INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

NGS4Cloud: Cloud-based NGS Data Processing

Alexandre Almeida

João Forja

Projeto e Seminário

Licenciatura em Engenharia Informática e Computadores

Beta Final Report

Supervisors: Cátia Vaz

José Simão

Alexandre P. Francisco, IST

June 2016

Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Projecto e Seminário

NGS4Cloud: Cloud-based NGS Data Processing

Authors
Alexandre Almeida, nº 40640
João Forja, nº 41087

Supervisors
Cátia Vaz, ISEL
José Simão, ISEL
Alexandre P. Francisco, IST

Table of contents

List of Figures

List of Tables

1

Introduction

Next-Generation Sequencing (NGS) technologies are greatly increasing the amount of genomic computer data, revolutionizing the biosciences field and leading to the development of more complex NGS Data Analysis techniques (Shuster, 2008). These techniques, known as pipelines or workflows, consist of running and refining a series of intertwined computational analysis and visualization tasks on large amounts of data. These pipelines involve the use of multiple software tools and data resources in a staged fashion, with the output of one tool being passed as input to the next one.

Due to the complexity of configuring and parametrizing pipelines, the use of NGS Data Analysis techniques is not an easy task for a user without IT knowledge. Moreover, knowing input data can be as much as terabytes and petabytes, pipelines execution require, in general, a great amount of computational resources.

NGSPipes framework is devised to solve the first issue, allowing to easily design and use pipelines, without users need to configure, install and manage tools, servers and complex workflow management systems (Dantas and Fleitas, 2015). In the context of NGSPipes, a pipeline is a series of commands of NGS data-processing tools chained by input and output. NGSPipes offers a DSL (domain-specific language), to describe pipelines. The DSL's syntax provides primitives to specify the sequence through which each tool command is executed, to specify arguments, and to chain commands' inputs and outputs. To know each tools' properties and commands, NGSPipes uses tools' meta-data which is saved in a user-provided repository.

However, at the present, NGSPipes operates only in a single machine, which will most likely lack the necessary resources to execute the pipeline in a reasonable period of time, if at all. Even if users do have access to a powerful machine, there are still problems that need to be solved in order to execute a pipeline as efficiently as possible: NGSPipes does not support parallel execution of pipeline tasks, nor

splitting input data of tools's commands into multiple fragments to be processed in parallel.

Cloud technologies are sought as a solution to solve the lack of resources of an average computer, a feature that will substantially improve NGSPipes solution, providing users with big clusters of powerful machines to run pipelines more efficiently (Armbrust et al., 2010). The solution we developed, NGS4Cloud, integrates NGSPipes with the capability of running pipelines in a remote cluster. It also extends it with parallel execution of pipeline tasks, in either different cores of the same machine or different machines, as well as data partitioning. On top of that, it also allows the multi-core execution of tools' commands that support it. NGS4Cloud tries to achieve these goals while standing to the usability standards of NGSPipes, which aim to keep pipelines easy to use by non-expert IT users.

To implement NGS4Cloud, we developed an execution engine that analyses the pipeline and builds a graph of tasks. This graph of tasks reflects the dependency among tasks and allows to infer what can be executed in parallel and what can only be executed in serial. From it, the engine deploys the tasks in a cluster of machines governed by the Mesos's batch job scheduling framework Chronos.

1.1 Outline

This work is is divided in 6 chapters.

In chapter 2 we will describe the domain of our problem in more depth and our approach to solve it.

In chapter 3 we will introduce the technologies that support NGS4Cloud.

In chapter 4 we will overview NGS4Cloud, namely its components and how the goals pointed out in chapter 2 will be reached with this framework.

In chapter 5 we will discuss implementation details about each of the components that compose NGS4Cloud, their functionalities and their interactions.

In chapter 6 we will talk about the plan delineated in the project proposal and how we're following it.

2

Problem Description

DNA sequencing is used in multiple fields like molecular biology, evolutionary biology, medicine and forensics. There has been a big increase on the amount of data produced by DNA sequencing since the appearance of Next-Generation Sequencing methods, creating the need to further develop techniques and programs to analyse the data. Due to that, a plurality of tools have appeared to process DNA sequences, or NGS data. These tools' commands are usually executed in structured data workflows and the output of one is the input of others, forming a pipeline. Summing up, in the context of NGS data processing, a pipeline is a series of intertwined computational analysis and visualization tasks on large amounts of data.

NGS data analysis techniques raise some challenges since, depending on the tools' commands and the amount of data processed, the execution of a pipeline may take days or even weeks; the design and parameterization of a pipeline and setup of tools are not easy tasks for users with little or no IT experience, like many biologists; and being able to reproduce a pipeline is rather valuable.

We think a good application should solve the before mentioned challenges by attending to the following objectives:

1. Make the design and parametrization of a pipeline user friendly and easy for non-IT experts;
2. Facilitate the reproducibility of a pipeline;
3. Run the pipelines efficiently, making use of a cluster of powerful machines and parallelization techniques.

NGSPipes is a framework to easily design and use pipelines, relying on state of the art cloud technologies to execute them without users need to configure, install and manage tools, servers and complex pipeline systems. NGSPipes fulfils the first two objectives, but fails to accomplish the third. NGSPipes does not fully explore the cloud services and remote execution to the maximum, nor the parallelization of

a pipeline and its tasks[1], as right now it only allows to execute pipelines on a single machine, running its tasks sequentially, one at a time.

In order to allow a user to abstract from the necessary resources to execute a pipeline and for NGSPipes to be able to validate a given description of a pipeline, it has a component called tool `meta-data` repository. The tool `meta-data` repository is a repository which contains information such as: the computational resources needed to execute a tool, the commands a tool allows, the arguments of each command, and the docker image (see 3.1) in which the tool is installed. Since the setup of a tool can be a hard task, NGSPipes uses docker images to setup and deploy tools.

The tools used to process NGS data are, in essence, batch jobs executors. They are usually developed by software engineers connected to the biology field. There is not a main entity when it comes to developing NGS data processing tools and no sort of standards for the tools are implemented, resulting in a wide variety of tools that operate and behave differently. Due to this variety there isn't much interoperability between them and it becomes difficult to find patterns. Each tool has different commands that receive different arguments and need different amounts of computational resources to execute. Therefore, NGSPipes lays the responsibility of describing each tool used in the pipeline to the users, by requiring them to specify each tool's information and save it in the tool meta-data repository. Notice that users that fill in tool meta-data information are usually tech-savvy, like bioinformaticians, since tool's description can be used independently by multiple users in different contexts.

To facilitate the design and parametrization of a pipeline, NGSPipes offers a DSL that allows users to produce files containing a pipeline description, that can be executed by NGSPipes engine in any machine that runs it. Together with inputs being easily changed, this makes pipelines reproducible. Notice that the users that produce `.pipes` files using the DSL may not have IT skills, unlike the ones producing the tools' descriptions.

2.1 NGS4Cloud Solution

We are creating NGS4Cloud in order to fulfil all the objectives listed above. To fulfil the first two objectives we are integrating NGSPipes and making use of its DSL and tool meta-data repository. To achieve the third objective we treat remote execution in a cluster of machines and task parallelization as separate concepts.

[1] In the context of a pipeline, a task is the execution of a tool's command.

To explore the remote execution of pipelines in a cluster of machines, we are using Mesos (see 3.2) which allows to easily manage clusters of machines assuring an efficient use of resources, provides fault tolerance and high availability, and has support for Docker (see 3.1), the technology used by NGSPipes to setup and deploy tools. Mesos also decouples NGS4Cloud from a specific cloud service. Since pipeline tasks are batch jobs, we are using the Chronos framework (see 3.3), a batch job scheduler framework that runs on top of Mesos and offers a RESTApi for scheduling jobs with dependencies. The cluster also has a distributed file system which serves as working directory for the execution of the pipeline, allowing all cluster machines to access the files being produced. This way we solve the problem of transferring the necessary files, to execute a tool, between machines. We have a software component, the Monitor (see 5.4), responsible for reading a JSON file containing a description of the tasks and its dependencies, and using Chronos to schedule jobs corresponding to the tasks.

NGS4Cloud system does not include a cluster and this must be previously set up. However, it does seek to use the cluster resources efficiently and offer an autonomous process for installing tools, uploading inputs and downloading the outputs. NGS4Cloud requires the cluster to be properly configured to run the pipeline (see 5.3). Concerns like guaranteeing the cluster has the necessary resources to execute the pipeline or giving priority of task execution to a machine which already has the necessary inputs to execute the task in memory, are not part of our project. Our only preoccupation regarding these aspects is that NGS4Cloud does not need modifications when working against different clusters, as long as they fulfil the requirements (see 5.3).

Regarding the parallelization, we cannot forget that pipelines must remain simple to design and use, therefore we try to create a high abstraction over the parallelization making this feature as transparent as possible to users, so it does not have a negative impact in usability by non-IT experts. We split parallelization in three levels: multi-core execution of tool commands that support it; parallel execution of independent tasks of a pipeline; and data partitioning in order to process the fragments in parallel executions of the same tool command.

The optimal partitioning of the input files for parallel execution is not solved by NGS4Cloud. This responsibility is entirely delegated to the users when they describe the pipeline using the DSL. If a user decides to split a file, then it will be partitioned as requested. Other ideas of parallelization like streaming data between

tools to allow tasks to process different parts of the stream in parallel are not taken into account, since most NGS data processing tools do not support it.

To support the multi-core execution of a command, we modified the tools' meta-data to indicate if a tool command supports multi-core execution and, if so, in how many cores; to support the parallelization of independent tool commands we automatically infer from the pipeline description dependencies based on the outputs and inputs of commands; to support the data partitioning we added a field to the tool meta-data that identifies whether a tool is a NGS processing tool or used to split, join, or list files.

Our solution also includes a software component, the analyser (see 5.1), responsible for interpreting a pipeline description and cross it with the tool meta-data repository, producing tasks which correspond to each command, and a DAG (directed acyclic graph) representing the dependencies between tasks. The vertices of the DAG are the tasks, while the edges represent the dependencies. Posteriorly, the DAG is converted into a JSON object which is saved on a file: the same JSON file that the Monitor reads. This file is called intermediate representation (see 4.2.5) whose benefits are explained on the chapter about the architecture.

3

Related Technologies

In this chapter we introduce the technologies that support NGS4Cloud.

3.1 Docker

Docker[1] is an open-source project which wraps and extends Linux containers technology to create a complete solution for the creation and distribution of containers. The Docker platform provides a vast number of commands to conveniently manipulate containers.

A *container* is an isolated, yet interactive, environment configured with all the dependencies necessary to execute an application. The use of containers brings advantages such as:

- Having little to no overhead compared to running an application natively, as it interacts directly with the host OS kernel and no layer exists between the application running and the OS;
- Providing high portability since the application runs in the environment provided by the container; bugs related to runtime environment configurations will almost certainly not occur;
- Running dozens of containers at the same time, thanks to their lightweight nature;
- Executing an application by downloading the container and running it, avoiding going through possible complex installations and setup.

To easily configure the virtual environment that the container hosts, Docker provides Docker images. *Images* are snapshots of all the necessary tools and files to execute an application. Containers can be started from images, the same way virtual machines run snapshots.

[1] https://www.docker.com/, last visited on 2016/04/26

To effortlessly distribute images, Docker provides *registries*. These are public or private stores where users may upload or download images. Docker provides a cloud-based registry service called DockerHub[2].

3.2 Mesos

Apache Mesos[3] is a distributed, high availability, fault tolerant system that allows for multitenancy with the support of containers technologies like Linux containers and Docker. Mesos architecture has three main entities: *masters*, *slaves*, and *frameworks*.

A Mesos cluster can have one or more masters, responsible for managing the resources provided by the slaves. Using Zookeeper[4], Mesos implements a leader election technique to ensure fault tolerance: if the leading master fails, other masters will be ready to replace it.

Slaves are the cluster machines where tasks will be executed. After launching a slave, it uses Zookeeper to register with the current leading master and advertises its available computational resources, such as CPU, memory and disk. A slave can be configured with the list of resources it shall advertise. If omitted, it will automatically offer all the available resources.

A framework is a Mesos application responsible for scheduling and executing user provided tasks in the cluster. It can be split into two components: the *scheduler* and *executors*. Using Zookeeper, the scheduler detects the leading master and proceeds to registering with it. The scheduler then starts receiving resource offers from the leading master. If it has pending tasks to be executed, prompted by users, the scheduler chooses a suitable resource offer and launches an executor on the respective slave to run the tasks. Executors are process containers launched in slaves and are responsible for running user submitted tasks. The built-in Mesos executors offer the possibility to launch tasks from shell scripts or docker containers.

3.3 Chronos

Chronos[5] is a Mesos framework specialized in scheduling batch jobs. Similarly to Mesos, it uses Zookeeper to ensure fault tolerance. It is possible to schedule command line jobs and docker jobs through a user interface or through a RESTApi. Jobs can

[2] https://hub.docker.com/, last visited on 2016/04/26
[3] http://mesos.apache.org/, last visited on 2016/04/26
[4] https://zookeeper.apache.org/, last visited on 2016/05/02
[5] https://mesos.github.io/chronos/, last visited on 2016/04/26

be scheduled to run on a specified date and in repeating intervals, with the option to specify the frequency at which the job is meant to rerun. Chronos also allows job dependency scheduling.

3.4 NGSPipes

In these subsections we explore the components of *NGSPipes* that are used in the NGS4Cloud solution. We explain how we modified or extended them to meet the new requirements in 4.2.1 and 4.2.2.

3.4.1 Tool's meta-data and Repository

Bearing in mind the diversified nature of the NGS data processing tools and the biologists need to use specific tools for different tasks, NGSPipes manages to treat tools like plug-ins. These plug-ins are known as *tool meta-data*. This is stored in a hierarchical directory system organised repository, known as *tool meta-data repository*. This repository may be local or remote. Using a remote repository allows us to access the data on the repository independently of the machine we are using, and also gives us a backup of the data. The repository has a file `Tools.json` with an array of the tools' names contained in the repository, and a folder per tool, named after the stored tool. Each folder contains the tool meta-data, which has to supply the following information:

- A descriptor object, kept in a JSON file named `Descriptor.json` which contains information about the tool, its commands, and its commands' arguments.
- A JSON file named `Configurators.json` which contains an array of configurators for the tool. A configurator is an object that holds the basic information to setup the tool.
- One file per configurator in the `Configurators` array. The file name corresponds to the configurator name plus the extension `.json`. In the case of Docker, the configurator contains the name of the docker image, and an array with the setup commands. The docker image name maps to an image stored in DockerHub registry, being easily distributed by downloading from any machine that needs to run the image.

Schemas for the `.json` files mentioned above can be found in the NGSPipes tool meta-data wiki[6].

[6] https://github.com/ngspipes/tools/wiki, last visited on 2016/06/11

New tools' information can be added to the tool meta-data repository and new tool images to DockerHub. By recurring to tool meta-data, biologists can use almost any tool as long it is added to the tool meta-data repository they use.

3.4.2 Domain-specific language

In this subsection we look into the *domain-specific language* used to describe pipelines. A `.pipes` file is a file whose content is a description of a pipeline. The description is done using the DSL. The DSL was created to offer users a way to describe and parametrize pipelines. It also promotes reproducibility since parameters are easily changed and descriptions can be stored and shared with a single file. The tool meta-data and the DSL, when combined together, provide programmers with a mechanism to validate the descriptions of the pipelines by matching the description parameter values with the tool meta-data information, regarding tool names, command names, argument names, and argument types.

The DSL has 5 primitives: *Pipeline*, *tool*, *command*, *argument* and *chain*.

Pipeline - Since a pipeline is composed by the execution of one or more tools, it must be defined the tools' repository, i.e., all the information necessary with respect to the available tools. To define this repository in the pipeline it is necessary to identify not only where it is stored, but also the type of storage (locally or remotely, like GitHub) to know how to process that information. Formally, the pipeline must follow the following grammar:

```
Pipeline: 'Pipeline' repositoryType repositoryLocation '{' (tool)+ '}';
```

where `(tool)+` represents that a pipeline is composed by the execution of one or more tools (notice that, as will be further explained, the tool execution may include the execution of one or more commands).

tool - Each tool is specified in the pipeline by its name, its configuration file name (without extension), and by the set of commands within the tool that will be executed within this pipeline. The tool primitive follows the grammar:

```
tool: 'tool' toolName configuratorFileName '{' (command)+ '}';
```

where `(command+)` represents the commands of a tool to be executed. One ore more commands can be specified here and may even be specified repeated commands with different arguments, and a tool and its commands can be specified more than once in a pipeline. A file with the name `configuratorFileName` must exist in the tool repository stated, when using the *Pipeline* primitive (parameter `repositoryLocation`), within the tool information, a configuration file named `configuratorFileName`, with JSON Format. This file must define a JSON object with the property build

set as `configuratorFileName`. With this information together with the repository information, the environment for executing the tool's commands is specified.

command - As mentioned before, there may exist a set of commands within the tool that should be executed within a pipeline. For executing each command, it is necessary to identify its name, which is unique in the tool context and to set the values for each required parameters (optional parameters may not be specified). We refer to the command parameters in NGS4Cloud language as arguments, since we only specify in the pipeline the parameters for which we have values to set. The command specification must follow the grammar:

```
command: 'command' commandName '{' (argument | chain)+ '}';
```

where `(argument | chain)+` represents that there may exist a list of arguments within this command as well as a list of chains. As mentioned before, a command can be executed more than once in a pipeline, and in that case it must be described the amount of times it is to be executed, always passing the desired arguments and chains.

argument - The argument definition has the following syntax:

```
argument: 'argument' argumentName argumentValue;
```

chain - The *chain* primitive allows to set an argument of a command with the produced output of other command. It links outputs to inputs. Sometimes the produced output is returned as files with names given internally by the command (identified in each tool-meta-data through property `outputs`). Alternatively the output files name may be given explicitly as an argument to the command. In both situations, it is common that other commands use these output files to continue processing the pipeline. The primitive *chain* has a simplified version, which can be used when the output is from the previous stated command in the pipeline specification. In this case, we only specify the name of the output file to chain with the given argument. Another version of the primitive allows the user to chain between different tools commands that are now followed. Here the user must state the tool name, the command name, the output and argument names. A last version of the primitive *chain* is when the name of the tool can be omitted, but it is necessary to specify the name of the command, of the argument and also the output. This applies to cases where the chain occurs between two commands of the same tool. Thus, the chain specification must follow the grammar:

```
chain: 'chain' argumentName ( ( toolName )? commandName)? outputName;
```

Examples of pipeline descriptions can be found in the NGSPipes DSL wiki[7].

[7] https://github.com/ngspipes/dsl/wiki, last visited on 11/06/2016

4

Architecture

In this chapter we describe each component of the system, how they interact, and which technologies supports them.

4.1 Overview

Figure 2.1 is a diagram that shows the system's main components and their interactions. There are five components:

1. Tool meta-data repository;
2. Domain-specific language;
3. Analyser;
4. Intermediate representation;
5. Monitor.

The interaction begins when a user provides a file with a pipeline's description, specified using the NGS4Cloud DSL, to the system. The analyser inspects the given file and, using the information stored in the tool meta-data repository, produces the instructions to execute the pipeline, along with the required computational resources for its execution. These instructions are then written to a file and locally stored, giving origin to an intermediate representation. Having the analyser concluded its job, the monitor is launched. It converts the intermediate representation into jobs' descriptions readable by Chronos and, using Chronos's REST API, schedules them for execution. Having launched the pipeline, the user can now query the system to know its current state. This results in a series of requests from the Monitor to Chronos. When the pipeline finishes the execution, the user can make a request to the monitor to download its outputs.

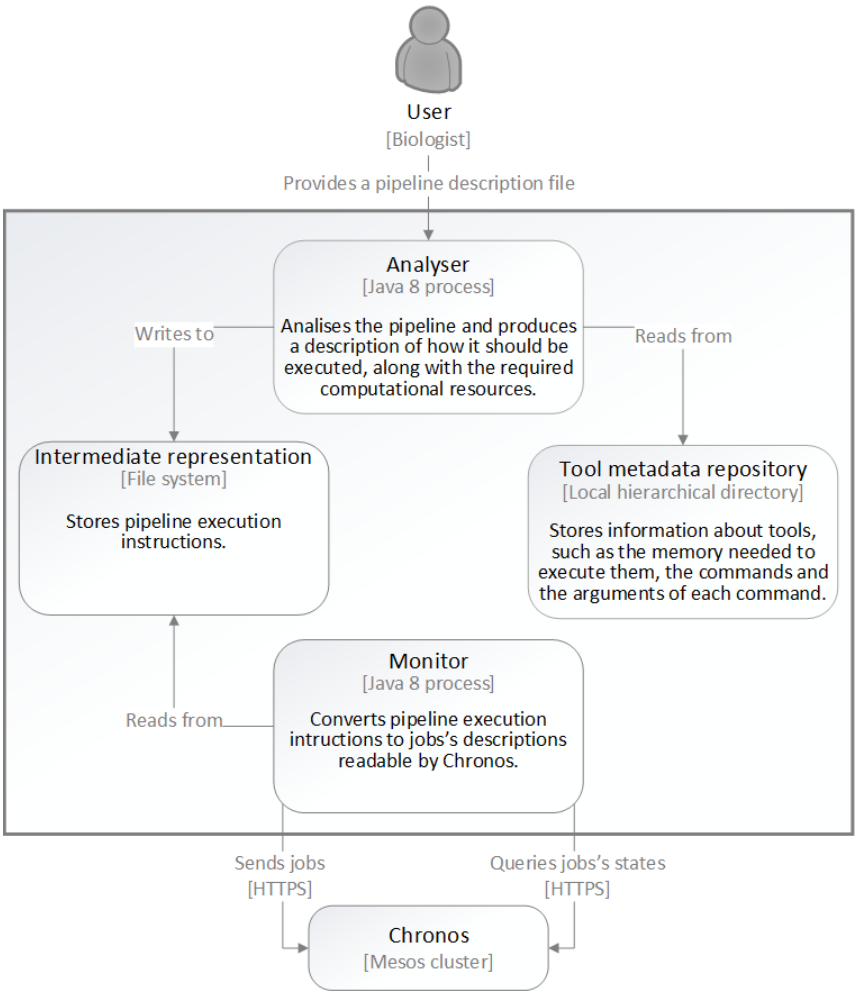

Figure 4.1. Architecture container diagram

4.2 Components

In the following subsections we take a brief look at the components of the NGS4Cloud solution.

4.2.1 Tool Meta-data Repository

In NGS4Cloud we adopted NGSPipes solution and modified it to suit the new needs. We now require users to specify in the tool meta-data information regarding the tool type, each commands' recommended computational resources and input arguments. The tool type is necessary as NGS4Cloud has 4 different tool types, which are NGS data processing tools, splitting tools, joining tools, and listing tools, while NGSPipes had only the first. The new tool types were introduced to support data partitioning. In 5.1 tool types will be more thoroughly explained. The computational

resources information was added as this information is important to help running each command in a machine that supports its computational resources needs. This includes multi-core execution of tool's commands that support it. Input arguments were added to help the analyser inferring dependencies between pipeline tasks. This too will be better explored in 5.1.

As of now, NGS4Cloud only has support for repositories stored locally or in GitHub. If users pretend to use other storages, they will have to extend the code to support it.

The schemas for the updated JSON files to match NGS4Cloud modifications to the tool meta-data mentioned above can be found in appendix C.

4.2.2 Domain-specific language

The DSL is a component reused from NGSPipes and no syntax modifications were done to it or primitives added.

In 5.1, we will explain how although we did not modify the syntax of the DSL from NGSPipes to NGS4Cloud, we did change how it is processed. We explain how it is processed and transformed into an intermediate representation and the motivation for that.

Since there were no syntax modifications, applications made in the context of NGSPipes that work with the DSL, like editors, can also be used in the context of NGS4Cloud.

4.2.3 Analyser

It's the analyser who is tasked with mapping the pipeline description into an intermediate representation, which has all the information needed for the parallel execution of the pipeline.

The analyser receives a pipeline description and validates it against the information stored in the tool meta-data repository. Then it associates to each tool of the pipeline the resources required for its execution with information which is also stored in the tool meta-data repository. Through the analysis of the pipeline's description it then proceeds to create a topologically sorted representation based on the dependencies between tools' commands. This representation is stored in a file. The analyser is developed in Java8 as it reuses components from NGSPipes's engine which is written in Java.

4.2.4 Monitor

The Monitor parses a pipeline execution instructions file and produces the corresponding jobs description that Chronos understands. The monitor then schedules jobs using the respective descriptions. Besides scheduling jobs, the monitor also queries Chronos to check their progression state and allows the user to download the pipeline's final outputs. We implemented the Monitor in Java8.

Mesos and Chronos

Aside from characteristics like fault tolerance and high availability, Mesos provides support for Docker containers and is currently the target of plenty of development efforts. Having support for Docker is an essential feature for our system since the NGS data processing tools setup is achieved through Docker images: Docker enables tools' automatic setup. By accompanying Mesos development, NGS4Cloud may benefit from future Mesos updates. Hoping that Mesos some day can manage an entire grid system and not a single cluster of machines, our system will also be able to run on a grid system. From a set of Mesos frameworks, Chronos is the only one we found focused on batch job scheduling which offers Docker support, job dependency scheduling and an API.

4.2.5 Intermediate Representation

Unlike Chronos, there are batch job scheduling frameworks that don't offer an API, but instead offer a library to schedule jobs. These libraries are not necessarily written in Java, the analyser's development language. In order to decouple the Monitor from the Java programming language, thus allowing for a wider range of scheduling frameworks to be explored, we established an intermediate representation in JSON format that is stored in a text file.

4.3 Data Model

The main entity of NGS4Cloud is the pipeline. A pipeline, in our domain problem, can be seen as a directed acyclic graph (DAG) where the vertexes are the tools' commands and the edges the input dependencies between them. To help representing a pipeline in NGS4Cloud we created the abstraction task. Task represents a tool command, a list of tasks represents a pipeline. A task contains the following properties:

- id
- dockerImage
- command
- mem
- disk
- cpus
- parents

The `id` property is an integer which identifies a task in the current pipeline. The `dockerImage` property is the name of the docker image stored in DockerHub which will be used to set up a tool, which will be executed on the cluster. The `command` property is the command line instruction which will cause the execution of the tool. The `mem` property is the necessary RAM, in MB, to execute the command. The `disk` property is the necessary disk space, in MB, to execute the command. The `cpus` property is the number of necessary cores to execute the command, and it can be fractional. The `parents` property is the representation of the dependencies between tasks; it is a list which contains the id of all the tasks a task depends upon.

The abstraction task is present throughout all of NGS4Cloud execution, although not always in the same format.

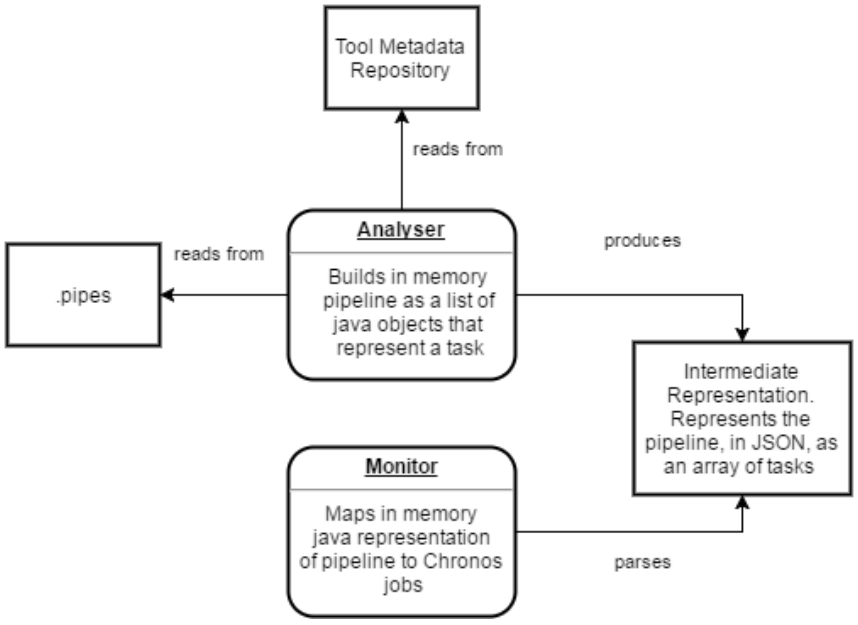

Figure 4.2. Pipeline abstraction flow diagram

Figure 4.2 expresses all the transformations the pipeline representation suffers during the execution of NGS4Cloud. To understand why these transformations occur, see 4.2.5. The pipeline representation, built on tasks, is firstly created in the analyser with the information present in the tool meta-data repository and in the `.pipes` file. The pipeline exists in the analyser as a list of java objects where each represents a task. These objects are then used to write the intermediate representation, in JSON. In the intermediate representation the pipeline is represented by a property named tasks, which is an array. Each object of the array tasks represents a task. The monitor will parse the intermediate representation, and then the pipeline will again be represented as a list of java objects that each represent a task. Having the tasks in memory the monitor will map each one of them to a Chronos Job and send them to be processed.

4.4 Conclusion

We believe we've achieved a highly modular architecture where each component is loosely coupled with each other, and therefore easily replaced. This makes our system more testable, extensible and adaptable. Scalability is also a quality due to the use of Mesos.

5

Implementation

In this chapter we see in more detail each component of NGS4Cloud, how they interact and their functionalities.

5.1 Analyser

In this section we will describe the *analyser*, in particular:

- How it processes the pipeline description file;
- How it uses the tool meta-data repository;
- How it handles data partitioning;
- How it determines the dependencies between pipeline tasks.

The analyser is the component that parses and processes the `.pipes` files and creates a file holding the intermediate representation that contains the instructions and information, like resources, necessary to run the pipeline. Using the pipeline description and the tool's meta-data repository to validate and enrich the pipeline description, the analyser produces an intermediate representation containing detailed instructions and information on how to execute the pipeline.

The analyser is a command line application with a single functionality. To produce an intermediate representation JSON file, from a pipeline description, `.pipes` file. For each command it will produce one or more tasks which will be specified in the intermediate representation. It receives as a parameter the file name of the `.pipes` file, the URL of the starting inputs, and the relative paths of the output files the users want to have access to once the pipeline is done running. The input URL can point to: a folder on the file system; or a remote located `.zip` or `.tar` file, that when extracted results in a folder with all the pipeline inputs.

Using the information passed in the pipeline description and present in the tool meta-data repository, the analyser can verify that the selected outputs will be produced when the pipeline is executed.

To parse the pipeline description, the NGSPipes parser is reused. The analyser generates in-memory representations of Java objects that resemble the pipeline. The information of these objects is validated using the tool meta-data repository to check the correctness of the different tools, commands and arguments preventing the user from inserting incorrect information and producing erroneous pipelines.

Additionally, the analyser fetches from the tool meta-data repository, the information regarding each command's recommended computational resources, the tools' configurators and the tool type.

Combining the tool meta-data repository information about command inputs and outputs and the value of the corresponding arguments, present in the newly generated Java pipeline representation, the analyser also infers dependencies between tasks. If the output of command A is the input of command B, then command A depends on command B, thus B must be executed only after A, in series. These dependencies are represented in the intermediate representation through the tasks' property `parents`.

Each tasks' `command` property string is then composed from the information contained in the Java objects correspondent to the command and its arguments. Since not every command's arguments are composed (or concatenated) in the same way, the tool meta-data repository contains information regarding the tool argument composer. Our library supports some common arguments composer and more can be added.

Data partitioning is solved in the analyser. Initially, we planned to automatize the data partitioning but figured that an optimal, generic way, to partition the data is not trivial in the DNA sequencing and NGS data world, since there is a plethora of different ways to split the different file formats. Instead, we give the users the option to break the data how they find appropriate. We decided to allow different types of tools besides the NGS data processing tools: the *splitting* tools, the *listing* tools, and the *joining* tools. These tools are utilised just like any other tool through the DSL. They are, however, processed differently by the analyser, depending on their tool type, identified in the tool meta-data.

Unlike NGS data processing tools, where each command is mapped into one task, a splitting command within a splitting tool generates one task corresponding to the

splitting of the file plus N tasks per command, where N is the number of partitions of the file whereas each task processes a partition of the file.

Data partitioning allows users to work with and process multiple files having to specify each command only once, while treating the files like a single file. It means that when users split a file in ten, they do not have to include the same tool ten times in the pipeline for every partition, just like if they had split the file using a splitting tool: the analyser will do that for them.

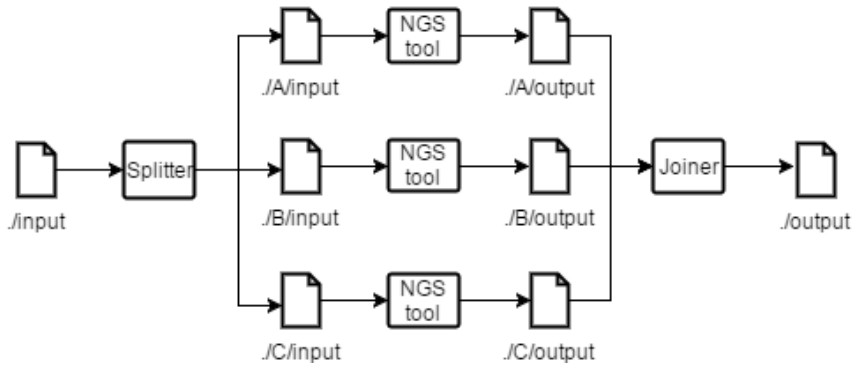

Figure 5.1. Splitting and Joining tools example

Figure 5.1 shows how a file named `input` is split originating three different files with the same name, stored in directories with different names.

For each partition the analyser will generate a directory where it stores the file partition with the same name it had before being partitioned. For every command specified in the pipeline description that uses the partitioned file, it is generated a task where the input path (partitioned file's path) is concatenated with the name of the directory where the partition of the file is stored. Multiple directories are created to avoid name collision between files generated.

Joining tools generate a task to join the partitions with the name of the input, that are stored in analyser generated directories (through either splitting or listing) corresponding to that input. Commands whose input depend on the join output will no longer have their tasks multiplied per partition.

In figure 5.2 a user wants to process different files of the same type, using the same tools, without having to specify each command more than once. Listing tools move and rename the files to match the same pattern as the splitting tools. After files are listed, they can be treated as one, as if it had been a split.

Listing tools generate a task to move the files to the newly generated directories and change the files names to match the name used in the `.pipes` file. Listing tools

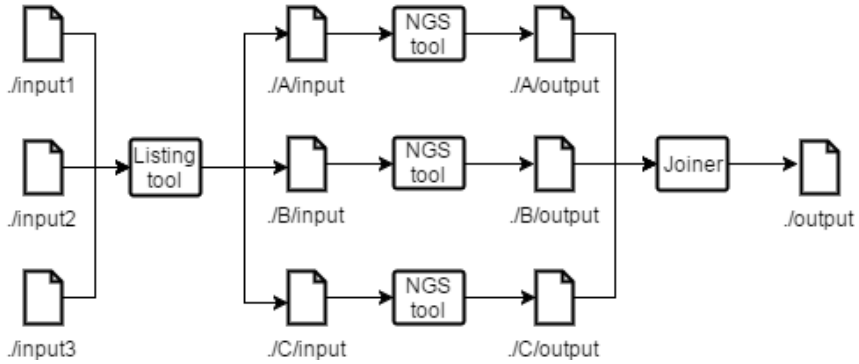

Figure 5.2. Listing and Joining tools example

have the same effect of the splitting tools on commands that depend from them. The listing tools purpose is to allow a user to provide multiple partitions of a file and treat them as one in the `.pipes` file.

The way data partitioning and dependencies inference is implemented, allows users to benefit from parallelization without adding complexity to the DSL and the process of writing a `.pipes` file. Users can now also describe the pipeline without caring for the order they state the commands since the order of their execution is determined by their inputs and outputs. They do need to bear in mind the semantics of chain and that it points to previously stated commands or tools.

The analyser also skims all outputs produced by the different commands to fill an array, belonging to the intermediate representation, with the directories that need to be created to process the pipeline with no problems.

With the current version of the analyser, we have achieved a solution that allows users to partition data and that allows to infer a topological graph from task dependencies, enabling the parallelization of the pipeline execution, without increasing the DSL complexity.

5.2 Intermediate Representation

In this section we explore the intermediate representation and explain what each property represents, how it is produced from a pipeline description and how it should be used by a Monitor.

The *intermediate representation* was created to decouple the analyser from the Monitor at the language level. This means that we can develop the analyser in a different programming language than the Monitor. Currently we use the Chronos framework to schedule jobs on the cluster. The Chronos framework makes available

a RESTApi, which means that we can interact with the framework without depending on a specific language. However, there are other Mesos's frameworks and there might be in the future more Mesos's frameworks that do not provide a RESTApi and, instead, provide a client library to schedule the jobs, which are not necessarily written in Java. This way, using the intermediate representation allows NGS4Cloud to use other frameworks to schedule jobs.

The intermediate representation is a JSON object, stored in a JSON format file. It follows the schema found in appendix A. The properties of the IR object are:

- Inputs
- Tasks
- Directories
- Outputs

Inputs

The `inputs` property is an array of input objects where each object is comprised of the following properties:

- `src` - a string, whose value is always an URL, thus allowing users to specify locally or remotely stored files;
- `filesDests` - an array of objects with the properties `file` and `dest`. `file` is a string and indicates a file that is going to be used as input of the pipeline. More specifically, the property `file` is a relative path to the file inside a directory indicated by the property `src` case `src` indicates a local directory. Case `src` indicates a remote URL, the property `file` will be a path relative to the local to where the directory is downloaded. The property `dest` is a string and indicates the path relative to the working directory of the pipeline, in the cluster (or wherever it is executed), where the file should be saved. In the context of NGS4Cloud, the inputs are passed as argument to the analyser.

Tasks

The `tasks` property is an array of tasks. Each task corresponds to the execution of a tool command, whether it is a NGS data processing tool, a splitting tool, a joining tool, or a listing tool. The *task* entity is the representation of a task in the context of a pipeline. It has information about the tool command to be executed, like the required resources and the docker image to setup the tool, as well as information

about the execution itself, like which tasks it depends upon. Its properties are the following:

- `id` - unique inside a pipeline, identifies a task;
- `dockerImage` - string which identifies where the tool is installed. Is relevant to setup the tool;
- `command` - will cause the execution of the command with the necessary inputs. Is a string;
- `mem` - is an integer which specifies the amount of RAM, in MB, needed to run the command;
- `disk` - is an integer which specifies the amount of disk, in MB, needed to run the command;
- `cpus` - is a decimal which specifies the number of cpus needed to run the command;
- `parents` - array which contains the ids of all the tasks from which the task depends.

Directories

The `directories` property is an array of paths that contains every single directory that is necessary to exist in the cluster's file system to properly execute the pipeline. This is necessary to take the responsibility of analysing the command and determining which directories should be created, during the execution of the tasks, away from the Monitor. The values present in this array are inferred by the analyser from the description of the pipeline and the meta-data of tools.

Outputs

The `outputs` property consists on an array of strings and each one identifies which files the user desires to download after the execution of the pipeline. The values contained in the array are passed by the user in the analyser's command line.

The intermediate representation is a format to describe pipelines. It has no explicit dependencies towards the NGS4Cloud context and can be used in other projects that need to represent a pipeline. It can be created via mechanisms other than the analyser's parse and processing of a `.pipes` file.

5.3 Cluster

In this section we will cover the current structure of the cluster used for the remote execution of the pipeline.

When building the *cluster* structure our goals were to make efficient use of all its available resources, to guarantee we could execute independent pipeline tasks, in parallel, and to ensure it had support for Docker in order to allow an easy installation of any NGS data tool used in the pipeline.

As already said in 4.2.4, to meet the goals set for the cluster we used the frameworks Mesos and Chronos. Mesos guarantees that the resources of our cluster will be efficiently used and it has Docker support. Chronos is a Mesos framework built for the execution of batch job which has support for dependent jobs.

Although Mesos has support for Docker, it is not included in its distribution. Thus all Mesos-Slaves of the cluster must have Docker installed.

However, the execution of the pipeline in a distributed environment brought a problem regarding the access of all machines to the files being produced during its execution. To solve this problem we integrated in the cluster a distributed file system to which all the Mesos-Slaves must have access. This way all Mesos-Slaves can execute any task of the pipeline without having to do any special procedure to have access to the file they are going to process. Currently; the distributed file system configured in the cluster must follow the Network File System Protocol (NFS)[1], which allows access to the distributed file system like a local storage is accessed, thus allowing us to use a task's tool command exactly like it was specified in the intermediate representation.

As referenced in 5.1, the pipeline inputs can be remotely located in compressed files `.zip` or `.tar`. Due to this fact, all Mesos-Slaves should have, as well, the application wget[2], so that the files can be downloaded to the cluster. They should also have the applications unzip and tar[3], in order to be able to extract the input files from the compressed file.

When setting up the execution of the pipeline there is the concern to build a hierarchy of directories that match the necessary structure to execute all the pipeline tasks, as explained in 5.2. This means that before starting the pipeline execution we need to access the distributed file system to build the required directories. To achieve this, the cluster must provide a machine capable of being accessed through

[1] https://tools.ietf.org/html/rfc1094, last visited on 05/06/2016
[2] https://www.gnu.org/software/wget/, last visited on 05/06/2016
[3] https://www.gnu.org/software/tar/, last visited on 05/06/2016

Secure Shell (SSH)[4] so that the directories can be created. Moreover the pipeline input files can also be stored in the user's machine, therefore SSH is also needed to upload the initial input files that are located in the user's machine. Besides, the SSH connection is also necessary to download the pipeline outputs.

As already mentioned, the cluster must have Chronos installed. However, besides having Chronos installed, an endpoint to the Chronos RESTApi must be made available so that NGS4Cloud can schedule the pipeline tasks to be executed.

A cluster used by NGS4Cloud must provide:

1. An endpoint to Chronos RESTApi;
2. Support the execution of Docker jobs on all Mesos-Slaves;
3. A NFS accessible on all Mesos-Slaves;
4. SSH access to a cluster machine that can interact with the cluster's NFS;
5. Support for wget, unzip and tar on all Mesos-Slaves

Creating an optimized cluster infrastructure to process NGS Data is not a goal of our project. However, we believe that the current constraints of the cluster structure do not limit the possibility of using a fully optimized cluster for this purpose.

The use of Mesos and Chronos, besides allowing us to achieve the goals set in the beginning of this chapter, also decouples our solution from any concrete cloud service like Microsoft Azure or Amazon Cloud Services. Since Mesos and Chronos are currently being actively developed, we hope that NGS4Cloud will be able to take advantage of improvements to these technologies. It is also important to notice that Mesos and Chronos provide features that could improve NGS4Cloud in ways that are not contemplated in this project. As an example, although data locality is not considered when executing a pipeline, Mesos offers a system called `constraints` that enables the locality of a file to be taken into account when choosing the cluster machine to execute the pipeline task.

5.4 Monitor

In this section we explore the software component monitor, namely its functionalities, the multiple interactions it has with the cluster, and how they are supported.

The `monitor` is a command line application which has three functionalities:

1. Launch the remote execution of the pipeline from an intermediate representation;
2. Consult the current status of the launched pipeline;

[4] https://tools.ietf.org/html/rfc4253, last visited on 05/06/2016

3. Download the outputs produced by the pipeline.

To support the last two functionalities, the monitor uses persistent storage to save the information about the Chronos Jobs that constitute a pipeline execution and what output files should be downloaded once the execution is finished.

5.4.1 Launch pipeline execution

To launch the execution of the pipeline a user can run the following command:

```
java Monitor.jar launch ir.json
```

The ir.json file used on the command above is a file containing the intermediate representation of the pipeline to be launched.

The process to launch the pipeline starts by parsing the given intermediate representation into java objects that will contain the information stored passed in the intermediate representation.

After parsing the intermediate representation follows the initialization of the cluster. In this step an SSH access to a machine of the cluster will be made in order to set up a proper environment for the execution of the pipeline. Specifically it will be created, on the distributed file system, the hierarchy of directories needed for the execution and will be uploaded the input files that are stored on the user's machine. Notice that the information needed to do the set up was obtained from the memory representation of the intermediate representation.

In situations where the input files are remotely stored the setup is not yet completed. We need as well to download those inputs to the cluster. As referred in 5.1 a remotely stored input files must be compressed with an extension `.zip` or `.tar`. This is done through a Chronos job that will use the application wget to download the compressed input file, extract its content and then move each input file to its destination, which is described in the intermediate representation. The application used to extract the input file, unzip or tar, is determined by the extension of the same. The download of remotely stored input files is not done during the SSH connection to a machine of the cluster because that would mean the connection would have to stay established until the download is over, which may be hours or even days depending on the dimension of the input files. This is a concern due to the large dimension NGS data files can have. Making this procedure a Chronos job eliminates the necessity for a long SSH connection to a cluster's machine, reducing the possibility of failure during the pipelines launch due to problems with the user's machine network connection. We know that an input file is stored locally or remotely from

the property `src` of the intermediate representation. Since it is always an URL, we can infer from the scheme the input file locations.

After the setup is complete we can launch the tasks described in the intermediate representation as Chronos Jobs. We will do this using Chronos RESTApi. if a Chronos Job was launched to download the inputs files to the cluster, all the Chronos Jobs created from a task will have a dependency to the download job. This is necessary in order to assure no Job starts before the cluster set up is completed.

After launching the execution, an id to identify it is generated. The id is then saved in a persistent storage. It is associated with the name of the Chronos Jobs that compose the pipeline and the output files that should be downloaded to the user's machine once the execution is finished. The output files that should be downloaded are specified in the intermediate representation.

This command returns the id of the pipeline execution.

5.4.2 Consult pipeline status

To consult a pipeline progression a user can run the following command:

```
java Monitor.jar status id
```

This command receives an id which identifies a pipeline execution.

Using the id, the monitor accesses the persistent storage and fetches the name of all the Chronos jobs associated with it. Then a request to list all the jobs current status is made to the Chronos RESTApi. The information retrieved from the request is then matched with the information fetched from the persistent storage in order to know which jobs belong to the pipeline execution specified with the id.

If any of the Chronos Jobs that constitute the pipeline failed, this command will return an error message. In case all the jobs are finished, a success message is returned. When the execution is still in progress and no error has occurred, a message indicating how many jobs have been finished and how many are in total is presented.

5.4.3 Download pipeline outputs

To download the outputs of the pipeline a user can run the following command:

```
java Monitor.jar outputs id
```

This command receives an id which identifies a pipeline execution.

First an access to the persistent storage is made to retrieve all the stored information regarding a pipeline execution. In the same way that is done when consulting the pipeline's progress status, a request to the Chronos RESTApi will be made, and

its result will be matched with the data stored in the persistent storage to determine if the pipeline execution has already finished. If the execution is yet to be concluded, an error message is returned. When the pipeline has finished executing, an SSH connection is established to a cluster machine, which has access to the distributed file system, and the specified outputs will be downloaded to the directory in which the user has executed this command.

5.4.4 Configurations

The monitor configurations are done through the environment variable `NGS4_CLOUD_MONITOR_CONFIGS`, whose value points to a file in which all the configurations are specified.

The configuration is a `.txt` file containing pairs name-value separated by an equals sign. In it are stored information regarding the SSH access to the cluster, the path to a directory of the cluster's distributed file system, and the Chronos endpoint. When writing the configuration file the following properties are required:

- `SSH_HOST`;
- `SSH_PORT`;
- `SSH_USER`;
- `CHRONOS_HOST`;
- `CHRONOS_PORT`;
- `CLUSTER_SHARED_DIR_PATH`.

`SSH_HOST` and `SSH_PORT` define the endpoint to access a machine of the cluster, via SSH, which has access to the directory belonging to the distributed file system of the cluster. `SSH_USER` defines the user used when establishing an SSH connection to cluster's machine. `CHRONOS_HOST` and `CHRONOS_PORT` define the endpoint to access Chronos. `CLUSTER_SHARED_DIR_PATH` is the path to a directory of the distributed file system.

To establish an SSH connection using password authentication one should define the property `SSH_PASS`. To establish an SSH connection using public key authentication one must define the following properties:

- `SSH_PRIVATE_KEY_PATH`;
- `SSH_PRIVATE_KEY_PASS`;
- `SSH_PUBLIC_KEY_PATH`.

`SSH_PRIVATE_KEY_PATH` is the path to the file with the private key; the password associated with the private key is `SSH_PRIVATE_KEY_PASS`; and `SSH_PUBLIC_KEY_PATH` is the path to the file with the public key.

The configuration file must have the information necessary to do one type of authenticated SSH access to the cluster. Unauthenticated SSH accesses are not supported. An example file can be found in appendix B.

6

Plan and progress

Date	Duration(in weeks)	Assignments
03/03/2016	1	- Introduction to Docker and Mesos technologies
10/03/2016	1	- Further study of Mesos technology - Introduction to Marathon and other Mesos frameworks
17/03/2016	1	- Write project proposal - Investigate existent batch job scheduling Mesos frameworks - Study Mesos custom frameworks
24/03/2016	1	- Write project proposal - Further study Mesos custom frameworks
31/03/2016	1	- Deliver project proposal - Study Chronos framework
07/04/2016	2	- Develop monitor
21/04/2016	1	- Test monitor - Wrap up progress report
28/04/2016	1	- Deliver progress report
05/05/2016	3	- Develop analyser
26/05/2016	1	- Test analyser
02/06/2016	1	- Create poster - Prepare Beta version for delivery
09/06/2016	1	- Deliver poster - Deliver Beta version
16/06/2016	2	- Test NGS4Cloud - Finish the report
30/06/2016	1	- Final delivery

Table 6.1. Plan calendar

As predicted in the progress report, the analyser and monitor were developed and are near their final state. We hope to follow the schedule and test NGS4Cloud as a whole in the next couple of weeks, and deliver the final solution in July.

References

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672. 3

B. Dantas and C. Fleitas. Infraestrutura de suporte à execução de fluxos de trabalho para bioinformática. Diploma Thesis, Instituto Superior de Engenharia de Lisboa, 2015. 2

D. Greenberg. *Building applications on Mesos*. O'Reilly, Sebastopol, 2016. ISBN 149192652X.

R. Ignazio. *Mesos in Action*. Manning Pubns Co, City, 2016. ISBN 1617292923.

J. Nickoloff. *Docker in action*. Manning Publications, Shelter Island, NY, 2016. ISBN 1633430235.

S. C. Shuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18, Jan. 2008. 2

Appendices

A Intermediate Representation Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "inputs": {
          "type": "array",
          "items": {
            "type": "object",
                "properties": {
                  "src": {"type": "string", "format":"uri"},
                  "filesDests": {
                    "type": "array",
                        "items":{
                          "type": "object",
                          "properties": {
                            "file": {"type": "string", "minLength": 1},
                                "dest": {"type": "string", "minLength": 0}
                          },
                          "required": ["file", "dest"]
                        }
                  }
                }
```

```
    }
  },
  "outputs": {
    "type": "array",
    "items": { "type": "string" }
  }
  "tasks": {
    "type": "array",
    "items": {
      "type": "object",
          "properties":{
            "id":{"type":"integer"},
            "dockerImage":{"type":"string", "minLength": 1},
            "command":{"type":"string", "minLength": 1},
            "mem": {"type":"integer"},
            "disk": {"type":"integer"},
            "cpus": {"type":"integer"},
            "parents": {
              "type": "array",
                  "items":{
                    "type": "integer"
                  }
            }
          },
```

```
                "required": ["id", "dockerImage", "command", "mem", "disk", "cpus", "parents"]
            }
        },
        "directories": {
            "type": "array",
            "items": { "type": "string" }
        }
    },
        "required": ["inputs", "outputs", "tasks", "directories"]
}
```

B Monitor Configuration File

SSH_HOST = 127.0.0.1
SSH_PORT = 22
SSH_USER = ngs4cloud
SSH_PRIVATE_KEY_PATH = C:\Users\NGS4Cloud\privkey.ppk
SSH_PRIVATE_KEY_PASS = privkeyPass123
SSH_PUBLIC_KEY_PATH = C:\Users\NGS4Cloud\pubkey
CHRONOS_HOST = 127.0.0.1
CHRONOS_PORT = 8080
CLUST_SHARED_DIR_PATH = /shared

C TMR Schemas

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "author": {
      "type": "string"
    },
    "version": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "documentation": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
```

```
"setup": {
  "type": "array",
  "items": {
    "type": "string"
  }
},
"toolType": {
  "type": "string",
  "enum": [
    "NGS",
    "spliter",
    "joiner",
    "lister"
  ]
},
"requiredMemory": {
  "type": "integer"
},
"recommendedCpus": {
  "type": "integer"
},
"recommendedDiskSpace": {
  "type": "integer"
},
```

```
"commands": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string"
      },
      "command": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "priority": {
        "type": "integer"
      },
      "argumentsComposer": {
        "type": "string"
      },
      "arguments": {
        "type": "array",
        "items": {
          "type": "object",
```

```
"properties": {
  "name": {
    "type": "string"
  },
  "argumentType": {
    "type": "string",
    "enum": [
      "int",
      "file",
      "string",
      "double",
      "directory"
    ],
    "isRequired": {
      "type": "string",
      "enum": [
        "true",
        "false"
      ]
    }
  },
  "description": {
    "type": "string"
  }
```

```
      },
      "required": [
        "name",
        "argumentType",
        "isRequired",
        "description"
      ]
    }
  },
  "outputs": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string"
        },
        "description": {
          "type": "string"
        },
        "outputType": {
          "type": "string",
          "enum": [
            "directory_dependent",
```

```
            "file_dependent",
            "independent"
          ]
        },
        "argument_name": {
          "type": "string"
        },
        "value": {
          "type": "string"
        }
      },
      "required": [
        "name",
        "description",
        "outputType",
        "argument_name",
        "value"
      ]
    }
  },
  "inputs": {
    "type": "array",
    "items": {
      "type": "object",
```

```
  "properties": {
    "name": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "inputType": {
      "type": "string",
      "enum": [
        "directory_dependent",
        "file_dependent",
        "independent"
      ]
    }
  },
  "required": [
    "name",
    "description",
    "inputType",
    "argument_name",
    "value"
  ]
}
```

```
            }
        },
        "required": [
            "name",
            "command",
            "description",
            "priority",
            "argumentsComposer",
            "arguments",
            "outputs",
            "inputs"
        ]
    }
  }
},
"required": [
  "name",
  "author",
  "version",
  "description",
  "documentation",
  "setup",
  "toolType",
  "requiredMemory",
```

```
    "recommendedDiskSpace",
    "recommendedCpus",
    "commands"
  ]
}
```