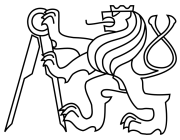
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Interactive Cloud-Based Platform for Parallelized Machine Learning of Astronomical Big Data
Student:	Bc. Jakub Koza
Supervisor:	RNDr. Petr Škoda, CSc.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

The goal is the integration of currently independent components, such as VO-Cloud, Jupyter, Spark, HDFS, and machine learning libraries in a flexible environment. The resulting system will allow the web-controlled data acquisition, job scheduling and interactive visualisation of results for a number of independent users conducting machine learning experiments on a parallel computing infrastructure.
1) Make a survey of feasible integration techniques.
2) Analyse and design the workflow to query and acquire data, to preview and clean them, to apply pre-processing and optional dimensionality reduction, to send them to cloud for machine learning, and to return the results for visualisation in Jupyter in a seamless and scriptable way.
3) Design the required missing modules and control logic.
4) Realise the platform and try to make it easily portable using virtual environment as Docker.
5) Discuss user experience and performance of your solution and suggest future improvements.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague January 28, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Interactive Cloud-Based Platform for Parallelized Machine Learning of Astronomical Big Data

Bc. Jakub Koza

Supervisor: RNDr. Petr Škoda, CSc.

9th May 2017

Acknowledgements

I would like to thank my supervisor, RNDr. Petr Škoda, CSc., for his help and for giving me this opportunity, and Jana Doležalová for long-lasting support. This research was supported by the grant COST LD-15113 of the Ministry of Education Youth and Sports of the Czech Republic.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Koza, Jakub. *Interactive Cloud-Based Platform for Parallelized Machine Learning of Astronomical Big Data.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

VO-CLOUD je distribuovaný systém, který poskytuje uživatelům prostor a výkon pro vytváření výpočetně náročných astronomických experimentů skrze rozhraní webového prostředí. Cílem této diplomové práce je navrhnout a implementovat nové komponenty a integrovat tyto komponenty do systému VO-CLOUD za účelem přidání možností vizualizace souborů astronomicých spekter, využití technologie Jupyter Notebook, která poskytuje uživatelům prostředí k interaktivnímu experimentování, a využít výpočetní klastr Hadoop společně s technologií Apache Spark.

Klíčová slova VO-CLOUD, Virtuální Observatoř, Hadoop, Spark, Jupyter, Docker, Java EE, UWS, astroinformatika

Abstract

The VO-CLOUD is a distributed system capable of providing users with a storage and computability to conduct astronomical experiments in a web based environment. The aim of this Master's thesis is to design and implement additional components and to integrate them to the VO-CLOUD system in order to add capabilities to visualise astronomical spectra files, to provide users with

the Jupyter Notebook interactive experimenting environment and to utilize the Hadoop computational cluster by using the Apache Spark technology.

Keywords VO-CLOUD, Virtual Observatory, Hadoop, Spark, Jupyter, Docker, Java EE, UWS, astroinformatics

Contents

Introduction **1**

1 Technology overview **3**
1.1 Virtual Observatory 3
1.2 Java EE 4
1.3 Universal Worker Service 10

2 Analysis of the current solution **15**
2.1 Architecture 15
2.2 Deployment 21
2.3 Workflow example 22

3 Requirements analysis **25**
3.1 New technologies 25
3.2 Functional requirements 31
3.3 Non-functional requirements 32

4 Realisation **35**
4.1 Astronomical spectra plotting capability 35
4.2 Jupyter Notebook environment integration 44
4.3 Apache Spark and HDFS integration 52
4.4 Future improvements 60

Conclusion **63**

Bibliography **65**

A Acronyms **69**

B Contents of enclosed DVD **71**

C Universal worker XML configuration file schema **73**

D Spark worker XML configuration file schema **75**

E Master server README file **77**
E.1 VO-CLOUD Master server 77

F Spark worker README file **81**
F.1 Spark worker 81

List of Figures

1.1 Fragment of simple user account Entity JPA class 5
1.2 Servlet code fragment example 7
1.3 Servlet mapping configuration 8
1.4 Relations of UWS objects 12
1.5 State machine of UWS job's execution phase 13

2.1 Universal worker configuration fragment 20
2.2 Deployment diagram 22

4.1 Example of astronomical spectrum plot of star *31 Pegasi* 36
4.2 Example of integrated `spectraviewer` application 43
4.3 JupyterHub solution Docker deployment 50
4.4 Hadoop cluster deployment on Ondřejov servers 53
4.5 Apache Avro schema JSON 56
4.6 Fragment of code serializing spectra files to the Avro format 56

List of Tables

1.1 Often used actions in UWS REST binding 12

3.1 Comparison of file size behaviour in HDFS 29

Introduction

The research of the night sky of nowadays is not only focused on data acquisition using big astronomical telescopes producing regularly big amount of data. The crucial part of the research is to actually unearth significant information inside those data. VO-CLOUD is a distributed system that has been developed to help astronomers with exactly this part. It allows astronomers to acquire data from big astronomical archives, execute preprocessing and data mining jobs on distributed workers and visualize the final results of the specific data mining method to the user. However, the problem is that currently there is no way to visualise or explore data that are already stored on the VO-CLOUD server. The visualization is critical because the astronomers should be able to review the state of spectra in the every stage of spectra processing.

The aim of this thesis is to analyse a present workflow and deployment of the VO-CLOUD server and to design a solution that would allow a user to visualize astronomical spectra inside a web browser application and explore them easily using an integrated Jupyter web application. Further, presented thesis examines ways in which the VO-CLOUD server can be extended to allow a user to enqueue jobs that use Apache Spark framework for a large-scale data processing and Hadoop Distributed File System (HDFS) as a storage for this kind of jobs. Lastly, the possibility of involvement of the Docker container platform technology is examined in order to facilitate the deployment of certain parts of the VO-CLOUD system.

Chapter 1

Technology overview

VO-CLOUD is complicated system that adopts many concepts and technologies that reader should understand before reading this Master's thesis. Also, the task of the work is to integrate additional technologies to already created system. This chapter is dedicated to the explanation of these concepts and technologies that will be used later on in the text.

1.1 Virtual Observatory

The Virtual Observatory (VO) concept is nowadays very popular among astronomy community. Whereas in the past astronomers had to wait even a couple months to access the telescope, today they can practically instantly access data they want using the concept of VO. Virtual Observatory addresses challenges such as data management, analysis, distribution and interoperability [1].

> "The VO is a system in which the vast astronomical archives and databases around the world, together with analysis tools and computational services, are linked together into an integrated facility." [1]

The VO concept and additional associated technologies and recommendations have been developed by the International Virtual Observatory Alliance (IVOA). The IVOA is an organisation with a mission to "facilitate the international coordination and collaboration necessary for the development and deployment of the tools, systems and organizational structures necessary to enable the international utilization of astronomical archives as an integrated and interoperating virtual observatory." [2]

The VO-CLOUD system is tightly connected with the concept of Virtual Observatory. It allows user to obtain data from the remote services implementing the VO principles using special VO protocols, store the data in the

provided storage, preprocess them to an appropriate format and apply a specific data mining method. All of this can be operated easily within a user's web browser.

1.2 Java EE

The whole current solution of the VO-CLOUD system is built upon Java EE Programming Language Platform (Enterprise Edition). The Java EE platform is an extension of the Java SE platform (Standard Edition) which provides the core functionality for the Java programming language. The Java EE enriches the Java SE platform with additional concepts and technologies that are mostly used in server multi-tiered environments and makes the development of Java server applications much easier.

> "The aim of the Java EE platform is to provide developers with a powerful set of APIs while shortening development time, reducing application complexity, and improving application performance." [3]

Unlike the Java SE platform where every built application can be executed directly on the Java Virtual Machine (JVM) – the environment where every Java application is running, the Java EE applications are usually deployed into an environment that supports all Java EE technologies that the application intents to utilize. This environment is called Java EE server. The Java EE serves is an application that implements APIs from the Java EE platform and provides the standard Java EE services [3]. There are many implementations of the Java EE server. The reference implementation originally started by Sun Microsystems, nowadays developed by Oracle Corporation is an open-source server called GlassFish[1]. There are many more implementations of the Java EE server, some of them are open-source other are commercial. The one that is necessary to mention here is an open-source server WildFly[2] originally developed by JBoss, now continuously developed by Red Hat. The WildFly is the server where the VO-CLOUD system is currently running on.

The Java EE specification contains many technologies that should simplify development of the server sided applications. Following sections are dedicated to the explanation of Java EE technologies that are related to the VO-CLOUD system.

1.2.1 Java Persistence API

The Java Persistence API (JPA) is a technology that considerably simplifies usability of relational databases inside Java EE applications using a principle

[1] `https://glassfish.java.net/`
[2] `http://wildfly.org/`

```
@Entity
public class UserAccount implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(unique = true, nullable = false)
    @Pattern(regexp = "[a-zA-Z]+([a-zA-Z0-9]" +
          "[._-]?)+[a-zA-Z0-9]")
    private String username;
    @NotNull
    private String passwordHash;

    public UserAccount() {}
    //getters, setters, equals, hashCode methods
}
```

Figure 1.1: Fragment of simple user account Entity JPA class

called Object-Relational Mapping (ORM).

> "The Java Persistence API (JPA) is a Java standards-based solution for persistence. Persistence uses an object relational mapping approach to bridge the gap between an object-oriented model and a relational database." [3]

A programmer implementing an application using relational database does not have to have any knowledge of Standard Query Language (SQL) – a language that is used for querying and manipulating data inside relational databases. The JPA framework does everything for him. A programmer simply implements a Java class (in terms of JPA called Entity class) and annotates it and its attributes with special Java annotations. The framework creates mapping between these objects and tables inside a relational database. Figure 1.1 demonstrates an fragment of a simple Entity class representing a user account. The class will be mapped to the table `useraccount` inside relational database containing exactly 3 columns: `id`, `username`, `passwordhash`. The JPA frameworks provides special class `EntityManager` that provides API for communication with the relational database using above mentioned Entity classes. If a programmer requires more complicated database queries he can also use the Java Persistence Query Language (JPQL) – a simple string-based language similar to SQL used to query entities and their relationships [3].

One great advantage of using the JPA framework is the fact that an application does not have to know any information about database itself. The application only specifies the name of so-called Persistence Unit. The Persis-

tence Unit is configured on the Java EE server and the configuration consists of items such as database connection URL, login credentials, query timeouts, connection drivers and many others. The principle of pulling configuration from application to server ensures portability of the application. In fact, application on one server can use for example PostgreSQL[3] relational database and it can also be redeployed without recompilation to the server utilizing MySQL[4] relational database.

1.2.2 Java Servlet Technology

The Java Servlet Technology is very important in the Java EE specification because many other technologies are built upon it.

> "A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes." [3]

Servlets have very simple lifecycle. The lifecycle is controlled by the web container of the Java EE server where the servlet has been deployed. When a request is mapped to a servlet, the container performs following steps on order to serve a response. [3]

1. If container does not contain an instance of the servlet, the container:
   a) loads the servlet class if it has not been done already,
   b) creates an instance of the servlet class,
   c) calls servlet's method `init` to perform servlet initialization.
2. Calls `service` method of the servlet instance with two method parameters representing servlet request object and servlet response object.

Container can also decide that a servlet instance is no longer necessary and remove it from the container. Before it does so it finalizes servlet by calling method `destroy`.

From the implementation point of view the Java Servlet Technology is implemented in packages `javax.servlet` and `javax.servlet.http`. The first package contains interface `Servlet` that every servlet class must implement. The most important methods of this interface are aforementioned methods `init`, `service` and `destroy` [4]. The first package also contains one of the

[3] `https://www.postgresql.org/`
[4] `https://www.mysql.com/`

```
@WebServlet("/hello-world")
public class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public void doGet (HttpServletRequest req,
                       HttpServletResponse res)
    throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        out.println("Hello, world!");
        out.close();
    }
}
```

Figure 1.2: Servlet code fragment example

class implementing `Servlet` interface – `GenericServlet`. This class can be used to implement a generic service – protocol independent servlet.

The most important subclass of `GenericServlet` is `HttpServlet` from package `javax.servlet.http` that provides an abstract implementation of the HTTP protocol [4]. Method `service` in implementation of this class delegates requests to one of the method `doXXX` where `XXX` is one of the methods of HTTP protocol (`GET`, `HEAD`, `OPTIONS`, `POST`, `PUT`, `TRACE` or `DELETE`). Figure 1.2 demonstrates an example of a simple "Hello, World!" servlet application. The application returns string "Hello, world!" whenever HTTP `GET` method is called on the servlet's endpoint (for example when web browser connect to the servlet's endpoint URL).

It is important to understand what the servlet's endpoint actually is and how to specify it. As it is possible to see in the example 1.2, the class `HelloServlet` is annotated with `WebServlet` annotation. The value of this annotation specifies a path relative to the path of deployed Java EE application. For instance, if the servlet application is deployed on the URL address `http://example.org/app`, the servlet endpoint from the example 1.2 is `http://example.org/app/hello-world`. The `WebServlet` annotation can be also replaced by utilizing a configuration file `web.xml`, where deployment configurations are specified in XML format. An example of such a configuration of `HelloServlet` from the example 1.2 can be seen in the figure 1.3.

1.2.3 JavaServer Faces

JavaServer Faces (JSF) is an important technology of the Java EE platform that focuses on simplification of web user interface development. It is built upon Java Servlet technology. In contrast to Java Servlet technology where

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" ... >
<servlet>
  <servlet-name>Hello Servlet</servlet-name>
  <servlet-class>servlet.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Hello Servlet</servlet-name>
  <url-pattern>/hello-world</url-pattern>
</servlet-mapping>
</web-app>
```

Figure 1.3: Servlet mapping configuration

servlet class contains implementation of both presentation and behavioral part of user interface, JavaServer Faces framework splits these parts to different units.

> "One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behaviour and presentation for web applications." [3]

The implementation of web user interface of the Java EE application using JavaServer Faces framework consists of two different types of files: XHTML files and so-called Managed Beans. XHTML files represent a presentation part of the user interface – visual side of one page in standardized XML format [3].

There are two different types of XML tags that can be used inside XHTML format. Standard HTML tags and special JSF tags. Whereas HTML tags have no special meaning for the JSF framework and they are mostly passed directly to the client's web browser, the JSF tags add additional functionality beyond the static HTML pages. They allow to bind data changes, actions and events of the page to Java methods specified in Managed Beans using a special syntax called Expression Language [5]. JSF tags can represent any web view component from a single text field to complicated data table offering sorting and filtering functionality. For example following fragment of XHTML code represents a simple input text field of the HTML input form:

```
<h:inputText value="#{foo.username}" />
```

The `#{...}` syntax is in fact the Expression Language that binds value of this input field to the `username` field of the Managed Bean named `foo`. The binding has two functionalities:

- When a page is being rendered for a client the value of the input field is set to the value of `username` field in `foo` Managed Bean.

- When the input form is filled and submitted back to a server by client the new value of this input field is stored in the Managed Bean.

The set of JSF tags is easily extendable by using additional XML namespaces in the root tag of the XHTML document [6]. Using this principle, one can use additional extended components that are not available in pure JSF framework. VO-CLOUD system uses one of such popular JSF extensions – the open source framework named PrimeFaces[5].

Managed Bean by definition [3] is a simple Java class that must fulfil following rules:

- It must have non-parametric constructor in order to be able to instantiate it anytime without parameters.
- It must have defined name that is available for Expression Language. Managed beans have usually default name inferred from the name of Java class, however it can be renamed using XML descriptor files or Java annotations.
- It must define a scope.

Choosing the right scope for every Managed Bean is an important part of application design. "Scope defines how application data persists and is shared."[3] The two most important scopes are request and session scopes. Data inside Managed Beans annotated with request scopes survive only for a single client's HTTP request, whereas data from session scoped beans are saved to a session storage of the specific client and they survive multiple HTTP requests. The drawback of the session scoped approach is the fact that session storage consumes memory of a server and it complicates scalability of the application because it requires replication of the session storage to additional server instances.

1.2.4 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a powerful technology that is also part of the Java EE specification. Enterprise bean is a component that runs inside EJB container, a runtime environment in the Java EE server [3]. It is important to note that not every Java EE server has implemented the EJB container and so application written with the EJB functionality cannot be deployed to such servers. For example Apache Tomcat[6] server supports many Java EE technologies, however it is only servlet-based server and so there is no support for EJB components. Part of the VO-CLOUD system uses EJB thus it is necessary to deploy them to EJB enabled server such as earlier mentioned GlassFish or WildFly server.

[5] `https://www.primefaces.org`
[6] `http://tomcat.apache.org`

The purpose of EJB components is to encapsulate the business logic of an application and to make an development of large, distributed application somewhat easier [3]. The EJB framework pulls responsibilities like security, concurrency and transaction management from a developer to the EJB container itself. It also provides special API, like asynchronous method invocation and timer service that allows to schedule an operation execution or to execute periodically in specified time intervals.

1.3 Universal Worker Service

The Universal Worker Service (UWS) pattern is an important concept that is involved in the VO-CLOUD system. The UWS recommendation has been developed by IVOA organization and it is extensively used in astroinformatics in cases where synchronous and stateless services are not an option. First, let's explain what these terms actually mean.

The majority of simple web services are synchronous and stateless. Synchronous service is such a service where client waits for the service response after sending a request. Stateless service means that service does not have to remember any state of a communication with client. The state is stored on the client's side and every client's request contains all information necessary for processing a response. This is one of basic principles of so-called RESTful services where the REST (Representational State Transfer) is an architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI) [7].

There are two reason why synchronous and stateless services are sometimes not sufficient:

- Processing of response from the passed request can take very long time, sometimes even days.
- Service parameters or results can take large amount of memory space and transferring them through a data channel is not achievable in reasonable time.

It is often necessary to work with big data and long running tasks in astroinformatics. UWS pattern has been developed as a way of solving aforementioned problems.

The UWS recommendation specifies how to build asynchronous, stateful and job oriented services [8]. Whereas in synchronous services responses are expected to be almost instantly processed from requests and passed back to a client, clients using asynchronous services of UWS pattern only get identifier of newly created job. Issued jobs are being executed on a server without necessity of client's interaction or even a connection. After the execution of a

job is done, results are being linked with the job's identifier. Clients use job's identifier to query phase of execution of the job, to download results and to manipulate with the job itself. They can for example abort already running job if its results are no longer relevant.

"A UWS consists logically of a set of objects that may be read and written to in order to control jobs."[8] These objects are addressable as a distinct web resources and each object has its own URI – it uses the same principle of binding as in RESTful services. Relations between individual UWS objects can be seen in figure 1.4.

- *Job* – represents a single executable job
- *Job List* – top-level resource collection; every job must be inside one of these collections
- *Phase* – represents the execution phase of a job
- *RunID* – a unique identifier of a job inside a job list collection
- *Owner* – an identifier representing the job's owner
- *Execution Duration* – maximal duration of a job execution in seconds; service provider aborts the job if the duration is exceeded
- *Destruction Time* – absolute time when the job and its results should be removed from a job list collection
- *Quote* – a UWS service prediction when the job is likely to complete
- *Error* – human-readable message specifying the reason why a job failed
- *Parameter List* – list of parameters passed to a UWS service
- *Result List* – list of Result objects
- *Results* – an object representing one of the results of a job execution

A UWS pattern uses the same guidelines for mapping operations over resources to the HTTP methods as in a RESTful web services – HTTP `GET` method is mapped to operation read, `POST` to operation create, `DELETE` to operation delete and `PUT` to operation update. The set of operations create, read, update and delete are often shortened to an abbreviation CRUD. Table 1.1 shows the most important operations over a UWS service API.

It is also important to mention in more detail a UWS object *Phase* that specifies the current execution phase of a respective job. "The job is treated as a state machine with the Execution Phase naming the state." [8] Phases' names are self-explanatory and their state machine diagram can be seen in the figure 1.5.

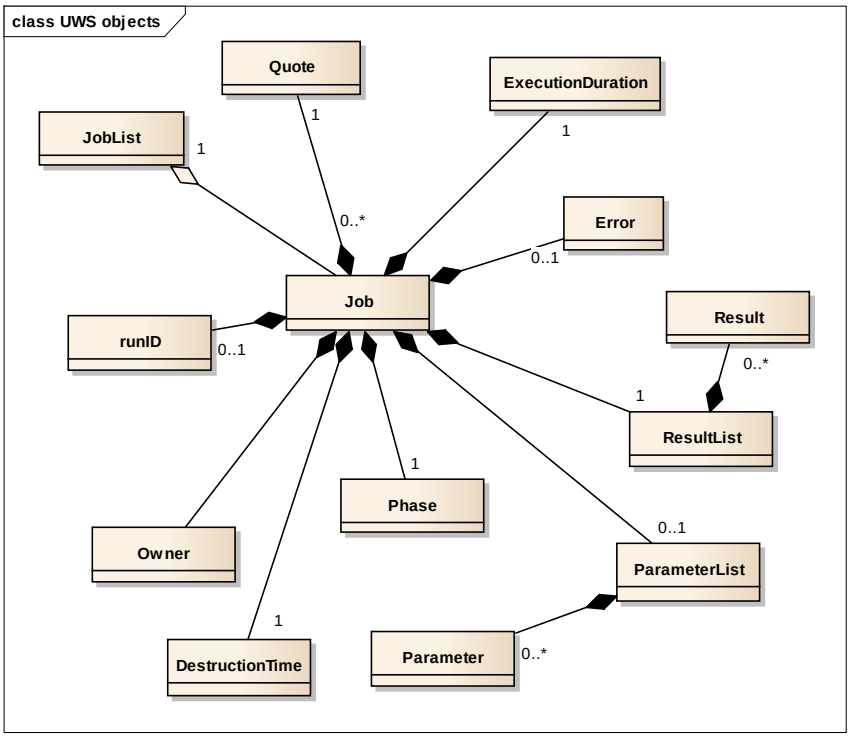


Figure 1.4: Relations of UWS objects [5]

Table 1.1: Often used actions in UWS REST binding [5]

Method	**URI**	**Description**
GET	/{jobList}	listing of all Jobs
GET	/{jobList}/{id}	summary of specified Job
GET	/{jobList}/{id}/phase	phase of the specified Job
GET	/{jobList}/{id}/results	results of the specified Job
POST	/{jobList}	creates new Job
POST	/{jobList}?PHASE=RUN	creates new Job and puts it into execution queue
POST	/{jobList}/{id}/phase ?PHASE=RUN	puts already created Job into execution queue
POST	/{jobList}/{id}/phase ?PHASE=ABORT	aborts specified Job
DELETE	/{jobList}/{id}	deletes specified Job

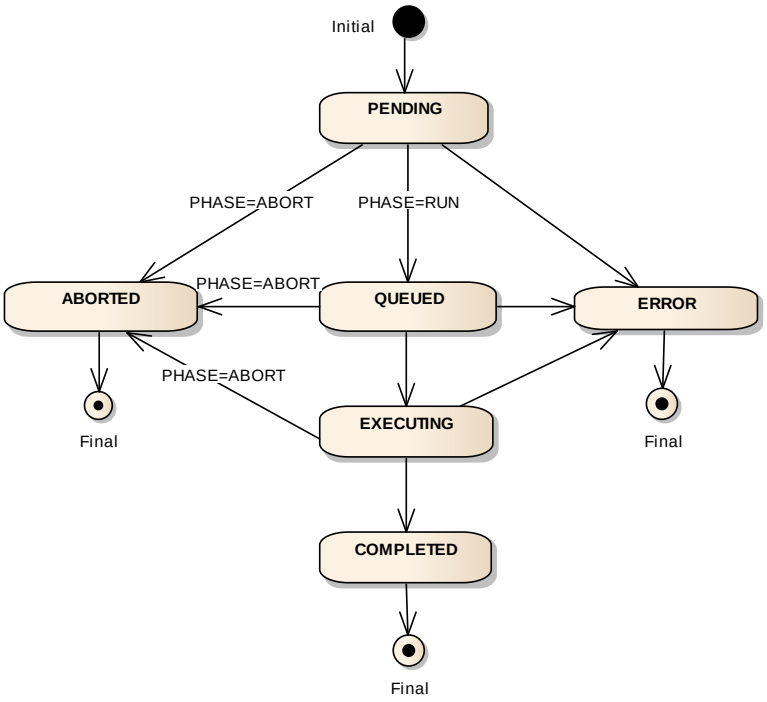


Figure 1.5: State machine of UWS job's execution phase [5]

Chapter 2

Analysis of the current solution

The understanding of the current solution of VO-CLOUD system is crucial for the task of the thesis. This chapter is dedicated to a description of VO-CLOUD system's responsibilities, explanation of the system's architecture, description of the system's workflow and the current state of deployment on the servers. First, let's describe architecture of the VO-CLOUD system.

2.1 Architecture

VO-CLOUD is a distributed system, which means that it is comprised of hardware or software components located at networked computers that communicate and coordinate their actions only by passing messages to achieve their task [9]. The system consists of three main components:

- Master server – The main component of the distributed system. Web application that an experimenting user communicates with.
- Universal worker – The computational component. Master server delegates the computational tasks conducted by an experimenting user to these components. Provides web service for communication with the Master server.
- Specific preprocessing or data mining application – An application that takes passed data and creates an output requested by the user. This application is called as a process from the Universal worker.

It is important to fully understand each of these components and therefore following sections are dedicated to fully explain their technologies and responsibilities.

2.1.1 Master server

Master server is a web application completely written in the Java EE platform.

Responsibilities of the Master server are following:

1. Provide web GUI for communicating with the experimenting user's web browser application.
2. Provide storage where users can save their data and use them for further experiments and provide web interface to manage the storage.
3. Allow users to upload new files to the Master server's storage directly from the user's device.
4. Allow users to download new files to the Master server's storage from the passed HTTP or FTP resource URL.
5. Allow users to download new files from spectra archives using special astronomical protocols SSAP and DataLink.
6. On the user's request enqueue new computational job to the Universal worker, await job's completion, download results back and present them to the user.
7. Allow user to abort currently running job.

Whole web user interface is written using JavaServer Faces (JSF) technology. Master server also requires database for persisting multiple pieces of information, for example user accounts and or executional jobs that have been created by individual users. VO-CLOUD uses Java Persistence API (JPA) framework for utilizing the persistence storage. Moreover, for simplification of transaction and security management VO-CLOUD Master server utilizes Enterprise JavaBeans technology. This means that the Master server has to be deployed on a Java EE server containing EJB container and thus supporting EJB technology.

Users communicate only with the user interface of the Master server. Every user that wants to work with VO-CLOUD system must be authenticated and thus one of the Master server's additional responsibilities is to offer a registration form to newcomer users. When user logs in the system, he can do set of operations depending on his authorization level – user role. VO-CLOUD distinguishes between three following user roles:

- *USER* – User with this role has read-only access to the VO-CLOUD's storage and can create new jobs from the set of non-restricted job types.
- *MANAGER* – User with this role has in addition to *USER* role write permissions to the system's storage and he can also create jobs of a restricted type.

- *ADMIN* – User with this role has in addition to *MANAGER* permissions to view jobs of all users in the system, to change users' settings (e.g. set new password or change user role) and to change configuration of available Universal workers.

The storage of VO-CLOUD system is directly mapped to the filesystem where VO-CLOUD has been deployed – it has tree structure of files and folders. There are in total five ways to get data to the system's storage:

1. A user can directly upload files from his local device through VO-CLOUD's web user interface.

2. A user can command the server to download file/files from remote locations using FTP or HTTP protocol. By using this method server can download multiple files if passed location points to folder in FTP server or to directory listing of HTTP protocol.

3. A user can command the server to download astronomical spectra from VO databases using protocols SSAP and DataLink. These two protocols have been developed as IVOA recommendations. SSAP is basically a protocol that allows to query astronomical spectra fulfilling specified filter conditions and it returns a list of spectra together with metadata [10]. These spectra could be either directly downloaded, or, if VO service supports it, the DataLink protocol can be used to apply additional spectra transformations on the service provider's side before a download [11].

4. A user can command the server to store an output of any computational job to the system's storage.

5. VO-CLOUD's storage can be also modified by directly modifying a folder structure on the side of server where the Master server has been deployed (e.g. by connecting to the server directly through SSH protocol and modifying data using terminal commands).

Master server's user interface also offers operations to download selected files from server's storage to user's device, to delete selected files, to create new folders, to rename files, etc. Note that operations changing the storage's state can be issued only by users that have user role *ADMIN* or *MANAGER*.

Master server provides functionality for user to create a new computational job. Every job is represented by a job type and a configuration in JSON data format. Job type is in fact the choice of a specific preprocessing or data mining application. Job types are divided to two categories:

- *Non-restricted jobs* – Jobs that can be created and executed by any logged in users.

- *Restricted jobs* – Jobs that can be created and executed only by users with higher permissions because to fully utilize their potential it is necessary to have write permissions to the system's storage.

There are currently three types of jobs that can be executed on the VO-CLOUD system. Preprocessing and Random Decision Forest (RDF) method that has been implemented by Andrej Palička in his Bachelor's thesis [12] and Self-Orginizing Maps (SOM) method that has been developed by Lukáš Lopatovský in his Bachelor's thesis [13]. Preprocessing job type takes spectra stored in the system's storage and it preprocesses them to the format that is an input of RDF and SOM job types. Due to necessity to store data from preprocessing job type back to the VO-CLOUD's storage the preprocessing job type is set as a restricted job. RDF and SOM job types are non-restricted.

In fact, multiple workers supporting a single specific job type can be configured. Master server selects from the list one worker that is the least loaded and delegates the computational job on it through UWS worker's interface. After the execution of a job has started, the Master server periodically checks worker's job phase through UWS API and when the execution stops, the Master server downloads results from the worker and it commands worker to delete results on its side. User can view the execution phase of every created job. Jobs' phases are directly mapped to phases of UWS pattern (see figure 1.5).

2.1.2 Universal worker

Computational jobs are not executed by the Master server itself but they are delegated to computational components of the distributed system – generic workers. Worker has following responsibilities in the VO-CLOUD system:

1. It provides UWS service that the Master server communicates with.
2. It parses JSON configuration for every new job and downloads all necessary files from the Master server that are listed in the configuration and are therefore necessary for computation.
3. It passes JSON configuration together with downloaded files to a specific preprocessing or data mining application.
4. It stores computed results until the Master server downloads them.

The worker is a relatively simple web application written in the Java EE platform. As it has been already stated in its responsibilities it must expose a UWS interface that the Master server communicates with. A JavaServlet Techology is involved in the UWS service implementation. No additional Java EE technologies (especially EJB) are used in the worker's implementation and thus it can be deployed on lightweight Java EE server that has no EJB container (e.g. Tomcat).

In the original implementation of VO-CLOUD system (originally named VO-KOREL [14]) there had to be an implemented application for each individual type of workers. Every new preprocessing or data mining method required also a new worker application implementation. Moreover, every server where such workers have been deployed can have different settings, i.e., the path to the preprocessing or data mining application could be different. It was necessary to build an individual package for each server and worker type. Source codes of these applications were almost identical with the exception of a few lines of codes and configuration strings. This approach was detrimental for maintenance as every minor change in the source code required multiple recompilations and deployments.

This approach was changed as the result of Jakub Koza's Bachelor's thesis that brings a new concept called Universal worker.

> "A universal worker is a new type of the servlet based application that is used instead of all other worker application types. The idea is to deploy only one instance of universal worker application on one computer worker node where multiple computational executable applications are supported."[5]

Universal worker is configured using a XML configuration file that matches XSD schema specially created for a Universal worker concept [5]. The schema can be seen in appendix C. Universal worker uses multiple job list collections instead of only one – one job list per one `worker` XML tag configured in the XML configuration file. The fragment of a such worker's configuration can be seen in figure 2.1.

As can be seen in the example 2.1 the most important part of worker's configuration is actually a specification of a process call. Whenever some job should be started on the Universal worker, the worker actually creates a new working directory. All files that are necessary for a job execution are downloaded to this directory. Also the JSON configuration that was passed as a job parameter is saved into this directory as a file. Finally, the process specified in the XML configuration is executed in this directory and a path to the configuration JSON file is passed as a parameter to this process (this is caused by the last `command` tag in XML configuration containing special substitution sequence `${config-file}`.

Workers have been specifically designed to constitute the distributed part of the VO-CLOUD system. There can be multiple workers on multiple machines. If there are more workers for a single job type, VO-CLOUD system automatically chooses the one that is the least loaded. In the matter of deployment it is essential that VO-CLOUD's Master server has network visibility to individual workers in order to communicate with their UWS interface. However, workers do not have to be exposed to users' devices at all. If it is expected that workers should be able to download data from VO-CLOUD's Master server the visibility must be bidirectional.

```
...
<ns:worker>
  <ns:identifier>preprocessing</ns:identifier>
  <ns:description>Preprocessing</ns:description>
  <ns:restricted>true</ns:restricted>
  <ns:binaries-location>/usr/local/
    workers/preprocessing</ns:binaries-location>
  <ns:exec-command>
    <ns:command>python3</ns:command>
    <ns:command>${binaries-location}/
      run_preprocessing.py</ns:command>
    <ns:command>${config-file}</ns:command>
  </ns:exec-command>
</ns:worker>
...
```

Figure 2.1: Universal worker configuration fragment

2.1.3 Specific preprocessing or data mining application

VO-CLOUD's Universal worker component would be useless without an application that is capable of preprocessing passed data or unearthing new relevant information from them. As it has been already stated there are currently three of these applications – Preprocessing, SOM and RDF. All of them are written in Python programming language and their behaviour can be altered by changing an input JSON configuration. These applications are simply called as a new process from the Universal worker component. Despite Python being used as a technology for all three job types, there is no limitation on technology used, i.e., any process that can be executed on worker's hosted system can be used for purposes of the Universal worker component.

Universal worker redirects standard output stream and standard error stream of application's process to its own temporal files that are afterwards passed to the Master server together with results. Also the exit status code of the process is passed back to the Master server. If the status code is equal to zero, the process is considered to be successfully ended and the job's phase is set to `COMPLETED` state. Otherwise, job's phase is set to `ERROR` state. A user can go through standard output and error files on the Master server to uncover a reason for the process failure.

Applications executed by the Universal worker can also create an visualization output that the Master server can present to the user. It was designed this way because every job type can require different type of visualization. Visualization is optional and there are two types of visualization that the Master server can utilize:

- *Static visualization* – Process produces static image/images that are placed directly to the working directory. These images are directly presented to a user in a web interface. Master server supports following image formats: `PNG`, `JPEG`, `GIF`.

- *Dynamic visualization* – Process produces a simple web application. In order to do so the process must produce `index.html` file as a starting point of the web application that must be placed directly in the working directory. Master server renders content of this file inside a special HTML tag `IFRAME` that basically allows to run another web page inside a web page. Process can also produce additional HTML files and link those files via standard hypertext relative links. It can also contain a JavaScript code for additional scripting capabilities. By using this approach the computational application can for example render a complicated clickable visualization with spectra rendering capabilities.

2.2 Deployment

In order to be able to continue with this Master's thesis it is important to explain the deployment of the current solution as this is the state that is going to be extended. VO-CLOUD system is currently deployed on two servers at Stellar Department of the Astronomical Institute of the Czech Academy of Sciences in Ondřejov. These servers are named `vocloud-dev` and `betelgeuse`. Whereas `vocloud-dev` is only a virtual server with relatively small amount of computational resources, `betelgeuse` is a powerful physical server with 12 CPU cores supporting Hyper-Threading technology (24 virtual CPU cores) and 128 GiB RAM memory. However, unlike the `vocloud-dev` server, the `betelgeuse` server is especially for security reasons not available publicly. Therefore, there is a reverse proxy server Nginx[7] deployed on the `vocloud-dev` server. The reverse proxy simply forwards all incoming HTTP/HTTPS requests starting with URI `/vocloud-betelgeuse` to the Java EE server hosted on the `betelgeuse` server. It also redirect URI `/` to the previous one, therefore the VO-CLOUD system is available on URL address `https://vocloud-dev.asu.cas.cz`.

Whole VO-CLOUD system is currently deployed on the `betelgeuse` server on the Java EE server called WildFly[8]. There is a single Master server instance and a single Universal worker instance deployed on the WildFly server. The Universal Worker is configured so that it provides a job execution service for Preprocessing, RDF and SOM job types. Master server requires a relational database for its functionality. There is a PostgreSQL[9] database that is

[7] `http://nginx.org`
[8] `http://wildfly.org`
[9] `https://www.postgresql.org`

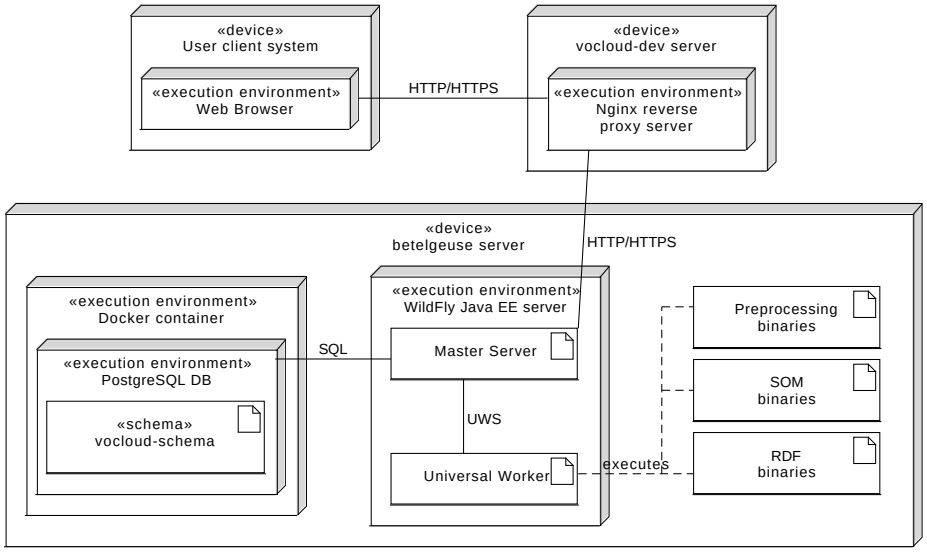


Figure 2.2: Deployment diagram

deployed also on the `betelgeuse` server inside a Docker container. Docker[10] is very powerful technology that considerably simplifies deployment of application components on different machines. This technology is explained in detail in the following chapter because it is a crucial technology for for the purposes of this work.

Deployment diagram of the current solution can be seen in the figure 2.2.

2.3 Workflow example

For the sake of completeness, let's describe a scenario of user's communication with the VO-CLOUD system. A user in this scenario needs to download data from a VO archive, to apply preprocessing on them and to apply Self-Organazing Maps (SOM) method to find similarities between passed astronomical spectra [13]. As the user intents to use the restricted job type – *Preprocessing* – he must have either a `MANAGER` user role or `ADMIN` user role.

1. User logs into the VO-CLOUD system using his username and password.

2. User navigates through VO-CLOUD's storage tree structure and selects/creates a directory where spectra from VO archive should be downloaded.

[10] `https://www.docker.com`

3. User clicks *Append new files using SSAP* button, he fills in all necessary input parameters and he configures a DataLink protocol settings (if DataLink protocol is supported by the VO archive and if user wants to use it).

4. User commits the download request. Progress of this request and possible errors can be seen on a dedicated page.

5. When the download is completed, user continues on page of a Preprocessing job creation. He either creates a JSON configuration file from scratch or he selects one of the pre-created configurations. Files that should be preprocessed and preprocessing parameters are specified in the JSON configuration.

6. User starts a preprocessing job. Progress can be seen on a dedicated page.

7. After the preprocessing job is completed, user opens details of the job and checks that there is an expected output.

8. User then stores an output of the preprocessing job to system's storage in order to have it as a source for additional experiments.

9. In a similar way to the preprocessing job, user creates a new SOM job and as an input he selects an output of the preprocessing job stored in the system's storage.

10. When the SOM execution job is completed, user opens details of the job. SOM job type provides an interactive visualization that user can deeply explore and it can help him to unearth new interesting information.

Chapter 3

Requirements analysis

This chapter is dedicated to the description of all functional and non-functional requirements that are demanded by the new version of the VO-CLOUD system. The fulfilment of these requirements is the aim of this Master's thesis. Before diving into these requirements it is important to explain all new technologies that are going to be involved in the new version of the system.

3.1 New technologies

The basis of this work is in fact integration of many new technologies to an existing solution of the VO-CLOUD system in order to simplify scientific work with the system and to extend its capabilities. The comprehension of these technologies is crucial for the practical part of this thesis, thus this section should introduce these technologies to the reader.

3.1.1 Docker

One of the most important technologies that was used in the practical part is Docker. Docker technology has been already mentioned in the section explaining VO-CLOUD's deployment 2.2 because the database server of the currently deployed solution uses a Docker container. Docker is a software container platform where piece of software is packaged into isolated containers [15]. The containers are functionally very similar to the virtual machines, however, containers in Docker platform do not bundle a full operating system, but only software libraries and settings required to make the software work as needed [15].

> "Docker containers running on a single machine share that machine's operating system kernel; they start instantly and use less compute and RAM. Images are constructed from filesystem lay-

> ers and share common files. This minimizes disk usage and image downloads are much faster."[16]

Docker containers are isolated from each other and they are even isolated from the system hosting the Docker containers. This approach offers great security benefits. Nevertheless, it is possible to link Docker containers together using a Docker linking functionality, if it is necessary. For example, it is a standard approach to deploy a database and an application server on two different Docker containers and link these two containers together.

There are two important terms in Docker terminology that must be explained:

- *Docker image* – Docker images are basically immutable snapshots of a Docker container. They have layered structure comprised of commits that were applied on the base image. Docker images are created by calling a `build` Docker command that takes a special `Dockerfile` file containing instructions that are gradually processed to produce the final Docker image. It is also possible to extend an existing image to create a new image with added functionality. Built Docker images can be pushed to remote repositories and thereafter they can be easily pulled on every device that requires them.

- *Docker container* – Docker containers are instances of Docker images. Every Docker image can be started using a Docker command `run` and this is the way to create a new Docker container. Multiple Docker containers can be created from a single Docker image and multiple Docker containers can be running on a single host machine.

For illustration, let's demonstrate an example of fast PostgreSQL database deployment using a Docker technology by using a single terminal command:

```
docker run --name db -d -p 5432:5432
           -e POSTGRES_USER=<username>
           -e POSTGRES_PASSWORD=<password>
           postgres
```

This is how a database server is currently deployed on the `betelgeuse` server. The command does following actions:

1. It pulls `postgres` Docker image from the public Docker repository[11] if it is not pulled already in the hosted system.

2. It creates a new Docker container named `db`. Database in the container has specified `username` and `password`.

[11] `https://hub.docker.com`

3. It exposes TCP port 5432 from the container to port 5432 on the hosted system.

4. It starts a newly created container in the background and it prints unique container's identifier.

The most time consuming operation is pulling of Docker image from a repository. If the image is already pulled on the hosted system, the Docker `run` command is processed almost instantly.

3.1.2 Apache Hadoop

The Apache Hadoop is a an open-source project developed by the Apache Software Foundation. It is a framework specially developed for the distributed processing of large data sets across clusters of computers using a single programming model. The Apache Hadoop infrastructure can scale up from the single machine to thousands of servers, each offering its computational and storage resources [17]. The Apache Hadoop consists of following modules[17]:

- *Hadoop Common* – The common utilities that are necessary for other Hadoop modules.
- *Hadoop Distributed File System (HDFS)* – Specially designed distributed file system that provides high-throughput access to stored data. This module is explained in more detail in the next section 3.1.3.
- *Hadoop YARN* – A framework for managing cluster resources and scheduling computational jobs.
- *Hadoop MapReduce* – A system of parallel processing of large data sets. Even though MapReduce is very important concept in the Apache Hadoop infrastructure, this technology is not involved in the practical part of this thesis and it has been substituted for the Apache Spark compute engine. The Apache Spark is explained in section 3.1.4.

It is important to note that the whole Apache Hadoop framework is built on the Java technology.

3.1.3 Hadoop Distributed File System

As it has been already stated, Hadoop Distributed File System (HDFS) is one of the core modules of the Apache Hadoop framework infrastructure. It is a distributed filesystem very similar to other distributed file systems, however, HDFS is highly fault-tolerant and it was specifically designed to be deployed on low-cost hardware [18].

HDFS has a master/slave architecture and it consists of two types of components:

- *Name Node* – Master component that manages the file system namespace and regulates access to files by clients [18]. There is only one Name Node component on the HDFS cluster.
- *Data Node* – The slave component that manages storage attached to the nodes that they run on [18]. There is usually one Data Node component per each device connected to the HDFS cluster.

Data in HDFS are stored inside blocks. The size of blocks depends on the HDFS settings, however, it is usually 64 MB or 128 MB. Data Nodes have no knowledge about files stored inside HDFS [18]. They manage only their file blocks – each block is stored as a separate file in a local filesystem of Data Node's device.

An image of entire HDFS filesystem is kept by the Name Node inside its memory. Name Node also decides where each file's data block should be stored (on which Data Node) and also where other replicas of the same data block should be stored. Replication factor (a number stating how many Data Nodes should carry a given data block) can be set on every HDFS file differently. Replication is an essential feature that ensures fault-tolerance functionality and it also improves computational times on some jobs as some data blocks do not have to be moved from one Data Node to another.

Despite the fact that the HDFS has been designed for big data processing, it is important to point out that it was designed for smaller amount of big files than for many small files. According to Tom White [19], every file, directory and block in HDFS is represented by an object inside Name Node's memory that takes approximately 150 bytes. For illustration, let's demonstrate an example of storing in total 10 TB of data. In the first scenario data are distributed in 100 kB files, whereas in the second scenario data are distributed in 1 GB files. Let's consider that HDFS uses 128 MB sized blocks. As can be seen in table 3.1, the first scenario requires 15 GB of Name Node's memory, whereas the second scenario only 12 MB. Furthermore, HDFS is primarily designed for streaming access of large files and not of small files, because it causes lots of seeks and lots of hopping from Data Node to Data Node to retrieve each small file [19]. The problem with small files is very urgent in this case as data here usually consist of astronomical spectra – each in a separate approximately 50 kB sized file.

3.1.4 Apache Spark

Apache Spark is an open-source big data general-purpose cluster computing system written in Scala programming language, that provides high-level APIs in Java, Python, Scala and R programming languages and it also provides an optimized engine that supports general execution graphs [20]. Apache Spark software comes also with a rich set of higher-level tools like special Spark's Machine Learning Library (MLlib), which consists of common learning algorithms

Table 3.1: Comparison of file size behaviour in HDFS

Total data size	10 TB	10 TB
Block size	128 MB	128 MB
Size of each file	100 kB	1000 MB
Files count	100 000 000	10 000
Blocks per file	1	8
Blocks count	100 000 000	80 000
Total memory	15 GB	12 MB

and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs [21].

The great advantage of Apache Spark software is that it can utilize an existing Hadoop infrastructure. Spark jobs can be executed in a standalone mode on a single device or in a simple cluster created specially for the purposes of Spark using a tools bundled in the Apache Spark binaries. However, it can also enqueue a Spark job using already mentioned module of Hadoop cluster – Hadoop YARN – that manages cluster resources and schedules computational jobs. The reason why this is usually a better solution for big clusters is that Hadoop YARN is usually precisely configured regarding available resources on each cluster's node. Moreover, if multiple jobs are enqueued for execution on the Hadoop YARN, the scheduler rather delays an execution of later jobs to offer the executing job full cluster capabilities. Spark framework can also fully utilize the Hadoop Distributed File System (HDFS) as a source of input data sets or as an output job storage.

It is important to explain the main difference between the Hadoop MapReduce approach and Spark approach. MapReduce has been designed specifically for one-pass computation – it has one Map phase and one Reduce phase. However, many algorithms require multiple-pass computations. If they would be converted to the MapReduce pattern they would require multiple MapReduce jobs to be executed – output of the preceding one would be an input of the successive one. All these intermediate outputs have to be stored in the distributed file system before the next step begins, hence, this approach tends to be slow due to replication and disk storage [22]. Spark on the other hand allows programmers to hold results in memory instead of writing them to disk, especially when they need to work on the same dataset multiple times [22]. If data do not fit into the memory, Spark framework automatically stores part of the data to disk. Since Spark prefers using a fast RAM memory to using a slow disk, jobs executed in Spark framework are usually many times faster than by using a Hadoop MapReduce.

3.1.5 Jupyter Notebook

The Jupyter Notebook is a web-based application that extends console based approach to interactive computing in a qualitatively new direction [23]. In the past it was common to use some console-based application for an interactive experimenting over a set of data. However, this approach had many disadvantages that the Jupyter Notebook application is trying to improve.

The Jupyter Notebook application implemented in Python programming language runs as a server on the machine where it was started by using a single command:

```
jupyter notebook
```

User can connect to it using his favourite web browser application. The directory where this command has been invoked is important because it is set as a working directory for the Jupyter Notebook application. The application consists of three following components:

- `Notebook Dashboard` – It is basically a file browser that lets user go through the working directory, create, view, modify, delete files, start Notebook documents and inspect already running Notebook documents and Kernels.

- `Notebook document` – Notebook document is the most important component of the Jupyter Notebook applications. "Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc. . .) as well as executable documents which can be run to perform data analysis."[24]

- `Kernel` – The source code from a Notebook document is executed in the component called Kernel. It is basically the computational engine of the Jupyter Notebook application. Multiple Kernels can be configured in the single Jupyter Notebook application, each can support different programming language. For example, one Kernel could execute source code from Notebook documents as a Python code, second as an R code, third as a Scala code, etc. . .

One of the great advantages of the Jupyter Notebook application is the fact that the server can be started on one device and a client can connect to it from another device. All computations take place on the side of the server inside a Kernel component, thus a client can induce computationally challenging experiments on the server side using only a web browser on his potentially slow device and see results either in simple text format or in more sophisticated output, e.g., plots or tables.

3.2 Functional requirements

This section lists all new functional requirements that are demanded by the new version of the VO-CLOUD system.

FR 1 Master server must provide an ability to plot selected astronomical spectrum file saved in its storage.

FR 2 Master server must be able to plot multiple spectra files in the same plot.

FR 3 Master server must be able to plot large `csv` files containing multiple spectra.

FR 4 Master server must provide a zooming and panning functionality in the plotted spectra.

FR 5 Plotted spectra must be able to be exported into `ps`, `eps`,`pdf`, `png`, `raw` and `svg` image format.

FR 6 Plotter must be able to load `x` axis values from an external `meta.xml` file if a spectrum file does not contain it itself.

FR 7 Plotter must be able to plot following spectra file formats: `fit`, `fits`, `vot`, `csv`.

FR 8 Master server must provide an ability to switch to Jupyter Notebook environment.

FR 9 Files from the Master server's storage must be visible and available for experimenting from the integrated Jupyter Notebook environment.

FR 10 Files from completed jobs in the Master server must be visible and available for experimenting from the integrated Jupyter Notebook environment.

FR 11 The Jupyter Notebook must support following Kernel types:

- Python 2
- Python 3

FR 12 Job and storage files from the Master server available to Jupyter Notebook must be read only.

FR 13 The Jupyter Notebook environment must be available only to users authenticated in the VO-CLOUD system.

FR 14 The Jupyter Notebook environment must be available only to users with a user role either `MANAGER` or `ADMIN`.

FR 15 The Jupyter Notebook environment must provide a writeable directory where users can create their own files and Jupyter Notebooks.

FR 16 Every user's Jupyter Notebook writable directory must be isolated from all other users.

FR 17 Users should not authenticate again for accessing the Jupyter Notebook environment. Authentication should be straightforward when user is already authenticated in the VO-CLOUD's Master server.

FR 18 There must be a possibility to create new worker types utilizing the Apache Spark technology – Spark workers.

FR 19 Spark workers must be able to download files from the Master server's storage directly to the specified path in HDFS.

FR 20 Spark workers must have a defined set of default parameters that are passed to the Spark job.

FR 21 The set of Spark job parameters should be optionally configured by a user in the JSON configuration during the Spark job creation.

FR 22 The output of Spark job should be optionally downloadable from the HDFS back to the VO-CLOUD system.

FR 23 Master server must provide a possibility for user to browse the HDFS. User should be able to modify the HDFS in the same way as the Master server's storage.

FR 24 Plotter must be able to plot spectra files stored in the HDFS.

FR 25 Files stored inside HDFS must be visible and available for experimenting from the integrated Jupyter Notebook environment in the same way as the files from the Master server's storage.

3.3 Non-functional requirements

NFR 1 For security reasons, whole VO-CLOUD system must secure its communication over HTTPS protocol. Incoming connections over HTTP protocols must be redirected to HTTPS connections.

NFR 2 Newly implemented modules should use Docker technology in order to make deployment more straightforward.

NFR 3 Source codes of the VO-CLOUD system must be published under the Open Source license and they must be publicly available on a public repository.

NFR 4 The new Spark worker type must be able to run on the same application server as the Master server as well as on an application server on a different machine.

Chapter 4

Realisation

There are three main goals in the extension of the current solution of the VO-CLOUD system:

- Implementation of spectra plotter
- Integration of Jupyter Notebook environment
- Integration of HDFS and Apache Spark

This chapter is dedicated to the explanation of all of these goals in detail in the following sections.

4.1 Astronomical spectra plotting capability

The current version of the VO-CLOUD system has been originally designed to not differentiate between file types saved in the Master server's storage. It was user's responsibility to know what is an actual representation of respective files. Workers of the VO-CLOUD system have been designed in the exactly same way. They basically take a JSON configuration containing the list of files that they should download from the Master server. Then they pass the downloaded files and the same JSON configuration file to some computational process. The process is actually the element that should know what files it is working with. If it is desirable, the process can create a visualization output that Master server can present to the user.

For instance, the Preprocessing job type currently deployed on the VO-CLOUD system processes passed astronomical spectra files and produces an `csv` file containing a preprocessing output. It also produces a simple web application – interactive plotter for output spectra that utilizes `dygraphs`[12] – an open source JavaScript charting library. The VO-CLOUD system simply

[12] `http://dygraphs.com`

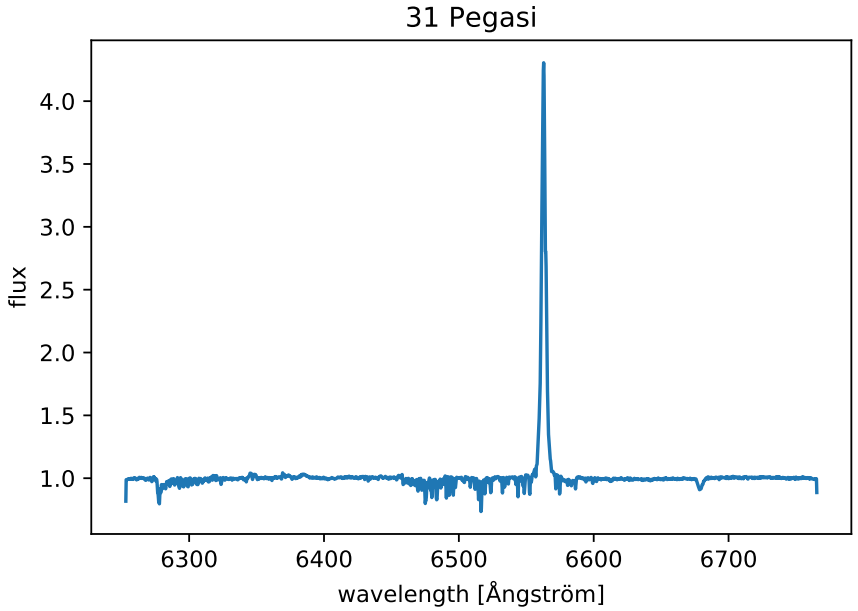


Figure 4.1: Example of astronomical spectrum plot of star *31 Pegasi*

takes the Preprocessing binaries as a black box that requires a JSON configuration and some input files and then it collects all the Preprocessing produces and it presents it back to the user as a set of files and an in-browser application. VO-CLOUD does not have to know anything about file types at any time currently.

Despite the fact that the VO-CLOUD system has been designed generally for any kind of data processing, nowadays it is especially used for processing of astronomical spectra files.

> *Astronomical spectrum file* – A file containing a record of an astronomical spectrum together with additional metadata describing when the spectrum was recorded, under what conditions, how it was processed, etc... Astronomical spectrum is a very important concept of the stellar astronomy, as it is a record of the electromagnetic radiation radiating from the observed object. Astronomical spectrum can be easily visualized as a function of its wavelength and a radiative energy so-called flux. [25] An example of plotted spectrum can be seen in figure 4.1.

Whereas the VO-CLOUD system's storage mostly contains only astronomical spectra files, there is currently no way to visualize them other than by sending them to a specific job that is able to plot them or by downloading them to the user's device and visualize them in some other application (e.g.

Spectral Analysis Tool (SPLAT)[13]). These approaches are not suitable as experimenting astronomers often require to visualize the selected set of spectra to check that spectra wavelengths are correctly cut after preprocessing phase, that its values are correctly normalized to $[0.0, 1.0]$ interval and so on. Therefore new functional requirements on the VO-CLOUD system emerged – to plot as much astronomical spectra file types stored in the system's storage as possible.

4.1.1 Core visualisation problems statement

Whereas it seems that the visualisation of astronomical spectra files is really straightforward, the opposite is true. The fact that the VO-CLOUD is a web application brings some problems that complicate the visualisation. This section is dedicated to description of these problems.

Multiple spectra file formats

Astronomical spectra can be stored in multiple specialized formats and even inside these formats there can be more ways in which spectra can be stored. The desired spectra visualiser should naturally support as much of these options as possible. Based on functional requirements, the visualiser should support following spectra file formats:

- *Flexible Image Transport System (FITS)* – It is a format for astronomical spectra files with extension `.fit` or `.fits` and it was designed in order to facilitate the interchange of astronomical image data between observatories. "A FITS file consists of a sequence of one or more header and data units (HDUs) optionally followed by special record. The structure of a FITS file is based on blocks with a length of 2880 8-bit bytes (23040 bits)"[26] There are more ways to store an astronomical spectrum inside a FITS file. The most straightforward way is to store two vectors with the same length – a vector of `x` axis values defining points in a wavelength and a vector of `y` axis values defining the actual flux value of the respective wavelength point ("respective" means that the point has the same index in vector). Another way is to write only a vector of flux values and to describe a wavelength axis in FITS metadata, e.g., define the value wavelength of the first point and the step distance to the next wavelength point either in linear or logarithmic scaling. However, the problem is that the names of metadata keys are not usually standardized and moreover some FITS files do not even fully meet the FITS specification, thus it is almost impossible to implement a visualizer supporting every astronomical spectrum FITS file.

[13] `http://star-www.dur.ac.uk/~pdraper/splat/`

- *VOTable* – The VOTable format is a recommendation developed by the IVOA organisation and it is newer than FITS specification. "The VOTable format is an XML standard for the interchange of data represented as a set of tables."[27] The table in this context contains an unordered set of rows and each row contains a sequence of cells. The VOTable format can be utilized in many ways (e.g. it is used in already mentioned protocol SSAP as a carrier of the SSAP query results), however, it is mainly utilized as a format for astronomical spectra files as it can carry spectra data as well as metadata. There are two main ways to store spectra inside VOTable format files:

  - `TABLEDATA` – Two vectors of an astronomical spectrum are mapped as rows with two columns – wavelength and flux pairs – using table XML elements (`TR` and `TD` elements – the same as in HTML).
  - `BINARY` – Wavelength and flux vectors are serialized in the binary format that is intended to be easy to read by parsers. It is basically a sequence of cells serialized as a sequence of bytes. [27] Every cell belongs to some row and some column and the size in bytes of the cell is defined by the column's data type. Column data types and additional metadata are specified at the beginning of the VOTable file.

- *CSV* – Whereas FITS and VOTable files represent a single astronomical spectrum, sometimes it is desirable to have multiple astronomical spectra stored inside a single file. One of the most useful formats for this purpose is a simple Comma-Separated Values (CSV) format. The CSV file contains each spectrum on a single row and for each spectrum it contains multiple values separated by a comma character (sometimes other characters such as a space or a semicolon can be used, however, a comma is used the most often). The first value of a spectrum row contains a spectrum identifier (e.g. the name of original file where spectrum has been taken from) and the rest of row values represents the flux spectrum vector. The CSV spectra file does not contain any header row. The great advantage of this approach is that the CSV spectra file can be easily split to multiple smaller CSV files. It is important to note that the CSV spectra files contain no information about wavelength vector. The CSV spectra file is usually an output of some preprocessing method that takes multiple spectra as an input and preprocesses them to a single CSV spectra file. Input spectra are interpolated to the same wavelength values and the resulting wavelength vector is exported as a `meta.xml` file – the file in VOTable format containing a row with the wavelength vector. Data mining methods do not usually need the `meta.xml` file as they mostly work only with spectra flux vectors. The

`meta.xml` serves especially for purposes of visualisation and additional preprocessing methods.

Spectra visualizer must support plotting of multiple astronomical spectra together. User can select either one file or multiple files to plot. These selected files may have different format. The CSV file must be plottable by itself (in this case it should be assumed that the wavelength vector contain values of $f(x) = x$ function where x is an index of a flux vector value) or with correct wavelength values if `meta.xml` file is specified.

Data volume problem

There is a big difference between visualising a single astronomical spectrum file with size of approximately 50 kB and multiple spectra (stored as multiple files or a single big CSV file) with total size in megabytes. If spectra for visualising were really small the best solution would be to transfer all necessary data to client's web browser and visualise them using a JavaScript code. The problem is that data are usually too big for web browser's JavaScript interpret to handle. Moreover, users with limited network connectivity would have to wait a very long time because web browsers usually wait until all data have been downloaded before they pass them to the JavaScript code – there is no way to visualise spectra continually as new data are being downloaded.

The better way to solve the big data problem visualisation is to generate an image of a plot on the server's side and then send it to user's web browser that simply shows the image. There are a few advantages to this solution:

- Client code have no responsibility of differentiating between multiple spectra file formats.
- There are no requirements on computational capabilities of user's device.
- There is no need for transfer of big amount of data from server to user's device – only one image of a desired quality.

This approach seems to be better than the first one, however, there is also a great disadvantage. An ability to zoom in the resulting image is conditioned by the image quality. Images would either have to be unnecessarily large or the quality after image zooming would be unacceptable. It is necessary to find the compromise between the first and the second approach mentioned in this section in order to implement a sufficient solution for the resulting astronomical spectra visualiser.

Technology integration problem

As it have been already explained in the previous section 4.1.1, the spectra visualising must at least partially take place on the server side. In order to do so it is important to implement mainly two following components:

- A parsing module for all possible astronomical spectra file types that takes a set of files as an input and returns a set of wavelength vector and flux vector pairs – one pair for each astronomical spectrum.

- A plotting module that takes output of the previous module and it plots all spectra into a plot image.

The problem is that there are no libraries implemented in the Java language that are able to parse FITS or VOTable spectra file formats. Whereas it would be a relatively easy task to implement VOTable parser in Java as it is basically a XML document and Java offers technologies for straightforward XML document parsing, the FITS format parsing would have to be implemented whole from scratch. Also, Java offers almost no tools with plotting capabilities. Plotting libraries written in Java language are mostly targeted at desktop applications and they do not fit with purposes of a web application.

On the other hand, Python programming language seems like a right way to go. Parsing of both FITS and VOTable spectra file formats can be implemented easily using the Astropy Python project – a community effort to develop a core package for astronomy using the Python programming language and improve usability, interoperability, and collaboration between astronomy Python packages [28]. Python also offers an excellent plotting library named Matplotlib.

> "Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems."[29]

The solution implemented in the Python programming language would be a straightforward to do, however, the VO-CLOUD system is implemented in the Java programming language. It is important to decide whether it is better to implement the spectra visualiser in Python language and to make an integration with the current solution more difficult, or whether to implement it in Java language to have an integration trivial but to implement parsing and plotting modules all from scratch.

4.1.2 Solution

After considering all above stated problems I have eventually decided to implement the whole astronomical spectra visualiser in the Python language as a new web application and then integrate this application to the current solution of the VO-CLOUD system. By using the Python programming language many thing have been simplified as it is possible to delegate many application responsibilities to libraries that this application utilizes.

The application was named `spectraviewer` and it was implemented as a web server application by utilizing a Python package named Tornado – a web

framework and asynchronous network library [30]. The data volume problem 4.1.1 has been solved smoothly by utilizing the WebSocket protocol. The WebSocket is a protocol that uses a transport layer of HTTP protocol in order to create bidirectional communication between a server and a client [31]. After assembling a WebSocket connection between a client and a server, the client can send a message to the server as well as the server can send a message to the client. Due to the fact that Tornado server is implemented on asynchronous principles, it is very easy to implement the WebSocket protocol using the Tornado Python package. Every WebSocket protocol event (connection opened, connection closed, message received) on the server side triggers a method call of a respective `tornado.websocket.WebSocketHandler` class instance. New instance of this class is created for every new incoming WebSocket connection.

The application works in the following way:

1. Client sends the list of spectra he would like to visualise.
2. Application parses the listed spectra and it saves the plot figure inside its temporal key-value storage – the key is a randomly generated unique identifier and the value is the figure itself.
3. Application responds to client with the HTML template containing JavaScript client code and the storage identifier.
4. Client renders the HTML template and he creates a new WebSocket connection to the specific server endpoint passing the storage identifier as an argument.
5. Server links newly received WebSocket connection with the figure stored inside the storage.
6. Server sends a message to the WebSocket connection containing an image of the figure.
7. Client shows the received image in the page.
8. Client can use panning or zooming tools on the image. When he does so, the parameters of expected transformation are sent to the server through the WebSocket connection.
9. Server applies desired transformation on the linked figure and it sends back a new image through the WebSocket connection.
10. Client can repeat steps 8 and 9.
11. When client closes the page, the WebSocket connection is closed and the server removes the relevant figure from the temporal key-value storage.

The Matplotlib Python package has been highly utilized in this solution as it provides a simple API for complex plots creation and, moreover, it contains an implementation of `WebAgg` backend – an engine that contains key-value store for plot figures, integrated support for zooming and panning tools and prepared JavaScript code for client side. This engine has been specially created for Jupyter Notebook environment to allow interactive working with Matplotlib plots, however, it perfectly fits with purposes if this application as well.

The realisation of spectra file parsers is very simple. The application utilizes Astropy Python package for parsing files in FITS and VOTable formats. In the FITS format it was necessary to distinguish between formats containing the wavelength vector inside the data part and formats having the wavelength vector described by metadata either in linear or logarithmic scaling. In the case of CSV files, if there is `meta.xml` file containing the wavelength vector in the set of selected files to plot, spectra from CSV file are plotted by using this wavelength vector, otherwise it is assumed that wavelength vector contains values $0, 1, 2, \ldots$

The last important part was to integrate the newly created Python application to the VO-CLOUD system. The application utilizes the fact that the VO-CLOUD Master server's storage is actually directly accessible on the filesystem of the hosting system. The `spectraviewer` application is also deployed on the same server, thus the only thing that is necessary to be passed from the VO-CLOUD's Master server to the `spectraviewer` in order to visualise selected spectra is a set of spectra paths in the Master server's storage. There is no need to move possibly large spectra from one application to another – spectra plotting application simply works on the same data set as the VO-CLOUD's Master server. Also, there is no need for communication between the Master server and the `spectraviewer` application as all communication goes through the client's web browser. The workflow of spectra plotting is following:

1. User selects desired spectra files for plotting in the Master server's user interface.
2. User clicks *Plot selected spectra* button.
3. Master server creates a dialog window containing an `IFRAME` HTML element pointing to the view endpoint of the `spectraviewer` application.
4. User's web browser automatically makes a GET HTTP request to the endpoint and renders the response in the `IFRAME` window.
5. User now communicates with the `spectraviewer` application (see 4.1.2).
6. User closes the dialog window – the WebSocket connection is closed.

The view endpoint URI mentioned in the step 3 is simply a web resource with URI `/view` that takes two parameters:

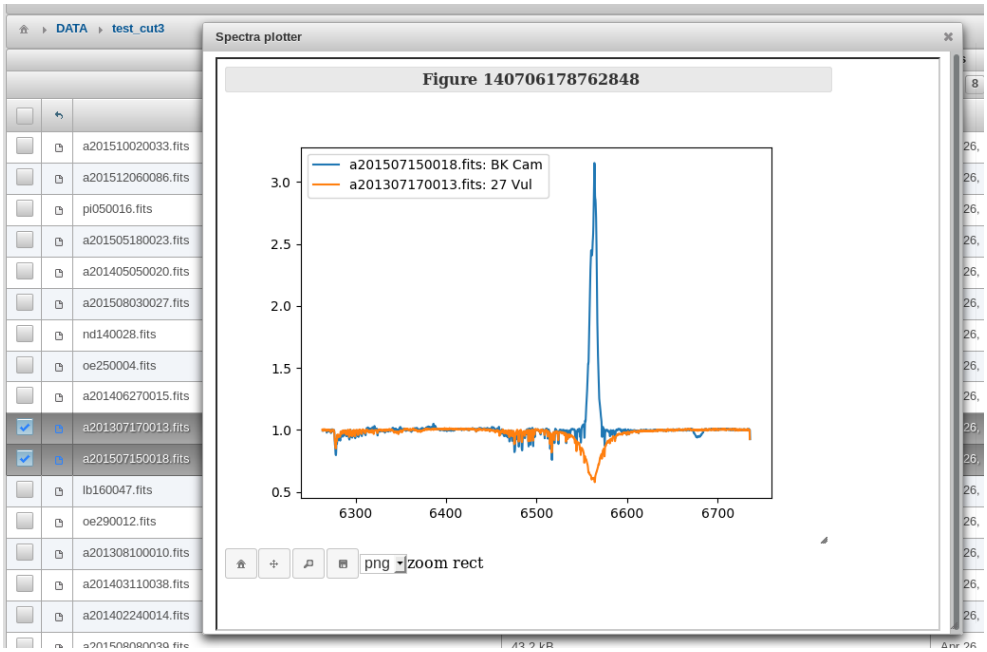


Figure 4.2: Example of integrated `spectraviewer` application

- `spectra` – List of spectra file paths separated by a comma character.
- `prefix` – Optional parameter that specifies a path that every path from `spectra` parameter should be prefixed with.

For instance, if user wants to visualise two spectra files in a directory `DATA/test`, the resulting URI would be:

```
/view?prefix=DATA%2Ftest%2F&
        spectra=mi140017.fits%2Cth210042.fits
```

Also, as you can see in the example, special characters (in this case slashes and commas) are URL escaped. An example of the `spectraviewer` application can be seen in figure 4.2.

Docker

In order to simplify deployment on the server the whole `spectraviewer` application has been bundled as a Docker image. Created Docker image has been also pushed to the public Docker Hub repository[14] under name `kozajaku/spectraviewer`. The whole application is now deployable by executing a single command:

[14] `https://hub.docker.com`

```
docker run -p 7000:7000 --name spectraviewer -d
           -v /vocloud/storage:/tmp/filesystem:ro
           kozajaku/spectraviewer
```

The command simply pulls the image from the public repository (if it has not been pulled already), it mounts the `/vocloud/storage` directory on the hosting system to the `/tmp/filesystem` as a read-only directory in the container, it exposes TCP port 7000 from the container to the port 7000 on the hosting system, it names newly created container `spectraviewer` and it starts the container in a detached mode.

4.2 Jupyter Notebook environment integration

As it has been already mentioned in the section 3.1.5, the Jupyter Notebook environment is a very powerful tool that can help users to create complex experiments in Jupyter Notebook documents, to invoke code on the server side and to collect results back by using only a web browser application. The created `spectraviewer` tool can be easily used to quickly visualise a selected set of astronomical spectra files, however if user needs some other way of visualisation or if he requires some programmatic way to filter spectra or to preprocess them before visualisation, the `spectraviewer` tool is not sufficient. In order to do so, a user would have to download the desired set of astronomical spectra files from the VO-CLOUD's Master server to his device and then conduce experiments over these spectra on his local device (e.g. in the Jupyter Notebook environment running directly on his device). This approach is not really straightforward as users need to start their own instance of the Jupyter Notebook environment on their device which could be tricky if they are not familiar with Python packages installation. Moreover, they need to download whole astronomical spectra files even if they are for instance interested only in the first few spectra (lines) of a big CSV file.

One of the main goals of this Master's thesis is to find a way to integrate the Jupyter Notebook environment directly to the VO-CLOUD system. An expectation is that a user that is interested in the set of data stored inside the Master server's storage would be able to transition to the Jupyter Notebook environment from the VO-CLOUD by clicking a single button. The environment would run on the side of the VO-CLOUD server and it would have a prepared Jupyter Kernel for Python language. The Kernel would have installed all scientific and visualisation Python packages that the user would be likely to use. Also, the Jupyter Notebook environment would have a direct access to all data stored on the VO-CLOUD's storage thus a user would be able to directly utilize files without the necessity to copy them to a new location. A user would even be able to implement a new way to visualise astronomical spectra if he decided to do so.

The whole Jupyter Notebook server is implemented in the Python programming language and its implementation utilizes the Tornado web framework – the same framework that is involved in the implementation of the `spectraviewer` application. When the Jupyter Notebook server is started, it exposes the web user interface on a TCP port (implicitly 8888) and the directory where the `jupyter notebook` command has been executed is considered as the working directory for the Jupyter Notebook environment. From the Jupyter Dashboard a user can create a new Notebook document in the current directory but he can also open a terminal window. The terminal window works exactly the same way as if a user would connect directly to the hosting server and execute terminal commands in the server's console. The terminal window has exactly the same permissions as a system user under which the Jupyter server has been started.

4.2.1 Core Jupyter integration problems

There are a few problems that complicate deployment and integration of the Jupyter Notebook environment to the VO-CLOUD system.

Authorization problem

Functional requirements state that only users authenticated in the VO-CLOUD system can have access to the Jupyter Notebook environment. Moreover, only users with a user role either `MANAGER` or `ADMIN` can have access to the environment. The Jupyter Notebook implements a token-based authorization that could be utilized for this purpose. The Jupyter's token-based authentication is enabled by default in newer versions of the Jupyter Notebook and it works in the following way:

1. When the Jupyter Notebook is started the token is either passed as a program parameter or it is generated randomly.

2. User who knows the token passes the token as a HTTP parameter in the first request to the Jupyter server. If the parameter is not passed or is invalid the Jupyter server redirects user to a dedicated page where user can insert the token as a form input.

3. If the token passed by a user is correct server sends an authorization cookie.

4. From now user sends the authorization cookie in every request instead of the token and the server accepts it as a valid authentication.

The token-based authentication could be theoretically involved in the integration in the following way:

1. The Jupyter Notebook server is started with a specific token and the VO-CLOUD Master server knows this token.
2. When a user demands a transition from the Master server to the Jupyter server, the Master server checks that the user has an authorization to do so.
3. If the user has an authorization, the Master server sends the Jupyter authentication token to the user.
4. User connects to the Jupyter Notebook server using the authentication token.

The stated solution is very simple, however, there is a security problem – there is only one security token and it is valid all the time. If any user's device would be compromised, any potential attacker having this token would be able to authenticate to the Jupyter Notebook environment and to utilize the resources of the server where it have been deployed. In order to change the token it would be necessary to generate a new token, to restart the Jupyter Notebook server and set the new token to it and also to set the new token to the VO-CLOUD Master server. Therefore, it is necessary to find a better solution for this problem.

Users isolation problem

Beside the authorization problem, functional requirements also state that users with access to the Jupyter Notebook environment must have working directories isolated from each other, i.e., Jupyter Notebook documents and additional files that users store inside the Jupyter's working directory should be inaccessible by all the other users. The Jupyter Notebook environment has not been designed to solve this problem. It is necessary to have multiple Jupyter Notebook servers deployed – one for each user. However, this brings additional problems with authorization as a user has to connect to the relevant Jupyter server with correct authentication credentials.

System isolation problem

It is important to mention that all users with access to the Jupyter Notebook environment have basically the same permissions as the system user under which the Jupyter Notebook environment has been started. Users can either utilize the code executed inside the Jupyter Notebook documents or they can directly use the terminal feature of the Jupyter Dashboard. If the hosting system has properly configured permissions for this system user, there should be no problem. However, there is always a chance of some access rights misconfiguration that would expose the whole system and its applications to the security risk.

4.2.2 Solution

The integration problems can be solved elegantly by utilizing a JupyterHub project together with Docker technology.

4.2.2.1 JupyterHub

The JupyterHub project is a set of processes that together provide an ability for users to log to their own separate instance of the Jupyter Notebook server – each with different authentication credentials and with a different working directory [32]. The JupyterHub is comprised of three major subsystems[32]:

- *Single-User Jupyter Notebook Server* – The Jupyter Notebook server that is started for every user that logs in. The object that starts these servers is called *Spawner*.
- *Proxy* – The public part of the JupyterHub project implemented in JavaScript programming language upon *Node.js*[15] technology. The proxy dynamically routes HTTP requests to the Hub and to the Single-User Jupyter Notebook servers.
- *Hub* – Manages user accounts, authentication and coordinates individual Jupyter Notebook servers using the Spawner. The Hub is implemented in Python by using the Tornado web server.

For the sake of this work, it is important to describe how the JupyterHub authentication and server spawning work[32]:

1. Proxy implicitly forwards all incoming request to the Hub process exposed on URI `/hub/`.
2. When a user is authenticated correctly in the Hub, the Hub's Spawner creates a new Jupyter Notebook server instance for the user.
3. The Proxy is notified to forward `/user/<username>/*` requests to the newly created server instance (`<username>` is substituted for user's username).
4. Two HTTP cookies containing an encrypted token are created. One for `/hub/` and another for `/user/<username>`.
5. User is redirected to the newly created Jupyter Notebook server instance.

After ensuing previous steps, the Jupyter Notebook server instance for the specific user has been started and exposed on the JupyterHub's proxy. However, there must be an additional authorization layer on the side of the Jupyter Notebook server, otherwise anyone would be able to access the server instance[32]:

[15] `https://nodejs.org`

1. The Jupyter Notebook server instance forwards the encrypted cookie to the Hub for authorization.
2. If the cookie is valid, the Hub responds with the user's username.
3. If the user is the owner of the Jupyter Notebook server instance, access is allowed.
4. If the username is wrong or the cookie is invalid, the user is redirected to `/hub/login`.

4.2.2.2 JupyterHub Spawner

Each Jupyter Notebook server instance is started by the Hub subsystem by an object called Spawner. The Spawner object has following responsibilities[32]:

- Start the Jupyter Notebook server process.
- Poll whether the process is still running.
- Stop the process when necessary.

There are many implementations of the JupyterHub spawner object. The implicit one is called `LocalProcessSpawner`. This spawner implementation works only on UNIX systems as it spawns new server instances as a process under the UNIX system user with name matching the authenticated one in the JupyterHub authentication process. There are cases where this solution could be sufficient, however, in this case there is no mapping between UNIX system users and users inside the VO-CLOUD system.

The crucial implementation of the JupyterHub Spawner that is utilized in the VO-CLOUD–JupyterHub integration is named `DockerSpawner`. This spawner implementation starts for each authenticated user a Docker container that packages the whole Jupyter Notebook server. The utilization of Docker containers has also a great advantage. The environment inside the running Docker container is isolated from the hosting system, therefore, this approach very effectively deals with the system isolation problem 4.2.1. If the Jupyter Notebook server would have badly configured access permissions, in the worst case scenario a user could break only the server on the container itself. Other Docker containers or the hosting system are inaccessible from the inside of the Docker container.

4.2.2.3 JupyterHub Authenticator

The JupyterHub Authenticator is another important object in the Hub subsystem. Its responsibility is to provide authentication capabilities to the JupyterHub server. Practically, the Authenticator is any Python class that inherits from the class `jupyterhub.auth.Authenticator`. It consists of a single

method `authenticate` that basically takes a username and a password of a user that is trying to authenticate. If the user's credentials are correct the method must return the user's username. Otherwise, the method must return the special Python value `None`.

In order to integrate the JupyterHub to the VO-CLOUD system, it was necessary to design and implement the way of authentication. The workflow of the authentication is done in the following way:

1. User connects to the VO-CLOUD's Master server and logs in with his credentials.
2. In order to transition to the Jupyter Notebook environment, user clicks the *Jupyter* button in the Master server's user interface.
3. Master server generates a new randomly generated token, links the token with the user's username and saves it temporarily in the in-memory storage.
4. Master server sends the token to the user's web browser.
5. User's web browser does a HTTP POST request to the JupyterHub's login endpoint `/hub/login`. The POST request contains two parameters – the username identical to the user's username on the Master server and the token.
6. JupyterHub delegates the authentication task to the authenticator's implementation – the `VocloudAuthenticator`.
7. The `VocloudAuthenticator` does a HTTP POST request to the Master server's token checking endpoint. It passes the token as a POST parameter.
8. If token is valid and not expired, the Master server returns the username of the user account linked with this token and it invalidates the token.
9. The `VocloudAuthenticator` checks that the username received from the Master server matches the one received from the user's web browser.
10. If usernames match it returns the username to the JupyterHub.
11. User is now authenticated to the JupyterHub.

There is no other way to authenticate to the JupyterHub than to transition from the VO-CLOUD's Master server using the provided token. Every generated token is valid only for a limited amount of time and it is invalidated as soon as it is used. This solution is significantly better from a security point of view than the solution explained in the section 4.2.1, as there is no way to utilize potentially caught token, since it is valid only for a very short period of

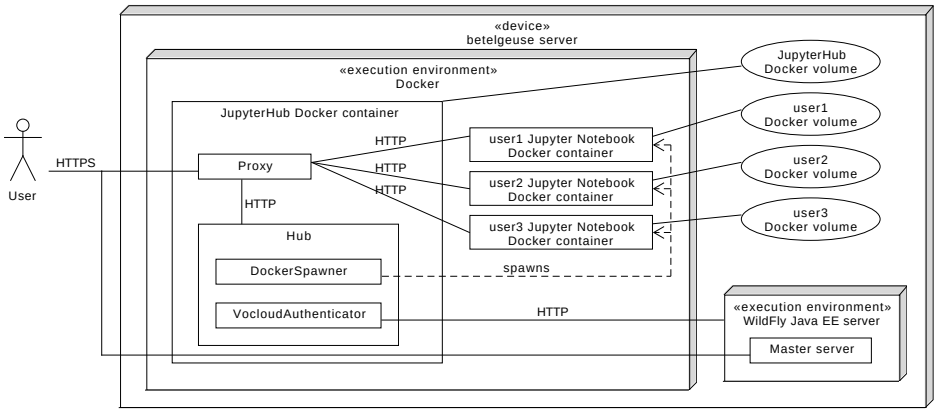


Figure 4.3: JupyterHub solution Docker deployment

time. Security cookies are also difficult to exploit, as they contain encrypted information identifying the user's web browser and device.

The implementation of the VO-CLOUD's Master server has been extended to support the new functionality of the authentication token endpoint. It was implemented as a very simple RESTful service. The token in-memory storage has been implemented as a Singleton EJB bean.

4.2.2.4 Deployment

The Docker technology has been used both for Jupyter Notebook server instances and for the whole JupyterHub itself. This solution is really interesting as there is the JupyterHub Docker container that requires to spawn additional Docker containers with the Jupyter Notebook server instances on the same server where the JupyterHub container is deployed itself, but not inside the JupyterHub's container. Moreover, in order to have access to the Master server's storage and jobs directory it is necessary to mount these directories to every one of the individual Jupyter Notebook Docker containers. Every Jupyter Notebook Docker container has its own working directory that is backed in the filesystem of the hosting system.

In order to be able to start a new Docker container from the inside of another Docker container but not inside the Docker container itself, it is necessary to mount the Docker socket file `/var/run/docker.sock` from the hosting server to the spawning Docker container. The Docker socket is basically a client for communicating with the Docker daemon process – the Docker container with the mounted Docker socket will gain an ability to control the Docker daemon in the same way as it can be done directly from the hosting server.

The deployment of the JupyterHub solution can be seen in the figure 4.3. The whole deployment solution has been implemented as a project named `vocloud-jupyterhub` and the deployment consists of adjustment of `.env` configuration file and of invocation of two commands:

```
make
docker-compose -d
```

The first command creates all necessary Docker networks and volumes and then it builds the Jupyter Notebook server Docker image and the JupyterHub image. The second command starts the JupyterHub container in a detached mode and exposes its web interface on the TCP port specified in the `.env` configuration file.

Since JupyterHub is running on the `betelgeuse` server, it was necessary to expose the server also in the reverse proxy on the `vocloud-dev` server. The small improvement was also done – when a user accesses the `/hub/login` URI of the JupyterHub server using a HTTP GET method, the reverse proxy sends a redirect back to the user pointing to the VO-CLOUD's Master server login page. Now when a user logs out from the JupyterHub server or when he randomly accesses some JupyterHub's resource without authentication, he is automatically redirected to the VO-CLOUD's login page.

4.2.3 Summary

Every user now has access to his own instance of the Jupyter Notebook server that is started on demand as a Docker container. Every user has his own working directory isolated from all other users where he can create new files and Notebook documents. Also, in this working directory there are two read-only directories mounted from the hosting system – the Master server's storage and jobs directory. Users have direct read access to all files saved in these directories without a necessity to copy files from them to some other location.

Users have access to the terminal window feature of the Jupyter Notebook environment, however, this terminal has access only to the specific Docker container as it is isolated from the hosting system. Users can use the terminal window to install additional Python packages, that are not implicitly provided, however, the Jupyter Notebook Docker image should be already provided with all necessary Python packages such as Matplotlib, Astropy, NumPy, pandas and many others.

User can transition to his Jupyter Notebook environment from the VO-CLOUD system by clicking only a single button – whole authentication process is done automatically in the background. When user logs out from the Jupyter Notebook environment, he is automatically redirected back to the VO-CLOUD system.

4.3 Apache Spark and HDFS integration

The final goal if this Master's thesis is to find a way to integrate the VO-CLOUD system with the Hadoop infrastructure in order to be able to utilize the distributed file system HDFS and to start Apache Spark jobs using the Hadoop YARN scheduler. The current solution of the VO-CLOUD computational workers is usable, however, the usability of workers is limited by two factors:

- The set of input data must be always downloaded again from the Master server's storage to the computational worker for every individual job. This approach enables the deployment of workers on additional separated devices.
- The worker's computational task is always executed on a single device of the specific worker. The computational capability is limited by CPU and memory resources of the single device.

While the current solution of the VO-CLOUD system can be easily applied on a processing of a limited amount of data set, the need has emerged to be able to process the whole astronomical spectra archive LAMOST-DR1. The Large Sky Area Multi-Object Fibre Spectroscopic Telescope (LAMOST) is a meridian active reflecting Schmidt telescope located in Xinglong Station of national Astronomical Observatory in China [33]. Data Release 1 (DR1) of this telescope's observations comprises of 2,202,000 astronomical spectra files encoded in FITS format. Every astronomical spectrum file takes up approximately 90 kB of a disk space and the whole uncompressed archive in total takes up 189 GiB of a disk space. It is unrealistic to use current computational workers for purposes of processing the whole spectra archive, as the whole archive would have to be stored in the VO-CLOUD's storage and also it would have to be downloaded to a worker for every computational job. It is necessary to design a better solution – utilize the capabilities of the Apache Spark and the Apache Hadoop infrastructure.

4.3.1 Hadoop deployment

Firstly it was necessary to deploy the Hadoop infrastructure to the servers in Stellar Department of the Astronomical Institute of the Czech Academy of Sciences in Ondřejov where the VO-CLOUD system is also running. It was decided that the Hadoop computational cluster would consist of two servers:

- `betelgeuse` – The server where the VO-CLOUD system is currently deployed. It has 12 CPU cores supporting the Hyper-Threading technology (24 virtual CPU cores) and 128 GiB RAM memory.
- `antares` – The server with 8 CPU cores and 24 GiB of RAM memory.

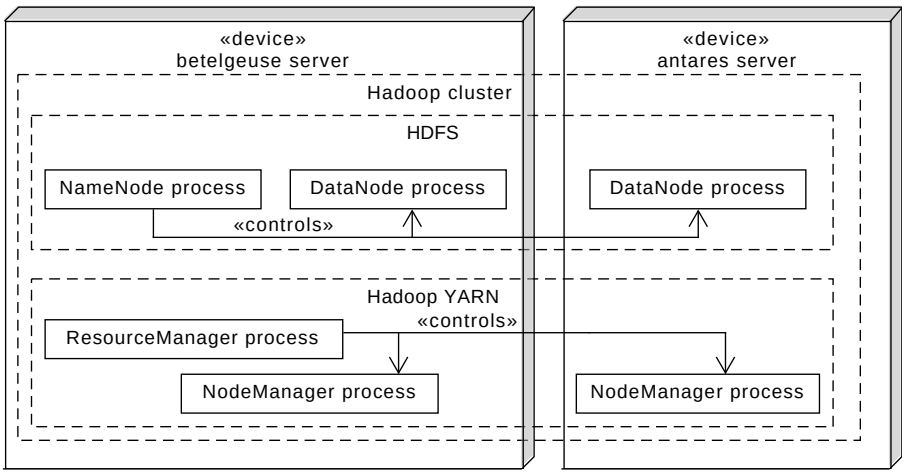


Figure 4.4: Hadoop cluster deployment on Ondřejov servers

As it has been already explained in the section 3.1.3, the Hadoop Distributed File System (HDFS) is comprised of `NameNode` and `DataNote` processes. The `DataNode` processes is basically the component that saves the data blocks in the device's filesystem. The `NameNode` is the controlling component that has information about all files stored in the HDFS, their data blocks and where are these blocks saved. In this case, the `DataNode` process is running on both `betelgeuse` and `antares` server and the `NameNode` controlling process is running only on the `betelgeuse` server.

The Hadoop YARN has been deployed very similarly. It consists of two processes:

- `ResourceManager` – The YARN scheduler and resource managing process that has information about all available `NodeManager` processes.
- `NodeManager` – The process that can receive a computational work from the `ResourceManager`.

The `betelgeuse` server runs both processes, whereas the `antares` server runs only the `NodeManager` process.

The diagram of the whole Hadoop infrastructure deployment can be seen in the figure 4.4.

4.3.2 Apache Spark

The installation of the Apache Spark was very simple – it was only necessary to download the Apache Spark binaries and to properly define environment

variables to correctly point to the path of the Apache Hadoop installation directory.

All jobs that are expected to be executed inside the Apache Spark environment are submitted using the `spark-submit` script that is bundled with the Apache Spark installation package. Multiple parameters can be passed to the `spark-submit` script. The most important are:

- `--master` – Defines where the Spark job should be running. In order to have the job managed by the Hadoop YARN scheduler, it is necessary to pass `yarn` as a value for this parameter.
- `--deploy-mode` – Defines where the execution driver should run. The driver is the application that orchestrates the job execution of individual executors. There are two options that can be passed to this parameter:
  - `client` – The driver should run on the side of the device where the `spark-submit` script has been executed. This option is picked when it is necessary to instantly see the progress of job's execution.
  - `cluster` – The driver should run on any device in the cluster. The cluster's resource manager simply picks the best suitable cluster's node for this task.
- `--num-executors` – The count of computational executors. This option is only used together with the `--master yarn`. In fact, the Hadoop YARN allocates its resources for computational containers named executors. Every executor can run only on a single cluster node, however, multiple executors can run on the single node. Each executor requires to have an allocated specific amount of CPU cores and a specific amount of RAM memory. The Spark computation on YARN can start when the desired amount of executors have been started with all required resources.
- `--executor-cores` – A number of CPU cores allocated per each executor.
- `--executor-memory` – A RAM memory amount allocated per each executor.

4.3.3 Small files problem

As it has been already explained in the section 3.1.3, the HDFS is ineffective in storing a big amount of small files, because it uses large data blocks (e.g. 128 MB). It is significantly better when there is smaller amount of big files. In order to be able to execute Spark computational jobs over the LAMOST-DR1 spectra archive, at first it is necessary to copy the whole archive to the HDFS to make the data available on all cluster nodes. The problem is that

the LAMOST-DR1 archive is comprised of millions of small spectra files that the HDFS cannot handle.

In order to solve the problem it is necessary to find a way to merge multiple small files together to make a big file. In some problem instances this task is very trivial. For instance, it is easy to merge multiple CSV files together by appending them one after another. However, some file formats are not mergeable, as they have a complex structure. The FITS format is unfortunately one of these formats, therefore it is necessary to find a better way to merge astronomical spectra FITS files together.

4.3.3.1 SequenceFile

`SequenceFile` seems to be a good solution to the small files problem. It is a flat file consisting of binary key/value pairs and methods for its reading and writing are part of the Hadoop API [34]. In this context, multiple astronomical spectra files would be merged into a single `SequenceFile`, where the key would be the name of the original spectrum file and the value would be the content of the spectrum file itself. `SequenceFile` stores the key/value pairs serialized in a binary format one after another. It does not offer an ability to quickly find a desired key (i.e. file name) as it has no indexes to the keys stored inside the `SequenceFile`, however, this functionality is not even required in this case, as all spectra need to be processed.

Unfortunately, there is a serious problem that makes the deployment of the SequenceFile format almost impossible. Apache Spark jobs can be implemented in three programming languages – Java, Scala and Python. The SequenceFile API methods have been programmed in a Java language and these methods utilize a Java Serialization mechanism. This API can be used naturally in Java and also in Scala as it runs on the Java Virtual Machine and it can call any Java API. The problem is the Python programming language, for it does not have the Serialization mechanism from the Java language and thus it naturally has no implementation of SequenceFile format. It is expected that the Python programming language could be used for programming a Spark job, therefore it is necessary to find a better way of spectra files merging.

4.3.3.2 Apache Avro

Apache Avro is a data serialization system that can be utilized to solve the small files problem. The Avro relies on schemas that are written in a JSON format. Every file that was written in the Avro format contains, apart from the data, the JSON schema itself. Data in the Avro format are saved in the compact binary format derived from the JSON format and they are stored as a sequence of rows. Every row represents one record with the format matching the defined Avro schema. The Avro has APIs written in many languages including Java, Scala and Python. [35]

```
{
  "type": "record",
  "name": "FitsFiles",
  "aliases": ["Fits"],
  "fields" : [
    {"name": "name", "type": "string", "doc": "Fits
       file name"},
    {"name": "content", "type": "bytes", "doc": "
       Binary content of the fits file"}
  ]
}
```

Figure 4.5: Apache Avro schema JSON

```
...
writer = DataFileWriter(open("output.avro", "wb"),
   DatumWriter(), schema, codec="deflate")
for f in filenames:
    with open(f, "rb") as fd:
        content = fd.read()
    writer.append({"name": f, "content": content})
    writer.close()
...
```

Figure 4.6: Fragment of code serializing spectra files to the Avro format

Let's illustrate the Avro serialization format on the current problem. It is basically necessary to achieve the same functionality as in the SequenceFile format. The designed schema can be seen in the figure 4.5. The schema contains definition for two fields (i.e. two columns) – the first (`name`) specifies the name of the original spectrum file and the second (`content`) specifies the binary content of the file itself. The only action that remains is to use this schema to serialize spectra from the LAMOST-DR1 archive to Avro format and to push the Avro files to the HDFS. The simple tool named `spectra-avro-serializer` has been implemented for this purpose in the Python language. The most important fragment of code of this tool that takes the schema and spectra files and serializes them to a single Avro file can be seen in the figure 4.6.

The whole archive has been processed by the `spectra-avro-serializer` tool. Instead of 2,202,000 small spectra files, the archive now consists of only 1,169 files in serialized Avro format. Moreover, as can be seen in the example 4.6, the `deflate` compression codec has been utilized to make the archive even smaller. Instead of 189 GiB it takes up only 85.7 GiB of storage

space. All newly created Avro files have been moved to the HDFS and can be now used as a data source for any Apache Spark job.

4.3.4 VO-CLOUD integration

One of the goals of this work is to find a way to integrate the Spark job submitting feature utilizing the HDFS to the existing solution of the VO-CLOUD system. Currently there is only one implemented tool that utilizes the Apache Spark – the preprocessing tool named `vocloud_spark_import`. However, there is expected to be more tools in the future that would utilize an output of the preprocessing tool and produce significant results. Therefore, it is necessary to design the integration solution in a general way to allow an easy adoption of new Spark job types.

Every Spark job method is expected to work in the following way:

- The binaries of the Spark job (hereinafter Application) are prepared in some specific directory on the server.
- The Application is written in either Java or Scala or Python programming language.
- The Application expects to be executed with exactly one parameter – the path to the JSON configuration file.
- The JSON configuration contains parameters that define a behaviour of the Application. It is provided by a user.
- Every individual Application could require a different set of the parameters passed to the `spark-submit`.
- A user can amend the `spark-submit` parameters.
- The Application takes as an input data stored inside the HDFS.
- The Application produces output to the HDFS.

In the current state of deployment the VO-CLOUD system is deployed on the server that is also a part of the Hadoop cluster. This generally does not have to be true as the cluster could theoretically run on a different set of servers. The integration solution must be designed generally to also meet this requirement.

4.3.4.1 Spark Worker

The solution has been designed as a new type of the VO-CLOUD Worker named Spark Worker. Whereas the Universal Worker focuses on a general execution of processes, the Spark Worker is deeply focused on an execution

of Apache Spark jobs. The Master server has not been altered in any way. The Spark Worker from the Master server's point of view provides the same functionality as the Universal Worker – it is exposed as a UWS service and for every new job it expects a JSON configuration.

The task of the Spark Worker is to provide a UWS service endpoint for each available Application. The workflow of the Spark Worker can be described in the following way:

1. Spark Worker receives a Spark job execution request. The JSON configuration is passed as a request parameter.
2. Spark Worker prepares the input data on the HDFS if necessary.
3. Spark Worker prepares the `spark-submit` script parameters.
4. Spark Worker prepares the JSON configuration file for the Application.
5. Spark Worker executes the Application – it calls the `spark-submit` script and passes to it prepared parameters, Application binaries and a path to the prepared configuration file.
6. Spark Worker waits until the execution finishes.
7. Spark Worker downloads the output data from the HDFS if desired.
8. Spark Worker results are downloaded by the Master server.

Similar to the Universal Worker, the Spark Worker is also based on the XML configuration file that specifies the whole Spark Worker. The XSD schema of this configuration file can be seen in the Appendix D. The most notable tag from the XML configuration file is `<submit-params>` that specifies the set of parameters that are passed to the `spark-submit` script. This tag can be specified in two places of the XML document – in the XML root tag and in the definition of each worker (Application). The parameters specified in the XML root tag define an implicit set of parameters that is utilized for every Application. The parameter set can be extended or overwritten by parameters specified in the worker's definition. Moreover, the JSON configuration can contain an additional set of parameters that extends or overwrites parameters defined in the XML configuration. This allows user to have a full control over the Spark execution, however, if he does not specify any parameters the implicit ones defined in the Spark Worker XML configuration file are used.

The JSON configuration that user passes to the Spark Worker consists of the following JSON objects:

- `download_files` – Defines a set of files that should be downloaded from the VO-CLOUD's Master server and stored in the HDFS in the specified paths.

- `spark_params` – Defines a set of additional `spark-submit` script parameters.
- `job_config` – Defines an actual JSON configuration that should be passed to the Application.
- `copy_output` – Defines a set of files or directories in the HDFS that should be downloaded to the Spark Worker and subsequently to the VO-CLOUD's Master server.

All these specified JSON objects are optional. If they are not defined, no files copied between the Spark Worker and the HDFS and the unmodified parameters specified in the Spark Worker's XML configuration file are used. If the `job_config` parameter is not specified, the whole passed JSON configuration is considered as an input for the Application. For instance, if user wants to execute a Spark job that takes as an input data that are already available on the HDFS and if user does not want to modify implicit `spark-submit` parameters, he passes the exactly same JSON configuration as is expected by the Spark computational job.

Examples of the Spark Worker configuration and an explanation of each possible parameter are described in detail in the Appendix F.

4.3.4.2 Utilizing a different Hadoop cluster

The designed and implemented solution of the Spark Worker is not tightly coupled with the Ondřejov servers infrastructure. It is also possible to use a Hadoop cluster that is deployed on a different set of servers. In order to utilize the different Hadoop cluster it would be necessary to install the Spark Worker on some of the cluster's node. Similar to the Universal Worker, the Spark Worker also uses only the JavaServlet technology (not EJB), therefore it is necessary to deploy it inside a Java EE servlet-enabled server (e.g. Tomcat). The web port of the Java EE server must be accessible for the Master server, therefore the port must be either exposed publicly or the routing tunnel has to be defined. Also, if the feature to download data from the Master server's storage to the HDFS is required it is necessary to have the Master server web port accessible for the Spark Worker. This is usually not problem, as the Master server is usually available publicly. However, in some cases the cluster could have firewall rules that block the access to the public network. In this case the firewall exception would have to be made.

4.3.4.3 HDFS browsing capability

The Spark Worker allows users to create and execute Apache Spark jobs running on the Hadoop cluster in the same way as any other jobs that he is used to use. He simply passes the JSON configuration that contains all necessary

information to conduce a preprocessing or a computational experiments. Despite the fact that he can specify in the configuration to download files from the Master server's storage to the HDFS, to execute Spark job and to download the results from the HDFS back to the Master server, the usual way to use this worker is to use the data already available in the HDFS and leave the results in the HDFS without downloading anything. The problem is that user has currently no way to browse the HDFS other than by connecting directly to the cluster using the SSH protocol and by executing terminal commands. This approach is insufficient as he needs to have access rights to the server.

This problem has been solved elegantly by utilizing a project developed by Cloudera Inc. called `hdfs-nfs-proxy` [16]. The project simply allows to mount the whole HDFS as a directory in the UNIX filesystem. The HDFS has been mounted as a new directory inside the Master server's storage. Now users can view the HDFS directly from the Master server's user interface. They can even create/delete/modify files and folders that are saved in the HDFS if they are in a user role with write permissions. Moreover, the HDFS directory is available for both the `spectraviewer` tool and the Jupyter Notebook server, therefore they can easily visualise spectra stored in the HDFS and they can also conduct experiments inside the integrated Jupyter Notebook environment.

4.4 Future improvements

There are a few improvements that could be done in the future. The first one is related to the web user interface of the Master server. When a user creates a new job he has to specify the JSON configuration that is passed to the relevant Worker (either Universal or Spark). There is a possibility to choose a configuration from the set of prepared configurations and then to alter it, however, this is still very counter-intuitive way from the user's point of view as he still needs to modify some possibly complex configuration. One of the future improvements could be to replace the direct JSON configuration modification for a set of HTML input elements where user could simply modify only the parameters that are relevant for him in a dedicated input box. This could make a process of jobs creation much easier.

Another possible improvement could be to implement the Avro serialization functionality directly to the Spark Worker. Currently the Avro serialization format has been used only to process the LAMOST-DR1 archive in order to be able to get the whole archive to the HDFS. It is possible to configure the Universal Worker to involve the same Avro serialization tool that was created for the serializing of the whole archive, however, in the future it would be better to implement it directly to the Spark Worker.

Lastly, it would be useful to package the whole VO-CLOUD system as a set of Docker images and then create some deployment scripts. As can be seen

[16] `https://github.com/cloudera/hdfs-nfs-proxy`

in the Master server's installation guide in the Appendix E, it can be really tricky for inexperienced users to deploy their own instance of the VO-CLOUD system. If an example instance of the VO-CLOUD system would be packaged as a Docker image, he could start the whole VO-CLOUD example instance by executing only a few terminal commands.

Conclusion

The goal of this Master's thesis has been met. The fundamental concepts, the workflow and the deployment of the original VO-CLOUD system have been analysed. Many new components have been implemented and integrated into the VO-CLOUD system. A user using the VO-CLOUD system can now easily visualise selected astronomical spectra directly in the web user environment or he can seamlessly transition to the Jupyter Notebook environment where he can conduct his own experiments or implement other ways of visualisation. He can also create complex computationally intensive jobs that are executed on the Hadoop computational cluster by utilizing the Apache Spark technology. The Docker technology has been highly involved in the design and implementation of new components.

I have gained a valuable experience during the process of designing and the implementation of new components especially in the areas of Apache Hadoop, Apache Spark, Docker and Jupyter Notebook technologies. Also I have acquired a knowledge about the fundamental concepts of VO technologies and about the astronomy in its entirety.

Bibliography

[1] Hanisch, R.; Quinn, P. *International Virtual Observatory Alliance [online]*. The IVOA, [cit. 2017-04-27]. Available from: `http://www.ivoa.net/about/TheIVOA.pdf`

[2] IVOA. What is the IVOA [online]. [cit. 2017-04-27]. Available from: `http://ivoa.net/about/what-is-ivoa.html`

[3] Oracle. *Java Platform, Enterprise Edition; The Java EE Tutorial; Release 7 [online]*. September 2014, [cit. 2014-05-05]. Available from: `https://docs.oracle.com/javaee/7/JEETT.pdf`

[4] Oracle. Java(TM) EE 7 Specification APIs [online]. [cit. 2017-04-28]. Available from: `http://docs.oracle.com/javaee/7/api/`

[5] Koza, J. *Design and implementation of a distributed platform for data mining of big astronomical spectra archives*. Bachelor's thesis, Czech Technical University in Prague,Faculty of Information Technology, Prague, 2015, doi:10.5281/zenodo.44641.

[6] WWW Consorcium. *Extensible Markup Language (XML) 1.0 (Fifth Edition) [online]*. November 2008, [cit. 2017-04-29]. Available from: `http://www.w3.org/TR/REC-xml/REC-xml-20081126-review.html`

[7] Fielding, R. T.; Taylor, R. N. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, May 2002: pp. 115–150, ISSN 1533-5399, doi:10.1145/514183.514185. Available from: `http://doi.acm.org/10.1145/514183.514185`

[8] Harrison, P.; Rixon, G. IVOA Recommendation: Universal Worker Service Pattern Version 1.0. *ArXiv e-prints*, October 2011, `1110.0510`. Available from: `http://adsabs.harvard.edu/abs/2011arXiv1110.0510H`

[9] Coulouris, G.; Dollimore, J.; et al. *Distributed Systems: Concepts and Design (5th Edition)*. Pearson, 2011, ISBN 0132143011.

[10] Tody, D.; Dolensky, M.; et al. IVOA Recommendation: Simple Spectral Access Protocol Version 1.1. *ArXiv e-prints*, March 2012, 1203.5725. Available from: http://adsabs.harvard.edu/abs/2012arXiv1203.5725T

[11] Laurent, M.; Bonnarel, F.; et al. *IVOA Recommendation: DataLink Protocol Version 1.0 [online]*. The IVOA, May 2013, [cit. 2017-04-30]. Available from: http://www.ivoa.net/documents/Notes/DataLink/20130502/NOTE-DataLinkProposal-1.0-20130502.pdf

[12] Palička, A. *Application of Random Decision Forests in Astroinformatics*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2014.

[13] Lopatovský, L. *Application of Self-Organizing Maps in Astroinformatics*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2014.

[14] Mrkva, L. *VO-KOREL, server for astronomical cloud computing*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2012.

[15] Docker Inc. What is Docker [online]. [cit. 2017-05-02]. Available from: https://www.docker.com/what-docker

[16] Docker Inc. What is a Container [online]. [cit. 2017-05-02]. Available from: https://www.docker.com/what-container

[17] The Apache Software Foundation. What is Apache Hadoop? [online]. [cit. 2017-05-02]. Available from: http://hadoop.apache.org/

[18] The Apache Software Foundation. HDFS Architecture Guide [online]. [cit. 2017-05-02]. Available from: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[19] White, T. The Small Files Problem [online]. February 2009, [cit. 2017-05-02]. Available from: http://blog.cloudera.com/blog/2009/02/the-small-files-problem/

[20] The Apache Software Foundation. Spark Overview [online]. [cit. 2017-05-04]. Available from: http://spark.apache.org/docs/latest/

[21] The Apache Software Foundation. Machine Learning Library (MLlib) Guide [online]. [cit. 2017-05-04]. Available from: http://spark.apache.org/docs/1.6.3/mllib-guide.html

[22] Penchikala, S. Big Data Processing with Apache Spark [online]. [cit. 2017-05-04]. Available from: https://www.infoq.com/articles/apache-spark-introduction

[23] Jupyter Team. The Jupyter Notebook [online]. 2015, [cit. 2017-05-04]. Available from: http://jupyter-notebook.readthedocs.io/en/latest/notebook.html

[24] Jupyter Team. What is the Jupyter Notebook? [online]. 2015, [cit. 2017-05-04]. Available from: http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html

[25] Tennyson, J. *Astronomical Spectroscopy: An Introduction to the Atomic and Molecular Physics of Astronomical Spectra (Immperial College Press Advanced Physics Texts)*. Imperial College Press, 2005, ISBN 1860945139.

[26] Allen, S.; Wells, D. MIME Sub-type Registrations for Flexible Image Transport System (FITS). RFC 4047, RFC Editor, April 2005.

[27] Ochsenbein, F.; Williams, R.; et al. IVOA Recommendation: VOTable Format Definition Version 1.3. 2011, arXiv:1110.0524.

[28] Astropy Collaboration; Robitaille, T. P.; et al. Astropy: A community Python package for astronomy. *Astronomy and Astrophysics*, volume 558, Oct. 2013: A33, doi:10.1051/0004-6361/201322068, 1307.6212.

[29] Hunter, J. Matplotlib: A 2D graphics environment. *Computing in Science and Engineering*, volume 9, no. 3, 2007: pp. 99–104, doi:10.1109/MCSE.2007.55, cited By 1106. Available from: https://www.scopus.com/inward/record.uri?eid=2-s2.0-34247493236&doi=10.1109%2fMCSE.2007.55&partnerID=40&md5=29e85ef102f6f3e89c7c074bcf360684

[30] The Tornado Authors. *Tornado Documentation; Release 4.5.1 [online]*. April 2017, [cit. 2017-05-06]. Available from: https://media.readthedocs.org/pdf/tornado/stable/tornado.pdf

[31] Fette, I.; Melnikov, A. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011, http://www.rfc-editor.org/rfc/rfc6455.txt. Available from: http://www.rfc-editor.org/rfc/rfc6455.txt

[32] Project Jupyter team. *JupyterHub Documentation; Release 0.7.2 [online]*. February 2017, [cit. 2017-05-07]. Available from: https://media.readthedocs.org/pdf/jupyterhub/stable/jupyterhub.pdf

[33] National Astronomical Observatories. LAMOST Telescope [online]. 2012, [cit. 2017-05-08]. Available from: http://www.lamost.org/public/instrument?locale=en

[34] The Apache Software Foundation. Apache Hadoop Main 2.7.3 API [online]. 2016, [cit. 2017-05-08]. Available from: https://hadoop.apache.org/docs/stable/api/

[35] The Apache Software Foundation. Apache Avro™ 1.8.1 Documentation [online]. 2016, [cit. 2017-05-08]. Available from: https://avro.apache.org/docs/current/

Appendix A

Acronyms

API Application Programming Interface

CPU Central Processing Unit

CSV Comma-Separated Values

DB DataBase

DR1 Data Release 1

EE Enterprise Edition

EJB Enterprise JavaBean

FITS Flexible Image Transport System

FR Functional Requirement

FTP File Transfer Protocol

GUI Graphical User Interface

HDFS Hadoop Distributed File System

HDUs Header and Data Units

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

IVOA International Virtual Observatory Alliance

JPA Java Persistence API

JPQL Java Persistence Query Language

JSF JavaServer Faces

JSON JavaScript Object Notation

LAMOST Large Sky Area Multi-Object Fibre Spectroscopic Telescope

NFR Non-functional Requirement

ORM Object-Relational Mapping

RAM Random Access Memory

RDF Random Decision Forests

REST Representational State Transfer

SOM Self-Organizing Maps

SPLAT Spectral Analysis Tool

SQL Structured Query Language

SSAP Simple Spectral Access Protocol

SSH Secure Shell

TCP Transmission Control Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

UWS Universal Worker Service

VO Virtual Observatory

XHTML Extensible HyperText Markup Language

XML Extensible Markup Language

XSD XML Schema Definition

YARN Yet Another Resource Negotiator

Appendix B

Contents of enclosed DVD

- `readme.txt` ..... the file with DVD contents description
- `src` ..... the directory of source codes
  - `impl` ..... implementation sources
    - `repositories.txt` ... the file containing list of GitHub repositories
    - `spectra-avro-serializer` ..... Avro serializer tool sources
    - `spectraviewer` ..... `spectraviewer` tool sources
    - `vocloud` ..... VO-CLOUD master server and workers sources
    - `vocloud-authenticator` ..... `VocloudAuthenticator` sources
    - `vocloud-jupyterhub` ..... `vocloud-jupyterhub` sources
    - `vocloud_spark_import` ..... `vocloud_spark_import` sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `zzp.txt` ..... the thesis task in a plain text format

Appendix C

Universal worker XML configuration file schema

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3      targetNamespace="http://vocloud.ivoa.cz/universal/schema"
4      xmlns:tns="http://vocloud.ivoa.cz/universal/schema"
5      elementFormDefault="qualified">
6    <xsd:complexType name="worker">
7      <xsd:sequence>
8        <xsd:element name="identifier" type="xsd:token"/>
9        <xsd:element name="description" type="xsd:string"/>
10       <xsd:element name="restricted" type="xsd:boolean" default=
             "false"/>
11       <xsd:element name="binaries-location" type="xsd:string"/>
12       <xsd:element name="exec-command" type="tns:command-list"/>
13     </xsd:sequence>
14   </xsd:complexType>
15   <xsd:complexType name="command-list">
16     <xsd:sequence>
17       <xsd:element name="command" type="xsd:string" maxOccurs="
             unbounded"/>
18     </xsd:sequence>
19   </xsd:complexType>
20   <xsd:element name="uws-settings">
21     <xsd:complexType>
22       <xsd:sequence>
23         <xsd:element name="vocloud-server-address" type="xsd:
               anyURI"/>
24         <xsd:element name="local-address" type="xsd:anyURI"/>
```

```
25          <xsd:element name="max-jobs" type="xsd:positiveInteger"
                default="4"/>
26          <xsd:element name="description" type="xsd:string"/>
27          <xsd:element name="default-destruction-interval" type="
                xsd:positiveInteger" minOccurs="0"/>
28          <xsd:element name="max-destruction-interval" minOccurs="
                0" type="xsd:positiveInteger"/>
29          <xsd:element name="default-execution-duration" default="
                3600" minOccurs="0" type="xsd:positiveInteger"/>
30          <xsd:element name="max-execution-duration" default="3600
                " minOccurs="0" type="xsd:positiveInteger"/>
31          <xsd:element name="workers">
32            <xsd:complexType>
33              <xsd:sequence>
34                <xsd:element name="worker" maxOccurs="unbounded"
                      minOccurs="0" type="tns:worker"/>
35              </xsd:sequence>
36            </xsd:complexType>
37          </xsd:element>
38        </xsd:sequence>
39      </xsd:complexType>
40    </xsd:element>
41  </xsd:schema>
```

Appendix D

Spark worker XML configuration file schema

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3      targetNamespace="http://vocloud.ivoa.cz/spark/schema"
4      xmlns:tns="http://vocloud.ivoa.cz/spark/schema"
5      elementFormDefault="qualified">
6    <xsd:complexType name="params-list">
7      <xsd:sequence>
8        <xsd:any minOccurs="0" maxOccurs="unbounded"
             processContents="skip"/>
9      </xsd:sequence>
10   </xsd:complexType>
11   <xsd:complexType name="environment">
12     <xsd:sequence>
13       <xsd:any minOccurs="0" maxOccurs="unbounded"
             processContents="skip"/>
14     </xsd:sequence>
15   </xsd:complexType>
16   <xsd:complexType name="worker">
17     <xsd:sequence>
18       <xsd:element name="identifier" type="xsd:token"/>
19       <xsd:element name="description" type="xsd:string"/>
20       <xsd:element name="submit-params" type="tns:params-list"
             minOccurs="0"/>
21       <xsd:element name="submit-target" type="xsd:string"/>
22     </xsd:sequence>
23   </xsd:complexType>
24   <xsd:element name="uws-settings">
```

```
25      <xsd:complexType>
26        <xsd:sequence>
27          <xsd:element name="vocloud-server-address" type="xsd:
                anyURI" minOccurs="0"/>
28          <xsd:element name="local-address" type="xsd:anyURI"/>
29          <xsd:element name="spark-executable" type="xsd:string"/>
30          <xsd:element name="hadoop-default-fs" type="xsd:string"/
                >
31          <xsd:element name="max-jobs" type="xsd:positiveInteger"
                default="4"/>
32          <xsd:element name="description" type="xsd:string"/>
33          <xsd:element name="environment" type="tns:environment"/>
34          <xsd:element name="submit-params" type="tns:params-list"
                 minOccurs="0"/>
35          <xsd:element name="workers">
36            <xsd:complexType>
37              <xsd:sequence>
38                <xsd:element name="worker" maxOccurs="unbounded"
                      minOccurs="0" type="tns:worker"/>
39              </xsd:sequence>
40            </xsd:complexType>
41          </xsd:element>
42        </xsd:sequence>
43      </xsd:complexType>
44    </xsd:element>
45  </xsd:schema>
```

Appendix E

Master server README file

E.1 VO-CLOUD Master server

E.1.1 Requirements

- JDK 7+
- Application server supporting Java EE 7 with EJB container support (Wildfly, Glassfish, ...)
- Database (PostgreSQL, MySQL, ...)
- Maven tool for project building

E.1.2 Install guide

For instance I will use Debian amd64 with Wildfly 8.2 application server, JDK 8 and PostgreSQL 8.4.

1. Install JDK 8
   - Download JDK from `http://www.oracle.com/technetwork/java/javase/downloads/index.html` in zip file form, for example `jdk-8u45-linux-x64.tar.gz`
   - Extract archive to `/usr/lib/jvm`
   - Setup enviroment variables for Java – add these lines to the end of `/etc/profile`

     ```
     export JAVA_HOME=/usr/lib/jvm/jdk1.8.45
     export PATH=$JAVA_HOME/bin
     ```

2. Install WildFly 8.2.0
   - Download zip from `http://wildfly.org/downloads/`
   - Extract archive to the `/usr/local`
   - In the newly extracted WildFly directory execute `bin/add-user.sh` and setup a new WildFly administering user

3. Start Wildfly by executing `bin/standalone.sh`. Server should successfully start. If everything went OK:
   - Server is running on `http://localhost:8080/`
   - Admin console on `http://localhost:9990/`
4. Install and configure PostgreSQL database server
   - Install PostgreSQL using `apt-get install postgresql`
   - Log in as a postgres user `su - postgres` and start client command line `psql template1`
   - Type in following commands to setup database for vocloud:

     ```
     CREATE USER vocloud WITH PASSWORD 'vocloud';
     CREATE DATABASE vocloud;
     GRANT ALL PRIVILEGES ON DATABASE vocloud TO vocloud;
     ```

     Note: You should really not use the same password as username. Do not forget to change it!

   It is also possible to use Docker technology to install PostgreSQL inside a Docker container. To do this, execute the following command:

   ```
   docker run --name db -d -p 5432:5432
              -e POSTGRES_USER=<username>
              -e POSTGRES_PASSWORD=<password>
              postgres
   ```

5. Configure database resource in WildFly:
   - Log into the WildFly admin console at `http://localhost:9990/`
   - Type in credentials of administrating user
   - Download JDBC for PostgreSQL from `https://jdbc.postgresql.org/`
   - In the admin console navigate to Deployments section
   - Click Add
   - Select downloaded JDBC `.jar` file and click OK
   - Enable newly uploaded JDBC driver
   - Navigate to a Configuration tab
   - Select Datasources
   - Click Add and insert following values:
     - `Name: VocloudDS`
     - `JNDI Name: java:jboss/datasources/vocloud`
   - Click Next
   - Select the newly deployed PostgreSQL JDBC driver
   - Click Next
   - Insert following values:
     - `Connection URL: jdbc:postgresql://localhost:5432/vocloud`

    - Username: <your-database-username>
    - Password: <your-database-password>
  - Click Done
  - Enable newly created VocloudDS

  Datasource can be tested in the section Connection>Test connection – ping test should be successful.
6. Configure e-mail resource in WildFly:
   It is necessary to have an email address that serves as the source of emails sent by vocloud. For instance, I will use address `vocloud@vocloud.org` where SMTP is running on port 465 and the host address of the SMTP server is `smtp.vocloud.org`.

   - Navigate to Configuration section
   - Select Socket Binding
   - Click View on standard-sockets
   - Select Outbound Remote section
   - Click Add and insert:
     - Name: vocloud-smtp
     - Host: smtp.vocloud.org
     - Port: 465
   - Click Save
   - Navigate to the Mail subsystem section
   - Click Add and insert:
     - JNDI Name: java:jboss/mail/vocloud-mail
   - Click View on the newly created mail session
   - Click Add and insert:
     - Socket binding: vocloud-smtp
     - Type: smtp
     - Username: <username-to-the-email-server>
     - Password: <password-to-the-email-server>
   - Check Use SSL (if the port is 465)
   - Click Save

7. Configure security in WildFly

   - Navigate to Security Domains in Configuration section
   - Click Add and insert:
     - Name: VocloudSecurityDomain
   - Click Save
   - Click View on the newly created security domain
   - Click Add and insert:
     - Code: Database
     - Flag: required

- Click Save
- Now click on the newly created Login module
- Click on Module Options
- Add the following key=value pairs:
  - `dsJndiName` = `java:jboss/datasources/vocloud`
  - `principalsQuery` = `select pass from useraccount where username=?`
  - `rolesQuery` = `select groupName, 'Roles' from useraccount where username=?`
  - `hashAlgorithm` = `SHA-256`
  - `hashEncoding` = `hex`

8. Create master server's `vocloud.war` package

   - Navigate to the VO-CLOUD's master server application's directory
   - Execute `mvn package`
   - Package should be now created in `target/vocloud.war`

9. Deploy `vocloud.war` package to the WildFly server

   - Log into the WildFly's admin console
   - Navigate to section Deployments
   - Click Add
   - Select `vocloud.war` file
   - Submit
   - Enable the newly deployed application

   VO-CLOUD master server should now be running at `http://localhost:8080/vocloud`

10. Create admin account

    - Open VO-CLOUD master server application in web browser
    - Click Register
    - Register a new account with username `admin`

    This account now has administrator privileges.

Appendix F

Spark worker README file

F.1 Spark worker

F.1.1 Requirements

- JDK 7+
- Java application server supporting Java servlet technology (Tomcat, WildFly, ...)
- Maven tool (if building is necessary)
- Spark deployable application for each Spark worker type

F.1.2 Install guide

For instance I will use Debian amd64 with WildFly 8.2 application server, JDK 8 and Maven 3.1.

1. Install JDK 8
   - Download JDK from `http://www.oracle.com/technetwork/java/javase/downloads/index.html` in zip file form, for example `jdk-8u45-linux-x64.tar.gz`
   - Extract archive to `/usr/lib/jvm`
   - Setup environment variables for Java – add these lines to the end of `/etc/profile`:

     ```
     export JAVA\HOME=/usr/lib/jvm/jdk1.8.45
     export PATH=$JAVA_HOME/bin
     ```

2. Install WildFly 8.2.0
   - Download zip from `http://wildfly.org/downloads/`
   - Extract archive to the `/usr/local`
   - In the newly extracted wildfly directory execute `bin/add-user.sh` and setup a new WildFly administering user.

3. Start Wildfly by executing bin/standalone.sh. Server should successfully start. If everything went OK:
   - Server is running on `http://localhost:8080/`
   - Admin console on `http://localhost:9990/`
4. Configure spark-worker configuration file (optional step if you want another configuration that it is in pre-built archive)
   - Download sources for spark-worker
   - Go to `src/main/resources/`
   - Adjust `uws-config.xml` file
   - Go back to sources root
   - Execute command `mvn package`
   - Worker is compiled and the deployable archive is created in `target/spark-worker.war`
5. Deploy spark worker to Wildfly
   - Open WildFly admin console on `http://localhost:9990/`
   - Login with the credentials of administrating user
   - Navigate to Deployments section
   - Click Add
   - Select deployable `spark-worker.war` archive
   - Click OK
   - Enable the newly deployed application

   UWS service should now be running on `http://localhost:8080/spark-worker/uws`

Note: This is only description of spark-worker application which serves as the mediator between the master server and spark submit script. In order to make a worker fully functional you have to set proper configuration values into the UWS configuration file matching your running Spark instance.

F.1.3 Configuration file description

Configuration of the Spark worker is define by the xml file containing all necessary information for the Spark worker deployment. The schema of the XML configuration file is specified by `XSD` file and is located in `src/main/resources/configSchema.xsd`.

Let us explain the configuration file format on the example:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ns:uws-settings
3         xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
4         xmlns:ns='http://vocloud.ivoa.cz/spark/schema'
```

```
5         xsi:schemaLocation='http://vocloud.ivoa.cz/spark/schema
              configSchema.xsd'>
6     <ns:vocloud-server-address>http://localhost:8080/vocloud-
          betelgeuse</ns:vocloud-server-address>
7     <ns:local-address>http://localhost:8080</ns:local-address>
8     <ns:spark-executable>/opt/spark/bin/spark-submit</ns:spark-
          executable>
9     <ns:hadoop-default-fs>hdfs://betelgeuse:9000</ns:haddop-
          default-fs>
10    <ns:max-jobs>4</ns:max-jobs>
11    <ns:description>Spark UWS worker</ns:description>
12    <ns:environment>
13        <HADOOP_CONF_DIR>/opt/hadoop/etc/hadoop</HADOOP_CONF_DIR
              >
14    </ns:environment>
15    <ns:submit-params>
16        <conf name="spark.driver.maxResultSize">12g</conf>
17        <conf name="spark.yarn.executor.memoryOverhead">4096</
              conf>
18        <master>yarn</master>
19        <driver-memory>4g</driver-memory>
20        <deploy-mode>client</deploy-mode>
21        <num-executors>5</num-executors>
22        <executor-cores>3</executor-cores>
23        <executor-memory>4g</executor-memory>
24    </ns:submit-params>
25    <ns:workers>
26        <ns:worker>
27            <ns:identifier>spark-preprocessing</ns:identifier>
28            <ns:description>Spark preprocessing</ns:description>
29            <ns:submit-params>
30                <packages>com.databricks:spark-avro_2.10:2.0.1</
                      packages>
31                <py-files>
32                    /home/hadoop/workflow-test/preprocessing/
                          vocloud_spark_import/dist/
                          vocloud_spark_preprocess-0.1.0-py2.7.egg
33                </py-files>
34            </ns:submit-params>
35            <ns:submit-target>
36                /home/hadoop/workflow-test/preprocessing/
                      vocloud_spark_import/bin/vocloud_preprocess.py
37            </ns:submit-target>
38        </ns:worker>
```

```
39      </ns:workers>
40  </ns:uws-settings>
```

- `vocloud-server-address` [optional] – Specifies URL address to the deployed vocloud server. This URL is necessary when the worker needs to download some data from the vocloud server. Note that in order to do so you will have to arrange the network visibility from the worker to master server and vice versa.
- `local-address` – Hostname URL to the worker server from the master server point of network view.
- `spark-executable` – Path to the `spark-submit` script on the filesystem.
- `hadoop-default-fs` – URL locator of the HDFS filesystem.
- `max-jobs` – Maximum count of jobs that this worker allows to be run concurrently. Note that Spark execution manager (e.g. YARN) can have additional restrictions to the count of jobs/resources requirement.
- `description` – Description of this UWS worker.
- `environment` [optional] - The sequence of optional tags settings the environment variables to be passed to the `spark-submit` script. For this current instance the `HADOOP_CONF_DIR` variable is set to be able to use `--master yarn` parameter properly.
- `submit-params` [optional] – This complex tag can be either in the root `uws-settings` tag or in the `worker` tag (see later). It specifies implicit parameters to be passed to the `spark-submit`. Parameters from the root tag can be overriden by the parameters specified in the `worker` tag and both parameter specification can be overridden by the parameters specified in the job's configuration file. Parameters are specified in the following format:

  ```
  <param-name>param-value</param-name>
  ```

  This statement is translated to `--param-name param-value` in the `spark-submit` script. Note: `<conf>` tag have a special form:

  ```
  <conf name="conf-name">conf-value</conf>
  ```

  that is translated to `--conf conf-name=conf-value`. There can be multiple `<conf>` tags.
- `workers` – Contains sequence of `<worker>` tags.
- `worker` – Contains configuration for the single worker type instance. It contains following tags:

- `identifier` – Identification if the worker. Must not contain space character.
- `description` – Description of the worker.
- `submit-params` – Same as in the root tag.
- `submit-target` – Path to the file that should be passed to the `spark-submit` script.

F.1.4 Job configuration

The following JSON is an example of the spark job configuration.

```
{
    "download_files": [
        {
            "urls": [
                "vocloud://DATA/allspec-ond700-prep/prep.csv",
                "vocloud://DATA/allspec-ond700-prep/prep2.csv"
            ],
            "folder": "/user/test/input1/"
        },{
            "urls": ["vocloud://DATA/folder/st.csv"],
            "folder": "/user/test/input2/"
        }
    ],
    "spark_params": {
        "num-executors": "2",
        "executor-cores": "4",
        "conf": {
            "spark.driver.maxResultSize": "12g",
            "spark.yarn.executor.memoryOverhead": "4096"
        }
    },
    "job_config": {
        "dataset": "hdfs:///user/workflow-test/lof-input/
           preprocessed.csv",
        "min_pts": 15,
        "output": "hdfs:///user/workflow-test/output/lof_kepler-
           out.csv"
    },
    "copy_output": [
        {
            "path": "/user/workflow-test/output/lof_kepler-out.
               csv",
            "output_name": "preprocessed.csv",
            "merge_parts": true
```

```
32          }
33      ]
34  }
```

Most of the configuration JSON file is optional. The only mandatory part is `job_config` object part that specifies the configuration file of the Spark application. The content of this object will be written in the temporary file and the path will be passed to the `spark-submit` script as the last parameter. If the configuration does not contain the `copy_output` item, the whole configuration file is considered as the config for the `spark-submit` script – in this case it would be:

```
1  {
2      "dataset": "hdfs:///user/workflow-test/lof-input/
           preprocessed.csv",
3      "min_pts": 15,
4      "output": "hdfs:///user/workflow-test/output/lof_kepler-out.
           csv"
5  }
```

- `download_files` – Specifies files that should be downloaded from the vocloud filesystem (or some other URL) and saved to the hdfs to the specified path before the spark job itself is executed. It must contain array where each item is object containing two mandatory items:
  - `urls` – Array of string containing the remote file path. It supports `http`/`https` protocol and if the path has scheme `vocloud` the files are downloaded from the vocloud's filesystem. Note: in order to do so it is necessary that worker has properly set the path to the vocloud server and the server is directly visible on the network.
  - `folder` – Target path on HDFS where the files specified in the `urls` part should be saved. Save fails if the path already exists.

  Note: In order to be able to download files into the HDFS it is necessary that the worker application has properly setup write permission to the HDFS. This is usually done by adding user under which the worker application is started to the `supergroup` group.
- `spark_params` – Allows user to override any parameters passed to the `spark-submit` script. It contains JSON object where each item `"name": "value"` is translated to the parameter `--name value`. The only exception is an item named `conf` that if present must contain additional JSON object where each item `"name": "value"` is translated to `--conf name=value`. Parameters here can override the default one specified in the xml configuration file.
- `job_config` – Specifies the configuration for the Spark job itself. See above.

- `copy_output` – Allows user to obtain files from the hdfs back to the vo-cloud. It must contain JSON array containing JSON objects containing following items:
  - `path` – Path to the file or folder on the HDFS.
  - `output_name` [optional] – Name of the copied file or directory. If not present, tries to find out the file/folder name from the `path` parameter.
  - `merge_parts` [optional] – Spark jobs usually produce results as folder containing `part_xxx` files. If this item is set to `true` the worker merges these parts together to produce a single file. This item is optional, default value is set to `false`.