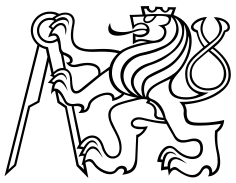
Czech technical university in Prague
Faculty of Civil engineering

# Master's thesis

Prague 2018 Bc. Adam Laža

Czech technical university in Prague
Faculty of Civil engineering
Study programme Geodesy and cartography
Branch Geomatics



Master's thesis

# Process isolation in PyWPS framework

## Izolace procesů ve frameworku PyWPS

Supervisor: Ing. Martin Landa, Ph.D.
Department of Geomatics

Prague 2018

Bc. Adam Laža

**ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE**
**Fakulta stavební**
Thákurova 7, 166 29 Praha 6

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: Laža Jméno: Adam Osobní číslo: 396924

Zadávající katedra: Katedra geomatiky

Studijní program: Geodézie a kartografie

Studijní obor: Geomatika

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce: Izolace procesů ve frameworku PyWPS

Název diplomové práce anglicky: Process isolation in PyWPS framework

Pokyny pro vypracování:
Diplomová práce se věnuje možnostem izolace procesů v rámci frameworku PyWPS jako jedné z open source implementací standardu OGC WPS (Web Processing Service). Na základě rešerše budou specifikována možná řešení izolace či kontejnerizace WPS procesů pro předem definované scénáře. V rámci práce se počítá taktéž s návrhem implementace vybraného scénáře v programovacím jazyku Python.

Seznam doporučené literatury:
Scott Gallagher: Mastering Docker, ISBN: 978-1787280243
Sébastien Goasguen: Docker Cookbook, ISBN: 978-1491919712
Deepak Vohra: Pro Docker, ISBN: 978-1484218297
OGC® WPS 2.0 Interface Standard

Jméno vedoucího diplomové práce: Ing. Martin Landa, PhD.

Datum zadání diplomové práce: 11.10.2017 Termín odevzdání diplomové práce: 7.1.2018
*Údaj uveďte v souladu s datem v časovém plánu příslušného ak. roku*

[redacted] [redacted]
Podpis vedoucího práce Podpis vedoucího katedry

## III. PŘEVZETÍ ZADÁNÍ

*Beru na vědomí, že jsem povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je nutné uvést v diplomové práci a při citování postupovat v souladu s metodickou příručkou ČVUT „Jak psát vysokoškolské závěrečné práce“ a metodickým pokynem ČVUT „O dodržování etických principů při přípravě vysokoškolských závěrečných prací“.*

[redacted] [redacted]
Datum převzetí zadání Podpis studenta(ky)

**Abstract**

This master thesis is dedicated to a process isolation in PyWPS framework as one of the OGC WPS implementations. OGC WPS is Web Processing Service Standard defined by Open Geospatial Consortium.

The first part describes the standard itself including all three mandatory operations *GetCapabilities*, *DescribeProcess* and *Execute*. At the end of the first part some implementations of the standard are mentioned.

The second part concentrates on *PyWPS*, one of the WPS implementations written in Python. Readers are introduced to the current state of PyWPS as well as to *PyWPS-demo* project, a demo server instance, which the implementation part is based on. A research about possible solutions of process isolation follows and then *Docker* technology is described as final choice for implementaion.

The third part covers the implementation of Docker containers for process isolation. The workflow of *Execute* operation is described in detail and brand new *Container* class with all its methods is introduced.

**Keywords:** OGC WPS, PyWPS, Docker container, Python, process isolation, Web Processing Service, geoprocessing.

**Abstrakt**

Tato diplomová práce se věnuje možnostem izolace procesů v rámci frameworku PyWPS jako jedné z implementací OGC WPS. Web Processing Service je standard vydaný a dále rozšiřovaný Open Geospatial Consorciem.

První část popisuje samotný standard včetně všech základních požadavků *GetCapabilities*, *DescribeProcess* a *Execute*. V závěru první části jsou zmíněny některé z implementací WPS standardu.

Druhá část se zaměřuje na *PyWPS*, což je implementace WPS standardu napsaná v programovacím jazyce Python. Čtenáři jsou seznámeni jak se současným stavem PyWPS, tak s projektem *PyWPS-demo*, ukázkovou instancí PyWPS serveru, na kterém je postavena praktická část. Následuje rešerše, která mapuje možné řešení izolace procesů, a nakonec je popsána *Docker* technologie, která slouží pro kontejnerizaci. Tato technologie byla vybrána pro samotnou implementaci izolace.

Poslední část se zabývá použitím Docker kontejnerů pro izolaci procesů. Detailně je vysvětleno, jak funguje *Execute* operace a následně je popsána nově vytvořená třída *Container* se všemi svými metodami.

**Klíčová slova:** OGC WPS, PyWPS, Docker kontejner, Python, izolace procesu, geoprocesing, zpracování dat.

**Declaration of authorship** I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged. Formulations and ideas taken from other sources are cited as such.

In Prague .....  .....

(author sign)

**Acknowledgement** Foremost, I would like to thank my parents for their long-time support during my studies. My thanks also belong to Jáchym Čepický for his provided insight into PyWPS. Then I want to thank Martin Landa, my supervisor, not only for his guidance during the work on the thesis, but also that he revealed me the way to programming.

# Contents

# Introduction

With the huge progress of technologies, our society is becoming more and more digitalized and the amount of various data is getting bigger and bigger. There are data all around us and demand for applications or services based on the data is growing. However, the data in its raw form may not be sufficient to make a conclusion. More often the data need to be processed and used as inputs data for some kind of analyses. With increasing number of gathered data such as satellite images or remotely sensed data, any manual processing is almost inconceivable. The data processing needs to be done in a systematic and fully-automized way.

Therefore, in order to be able to process data independently of the type of acquisition, format or platform, international standard interfaces and standardized frameworks are necessary. The Open Geospatial Consortium, Inc. (OGC) - an organization oriented toward open geospatial standards - researches and establishes technical standards for data compatibility and interoperability technical standards. Besides quite famous and used standards as WMS and WFS, there exists the WPS standard. The WPS standard defines an interface that facilitates the publishing of geospatial processes. It provides rules how inputs and outputs are handled. There are several implementations of WPS standard. This work is primarily focused on the *PyWPS* - a WPS implementation written in Python.

The main topic of this thesis is process isolation in PyWPS framework. A process is just some geospatial operation which has its defined inputs and outputs and which is deployed on a server. The server is able to execute multiple processes at the same time. This thesis deals with the isolation of individual processes, especially for security and performance reasons. With every process fully isolated, so they cannot interact with each other, the higher security level is ensured.

The thesis is composed of several parts. The first part describes the WPS standard, its operations *GetCapabilities*, *DescribeProcess* and *Execute* and inputs and outputs structures. A quick overview of some implementations of WPS standard follows and brings a basic information about them.

Nevertheless, this work is dedicated to PyWPS, an implementation in Python. In the second part, its current state is described as well as *pywps-demo* - a side

project providing demo server instance - which the practical part is based on. Following research covers various projects and technologies which were considered as a solution for process isolation. Eventually, the Docker technology is chosen for the implementation part. Docker has been selected as one of the most used technology for containerization. It puts every process into a separate container so the isolation is ensured. Moreover, Docker provides a mechanism to pause, stop and start a container so it looks like a possible solution for the future WPS 2.0.0 standard implementation which requires this functionality. Using Docker, it also opens new possibilities, e.g. being able to deploy running job to cloud.

The third part describes the implementation. It explains the Execute operation workflow, a process execution and how the Docker containers are used for the PyWPS process isolation. New *Container* class, which was developed during the work on this thesis, is introduced as well as its methods.

# Part I

# Web Processing Service

# 1 Web Processing Service

## 1.1 History

The first mention of the Web Processing Service was in October 2004. Back then it was named Geoprocessing Service [1]. The specification was first implemented as a prototype in 2004 by Agriculture and Agri-Food Canada (AAFC). In its further development during a Geoprocessing Services Interoperability Experiment [2] the name was changed to "Web Processing Service" to avoid the acronym GPS, since this would have caused confusion with the conventional use of this acronym for Global Positioning System [6]. The first version of WPS was released in September 2005 [3]. The experiment demonstrated that various clients could easily access and bind to services which were set up according to the WPS Implementation specification.

Currently two major versions of WPS Standard exist. The WPS version 1.0.0 is currently most used. If not explicitly said this thesis is dedicated to the version 1.0.0. The WPS version 2.0.0 was released in 2015 [7].

## 1.2 Open Geospatial Consortium

The OGC *Open Geospatial Consortium* is an international non-profit organization committed to making quality open standards for the global geospatial community. These standards are made through a consensus process and are freely available for anyone to use. The OGC members come from government, commercial organizations, NGOs, academic and research organizations.[4]

A predecessor organization, OGF, the Open GRASS Foundation, started in 1992. From 1994 the organization used the name *Open GIS Consortium*, in 2004 the Board changed the name to *Open Geospatial Consortium*.[5]

Some of the widely-use OGC standards are:

- WCS, WMS, WFS, WMTS or WPS - standards for web services
- GML, KML - standards for XML-based languages

## 1.3 Web Processing Service

The OGC Web Processing Service (WPS) Interface Standard defines a standardized interface that facilitates the publishing of geospatial processes. Also provides rules how to standardize requests and responses for geospatial processing services.

*Process* means any operation on spatial data from simple ones like maps overlay or buffering to highly complex as complicated global models. Any kind of GIS functionality can be offered to clients across a network with correctly configured WPS.

*Publishing* means creating human-readable metadata that allow users to discover and use service as well as making available machine-readable binding information.

*Data* can be both vector or raster data and can be delivered across the network or be available at the server.

The interface does not specify any specific processes that can be implemented by a WPS nor any specific data inputs or outputs. Instead it specifies generic mechanisms to describe any geospatial process and data required and produced by the process. The interface does not only provide mechanisms for calculation but also mechanisms that identify required data, initiate the calculation and manage output data so clients can access it.

Web Processing Service as one of the OGC web services specifies three types of requests which can be requested by a client and performed by a WPS server. The implementation of these three requests is mandatory by all servers:

***GetCapabilities*** - The request returns to the client a Capabilities document that describes the abilities of the specific server implementation. It also returns the name and abstract of each of the processes that can be run on a WPS instance.

***DescribeProcess*** - The request returns details about the processes offered by a WPS instance. It describes required inputs and produced outputs and their allowable formats.

***Execute*** - The request allows the client to run a specified process with provided parameters and returns produced outputs.

These operations are very similar to other OGC Web Services such as WMS, WFS, and WCS. Common interface aspects are defined in the OGC Web Services Common Implementation Specification [8]. As seen in the class diagram at Fig. 1 the WPS interface class inherits the GetCapabilities operation from OGCWebService interface class. The operations Execute and DescribeProcess are specific for the WPS. The WPS operations are based on HTTP GET[1] and POST[2] requests.

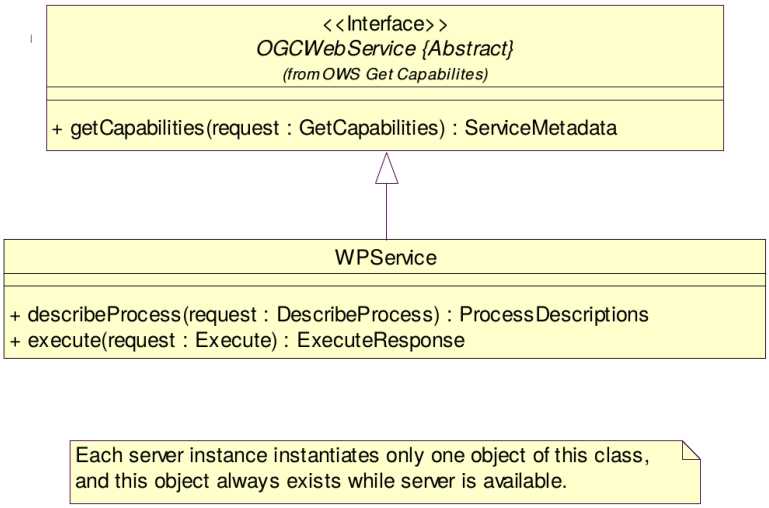



Figure 1: WPS interface UML description, source: [6]

The GetCapabilities and DescribeProcess shall use HTTP GET with KVP encoding and Execute operation shall use HTTP POST with XML encoding. Summarized in Tab. 1.

| Operation | Request encoding | |
|---|---|---|
| | Mandatory | Optional |
| GetCapabilities | KVP | XML |
| DescribeProcess | KVP | XML |
| Execute | XML | KVP |

Table 1: Operations request encoding

[1] *HTTP GET* requests data from a specified resource. Data are sent in the URL of a GET request.

[2] *HTTP POST* submits data to be processed to a specified resource. Data are sent inside the HTTP message body of POST request.

**KVP encoding** are key-value pairs usually sent via HTTP GET request method encoded directly in the URL. The keys and values are separated with = sign and each pair is separated with & sign or with ? sign at the beginning of the request. Example could be the get capabilities request:

Listing 1: GetCapabilities with KVP encoding.

```
http://server.domain/wps?service=WPS&request=GetCapabilities&
    version=1.0.0
```

In this example, there are 3 pairs of input parameter: service, request and version with values WPS, GetCapabilities and 1.0.0 respectively.[17]

**XML payload** is XML data sent via HTTP POST request method. The XML document can be more rich, having more parameters, better to be parsed in complex structures. The Client can also encode entire datasets to the request, including raster (encoded using base64) or vector data (usually as GML file).[17]

Listing 2: GetCapabilities XML payload example

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:GetCapabilities language="cz" service="WPS" xmlns:ows="http
    ://www.opengis.net/ows/1.1" xmlns:wps="http://www.opengis.net/
    wps/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
     http://schemas.opengis.net/wps/1.0.0/
    wpsGetCapabilities_request.xsd">
  <wps:AcceptVersions>
    <ows:Version>1.0.0</ows:Version>
  </wps:AcceptVersions>
</wps:GetCapabilities>
```

### 1.3.1 GetCapabilities

The GetCapabilities operation is mandatory. The operation allows a client to retrieve capabilities document (metadata) from a server. The response XML document contains service metadata about the server and all implemented processes description.

| Name | Optionality and use | Definition and format |
|---|---|---|
| service=WPS | Mandatory | Service type identifier text |
| request=GetCapabilities | Mandatory | Operation name text |
| AcceptVersion=1.0.0 | Optional | Specification version |
| Sections=All | Optional | Comma-separated unordered list of sections |
| updateSequence=XXX | Optional | Service metadata document version |
| AcceptFormats=text/xml | Optional | Comma-separated prioritized sequence of response formats |

Table 2: GetCapabilities operation request URL parameters, source: [8]

**GetCapabilities request**

- *service* - A mandatory parameter, WPS is only possible value.
- *request* - A mandatory parameter, GetCapabilities is only possible value.
- *version* - An optional parameter, version number. Three non-negative integers separated by a decimal point. Servers and their clients should support at least one defined version.
- *sections* - An optional parameter that contains a list of section names. Possible values are: *ServiceIdentification, ServiceProvider, OperationsMetadata, Contents, All.*
- *updateSequence* - An optional parameter for maintaining the consistency of a client cache of the contents of a service metadata document. The parameter value can be an integer, a timestamp, or any other number or string.
- *format* - An optional parameter that defines response format.

A client can request the GetCapabilities operation with parameters from the Tab. 2. A corresponding request URL looks like:

```
http://localhost:5000/wps?service=WPS&request=GetCapabilities&AcceptVersion=1.0.0&Section=ServiceIdentification,OperationsMetadata&updateSequence=XXX&AcceptFormats=text/xml
```

**GetCapabilities response** When GetCapabilities operation is requested a client retrieve service metadata document that contains sections specified in *sections* parameter. If the parameter value is *All* or not specified than all sections are retrieved.

- *ServiceIdentification* - Server metadata.
- *ServiceProvider* - Server operating organization metadata.
- *OperationsMetadata* - Metadata about operations implemented by the WPS server, including URLs to request them.
- *ProcessOfferings* - List of processes with name and brief description implemented by the WPS server.

In addition to sections each GetCapabilities response should contain:

- *version* - Specification version for GetCapabilities operation.
- *updateSequence* - Server metadata document version, value is increased whenever any change is made in complete service metadata document.

**GetCapabilities exceptions** In case that WPS server encounters an error a client retrieves an exception report message with one of the exception code:

- *MissingParameterValue* - GetCapabilities request does not contain a required parameter value.
- *InvalidParameterValue* - GetCapabilities request contains an invalid parameter value.
- *VersionNegotiation* - Any version from AcceptVersions parameter list does not match any version supported by the WPS server.
- *InvalidUpdateSequence* - Value of updateSequence parameter is greater than current value of service metadata updateSequence number.
- *NoApplicableCode* - Other exceptions.

### 1.3.2 DescribeProcess

The DescribeProcess operation is mandatory. The operation allows clients to retrieve a detailed description of one or more processes implemented by a WPS server. The detailed information describes both required inputs and produced outputs and allowed formats.

| Name | Optionality | Definition and format |
|---|---|---|
| service=WPS | Mandatory | Service type identifier text |
| request=DescribeProcess | Mandatory | Operation name text |
| version=1.0.0 | Mandatory | WPS specification version |
| Identifier=buffer | Optional | List of one or more process identifiers, separated by commas |

Table 3: DescribeProcess operation request URL parameters, source: [8]

#### DescribeProcess request

- *service* - Mandatory parameter, WPS is only possible value.
- *request* - Mandatory parameter, DescribeProcess is only possible value.
- *version* - Mandatory parameter, version number. Three non-negative integers separated by decimal point. Servers and their clients should support at least one defined version.
- *Identifier* - Optional parameter, list of process names separated by comma. Another possible value is *all*.

The DescribeProcess operation can be requested with parameters from Tab. 3. A corresponding request URL looks like: `http://localhost:5000/wps?request=DescribeProcess&service=WPS&identifier=all&version=1.0.0`

**DescribeProcess response** A response to DescribeProcess request contains one or more process descriptions for requested process identifiers. Each process description contains detailed information about process in ProcessDescription XML

element (see Tab. 4) including process inputs and outputs description. The number of inputs or outputs is not limited.

| Name | Optionality | Definition and format |
|---|---|---|
| Identifier | Mandatory | Process identifier |
| Title | Mandatory | Process title |
| Abstract | Optional | Brief description |
| Metadata | Optional | Reference to more metadata about this process |
| Profile | Optional | Profile to which the WPS process complies |
| processVersion | Mandatory | Release version of process |
| WSDL | Optional | Location of a WSDL document that describes this process |
| DataInputs | Optional | List of the required and optional inputs |
| ProcessOutputs | Mandatory | List of the required and optional outputs |
| storeSupported | Optional | Complex data outputs can be stored by WPS server |
| statusSupported | Optional | Execute response can be returned quickly with status information |

Table 4: Parts of ProcessDescription data structure, source: [6]

Three data types of input or outputs exist:

- *LiteralData* - any string. It is used for passing single parameters like numbers or text parameters. There can be set *allowedValues* restriction. It can be a list of allowed values or input data type. Additional attributes such as *units* or *encoding* can be set as well.
- *ComplexData* - Complex data can be raster, vector or any file-based data, which are usually processed. ComplexData are often result of the process. The input can be specified more using mimeType[3], XML schema or encoding.
- *BoundingBoxData* - BoundingBox data are specified in OGC OWS Common specification as two pairs of coordinates (for 2D and 3D space). They can

[3] *mimeType* - is a standardized way to indicate the nature and format of a document. Browsers often use the MIME type (and not the file extension) to determine how it will process a document.

either be encoded in WGS84 or EPSG code can be passed too. They are intended to be used as definition of the target region.

**DescribeProcess exceptions** In case that WPS server encounters an error a client retrieves an exception report message with one of the exception code:

- *MissingParameterValue* - GetCapabilities request does not contain a required parameter value.
- *InvalidParameterValue* - GetCapabilities request contains an invalid parameter value.
- *NoApplicableCode* - Other exceptions.

### 1.3.3 Execute

The Execute operation is mandatory. The operation allows clients to run a specified process implemented by a server. Inputs can be included directly in the request body or be referenced as a web-accessible resource. The outputs are returned in XML response document, either directly embedded within the response document or stored as a resource accessible by returned URL.

| Name | Optionality | Definition and format |
|---|---|---|
| service | Mandatory | Service type identifier text |
| request | Mandatory | Operation name text |
| version | Mandatory | WPS specification version |
| Identifier | Mandatory | Process identifier |
| DataInputs | Optional | List of inputs provided to this process execution |
| ResponseForm | Optional | Response type definition |
| language | Optional | Language identifier |

Table 5: Parts of Execute operation request, source: [6]

**Execute request** Execute request is usually sent via HTTP POST request. It triggers an execution of a specified process if no exceptions raised (for instance ServerBusy). Execute request contains parameters from Tab. 5. Example of Execute XML response document sent within the POST request can be found at App. A.

**Execute response** Usually the Execute operation response document is an XML document. The only exception is in case when a response form of *RawDataOutput* is requested, execution is successful and only one complex output is created, then directly the produced complex output is returned. The result can be inserted directly inline the response document or be referenced as web-accessible resource, it depends on ResponseForm request elements.

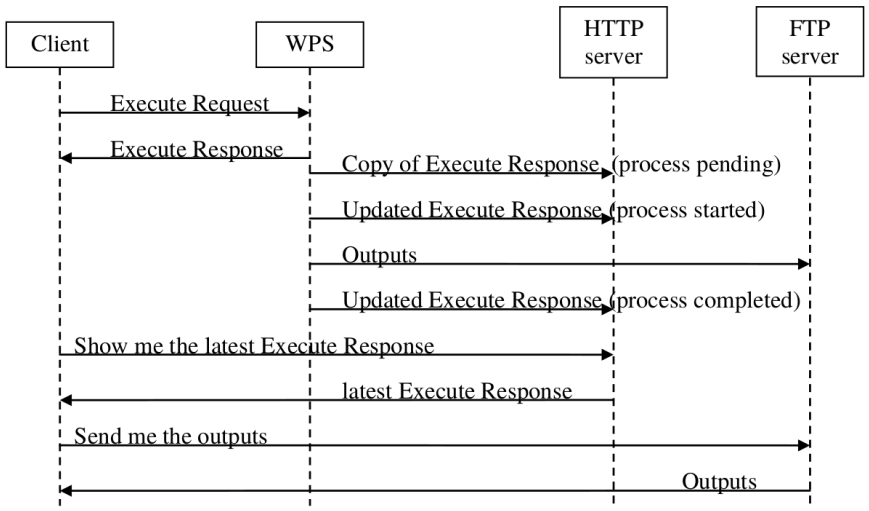



Figure 2: Sequence diagram: a client requests storage of results, source: [6]

In synchronous mode the response document is returned when the process execution is completed. However in asynchronous mode it is possible to get response document right after sending a request. In this case, returned response document contains a URL link from which the final response document can be retrieved after completed process execution. A client can request execution status update to find out the amount of processing remaining if the execution is not completed. Shown in Fig. 2.

| Name | Optionality | Definition and format |
|---|---|---|
| service | Mandatory | Service type identifier text |
| version | Mandatory | WPS specification version |
| language | Mandatory | Language identifier |
| statusLocation | Optional | Reference to location where current ExecuteResponse document is stored |
| serviceInstance | Mandatory | Reference to location where current ExecuteResponse document is stored |
| Process | Mandatory | Process description |
| Status | Mandatory | Execution status of the process |
| DataInputs | Optional | List of inputs provided to this process execution |
| OutputDefinitions | Optional | List of definitions of outputs desired from executing this process |
| ProcessOutputs | Optional | List of values of outputs from process execution |

Table 6: Parts of ExecuteResponse data structure, source: [6]

# 2 WPS implementations

The OGC WPS is just an interface standard that provides rules for standardizing requests and responses. It also defines how clients can request the execution of defined processes and how the outputs are handled. There are several projects that implement this standard across the platforms or programming languages.

## 2.1 deegree

*deegree* is open-source community-driven project for spatial data infrastructure written in Java. Besides from the other OGC Web Services it implements also WPS standard 1.0.0. The implementation offers sending request with KVP, XML or SOAP encoding, asynchronous/synchronous execution and API for implementing processes in Java. On their website there is a WPS demo [4] where all operations *GetCapabilities*, *DescribeProcess* and *Execute* with various processes can be tested.


Figure 3: deegree project logo

## 2.2 52°North WPS

The *52°North* is the open-source software initiative. It is an international network of skilled specialists from research, public administration or industry. The initiative works on several projects and develop new technologies. One of them is the 52°North WPS project.


Figure 4: 52°North project logo

[4] `http://demo.deegree.org/wps-workspace/`

The WPS project is full Java-based open-source implementation of the WPS 1.0.0. The back-end side implements only version 1.0.0 and it does not seem there is any progress in implementation of version 2.0.0. On the other hand on the 52°North GitHub there is a repository *wps-js-client*[5] that is standalone Javascript WPS Client. The client enables building and sending requests against both WPS 1.0.0 and WPS 2.0.0 instances as well as reading the responses.

52°North offers synchronous/asynchronous invocation with both HTTP-GET and HTTP-POST request. All results can be stored as a web-accessible resource, WMS, WFS or WCS layer. Raw data inputs/outputs are also supported. Various extensions for different computional backends exist: WPS4R (R Backend), GRASS extension, Sextante or ArcGIS Server Connector.

## 2.3 GeoServer

*GeoServer* is Java-based server to store, view or edit geospatial data. Designed for interoperability, GeoServer conforms all OGC standards. More famous WMS, WFS and WCS services are part of GeoServer core, however WPS implementation is available as extension.


Figure 5: GeoServer logo

The WPS extension is capable of direct reading and writing data from and to GeoServer. Therefore it is possible to create processes based on inputs served from GeoServer as well as storing the outputs in the catalog.

Since GeoServer implements WPS standard 1.0.0, it supports the *GetCapabilities*, *DescribeProcess* and *Execute* operations. Apart of these, it also implements *GetExecutionStatus* and *Dismiss* operations. The *Dismiss* operation serves for asynchronous requests to get progress report and eventually retrieve the result data. A

[5] `https://github.com/52North/wps-js-client`

client sends in the GetExecutionStatus request a mandatory *executionId* parameter to specify the process. The executionId is also mandatory parameter for Dismiss operation. The Dismiss operation cancels an execution of the process of given executionId. As seen in Fig. 7, GeoServer offers *Progress status page* where progress of all executions can be reviewed as well as dismissing of each execution can be done.

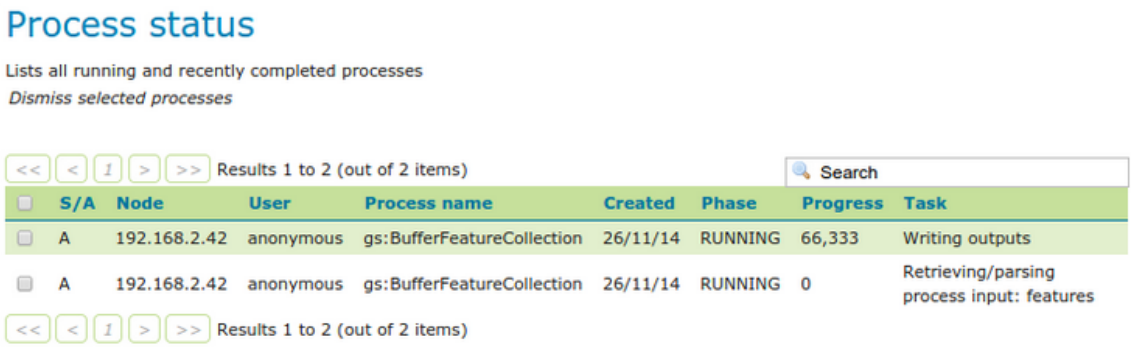



Figure 6: Process status page, source [10]

## 2.4 ZOO-Project

*ZOO-Project* is a WPS implementation writen in C, Python and Javascript. It is an open-source project released under MIT licence. The platform is composed of several components:

- WPS Server - ZOO-kernel is a server-side implementation written in C.
- WPS Services - ZOO-services is a set of ready-to-use web services based on libraries such as *GDAL*, *GRASS GIS* or *CGAL*.
- WPS API - ZOO-API is a server-side Javascript API for creating and chaining WPS web services.
- WPS Client - ZOO-client is a client-side Javascript library for interacting with WPS Services.

ZOO-Project is the first and in this time probably the only one full implementation of the WPS 2.0.0 standard. Apart from *GetCapabilities*, *DescribeProcess* and

*Execute* operations from WPS 1.0.0 standard it also implements *GetStatus*, *GetResult* and *Dismiss* operations from WPS 2.0.0.

To comply WPS 2.0.0 ZOO-Project must support synchronous/asynchronous invocation with both HTTP-GET and HTTP-POST request. There is optional MapServer support so an output can be stored in MapServer catalog. It is convenient to publish results directly as WMS, WFS or WCS resources.

## 2.5 ArcGIS Server

*ArcGIS Server* is server-side GIS software developed by *Esri*. It is capable of creating and managing GIS Web services, applications and data. It allows exposing the analytic capability of ArcGIS to web as a *Geoprocessing service*. A geoprocessing service consists of one or more geoprocessing tasks. A geoprocessing task can be any ArcGIS tool. It is possible to publish Geoprocessing service with the WPS capabilities enabled, however only WPS 1.0.0 standard is supported.


Figure 7: Esri logo

All published services have specified the minimum and maximum number of available instances. These instances run on the container machines within processes. The isolation level determines whether these instances run in separate processes or shared processes.

- *High isolation*- Fig. 2.5 - each instance runs in its own process. If something causes the process to fail, it will only affect the single instance running in it.
- *Low isolation* - Fig. 2.5 allows multiple instances of a service configuration to share a single process, thus allowing one process to handle multiple concurrent, independent requests. This is often referred to as multithreading.

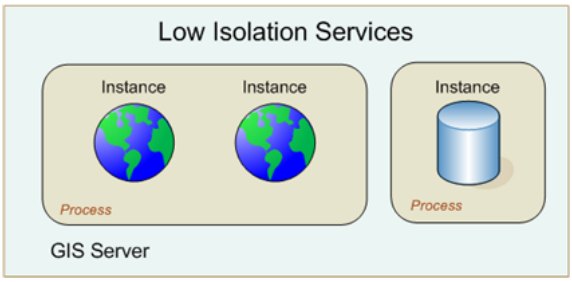



Figure 8: Low isolation, source [12]

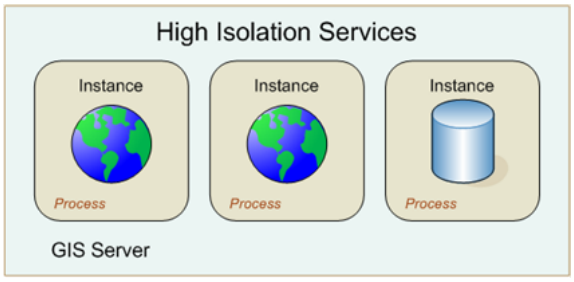



Figure 9: High isolation, source [12]

The advantage of low isolation is that it increases the number of concurrent instances supported by a single process. Using low isolation can use slightly less memory on your server. However, this improvement comes with some risk. If a process experiences a shutdown or crash, all instances sharing the process are destroyed. It is strongly recommended that you use high isolation.[12]

## 2.6 PyWPS

PyWPS is a server-side implementation of the WPS standard written in Python. This project will be described in depth in the Sec. 3.

# Part II

# PyWPS

# 3 PyWPS

## 3.1 History

The origin of PyWPS started in 2006 as a student project. The first presentation was held at the FOSS4G 2006 conference in Lausanne titled 'GRASS goes to web: PyWPS'. During November 2006 the version 1.0.0 was released together with WUIW and Embrio projects that brought the funcionality of GRASS GIS and general web interface able to handle any WPS server.[15][16]

In 2007 PyWPS 2.0.0 was released supporting WPS standard 0.4.0. New version improved stability and approached on the standard implementation. It came with new WPS client and WPS plugin for OpenLayers [6].

Next year in 2008 PyWPS 3.0.0 was released with support for WPS 1.0.0. It was possible to run multiple WPS instances with one PyWPS installation. This version had simple code structure and contained examples of processes.

The newest version is PyWPS 4.0.0 from 2016 when PyWPS-4 branch was merged to official PyWPS repository as its master branch. New version is described in following Sec. 3.2.


Figure 10: PyWPS project logo

## 3.2 PyWPS 4.0

PyWPS-4 is the most current version of PyWPS. Rewriting from scratch involved these major changes:

- It is written in *Python 3* with backward support for Python 2.7.

[6] `http://openlayers.org/`

- It utilizes native Python bindings to existing projects (GRASS GIS).
- New popular formats like *GeoJSON*, *KML* or *TopoJSON* are reflected and their support is provided.
- PyWPS project has changed the license from *GNU/GPL* to *MIT*.
- PyWPS 4.0 is implemented using the *Flask* framework.
- A C-based XML parser *Lxml* is used to handle XML files.
- *OWSLib* structures are used for some data types.

## 3.3 PyWPS-demo

PyWPS-demo is a small side project distributed with PyWPS. It is a simple demo instance of PyWPS server running on *Flask*[7]. Flask is a microframework for web applications in Python. Flask provides built-in development server and debugger and RESTful request dispatching. Starting PyWPS-demo server with Flask is very simple and can be done with command in Lst. 3. After starting the PyWPS-demo server the PyWPS homepage can be visited at: `http://localhost:5000`.

Listing 3: Starting PyWPS-demo server

```
python3 demo.py
```

PyWPS-demo comes with several demo processes:

- *area.py* - Process calculates area of given polygon.
- *bboxinout.py* - Process transforms bounding box to another EPSG.
- *buffer.py* - Process returns buffers around the input features, using the GDAL library.
- *centroids.py* - Process returns a GeoJSON with centroids of features from an uploaded GML.

[7] `http://flask.pocoo.org/`

- *feature_ count.py* - Process counts the number of features in an uploaded GML.
- *grassbuffer.py* - Process uses the GRASS GIS *v.buffer* module to generate buffers around inputs.
- *sayhello.py* - Process returns a literal string output with Hello plus the inputed name.
- *sleep.py* - Process will sleep for a given delay or 10 seconds if not a valid value.
- *ultimate_ question.py* - The process gives the answer to the ultimate question of "What is the meaning of life?"

Except these example processes the demo offers also example configuration file. Configuration file contains several parameters in these four sections:

- *metadata* - parameters containing information for metadata creation.
- *server* - definition of path to workdir and output directories, maximum number of parallel running or stored processes.
- *logging* - logging level setting, path to log file and log database.
- *grass* - GRASS settings.

# 4 Process isolation in PyWPS

## 4.1 Asynchonous requests

Right now in PyWPS 4.0 version a PyWPS server instance is able to run multiple concurrent processes in parallel. The server is configured for maximal amounts of concurrently running processes at the same time and for maximum of waiting processes in a queue, to later start their execution once new slots are available. If the new Execute request is received and the maximal amount is exceeded, the request is rejected and user is informed in response (see Lst. 4).

Listing 4: Resource exceeded exception

```
<?xml version="1.0" encoding="UTF-8"?>
<ows:ExceptionReport xmlns:ows="http://www.opengis.net/ows/1.1"
    version="1.0.0">
    <ows:Exception exceptionCode="ServerBusy">
        <ows:ExceptionText>
            Maximum number of parallel running processes reached.
                Please try later.
        </ows:ExceptionText>
    </ows:Exception>
</ows:ExceptionReport>
```

To facilitate the management of concurrent processes, process metadata are stored into a local database. This database is used for logging and saving waiting Execute requests in the queue and invoking them later on. The database will also enable the implementation of pausing, releasing and deleting running process. These features will allow PyWPS to comply with WPS version 2.0.0.

## 4.2 Current state

At the beginning of every process execution its own temporary directory *workdir* is created. During the execution temporary files and continuous outputs are stored in this directory. After successful execution final outputs are moved to *outputs*

directory. Both directories *outputs* and *workdir* are configurable and user can change path to them.

Listing 5: pywps.cfg - mode parameter

```
[processing]
mode=multiprocessing
```

Current version of PyWPS offers two solutions for running parallel processes:

- Multiprocessing
- Job Scheduler Extension[8]

If the execute request is sent asynchronously the type of process constructor is chosen depending on configuration parameter *mode* in section *processing* which is by default *multiprocessing* or can be changed to *scheduler*.

**Multiprocessing** By default for processes running in the background, the Python *multiprocessing* module is used – this makes it possible to use PyWPS on the Windows operating system too.

The number of processes running in parallel is configirable by parameter *parallelprocesses* of section *server* in configuration file. In the Fig. 11 two running processes are shown. A client sends an Execute request to a server. Server sends back to the client an ExecuteResponse that *Process1* (green in the figure) was accepted and starts its execution. During the execution the process updates its status. The interval of status updates depends on the code of the Process1. Process1 must support status update otherwise it cannot be run in asynchronous mode.

During the execution of Process1 server receives another Execution request. It sends back the Execution response and starts the execution of *Process2* (blue in the figure). Separated Python *Process*[9] is created. Both of the processes run on the host machine, however both have own memory space. Their executions run concurently and client can request their status. In the figure, the Process2 ended

[8]Job Scheduler Extension is currently only in *develop* branch of PyWPS.

[9]Explanation of term Python *Process* and its differences to *Thread* is in next paragraph.

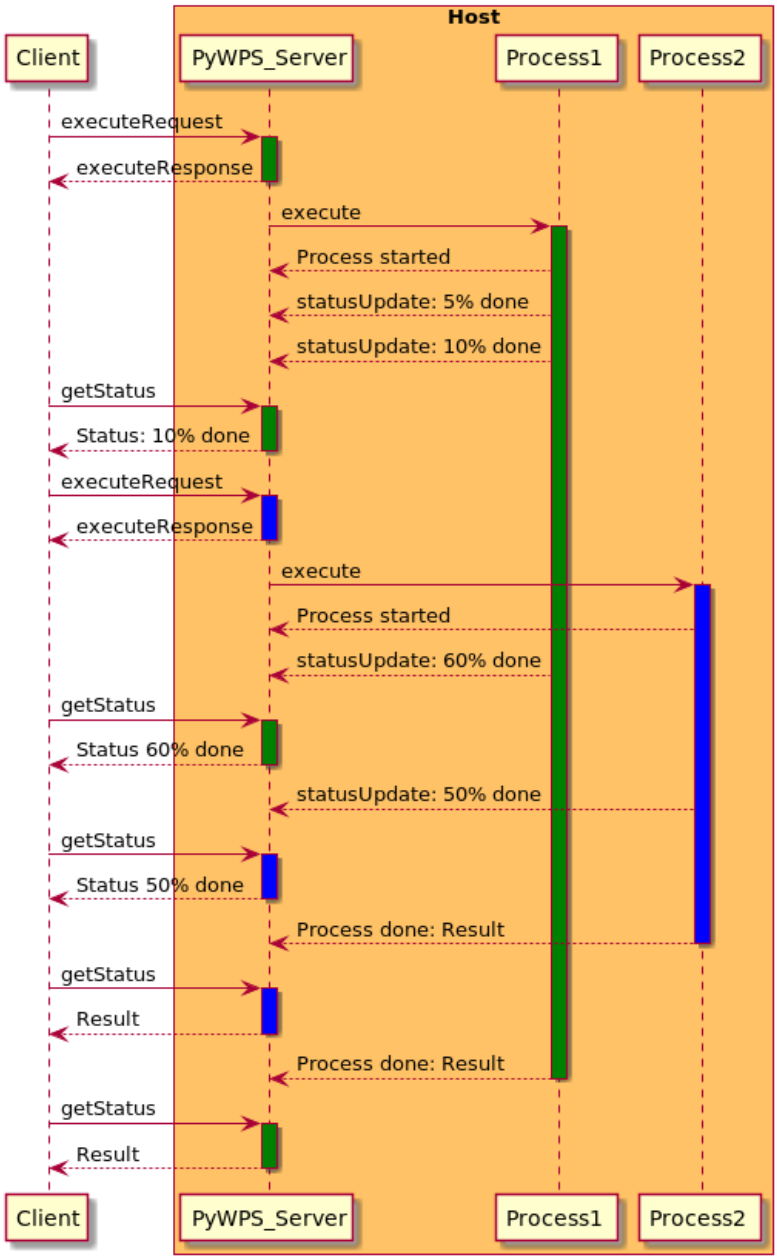



Figure 11: Sequence diagram: Multiprocessing

first and client can retrieve the result from the server. Once the Process1 ends, the client can retrieve its result from the server as well.

It is important to say that in case of multiprocessing, processes run concurently with its own memory space, nevertheless they are not isolated. They run on the same host machine and share the resources. There are even methods like *Pipe()* that enable communication between processes.

**Process vs Thread** In Python there are two ways to achieve *pararellism*. It is *multiprocessing* [10] with using processes and *threading*[11] with threads. The main difference is that threads run in the same memory space, while processes have separate memory. Multiprocessing takes advantages of multiple CPUs and cores while threads are more lightweighted and have low memory footprint. In case of PyWPS asynchronous requests, for every execution its own process with its own memory space is created.

**Job Scheduler Extension** PyWPS scheduler extension offers possibilities to execute asynchronous processes out of the WPS server machine. This extension enables to delegate execution of processes to a scheduler system like *Slurm*, *Grid Engine* and *TORQUE* from Adaptive Computing. These schedular systems are usually located at *High Performance Compute (HPC)* centers.


Figure 12: Grid Engine


Figure 13: Slurm


Figure 14: TORQUE

The PyWPS scheduler extension uses the Python *dill* library to dump and load the processing job to/from filesystem. The batch script executed on the scheduler system calls the PyWPS *joblauncher* script with the dumped job status and executes the job (no WPS service running on scheduler). The job status is updated on the filesystem. Both the PyWPS service and the joblauncher script use the same PyWPS configuration. The scheduler assumes that the PyWPS server has a shared filesystem with the scheduler system so that XML status documents and WPS outputs can be found at the same file location. The interaction diagram how the communication between PyWPS and the scheduler works is displayed in Fig. 16.

[10] `https://docs.python.org/3/library/multiprocessing.html`
[11] `https://docs.python.org/3/library/threading.html`

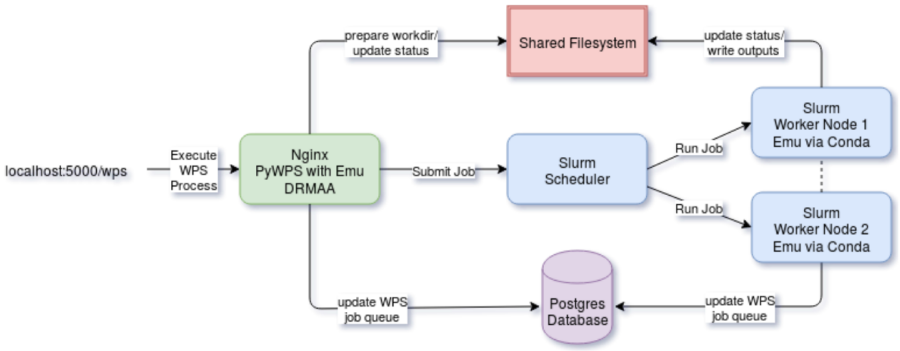



Figure 15: Example of PyWPS scheduler extension usage with Slurm, source: [17]

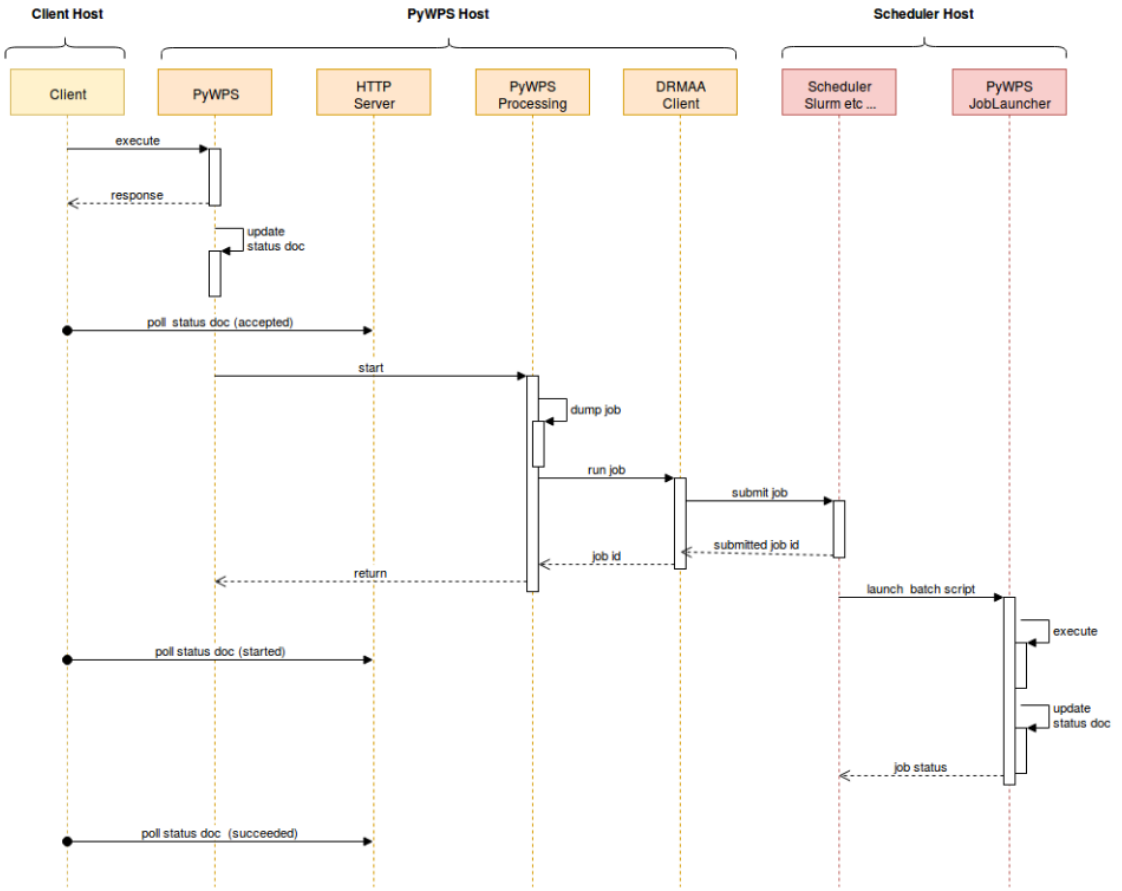



Figure 16: Communication between PyWPS and scheduler, source: [17]

## 4.3 Possible solutions for process isolation

In previous section there were described two mechanisms for running parallel processes. Nevertheless in case of Python module *Multiprocessing* the processes are not really isolated. They run concurrently but they can share resources and there are even methods like *Pipe()* that enables communication between processes.

On the other hand *Job Scheduler Extension* depends on *dill* library as well as on some external scheduler systems like *Slurm*, *Grid Engine* or *TORQUE*.

In this section there are described some other solutions. Some were suggested by PyPWS developers with encouragement to make a feasible study. Others were discovered during research on the internet forums like StackOverflow and few of them were referenced in the documentation of other projects. During the research two requirements were considered.

- The solution provides a mechanism for full isolation. This is a must-have requirement.
- The solution provides a mechanism for start/pause/stop process execution. This is a nice-to-have requirement as this functionality will be required to comply WPS 2.0.0 standard.

Finnaly these solutions were considered:

- Celery
- Docker
- psutil
- SandboxedPython
- VM

### 4.3.1 Celery

*Celery* is a task queue system written in Python. It helps to distribute work across threads and even machines. Basic term is a *task*. A task is a unit of work and it is an input into the task queue. The task queue is constantly monitored for new work to perform.

To communicate between client and workers Celery uses a *broker*. The communication is via messages. To initiate a task the client adds a message to the queue and the broker then delivers the message to a worker. Multiple workers and brokers can be added so there is assured high availability and horizontal scaling.

Celery provides worker remote control client in class *celery.app.control.Control*. The class offers following functions:

- **revoke** - Tell all (or specific) workers to revoke a task by id. If a task is revoked, the workers will ignore the task and not execute it after all.
- **shutdown** - Shutdown worker(s).
- **terminate** - Tell all (or specific) workers to terminate a task by id.

### 4.3.2 Docker

*Docker* is one of the most used technology regarding containerization. This technology is described in depth in Sec. **??**

### 4.3.3 psutil

*psutil* is Python library for process and system management. It handles system monitoring, limiting process resources and the management of running processes. Its implementation is based on UNIX command line tools. psutil offers functions applied to these sections:

- CPU - functions for CPU statistics such as CPU utilization percentage, frequency and others.
- Memory - functions for system memory usage and swap memory statistics.

- Disks - functions for disk statistics such as disk usage or disk IO operations counter.
- Network - functions for network IO operations or network connection statistics.
- Sensors - functions for statistics about fans, battery or hardware temperature.
- Others - functions for boot time and users statistics.
- Processes - functions will be described in detail later.

**Processes** - Class *psutil.Process* represents an OS process with given pid. The class is bound with a process via its PID[12]. The *Process* class offers these methods for starting/pausing:

- suspend() - The method suspends a process using *SIGSTOP* signal.
- resume() - The method resumes a process using *SIGCONT* signal.
- terminate() - The method terminates a process using *SIGTERM* signal.
- kill() - The method kills a process using *SIGKILL* signal.

### 4.3.4 Sandboxed Python

The general goal of a sandbox is to run applications securely inside isolated environment they cannot escape from and affect other parts of the system. Developers use them to run untrusted code inside. It is quite difficult to develop fully sandboxed solution due to Python complexity. The basic problem is that Python introspection allows several ways to escape out of the sandbox. True security requires an overall design with many security considerations included. Some of the projects that can run Python code in a sandbox are:

- PyPy
- Jython

[12] *PID* is a process identifier. It is a number used by operating system to uniquely identify an active process.

**PyPy** PyPy is Python interpreter written in RPython that implements full Python language and very closely emulates the behavior of CPython. PyPy offers fully portable sandboxing feature similar to OS-level sandboxing (e.g. SECCOMP). It is not sandboxing at the Python language level so it does not put any restriction on any Python functionality.

Untrusted Python code that is intended to be sandboxed is launched in a subprocess, that is a special sandboxed version of PyPy. All its inputs/outputs are not directly performed but are serialized to a stdin/stdout pipe. The outer process reads the pipe and afterward decides which commands are allowed.

**Jython** Jython is Python language interpreter for Java. Java offers strong sandboxing mechanisms. The security facility in Java that supports sandboxing is the *java.lang.SecurityManager*. By default, Java runs without a SecurityManager.

**pysandbox** A prove, that it is very difficult to develop some kind of sandbox with all security holes considered, could be a project *pysandbox*[13]. After working on it for 3 years, during which the project was used on various production servers by other developers, its author declared that the project is broken by design. In his post to the python-dev mailing list [18] the author explained that with every vulnerability founded it became more difficult to actually write a real code:

*"To protect the untrusted namespace, pysandbox installs a lot of different protections. Because of all these protections, it becomes hard to write Python code. Basic features like "del dict[key]" are denied. Passing an object to a sandbox is not possible to sandbox, pysandbox is unable to proxify arbitary objects.*

*For something more complex than evaluating "1+(2*3)", pysandbox cannot be used in practice, because of all these protections. Individual protections cannot be disabled, all protections are required to get a secure sandbox."*

### 4.3.5 Virtual Machine/Vagrant

Using full virtualization for process isolation is mentioned here but in fact it is hard to imagine this solution could work in practice. *Vagrant* is a tool for managing

[13] `https://github.com/vstinner/pysandbox`

and building virtual machines. It provides a way how to manage various virtual machines in an automatized way e.g. using scripts. There also exists a Python package *python-vagrant* that offers Python bindings for interacting with Vagrant.

However in our use-case using Vagrant would mean that for every process execution a separate virtual machine would be created. Depending on the process algorithm complexity the process execution can last from milliseconds to hours or days. On the other hand building a virtual machine and booting into it last at least few seconds. That is why it is hard to imagine using virtual machine, which takes few seconds to boot up, to isolate process, which execution lasts less than a second.

# 5 Docker

sec:Docker **Containerization** is a lightweight alternative to full machine virtualization. It involves encapsulating an application into a container with its own operating environment. It helps to run a containerized application on any physical machine without any worries about dependencies. The origin of containerization lies in the *LinuX Containers LXC* format. Containerization works only in Linux environments and can run only Linux applications.


Figure 17: Docker logo

Docker is not the only technology for containerization. Other alternatives exist, it is *Kubernets*, *CoreOS rkt*, *Open Container Initiative (OCI)*, *Canonical's LXD*, *Apache Mesos and Mesosphere* and many others. However Docker is a leader on the field of containerization and with most public traction is de facto considered as a container standard. That's why the Docker was chosen for this thesis as a container technology. So from this point on any term *container* refers to Docker container.


Figure 18: Kubernetes


Figure 19: CoreOS rkt


Figure 20: Canonical's LXD


Figure 21: Apache mesos

**Docker** is a Linux container technology that allows package and ship applications and everything it needs to execute into a standard format, and run them on any infrastructure.

## 5.1 Virtual machine vs. Docker container

Both virtual machines and Docker containers are two ways how to deploy multiple, isolated applications on a single platform. They both offer a way to isolate an application and its dependencies into a self-contained unit that can run anywhere. They both offer some kind of virtualization. They differ in architecture, see Fig. 22, 23.

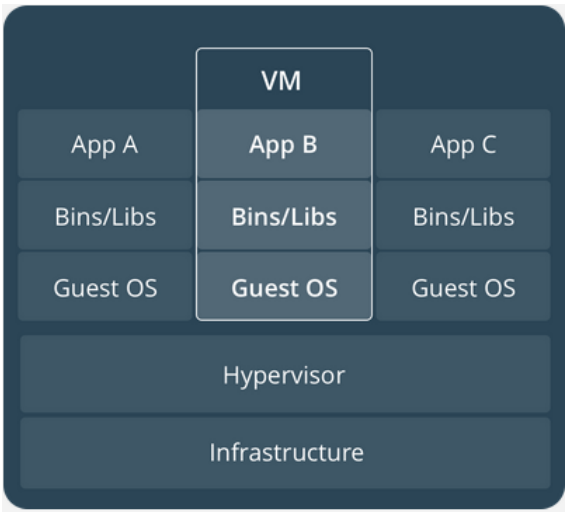



Figure 22: Virtual machine architecture, source [13]

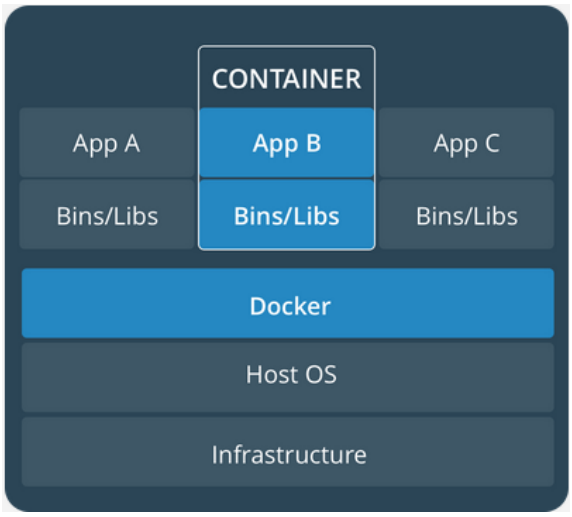



Figure 23: Containers architecture, source [13]

### 5.1.1 Virtual machine

Let's start with a virtual machine (Fig. 22) and its layers description from the bottom up:

- *Infrastructure* - It can be a PC, developer's laptop, a physical server in data-center but as well a virtual private server in the cloud as Microsoft Azure or Amazon Web Services.

- *Host OS* - Host operating system. In case of native hypervisor this layer is missing. In case of hosted hypervisor it is probably some distribution of Linux, Windows or MacOS.
- *Hypervisor* - Also called virtual machine monitor (VMM). It allows hosting several different virtual machines on a single hardware. There are two types of hypervisors:
  - Type 1 - Also called *bare metal* or *native*. This type is run on the host's hardware to control it as well as manage the virtual machines on it. It is much faster and more efficient. This type hypervisors are KVM, Hyper-V or HyperKit.
  - Type 2 - So called *embedded* or *hosted* hypervisors. These hypervisors are run on a host OS as a software. They are slower and less efficient on the other hand they are much easier to set up. It includes VirtualBox or VMWare Workstation.
- *Guest OS* - Guest operating system. Each VM requires own guest operating system which is controlled by the hypervisor. Each guest OS needs its own CPU and memory resources and starts on hundreds of megabytes in size.
- *Bins/Libs* - Each guest OS needs various binaries and libraries for running the application. It can be *python-dev* or *default-jdk* packages as well as personal packages to run the application.
- *Application* - The application source code that is desired to be run isolated. Therefore each application or each version of the application has to be run inside of its own guest OS with own copy of bins and libs.

### 5.1.2 Docker container

Now, what is different regarding containers (Fig. 23):

- *Infrastructure* - PC, laptop, physical or virtual server.
- *Host OS with container support* - Any OS capable of run Docker. All major distributions of Linux are supported and there are ways to run Docker even on MacOs and Windows too.

- *Docker engine* - Also called Docker daemon. It is a service that runs in the background on host operating system. It manages all interaction with containers.
- *Bins/Libs* - Binaries and libraries required by the application. They get built into special packages called *Docker images*. The Docker daemon runs those images.
- *Application* - Each application and its library dependencies get packed into the same Docker image. It is managed independently by the Docker daemon.

But the architecture is not the only one difference:

- Docker uses Docker daemon to manage containers, hypervisor manages virtual machines.
- The Docker daemon communicates directly with host OS and manage resources for each container.
- VMs usually boot up in a minute and more, containers start in seconds.
- Docker virtualizes operating systems, using VMs is hardware virtualization.
- VM and container vary in size. VMs start at hundreds of megabytes. A container can be smaller than one megabyte.
- Containers share the kernel although they are isolated. VMs are monolithic and stand-alone.

## 5.2 Dockerfile

Dockerfile is a core file that contains the instruction to be performed when an image is built. It usually consists of commands to install packages, calls to other scripts, setting environmental variables, adding files or setting permissions. In Dockerfile there is also defined what image is to be used as a base image for the build.

### Dockerfile instructions

- *FROM* - The FROM instruction defines the base image for next instructions and initializes a new build stage. Every Dockerfile has to start with FROM command. The only exception is ARG command which can be before FROM command.
- *ARG* - The ARG instruction defines a variable that users can pass at build-time to the builder.
- *ENV <key>=<value>* - The ENV instruction sets the environment variables. It is key-pair value.
- *LABEL* - The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. It can be anything from version number to a description.
- *ADD <src> <dest>* - The ADD instruction copies files or directories from source and adds them at the destination path. It also unzips or untars files when added.
- *COPY <src> <dest>* - Similar to the ADD instruction it copies files or directories from source and adds them to the destination path. This command doesn't provide any kind of decompression.
- *RUN <command>* - The RUN instruction will execute any defined command and commit the results.
- *CMD ["executable","param1","param2"]* - The CMD instruction provides defaults for an executing container. It can include an executable. In case the executable is omitted the CMD instruction must be used together with the ENTRYPOINT instruction. There can be only one CMD instruction in Dockerfile. In case there is more CMD the last one will be used.
- *ENTRYPOINT* - The ENTRYPOINT defines a container configuration that will run as executable.
- *WORKDIR /path/to/dir* - The WORKDIR instruction sets the working directory for any RUN, CMD, COPY and ADD instruction that follows in Dockerfile.

- *EXPOSE* - The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.
- *VOLUME* - The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from the native host or other containers.

Except for the FROM instruction, all the instructions can be defined from the command line when starting docker container. There are more Dockerfile instructions however they are not relevant to this thesis as there are never used in practical part. A Dockerfile, which was created during the work on the thesis, is available at App. E.

# Part III

# Implementation

# 6 Implementation introduction

## 6.1 pywps-demo

During the implementation the *pywps-demo* (Sec. 3.3) running on *Flask* framework was used. This demo server instance runs on host machine server at port 5000 as well as the image built from its Dockerfile is used for every container creation. For developing purpose some sections were added to configuration file as well some minor changes for instance in server routing were made. The diff file to pywps-demo is in App. G.

### 6.1.1 pywps-demo Dockerfile

*pywps-demo* is also available as a Dockerfile and as mentioned the image built from this Dockerfile is used for container creation. Before the work on this thesis started, the pywps-demo project had offered two dockerfiles, both based on *alpine Linux* distribution. The first one *pywps-flask* was the default implementation using only Flask while the second one *nginx* implements pywps using Nginx and Green unicorn as WSGI server. During the implementation only the *pywps-flask* Dockerfile was used. However it was necessary to modify the Dockerfile because it did not contain *GDAL* library which is required for most of the demo processes that *pywps-demo* offers.

During implementaion a new version of Dockerfile with support of GDAL was created in collaboration with PyWPS developers. There were some issues with *Xerces* libraries whose packages are not available for alpine distribution and its manual instalation was necessary. The newly-created Dockerfile is available in App. E. At the time of finishing this thesis pywps-demo offers dockerfiles based on *alpine* and *ubuntu* Linux distribution.

## 6.2 OWSLib

A Python package *OWSLib* was used for forwarding requests from PyWPS server instance running on host machine to PyWPS server instance running inside a Docker

container. Some bug fixing which is mentioned in Sec.9.2.2 was necessary[14]. Complete diff is available in App. F.

## 6.3 PyWPS

Most changes have been done in core PyWPS project. Almost all changes were made in *processing* module. To this module new file *container.py* containing the *Container* class was added. Complete diff is available at App. H.

[14]Pull request at: `https://github.com/geopython/OWSLib/pull/410`

# 7 Operations overview

PyWPS in current version 4.0.0 implements all mandatory operations: *Execute*, *GetCapabilities*, *DescribeProcess*. Operations are handled by corresponding methods *execute()*, *get_ capabilities()* and *describe()* in *Service* class.

However both *GetCapabilities* and *DescribeProcess* operations run in synchronous mode only. After sending a request, a client receives back GetCapabilities or DescribeProcess response (both detaily described in 1.3.1 and 1.3.2). Both operations return only information or description about process but do not trigger the execution of the process. It is supposed the response to GetCapabilities and DescribeProcess is returned almost immediately. During the GetCapabilities and the DescribeProcess operations a process execution is not started and therefore there is no starting process to be isolated. That is why whole contribution of this thesis only applies to *Execute* operation.

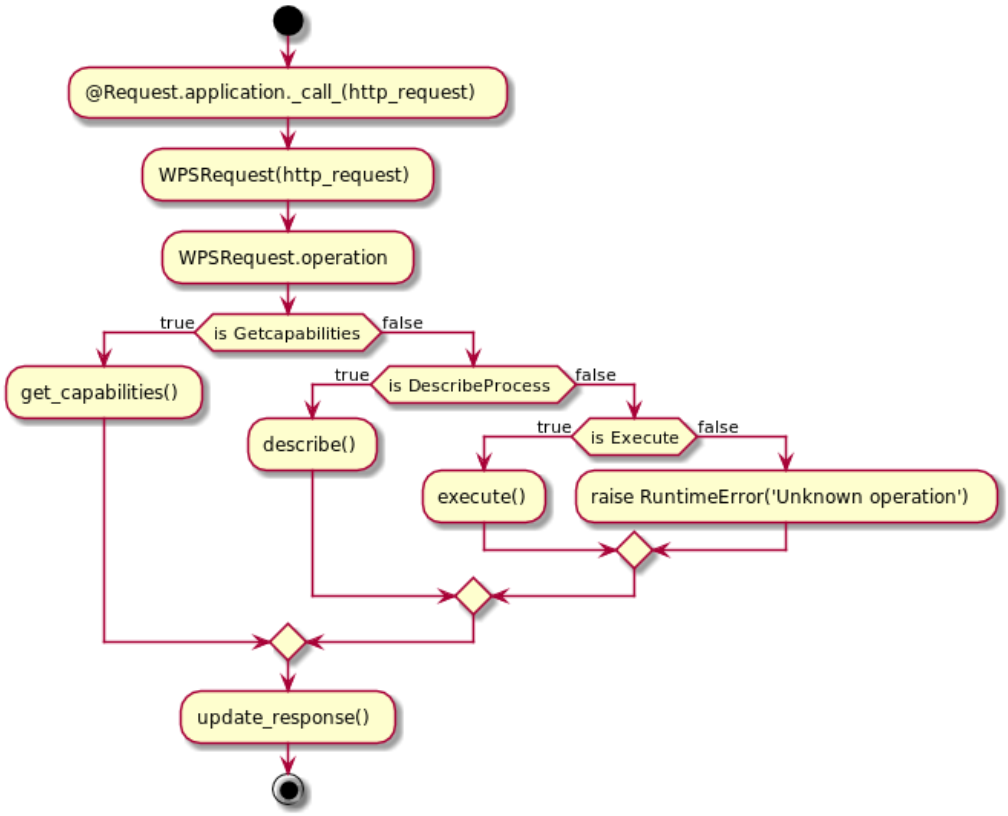



Figure 24: PyWPS operations activity diagram, source: author

# 8 Execute operation

## 8.1 Service.execute()

As mentioned in previous section Sect. 7, *Execute* operation is handled by *execute()* method. Inputs for the method are:

- *identifier* (string) - a name of the process which execution is requested and which is supported by WPS server.
- *wps_request* (WPSRequest object) - an object containing original HTTP request.
- *uuid* (integer) - unique identifier of process execution.

The flowchart of the process execution is displayed in Fig. 25. At first a deepcopy of the process instance is created so that processes cannot override each other. Then a temporary working directory *workdir* is created and set as a current workdir for the process execution. To the workdir all input files are copied as well as all temporary files and outputs are stored here. Then the method *_parse_and_execute()* is called (see Fig. 26). Here the inputs are parsed, in case of a web-referenced input the data are downloaded to workdir, in case of data sent within a request the data are saved into a file in workdir. The process execution afterward runs in *Process.execute()* method. This method returns a *wps_response* - an instance of *WPSReponse* object.

## 8.2 Process.execute()

The method *execute()* of class *Process* contains crucial if-statement where is decided whether the process will be run in asynchronous or synchronous mode. Running in asynchronous mode can be enforced by setting both attributes *status* and *storeExecuteResponse* of the *ResponseDocument* element in the ExecuteRequest XML to True.

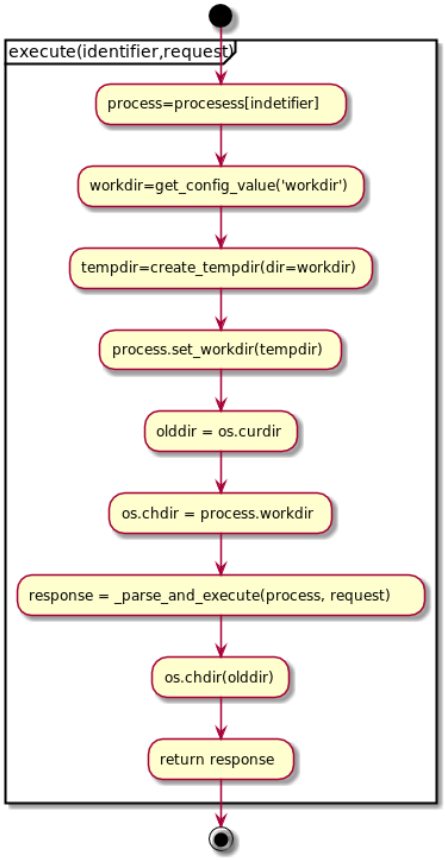



Figure 25: Activity diagram: method *Service.execute()*, source: author

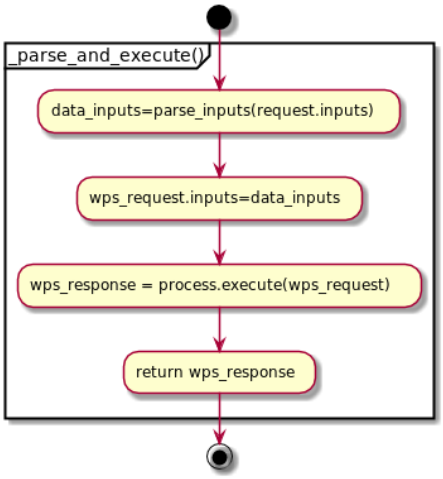



Figure 26: Activity diagram: method *Service._parse_and_execute()*, source: author

Listing 6: ReponseForm element of ExecuteRequest XML

```
<wps:ResponseForm>
 <wps:ResponseDocument status="true" storeExecuteResponse="true">
  <wps:Output asReference="true">
   <ows:Identifier>buff_out</ows:Identifier>
  </wps:Output>
 </wps:ResponseDocument>
</wps:ResponseForm>
```

No matter whether the process runs synchronously or asynchronously there is always a control how many parallel processes are currently running. The number of the maximum of concurrently running processes can be configured. If the process is

asynchronous and the number of currently running processes exceeds the maximal number, the process is stored and its execution is started lately. In case of the synchronous process the *ServerBusy* exception is raised. If the number of processes is smaller than the maximal number of concurrent processes, the process can be executed. In synchronous mode the *_run_process()* is called, in asynchronous mode the method *_run_async()* is called. The activity diagram of the *Process.execute()* is displayed in Fig.27.

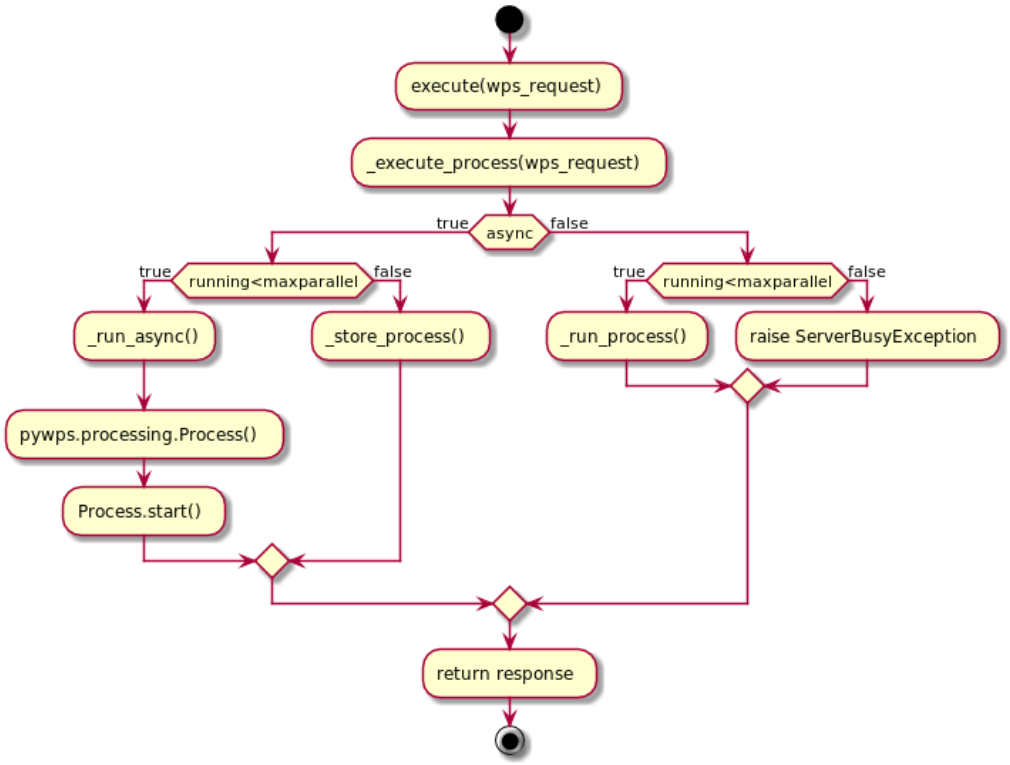



Figure 27: *Activity diagram: Process.execute()*, source: author

## 8.3 Processing module

Until now the operations described in this thesis was not modified. Requirements which have been considered during the implementation of Docker technology were that the source code will be modified slightly, the process isolation will be easily inserted and the project structure will be kept the same. Keeping this in mind changes in source code were made only in *processing* module.

As mentioned in Sec. 4.2, PyWPS uses solely the Python package *Multiprocessing* in production version. In develop branch there is also *Scheduler* extension as one of the option for multiprocessing. In this thesis another option *Docker* for processing was added. The desired option for processing can be configured in configuration file via parameter *mode* in section *processing* (see Lst. 7), possible values are:

- docker - new option
- scheduler
- multiprocessing - default option

Listing 7: Processing mode configuration

```
[processing]
mode=docker/scheduler/multiprocessing
```

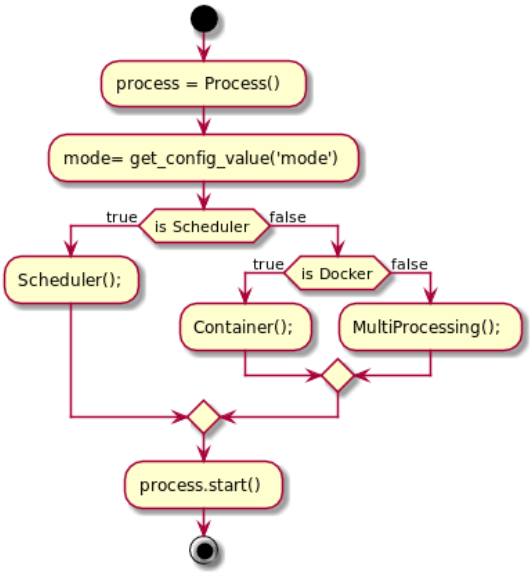



Figure 28: Activity diagram: Method *Process._run_async()*, source: author

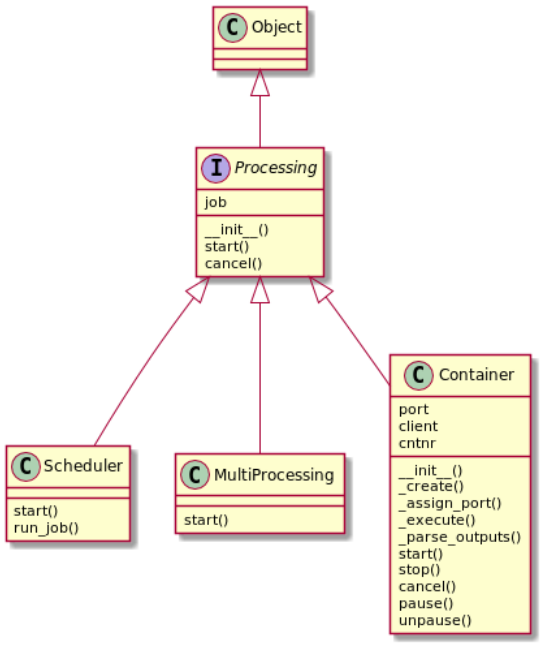



Figure 29: Class diagram: *Processing* class, source: author

The whole Docker implementation is in *Container.py* module. The class *Container* handles containers creation, interaction with server, file-system mounting and all container management.

# 9 *Container* class

The main idea of process isolation using Docker is quite simple. For every process execution one separate Docker container is created. Instead of starting process execution on the host PyWPS server after receiving ExecuteRequest from the client, the ExecuteRequest is forwarded to PyWPS server running inside Docker container. The process execution runs inside the container. After successful process execution the outputs are available at the host server. The host server and the container share the same process workdir at filesystem.

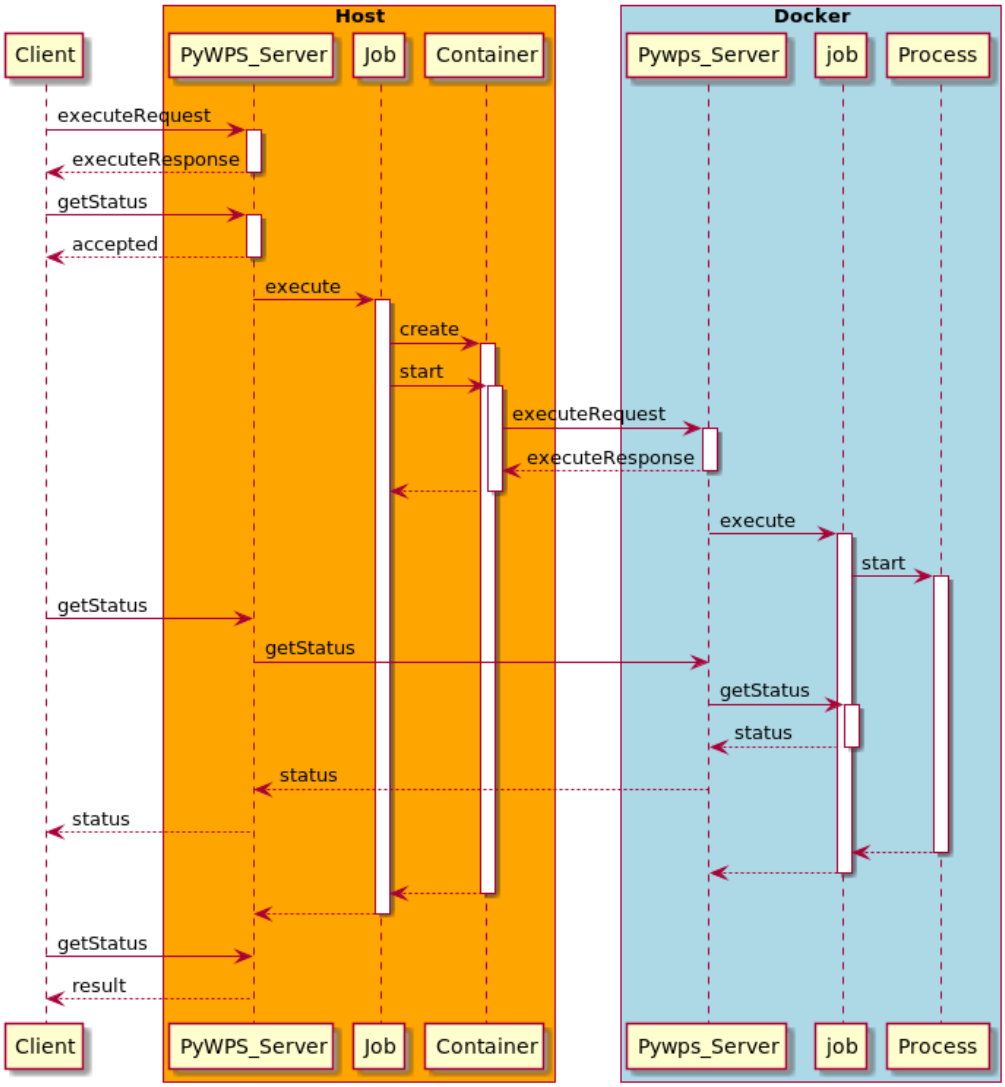



Figure 30: Sequence diagram: Process execution using Docker, source: author

## 9.1 *Container* class constructor

*Container* class is initialized with standard Python method *__init__()*. As an inheritor of *Processing* class, at first the parent constructor *super().__init__()* is called. Follows description of methods which are called inside the constructor method.

Listing 8: *Container* class constructor

```
def __init__(self, process, wps_request, wps_response):
    super().__init__(process, wps_request, wps_response)
    self.port = self._assign_port()
    self.client = docker.from_env()
    self.cntnr = self._create()
```

### 9.1.1 *Container._assign_port()*

The method returns the number of available port. The port is chosen from range <*port_min*, *port_max*> which are both configurable values. If no port from the range is available, the method returns *NoAvailablePortException*. Schema of ports assignment to each container is in Fig. 31.

### 9.1.2 *docker.from_env()*

The *docker* is a Python library for the Docker Engine API. *from_env* method returns an instance of *DockerClient* class which is a client to communicate with the Docker daemon. The returned client is configured from the same variables as the Docker command-line client.

### 9.1.3 *Container._create()*

The *_create* method reads following values at the beginning:

- *cntnr_img* - Name of the image the container will be created from. The name of the image must be the same as the tag set by the *-t* parameter in *docker build* command when the image is built from Dockerfile.

Listing 9: Docker build command

```
docker build -t image_name /path/to/dockerfile
```

- *prcs_inp_dir* - Path to process workdir from *self.job.wps_response.process.workdir*. It is a directory where the inputs for the process are stored.
- *prcs_out_dir* - Path to server output directory where all outputs are stored. The path is taken from *outputpath* parameter of section *server* in the configuration file.
- *dckr_inp_dir* - Path to input data directory of WPS instance running inside Docker container. It is taken from *dckr_inp_dir* of *processing* section.
- *dckr_out_dir* - Path to output directory of WPS instance running inside the container. It is taken from *dckr_out_dir* of *processing* section.

The method returns an instance of *Container* class from *docker* module. The container is created by *self.client.containers.create()* method.

The method takes optional parameter *ports*. It is a dictionary that defines ports to bind inside the container. The keys of the dictionary are the ports to bind inside the container (port 5000 inside *Container 1* and *Container 2* at Fig. 31). The values of the dictionary are the corresponding ports to open on the host (port 5050 for *Job1*, port 5051 for *Job2* at Fig. 31).

Another optional parameter is *volumes*. It is a dictionary to configure volumes mounted inside the container. The key is the host path and the value is a dictionary with the keys: *bind* - the path to mount the volume inside the container, and *mode* - either *rw* to mount the volume read/write, or *ro* to mount it read-only.

Listing 10: *create()* method

```
self.client.containers.create(cntnr_img, detach=True,
ports={"5000/tcp": self.port},
volumes={prcs_out_dir: {'bind': dckr_out_dir, 'mode': 'rw'},
         prcs_inp_dir: {'bind': dckr_inp_dir, 'mode': 'ro'}})
```

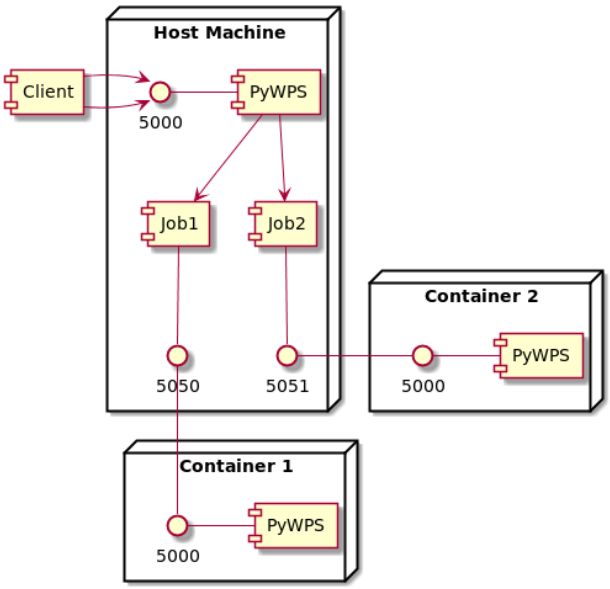



Figure 31: Ports assignment schema, source: author

Every container created with defined parameters *volumes* and *ports* will have output directory on the host mounted into the container output directory as well as the process workdir at host machine mounted into container directory with data. Therefore, all inputs downloaded to process workdir will be available for the container and all outputs produced after process execution will be stored at host machine output directory. Displayed in Fig. 32.

## 9.2 *Container.start()* method

When a container is created the *start()* method is called and the container is started. In the time of finishing this thesis the method looks like at Lst.11. In the current state is used the method *_dirty_clean()*. It assures that the container is removed after the successful execution, temporary workdir is cleaned and it also updates the process status in the database. Unfortunately, it causes that the process runs synchronously. To solve this problem is one of the future goals. The schema of *Container.start()* method is in Fig. 33

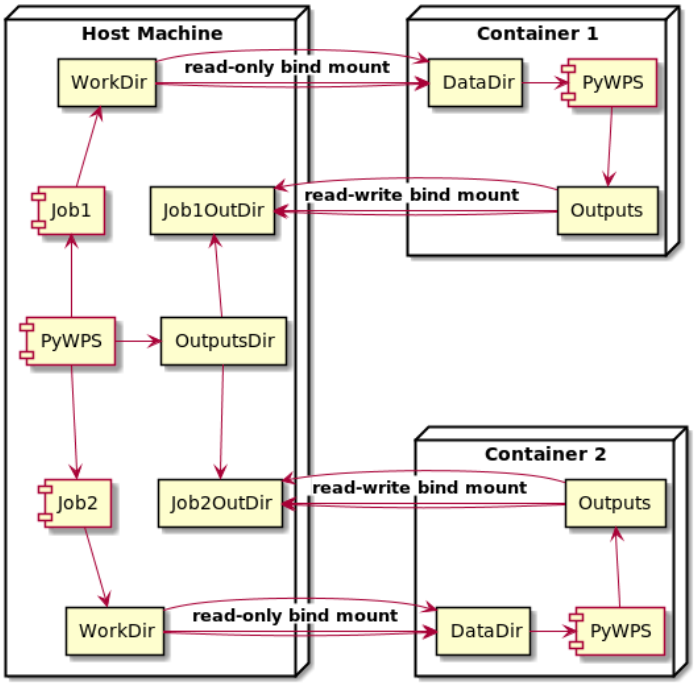



Figure 32: Schema of mounting directories, source: author

Listing 11: *Container.create()* method

```
def start(self):
    self.cntnr.start()
    time.sleep(0.5)
    self._execute()
    self._parse_status()
    self._dirty_clean()
```

### 9.2.1 *docker.container.start()*

*start()* method of *Container* class from Python module *docker*. The method is similar to *docker start* command. It starts the Docker container. Then the method *time.sleep()* is called to wait half a second after which the container is ready to use.

### 9.2.2 *Container._execute()*

*_execute()* method handles forwarding execution from server to container. For sending request to container *OWSLib* library (Sec. 6.2) is used.

**OWSLib** is a Python package for client programming with OGC web services interface standards. However before it was possible to use this package it was necessary to fix a bug in the *wps* module. The bug caused outputs in the Execute response, which were referenced as web-accesible resource, not to be parsed because of wrong handling with *xlink* namespace. The bug-fix diff file is available in App. F.

Listing 12: *Container._execute()* method

```
def _execute(self):
    url_execute = "http://localhost:{}/wps".format(self.port)
    inputs = get_inputs(self.job.wps_request.inputs)
    output = get_output(self.job.wps_request.outputs)
    wps = WPS(url=url_execute, skip_caps=True)
    self.execution = wps.execute(self.job.wps_request.identifier,
                                 inputs=inputs, output=output)
```

The method calls *get_inputs()* that returns all inputs transformed into a list of tuples in form *(input_name, input_value)*. In case of ComplexData input, the input value is replaced with path to file. It is necessary to transform the inputs into the list of tuples because it is the required form for *WebProcessingService.execute()* method. Example for demo process *buffer* at Lst. 14.

Listing 13: *get_inputs* return value

```
the_inputs = [('poly_in', 'file:///pywps-flask/data/point.gml'),
              ('buffer', '1.0')]
the_outputs = [('buff_out', 'true')]
```

Then the method calls *get_outputs()* that returns list of tuples in form *(output_name, asReference_attr_value)*. It is necessary to transform the outputs into the list of tuples because it is the required form for *WebProcessingService.execute()* method.

*WebProcessingService* object from OWSLib package is responsible for sending request to container. Its constructor takes URL of the WPS server running inside container. Container URL varies depending on the port assigned to the container. Then the *WPSExecution* object is assigned to the *Container* instance. The

WPSExecution object is returned from *WebProcessingService.execute()* method that takes as inputs process identifier, list of inputs from *get_ inputs()* and outputs from *get_ outputs()*.

### 9.2.3 *Container._ parse_ status()*

The method takes path to status location from *WPSExecution.statusLocation* and copies it to *Job.process.status_ url*. Then the *WPSReponse* object is updated by *Job.wps_ response.update_ status()* with *WPSExecution.statusMessage*. It means the *WPSResponse* object at host machine WPS server adopts *statusMessage* and path to *statusLocation* from *WPSExecution* object that handles the process execution inside the container. The process execution inside the container updates its status into the file that is located in container output directory. This directory is shared with WPS server at host machine so it is available even for the client.

### 9.2.4 *Container._ dirty_ clean()*

The method cares about stopping and removing Docker container, removing job workdir and original status XML. This method prevents from accumulation of running Docker containers and temporary files in workdir directory. On the other hand there is missing functionality for the process managment in database. In the current state when using Docker, the processes on the server are not ended even though the result is already returned from the container. These pseudo-running processes accumulate on the server and some other processes can be rejected because the limit of maximal running processes is reached. This must be solved in the future.

Listing 14: *get_ inputs* return value

```
def _dirty_clean(self):
    time.sleep(1)
    self.cntnr.stop()
    self.cntnr.remove()
    self.job.process.clean()
    os.remove(self.job.process.status_location)
```

*time.sleep(1)* is called to wait one second so the running process execution inside the container can be finished. The parameter 1 second is hardcoded and serves just now when the development is not done. *stop()* and *remove()* methods of class *Container* from *docker* module are similar to docker commands *docker stop container_id* and *docker rm container_id*.

*Job.process.clean()* remove the job workdir so the temporary files do not cumulate at the server. *os.remove()* deletes the original status XML since the status XML from the container was sent back to the client.

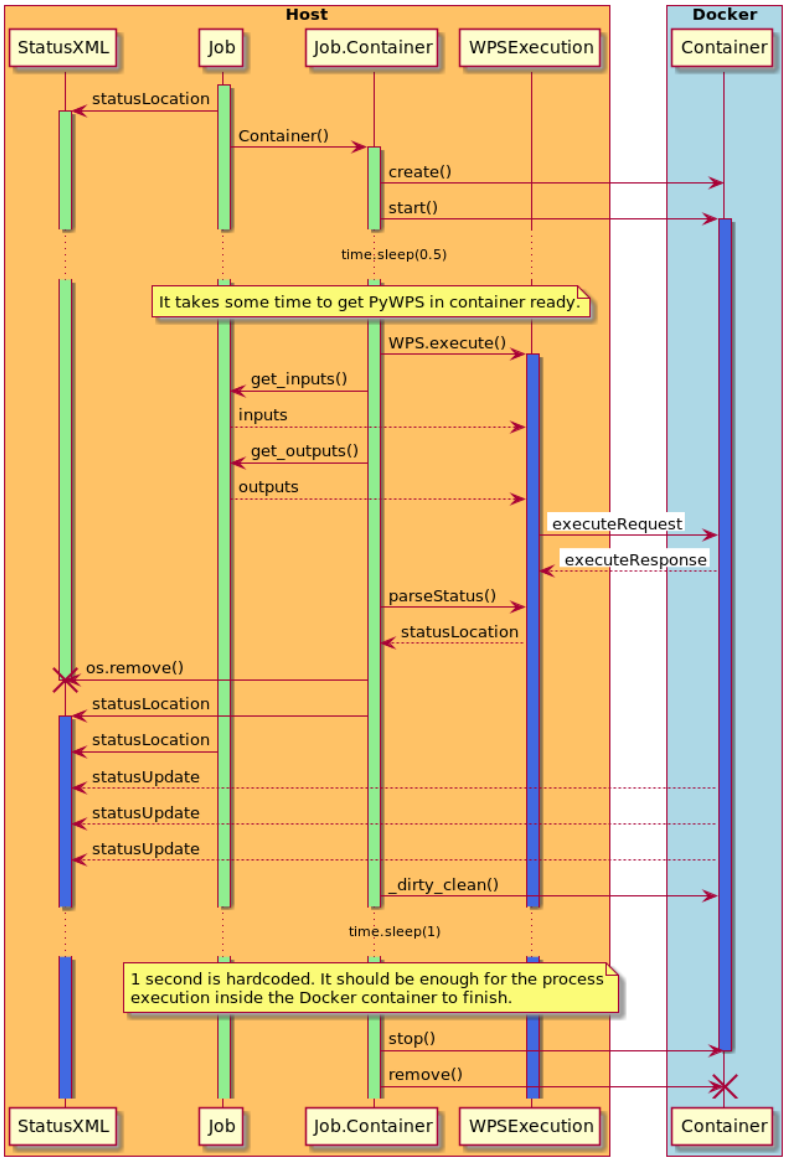



Figure 33: Schema of *WPSExecution*, source: author

# Conclusion

The goal of the thesis was to find and implement a solution for process isolation in *PyWPS*. This functionality was demanded by PyWPS developers who would appreciate the possibility to isolate each process execution. With every process fully isolated, a higher level of security is ensured. Moreover, without the isolation the processes have access to a file-system of a hosting OS.

But there are other reasons considered. One of them is a performance. Non-isolated processes share the resources of a host machine. In case that a client requests an execution of a process that is poorly designed, its execution can consume a lot of resources and thus it may slow down other process executions running in parallel. In the worst-case scenario a process execution can bring down the server.

In the first part, the thesis sets a theoretical background and the *WPS* standard is explained. Various projects which implement the standard are mentioned. Second part is dedicated to *PyWPS* as a Python implementation of the WPS. There is described the currentstate of the project followed by a research.

The reseach covers various solutions for the process isolation. The functionality for the isolation was the main criterion, however beside that the selected solution should provide some mechanisms to control the execution of the process. These mechanisms will be necessary for implementation of the WPS 2.0.0 standard.

The *Docker* Container Extension has been on the developers wishlist for a long time. The container encapsulates the process execution and also offers methods to start, pause, stop or kill the container and thus the execution. Moreover using Docker opens possibilities of Web Processing Service in a cloud computing infrastructure.

The architecture is based on *pywps-demo* project. It offers a demo server instance of PyWPS. When a process execution is requested, a server creates a Docker container with the demo server instance running inside. The request is forwarded into the container. The process execution runs inside the container but the container output directory is mounted into the server file-system so the results are available on the server.

At the time of submitting, the implementation is not finished. It provides working solution but there are several issues to be considered. First of all, the solution does not provide an asynchronous execution. A client receives an execute response that the process was accepted, after the execution is already finished.

Another problem is a management of containers. In the current state, a container is stopped and removed after a successful process execution, however it is achieved with the *_dirty_clean()* method which is inappropriate for production environment.

Another problem is cumulation of processes on a server. These pseudo-running processes remain on the server and block other processes even though a client has already received the results. There is missing connection between a container and a server when a process execution inside the container is done.

All these issues prevents from integration the Docker container extension into the official PyWPS repository. Nevertheless, I would like to continue on the development and solve all problems so I can make a pull request at least into PyWPS develop branch. During the implementation I have contributed into *pywps*, *pywps-demo* and *OWSLib* projects. Pull requests into *pywps* and *pywps-demo* wait for the PyWPS developers feedback and problems solution mentioned above. The pull request[15] into OWSLib is already waiting for approval. All diff files, as well the text of the thesis and its source code, are available at GitHub repository of this thesis.[16]

[15] `https://github.com/geopython/OWSLib/pull/410`

[16] `https://github.com/ctu-geoforall-lab-projects/dp-laza-2018/`

# List of abbreviation

| | |
|---|---|
| **API** | Application Programming Interface |
| **CGAL** | Computational Geometry Algorithms Library |
| **GDAL** | Geospatial Data Abstraction Library |
| **GIS** | Geographic Information System |
| **HPC** | High Performance Compute |
| **KVP** | Key Value Pair |
| **MIME** | Multipurpose Internet Mail Extension |
| **OGC** | Open Geospatial Consortium |
| **PID** | Process identifier |
| **SOAP** | Simple Object Access Protocol |
| **URL** | Uniform Resource Locator |
| **VM** | Virtual Machine |
| **VMM** | Virtual Machine Monitor |
| **WPS** | Web Processing Service |
| **WMS** | Web Map Service |
| **WFS** | Web Feature Service |
| **WCS** | Web Coverage Service |
| **XML** | eXtensible Markup Language |

# References

[1] Mark Reichardt *OGC Newsletter - October 2004, OGC document number 04-043* [online]. URL: <http://www.opengeospatial.org/pressroom/newsletters/200410>

[2] Sam Bacharach *OGC announces Web Processing Services Interoperability Experiment* [online]. URL: <http://www.opengeospatial.org/pressroom/pressreleases/414>

[3] Open Geospatial Consortium Inc. *OpenGIS ® Web Processing Service, OGC document number 05-007r4, ver. 0.4.0* [online]. URL: <https://portal.opengeospatial.org/files/?artifact_id=13149&version=1&format=doc>

[4] Open Geospatil Consorcium *OGC websites* [online]. URL: <http://www.opengeospatial.org/>

[5] Open Geospatil Consorcium *OGC History (detailed)* [online]. URL: <http://www.opengeospatial.org/ogc/historylong>

[6] http://www.opengeospatial.org/pressroom/newsletters/200410

[7] Open Geospatial Consortium *OGC® WPS 2.0 Interface Standard Corrigendum 1, OGC document number 06-121r3* [online]. URL: <https://portal.opengeospatial.org/files/?artifact_id=13149&version=1&format=doc>

[8] Open Geospatial Consortium Inc. *OGC Web Services Common Specification, OGC document number 14-065* [online]. URL: <https://portal.opengeospatial.org/files/?artifact_id=20040>

[9] deegree devs *deegree documentation* [online]. URL: <http://download.deegree.org/documentation/3.3.20/html/>

[10] GeoServer devs *GeoServer documentation* [online]. URL: <http://docs.geoserver.org/>

[11] ZOO-Project devs *ZOO-Project documentation* [online]. URL: <http://zoo-project.org/docs>

[12] Esri *Tuning and configuring services* [online]. URL: <http://server.arcgis.com/en/server/latest/publish-services/linux/tuning-and-configuring-services.htm>

[13] Docker *Docker documentation* [online]. URL: <https://docs.docker.com/>

[14] Jáchym Čepický, Luís Moreira de Sousa *New implementation of OGC Web Processing Service in Python programming language.* [online]. URL: <https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLI-B7/927/2016/isprs-archives-XLI-B7-927-2016.pdf>

[15] Jorge de Jesus, Luca Casagrande, Jáchym Čepický *PyWPS a tutorial for beginners and developers* [online]. URL: <https://www.slideshare.net/JorgeMendesdeJesus/pywps-a-tutorial-for-beginners-and-developers>

[16] PyWPS developers *PyWPS History* [online]. URL: <http://pywps.org/history/>

[17] PyWPS developers *PyWPS documentation* [online]. URL: <http://pywps.readthedocs.io/>

[18] Victor Stinner *The pysandbox project is broken* [online]. URL: <https://lwn.net/Articles/574323/>

[19] Victor Stinner *Linkage of OGC WPS 2.0 to the e-Government Standard Framework in Korea: An Implementation Case for Geo-Spatial Image Processing* [online]. URL: <http://www.mdpi.com/2220-9964/6/1/25/pdf>

# Part IV

# Appendix

# A Execute request example

Listing 15: Execute request example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<wps:Execute service="WPS" version="1.0.0" xmlns:wps="http://www.
    opengis.net/wps/1.0.0" xmlns:ows="http://www.opengis.net/ows
    /1.1" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation
    ="http://www.opengis.net/wps/1.0.0 ../wpsExecute_request.xsd">
 <ows:Identifier>buffer</ows:Identifier>
 <wps:DataInputs>
  <wps:Input>
   <ows:Identifier>poly_in</ows:Identifier>
   <wps:Reference xlink:href="http://localhost:5000/static/data/
       point.gml" />
  </wps:Input>
  <wps:Input>
   <ows:Identifier>buffer</ows:Identifier>
   <wps:Data>
    <wps:LiteralData>1</wps:LiteralData>
   </wps:Data>
  </wps:Input>
 </wps:DataInputs>
 <wps:ResponseForm>
  <wps:ResponseDocument status="true" storeExecuteRsponse="true">
   <wps:Output asReference="true">
    <ows:Identifier>buff_out</ows:Identifier>
   </wps:Output>
  </wps:ResponseDocument>
 </wps:ResponseForm>
</wps:Execute>
```

# B Execute response example (async mode)

Listing 16: Execute response example (async mode)

```
<!-- PyWPS 4.0.0 -->
<wps:ExecuteResponse xmlns:gml="http://www.opengis.net/gml"
   xmlns:ows="http://www.opengis.net/ows/1.1"
   xmlns:wps="http://www.opengis.net/wps/1.0.0"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
   http://schemas.opengis.net/wps/1.0.0/wpsExecute_response.xsd"
   service="WPS" version="1.0.0" xml:lang="en-US"
   serviceInstance="http://localhost:5000/wps?service=WPS&request=
   GetCapabilities"
   statusLocation="http://localhost:5000/outputs/ce57acbe-f1f3-11
   e7-ad2a-0242ac110003.xml">
  <wps:Process wps:processVersion="0.1">
   <ows:Identifier>buffer</ows:Identifier>
   <ows:Title>GDAL Buffer process</ows:Title>
   <ows:Abstract>
    The process returns buffers around the input features,
    using the GDAL library
   </ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2018-01-05T09:38:41Z">
   <wps:ProcessAccepted>
    PyWPS Process buffer accepted
   </wps:ProcessAccepted>
  </wps:Status>
</wps:ExecuteResponse
```

# C Status XML example with referenced output

Listing 17: Status XML example

```
<wps:ExecuteResponse
   xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
   http://schemas.opengis.net/wps/1.0.0/wpsExecute_response.xsd"
   service="WPS" version="1.0.0" xml:lang="en-US"
   serviceInstance="http://localhost:5000/wps?service=WPS&request=
   GetCapabilities"
   statusLocation="http://localhost:5000/outputs/ce57acbe-f1f3-11
   e7-ad2a-0242ac110003.xml">
  <wps:Process wps:processVersion="0.1">
   <ows:Identifier>buffer</ows:Identifier>
   <ows:Title>GDAL Buffer process</ows:Title>
   <ows:Abstract>
    The process returns buffers around the input features,
    using the GDAL library
   </ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2018-01-05T08:38:30Z">
   <wps:ProcessSucceeded>
    PyWPS Process GDAL Buffer process finished
   </wps:ProcessSucceeded>
  </wps:Status>
  <wps:ProcessOutputs>
   <wps:Output>
    <ows:Identifier>buff_out</ows:Identifier>
    <ows:Title>Buffered file</ows:Title>
    <wps:Reference xlink:href="http://localhost:5000/outputs/
        ce57acbe-f1f3-11e7-ad2a-0242ac110003/point_buffer_42rkmvt1.
        gml" mimeType="application/gml+xml"/>
   </wps:Output>
  </wps:ProcessOutputs>
</wps:ExecuteResponse>
```

# D Status XML example with inline output

Listing 18: Status XML example

```
<wps:ExecuteResponse
   xsi:schemaLocation="http://www.opengis.net/wps/1.0.0
   http://schemas.opengis.net/wps/1.0.0/wpsExecute_response.xsd"
   service="WPS" version="1.0.0" xml:lang="en-US"
   serviceInstance="http://localhost:5000/wps?service=WPS&request=
   GetCapabilities"
   statusLocation="http://localhost:5000/outputs/1cd3e506-f1f7-11
   e7-8546-0242ac110003.xml">
  <wps:Process wps:processVersion="0.1">
   <ows:Identifier>buffer</ows:Identifier>
   <ows:Title>GDAL Buffer process</ows:Title>
   <ows:Abstract>
    The process returns buffers around the input features,
    using the GDAL library
   </ows:Abstract>
  </wps:Process>
  <wps:Status creationTime="2018-01-05T09:02:10Z">
   <wps:ProcessSucceeded>
    PyWPS Process GDAL Buffer process finished
   </wps:ProcessSucceeded>
  </wps:Status>
  <wps:ProcessOutputs>
   <wps:Output>
    <ows:Identifier>buff_out</ows:Identifier>
    <ows:Title>Buffered file</ows:Title>
    <wps:Data>
     <wps:ComplexData mimeType="application/gml+xml">
      <ogr:FeatureCollection xmlns:ogr="http://ogr.maptools.org/"
         xsi:schemaLocation="http://schemas.opengis.net/gml/2.1.2/
         feature.xsd">
       <gml:boundedBy>
        <gml:Box>
```

```
          <gml:coord>
            <gml:X>-0.9514645979959721</gml:X>
            <gml:Y>-0.986306232731747</gml:Y>
          </gml:coord>
          <gml:coord>
            <gml:X>1.048535402004028</gml:X>
            <gml:Y>1.013693767268253</gml:Y>
          </gml:coord>
        </gml:Box>
      </gml:boundedBy>
      <gml:featureMember>
      <ogr:point_buffer fid="point_buffer.0">
        <ogr:geometryProperty>
          <gml:Polygon>
            <gml:outerBoundaryIs>
              <gml:LinearRing>
                <gml:coordinates>1.04853540200403,0.013693767268253
                1.0471649367586,-0.038642188974691 0.857552396378976,
                -0.57409148502422 0.825681363460999,-0.615626623781584
                0.791680227481423,-0.655436839090605 0.75564218319056,
                -0.852331636516187 -0.496103633010996,-0.8249768006773
                -0.539249850288442,-0.795323227106697 -0.580784989006,
                -0.763452194188721 -0.620595204354827,-0.7294510582044
                </gml:LinearRing>
              </gml:outerBoundaryIs>
            </gml:Polygon>
          </ogr:geometryProperty>
        </ogr:point_buffer>
      </gml:featureMember>
    </ogr:FeatureCollection>
  </wps:ComplexData>
 </wps:Data>
</wps:Output>
</wps:ProcessOutputs>
</wps:ExecuteResponse>
```

# E Dockerfile

Listing 19: Dockerfile example

```
FROM alpine:latest
MAINTAINER Jorge S. Mendes de Jesus <jorge.dejesus@geocat.net>

ENV GDAL_VERSION 2.2.0
ENV XERCES_VERSION 3.2.0

RUN apk add --no-cache \
        git \
        gcc \
        bash \
        openssh \
        musl-dev \
        python3 \
        python3-dev \
        libxml2-dev \
        libxslt-dev \
        linux-headers \
        expat \
        expat-dev


RUN apk --update --no-cache add g++ libstdc++ make swig

# Xerces
RUN wget http://www.apache.org/dist/xerces/c/3/sources/xerces-c-${
    XERCES_VERSION}.tar.gz -O /tmp/xerces-c-${XERCES_VERSION}.tar
    .gz && \
     tar xvf /tmp/xerces-c-${XERCES_VERSION}.tar.gz -C /tmp && \
     cd /tmp/xerces-c-${XERCES_VERSION} && \
     ./configure --prefix=/opt/xerces && \
     make -j 4 && \
     make install
```

```
# Geos
RUN apk add --no-cache \
    --repository http://dl-cdn.alpinelinux.org/alpine/edge/
        testing \
    geos \
    geos-dev

# Install GDAL
RUN wget http://download.osgeo.org/gdal/${GDAL_VERSION}/gdal-${
    GDAL_VERSION}.tar.gz -O /tmp/gdal.tar.gz && \
        tar xzf /tmp/gdal.tar.gz -C /tmp && \
        cd /tmp/gdal-${GDAL_VERSION} && \
        CFLAGS="-g -Wall" LDFLAGS="-s" ./configure --with-expat=
            yes --with-xerces=/opt/xerces --with-geos=yes \
        && make -j 4 && make install

RUN cd /tmp/gdal-${GDAL_VERSION}/swig/python \
        && python3 setup.py build \
        && python3 setup.py install

RUN git clone https://github.com/lazaa32/pywps-flask.git

WORKDIR /pywps-flask
RUN pip3 install -r requirements.txt


EXPOSE 5000
ENTRYPOINT ["/usr/bin/python3", "demo.py","-a"]

#docker build -t pywps-flask .
#docker run -p 5000:5000 pywps-flask
#http://localhost:5000/wps?request=GetCapabilities&service=WPS
#http://localhost:5000/wps?request=DescribeProcess&service=WPS&
    identifier=all&version=1.0.0
```

# F OWSLib diff file

Listing 20: OWSLib diff file

```
diff --git a/owslib/wps.py b/owslib/wps.py
index c16e288..ce86f93 100644
--- a/owslib/wps.py
+++ b/owslib/wps.py
@@ -1117,13 +1117,15 @@ class Output(InputOutput):

    # extract wps namespace from outputElement itself
    wpsns = getNamespace(outputElement)
+   # extract xlink namespace
+   xlins =outputElement.nsmap['xlink']

    # <ns:Reference encoding="UTF-8" mimeType="text/csv"
    # href="http://cida.usgs.gov/climate/gdp/process/
        RetrieveResultServlet?id=1318528582026OUTPUT.601bb3d0-547f
        -4eab-8642-7c7d2834459e"
    # />
    referenceElement = outputElement.find(nspath('Reference', ns=
        wpsns))
    if referenceElement is not None:
-       self.reference = referenceElement.get('href')
+       self.reference = referenceElement.get('{{{}}}href'.format(
    xlins))
        self.mimeType = referenceElement.get('mimeType')

    # <LiteralOutput>
```

# G PyWPS-demo diff file (shortened)

Listing 21: pywps-demo diff file

```
diff --git a/pywps.cfg b/pywps.cfg
index a1ed125..f6e3981 100644
--- a/pywps.cfg
+++ b/pywps.cfg
@@ -28,14 +28,25 @@ maxrequestsize=3mb
 url=http://localhost:5000/wps
 outputurl=http://localhost:5000/outputs/
 outputpath=outputs
-workdir=/tmp
+workdir=workdir
+wd_inp_subdir=inputs
+wd_out_subdir=outputs
 maxprocesses=10
-parallelprocesses=2
+parallelprocesses=6
+
+[processing]
+mode=multiprocessing
+port_min=5050
+port_max=5070
+docker_img=pywps_container
+dckr_inp_dir=/pywps-flask/data
+dckr_out_dir=/pywps-flask/outputs

 [logging]
 level=INFO
 file=logs/pywps.log
 database=sqlite:///logs/pywps-logs.sqlite3
+format=%(asctime)s] [%(levelname)s] file=%(pathname)s line=%(
    lineno)s module=%(module)s function=%(funcName)s %(message)s
```

# H PyWPS diff file (shortened)

Listing 22: pywps diff file

```
diff --git a/pywps/exceptions.py b/pywps/exceptions.py
index 8483911..e0e1c57 100644
--- a/pywps/exceptions.py
+++ b/pywps/exceptions.py
@@ -150,3 +150,10 @@ class SchedulerNotAvailable(NoApplicableCode
    ):
      """Job scheduler not available exception implementation
      """
      code = 400
+
+
+class NoAvailablePortException(NoApplicableCode):
+    """
+    No port available for new docker.
+    """
+    code = 400
diff --git a/pywps/processing/__init__.py b/pywps/processing/
    __init__.py
index 03bf0af..63816e1 100644
--- a/pywps/processing/__init__.py
+++ b/pywps/processing/__init__.py
@@ -7,6 +7,7 @@
 import pywps.configuration as config
 from pywps.processing.basic import MultiProcessing
 from pywps.processing.scheduler import Scheduler
+from pywps.processing.container import Container
 # api only
 from pywps.processing.basic import Processing  # noqa: F401
 from pywps.processing.job import Job  # noqa: F401
@@ -16,6 +17,7 @@ LOGGER = logging.getLogger("PYWPS")

 MULTIPROCESSING = 'multiprocessing'
```

```
 SCHEDULER = 'scheduler'
+DOCKER = 'docker'
 DEFAULT = MULTIPROCESSING


@@ -30,6 +32,8 @@ def Process(process, wps_request, wps_response)
    :
      LOGGER.info("Processing mode: %s", mode)
      if mode == SCHEDULER:
          process = Scheduler(process, wps_request, wps_response)
+     elif mode == DOCKER:
+         process = Container(process, wps_request, wps_response)
      else:
          process = MultiProcessing(process, wps_request,
              wps_response)
      return process
diff --git a/pywps/processing/container.py b/pywps/processing/
    container.py
new file mode 100644
index 0000000..8f5151e
--- /dev/null
+++ b/pywps/processing/container.py
@@ -0,0 +1,155 @@
+class ClientError:
+    pass
+
+class Container(Processing):
+    def __init__(self, process, wps_request, wps_response):
+    def _create(self):
+    def _assign_port(self):
+    def start(self):
+    def stop(self):
+    def cancel(self):
+    def pause(self):
+    def unpause(self):
+    def _execute(self):
```

```
+    def _parse_outputs(self):
+    def _parse_status(self):
+    def _dirty_clean(self):
+def get_inputs(job_inputs):
+def get_output(job_output):

diff --git a/pywps/response/execute.py b/pywps/response/execute.
    py
index f78cfb0..e994de3 100644
--- a/pywps/response/execute.py
+++ b/pywps/response/execute.py
@@ -15,7 +15,7 @@ from pywps import WPS, OWS
-import pywps.dblog
+from pywps.dblog import update_response

@@ -38,6 +38,33 @@ class ExecuteResponse(WPSResponse):
          self.process = kwargs["process"]
          self.outputs = {o.identifier: o for o in self.process.
             outputs}

+    def update_status(self, message=None, status_percentage=None
    , status=None,
+                      clean=True):
```

# I Docker extension documentation (shortened)

Listing 23: Docker extension documentation

```
Docker Container Extension
==========================

To isolate each process execution it is possible to enable docker
    extension.
.. note:: The PyWPS process implementations are not changed by
    using the
  scheduler extension.
First of all install Docker from `website <https://docs.docker.
    com/engine/installation/linux/docker-ce/ubuntu/>`_.
Clone ``pywps-demo``::
  $ git clone https://github.com/lazaa32/pywps-flask.git
Install demo requirements from ``requirement.txt``. It will
    download all required packages including ``pywps`` core
    package::
  $ cd pywps-flask
  $ pip install -r requirements.txt

``pywps`` package was downloaded to ``src`` directory. Let's set
    the ``PYTHONPATH`` so ``pywps-demo`` knows where to find::
  $ EXPORT PYTHONPATH=$PYTHONPATH:$PWD/src/pywps-develop
If everything went OK, it should be now possible to run::
  $ python3 demo.py
However the demo still runs without Docker extension. First of
    all it is necessary to build an image from Dockerfile.
From the image all containers will be created::
  $ cd docker/isolation
  $ docker build -t container .
.. note:: The **-t** flag sets a name and optionally a tag in the
    **name:tag** format. The name of the image
    will be one of the parameter value in configuration file.
.. warning:: The image build can take up to several tens of
    minutes since some manual installation run on the
```

```
    background.
You can check the image was built by::
  $ docker images
To activate this extension you need to edit the ``pywps.cfg``
    configuration file and make the following changes::
  [processing]
  mode=docker
  port_min=5050
  port_max=5070
  docker_img=container
  dckr_inp_dir=/pywps-flask/data
  dckr_out_dir=/pywps-flask/outputs
``mode`` must be set to ``docker``. ``port_min`` and ``port_max``
    define the range of ports which can be
assigned to containers. ``docker_img`` must match to name of the
    image given by -t flag during the image build.
The docker extension is now enabled and every asynchronous
    request will be executed separately in a Docker
container.
```

# J List of tables and figures

## List of Tables



## List of Figures

# K ZIP file content

.

| | |
|---|---|
| **src** | |
| **diffs** | |
| owslib.diff | owslib diff file |
| pywps.diff | pywps diff file |
| pywps-demo.diff | pywps-demo diff file |
| **text** | |
| **LaTeX** | LaTeX source code |
| adam-laza-bp-2015.pdf | text of the thesis |
| **zadani** | |
| zadanidp.pdf | assignment of the thesis |