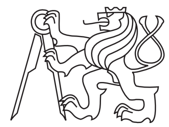
Insert here your thesis' task.

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



Bachelor's thesis

Design and implementation of a distributed platform for data mining of big astronomical spectra archives

Jakub Koza

Supervisor: RNDr. Petr Škoda, CSc.

12th May 2015

Acknowledgements

I would like to thank my supervisor, RNDr. Petr Škoda, CSc., for his help and for giving me this opportunity, and to Lumír Mrkva, the author of the original VO-CLOUD system, for support in the beginning of my implementation. We also acknowledge support of grant GAČR 13-08195S

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 12th May 2015 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Jakub Koza. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Koza, Jakub. *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, 2015.

Abstrakt

Cílem této bakalářské práce je rozšířit stávající distribuovaný systém VO-CLOUD, který poskytuje uživatelům prostor a výkon pro vytváření výpočetně náročných astronomických experimentů skrze rozhraní webového prostředí. Výsledný systém je schopný získávat vstupní data přímo z astronomických archívů pomocí speciálních astronomických protokolů SSAP a DataLink. Dále je schopný delegovat výpočty na distribuovaný výpočetní stroj a je schopný vizualizovat výsledky výpočtů uživateli přímo ve webovém prostředí.

Klíčová slova Virtuální Observatoř, SSAP, DataLink, UWS, Java EE, astroinformatika

Abstract

The aim of this bachelor's thesis is to extend current distributed system VO-CLOUD capable of providing users with a storage and computability to conduct astronomical experiments in a web based environment. The resulting system is capable of downloading input data directly from astronomical archives by using special astronomical protocols SSAP and DataLink. The system is able to delegate computations on a distributed computational machine and it

is able to visualise computational results directly in the web based environment.

Keywords Virtual Observatory, SSAP, DataLink, UWS, Java EE, astroinformatics

Contents

Introduction **1**

1 Analysis of the current solution **3**
1.1 Architecture . . . 3
1.2 Technologies . . . 4
1.3 Workflow . . . 12

2 Requirements analysis **15**
2.1 New concepts . . . 15
2.2 Functional requirements . . . 17
2.3 Non-functional requirements . . . 21

3 Worker realisation **23**
3.1 Universal worker concept . . . 23
3.2 Universal worker workflow . . . 26
3.3 Files downloading feature . . . 27
3.4 Visualisation . . . 29

4 Master server realisation **31**
4.1 VO-CLOUD storage . . . 31
4.2 Remote download feature . . . 32
4.3 SSAP and DataLink client . . . 32
4.4 Preprocessing . . . 34
4.5 Workers management . . . 34
4.6 Jobs load balancing . . . 36

5 Future development **37**

Conclusion **39**

Bibliography **41**

A Acronyms **43**

B Contents of enclosed DVD **45**

C Universal worker XML configuration file schema **47**

D Master server README file **49**

E Universal worker README file **53**

List of Figures

1.1 Deployment diagram of the current solution 4
1.2 Example of Singleton EJB using timer service 9
1.3 Relations of UWS objects 11
1.4 State machine of UWS job's execution phase 13

3.1 Universal worker configuration file 25
3.2 Configuration JSON file 27
3.3 JSON describing VO-CLOUD storage directory 28

4.1 Page with a directory listing 33
4.2 Class diagram of JPA Entity classes 35

List of Tables

1.1 Often used actions on UWS REST binding 12

Introduction

The research of the night sky have drastically changed in the last few years thanks to modernization of information technologies. Whereas in the past an astronomer had to wait even a couple months to access the telescope, today he has almost immediate access to data thanks to system called Virtual Observatory (VO), 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].

VO-CLOUD (originally called VO-KOREL which was extended by various data mining capabilities) is the system implementing basic principles and concepts of Virtual Observatory, where astronomers can conduct their experiments with computationally intensive data mining algorithms and visualize them in familiar and friendly graphical interface [2]. The significant disadvantage of this particular distributed system is the fact that data that usually may be quite big for the upcoming experiment have to be prepared in advance in the local storage of the experimenter and uploaded to the VO-CLOUD server each time during the experiment creation.

The aim of this work is to analyse workflow as well as implementation of contemporary VO-CLOUD server and its distributed workers (execution units of experiments), and perform design and implementation of the new version of VO-CLOUD server that will be capable of directly downloading data from remote resources using protocols like HTTP, FTP as well as IVOA specific protocol such as SSAP, DataLink. Downloaded data will be available for submission within experiment to the assigned workers for computation and then the results will be automatically downloaded back to VO-CLOUD server for possible visualization.

Chapter **1**

Analysis of the current solution

In this section I would like to briefly describe the state of currently implemented state of VO-CLOUD and the technologies involved.

1.1 Architecture

VO-CLOUD is distributed system which means that it is consisted of hardware or software components located at networked computers that communicate and coordinate their actions only by passing messages to achieve their task [3]. In the case of the VO-CLOUD system is composed of the following parts:

- One master server capable of communication with the experimenting user
- Several distributed nodes called workers that contain:
  - Binary files describing the long time running computational process
  - Simple application with the ability to communicate with the master server and to start computational process and to dispatch required data to it

The master server is the most important component of VO-CLOUD server. Its main purpose is to provide web interface for communication with an experimenting user and to delegate requested experiment computations to the chosen worker. Master server stores the information about all experiments in a database and periodically checks the state of experiments to see if their execution is already finished. In the positive case, results are downloaded from worker back to the master server and deleted on the worker side.

Workers in the current solution are distinguished by the type of the experiment computation they can execute. For example one type of worker could execute the process performing Random Decision Forests method used

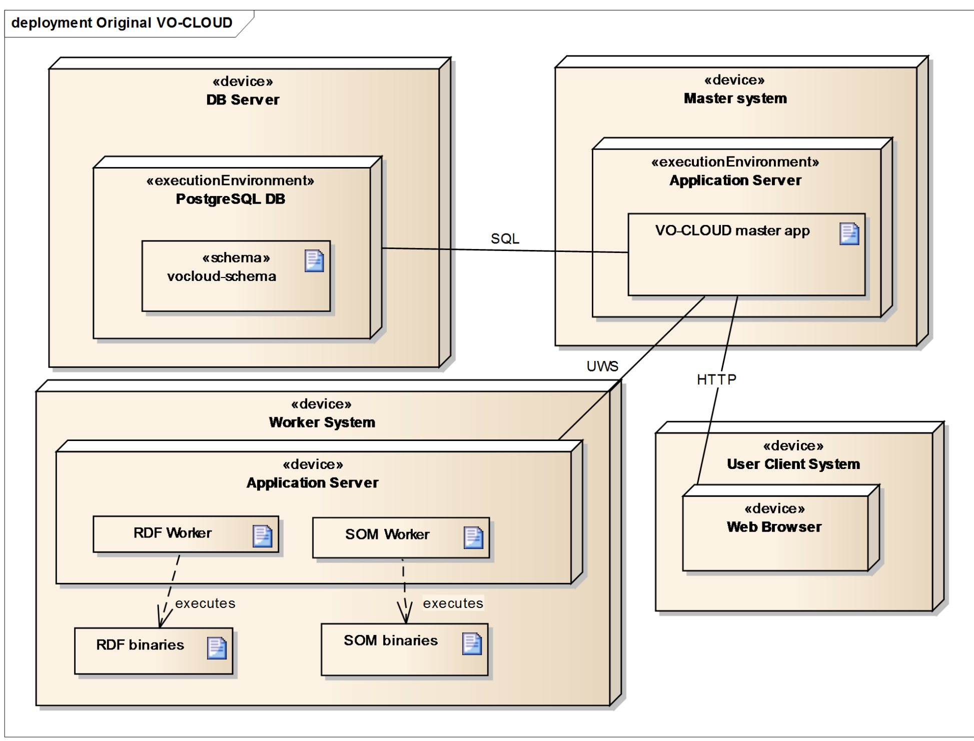


Figure 1.1: Deployment diagram of the current solution

to data mine information from passed astronomical spectrum data [4]. Generally, every worker consists of binary files which are executed over data as the new long time running process, and the lightweight application which manages the queue of executing jobs and starts the long running computational process. Although the technology of the computational process is not restricted, usually a program written in Python is used on workers.

VO-CLOUD deployment example is described by the deployment diagram in Figure 1.1. In this example there is only one worker machine (one Java Application Server) where two different applications are deployed. The first can dispatch computations to Random Decision Forests method binaries, the second one can dispatch to Self-organizing map method binaries. Communication between workers and master server is maintained via specialized Universal Worker Service protocol (UWS) which is described in the further section of this chapter.

1.2 Technologies

Both master server and applications starting computational process on workers are implemented in Java EE Programming Language Platform.

> ”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.” [5]

Master server itself uses significant amount of technologies specified in the Java EE platform specification.

1.2.1 Java Persistence API

There is numerous information on master server that is required to be stored in the database such as user accounts, list of available workers, history of experiments executions and many more. Java EE provides API called Java Persistence API which allows to automatically map Java objects to the relational database such as MySQL or PostgreSQL.

> ”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.” [5]

Java objects that should be mapped into the relational database are in the context of Java Persistence API called Entity classes. These Entity classes are mapped into tables in the database and instance variables are mapped as columns of these tables. Whole Entity class and its instance variables can be annotated by special JPA annotations to achieve demanded behaviour of the object/relational mapping, e.g., changing name of the columns, adding database constraints and so on. It is also possible to put settings to configuration file instead of annotating Java objects but the annotation way seems to be more intuitive.

1.2.2 JavaServer Faces

JavaServer Faces is the main technology used on the master server to communicate with a user through the web interface. It is a server-side component framework for building Java technology–based web applications [5].

> ”One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behaviour and presentation for web applications.” [5]

Source code of the presentation tier in JavaServer Faces technology is divided to XHTML pages and Managed Beans. Each XHTML file represents visual side of one page in the standardized format XML [6]. There are many tags that can be used inside these files. Whereas standard HTML tags are directly used as output for a user, the file is mostly composed of special JSF tags with

special meaning. These tags add functionality beyond static HTML pages and 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.

Managed Bean is the special type of Java class. By the JSF specification [5] Java classes used as Managed Bean must have defined non-parametric constructor to be able to dynamically instantiate them through the Java Reflection API [7]. Classes must also have specified name that will be used for identification of the Managed Bean in the Expression Language in XHTML files. Also, classes require to have defined scope. "Scope defines how application data persists and is shared." [5] The most commonly used scopes in JavaServer Faces applications are Request and Session scopes. Request scope stores data only during a single HTTP request, whereas Session scope stores across multiple HTTP requests and it is always bound to the specific user [5]. Both Expression Language name mapping and scope could be specified either with Java annotations or in JavaServer Faces configuration XML file. While JSF XHTML files describe mostly visual side of the page rendered to the user, Managed Java Beans define properties and functions for UI components described by the XHTML pages.

1.2.3 Java Servlet Technology

Java Servlet Technology is defined in the Java EE platform specification and it is used on both master server and workers. The Java EE specification says:

> "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." [5]

In the implementation of VO-CLOUD system only HTTP servlets are used. To implement such a servlet it is necessary to extend Java class `HttpServlet` placed in `javax.servlet.http` package. Every HTTP request aiming the servlet is dispatched to one of the servlet's inherited method depending on the HTTP method used in the client's request. For example HTTP POST method is dispatched to `doPost` servlet method, HTTP GET to `doGet` method and so on. These servlet methods can be simply overridden in the `HttpServlet` subclass to achieve desired functionality. List of all possible HTTP version 1.1 methods and their explanation is described in RFC 2616 [8]. Finally, the servlet must be registered to the demanded context path of the resulting web application. E.g., if the servlet was mapped to the path `/files/image.jpg` and the web application was deployed on the path `http://localhost/vocloud`, HTTP request with method GET to URL address `http://localhost/`

`vocloud/files/image.jpg` would be dispatched to the servlet's method `doGet`. Registration can be done either with Java annotation or in XML configuration file.

1.2.4 Enterprise Java Beans

Enterprise Java Bean (EJB) is a powerful technology and it is part of the specification of Java EE platform. EJB is a server-side component that encapsulates the business logic of the application, i.e., it contains the code that fulfils the purpose of the application [5]. There are many benefits to using EJB in the application, such as automatic transaction management, concurrency management and security authorization. Moreover, EJB technology provides API for asynchronous method invocation and possibility to schedule server-side activities in desired times.

Enterprise Beans run in the EJB container, a runtime environment within a compliant application server. Master server uses the EJB technology and therefore it is necessary to deploy the master server application to an application server supporting EJB such as GlassFish Server[1] or WildFly Server[2]. Worker application can be deployed on these servers too, nevertheless, thanks to the fact that it does not use EJB technology but only Java Servlet Technology, it can be deployed to application servers without the EJB container such as Apache Tomcat[3].

Enterprise Beans can be divided to two main following types:

- Session Beans
- Message-driven Beans

Session bean's main task is to encapsulate business logic that can be invoked programmatically by calling its methods [5]. These tasks dispatched to session bean by client are then executed on the server side inside the EJB container and so client is shielded from complexity of the business methods. There are many ways how a client can invoke EJB method of a session bean. For example, methods can be invoked remotely by using Java Remote Method Invocation technology.

> "The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language." [9]

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

This way of invocation could be used if the graphical user interface would be implemented as a Java desktop application and not a web application. In the master server methods of EJB session beans are invoked locally from Managed Beans of the JavaServer Faces framework. Managed JSF Beans use dependency injection technique to acquire instances of EJB Session Beans.

There are three types of Enterprise Session Beans.

- *Stateful Session Beans* are very similar to Session scope defined in JSF specification. Instance variables of stateful session bean are always bounded to a unique client that is using them. Nevertheless, in the case of the master server application, the Managed JSF Bean, into which the stateful session bean is injected, is considered as the client. Lifetime of such a stateful bean is determined by the scope of Managed Bean that the stateful bean is injected into. Stateful session beans are not much useful in the web applications since JSF provides possibility to keep information about users' sessions in Session scope annotated Managed Beans.

- *Stateless Session Beans* are not bound to a specific client. EJB container creates a pool of a few stateless bean objects and when a client needs to invoke method, one stateless bean object is pulled out of a pool and offered to the client. When message invocation ends the stateless bean object is returned back to the container's pool. For the subsequent method call the container can offer different instance of stateless bean and so it is not guaranteed that instance variables will be kept. "Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client." [5] Without any configuration every EJB method invocation is automatically wrapped in transaction and so a stateless session bean is often used for storing data to a database through Java Persistence API.

- *Singleton Session Beans* were introduced in the EJB specification version 3.1. They are instantiated only once per application and exists for the whole lifecycle of the application. [5] They are used in situations where it is necessary to share the same information among multiple clients. Also, they are often used in conjunction with timer service interface to compel EJB container to invoke a singleton's method in requested time point. Figure 1.2 shows simple example of singleton session bean with method `doSomeWork` that is invoked by EJB container every thirty seconds. Practically, the master server uses singleton timer service to periodically check experiments running on workers to see if they are already completed.

Message-driven Beans are special kind of enterprise beans that allow Java EE application to process messages asynchronously. They use Java Message

```
import javax.ejb.Singleton;
import javax.ejb.Schedule;
import javax.ejb.Startup;

@Startup @Singleton
public class SchedulerBean {

  @Schedule(second = "*/30", minute = "*", hour = "*",
            persistent = false)
  public void doSomeWork() {
    //called every 30 seconds
  }
}
```

Figure 1.2: Example of Singleton EJB using timer service

Service API (JMS), a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication [5]. Message-driven bean simply acts as a JMS message listener where the source of messages can be any application capable of creating and sending JMS message to a message queue created by an application server. Nevertheless, the master server does not use message-driven beans. Asynchronous method invocation in the master server is performed in session beans by using special method annotation `@Asynchronous` introduced in EJB 3.1. EJB container automatically calls them in the separated thread instead of using main invocation thread.

1.2.5 Universal Worker Service

The Universal Worker Service (UWS) is IVOA recommendation that defines how to manage asynchronous execution of jobs on a service [10]. The International Virtual Observatory Alliance (IVOA) is an organisation that focuses on the development of standards and recommendations that are needed to make Virtual Observatory system possible.

Simple web services are synchronous and stateless. Synchronous means that client waits for the end of execution of the request. If the client disconnects during execution from the service provider there is no reason to continue in the execution and the activity is abandoned. Stateless service means that service does not remember results of a previous activity [10]. Synchronous stateless services work well when following two criteria apply.

1. The service activity duration is short enough for client to maintain the connection with the service.

2. Size of parameters passed to the service and size of service results are small enough to be sent via data channel in reasonable time.

In astroinformatics these two criteria are often not fulfilled. In these cases it is necessary to use a service which is asynchronous and stateful.

Asynchronous service means that client can make many separated request to the service in the course of one activity. Activity on the side of service provider can last even for days. Services that are asynchronous are almost always stateful because naturally the service with a long running activity have to remember information about it and the service often provides interface to query the information out.

"The Universal Worker Service (UWS) pattern defines how to build asynchronous, stateful, job-oriented services." [10] Job in the context of UWS is representing work that should be executed on the side of a service provider. Job-oriented service means that the state is always peculiar to one specific job and the job is always owned by one client.

A UWS is consisted of a set of objects that can be read and written to in order to control jobs [10].

- *Job List* – the outermost object; contains all the other objects
- *Job* – object containing state of one job
- *Phase* – object describing execution phase of the job
- *RunID* – identifier of a job in the job list that should be unique
- *Owner* – object representing identifier of a creator of the job
- *Execution Duration* – the duration for which a job shall run in seconds; if exceeded, job is aborted
- *Destruction Time* – absolute time when the job shall be destroyed
- *Quote* – a UWS service prediction when the job is likely to complete
- *Error* – human-readable error message if the underlying job failed
- *Result List* – list of Result objects produced by the service
- *Parameter List* – list of parameters passed by the client to the service

The relations between individual UWS objects are described in Figure 1.3. In order to create a usable service the objects must be exposed in a particular interface which can be addressed over a particular transport mechanism – this combination is known as a "binding". [10] The most often used style binding for this purpose is REST binding. Representational State Transfer (REST) is a software architecture style consisting of guidelines for creating scalable web

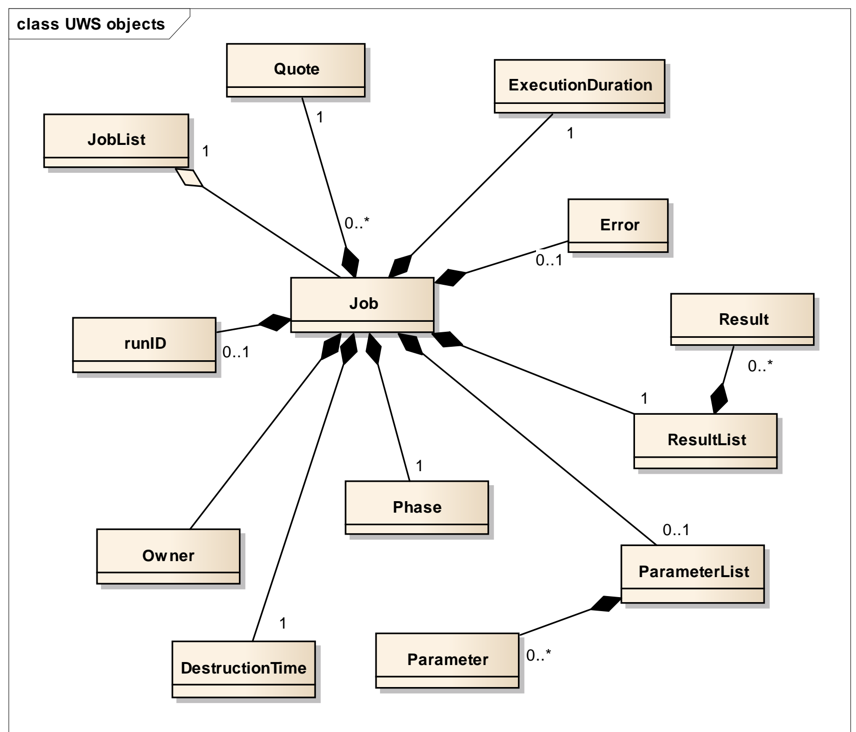


Figure 1.3: Relations of UWS objects

services built on the HTTP protocol. [11] Each of the UWS objects is mapped to a specific URI and for each URI the HTTP method GET can be used to fetch representation of that resource either in XML format (if the object is a container for other objects) or in a textual representation. UWS actions uses the same binding, nevertheless usually the HTTP method POST is used. A few useful examples of a UWS binding can be seen in Table 1.1.

It is also necessary to briefly explain the important property of a job – Execution Phase. "The job is treated as a state machine with the Execution Phase naming the state." [10] The most important phases are explained below

- PENDING – It is the first state of a job when it is created. This phase means that the job is accepted by the service but not yet started by the client.

- QUEUED – The job is started but the service has not yet assigned it to a processor.

- EXECUTING – The job has been assigned to a processor and it is now being executed.

Table 1.1: Often used actions on UWS REST binding

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

- COMPLETED – The job was successfully completed and its results may be collected.
- ERROR – The job failed to complete.
- ABORTED – The job has been manually aborted by the client.

Transitions between individual execution phases of a job are described in the state machine in Figure 1.4.

1.3 Workflow

This section briefly describes usual steps necessary to be performed by a user to create new experiment computation.

1. User prepares data on the local storage of his computer.
2. User creates configuration file that is peculiar to computational method chosen in one of the next steps.
3. User packages prepared data together with the configuration file in ZIP archive file.
4. User logs into the VO-CLOUD system through its web user interface with his credentials.
5. User selects *Create new job* option and he chooses the requested computational method.

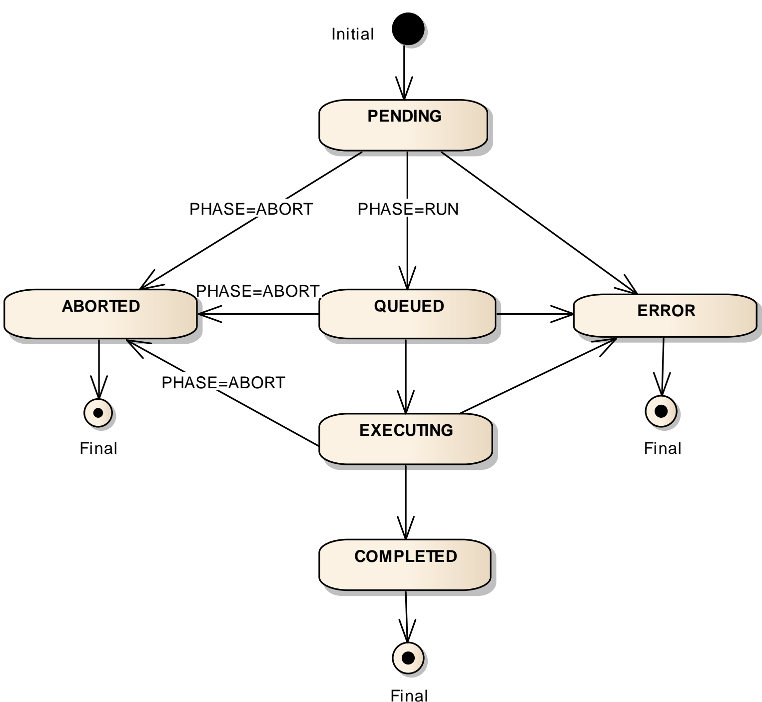


Figure 1.4: State machine of UWS job's execution phase

6. User describes the new experiment and he uploads prepared ZIP archive file with data and configuration file.

7. Optionally, user can edit archived configuration file through the web interface.

8. User sends new experiment to the execution queue for computation.

When the new job is enqueued master server chooses a suitable worker that is capable of computation the specified experiment type. Steps of the communication between the master server and the dedicated worker is following.

1. Master server publishes received ZIP archive file through HTTP URI that is reachable by the dedicated worker.

2. Master server sends "create new job" UWS request to the UWS service running on the worker. In case the user initiating experiment wants to run job immediately, master server sends PHASE=RUN UWS request to the worker. Address of the ZIP archive published through HTTP URI is passed as parameter in the UWS request.

3. Worker puts newly created job to its execution queue. If job was dispatched from master server with parameter PHASE=RUN, the execution phase of the job is set to QUEUED. Otherwise the execution phase of the job is PENDING. Execution phases of jobs are explained in section 1.2.5.

4. If the job is in phase QUEUED and the worker has free computational slot the job is started and its phase is changed to EXECUTING.

5. During the EXECUTING phase the ZIP file is downloaded from the master server first. Then enclosed files are unpacked from the archive. Finally a computational process is started by executing application peculiar for the type of the worker.

6. When the execution successfully finishes, the job execution phase is changed to COMPLETED. If execution fails the phase is set to ERROR. Worker packs the results and possibly other information about executed job, such as a process exit status and an error process output, into ZIP archive and publishes it through HTTP URI accessible to the master server. This HTTP URI is then set as a result of the job that is passed through UWS service.

7. Meanwhile, the master server periodically checks the phase of the job through UWS service. If the job is in one of the finished phase (COMPLETED, ERROR, ABORTED), the master server reads HTTP URI of the resulting ZIP file. ZIP file is then downloaded and unzipped to the folder dedicated for the particular job.

8. When the result ZIP file is successfully downloaded, the master server initiates UWS destroy request to free resources on the executing worker.

The user can now view results of the job through a web interface in the job details window.

Anytime during the job execution user can initiate abort operation on the job. In this case the master server sends UWS request with the parameter PHASE=ABORT. Even in this case results of the partially executed job are downloaded from the worker (if any). If the user did not set the option to start job instantly as soon as possible after creation (the PHASE=RUN parameter) and the job is still waiting on the worker with the phase PENDING, the user can initiate the start of the job at any time later through the web interface.

Chapter **2**

Requirements analysis

In this section I would like to describe all functional and non-functional requirements on the new version of the VO-CLOUD distributed system. Nevertheless, at first it is necessary to explain three new concepts that are involved in the new version of VO-CLOUD.

2.1 New concepts

2.1.1 VOTable

The VOTable format is an core XML standard created by IVOA organisation. It is designed for the interchange of data represented as a set of tables. "Wherever tabular data is transferred between Virtual Observatory components, VOTable provides the preferred serialization format." [12]

VOTable format has very complicated data model. Every VOTable consists of hierarchy of Metadata and associated TableData. Moreover, VOTable can optionally contain additional information that can be used to extend functionality of astronomical protocols using VOTable format. Metadata can contain numerous information about data contained TableData section, about source of the data, about service that have created the VOTable, about data types used in TableData section and so on. For example Fields are very important part of Metadata used to describe columns in the TableData section. TableData section is simply stream of Rows divided to Cells.

2.1.2 SSAP

The Simple Spectral Access Protocol (SSAP) is IVOA recommendation that defines a uniform interface to remotely discover and access one-dimensional spectra [13]. SSAP can be used from VO applications to access the associated spectra resources in VO archives. SSAP interface uses an HTTP GET-based interface to submit parametrized requests. The most frequent way to use

SSA protocol is with the HTTP parameter `REQUEST=queryData` – this represents operation that returns a table in VOTable format describing candidate datasets which can be retrieved, including standard metadata describing each dataset, and an access reference which can be used to retrieve the data [13].

```
http://vos2.asu.cas.cz/ccd700/q/ssa/ssap.xml
  ?REQUEST=queryData&POS=2.67,56.89&SIZE=2
```

returns structured document in VOTable format describing astronomical spectra found in CCD700 VO archive with a "cone search" defined by a position (POS parameter) and a radius (SIZE parameter).

It is important to note that SSAP `queryData` method does not return the spectral data by itself but only discovers the spectra matched by a combination of SSAP parameters and returns metadata and information about how to obtain them. There are usually two methods to obtain particular spectrum defined in VOTable returned by SSAP query.

- *Access Reference* – It is mandatory column in VOTable returned by SSAP query containing URL address where the required dataset can be directly downloaded.
- *DataLink* – It is the specialized protocol that is explained in the next section.

2.1.3 DataLink

The DataLink is the IVOA recommendation for protocol that is in close relationship with the SSA protocol.

> "Its specifity is to provide a binding mechanism and metadata structure necessary to describe connected datasets or secondary data for independant datasets discovered in previous VO operations." [14]

DataLink provides a suitable alternative for obtaining datasets of spectra discovered by the SSAP query, however it provides possibility to define additional parameters that can adjust data received from the service. Parameters can for example influence the result format of dataset or they can invoke preprocessing action like, e.g., spectrum normalization or cut of selected spectral lines. The way of an invocation of the DataLink protocol is very similar to SSAP – it uses HTTP GET-based interface to submit parametrized request on the DataLink resource URL.

To obtain desired dataset it is necessary to identify it first. The identifier for a DataLink protocol is a column in a VOTable returned by a SSAP called *PublisherDID*. Thereafter it is necessary to find out the DataLink's resource URL and parameters that DataLink supports and their supported values. This

information can be found at the end of VOTable returned by the SSAP query in case that a DataLink protocol is supported by a particular VO archive. However, nowadays there are only a few VO archives supporting DataLink protocols. In the rest of them there is no other option than to use direct download method with Access Reference to obtain datasets from VOTables acquired through SSAP query.

2.2 Functional requirements

Functional Requirement of the VO-CLOUD system can be divided to numerous sections.

2.2.1 General functional requirements

FR 1 Client must be able to communicate with the application through a web browser supporting HTML and JavaScript technologies.

FR 2 Communication between the application and the client is mediated by the HTTP protocol.

FR 3 HTTPS – extension of HTTP protocol for encrypted communication – it is not supported in this version of VO-CLOUD.

FR 4 Every client using VO-CLOUD must have a possibility to create his user account and log into it.

FR 5 The information about registered users must be stored in the database. For security reasons user's passwords must be hashed with SHA-256 hashing algorithm.

FR 6 Application must offer functionality to reset forgotten user's password and send the new one to his e-mail address.

FR 7 Logged user must have ability to change his password.

FR 8 User accounts must be divided to three different groups

- USER
- MANAGER
- ADMIN

FR 9 User logged as ADMIN must have possibility to administer all registered user accounts and he must be able to change the group of user's account and other user properties.

FR 10 User logged as ADMIN must have possibility to administer available workers and computation types, set their attributes and disable them if necessary.

2.2.2 VO-CLOUD storage functional requirements

New version of the VO-CLOUD system must provide storage where the data for upcoming experiments can be prepared instead of uploading them during the new job creation. Management of the storage is reserved for clients logged in with user group MANAGER or ADMIN. Standard user accounts with group USER have read-only access to this storage and they are not allowed to modify files saved in the storage by any way.

The *User* in the context of following functional requirements is considered as the client logged in with any user group (USER, MANAGER, ADMIN).

FR 11 User must be able to list all directories and files that are stored in the VO-CLOUD storage. Mandatory attributes that user must be able to see are names of files and directories, size of files and last modification time of files.

FR 12 User must be able to navigate through the directory structure to see files and directories that are nested inside directories.

FR 13 User must be able to download any chosen file to its local computer storage.

The *Manager* in the context of following functional requirements is considered as the client logged in with the user group MANAGER or ADMIN.

FR 14 Manager must be able to create new directory with specified name in the chosen directory.

FR 15 Manager must be able to rename a directory or a file.

FR 16 Manager must be able to delete a file or a directory recursively.

FR 17 Manager must have possibility to directly upload files from his local computer storage to the target directory.

FR 18 Manager must be able to initiate download from remote resource. As remote resource is considered following:

- a HTTP URL address of directly downloadable file
- a HTTP URL address of a remote folder that is to be recursively downloaded through HTTP protocol (In order to work this feature correctly the HTTP method GET called to the resource address must return list of links to its subdirectories and files. This is the standard feature of majority of HTTP servers called directory index listing.)
- a FTP URL address of directly downloadable file

- a FTP URL address of a folder that is to be recursively downloaded

FR 19 Manager must be able to initiate download of spectra from VO archive. Steps describing the use case of this feature is described in the following steps:

1. Manager directly uploads VOTable file with the desired spectra metadata or he specifies SSAP URL address, where the demanded VOTable can be queried.
2. VOTable is parsed by the server application and information about query status and spectra count are displayed to manager.
3. If the DataLink protocol is supported by the VO archive, information about its parameters and possible values are parsed from the VOTable and dynamically visualised to the manager.
4. Manager can decide whether the download method will be executed through direct Access Reference value or the DataLink protocol (if supported). If the DataLink is chosen manager can set dynamically visualised query parameters.
5. Finally manager submits request and the download task is initiated.

FR 20 Manager must be able to view history of remote downloads and SSAP downloads. The history must contain the following properties:

- *State* – current status of the download task (possible states are CREATED, RUNNING, COMPLETED, FAILED)
- *Creation time*
- *Finish time*
- *Target directory* – the directory in the VO-CLOUD storage where downloaded files shall be saved
- *Download URL* – address of the downloadable resource
- *Download log* – the log mostly containing information about download errors

FR 21 The HTTP, FTP and SSAP downloading service will not support resources requiring authorization in this version.

2.2.3 Job management functional requirements

FR 22 Every logged user must be able to create new experiment computation (in this context called job). System must dynamically generate list of all job types that are available for the logged user, i.e., the

user must not have option to create new job of the computation type that has no available workers.

FR 23 Job types must be divided to two main sections.

- *Standard job types* – jobs available for all logged users
- *Restricted job types* – jobs available only for managers, i.e., clients logged with user group MANAGER or ADMIN

Restricted job types must be invisible for non-managers. Administrator users must have possibility to set the job type restriction in the job administration page.

FR 24 After the user chooses desired job type a job creation window must be displayed. The window must have possibility to set the following information.

- *Project label* – label of the newly created job
- *Description* – optional description of the job
- *Email results* – option to send results to the user's email address after job completion
- *Configuration JSON file* – Configuration file in the JSON format that is to be used as input for worker. Configuration can be uploaded and edited in a text editor of the page. Configuration JSON file also contains information about the files that must be downloaded from the VO-CLOUD master server to worker and that are used as the source of a computation.

FR 25 The job creation page must provide two options for job saving. The first one only saves the job and sends it to the dedicated worker. The second option does the same but moreover sends PHASE=RUN parameter to the worker to set the job into QUEUED phase.

FR 26 If a user is logged in as a manager (user group MANAGER or ADMIN) the user must optionally be able to specify a folder in the VO-CLOUD storage where the results of the newly created job shall be copied if the job successfully finishes in phase COMPLETE.

FR 27 The information about created jobs must be able to be shown in the specialized page. Every user with the exception of administrators must be able to see only his own created jobs. Administrator users must moreover have possibility to see jobs of all users. The job list page must have the following information about jobs.

- *Job type*

- *Identifier* – unique identifier of the job and its owning user
- *Job label* – project label property of the job
- *Creation time*
- *Duration* – duration of the job's execution
- *Phase* – execution phase of the job

The job list page must also provide interface to invoke following job operations.

- start a job in phase PENDING
- abort a running job
- completely delete a job with possible results from VO-CLOUD
- show new page with additional job details

FR 28 The page showing details about a job must contain the same information that are present in the job list page. Moreover it must contain a button Run again which navigates to the create job page where input fields are pre-filled with the information from the source job.

FR 29 The job details page must visually represent a directory structure of downloaded job's results. Any file in this structure must be downloadable. Any textual file must be viewable directly on the page.

FR 30 If job results contains images (PNG, JPG or GIF) in the root folder or in the folder `result` or `results` the images must be directly rendered on the job details page.

FR 31 If job results contains HTML pages (HTM or HTML extension) in the root folder or in the folder `result` or `results` the pages must be shown directly on the job details page as the HTML pages nested in `iframe` HTML element.

FR 32 If a user is logged in as a manager (user group MANAGER or ADMIN) and the job is in phase COMPLETED the user must have possibility on the job details page to copy job's results to the specified folder in the VO-CLOUD storage.

2.3 Non-functional requirements

NFR 1 The worker application must be redesigned to allow quick deployment package creation and deployment.

NFR 2 The system must be able to recover when one of its worker is disconnected or when the connection between the server and a remote server is lost during files downloading.

NFR 3 The master server's client for SSAP and DataLink protocols must be able to communicate with different types of VO archive servers.

NFR 4 The master server and its workers must be able to run on one single application server as well as distributed on several application servers on different machines.

NFR 5 Source codes of the VO-CLOUD system must be published under the Open Source license and publicly available on a public repository.

NFR 6 Applications must be compatible with all application server types supporting necessary technologies and deployable on different platforms.

NFR 7 Users must be shielded from each other. One user must not be able too get input data and results of job of another user by any way (with the exception of administrator).

Chapter 3

Worker realisation

There are a few changes that had to be done in the worker design and implementation in order to preserve same functionality of VO-CLOUD system and to make a worker deployment easier. These changes are in detail explained in the following sections of this chapter.

3.1 Universal worker concept

As explained before the worker computing node consists of executable binary files and the servlet based application deployed on some Java application server (see Section 1.1). The problem is that for every new computational method, i.e., new executable computational application, it is necessary to create new servlet based application where the steps of an executable application invocation are defined in a source code. Moreover every computing node where the worker is deployed could have different parameters for example the path to executable computational application could differ. These parameters are specified in the resource configuration file which has to be built together with compiled source codes. Imagine example where there are two different computational applications on three computing nodes. In this case it would be necessary to create two different implementations of servlet based application and together to build six packages that have to be deployed on the correct application server.

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. This method has many benefits:

- The deployment on the application servers is easier.
- An application server uses less resources.

- Constraints on the execution queue such as maximum concurrent running jobs are now applied on the whole computing node and not only on computational application. This is more logical solution because a computing node is usually limited as the whole part.

- The capabilities of UWS protocol are fully utilized. The original solution used only one job list resource per application.

- Addressing of the UWS resources are more intuitive. For example compare two addresses of the original solution

  ```
  http://localhost/rdf/uws/
  http://localhost/som/uws/
  ```

  with the new solution

  ```
  http://localhost/universal/uws/rdf/
  http://localhost/universal/uws/som/
  ```

  The master server application can now hold information about available computing nodes and for every node list of possible computational methods that the node is capable of doing. A URL address of the UWS resource can be constructed of a URL address of the computing node and the computational method name.

Universal worker is configured through the XML document matching XSD schema specially created for this purpose. Figure 3.1 shows example of such XML configuration file where the computational method RDF is involved. Of course there can be many more `<ns:worker>` tags inside `<ns:workers>` tags in order to describe more computational methods. Tags `<ns:exec-command>` and `<ns:command>` are used to describe the way how to start the computational process.

The configuration file is in the new version of VO-CLOUD server packed in a deployment archive as a resource file. For each computing node it is then necessary to build its own deployment application with the right configuration file. However, the universal worker is designed to be able to download configuration file from dedicated remote repository for example from the master server. In this case it would be only necessary to deploy the same universal worker application on each computing node's application server and to put all configuration files into one place in the master server. Every universal worker would during its initialization download its configuration file from the master server through specialized interface and it would not be necessary for administrator to configure workers on the master server because it already knows the information. This feature is going to be implemented in the next version of VO-CLOUD system.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ns:uws-settings
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ns='http://vocloud.rk.cz/schema'
  xsi:schemaLocation='http://vocloud.rk.cz/schema configSchema.
      xsd'>
  <ns:vocloud-server-address>http://localhost/vocloud2</ns:
      vocloud-server-address>
  <ns:local-address>http://localhost/universal/uws</ns:local-
      address>
  <ns:max-jobs>2</ns:max-jobs>
  <ns:description>Universal UWS worker</ns:description>
    <ns:workers>
        <ns:worker>
          <ns:identifier>rdf</ns:identifier>
          <ns:description>RDF</ns:description>
          <ns:restricted>false</ns:restricted>
          <ns:binaries-location>/home/voadmin/RDF</ns:binaries-
              location>
          <ns:exec-command>
            <ns:command>python3</ns:command>
            <ns:command>
              ${binaries-location}/runRF.py
            </ns:command>
            <ns:command>${config-file}</ns:command>
          </ns:exec-command>
        </ns:worker>
    </ns:workers>
</ns:uws-settings>
```

Figure 3.1: Universal worker configuration file

3.2 Universal worker workflow

A universal worker workflow was slightly changed to meet the new functional requirements. This section describes the new workflow of a communication between a universal worker and the master server. In the previous version of VO-CLOUD the master server had to send to worker the ZIP archive containing data and configuration file (In fact the worker got URL address where the master server published the ZIP archive. Worker then had to download it.). In the new version of VO-CLOUD only configuration file is sent to the worker where addresses of necessary files are described and worker have to download them during a job execution. Consider the following example where the user have created a new job with parameter to start it as soon as possible (PHASE=RUN).

1. The master server chooses a worker capable of doing desired computational method.
2. The master server creates a new job on the selected worker by using HTTP method POST on the worker's UWS service (see Section 1.2.5). The configuration file is sent directly in this request as a HTTP parameter specified inside a request body with the `Content-Type` HTTP header set to `application/x-www-form-urlencoded` (configuration file content must be properly encoded by the master server).
3. The worker creates a new job in the queue and assigns it the decoded configuration file. The job's execution phase is set to QUEUED.
4. When there are free resources on the worker the execution phase is changed to EXECUTING.
5. Worker parses passed configuration file and it downloads required files specified in the configuration.
6. Worker executes computational application from the directory where required files have been downloaded.
7. When the process ends the worker deletes files that had to be downloaded and it packs the produced results with additional information (process error output, standard output and so on) into ZIP archive file.
8. The ZIP file is published to be visible for the master server and the URL address is added as a job result.
9. Job's execution phase is changed to COMPLETED.
10. The master server downloads the ZIP file from the worker and unpacks it in its job results directory.

```
{
  "download_files": [
    {
        "urls": ["vocloud://preparedData/data1"]
    },{
        "urls": ["vocloud://test/mh160020.fit",
                 "vocloud://test/datalink/tj180031"],
        "folder": "/folder1"
    }
  ],
  "normalize": true,
  "binning": true,
  "remove_duplicates": true,
  "decompose": {
    "iterations": 300,
    "kind": "PCA",
    "components": 30
  }
}
```

Figure 3.2: Configuration JSON file

11. The UWS command destroy is called to the worker after the download to the master server is finished.

12. Produced results can now be viewed by the user.

3.3 Files downloading feature

A universal worker implements feature that allows to specify which files should be downloaded. This configuration is done by adding new key-value pair inside the configuration JSON file where the key is `"download_files"`.

Example of such a configuration JSON file can be seen in Figure 3.2. In this example it is possible to see `"download_files"` key and other keys that are used for configuration of computational method. A value of `"download_files"` key is an array of elements where each element specifies one target folder in the worker working directory. Multiple files can be downloaded to one target directory simply by using more URL addresses in the `"urls"` key definition. If the `"folder"` element specifying the target directory is not present the whole worker working directory is considered as the target. As can be seen in the example the protocol in the URL address can be set to `vocloud` instead of standard `http` (which is of course supported too). This means that the

```
{
  "folders":[
    "dataInfo",
    "4270-4523",
    "4753-5005"
  ],
  "files":[
    "spec-55893-F9302_sp10-146.fits",
    "spec-55893-F9302_sp10-220.fits",
    "spec-55893-F9302_sp06-157.fits",
    "spec-55893-F9302_sp03-248.fits",
    "spec-55921-GAC_089N28_B3_sp04-078.fits",
    "spec-55921-GAC_089N28_B3_sp14-141.fits",
    "spec-55876-GAC_089N28_B3_sp13-231.fits"
  ]
}
```


Figure 3.3: JSON describing VO-CLOUD storage directory

address is automatically substituted to the address where the VO-CLOUD master server provides interface to download files from its storage. Moreover whole directories can be recursively downloaded from the master server. This is made possible by the master server's service for downloading files.

The master server's service for the downloading is realized by the simple Java servlet class `FilesystemDownloadServlet`. This servlet is registered to URI

```
/files/
```

and files from this service can be directly downloaded by appending the VO-CLOUD storage path of the file to the service resource. For instance if the desired file is named `text.jpg` and it is saved in the VO-CLOUD storage in a folder named `images` file can be downloaded from URI

```
/files/images/text.jpg
```

If the URI does not point to the file but the directory service returns JSON where the list of files and directories is specified. This feature can be used for recursive downloading of whole directories. Example of such a JSON can be seen in Figure 3.3.

3.4 Visualisation

It is important to note that the visualisation itself is not created on the master server from the downloaded results but on the workers where some visualisation is expected. For instance worker using a computational method of self-organizing maps (SOM) must create images or HTML pages that are packed as the part of its results and then directly viewable by the master server. Results in the form of HTML pages can moreover contain JavaScript source codes that allows to do dynamic visualisation and possible navigation and filtering in the set of computed results.

Chapter 4

Master server realisation

This chapter describes changes that had to be done in the master server in order to comply with functional and non-functional requirements.

4.1 VO-CLOUD storage

VO-CLOUD storage is the main reason why the whole VO-CLOUD system had to be redesigned and reimplemented. It serves as the storage where managers can prepare data that every user may involve in his experiment. The only thing the user have to prepare is the configuration JSON file specific to the computational method.

For all users and workers the VO-CLOUD storage simply serves as the service that provides interface to download files, to list files and directories and for authorized users to manage files and directories that are saved in the storage. In principle there are many places where the files and their directory structure could be physically saved. It is only necessary to map the operations of VO-CLOUD storage to the physical storage properly.

Master server VO-CLOUD is a standalone application which means that it is deployed on only one application server and operating system. VO-CLOUD storage could then by mapped directly on some specific directory in the master server's operating system. It is solution that is very simple, intuitive and moreover the storage could be easily managed directly from the operating system or with the help of protocols such as SSH or FTP.

For the purposes of the direct mapping of the storage to some specific folder the new specialized Stateless EJB bean name `FilesystemManipulator` was created. This bean provides business methods that can be used to retrieve information about the mapped directory and its subdirectories, to download a specied file, to create a new file from the passed data stream and to manipulate with the file or directory (remove, rename). All classes that require access to the VO-CLOUD storage must use the `FilesysteManipulator` EJB bean as the mediator for their operations. The great advantage is that the

`FilesystemManipulator` bean can have defined security constraints on its methods. For instance the method `deleteFileRecursively` that is capable of recursive deletion of a whole directory can be annotated with the security constraint to be available only for managers and administrators and not for common users. The authorization checking is then not only done in the presentation tier of the application but moreover in the business tier and the application is then potentially more secure.

4.2 Remote download feature

Administrator and managers have possibility to create a new task that will download desired files from the remote resource (HTTP or FTP server). This task is in the context of the VO-CLOUD application called *download job*. Two specialized EJB Stateless beans were implemented for this feature `DownloadManager` and `DownloadProcessor`. The `DownloadManager` provides method `enqueueNewURLDownload` for creating a new download job. This method stores information about the download job to the database and it asynchronously invokes a method from the `DownloadProcessor` bean where the download itself is initiated. Therefore, the `DownloadManager` bean only prepares the download job for asynchronous execution in `DownloadProcessor` bean.

The great advantage of the remote download feature is that it supports recursive downloads of whole directories. In the FTP protocol this is relatively easy. FTP provides command to list directories and files in the current working directory. The problem is with directories downloading in the HTTP protocol. Targeted HTTP server where the URL address represents a directory to be downloaded must have allowed feature called index directory listing, i.e., HTTP server must return list of URL links pointing to directory's files and subdirectories. Example of such a page can be seen in Figure 4.1. This directory listing page can be parsed by the `DownloadProcessor` bean and used for crawling through the directory tree structure to download whole targeted directory recursively.

4.3 SSAP and DataLink client

The master server provides possibility for managers and administrators to download data directly from the VO archives by using SSAP and DataLink protocols. At first it is necessary to get the VOTable representing the list of spectra to be downloaded. It is simple to get the VOTable. User either directly uploads the VOTable to the master server or he specifies the URL address of SSAP resource where the VOTable can be downloaded. Now the VOTable must be parsed. Originally I wanted to use Java Architecture for XML Binding technology (JAXB) in the implementation to directly convert

Index of /v458val/NEW/6255-6767

Name	Last modified	Size	Description
Parent Directory		-	
qi130014.fit	14-Sep-2007 14:41	20K	
qi130015.fit	14-Sep-2007 14:41	20K	
qi150018.fit	18-Sep-2007 01:03	20K	
qi150020.fit	18-Sep-2007 01:03	20K	
qi160018.fit	19-Sep-2007 02:22	20K	
qi160020.fit	19-Sep-2007 02:22	20K	
qj010012.fit	03-Oct-2007 21:53	20K	
qj010013.fit	03-Oct-2007 21:53	20K	
qj150019.fit	16-Oct-2007 05:15	20K	
qj150020.fit	16-Oct-2007 05:15	20K	

Figure 4.1: Page with a directory listing

VOTable structure into the set of Java mapped objects. However, the problem is that there are different versions of VOTable XML document and it would be complicated to create a new set of Java objects for each version of the VOTable schema. Finally, the Simple API for XML (SAX) principle of XML parsing was involved in the VOTable processing. The SAX parser simply goes gradually through the VOTable file and it calls method when one of the following events happen:

- an XML opening element was found
- an XML closing element was found
- characters between XML elements were found

SAX parsing method is potentially many times faster than JAXB because it does not have to convert all elements to their Java objects counterparts but instead it runs once through the XML file and it remembers only things that are necessary for further processing.

The important thing is that the parser must be able to recognize from a VOTable if the DataLink protocol is available. If so the parser must collect information about its possible parameters. The VOTable parser is available through the interface of the class `VotableParser`. Its method returns an instance of class `IndexedSSAPVotable` which consists of information that are necessary for downloading spectra through Access Reference column or possibly through the DataLink protocol (if supported).

The user can now choose if he wants to process downloading with the usage of Access Reference column or with the usage of DataLink protocol. If he chooses the DataLink protocol he can now specify parameters that the DataLink protocol supports. The web page where the user can specify the DataLink parameters is dynamically created according to the DataLink protocol information that are provided in the parsed VOTable.

After submitting the download task the new download job is created for each spectrum specified in the VOTable in the `DownloadManager` Stateless bean. The whole list of spectra is then asynchronously dispatched to `DownloadProcessor` bean where the download job is being processed.

4.4 Preprocessing

Usually for some data types and especially for data determined for a data mining it is necessary to do preprocessing on them (for instance to do normalization, rebinning and so on). Preprocessing is defined as an operation that takes selected spectra as an input and produces a file or files that must be saved to the VO-CLOUD storage to be available for common users.

The idea is to consider the preprocessing as a computational method that can be executed on workers. The preprocessing would be then defined as a method restricted only to managers and administrators. It is also necessary to implement feature to move the results of a job's computation into specified target folder in the VO-CLOUD storage. This feature is of course available only for managers and administrators because a common user does not have permission to save files into the VO-CLOUD storage.

In order to do preprocessing over downloaded data the storage manager must create a new preprocessing job and in the new job creation page he must set the targeted folder in the VO-CLOUD storage. After job completion the master server automatically copies result files to the targeted VO-CLOUD storage directory and the data can now be used by any common user for a new experiment computation.

4.5 Workers management

It is absolutely necessary for the master server to be informed what possible types of computation are available and which workers supports it in order to

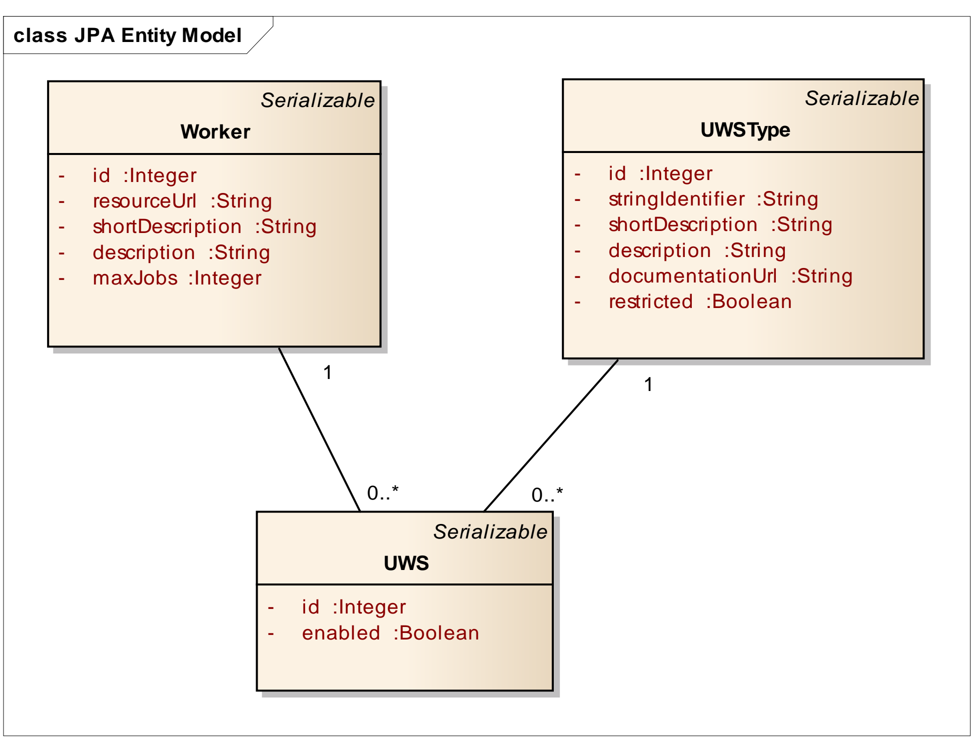


Figure 4.2: Class diagram of JPA Entity classes

effectively dispatch a job computation to them. In this version of VO-CLOUD the information about the possible computational types and workers must be set manually by the administrator through a specialized web page. However in the future versions of the VO-CLOUD it is expected that the information would be passed automatically by the registration of some newly deployed worker.

JPA Entity classes used for object/relational mapping to a database had to be redesigned in order to allow the future extendibility, to allow better load balancing method for computation dispatching and to make possible the dynamic rendering of possible job computational types in the web user interface. Moreover the new redesigned model is more logical in the context of a newly created concept Universal worker (see Section 3.1). Class diagram of the redesigned JPA Entity classes can be seen in Figure 4.2.

Class `Worker` represents a computational worker. Every computational worker must have defined resource URL address where the UWS service is available, short description of worker to be able to recognize, maximum number of jobs that can be started parallelly on the worker and optionally long description of the worker.

Class `UWSType` represents a computational method. The computational method must have defined string identifier which is used to name a job list

queue in workers, a short description that is used as the name of the computational method in a web user interface, a restricted flag that signalizes if the computational method is restricted only for managers and administrators and finally optional parameters long description and an URL address where the computational method is documented.

Class `UWS` is a mediator of the `M:N` relationship between a computational worker and a computational method. Moreover this class contains parameter `enabled` that allows administrator to disable usage of a specified computational method on some specified worker.

All three classes also contain parameter `id` that is mandatory by the JPA specification and it is used as a primary key in a relational database.

4.6 Jobs load balancing

In order to maximize throughput of the whole distributed VO-CLOUD system it was necessary to design an algorithm that would assign the newly created computational job to the least loaded worker. The algorithm follows these steps:

1. Find all computational workers that are able to execute desired computational method.
2. Choose the first computational worker.
3. Find out how many jobs assigned to this worker are in the execution phase EXECUTING.
4. Find out how many jobs can be run parallelly on this worker.
5. Count the difference count of maximum parallelly running jobs minus count of jobs in the phase EXECUTING.
6. Remember the difference and continue from the step 3. with a next computational worker if there is such.
7. Find out the worker with the greatest difference. If there are more workers with the same greatest difference choose randomly one.
8. Assign the new job to this chosen computational worker.

Chapter 5

Future development

There are many ways how to improve the VO-CLOUD distributed system. The most important thing is to make the deployment of the master server and of distributed workers easier. The following improvements in deployment are planned to be done in the future versions:

- Move the worker XML configuration file to one place where configurations could be downloadable by newly deployed workers.
- Add possibility for workers to register themselves to the master server through the specialized web service. It would not be than necessary for administrators to configure every worker manually on the master server.
- Simplify the installation of application servers and executable computational files by using a virtualization tool such as Docker[4].

The VO-CLOUD storage is planned to be improved too.

- Allow manager to select multiple files or directories and to delete them together.
- Allow operations move and copy.
- Introduce Simple Application Messaging Protocol (SAMP) [15] that would allow users to send spectra from the VO-CLOUD storage directly to the visualisation tool running on their computer such as SPLAT-VO.
- Add to the manager ability to stop currently running download job and to delete items from a download job history.

[4] https://www.docker.com/

Conclusion

The goal of this thesis has been met. The fundamental concepts and the workflow of the original VO-CLOUD system have been analysed and the master server and the universal worker parts of the distributed VO-CLOUD system have been successfully implemented. The VO-CLOUD system is now able to download astronomical spectra from VO archives by using astronomical protocols SSAP and DataLink, to run preprocessing on them, to feed them to the computational workers and to visually display the results of the computations to the users.

The concept of many different types of computational workers have been redesigned to one universal worker and therefore the process of deployment on the computation nodes is simplified.

I have gained a valuable experience during the process of designing and the implementation of the new version of VO-CLOUD system and I have acquired 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. 2015-05-04]. Available from: `http://www.ivoa.net/about/TheIVOA.pdf`

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

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

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

[5] 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`

[6] WWW Consorcium. *Extensible Markup Language (XML) 1.0 (Fifth Edition) [online]*. November 2008, [cit. 2015-05-05]. Available from: `http://www.w3.org/TR/REC-xml/REC-xml-20081126-review.html`

[7] Oracle. *Trail: The Reflection API [online]*. [cit. 2015-05-06]. Available from: `http://docs.oracle.com/javase/tutorial/reflect/index.html`

[8] The Internet Society. *Hypertext Transfer Protocol -- HTTP/1.1 [online]*. 1999, [cit. 2015-05-07]. Available from: `http://tools.ietf.org/pdf/rfc2616.pdf`

[9] Oracle. *Trail: RMI [online]*. [cit. 2015-05-08]. Available from: `https://docs.oracle.com/javase/tutorial/rmi/`

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

[11] 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

[12] Ochsenbein, F.; Williams, R.; Davenhall, C.; et al. IVOA Recommendation: VOTable Format Definition Version 1.2. *ArXiv e-prints*, Oct. 2011, 1110.0524. Available from: http://adsabs.harvard.edu/abs/2011arXiv1110.0524O

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

[14] Laurent, M.; Bonnarel, F.; Louys, M. *IVOA Recommendation: DataLink Protocol Version 1.0 [online]*. The IVOA, May 2013, [cit. 2015-05-10]. Available from: http://www.ivoa.net/documents/Notes/DataLink/20130502/NOTE-DataLinkProposal-1.0-20130502.pdf

[15] Taylor, M.; Boch, T.; Fitzpatrick, M.; et al. IVOA Recommendation: SAMP - Simple Application Messaging Protocol Version 1.3. *ArXiv e-prints*, 2011, 1110.0528. Available from: http://adsabs.harvard.edu/abs/2011arXiv1110.0528T

Appendix A

Acronyms

API Application Programming Interface

EE Enterprise edition

EJB Enterprise Java Bean

FTP File Transfer Protocol

GIF Graphics Interchange Format

GUI Graphical user interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IVOA International Virtual Observatory Alliance

JAXB Java Architecture for XML Binding technology

JMS Java message service

JPA Java Persistence API

JSF JavaServer Faces

JSON JavaScript Object Notation

PNG Portable Network Graphics

RDF Random Decision Forest

REST Representational State Transfer

RMI Remote method invocation

SAMP Simple Application Messaging Protocol

SAX Simple API for XML

SOM Self-organizing map

SQL Structured Query Language

SSAP Simple Spectral Access Protocol

SSH Secure Shell

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

UWS Universal Worker Service

VO Virtual Observatory

XHTML Extensible HyperText Markup Language

XML Extensible markup language

Appendix B

Contents of enclosed DVD

```
readme.txt ........ the file with DVD contents description
exe ............... the directory with deployable packages
src ............... the directory of source codes
  wbdcm ........... implementation 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
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://vocloud.rk.cz/schema"
5    xmlns:tns="http://vocloud.rk.cz/schema"
6    elementFormDefault="qualified">
7
8    <xsd:complexType name="worker">
9      <xsd:sequence>
10       <xsd:element name="identifier" type="xsd:token"/>
11       <xsd:element name="description" type="xsd:string"/>
12       <xsd:element name="restricted" type="xsd:boolean" default=
             "false"/>
13       <xsd:element name="binaries-location" type="xsd:string"/>
14       <xsd:element name="exec-command" type="tns:command-list"/>
15     </xsd:sequence>
16   </xsd:complexType>
17   <xsd:complexType name="command-list">
18     <xsd:sequence>
19       <xsd:element name="command" type="xsd:string" maxOccurs="
             unbounded"/>
20     </xsd:sequence>
21   </xsd:complexType>
22   <xsd:element name="uws-settings">
23     <xsd:complexType>
24       <xsd:sequence>
25         <xsd:element name="vocloud-server-address" type="xsd:
```

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

Appendix D

Master server README file

```
Requirements
============

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

Production 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 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

   apt-get install postgresql
   login as postgres "su - postgres" and run client "psql
      template1"
   then type 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!

5. Configure database resource in Wildfly

   Log into Wildfly admin console: http://localhost:9990/
   Type in credentials of administrating user

   Download JDBC for PostgreSQL https://jdbc.postgresql.org/
   In the admin console navigate to Deployments
   Click Add, select downloaded JDBC .jar file and click Ok
   Enable newly uploaded JDBC driver

   Navigate to Configuration tab
   Select Datasources
   Click Add
   Name: VocloudDS
   JNDI Name: java:jboss/datasources/vocloud
   Click Next
   Select postgresql jdbc driver
   Click Next
   Connection URL: "jdbc:postgresql://localhost:5432/vocloud" (
      without quotes)
   Username: vocloud
   Password: vocloud
```

```
   Click Done
   Enable VocloudDS

   Datasource can be tested in section Connection > Test
       connection
   Ping should be successful

6. Configure e-mail resource in Wildfly

   It is necessary to have an email address which will serve as
       the source of emails. 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
   Name: vocloud-smtp
   Host: smtp.vocloud.org
   Port: 465
   Click Save

   Navigate to Mail subsystem section
   Click Add
   JNDI Name: "java:jboss/mail/vocloud-mail" (without quotes)
   Click View on the newly created mail session
   Click Add
   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
    Name: VocloudSecurityDomain
    Click Save
    Click View on the newly created security domain
    Click Add
```

```
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. Deploy vocloud.war to the Wildfly server

```
Navigate to section Deployments
Click Add
Select vocloud.war file
Submit
Enable the vocloud.war

VO-CLOUD should now run on http://localhost:8080/vocloud
```

9. Create admin account

```
Using http://localhost:8080/vocloud/register.xhtml
Register a new account with username admin

This account have now administrator privileges.
```

Appendix E

Universal worker README file

```
Requirements
============

- JDK 7+
- Java application server supporting Java servlet technology (
    tomcat, wildfly, ...)
- Maven tool (if building is necessary)
- Executable computational application for each desired
    computational type

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 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 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 universal-worker configuration file (optional step
    if you want another configuration that it is in prebuilt
    archive)

    Download sources of universal-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/universal-worker.war

5. Deploy universal 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 universal-worker.war archive
    Click OK
    Enable the newly deployed application

    UWS service should now run on
    http://localhost:8080/universal-worker/uws


Note: This is only description of universal-worker application
    which serves as the mediator between the master server and
    executable computational application. In order to make a
    worker fully functional you have to set the configuration
    file of the universal-worker to point to the valid locations
```

```
 of the executable computational applications. For more
information see the documentation of the specific executable
 computational application.
```