.

2. **Jersey framework**

   A proper framework is decided to be used in this optimized web service. On the one hand, it helps to define standard criteria and protocols, and makes it easier in development; on the other hand, more importantly, it provides better maintainability of the application since it is actively and well maintained by programming experts.

AirAnalyzerService applies a mature production-ready framework in its REST interfaces implementation called Jersey. It is the reference implementation for RESTful web service in Java, extremely popular in industry. The more detailed reasons why it is chosen than other frameworks are explained in a separate section in Chapter 5.

3. **Caching Strategies**

   Caching, as described in Section 2.2, is one of the most widely used mechanisms to improve performance in modern applications. It will certainly come with some overhead for the system to deal with cache objects, but overall this should be compensated and covered by the benefit it brings. Thus, *AirAnalyzerService* is designed to use cache.

   There exist numbers of caching replacement strategies. In the specified use case two mainstream strategies: LRU and LFU are applied, combined with the expiration time. Since caching is newly introduced in *AirAnalyzerService*, it is possible to configure, e.g. whether to use it, which replacement strategy to use, size of the cache, etc. More details will follow.

4. **Conference and channel list files**

   Section 1.1 states that the main limitation of the current prototype is the repeated and redundant computation in the background service. To solve this problem, the idea in *AirAnalyzerService* is rather simple: only involve one-time computation and store the processing results for future access. Now that in the use case all the data are stored in files in a directory, it can also use files to store the processing results. Section 5.2.2 will discuss more in details how the files are managed.

   Besides, these two files have to be synchronized with the conference and channel data files in the directory. For instance, when new conference and channel data files are put into the directory, corresponding records need to be added in both files; on the contrary, if conference and channel data files are removed from the directory, their records in both files should be deleted as well.

5. **Score placeholder**

   A placeholder is used for scores of conferences and channels to improve service efficiency and enhance user experience. The reason behind is that the score need some processing time. This time can be very short, but there still exists a probability (though very small) that when the client makes a request for a conference or a channel, its score is not yet calculated. To cover this gap from the user perspective, *AirAnalyzerService* is designed to use an empty placeholder in the data object if the score is not ready, and refill the field after the calculation is completed. This will be further discussed in Section 5.2.6.

6. **Partial representation**

   This technique is introduced in [29] and is applied in this web service due to the fact that the statistics files of conferences and channels have a large size (each with a few hundred kilobytes) which significantly increases the cost of data transmission and server load. The statistics is designed to be an optional element in the resource and therefore each resource data can have two representations: one with statistics and one without statistics. They will be exposed by different REST APIs. It depends on the client whether to request the statistics or not.

   The reason for this design is that in the real case, the statistics represent the original raw information of conferences and channels. They have already been aggregated and calculated to scores which is able to give a more straightforward overview whether the recordings are good or bad. In general this should be enough for audio data analysis and evaluation. Thereby it is less likely for users to request statistics, and it is better to make them optional. More details can be found in Section 4.2.5.

More design concepts are introduced and explained in the following five sections.

4.2.2 Use cases

The use cases specify the client actions to an application or a service. In other words, they are the available services that can be provided to clients. In *AirAnalyzerService*, more specifically, it is the data retrieval service. The service intends to return general information as well as the processed results of conferences and channels. But how these data are constructed and exposed to clients remains unclear. This will be discussed in this section.

A lazy loading mechanism is applied in the frontend program. The basic information of conferences and channels, i.e. uuid, timestamp and score, is displayed automatically. But the statistics are shown only when users make a further request, e.g. by clicking a button. This results in two separate uses cases in *AirAnalyzerService*.

For each conference and channel, two definitions can be created beforehand:

- *Summary*

  Summary includes the basic information such as uuid, timestamp and score. For conferences, in particular, this also includes the number of channels and a list of channels, while these channels only contain the basic information.

- *Detail*

  Detail includes all the information, i.e. basic information and statistics.

Clients can get the *Summary* and *Detail* data by making two different requests. In addition for conferences, their channels are also available as an extra resource. All these use cases can be illustrated in Figure 4.2.

Four use cases are directly related to conferences: conference list, conference *Summary*, conference *Detail* and conference channels. Two use cases are channel related: channel *Summary* and channel *Detail*. These use cases are combined in the resource model design in the next section.

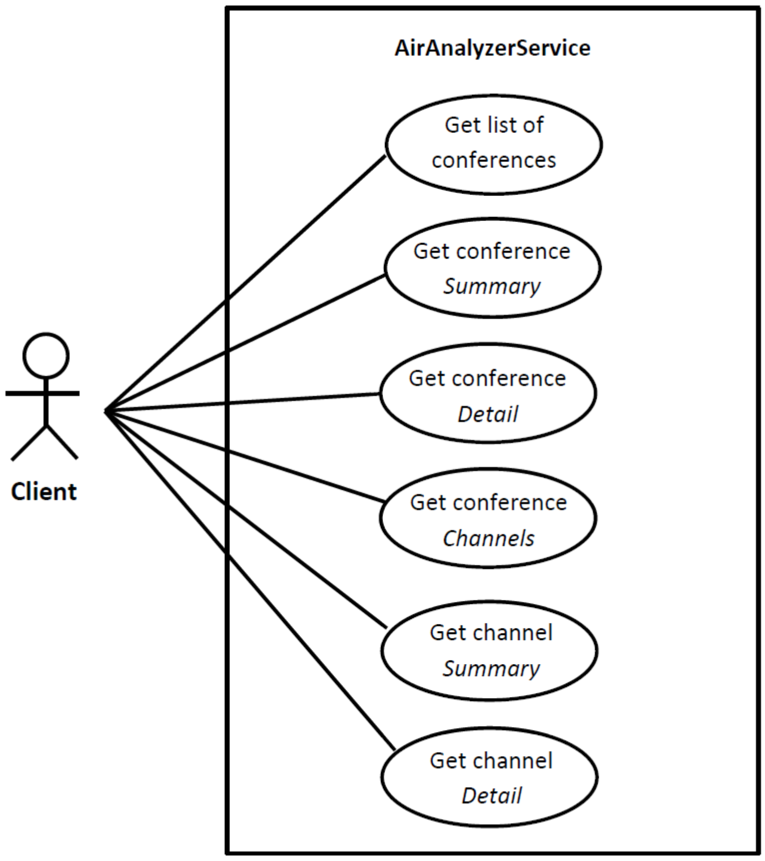


Figure 4.2: Use cases of AirAnalyzerService

4.2.3 System Architecture

As mentioned in Section 4.1.2, an extensible design is needed for the web service. This requires both the functionalities to be easily enhanced and the components to be flexibly exchanged in *AirAnalyzerService*. To achieve this goal, a modular design of the system architecture is conducted. Designing the service in modules also brings the benefit of making the components loose coupling to reduce the dependencies among them. The modules in the service are designed mainly based on the functional requirements stated in Section 4.1.1. Each module is responsible for providing corresponding functionality. A better maintainability and testability can therefore be achieved.

Figure 4.3 denotes the architecture of *AirAnalyzerService*. Note that in the specified use case, the frontend UI makes requests to the service via Intranet. However the web service is also accessible from other outside consumers by making HTTP requests via the same REST APIs. As shown in the figure, there are five main modules in the architecture design.

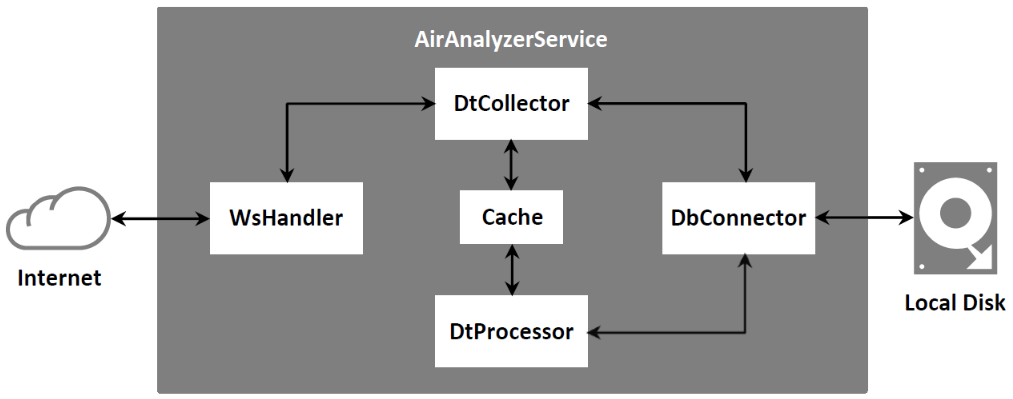


Figure 4.3: Architecture of AirAnalyzerService

1. *WsHandler*

   WsHandler is short for Web Service Handler. It acts as an entry point of the web service for the consumers. It defines a functional mapping between the URIs and the backend methods. When an HTTP request comes in, it looks into the mapping and makes a corresponding call to the method defined in the backend, i.e. the *DtCollector* (see next bullet point), to get the requested data. Then it wraps up the data object in JSON format and serializes it to send back to the consumer.

   To sum up, the responsibilities of *WsHandler* include:

   - Receive and parse HTTP requests from clients
   - Handle the HTTP requests by calling corresponding methods in *DtCollector*
   - Wrap the data object returned by *DtCollector* in desired format, i.e. JSON in this case
   - Respond to clients with requested data

2. *DtCollector*

   DtCollector is short for Data Collector. As its name suggests, it collects data for *WsHandler*. There are two ways for doing this: either fetching data from cache directly if it is cached or from disk through *DbConnector* (see next bullet point) if not. If the data is fetched from the disk, it needs to save it in the cache so that the next time the same request comes in it is able to fetch it directly from cache. The detailed process will be introduced using flowcharts in the next section.

To sum up, the responsibilities of *DtCollector* include:

- Receive data requests from *WsHandler*
- Collect requested data from cache if it is cached; Or collect requested data from disk by calling corresponding methods in *DbConnector* if it is not cached
- Cache requested data if it is fetched from disk
- Return collected data to *WsHandler*

3. *DbConnector*

   DbConnector is short for Database Connector. Although there is no database involved in this work (it uses files instead), it borrows its idea and applies the similar concept. *DbConnector* is the bridge to the data files stored in disk and thereby is the only way for *AirAnalyzerService* to realize file I/O.

   There are two kinds of operations in the file I/O of conferences and channels in this case: one is to read the statistics, and interpret the folder/file names to get uuids and timestamps (read-only); the other is to read/write the files that store the general information and scores. *DtCollector* requires only the first operation, while *DtProcessor* (see next bullet point) requires both operations.

   To sum up, the responsibilities of *DbConnector* include:

   - Receive file read/write requests from *DtCollector* and *DtProcessor*
   - Get general information (uuid, timestamp, etc.) of conferences and channels
   - Read statistics of conferences and channels in disk
   - Create, read, and update the conference list and channel list files

4. *DtProcessor*

   DtProcessor is short for Data Processor. It is meant to calculate the scores for conferences and channels based on their statistics read from data files by *DbConnector*. But first it needs to check the file directory if there are conferences with channels that have been newly added or deleted. For new conferences it fetches statistics, calculates scores and adds new records in both conference list and channel list files; for deprecated conferences it removes the corresponding records in the list files.

   The file update checking and processing is regularly active. The checking interval is also configurable.

   To sum up, the responsibilities of *DtProcessor* include:

- Check data files updates through *DbConnector*
- Calculate scores for new conferences and channels
- Store the calculated scores with general information in conference list and channel list files through *DbConnector*
- Update conference list and channel list files if conferences with channels are added or removed

5. *Cache*

   Cache is a separate and special module from those mentioned above because it can be disabled in user configuration. When it is enabled, it manages all the cache objects with CRUD operations, and executes cache object replacement with predefined strategies, e.g. LRU and LFU. It is a globally accessible object interacting with *DtCollector* and *DtProcessor*.

   To sum up, the responsibilities of *Cache* include:

   - Manage cache objects triggered by *DtCollector* and *DtProcessor*
   - Realize cache replacement by user configuration

4.2.4 Processing Workflows

In the architecture shown above, two modules are separately designed for specific purposes: *DtCollector* is to collect data for clients and *DtProcessor* is to process statistics in the background. They do not have direct interactions within each other, nor to be accessed by the outside world. Both of them need to work with cache or *DbConnector* for file I/O.

The design is resulted from the idea of decoupling the data fetching service and the backend business logics. The two components are independent to each other so that *AirAnalyzerService* can have two independent and parallel workflows. The first workflow involves all the components in the architecture except for *DtProcessor*, responsible for handling client requests; the second workflow involves only *DtProcessor*, *DbConnector* and cache, to calculate the scores and manage the list files for conferences and channels. Figure 4.4 shows the two workflows in flowcharts for *AirAnalyzerService*.

The left hand side of the figure indicates the cycle of one request handling. The actions in this workflow are passive and need to be triggered by requests coming from clients. This means each incoming request creates such a working cycle in the service. When such a request comes in, *WsHandler* looks up its function mappings and makes the corresponding method call to. *DtCollector* then firstly checks if the requested data is already stored in cache: if so, fetches it directly from in-memory cache and returns to the client; if not, makes a call to *DbConnector* to fetch the

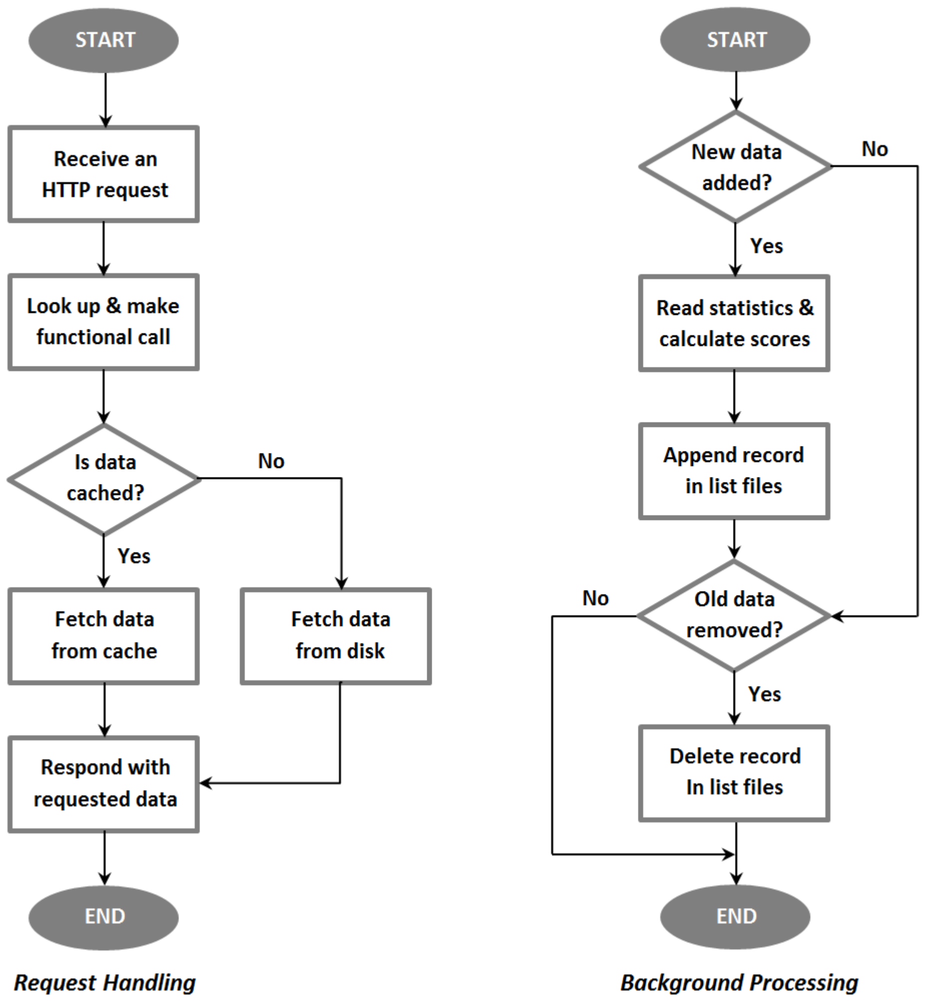


Figure 4.4: Workflows of AirAnalyzerService

data from data files in disk and returns. Meanwhile it will also create a new object in cache for the requested data for future access.

The right hand side of the figure illustrates only one cycle of the background processing. The actions in this workflow are active and can be triggered immediately when the server starts running (no client behaviors involved). The processing workflow is endlessly repeated in a pre-configured time interval unless the service is terminated. *DtProcessor* regularly checks the file updates in the directory and only executes the processing when data files are updated. The processing is differentiated for two situations (which may happen at the same time in one checking) as follows:

- **New conference and channel data added in directory**

  For newly added conferences with channels, *DtProcessor* reads statistics from data files, calculates their scores, and appends new records in the conference list and channel list files. For the first time of the service running, this involves the file creation and initialization, i.e. put in headers as well.

- **Old conference and channel data removed from directory**

For deprecated conferences or channels, *DtProcessor* deletes the records of them in the corresponding list files.

More details and examples related can be found in the chapter of implementation in Section 5.2.5.

4.2.5 Data Structure and Resource Model

Data structure is closely related to the resource model that will be exposed by REST interfaces, as it defines the form of data objects that are transmitted from server to client. Both are discussed in this section.

Previous chapters and sections have given a few hints to that how the conference and channel data files are constructed with their attributes and the relationships between conferences and channels. Based on that information the basic data structure can be illustrated in Figure 4.5.

Those marked in bold in Figure 4.5 are self-defined data classes with their own attributes and methods. The attributes are derived from the functional requirements in Section 4.1.1, representing the information that needs to be exposed to clients. The methods are typically simple getters and setters of the attributes. A few concepts to highlight:

- **Object attributes**

  Apart from the number of channels and a list of channels that only exist in a conference, each conference and channel has the attributes of uuid, timestamp, statistics and score. Statistics and score are with different forms and definitions in conferences and channels. Conference statistics are formed by only one field, mixSum, as they are extracted from the single mixSum file. Channel statistics consist of three fields, strProcessor, strEnhancer and mixOut as they are extracted from these files respectively. Channel score has three subsidiaries: avgPLI, avgLI and avgPL, while conference score only has the first two. Again the meanings behind these names are not concerned here.

- **Data types**

  All the attributes have been defined with data types, and all the methods are defined with their return types. As mentioned above the bold ones are customized, while the others are primitive types (e.g. int, double), abstract data types (e.g. Map) or utility types (e.g. LocalDateTime) that have already been defined in the programming language.

  The LocalDateTime is used for timestamp because, firstly it has inline functions to compare two timestamp and find out the earlier or later one; secondly,

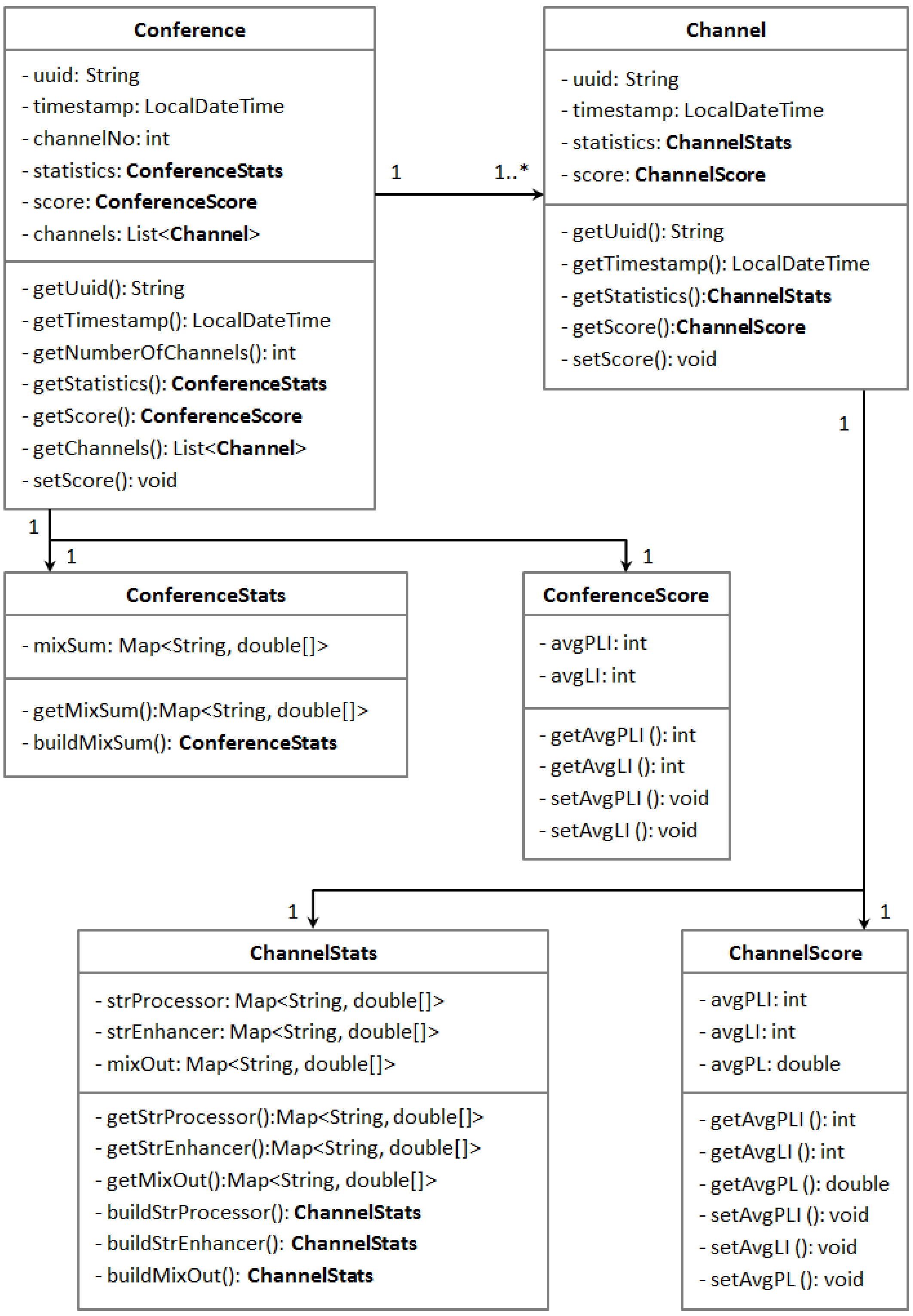

Conference
- uuid: String
- timestamp: LocalDateTime
- channelNo: int
- statistics: ConferenceStats
- score: ConferenceScore
- channels: List<Channel>
- getUuid(): String
- getTimestamp(): LocalDateTime
- getNumberOfChannels(): int
- getStatistics(): ConferenceStats
- getScore(): ConferenceScore
- getChannels(): List<Channel>
- setScore(): void
1
1..*
Channel
- uuid: String
- timestamp: LocalDateTime
- statistics: ChannelStats
- score: ChannelScore
- getUuid(): String
- getTimestamp(): LocalDateTime
- getStatistics():ChannelStats
- getScore():ChannelScore
- setScore(): void
1
1
1
1
ConferenceStats
- mixSum: Map<String, double[]>
- getMixSum():Map<String, double[]>
- buildMixSum(): ConferenceStats
ConferenceScore
- avgPLI: int
- avgLI: int
- getAvgPLI (): int
- getAvgLI (): int
- setAvgPLI (): void
- setAvgLI (): void
1
1
ChannelStats
- strProcessor: Map<String, double[]>
- strEnhancer: Map<String, double[]>
- mixOut: Map<String, double[]>
- getStrProcessor():Map<String, double[]>
- getStrEnhancer():Map<String, double[]>
- getMixOut():Map<String, double[]>
- buildStrProcessor(): ChannelStats
- buildStrEnhancer(): ChannelStats
- buildMixOut(): ChannelStats
ChannelScore
- avgPLI: int
- avgLI: int
- avgPL: double
- getAvgPLI (): int
- getAvgLI (): int
- getAvgPL (): double
- setAvgPLI (): void
- setAvgLI (): void
- setAvgPL (): void

Figure 4.5: Data Structure of AirAnalyzerService

it is easy to interpret and for both client and server to extract single elements (e.g. year, month, day, hour, minute, second).

The Map type is used for the statistics fields resulting from the data layout described in the functional requirements section. The key in the Map is set as the header name in a column of the file. The value is an array of double that stores the statistic numbers of the corresponding column.

- **Mappings**

  The figure also illustrates the hierarchical relationships and number of mappings between these objects: a conference object can contain one or multiple channel objects; one conference object has one statistics object and one score object, so as a channel object.

Combining the use cases and data structure, a more abstract hierarchical data model can be made. The model defines what information is provided for clients and how the data object is constructed to convey that multi-level information, as shown in Figure 4.6 and Figure 4.7. In this design conferences and channels are decoupled although channels belong to conferences. The reason for this is that in future channels might be independent to conferences as one channel may exist in different conferences.

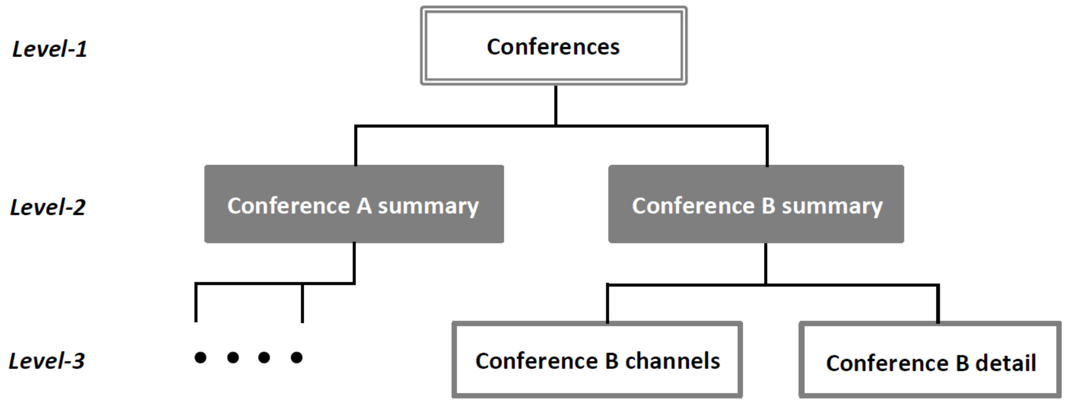


Figure 4.6: Data Model of Conference

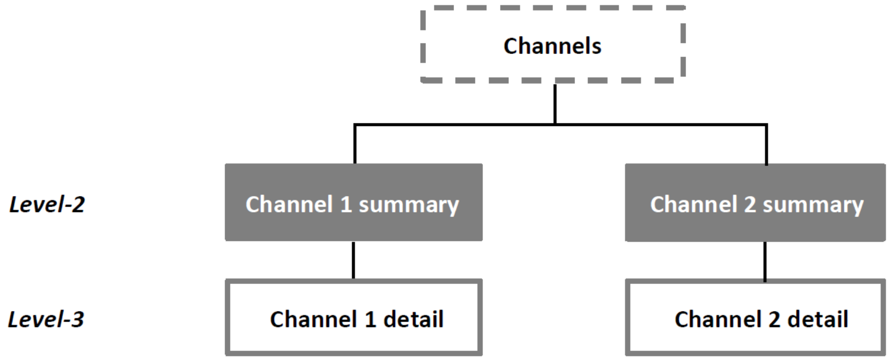


Figure 4.7: Data Model of Channel

Figure 4.6 shows conference data model in three levels. In Level-1, a conference list is defined which includes all the conferences with only summarized information, i.e. without statistics. This level gives an overview of the conferences. Level-2 exposes a specific conference with summarized information. Thus the first two levels provide the possibility for fetching both coarse-grained and fine-grained data. Level-3 exposes the channel information and also complete information of a conference, where the channel information is a *Summary* and the complete information is a *Detail* with statistics. Thus Level-2 and Level-3 realizes the partial representation to enable flexibility in data fetching, which can reduce the network traffic and server load in many cases.

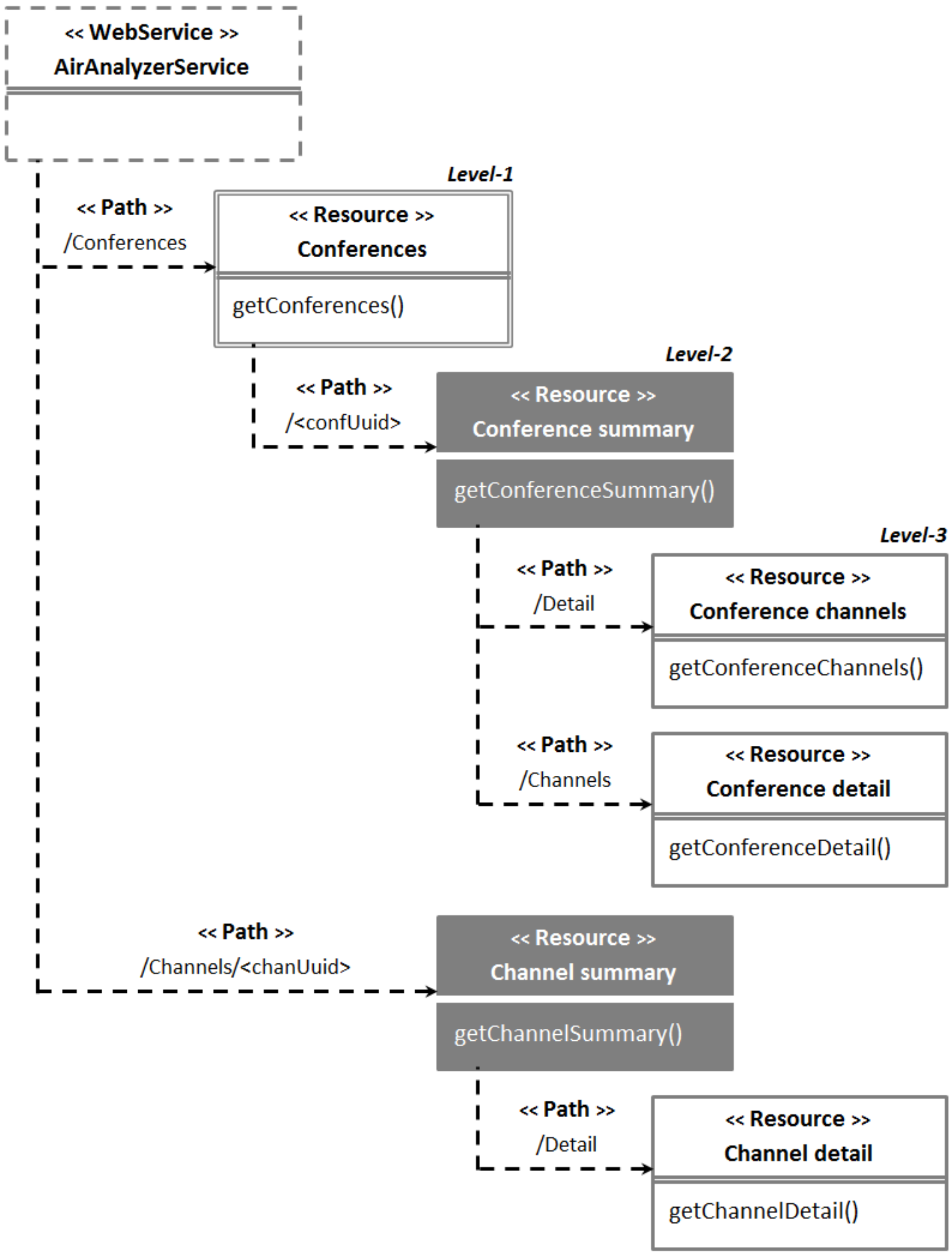


Figure 4.8: URI Model of AirAnalyzerService

Figure 4.7 shows the similar data model for channels. The difference is that Level-1 is unavailable in the current version of *AirAnalyzerService* (it is in a short dash block). This is because for now it is not necessary to separately manage and list all channels as they are still attached to conferences and are available in the conference channels objects. But it is designed in this way so that it can be easily extended when needed.

According to the hierarchical data model of conferences and channels, the URI model can be further designed. The URI model is a model that combines the data model with URIs in REST architecture. As discussed before, RESTful web services expose data as resources and each has a unique identifier, i.e. URI with which the clients can access. The URI model in *AirAnalyzerService* inherits the hierarchy of the data model, as shown in Figure 4.8.

To get the data related to a specific conference or a channel, the uuid of the object needs to be specified in the URI as indicated by `<confUuid>` and `<chanUuid>` in Figure 4.8. Following the uuid an extra tag can be added if further data, e.g. *Detail* or channels is required.

The levels indicated in the URI model can be referred to those in data models, so as the hierarchy. In reality, each URI will include the same prefix defining the name of the web service, host and port that are identified in a servlet container or application server the service is running on. For example, if *AirAnalyzerService* is running on the local machine with port number 8080, then the prefix should be `http://localhost:8080/AirAnalyzerService`. A list of available URI examples can be collected in Table 4.1 (here use '∼' to replace the prefix for simplicity).

URI	**Description**
`∼/QueryAPI/Conferences`	Request for *Summary* of all conferences
`∼/QueryAPI/Conferences/<confUuid>`	Request for *Summary* of one conference
`∼/QueryAPI/Conferences/<confUuid>/Detail`	Request for *Detail* of one conference
`∼/QueryAPI/Conferences/<confUuid>/Channels`	Request for *Summary* of channels of one conference
`∼/QueryAPI/Channels/<chanUuid>`	Request for *Summary* of one channel
`∼/QueryAPI/Channels/<chanUuid>/Detail`	Request for *Detail* of one channel

Table 4.1: List of Request URI

A prefix QueryAPI is set to take extensibility into account. In the current use case the application provides only query service to get data. Nevertheless, it is possible to extend the functionality for updating data in the future. In that case the URIs for those functions can be easily extended with the prefix, e.g. UpdataAPI.

4.3 Summary

Section 4.1 explains and analyzes the requirements for the optimized web service from the perspectives of functional and non-functional requirements. Apart from the basic functions for data processing and service, *AirAnalyzerService* needs to be a flexible and configurable service application with very good usability, maintainability, extensibility, testability, and more importantly, good performance of high throughput and low latency.

Section 4.2 discusses the design of *AirAnalyzerService* based on the specified requirements. It introduces the concepts and ideas of the modern techniques that are brought into the system, and explains why and how they are used to fulfill the requirements. Then follow five subsections to give more details regarding to the use cases, system architecture, processing workflows and data structure with resource model design.

Chapter 5

Implementation and Evaluation

In this chapter the implementation details are introduced and the evaluation of the optimized *AirAnalyzerService* is presented. It starts with the analysis and evaluation of current available web service frameworks, and explanation of reason why the targeted one, i.e. Jersey, is selected. Then more details in the implementation phase based on the system design are given, such as caching mechanisms, list files, user configuration, logging, object interactions, classes and interfaces. Finally the performance testing results regarding to throughput and latency of the service in different scenarios are discussed.

5.1 Web Service Frameworks Analysis

To ease the development process as well as the maintenance efforts in the future, the optimized web service *AirAnalyzerService* is decided to be built upon one of the state-of-the-art frameworks that have been listed in Section 2.1.4. This section presents the analysis of the frameworks and the selection procedure of the final framework Jersey.

5.1.1 General Analysis

To initially narrow down the candidate list, a few bottom line requirements can be emphasized beforehand: the web service needs to be a RESTful web service compatible to both Linux and Microsoft Windows operating systems, capable of data format transferring especially having JSON format support, and preferably built for C/C++ or Java developing language. Therefore, a few frameworks such as .NET Framework, Zend Framework and Gugamarket are out of concern. Further, Restbed is abandoned because of lacking of popularity, complete documentation and reference. That results in three popular and mature candidate frameworks that are considered: gSOAP, Apache Axis2 and Jersey.

A brief background research is conducted on these three frameworks, mainly focusing on their feature sets, usability, maintainability, and security. As successful and mature frameworks, all of the three frameworks are active and well maintained during the last few years, even with recent updates when this thesis work is ongoing. According to the research, they all have a sufficient feature set as needed already or can be realized through extra modules or plugins very easily. A few to be mentioned here are data compression, logging, garbage collection and most importantly, JSON format support as mentioned. Security is fully supported, including HTTPS (HyperText Transfer Protocol Secure), SSL (Secure Sockets Layer), authentication and digital signature. Speaking of usability, gSOAP is a little less convenient regarding to the ease of development, but it is easy to build and test, while the other two Java based frameworks need some more efforts on setting up the runtime environment, for example the configuration of servlet container.

There is one more issue to be clarified here: it is inappropriate to compare gSOAP and Apache Axis2 with Jersey. In fact they are not directly comparable at all since their infrastructures are not with the same level. gSOAP and Apache Axis2 are two more complete and complex frameworks originally designed for SOAP web services. Apart from REST, they also provide other protocols such as MTOM, WSDL and have full WS*-specification support. Jersey is purely a REST framework that is part of Glassfish, which is a project with the same level as gSOAP and Apache Axis2. But that is sufficient for this web service. Thereby it is extracted from Glassfish to compare with the other two frameworks. In other words, in the above comparison it is not the whole gSOAP and Apache Axis2, but their subset or module of RESTful implementation that is being compared.

As a result, Jersey, claimed as a production-ready reference implementation for the standard of JAX-RS (Java API for RESTful Web Services) and also considered more light-weight, is chosen over Apache Axis2 to the next step.

5.1.2 Performance Evaluation

Although gSOAP is more complex than Jersey, it is difficult to tell which one is better unless further evaluation is made. Since performance is another significant factor to be considered in a backend service, it is a significant criterion in the selection of framework.

For this purpose, two simple service applications with exactly the same functionality are developed using the two frameworks, respectively. The service provides only one HTTP GET operation to allow client, i.e. a web browser in this context, to make an HTTP request with query parameters. In this case the parameters include the id and the name of the user. The service then returns the current time with the interpreted parameters in JSON format, similar as an echo service. It is able to use multithreads to handle concurrent requests and the default maximum thread number is set to 200. Below is the example request and response in the test for the service:

- *Request*

  `http://localhost:8080/SimpleSrv/SimpleOps?id=123&name=Alice`

- *Response*

  ```
  {
      "Current time" : "01/01/2016 12:00:00"
      "id" : "123"
      "name" : "Alice"
  }
  ```

The performance test is completed using JMeter[1], a tool that simulates real world service consumption and provides detailed analysis statistics. It is able to automatically generate multiple client requests, concurrently and/or continuously, and interprets the received data. Meanwhile it measures several useful parameters as evaluation references, for example, throughput and latency which are two performance factors considered in this evaluation.

- **Throughput**

  Number of requests that the server can handle in one unit of time, specifically one second.

- **Latency**

  Time interval, measured just before one request is sent till the first response is received, not including any interpreting time from the client side. The deviation of latency is also measured which indicates the variation of latency for each request.

The performance is conducted involving both client side and server side. The client where JMeter is running is a local machine with two 2.9GHz Quad-core processors and 8GB RAM (Random Access Memory). The server where *AirAnalyzerService* is running is remote machine, Dell PowerEdge R610, with two 2.4GHz Quad-core processors and 12GB RAM.

In the test for both frameworks four contexts are considered regarding to concurrency and sample numbers: 10 users and 100 users send the request concurrently, which means JMeter generates 10 or 100 threads (each thread represents a user) to send the requests. To have a large enough data set to measure the throughput and also for better estimation, request generation for each user are repeated 100, 1,000 or 10,000 times such that 10,000 and 100,000 samples in total for each concurrency are measured. This results in a loop in JMeter to repeat generating the threads. In the figures below each context is denoted as: $users \times loops$. For example, 10×1000

[1]`http://jmeter.apache.org`

means 10 users send a request to the server in 1 second, i.e. almost simultaneously, and each of them repeats this request 1000 times. Hence total sample number is 10000. The four contexts are listed in Table 5.1.

Number of users	Number of loops	Number of samples
10	1,000	10,000
100	100	10,000
10	10,000	100,000
100	1,000	100,000

Table 5.1: Test Contexts of Performance Evaluation

Figure 5.1 shows the throughput of gSOAP and Jersey in different contexts. It explicitly illustrates that Jersey has better throughput in all situations and exceptional scalability due to the increase of concurrency and number of total samples.

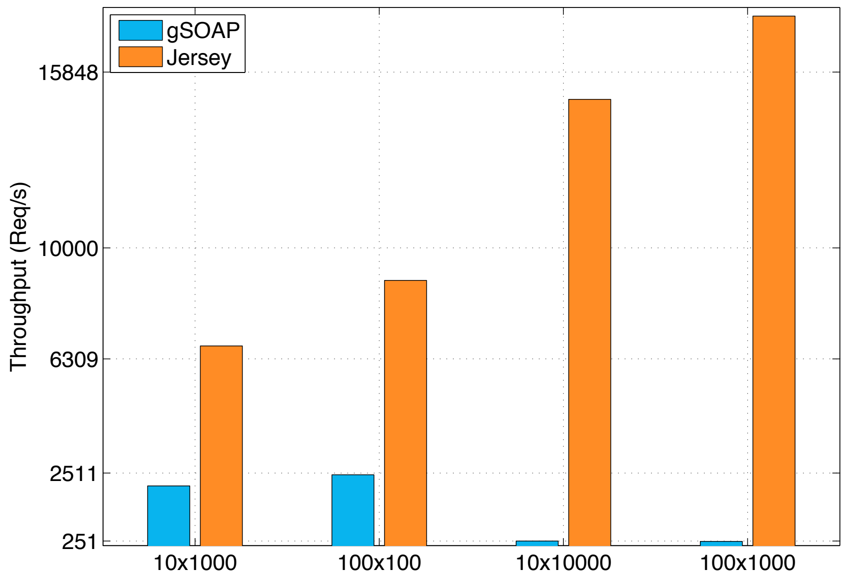


Figure 5.1: Throughput Comparison of gSOAP and Jersey

Latency statistics for both frameworks in each context are shown in Figure 5.2 to Figure 5.5. For each context a figure with box plot and histogram comparison is made. The box plot figure displays the data distribution based on the five metrics: bottom, first quartile, median, third quartile and top, also with some outliers existing outside the range between top and bottom. The first and third quartiles refer to the 25th and 75th percentiles of all samples respectively, while the bottom and top are 1.5 times these percentiles respectively. Since in some contexts the data are too concentrated to form a box, the histogram figure can also help identify the distribution.

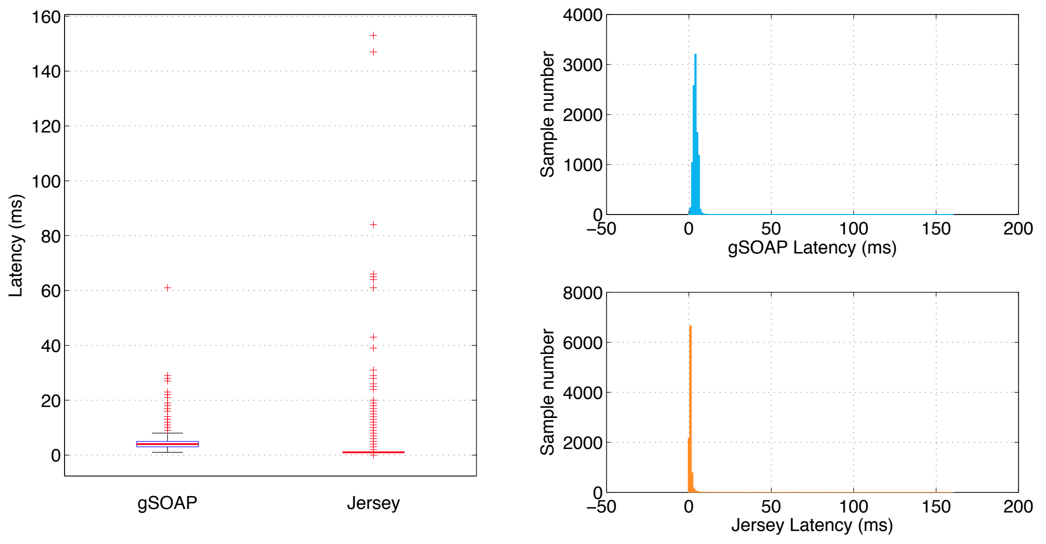


Figure 5.2: Latency Comparison of gSOAP and Jersey (10 × 1000)

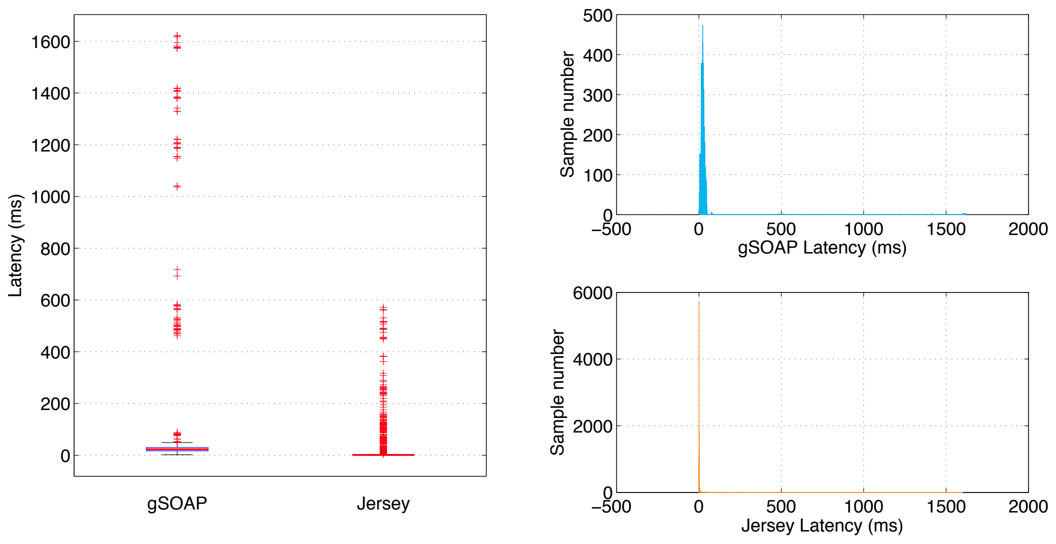


Figure 5.3: Latency Comparison of gSOAP and Jersey (100 × 100)

In general, Jersey has lower latency than gSOAP in all the contexts and is more stable according to the data distribution, except for the 10 × 1000 situation in Figure 5.2 where it has more outliers.

To sum up, Jersey has an overall better performance regarding to throughput and latency than gSOAP. The reasons behind are to be investigated in further research and is out of range of this thesis. But a few hypotheses can be addressed: gSOAP is originally designed for SOAP-based web services. To enable REST it implements an extra module but still has a heavyweight message structure than pure in nature RESTful web service as Jersey. In addition, due to the huge scalability gap between

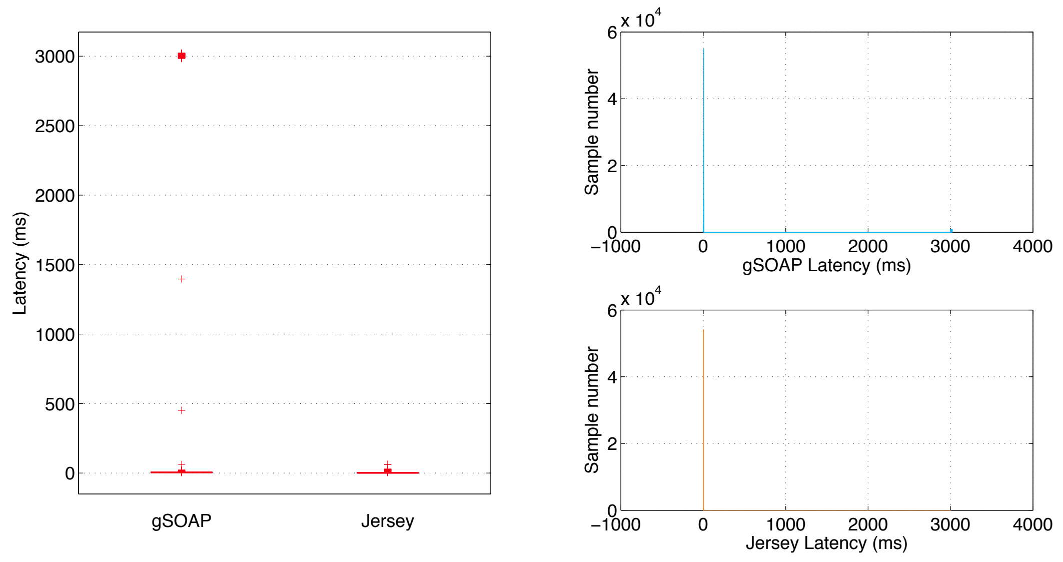


Figure 5.4: Latency Comparison of gSOAP and Jersey (10 × 10000)

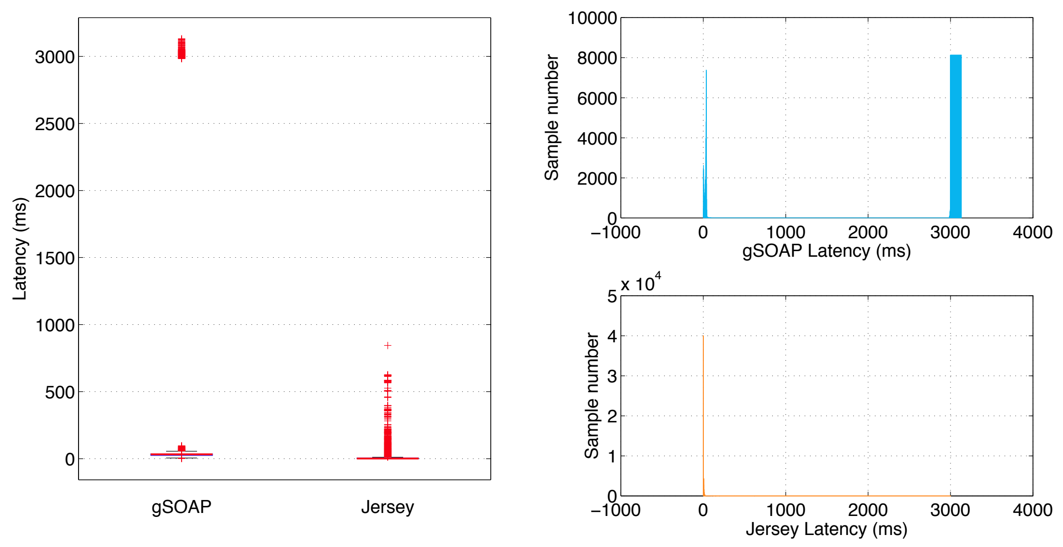


Figure 5.5: Latency Comparison of gSOAP and Jersey (100 × 1000)

the two frameworks the lower level implementation and optimization can be better in the servlet container where Jersey web service runs on.

5.1.3 Conclusion

Based on the theoretical research and testing results shown above, a conclusion can be drawn that the Jersey framework matches the best the optimized web service in this thesis work. It is a standard and mature, extensible and scalable, easy to use and maintain web service framework also with outstanding performance.

Therefore Jersey will be used as a selected framework to design and build the *Air-AnalyzerService* in the following steps in this thesis.

5.2 Implementation

Based on the description in the system design in the last chapter, this section gives more implementation details in the following six sections.

5.2.1 Caching Mechanisms

In this section the implementation of caching mechanisms is discussed, which involves the cache initialization and cache items management including creation, update, replacement and clearance. The Cache class with attributes and interfaces are introduced in Section 5.2.6, where the main idea and functionalities will be presented.

1. **Cache initialization**

   Cache is an individual object created immediately when *AirAnalyzerService* starts running. It is stored in memory and stays passive until *DtCollector* and *DtProcessor* invokes the operations of it. The cache object is initialized by four attributes as shown in Figure 4.15. A type is defined as the replacement strategy, e.g. LRU or LFU. An expiration time is set for all the cache items and used as the criterion in the cache item clearance. A size is an upper bound of memory space to store cache items which is used to trigger cache item replacement when its limit is reached. These first three attributes are user defined as will be shown in Section 5.2.3. The last attribute is a cache map created for cache item storage. Each record in the map is a key-value pair where the key is a name of String as the identifier and the value is the actual cache item. Cache items are designed based on the data resource model introduced in Section 4.2.5 as listed in Table 5.2. This map supports concurrency so that multiple threads are able to read and write its records without conflict or problem of consistency.

   To control the used cache size and ensure the size of all cache items does not exceed the limit, the size of each cache item has to be measured and specified. A specific open source Java library sizeOf[2] is used for this purpose. As shown in the table, the size of the cache items varies from bytes to megabytes. This is because the statistics file sizes of conference and channels have a large range. The *Summary* and *Detail* data object of conferences and channels are directly measurable, based on which the size of conference list and channel list can be calculated.

   However, it should be mentioned that due to the complexity of JVM (Java Virtual Machine) at runtime, the object size measurement is not accurate. But even such estimation is enough for the use case.

[2] `http://sizeof.sourceforge.net`

Key	**Value**	**Description**	**Size**
conference_list	List<Conference>	*Summary* of all conferences	1.3KB × number of conferences
`<confUuid>`_sum	Conference	*Summary* of a conference	1.3KB
`<confUuid>`_det	Conference	*Detail* of a conference	372KB
`<confUuid>`_chl	List<Channel>	*Summary* of all channels of a conference	0.4KB × number of channels
`<chanUuid>`_sum	Channel	*Summary* of a channel	0.4KB
`<chanUuid>`_det	Channel	*Detail* of a channel	1536KB

Table 5.2: List of Cache Items

2. **Cache item management**

   - **Creation**

     One single cache item is created only once, exactly when it is firstly accessed. Thereby this is realized by *DtCollector*. Before an item is created, it needs to check the used cache space if it will exceed the limit after this item is added. If so a replacement mechanism needs to be applied beforehand.

   - **Update**

     Cache item updates are associated with data object updates and depend on the characteristics of the data object itself. Different mechanisms are designed for different situations:

     – If one conference with channel data is added to the directory, the 'conference_list' cache item is deleted from the cache map. The new cache item with the updated data will be created the next time this data is accessed.

     – If, on the contrary, the conference with channel data is removed from the directory, all the related items such as *Summary*, *Detail*, channels are deleted from the cache map.

     – If one conference or channel data is cached but its score is not yet ready, then when score calculation is completed the corresponding cache item will be updated right away to ensure data consistency.

     – Each time a cache item is accessed, its attribute *lastAccessed* and *hitCount* will both be updated. The *lastAccessed* is reset to the current timestamp; the *hitCount* is incremented by one.

     – Each time a cache item is created, updated or deleted, the used space of cache will be updated accordingly. Once the space approaches or exceeds the maximal cache size, the replacement strategy below will be applied.

- **Replacement**

  When the cache size limit is reached, the replacement mechanism is applied according to the predefined replacement strategy. This is to clean some outdated items and leave the space for the new ones. There are two strategies available for configuration: LRU and LFU. For LRU, the *lastAccessed* time of all cache items will be checked and the one with the oldest access time will be removed from cache map; for LFU, the *hitCount* number of all cache items will be checked and the one with the smallest number will be removed from cache map.

- **Clearance**

  Cache clearance is used to provide a higher level cache control mechanism independent of the replacement strategies. A separate attribute, the expiration time, is defined for this. It is combined with the timestamp of each cache item to decide whether the item is out of date. A cache item is considered deprecated under the condition indicated in this formula:

  `Current time - Timestamp` $\geq$ `Expiration time`

  where current time refers to the moment of execution.

As cache is a new technique introduced in *AirAnalyzerService*, its mechanisms need to be configurable by users. More details will follow in Section 5.2.3.

5.2.2 Conference and Channel List Files

The conference and channel list files are used to store the calculated score and other general information of conferences and channels such as uuid and timestamp, for conferences there also involves number of channels. The main purposes of the list files on the one hand, are to avoid duplicate computation of statistics; on the other hand, since the uuid and timestamp are parsed and extracted from folder names or file names, this can avoid iterating file traverse for every disk interaction.

Both list files are created when *AirAnalyzerService* starts running for the first time. Since the files are stored in disk instead of memory, there is no need to recreate them for the second time running. When firstly created, the first line defining all the column headers will be added in the files, i.e. the general information with scores mentioned above. Each conference and channel item are appended in or deleted from the files as a record, similar as a relational database. The records, however, cannot be modified because the data files will not change once are added and only removal is possible. The operations for records are triggered by *DtProcessor*.

According to the data structure discussed in Section 4.2.5, the score of conference has two subsidiaries, avgPLI and avgLI, while the score of channels has the same two subsidiaries and one more called avgPL. These subsidiaries are all stored in the list files of conferences and channels as shown in Table 5.3 and Table 5.4.

ConfUuid	**TimeStamp**	**ChannelNo**	**avgPLI**	**avgLI**

Table 5.3: Conference List File

ConfUuid	**ChanUuid**	**TimeStamp**	**avgPLI**	**avgLI**	**avgPL**

Table 5.4: Channel List File

The channel list file involves the conference uuid of each channel for the reason that the channels in the list for a specific conference can be quickly found. This is useful and efficient in the case, for instance, a conference with channel data files are removed from directory and thus both list files have to be updated.

5.2.3 User Configuration

AirAnalyzerService allows users to configure several parameters for different requirements. The configuration is completed before running the service in a .properties file in the form of key-value pairs with respect to various attributes. The configuration is unit sensitive for flexibility. For example, the time related parameters support unit of hour (h), minute (m) and second (s); the memory size parameter recognizes byte (B), kilobyte (KB), megabyte (MB) and gigabyte (GB).

Following are details of the configurable parameters with explanation and examples.

1. ***Host* and *Port* of server**

   These are two important parameters that have to be included in the URIs to get data. The default value of *Host* is localhost or 127.0.0.1. In this case the service will automatically get the actual IP (Internet Protocol) address of the server and provide to clients. The *Port* can be configured as a four digits number and the most well known one is 8080. An example of configuration:

   Host = localhost
   Port = 8080

2. **Execution preferences**

   The execution preferences consist of three parameters:

- *DtProcessor_ExecFreq*: the time interval to invoke *DtProcessor*, which regularly checks file updates in disk and implements the processing procedure accordingly. This can be configured based on the update frequency of files. The default value is 1h (1 hour).

- *Stats_NumberOfLineToRead*: the number of lines to read from a statistics file. This is customized for the use case because the statistics are recorded by milliseconds. Each statistics record represents a sample of 10 milliseconds. Generally 1 minute records, i.e. 6000 lines statistics are enough for audio data analysis. Thus normally this parameter is set to 6000. But the file may contain statistics more than 1 minute so it can still be configured to read all of them.

- *RootDir*: the directory where all the conference and channel data is placed. In the use case it is a path of the local machine.

An example of configuration:

DtProcessor_ExecFreq = 1h
Stats_NumberOfLineToRead = 6000
RootDir = /uhome/workspace/data/

3. **Cache mechanisms**

The configuration for cache has five parameters:

- *CacheEnable*: whether to use cache or not. Default value: false.
- *CacheType*: the replacement strategy. Default value: LRU.
- *CacheTimeout*: the expiration time of each cache item. Default value: 1m (1 minute).
- *CacheCleanFreq*: the frequency for cache clearance. This is normally less or equal to *CacheTimeout*. Default value: 1m (1 minute).
- *CacheSize*: the maximal memory size to store cache items. This should always be larger than the largest data object Detail of a channel, i.e. 1.4MB. Default value: 2MB.

An example of configuration:

CacheEnable = true
CacheType = LRU
CacheTimeout = 2min
CacheCleanFreq = 30s
CacheSize = 20MB

5.2.4 Logging

To enhance the service usability and maintainability, logging mechanism is introduced in *AirAnalyzerService*. There are already numbers of handy, stable and mature libraries providing logging implementations. Therefore the logging support in this service is delegated to a popular external library called log4j[3]. Due to the fact that logging is not a research topic in this thesis, this section only gives a few examples of the logging information.

- **User configuration**

  Each time the service starts, a list of all the configuration parameters is logged. This is for the purpose of debugging and future investigation into errors or failures.

- **Request handling**

  Each incoming request is logged, together with the data collecting steps and results, e.g. whether the requested data is collected from cache or disk files, how many conferences or channels are found, the current status of memory size for the used and available cache items, etc. If it cannot find the requested data or results in warnings and errors during the collecting procedure this should also be logged.

- **Background processing**

  Each time when *DtProcessor* is invoked is logged, including each step of file update checking and statistics processing, e.g. whether the files are updated, how many conferences are added or deprecated, whether the CRUD operations to the list files are successful etc. It should also log any warning or error encountered.

There is more system runtime information to be logged other than the three listed above. For simplicity they will not be presented in this thesis.

5.2.5 Sequence Diagrams

For the purpose of decoupling the business logics of *AirAnalyzerService*, two separate system workflows are designed, as described in Section 4.2.4. The details of the internal processing procedure can be illustrated in two sequence diagrams. Note that these two diagrams only depict the typical and most complicated situations for the service. More use cases are possible in reality but have similar or simpler workflows, thus they are not shown here.

[3]`http://logging.apache.org/log4j/2.x`

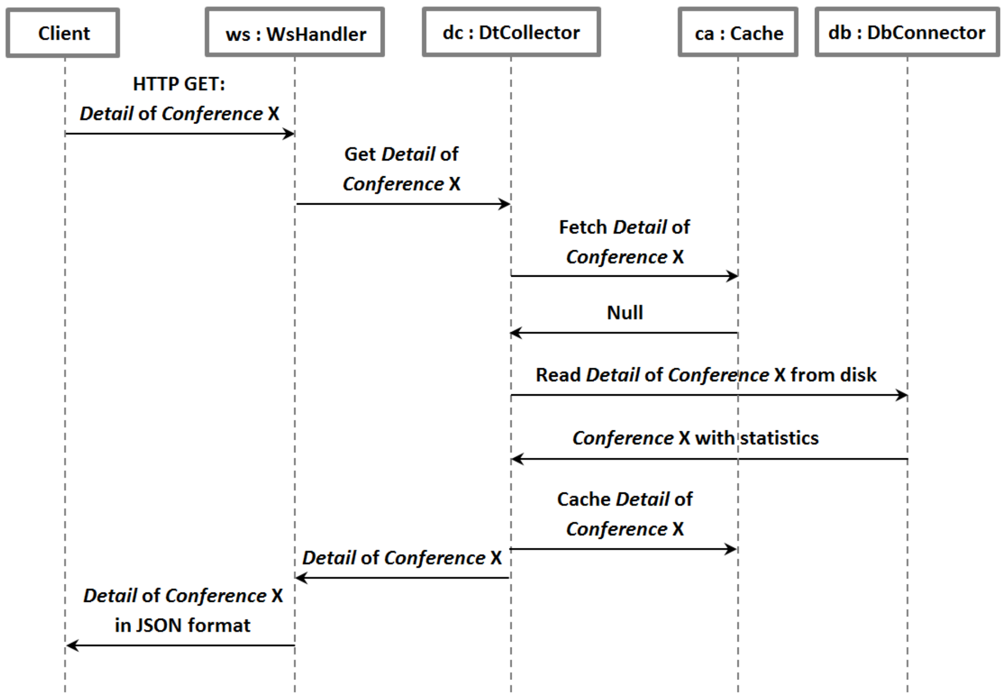

Client
ws : WsHandler
dc : DtCollector
ca : Cache
db : DbConnector
HTTP GET: *Detail* of *Conference* X
Get *Detail* of *Conference* X
Fetch *Detail* of *Conference* X
Null
Read *Detail* of *Conference* X from disk
Conference X with statistics
Cache *Detail* of *Conference* X
Detail of *Conference* X
Detail of *Conference* X in JSON format

Figure 5.6: Sequence Diagram of Conference Data Request Handling

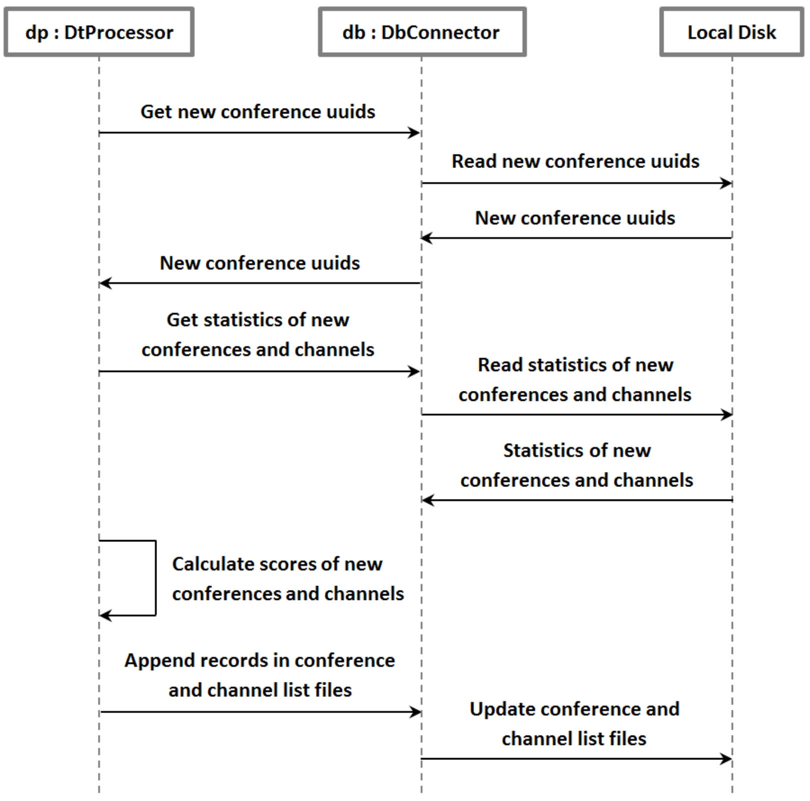

dp : DtProcessor
db : DbConnector
Local Disk
Get new conference uuids
Read new conference uuids
New conference uuids
New conference uuids
Get statistics of new conferences and channels
Read statistics of new conferences and channels
Statistics of new conferences and channels
Calculate scores of new conferences and channels
Append records in conference and channel list files
Update conference and channel list files

Figure 5.7: Sequence Diagram of Background Processing for New Conference Data

Figure 5.6 shows an example of a conference data request handling when the data is not cached. When a client requests for the detailed information of a conference, i.e. with statistics, the system firstly checks its cache whether this information has been cached before. If not as in this case the cache returns null, it reads the data from disk, caches the object and returns the data to the client. Similar workflow applies to other data requests.

Figure 5.7 denotes one cycle of the background processing of newly added conferences and channels. The system firstly gets a list of uuids of the new conferences. For each of the uuid it reads the statistics from disk, based on which calculates the scores of conferences and channels, and then appends all these conferences with their channels into the conference list file and channel list file, respectively. Similar workflow applies to the deprecated conference and channel data, but it is even simpler as no computation is needed.

5.2.6 Classes and Interfaces

Classes and interfaces of *AirAnalyzerService* are designed based on the components and their responsibilities introduced in the system architecture Section 4.2.3. This section presents the components in packages with classes and interfaces. Some useful software design patterns that have been applied in the components and classes design will also be discussed.

1. *WsHandler*

   Figure 5.8 shows the *WsHandler* component with the class of the same name. The class defines the interfaces to get corresponding data of conferences and channels from *DtCollector*, and maps these APIs to the REST interfaces. The return types of the interfaces are String because the data objects have been converted to JSON format.

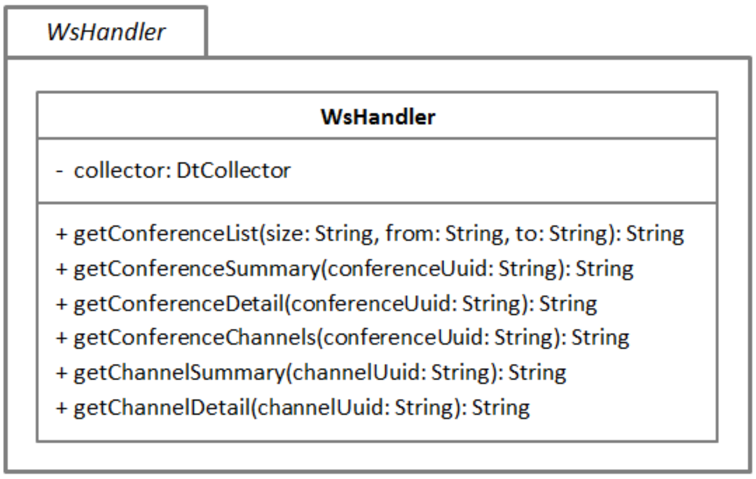

WsHandler
WsHandler
- collector: DtCollector
+ getConferenceList(size: String, from: String, to: String): String
+ getConferenceSummary(conferenceUuid: String): String
+ getConferenceDetail(conferenceUuid: String): String
+ getConferenceChannels(conferenceUuid: String): String
+ getChannelSummary(channelUuid: String): String
+ getChannelDetail(channelUuid: String): String

Figure 5.8: Class Diagram of WsHandler

2. *DtCollector*

   Figure 5.9 is the class diagram for *DtCollector*. All the interfaces for collecting data follow the API names defined in *WsHandler*. The return types, however, remain the actual types of data objects retrieved from the *DbConnector*.

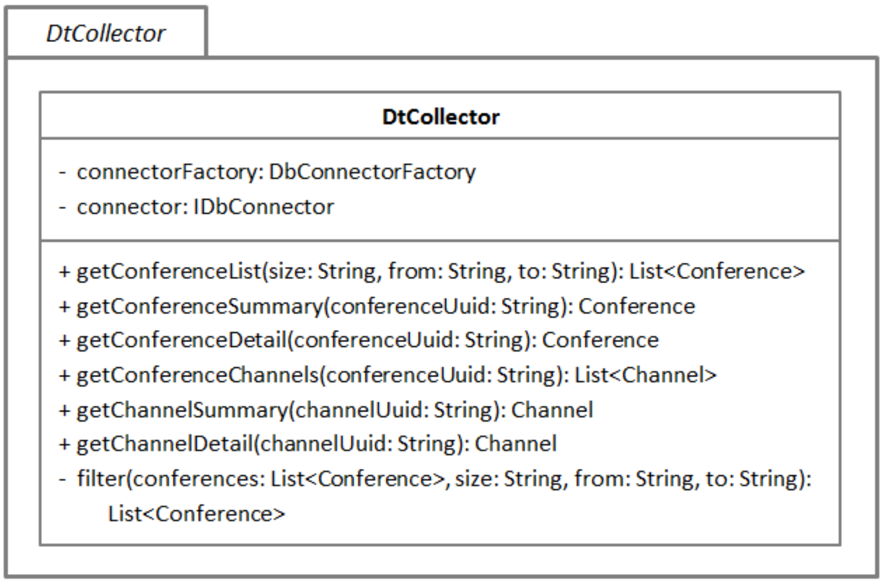

DtCollector
DtCollector
- connectorFactory: DbConnectorFactory
- connector: IDbConnector
+ getConferenceList(size: String, from: String, to: String): List<Conference>
+ getConferenceSummary(conferenceUuid: String): Conference
+ getConferenceDetail(conferenceUuid: String): Conference
+ getConferenceChannels(conferenceUuid: String): List<Channel>
+ getChannelSummary(channelUuid: String): Channel
+ getChannelDetail(channelUuid: String): Channel
- filter(conferences: List<Conference>, size: String, from: String, to: String): List<Conference>

Figure 5.9: Class Diagram of DtCollector

This class defines an extra private method to filter the conference list. Instead of returning data of all conferences, clients are able to filter the list by a number of conferences and timestamps, e.g. conferences earlier or later than a specific date and time. This mechanism intends to provide data fetching flexibility of the service. When partial data is filtered before transmission it also reduces the network traffic, retrieval latency and thus the performance and efficiency of *AirAnalyzerService*.

3. *DbConnector*

Figure 5.10 illustrates the *DbConnector* class diagram which applies the 'Factory Method' design pattern [30]. This pattern allows a flexible and transparent way of creating an object. It is used in *DbConnector* because of the following reason: in the current use case it is enough that the web service gets data from local disk, but from a long term perspective, the conference and channel data will be stored in cloud or migrated to other machines. In that case *AirAnalyzerService* is required to be easily extensible. 'Factory Method' now creates a *LocalDbConnector* object for the use of local disk, but in the future can also create another object for the use of cloud or other machine.

Nevertheless, all types of *DbConnector* objects created by the *DbConnectorFactory* have to deal with file I/O, therefore an *IDbConnector* interface with read, write and update methods is defined for them to inherit.

The *LocalDbConnector* defines the methods to get specified information of conferences and channels. Apart from the functional needs, they must also meet the requirements of lazy loading and partial representation and provide corresponding APIs.

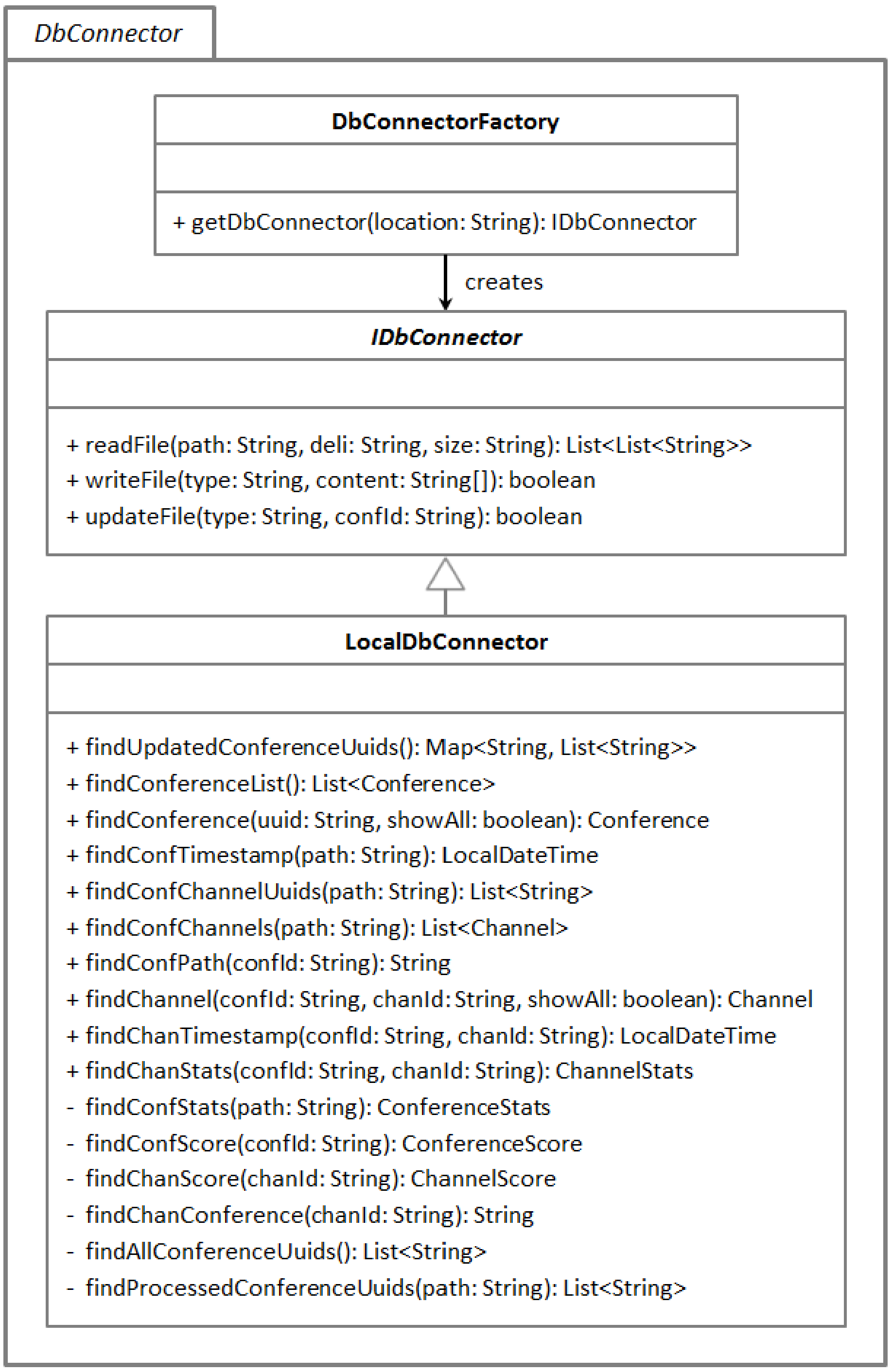

DbConnector
DbConnectorFactory
+ getDbConnector(location: String): IDbConnector
creates
IDbConnector
+ readFile(path: String, deli: String, size: String): List<List<String>>
+ writeFile(type: String, content: String[]): boolean
+ updateFile(type: String, confId: String): boolean
LocalDbConnector
+ findUpdatedConferenceUuids(): Map<String, List<String>>
+ findConferenceList(): List<Conference>
+ findConference(uuid: String, showAll: boolean): Conference
+ findConfTimestamp(path: String): LocalDateTime
+ findConfChannelUuids(path: String): List<String>
+ findConfChannels(path: String): List<Channel>
+ findConfPath(confId: String): String
+ findChannel(confId: String, chanId: String, showAll: boolean): Channel
+ findChanTimestamp(confId: String, chanId: String): LocalDateTime
+ findChanStats(confId: String, chanId: String): ChannelStats
- findConfStats(path: String): ConferenceStats
- findConfScore(confId: String): ConferenceScore
- findChanScore(chanId: String): ChannelScore
- findChanConference(chanId: String): String
- findAllConferenceUuids(): List<String>
- findProcessedConferenceUuids(path: String): List<String>

Figure 5.10: Class Diagram of DbConnector

4. *DtProcessor*

The class diagram of *DtProcessor* is shown in Figure 5.11. Here the 'Strategy' design pattern is used [30]. This pattern enables to change the class behavior or its algorithm at runtime. It is helpful for *AirAnalyzerService* since the methodology provided to calculate conference and channel scores may vary based on different algorithms, although at the moment only one algorithm is available. This makes it easy to implement other algorithms to exchange current algorithm. The strategies are to check if the data files are updated, and if yes, calculate scores for new conferences and channels and then update the list files for newly added and deprecated data files.

As mentioned in Section 4.2.1, the concept of lazy loading is introduced in the service. This happens when a conference or a channel is requested but its score is not yet ready. This data object can then be cached. After the score is calculated, the cached object has to be updated to ensure data consistency. This becomes another responsibility of *DtProcessor*, i.e. it needs to update the cache objects for changes of data objects. The other case is when a conference with its channels is removed from disk, the corresponding objects will also be removed from cache. These cache updates are realized in the private methods of *DtProcessor* class.

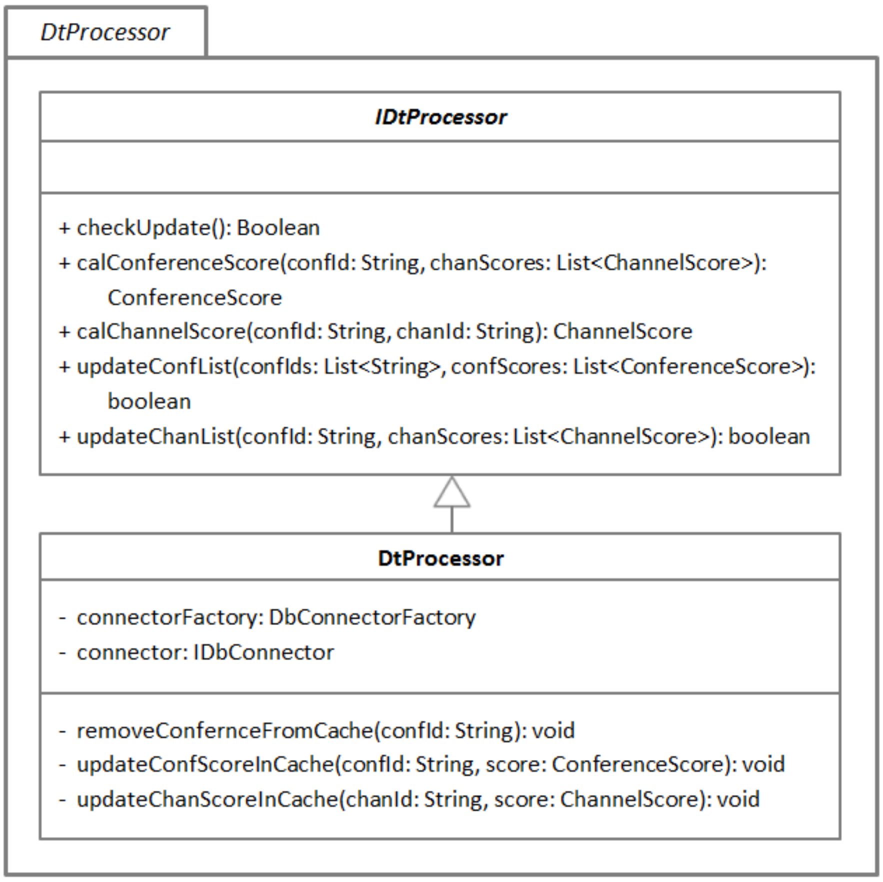

DtProcessor
IDtProcessor
+ checkUpdate(): Boolean
+ calConferenceScore(confId: String, chanScores: List<ChannelScore>): ConferenceScore
+ calChannelScore(confId: String, chanId: String): ChannelScore
+ updateConfList(confIds: List<String>, confScores: List<ConferenceScore>): boolean
+ updateChanList(confId: String, chanScores: List<ChannelScore>): boolean
DtProcessor
- connectorFactory: DbConnectorFactory
- connector: IDbConnector
- removeConfernceFromCache(confId: String): void
- updateConfScoreInCache(confId: String, score: ConferenceScore): void
- updateChanScoreInCache(chanId: String, score: ChannelScore): void

Figure 5.11: Class Diagram of DtProcessor

5. *Cache*

Figure 5.12 shows two classes in *Cache* module. The *Cache* class is designed as a global object that is created immediately when the service starts running. Its type (currently either LRU or LFU), expiration time and size are configured by users. It holds a cache map to manage all the cache items in key-value pairs, and defines all the CRUD operations for them. Each cache item has a cache object which is defined as generic types. In this case it can be any data resource described in the last section, e.g. a conference list, a specific conference or channel, either *Summary* or *Detail*. Besides, three more attributes are defined in the cache item independent of cache type:

- *timestamp*: used for the Expiration Time strategy. It is initially set when this cache object is created.
- *lastAccessed*: used for LRU strategy. It is updated each time the cache object is accessed.
- *hitCount*: used for LFU strategy. It is updated each time the cache object is accessed.

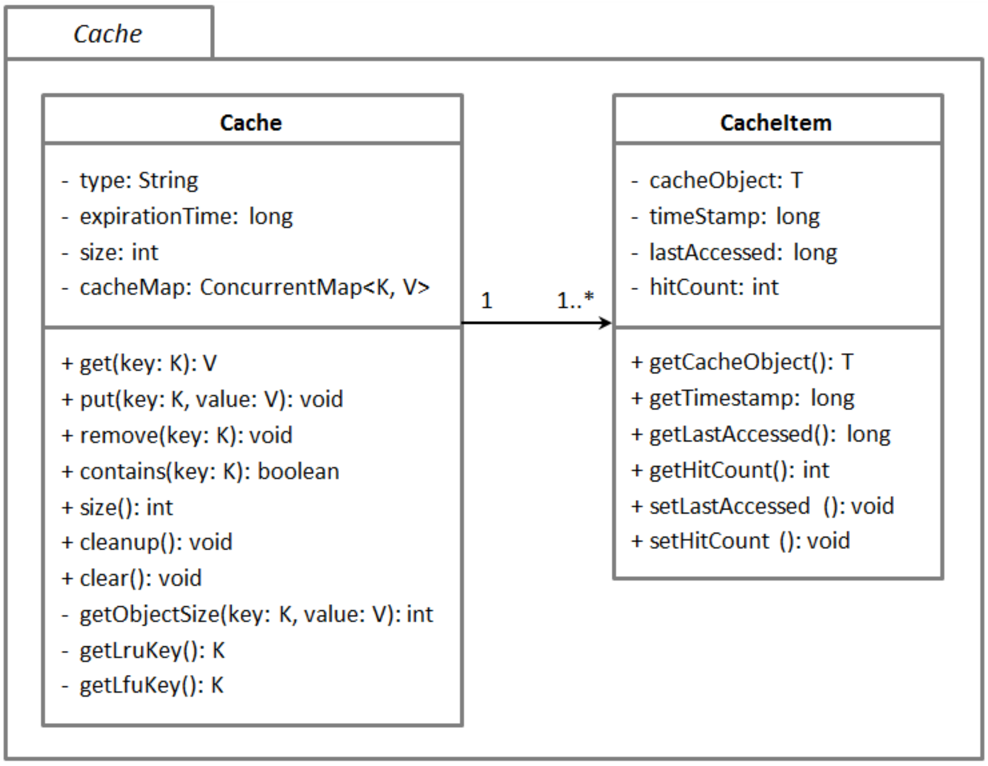


Figure 5.12: Class Diagram of Cache

5.3 Evaluation

The evaluation of the optimized web service is based on performance tests focusing on throughput and latency, as defined in Section 5.1.2. In this section the testing concepts are introduced and the test results are presented. The results have demonstrated that the overall design, selected framework and applied techniques of *AirAnalyzerService* have a positive impact on its efficiency and performance.

5.3.1 Performance Test Concepts

Due to the fact that *AirAnalyzerService* is flexible in various use cases and it allows multiple user configurations especially for caching mechanism, there are multiple factors in different dimensions that will influence its performance. All these factors need to be tested to evaluate their impacts on client requests handling and data retrieval. The factors can be set with different values regarding to different situations in different tests. In each single test a specified combination of parameter setting is made such that a multidimensional test results can be obtained. Table 5.5 shows all the factors with parameter values designed to be tested in the evaluation.

Number of conferences	**Number of users**	**Request-based data object size**	**Cache coverage**	**Cache type**
		0.4KB (channel *Summary*)		
	1	1.3KB (conference *Summary*)	0%	
10	10	13/130KB (conference list, depending on number of conferences)	25%	LRU
100	20		50%	LFU
	50	372KB (conference *Detail*)	75%	
		1536KB (channel *Detail*)	100%	

Table 5.5: Parameters of Test Factors

The factors are described as follows:

- **Number of conferences (and channels)**

  For a database, normally the more records it contains, the longer it takes to find the requested information. Similar rule applies here: the number of conference and channel affects the speed of retrieving specific data because time of traversing files varies. In reality, the number of conference can vary from a few to hundreds, while the number of channels for each conference varies from 2 to 10.

  To get enough number of conference data and channel data for testing, the data used in the following tests is duplicated from one typical conference with three channels which is standard in reality. Thus the 10 or 100 conference data is identical. This is acceptable here because the tests do not involve score calculation (which is already completed beforehand) and how the statistics are formed does not influence the data retrieval. The service needs to traverse the files to fetch requested data anyway, not caring the specific value in the files.

- **Number of (concurrent) users**

  The ability of handling concurrency is also an important characteristic for web services. Normally the more concurrent users making a request to a service, the longer latency it has for each of them. This can surely be improved by multiprocessing and multithread programming, but there will still be a queue when the number of users exceeds the number of processors and cores of the server machine. In reality, the number of concurrency can vary from 1 to 50 as *AirAnalyzerService* is not a public service but only for internal use.

- **Client request-based data object size**

  Currently *AirAnalyzerService* is able to serve six main types of resources, as introduced in Section 4.2.5. These resources are with different size of data objects and thereby have an impact on the time of retrieval, including serialization/deserialization, transmission and interpretation. This is straightforward as the larger the data size is, the slower the resource can be retrieved, even if compressing mechanisms are applied.

  In the following tests five of six typical requests are tested. The one for requesting channels of a conference is not that popular in reality and therefore is not included. According to Section 5.2.1, the related data object size varies from 0.4KB to 1536KB. This is also based on the testing data generated from real case mentioned above, and the number of lines to read from statistics files, as explained in Section 5.2.3, is 6000. This means the statistics files contain more lines of data, but only the first 6000 lines data are fetched and served.

- **Cache coverage**

  Cache size is configurable in the web service and will be initialized when the server starts. It happens frequently that the cache size is not large enough to store all the request data and that results in cache replacement mechanism taking place. Thus, the coverage of cache can influence the performance of the service in a way that the more data objects it can cache, the faster it can respond to clients. In reality, cache coverage can vary from 0% to 100%, i.e. cache can be disabled, or enabled with a fixed size to store part of the data objects, or configured with a large enough size to store all the data objects for every possible client request.

- **Cache type**

  As described in Section 5.2.1, two cache types are supported in *AirAnalyzerService*: LRU and LFU. As their names suggest, how they affect the performance is highly dependent on the concrete scenarios and user behaviors. For example, it is predictable that if resources are only popular in a certain period and will be updated or deprecated, then it is better to use LRU; however, if popular resources need to be accessed and reviewed periodically, LFU is a better mechanism to choose.

The tests are conducted in two scenarios for different purposes. The first scenario is to test the service performance in an ideal case where all clients request for the exactly same data. The main purpose is to test the new caching mechanism for the service by comparing the performance between handling requests with complete cache coverage and no cache at all. The second scenario is to test the impact of cache coverage in a more realistic situation that each client requests for different data objects.

For testing purpose a web container is needed. *AirAnalyzerService* is implemented with an embedded Grizzly server[4] which together with Jersey is a sub project of Glassfish as mentioned in Section 2.1.4. The default settings of the server are applied such that it automatically manages the CPU (Central Processing Unit), memory etc. system resources when dealing with concurrency.

Same as the framework performance evaluation presented in Section 5.1.2, all the following tests involving client side and server side are completed with JMeter. The client side where JMeter is running is the same machine as before with two 2.9GHz Quad-core processors and 8GB RAM. The server side where *AirAnalyzerService* is running is a remote machine, a Cisco N20-B6625-1 with four 2.6GHz six-core processors and 16GB RAM. The maximal bandwidth between client and server is 28Mbps.

5.3.2 Test of Cache Activation for Identical Client Requests

Caching mechanism is newly introduced in *AirAnalyzerService* aiming to enhance service performance and speed of data retrieval. This test is designed to test the performance of the service with and without cache, in which the parameter of cache coverage is set to 100% and 0%, i.e. cache enabled with large enough size or cache completely disabled, respectively. The cache type in this case does not matter since no replacement strategy is required. Thereby it is set to a default value, i.e. LRU, when cache is enabled. All other parameters listed in Table 5.5 will then be tested. According to the table, there are two possibilities in number of conference, four possibilities in number of concurrent users and five possibilities in request-based data object size. By multiplying these possibilities by the two in cache coverage mentioned above, there are totally 80 test cases in this scenario, in which both throughput and latency are measured.

To simplify the scenario in each test case, it is assumed that all users request for the same data object; to get a large enough data set for throughput measurement, it is configured that all those identical client requests are repeatedly generated, similar as the tests of the web service frameworks in Section 5.1.2.

The overall test results are shown in the following four figures. Figure 5.13 and Figure 5.14 denote the throughput comparisons in the two situations regarding to number of conferences, 10 and 100 respectively. Figure 5.15 and Figure 5.16 illustrate

[4]`https://grizzly.java.net`

the latency comparisons in those two cases, where the y axis is in log scale for better denotion. In each figure, four graphs based on different concurrency, i.e. 1, 10, 20 and 50 users, are depicted individually. Each of these graphs illustrates the throughput or latency regarding to different request-based data sizes, i.e. 0.4KB, 1.3KB, 13KB/130KB (10/100 conference data), 372KB and 1536KB, in the contexts of with and without cache. Note that 13/130 KB represents a list of conferences, while other sizes represent data of only one conference or channel. All values are derived as a mean value from three independent experiments. Some throughput values are too small to display in the figures which can be found in Appendix A.

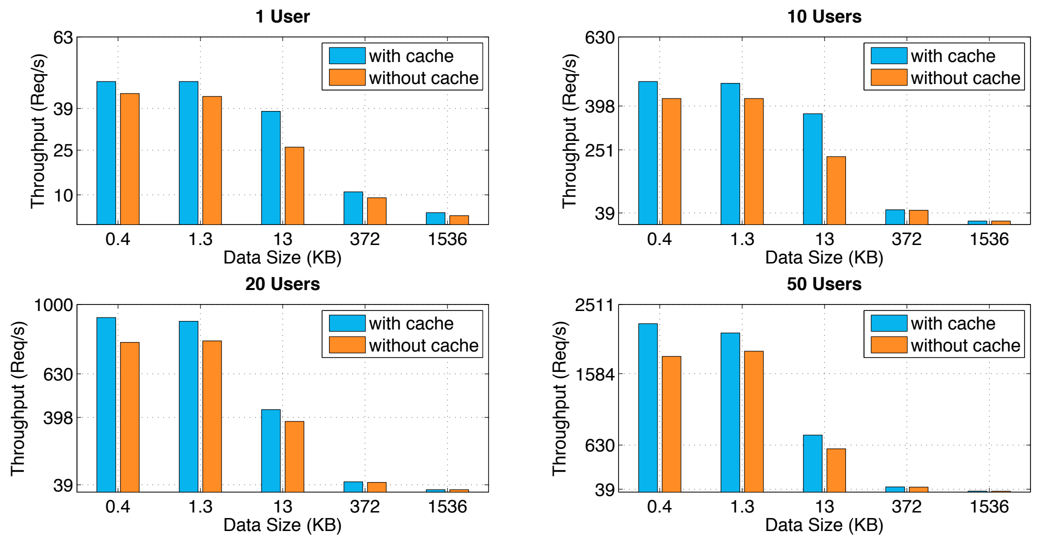


Figure 5.13: Throughput Comparison for Cache Activation
With Respect to Data Size (10 Conferences)

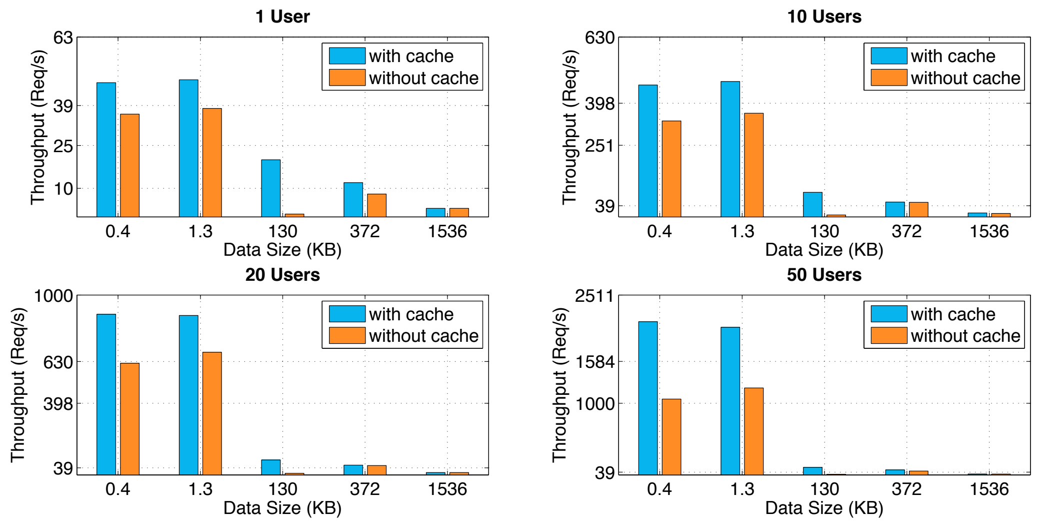


Figure 5.14: Throughput Comparison for Cache Activation
With Respect to Data Size (100 Conferences)

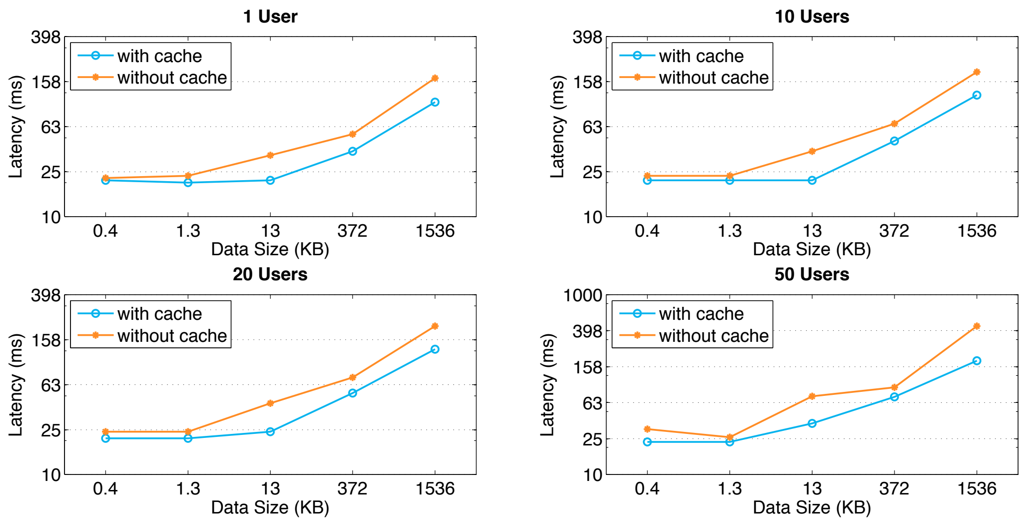


Figure 5.15: Average Latency Comparison for Cache Activation With Respect to Data Size (10 Conferences)

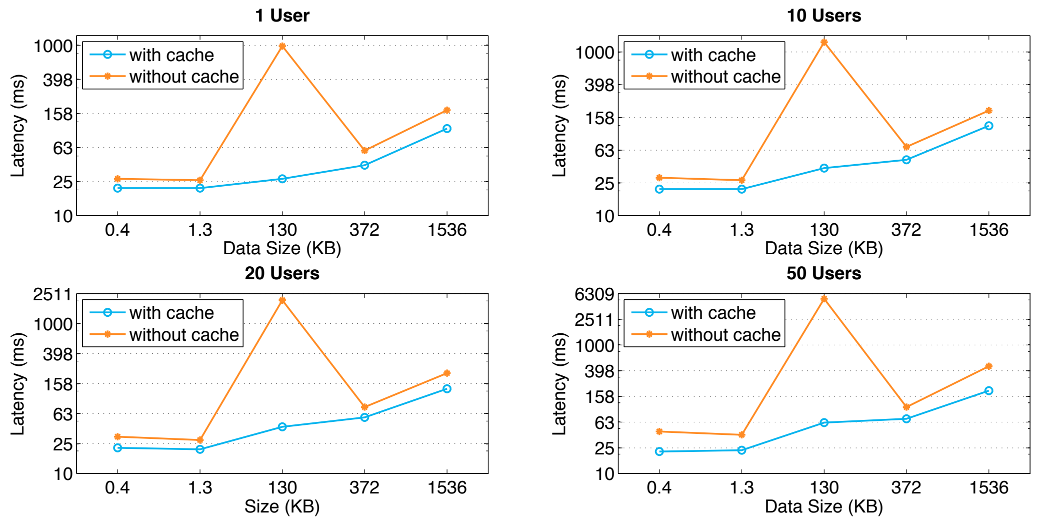


Figure 5.16: Average Latency Comparison for Cache Activation With Respect to Data Size (100 Conferences)

A few observations on throughput and latency can be addressed from Figure 5.13 to Figure 5.16:

- **Impact of cache on performance**

  From Figure 5.13 and Figure 5.14, no much throughput difference can be seen for large data size, e.g. a few hundred or over a thousand kilobytes. This is because the data object size dominates the response time and eliminates the impact of cache. For smaller data object size, *AirAnalyzerService* with cache has higher throughput than that without cache. The more conference data, the more enhancements the cache brings: For 10 conference data, it improves around 10% to 70%; for 100 conference data, it improves around

20% to 1500%. The largest difference comes from retrieving a conference list although the data object size is intermediate (130KB), where the system has to traverse all the conference files to fetch all their channel *Summary*. This can take a long time when there exist a large number of conferences, even making the impact of data object size negligible.

Figure 5.15 and Figure 5.16 shows that the service with cache obtains lower latency than that without cache, especially for larger data object size while for smaller size the overhead of using cache cannot compensate the beneficial speed it brings. For 10 conference data, it can be at maximum 2.2× times faster; for 100 conference data, generally it can be around 1.5× to 2.6× times faster. One exception is getting a list of 100 conferences with 130KB data object size as mentioned above, where caching is able to provide a 37× to 92× times faster service depending on the concurrency.

- **Impact of number of conferences on performance**

  When comparing Figure 5.13 with Figure 5.14 and Figure 5.15 with Figure 5.16, it can be seen that conference number does not have a performance impact on the service with cache. This is reasonable since the requested information is already stored in cache and can be immediately returned from memory, therefore it does not matter how much conference data is stored in the disk.

  However, for service without cache, throughput goes down significantly and latency goes up as the number of conferences increases because it takes more time to traverse the files. But this does not apply to large data size, e.g. a few hundred or over a thousand kilobytes, because that in this case takes lead in the response time and eliminates the impact of conference number.

- **Impact of number of concurrent users on performance**

  The comparison for concurrent users can be made by comparing the four graphs in each of the Figure 5.13 to Figure 5.16. In general both throughput and latency of *AirAnalyzerService* is in proportion to the number of concurrent users. The reason is that on the one hand, the server machine has multicore processors which can handle concurrency for a number of users to increase throughput; on the other hand, if the number of users exceeds the number of processing cores, there will be a queue which also increases latency for each request of user.

  Two exceptions are found though. One is that when the data object size is small, e.g. 0.4KB or 1.3KB, almost negligible difference can be seen in the latency of service with cache. This is because the main overhead here is the execution of the program, i.e. the minimum processing time dominates. The other is that when the data object size is large, e.g. 1536KB, the service throughput is bounded by data serialization and transmission. As a result 10 or 50 concurrent users do not cause any difference.

- **Impact of data object size on performance**

  The results for this comparison makes no surprise as the expectation is the larger size the data object is, the slower the service can respond. All the figures provide the proof of this with an exception when retrieving a list of 100 conferences. The reason has been explained above: the file traversing time takes the domination over object sizes and thereby it takes longer than fetching the *Detail* of a conference or channel.

Above gives the result of this performance test as a global view. However, more details of each single case can be tracked to further analyze the internal processing status of *AirAnalyzerService*, as shown in the following box plot and histogram figures. These figures are generated from the latency values of each request in each test case to illustrate the deviation. Since there are 80 test cases in total, only some typical cases are selected to present here.

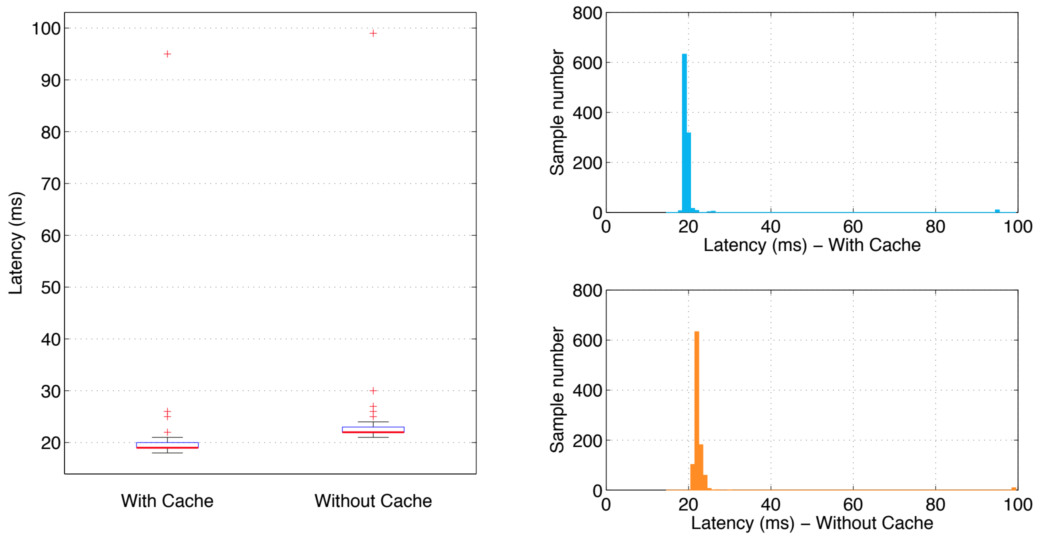


Figure 5.17: Latency Comparison for Cache Activation in Light Server Loads (10 conferences, 10 concurrent users, 0.4KB data size)

A general observation from the latency of those test cases is: The heavier the server load is (e.g. larger number of conferences and concurrent users, larger data object size), the smaller deviation the latency has. The reason is that when the server load is heavy enough to cause a queue for the requests, some of requests (outliers in the graphs) in the queue have to wait longer until they can be served. Besides, under heavy server loads the service without cache has larger deviation than that with cache.

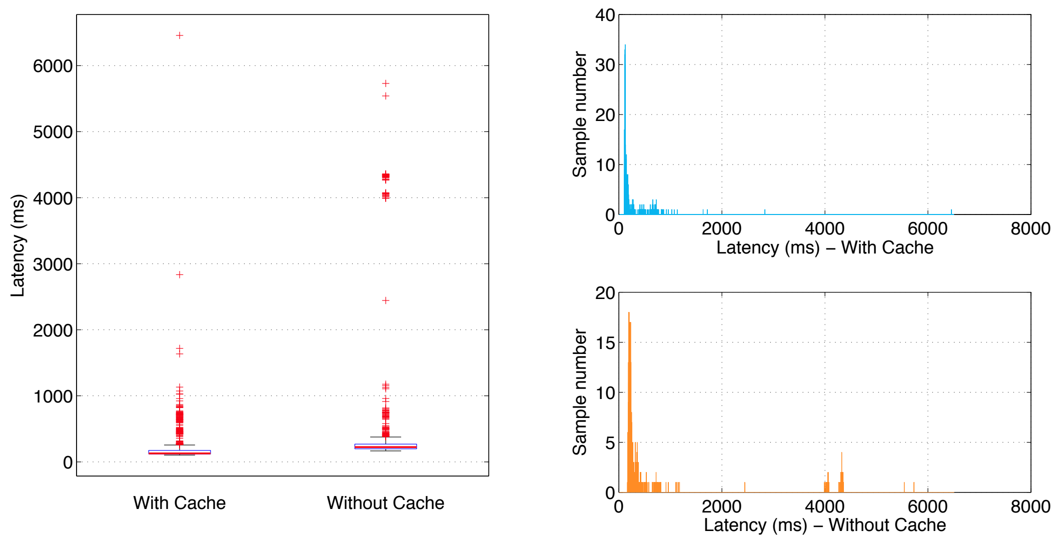


Figure 5.18: Latency Comparison for Cache Activation in Heavy Server Loads (100 conferences, 50 concurrent users, 1536KB data size)

Figure 5.17 shows an example of light server load in the test case with 10 conference data, 10 concurrent users and 0.4KB data size. The deviation of the service with and without cache makes almost no difference. Figure 5.18 denotes the deviation of service with and without cache under a heavier server load, i.e. 100 conference data, 50 concurrent users and 1536KB data size. It shows that the service without cache has larger deviation than that with cache. In both test cases the total number of samples is 1000. When comparing both figures it can also be seen that the service under heavy server loads has smaller deviation than that with light server loads.

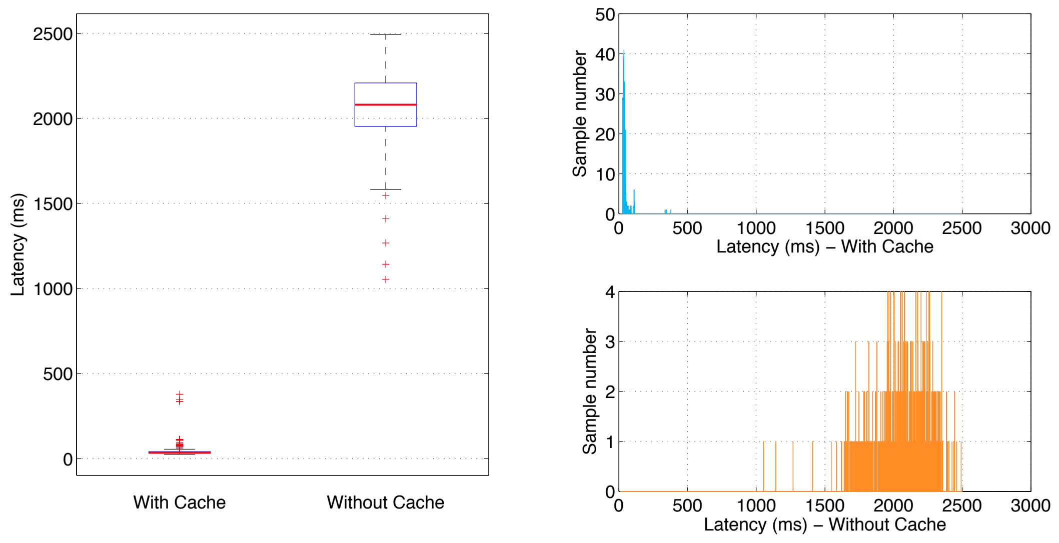


Figure 5.19: Latency Comparison for Cache Activation in Special Test Case (100 conferences, 20 concurrent users, 130KB data size)

An exception which has been mentioned before also exists in the deviation evaluation: the retrieval of a conference list (130KB data size) when cache is disabled and the number of conferences is large, i.e. 100 in this test. As it has been shown and explained that the throughput is extremely low and the latency is extremely high, the deviation is also large in this test case. Figure 5.19 denotes the test case with 20 concurrent users. A dramatic difference can be seen between cache enabled and cache disabled, where the total number of samples is 600.

Figure 5.13 to Figure 5.19 show a comparison of trend of the throughput, latency and deviation in variety of test cases. More details of this test can be found in Appendix A, where a table of all the result values in each test case is attached.

5.3.3 Test of Cache Coverage for Diverse Client Requests

The first test presented before simulates an ideal scenario that all users request for the same data to test the impact of cache activation in different situations. This second test, however, is designed to test the impact of cache coverage for a more realistic scenario that all users make different requests.

Due to the fact that the reality can be more complicated than simulation, the worst situation is assumed here: All users request for the largest data, i.e. the channel *Detail* with 1536KB data size. The concurrency is set to 10, 20 and 50 and the number of conferences is 100 which is closer to real cases. This test mainly focuses on the cache coverage so that all its parameters listed in Table 5.5 will be tested, i.e. 0%, 25%, 50%, 75% and 100%. This means, for example, if 10 concurrent users request for 10 different channel data and 50% cache coverage is pre-configured in the service, only 5 channel data can be cached. For each new coming request, the replacement mechanism will be triggered. From the results of the previous test, it makes little difference of throughput in the case of requesting large size of data. Thereby only latency and deviation are measured in this case.

In this test, not only the LRU, but also the LFU replacement mechanism is tested. Therefore each channel *Detail* needs to be accessed multiple times. This is realized by repeating each client request several times and thus, the mean value of latency can be calculated from all these samples. For instance, if 10 concurrent users request 10 different channel *Detail* and each user repeats 10 times, there are 100 samples in total and the average value of latency is computed for evaluation. The requested channels are normally distributed across the channel list, i.e. the users do not request 10 continuous channels. Instead, the first user requests the 1st channel, the second user requests the 11th one, the third user requests the 21st one, and so on. Note that the order of generation of all these requests is random and as a result the cached objects and the execution of LRU or LFU are random. This is actually useful because in real cases the client requests are also random.

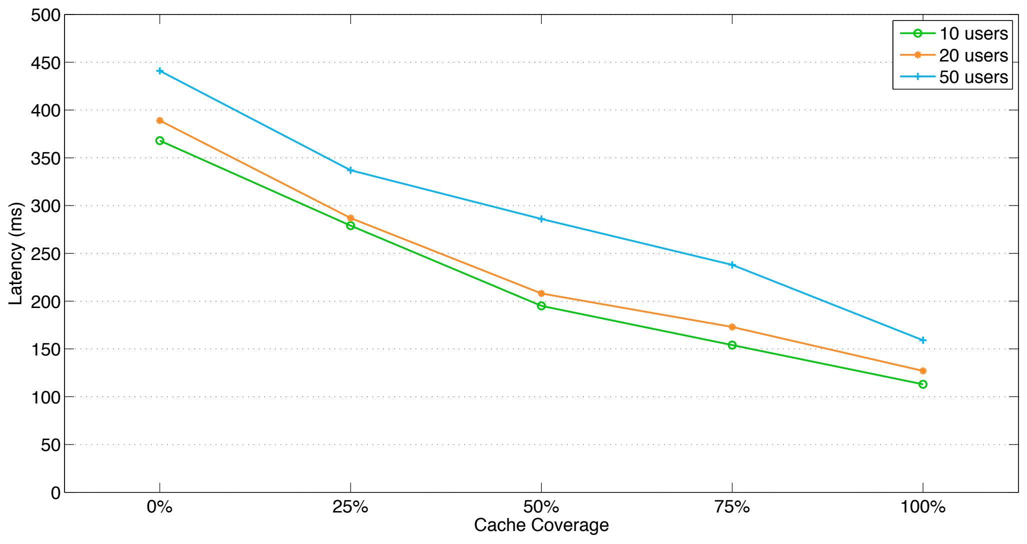


Figure 5.20: Average Latency Comparison for Cache Coverage
(100 conferences, 1536KB data size, LRU type)

Figure 5.20 denotes the comparison of average latency in different LRU cache coverage and concurrency. The observation is reasonable: The more the cache coverage, the faster the service response; the more the concurrency, the slower the service response. The latency in 100% cache coverage is around one third of 0% cache coverage. This proves again that the service with cache is more efficient.

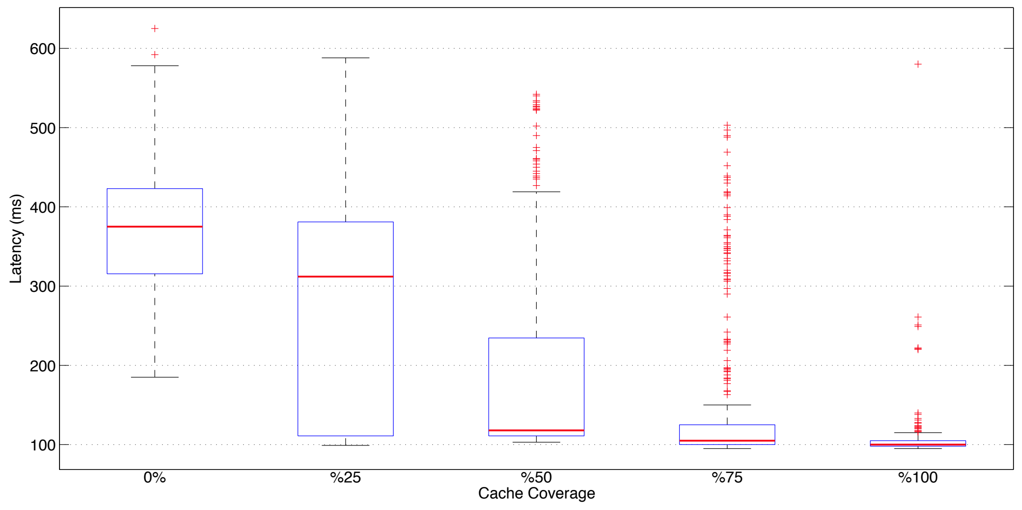


Figure 5.21: Latency Comparison for Cache Coverage
(100 conferences, 10 concurrent users, 1536KB data size, LRU type)

Another observation is from the comparison of the latency in Figure 5.20 and that in Figure 5.16. It shows that when users request different data from the service with cache, the latency is lower than when they request identical data. But for the service without cache, the latency of requesting different data is higher than

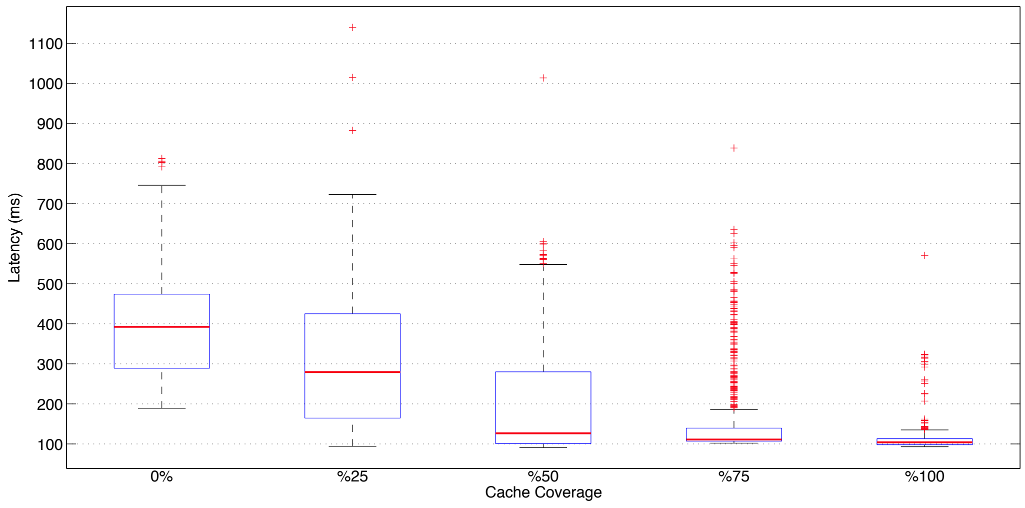


Figure 5.22: Latency Comparison for Cache Coverage
(100 conferences, 20 concurrent users, 1536KB data size, LRU type)

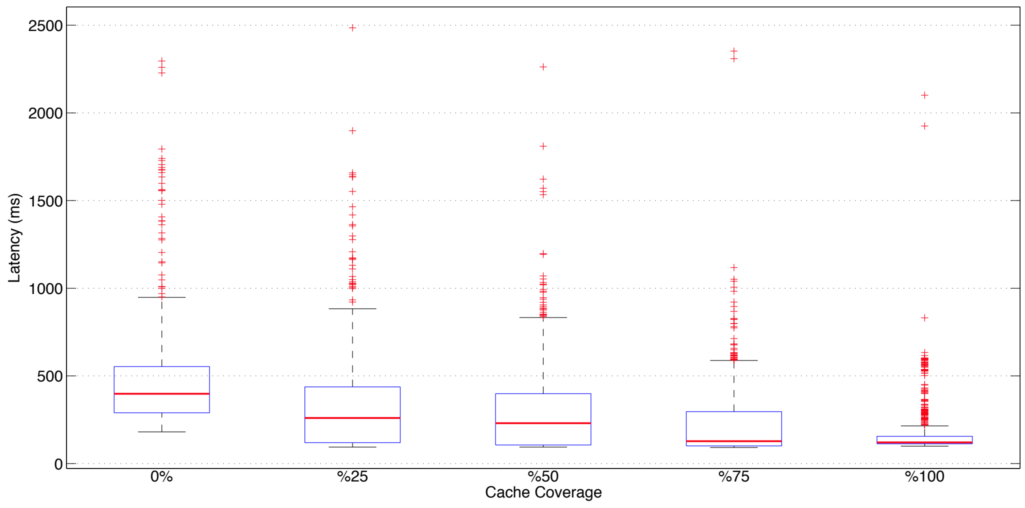


Figure 5.23: Latency Comparison for Cache Coverage
(100 conferences, 50 concurrent users, 1536KB data size, LRU type)

requesting identical data. The reason is as follows: For service with cache, when all users access the same object in memory, even concurrent read is supported, there can be a collision especially in high concurrency that increases the latency; but if the service involves no cache, i.e. data is retrieved from disk, the time of traversing files takes much longer than the system deals with the collision, therefore requesting for different data has higher latency.

Figure 5.21 to Figure 5.23 give details of latency comparison with respect to different cache coverage and concurrency. All figures show that in the case of cache activated in the service, more cache coverage results in smaller deviation. This is resulted from the random order of requests arrival and handling at the server side. In the case of large cache coverage, the latencies for each request vary less because the incoming requests are more likely to be cached already. Moreover, the figures shows that the service without cache has smaller deviation as those with 25%, 50% and 75% cache coverage. This is reasonable since there will be less difference in latency if all accesses are towards disk, instead of some towards disk and some towards memory. Only when all accesses are towards memory can it achieve the smallest deviation.

When comparing the three figures, it can be seen that for the same cache coverage, higher concurrency or more concurrent users cause larger deviation of latency. This is consistent with the results in the previous test as the service has to handle a longer queue of requests and involves more internal processes.

The same test has been conducted with LFU cache replacement strategy. Due to the randomness of the order of client requests, similar result as LFU strategy is obtained. It is appended in Appendix B for reference.

5.3.4 Conclusion

Based on the results of the two tests above, it can be summed up that the newly introduced caching mechanism enhances the overall performance of *AirAnalyzerService*. In particular, the throughput, latency as well as the deviation of the optimized web service are significantly improved in most of the test cases of various scenarios. In addition, the cache coverage has a noticeable influence on latency and deviation where the larger the cache size is configured, the lower latency and smaller deviation the service has. Therefore, whenever possible users are encouraged to enable cache with large enough size of their application in any case.

5.4 Summary

Section 5.1 analyzes three candidates of web service framework regarding to feature set, usability, maintainability, security and more importantly, performance evaluation based on specific tests. According to the analysis and evaluation it finalizes the selection of usage of Jersey framework for *AirAnalyzerService*.

Section 5.2 explains the implementation details based on the concepts and technologies introduced into the service optimization, such as caching mechanisms, list files, user configuration and logging; and the system design described in Chapter 4, including sequence diagrams, diagrams of classes and interfaces.

Section 5.3 presents the evaluation of *AirAnalyzerService* mainly focused on performance tests and discusses the test results in different scenarios. It shows that the caching mechanism in particular brings significant improvements and enhancements to the throughput, latency and deviation of the optimized web service.

Chapter 6

Discussion and Outlook

This chapter firstly summarizes the achieved goals of the optimized *AirAnalyzerService* and gives answers to the research questions proposed in Chapter 1. Then a discussion of service limitations follows, and based on the limitations it raises an outlook of the service that possible work can be performed in the future.

6.1 Achievements

AirAnalyzerService, as an optimized web service for audio data analysis, is able to read the data of conferences with channels, processes it with score calculation based on pre-defined algorithms, and serve the data upon client requests via REST interfaces in a simple and efficient way. It is designed and implemented with a new architecture, business logics, resource model and state-of-the-art web technologies based on the analysis of the requirements and bottleneck of the previously developed prototype.

The service is optimized in the following aspects:

1. **Configuration**

   The service allows multiple user configurations (Section 5.2.3) which are convenient to use. Users can configure the service properties based on practical needs and real situations. This also enhances the flexibility of the system.

2. **Usability**

   The REST architecture (Section 2.1.3) is used to expose data resources so that clients can retrieve data by making simple HTTP requests. An efficient URI model (Section 4.2.5) is designed in accordance with the interfaces to reduce the network traffic and retrieval latency. The model has a clear structure for conference list, individual conference and channel as well as a decoupled design of resources such as *Summary* and *Detail* of the conferences and channels. Furthermore, a filter mechanism (Section 5.2.6) is implemented for the conference list to provide convenience of data retrieval under different needs.

3. **Maintainability**

   From the detailed analysis and evaluation (Section 5.1) the Jersey framework is selected to be used in the implementation of the service. This is a mature, stable and reliable web service framework that is also easy to use and maintain. Besides, the current version of service involves an embedded web server Grizzly which is also well developed and maintained by external experts. This avoids extra efforts in web container configurations and maintenance in *AirAnalyzerService*. As mentioned above, logging can also help in the maintainability of the service.

   In addition, a logging mechanism is added to the service so that all the runtime information can be tracked and analyzed in case of any problem or system failure (Section 5.2.4).

4. **Flexibility and Extensibility**

   AirAnalyzerService has a modular architecture (Section 4.2.3) in which the components or concrete algorithms can be flexibly exchanged and extended. For example, in this use case the *DbConnector* is implemented to access conference and channel data in local disk. However, in other use cases the data can be stored elsewhere, e.g. on cloud. Then another implementation of *DbConnector* can be done to fulfill the new requirements. Another example is the algorithm of score calculation in *DtProcessor*. This can also be easily exchanged when needed.

5. **Testability**

   Unit tests have been conducted for *AirAnalyzerService*. For testability purpose, these codes will be supplied in the package for users. Thereby developers and test engineers have a reference and are able to efficiently make automatic testing.

6. **Performance**

   The performance improvement is mainly contributed by two mechanisms:

   - **File Caching**

     Two list files (Section 5.2.2) are created for storing general information and the calculated scores of conferences and channels, respectively. This mechanism effectively avoids redundant processing of statistics and as a result prevents repeated computation and resource consumption of the system.

   - **In-memory Caching**

     In-memory caching mechanism (Section 5.2.1) is used to store the concrete requested data objects in memory so that they can be accessed more quickly in the future. The cached objects can be updated if the corresponding object has been changed in the system to ensure consistency, and they can expire in a certain amount of time to prevent obsolete caches. According to the results from performance tests (Section 5.3) in the evaluation phase, the caching mechanism makes a significant enhancement for the optimized web service.

Based on the achievements summarized above as well as the work introduced in Chapter 4 and Chapter 5, the answers of the research questions proposed in Section 1.3 can be addressed as follows:

1. What kind of tool and technique can be used in the optimized web service for better maintainability and performance?

   As discussed in Section 4.2.1, the optimized web service, i.e. *AirAnalyzerService*, applies the REST architecture due to its simplicity and efficiency. Jersey framework is selected to be used for the service implementation after an analysis and performance evaluation of various web service frameworks has been completed. The architecture and framework provides good maintainability and performance.

   Furthermore, the caching mechanism with LRU and LFU replacement strategies and the list files for processing results storage are introduced in *AirAnalyzerService* to enhance service performance.

2. How is the optimized web service designed in order to ensure better flexibility and extensibility?

   AirAnalyzerService is designed as a modular web service with four main components and an extra cache component which can be optionally configured. Both the components themselves and the specific algorithms within them can be flexibly exchanged or extended under realistic needs. There are also multiple configurations possibly supported which give users flexible choices to run or use the service, for instance, configuration of host and port of the server, execution, cache etc.

3. How is the optimized web service implemented with integration of the proposed tool and technique?

   REST architecture is naturally supported in Jersey framework, where the mapping of URI and internal methods can be realized via pre-defined annotations. Jersey is included as an external library in the service together with the embedded Grizzly server so that *AirAnalyzerService* is able to run as a stand-alone service without further web containers.

   In addition, all the logics of caching are implemented in a separate optional component that allows users to configure all its properties such as activation, replacement strategy, size etc. The list files for conference and channel are used to store their general information and calculated scores so that a large number of file traversing can be avoided.

4. How does the optimized web service perform in the use case of audio data analysis?

   According to the performance tests presented in Section 5.3, *AirAnalyzerService* is proved to have good throughput, low latency and deviation when cache is enabled and configured with a large enough size. The tests are completed with multiple and concurrent client requests which also proves that the optimized service has good scalability.

The current version of *AirAnalyzerService* can be shipped as a software package and run on various platforms. Companies and organizations which provide web conferencing service can immediately use it for audio data analysis in order to improve speech quality. Using this service can significantly increase the efficiency of the workflow and thus reduce the business cost.

6.2 Limitations and Outlook

Although *AirAnalyzerService* has been successfully optimized from the previous prototype and proved to have good performance under request intensive environment, there are some limitations with respect to the evaluation and implementation that have been aware of. Within the time frame of this thesis work these limitations cannot be covered and therefore they will be put in the to-do list for outlook together with some further enhancement plan.

The limitations and outlook are focused on the following two aspects:

1. **Design and Implementation**

   The design and implementation can be improved in four perspectives:

   - **Score placeholder and data completion**

     At the moment, there still exists a potential gap of data completion between the score calculation and data serving in *AirAnalyzerService*: Client may get the information of a new conference or channel without its score if it is not processed. This situation is highly dependent on the execution frequency of *DtProcessor* that is configured by the system administrator since only the *DtProcessor* can calculate the score and update both of the list files and the cache object.

     To fill this gap, a possible solution is to couple the *WsHandler* and *DtProcessor* by making the latter one callable to the former one, as shown by the yellow dash arrow in Figure 6.1. As a result, when *WsHandler* finds the score field of the requested data from *DtCollector* is empty, it can invoke *DtProcessor* to process the statistics and fill in the resulting score before serving the data to the client.

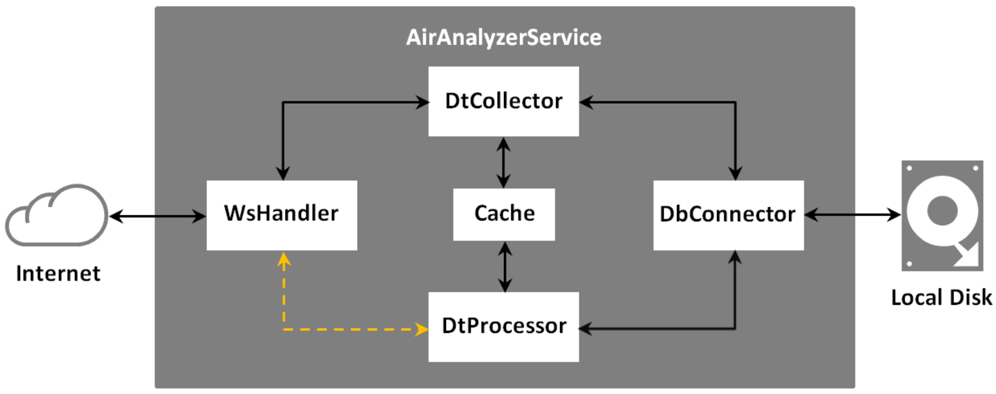


Figure 6.1: Architecture of Enhanced AirAnalyzerServicee

- **List file of conferences**

  As shown in the performance test in Section 5.3, in the case of large number of conferences with cache disabled, there always exist small throughput and huge latency when requesting the list of conferences. This is because retrieving the channels of each conference requires a large amount of time. The idea to solve this problem is to store the summarized information of channels for each conference in the conference list file, including the uuid, timestamp and score. Thereby *DtCollector* can collect all the required information in one single read of the file to serve the conference list request which can certainly result in performance improvement.

- **Caching replacement strategies**

  Section 2.2.2 mentions more cache replacement strategies other than LRU, LFU and expiration time which are used in *AirAnalyzerService*. These strategies are not only based on recency or frequency but also sizes, cost of retrieval etc., and some of them can be smarter and more efficient. In the further enhancement of design and implementation it is possible and helpful to bring these strategies into the system and evaluate the corresponding performance.

- **Security**

  AirAnalyzerService is not implemented with any security mechanism, which means if the service is running on a server, anyone who knows the server IP address and listening port of the service can make a request. This is fine for internal use, but for long term consideration it has to be secure. Basic authentication and authority checking can be added to the service in future work to solve this problem.

2. **Evaluatioin**

   The evaluation can be improved in two perspectives:

   - **Further investigation of current evaluation**

     In Section 5.3.2 and Section 5.3.3, two performance tests with different purposes and some discussions of the results are presented. However, the discussion and analysis are based on practical experience and theoretical hypothesis without verification. To investigate the test results in more details and verify the analysis, further scientific evaluation can be conducted and used for future improvements.

     One proposal is to measure the throughput and the latency in a more fine-grained way to find out the processing time of each step in the whole data retrieval procedure, e.g. file I/O, statistics processing, data serialization/deserialization, data transmission etc. If, for example, data serialization takes the majority time of latency, then the caching mechanism can be extended and applied after this step. Thus, the service only has to execute this step once for each data object which avoids repeating processes. As a result further enhancement of the service performance is possible.

   - **Evaluation on system resource consumption**

     Due to the fact that it is not a concerned characteristic in this use case, the system resource consumption, e.g. CPU and memory utilization, is not evaluated in the performance tests in this thesis. However, there is no doubt that it is an important aspect of evaluation for any software and application. Therefore this can be put in the future work.

Moreover, more implementation and evaluation work can be addressed regarding to the embedded server. The current version of *AirAnalyzerService* uses Grizzly, in which although the number of parallel threads can be configured to handle concurrent clients, unfortunately the default value is not given in the documentation. More servers are available such as Tomcat[1], Jetty[2], Simple[3] etc. which can have influence on the service performance.

6.3 Summary

Section 6.1 lists the achievements of *AirAnalyzerService* and gives a summary of the main enhanced points and techniques that have been applied. It also answers the research questions of this thesis based on the work introduced in previous chapters.

[1] `http://tomcat.apache.org/`
[2] `http://www.eclipse.org/jetty`
[3] `http://www.simpleframework.org`

Section 6.2 discusses the limitations of the current optimized web service regarding to design, implementation and evaluation. Then it gives the possible solutions to cover these limitations for future work.

Chapter 7

Conclusion

AirAnalyzerService is proposed in this thesis in order to optimize the backend service of a web application prototype for the purpose of audio data analysis. It is designed and implemented as a RESTful web service to retrieve, process and serve audio data by exposing simple and clear interfaces. Multiple and concurrent client requests are also supported. The system consists of several modules and components with user configuration and system logging supports. These properties provide better usability, flexibility, extensibility, and maintainability for both the server side and client side. In addition, modern tools and techniques, for example, Jersey framework, caching mechanism etc. are applied to enhance the service performance.

The evaluation shows that the service with caching mechanism has better throughput, lower latency and deviation than that without caching in most of the test cases, especially in the case of serving the conference list. A large size of caching configuration can also have a positive influence on the performance.

Nevertheless, the future work for *AirAnalyzerService* can be focused on the functionality and evaluation enhancement. The statistics processing can be triggered by requests such that clients are able to get the complete information of new data. More information can be cached and more caching strategies can be used to enhance performance. Security should also be concerned for the next step. Moreover, the current evaluation can be investigated into more details and more characteristics can be measured and analyzed. These results can be used to make further enhancement plans.

Appendix A

Test Results of Cache Activation

Section 5.3.2 shows the test results in figures to compare the throughput, latency and deviation in the cases of cache activated and deactivated. In order to give more details of each test case, the following figures show the tables of the values of throughput, latency and deviation based on which the figures are generated.

As mentioned before there are in total 80 test cases combined with the following factors and parameters:

Number of conferences: 10, 100
Number of concurrent users: 1, 10, 20, 50
Size of data object: 0.4KB, 1.3KB, 13KB/130KB (depending on the number of conferences), 372KB, 1536KB
Cache Coverage: 0%, 100%

Each cell in the tables shows the values of both cases of cache activated and deactivated (i.e. 100% and 0%), divided by '/'. For example, in the first cell '48/44', '48' is the throughput value when cache is activated while '44' is the value for cache deactivated. The units for throughput and latency not shown in the tables are:

Throughput: request/second (req/s)
Latency: millisecond (ms)

Deviation is the standard deviation of latency which does not have any unit.

Table A.1 denotes the test results for 10 conference data, while Table A.2 shows the results for 100 conference data.

	1	**10**	**20**	**50**	
Throughput	48/44	480/423	930/798	2253/1816	**0.4KB**
Latency	21/22	21/23	21/24	23/32	
Deviation	3/3	7/8	9/11	12/12	
Throughput	48/43	474/423	910/806	2129/1885	**1.3KB**
Latency	20/23	21/23	21/24	23/26	
Deviation	4/3	8/8	10/11	13/13	
Throughput	38/26	372/228	440/377	763/580	**13KB**
Latency	21/35	21/38	24/43	37/74	
Deviation	4/4	8/9	11/12	45/22	
Throughput	11/9	50/48	55/52	70/68	**372KB**
Latency	38/54	47/67	53/73	73/93	
Deviation	4/5	24/28	44/55	86/98	
Throughput	4/3	12/12	13/13	14/13	**1536KB**
Latency	104/170	120/193	130/209	184/447	
Deviation	9/7	18/21	78/64	178/1036	

Table A.1: Test Results of Cache Activation (10 Conferences)

	1	**10**	**20**	**50**	
Throughput	47/36	462/336	893/621	2138/1058	**0.4KB**
Latency	21/27	21/29	21/31	22/45	
Deviation	4/3	9/9	11/10	13/15	
Throughput	48/48	474/363	886/682	2061/1213	**1.3KB**
Latency	21/26	21/27	21/28	23/40	
Deviation	3/3	8/8	10/11	14/15	
Throughput	20/1	86/7	83/9	105/9	**130KB**
Latency	27/979	38/1316	42/2056	62/5254	
Deviation	5/9	27/33	36/196	88/434	
Throughput	12/8	52/51	54/52	71/54	**372KB**
Latency	39/58	48/69	56/77	71/108	
Deviation	7/5	20/19	46/51	71/125	
Throughput	3/3	14/12	13/13	13/12	**1536KB**
Latency	105/173	125/192	135/218	195/468	
Deviation	6/7	44/23	72/92	169/933	

Table A.2: Test Results of Cache Activation (100 Conferences)

Appendix B

Test Results of Cache Coverage

Section 5.3.3 mentions that in the test of cache coverage the cache type does not have a dramatic impact on the performance and therefore only shows the results for the LRU type. However, for a better comparison and reference the results of both LRU and LFU types are listed in Table B.1 below.

As mentioned before there are in total 30 test cases combined with the following factors and parameters:

Number of conferences: 100
Number of concurrent users: 10, 20, 50
Size of data object: 1536KB
Cache Coverage: 0%, 25%, 50%, 75%, 100%
Cache Type: LRU, LFU

	10	**20**	**50**	
Latency	368	381	441	**0%**
Deviation	105	131	253	
Latency	279/273	287/288	337/335	**25%**
Deviation	141/139	165/156	238/221	
Latency	195/208	208/219	286/257	**50%**
Deviation	149/140	150/149	231/201	
Latency	154/169	173/172	238/216	**75%**
Deviation	113/127	207/119	207/190	
Latency	113	127	169	**100%**
Deviation	21	63	151	

Table B.1: Test Results of Cache Coverage (100 Conferences)

Note that throughput is not measured in this test. In addition, in the cases of 0% and 100% cache coverage there is no difference between LRU and LFU because none of these replacement strategies will be executed. Thus, they share the same results. For other coverage parameters each cell shows the values of both cases of LRU and LFU, divided by '/'. For example, in the cell '279/273', '279' is the latency value of LRU while '273' is the value of LFU. The unit for latency is also ms as before and no unit for deviation.

List of Figures

List of Tables

Acronyms

API	Application Program Interface
CGI	Common Gateway Interface
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DDL	Data Definition Language
EJB	Enterprise Java Beans
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
I/O	Input/Output
IP	Internet Protocol
JAX-RS	Java API for RESTful Web Services
J2EE	Java 2 Platform, Enterprise Edition
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LFU	Least Frequently Used
LRU	Least Recently Used
NASA	National Aeronautics and Space Administration
OOA	Object-Oriented Architecture
RAM	Random Access Memory
REST	REpresentational State Transfer
RMI	Remote Method Invocation
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
RSS	Rich Site Summary
SDK	Software Development Kit
SDL	Services Description Language
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer

UDDI	Universal Description, Discovery and Integration
UI	User Interface
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WS	Web Service
WSDL	Web Service Definition Language
XML	Extensible Markup Language

Bibliography

[1] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to web services architecture. *IBM Systems journal*, 41(2):170, 2002. 2.1.1, 2.1

[2] Kieran Mockford. Web services architecture. *BT Technology Journal*, 22(1):19–26, 2004. 2.1.1

[3] Jørgen Thelin. A comparison of service-oriented, resource-oriented, and object-oriented architecture styles. In *OMG Workshop, Munich, Germany*, 2003. 2.1.2

[4] Doug Barry. Web services explained. *Online at: http://www. servicearchitecture. com/web-services/articles/web_services_explained. html Last accessed*, 6:30, 2010. 2.1.3, 2.2, 2.1.3, 2.3

[5] Jian Meng, Shujun Mei, and Zhao Yan. Restful web services: A solution for distributed data integration. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–4. IEEE, 2009. 2.1.3

[6] Leonard Richardson and Sam Ruby. *RESTful web services.* " O'Reilly Media, Inc.", 2008. 2.1.3

[7] Ricardo Ramos de Oliveira, Vieira Sanchez, Robson Vinicius, Julio C Estrella, Renata Pontin de Mattos Fortes, and Valerio Brusamolin. Comparative evaluation of the maintainability of restful and soap-wsdl web services. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the*, pages 40–49. IEEE, 2013. 2.1.3

[8] Greg Barish and Katia Obraczke. World wide web caching: Trends and techniques. *IEEE Communications magazine*, 38(5):178–184, 2000. 2.2.1

[9] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011. 2.2.1, 2.4

[10] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003. 2.2.2, 2.2.2

[11] Ws wiki stack comparison. http://wiki.apache.org/ws/StackComparison. Accessed: 2011-11-15. 3.1

[12] Feature comparison of wsf/c and gsoap. http://wso2.com/library/articles/feature-comparrision-wsf-c-gsoap. Accessed: 2008-09-24. 3.1

[13] Web service attachments support matrix. http://webservices20.blogspot.de/2008/10/web-service-attachments-support-matrix_11.html. Accessed: 2009-04-05. 3.1

[14] Better open source enterprise c++ web services: Introducing wso2 web services framework for c++. http://www.slideshare.net/wso2.org/better-open-source-enterprise-c-web-services. Accessed: 2009-11-11. 3.1, 3.1

[15] High performance web services in c++. http://wso2.com/library/articles/high-performance-web-services-c. Accessed: 2010-02-14. 3.1

[16] Nicholas Jillings, Brecht De Man, David Moffat, and Joshua D Reiss. Web audio evaluation tool: A browser-based listening test environment. 2015. 3.2.1

[17] Chris Cannam, Christian Landone, and Mark Sandler. Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1467–1468. ACM, 2010. 3.2.1

[18] Johannes Rainer, Fatima Sanchez-Cabo, Gernot Stocker, Alexander Sturn, and Zlatko Trajanoski. Carmaweb: comprehensive r-and bioconductor-based web service for microarray data analysis. *Nucleic acids research*, 34(suppl 2):W498–W503, 2006. 3.2.2

[19] Stephen W Berrick, Gregory Leptoukh, John D Farley, and Hualan Rui. Giovanni: a web service workflow-based data visualization and analysis system. *Geoscience and Remote Sensing, IEEE Transactions on*, 47(1):106–113, 2009. 3.2.2, 3.2, 3.2.2

[20] Duk-jin Kang. Web service-enabling digital video/audio-processing apparatus, and web service method and system therefor, October 7 2008. US Patent 7,433,873. 3.2.2

[21] Frank Dominguez Jr, Dave Moellenhoff, and Eric Chan. Java object cache server for databases, April 10 2012. US Patent 8,156,085. 3.2.3, 3.3

[22] Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the performance of semantic web applications with sparql query caching. In *The Semantic Web: Research and Applications*, pages 304–318. Springer, 2010. 3.2.3

[23] Jason H Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN*

conference companion on Object oriented programming systems languages and applications, pages 627–634. ACM, 2009. 3.2.4

[24] Fahad Aijaz, M Chaudhary, and Bernhard Walke. Performance comparison of a soap and rest mobile web server. In *Proc. of the Third International Conference on Open-Source Systems and Technologies (ICOSST 2009)*, 2009. 3.2.4

[25] Kishor Wagh and Ravindra Thool. A comparative study of soap vs rest web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications*, 2(5):12–16, 2012. 3.2.4

[26] Bo Yan and Guanling Chen. Appjoy: personalized mobile application discovery. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 113–126. ACM, 2011. 3.2.4

[27] Nicholas D Lane, Mashfiqui Mohammod, Mu Lin, Xiaochao Yang, Hong Lu, Shahid Ali, Afsaneh Doryab, Ethan Berke, Tanzeem Choudhury, and Andrew Campbell. Bewell: A smartphone application to monitor, model and promote wellbeing. In *5th international ICST conference on pervasive computing technologies for healthcare*, pages 23–26, 2011. 3.2.4

[28] Ei Ei Thu and Than Nwe Aung. Developing mobile application framework by using restful web service with json parser. In *Genetic and Evolutionary Computing*, pages 177–184. Springer, 2015. 3.2.4

[29] R R Mudholkar G M Tere and B T Jadhav. Improving performance of restful web services. *IOSR Journal of Computer Science*, pages 12–16, 2014. 6

[30] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995. 3, 4