SkyGrid Prototype - ”Cloud-friendly Grid platform”

Author:

Dimitrios Sarigiannis

Supervisor:

Andrey Ustyuzhanin

CERN openlab Summer Student report 2014

1 Abstract

Grid architecture is well-known for dealing with distributed computation for many years. Recently other computational models have started to emerge. For example, Hadoop that is based on map-reduce paradigm. The goal of this project would be specification of test cases and development of staging environment that is capable of dealing with those cases in order to provide comparison result between old and new models. Ultimately it would be helpful to identify weak points of both approaches as well as to figure out points of growth for both worlds.

Contents

2 Introduction

Grid architecture is well-known for dealing with distributed computation for many years. There is already a variety of architectures that have been studied and implemented such as PanDA. Moreover, SkyGrid was used for running jobs on cluster 40 machines (24 CPUs) for biology data processing and for MC generation for SHIP experiment. It takes about 10 days to generate 10^9 events. However, over the years, new ideas and challenges occurred such as privacy, faster navigation to data, independence of diversity of environments and so on.

The main challenge of our project is to build a prototype system for the SHiP grid-cloud framework and as a result the ability to run event simulation and event processing using grid interface to cloud technologies. Moreover, we care about providing support for private clouds and support for arbitrary container configuration so that we could benefit by running jobs in almost arbitrary environments. As a next step, It would be coveted to integrate our idea with a book keeping system so that the system and users could quickly and easily navigate to the output of the jobs using indexes. This project is separated into two parts. The first part introduces the reader to new concepts used in this project and is composed by section 3,4,5 and 6. The second part, presents and explain our own overall solution to the problem and it is composed by sections 7,8,9 and 10.

In the second section of this project, we provide some information about SHiP experiment at CERN. Then, in the third section, we present PAnDA main principal design features and architecture so that in the next sections, we could get a better idea about what are our main challenges. Next, to the fourth section, we give you an idea about what is Hadoop, since it is the main framework that we focus on our project, then we move on to the historical reasons of evolving Hadoop version 1 to Hadoop version 2 YARN and finally we present you the whole mechanism of Hadoop YARN. In Addition, in the fifth paragraph, we introduce you to docker, which is the main tool we used to support virtualization and we explain the main concepts of its new logic.

In the second part of our project, we present you the overall solution to our problem and therefore in the next section I emphasize to where was my personal contribution to this idea. Finally, we end up to a short conclusion about the whole idea and the expected results.

3 Ship Experiment

The SHiP Experiment is a new general-purpose fixed target facility at the SPS to search for hidden particles as predicted by a very large number of recently elaborated models of Hidden Sectors which are capable of accommodating dark matter, neutrino oscillations, and the origin of the full baryon asymmetry in the Universe. Specifically, the experiment is aimed at searching for very weakly interacting long lived particles including Heavy Neutral Leptons - right-handed partners of the active neutrinos; light supersymmetric particles - sgoldstinos, etc; scalar, axion and vector portals to the hidden sector.The high intensity of the SPS and in particular the large production of charm mesons with the 400 GeV beam allow accessing a wide variety of light long-lived exotic particles of such models and of SUSY. Moreover, the facility is ideally suited to study the interactions of tau neutrinos. [1]

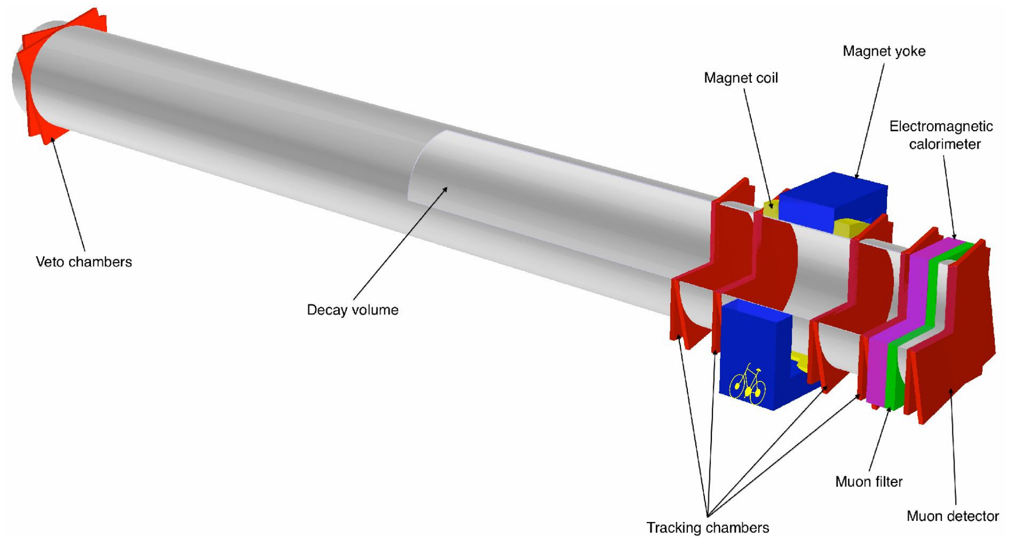


Figure 1: Experiment at the SPS to search for Hidden Particles

4 PanDA

In this section we will talk about what is PanDa, how it is designed and implemented and what is its architecture and workflow. The sources that we used in this section are [2] [3]

4.1 Introduction

The PanDA Production ANd Distributed Analysis system has been developed by ATLAS since summer 2005 to meet ATLAS requirements for a data-driven workload management system for production and distributed analysis processing capable of operating at LHC data processing scale. ATLAS processing and analysis places challenging requirements on:

- throughput
- scalability
- robustness
- efficient resource utilization
- minimal operations manpower
- tight integration of data management with processing workflow

PanDA was initially developed for US based ATLAS production and analysis, and assumed that role in late 2005. In October 2007 PanDA was adopted by the ATLAS Collaboration as the sole system for distributed processing production across the Collaboration. It proved itself as a scalable and reliable system capable of handling very large workflow. In 2008 it was adopted by ATLAS for user analysis processing as well.

4.2 Design and implementation

4.2.1 Principal design features

The principal features of PanDA's design are as follows:

- Support for both managed production and individual analysis.
- A coherent, homogeneous processing system layered over diverse and heterogeneous processing resources.
- Use of pilot jobs for acquisition of processing resources.
- Extensive direct use of Condor (particularly CondorG), as a job submission infrastructure of proven capability and reliability.
- Coherent and comprehensible system view afforded to users, and to PanDA's own job brokerage system, through a system-wide job database that records comprehensive static and dynamic information on all jobs in the system.
- A comprehensive monitoring system supporting production and analysis operations.
- System-wide site/queue information database recording static and dynamic information used throughout PanDA to configure and control system behavior from the regional level down to the individual queue level.
- Integrated data management based on dataset (file collection) based organization of data files, with datasets consisting of input or output files for associated job sets.
- utomated pre-staging of input data (either to a processing site or out of mass storage, or both) and immediate return of outputs, all asynchronously, minimizing data transport latencies and delivering (for analysis) the earliest possible first results.
- Support for running arbitrary user code (job scripts), as in conventional batch submission.

- Easy integration of local resources.
- Authentication and authorization is based on grid certificates, with the job submitter required to hold a grid proxy and VOMS role that is authorized for PanDA usage.
- Support for usage regulation at user and group levels based on quota allocations, job priorities, usage history, and user-level rights and restrictions.
- Security in PanDA employs standard grid security mechanisms.

4.2.2 Architecture and workflow

Jobs are submitted to PanDA via a simple python client interface by which users define job sets, their associated datasets and the input/output files within them. Job specifications are transmitted to the PanDA server via secure http (authenticated via a grid certificate proxy), with submission information returned to the client. This client interface has been used to implement PanDA front ends for ATLAS production, distributed analysis (e.g. pathena), and other experiments. The PanDA server receives work from these front ends into a global job queue, upon which a brokerage module operates to prioritize and assign work on the basis of job type, priority, input data and its locality, available CPU resources and other brokerage criteria. Allocation of job sets to sites is followed by the dispatch of corresponding input datasets to those sites, handled by a data service interacting with the distributed data management system. Data pre-placement is a soft (formerly strict) precondition for job execution: jobs are not released for processing until their input data is accessible from the processing site. When data dispatch completes, jobs are made available to a job dispatcher. Strict data colocation for analysis is being replaced by a softer requirement that data be accessible via an adequately performant network path, as determined by a networking site-to-site ’cost matrix’.

An independent subsystem manages the delivery of pilot jobs to worker nodes, using a number of remote (or local) job submission methods as defined in its configuration. A pilot once launched on a worker node contacts the dispatcher and receives an available job appropriate to the site. If no appropriate job

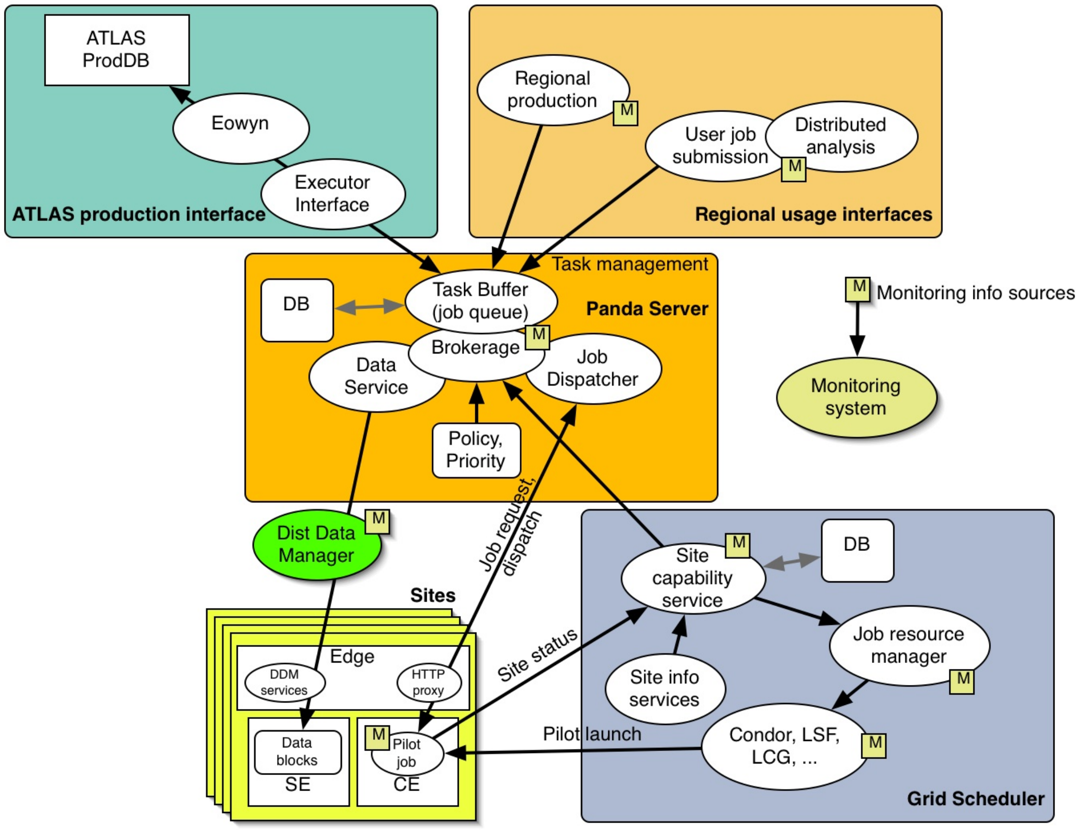


Figure 2: The PanDA architecture

is available, the pilot may immediately exit or may pause and ask again later, depending on its configuration (standard behavior is for it to exit). An important attribute of this scheme for interactive analysis, where minimal latency from job submission to launch is important, is that the pilot dispatch mechanism bypasses any latencies in the scheduling system for submitting and launching the pilot itself. The pilot job mechanism isolates workload jobs from grid and batch system failure modes (a workload job is assigned only once the pilot successfully launches on a worker node). The pilot also isolates the PanDA system proper from grid heterogeneities, which are encapsulated in the pilot, so that at the PanDA level the grid(s) used by PanDA appears homogeneous. Pilots generally carry a generic ’production’ grid proxy, with an additional VOMS attribute ’pilot’ indicating a pilot job. Analysis pilots may use glexec to switch their identity on the worker node to that of the job submitter (see PandaSecurity).

The isolation of the pilot system in the architecture allows for many different pilot system implementations to be smoothly integrated with PanDA, making it possible to integrate widely diverse processing resources with different requirements for a pilot service. These include supercomputers, cloud resources, and self-contained computing environments like NorduGrid.

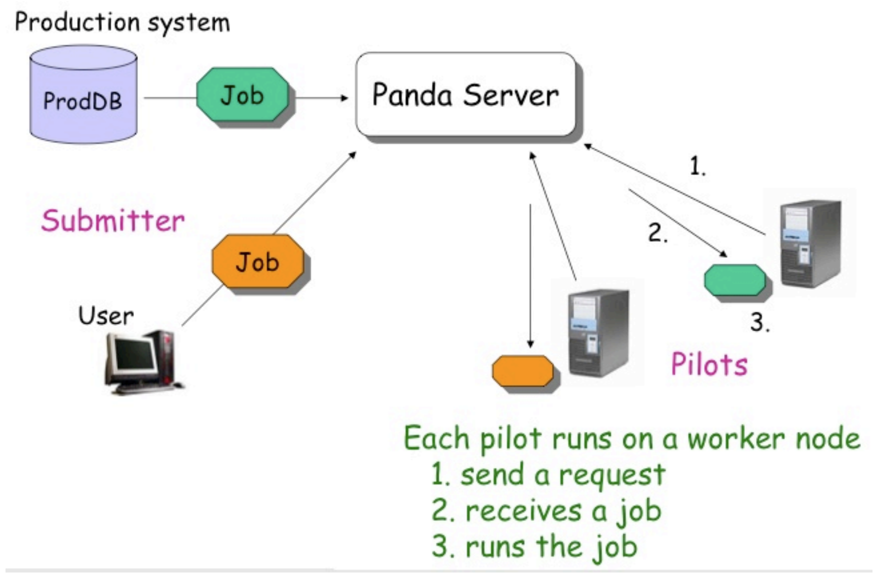


Figure 3: production system

A figure of the original implementation of PanDA is shown below.

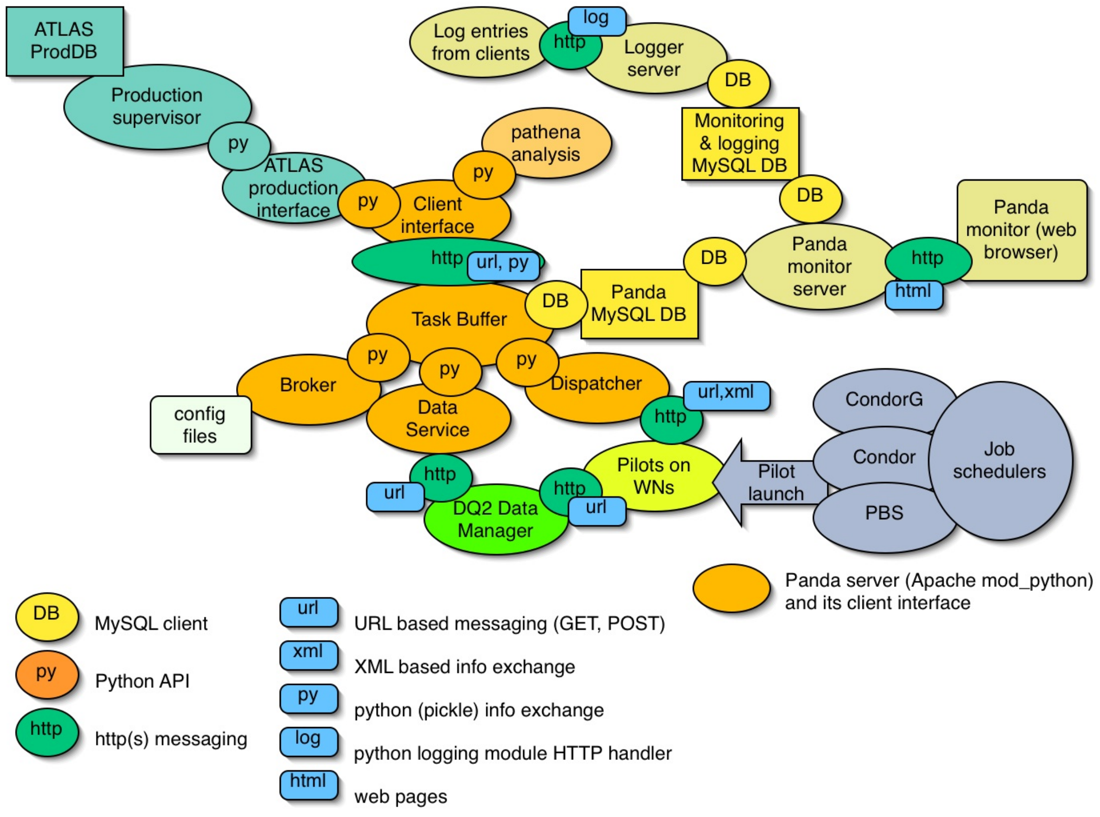


Figure 4: PanDA original implementation

5 YARN Approach

In this section we will discuss what is Hadoop with few words, explain why Hadoop is famous nowadays and present the constrains that drove Hadoop to the next generation which is YARN. Then, we will present summarily how YARN works and what are the advantages of its usage and finally we will describe HDFS and some relative tools that helped us in our project. The sources that we used in this section are [4] [5] [7] [8].

5.1 Introduction

YARN is the data operating system of Hadoop that enables you to process data simultaneously in multiple ways. YARN provides provides the resource management and pluggable architecture to enable a wide variety of data access methods to operate on data stored in Hadoop with predictable performance and service levels.

5.2 YARN: The next generation of Hadoop's compute platform

The main components of Hadoop YARN are:

- ResourceManager
- ApplicationMaster
- NodeManager
- A distributed application

YARN is the next generation of Hadoop's compute platform, as shown below.

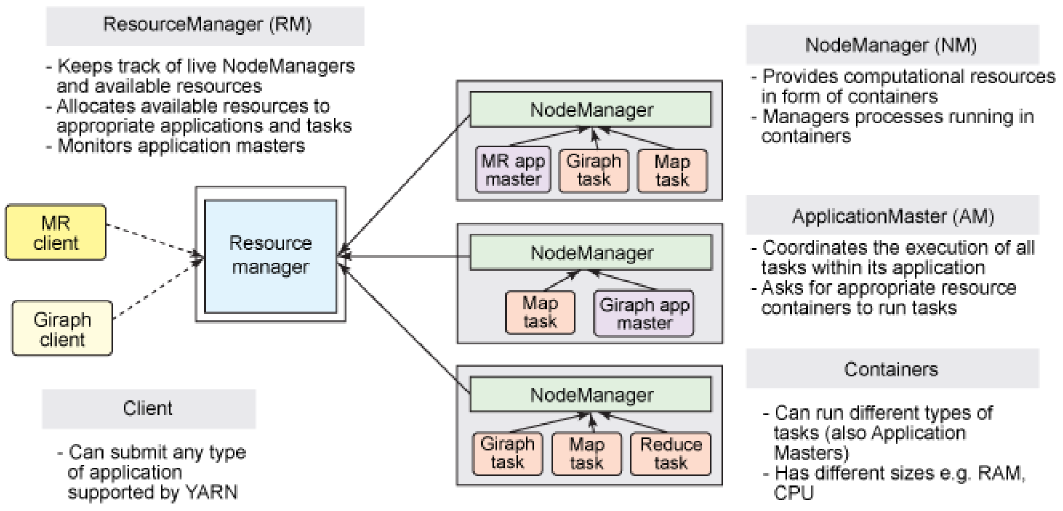

ResourceManager (RM)
- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters
MR client
Giraph client
Resource manager
Client
- Can submit any type of application supported by YARN
NodeManager
MR app master
Giraph task
Map task
NodeManager
Map task
Giraph app master
NodeManager
Giraph task
Map task
Reduce task
NodeManager (NM)
- Provides computational resources in form of containers
- Managers processes running in containers
ApplicationMaster (AM)
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks
Containers
- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

Figure 5: YARN Architecture

In the YARN architecture, a global ResourceManager runs as a master daemon, usually on a dedicated machine, that arbitrates the available cluster resources among various competing applications. The ResourceManager tracks how many live nodes and resources are available on the cluster and coordinates what applications submitted by users should get these resources and when. The ResourceManager is the single process that has this information so it can make its allocation (or rather, scheduling) decisions in a shared, secure, and multi-tenant manner (for instance, according to an application priority, a queue capacity, ACLs and data locality.

When a user submits an application, an instance of a lightweight process called the ApplicationMaster is started to coordinate the execution of all tasks within the application. This includes monitoring tasks, restarting failed tasks, speculatively running slow tasks, and calculating total values of application counters. These responsibilities were previously assigned to the single JobTracker for all jobs. The ApplicationMaster and tasks that belong to its application run in resource containers controlled by the NodeManagers.

The NodeManager is a more generic and efficient version of the TaskTracker. Instead of having a fixed number of map and reduce slots, the NodeManager has a number of dynamically created resource containers. The size of a container depends upon the amount of resources it contains, such as memory, CPU, disk, and network IO. Currently, only memory and CPU (YARN-3) are supported. cgroups might be used to control disk and network IO in the future. The number of containers on a node is a product of configuration parameters and the total amount of node resources (such as total CPUs and total memory) outside the resources dedicated to the slave daemons and the OS.

Interestingly, the ApplicationMaster can run any type of task inside a container. For example, the MapReduce ApplicationMaster requests a container to launch a map or a reduce task, while the Giraph ApplicationMaster requests a container to run a Giraph task. You can also implement a custom ApplicationMaster that runs specific tasks and, in this way, invent a shiny new distributed application framework that changes the big data world. I encourage you to read about Apache Twill, which aims to make it easy to write distributed applications sitting on top of YARN.

In YARN, MapReduce is simply degraded to a role of a distributed application (but still a very popular and useful one) and is now called MRv2. MRv2 is simply the re-implementation of the classical MapReduce engine, now called MRv1, that runs on top of YARN.

5.3 One cluster that can run any distributed application

The ResourceManager, the NodeManager, and a container are not concerned about the type of application or task. All application framework-specific code is simply moved to its ApplicationMaster so that any distributed framework can be supported by YARN — as long as someone implements an appropriate ApplicationMaster for it.

Thanks to this generic approach, the dream of a Hadoop YARN cluster running many various workloads comes true. Imagine: a single Hadoop cluster in your data center that can run MapReduce, Giraph, Storm, Spark, Tez/Impala, MPI, and more.

The single-cluster approach obviously provides a number of advantages, including:

- Higher cluster utilization, whereby resources not used by one framework could be consumed by another
- Lower operational costs, because only one ”do-it-all” cluster needs to be managed and tuned
- Reduced data motion, as there’s no need to move data between Hadoop YARN and systems running on different clusters of machines

Managing a single cluster also results in a greener solution to data processing. Less data center space is used, less silicon wasted, less power used, and less carbon emitted simply because we run the same calculation on a smaller but more efficient Hadoop cluster.

5.4 Application submission in YARN

This section discusses how the ResourceManager, ApplicationMaster, NodeManagers, and containers interact together when an application is submitted to a YARN cluster. The image below shows an example.

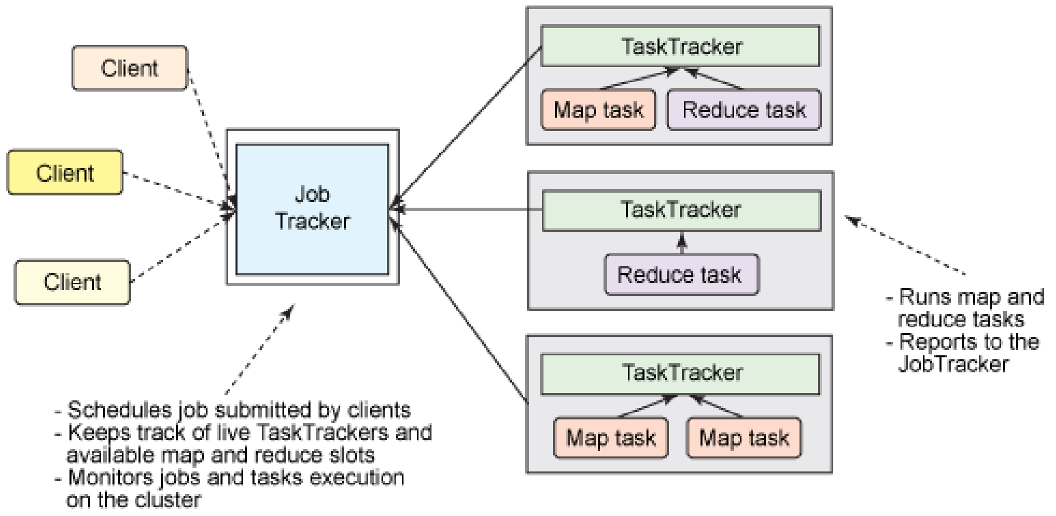

Client
Client
Client
Job Tracker
TaskTracker
Map task
Reduce task
TaskTracker
Reduce task
TaskTracker
Map task
Map task
- Runs map and reduce tasks
- Reports to the JobTracker
- Schedules job submitted by clients
- Keeps track of live TaskTrackers and available map and reduce slots
- Monitors jobs and tasks execution on the cluster

Figure 6: Application Submission in YARN

Suppose that users submit applications to the ResourceManager by typing the hadoop jar command in the same manner as in MRv1. The ResourceManager maintains the list of applications running on the cluster and the list of available resources on each live NodeManager. The ResourceManager needs to determine which application should get a portion of cluster resources next. The decision is subjected to many constraints, such as queue capacity, ACLs, and fairness. The ResourceManager uses a pluggable Scheduler. The Scheduler focuses only on scheduling; it manages who gets cluster resources (in the form of containers) and when, but it does not perform any monitoring of the tasks within an application so it does not attempt to restart failed tasks.

When the ResourceManager accepts a new application submission, one of the first decisions the Scheduler makes is selecting a container in which ApplicationMaster will run. After the ApplicationMaster is started, it will be responsible for a whole life cycle of this application. First and foremost, it will be sending resource requests to the ResourceManager to ask for containers needed to run an application's tasks. A resource request is simply a request for a number of containers that satisfies some resource requirements, such as:

- An amount of resources, today expressed as megabytes of memory and CPU shares
- A preferred location, specified by hostname, rackname, or * to indicate no preference
- A priority within this application, and not across multiple applications

If and when it is possible, the ResourceManager grants a container (expressed as container ID and hostname) that satisfies the requirements requested by the ApplicationMaster in the resource request. A container allows an application to use a given amount of resources on a specific host. After a container is granted, the ApplicationMaster will ask the NodeManager (that manages the host on which the container was allocated) to use these resources to launch an application-specific task. This task can be any process written in any framework (such as a MapReduce task or a Giraph task). The NodeManager does not monitor tasks; it only monitors the resource usage in the containers and, for example, it kills a container if it consumes more memory than initially allocated.

The ApplicationMaster spends its whole life negotiating containers to launch all of the tasks needed to complete its application. It also monitors the progress of an application and its tasks, restarts failed tasks in newly requested containers, and reports progress back to the client that submitted the application. After the application is complete, the ApplicationMaster shuts itself down and releases its own container.

Though the ResourceManager does not perform any monitoring of the tasks within an application, it checks the health of the ApplicationMasters. If the ApplicationMaster fails, it can be restarted by the ResourceManager in a new container. You can say that the ResourceManager takes care of the ApplicationMasters, while the ApplicationMasters takes care of tasks.

5.5 Interesting facts and features

YARN offers several other great features. Describing all of them is outside the scope of this article, but I've included some noteworthy features:

- Uberization is the possibility to run all tasks of a MapReduce job in the ApplicationMaster's JVM if the job is small enough. This way, you avoid the overhead of requesting containers from the ResourceManager and asking the NodeManagers to start (supposedly small) tasks.
- Binary or source compatibility for MapReduce jobs written for MRv1 (MAPREDUCE-5108).
- High availability for the ResourceManager (YARN-149). This work is in progress, and is already done by some vendors.
- An application recovery after the restart of ResourceManager (YARN-128). The ResourceManager stores information about running applications and completed tasks in HDFS. If the ResourceManager is restarted, it recreates the state of applications and re-runs only incomplete tasks. This work is close to completion and has been actively tested by the community. It is already done by some vendors.

- Simplified user-log management and access. Logs generated by applications are not left on individual slave nodes (as with MRv1) but are moved to a central storage, such as HDFS. Later, they can be used for debugging purposes or for historical analyses to discover performance issues.
- A new look and feel of the web interface.

5.6 HDFS

Hadoop Distributed File System (HDFS) is a Java-based file system that provides scalable and reliable data storage that is designed to span large clusters of commodity servers. HDFS, MapReduce, and YARN form the core of Apache Hadoop.

5.6.1 What HDFS does

HDFS was designed to be a scalable, fault-tolerant, distributed storage system that works closely with MapReduce. HDFS will "just work" under a variety of physical and systemic circumstances. By distributing storage and computation across many servers, the combined storage resource can grow with demand while remaining economical at every size.

These specific features ensure that the Hadoop clusters are highly functional and highly available:

- Rack awareness allows consideration of a node's physical location, when allocating storage and scheduling tasks
- Minimal data motion. MapReduce moves compute processes to the data on HDFS and not the other way around. Processing tasks can occur on the physical node where the data resides. This significantly reduces the network I/O patterns and keeps most of the I/O on the local disk or within the same rack and provides very high aggregate read/write bandwidth.
- Utilities diagnose the health of the files system and can rebalance the data on different nodes

- Rollback allows system operators to bring back the previous version of HDFS after an upgrade, in case of human or system errors
- Standby NameNode provides redundancy and supports high availability
- Highly operable. Hadoop handles different types of cluster that might otherwise require operator intervention. This design allows a single operator to maintain a cluster of 1000s of nodes.

5.6.2 How HDFS Works

An HDFS cluster is comprised of a NameNode which manages the cluster metadata and DataNodes that store the data. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, or namespace and disk space quotas.

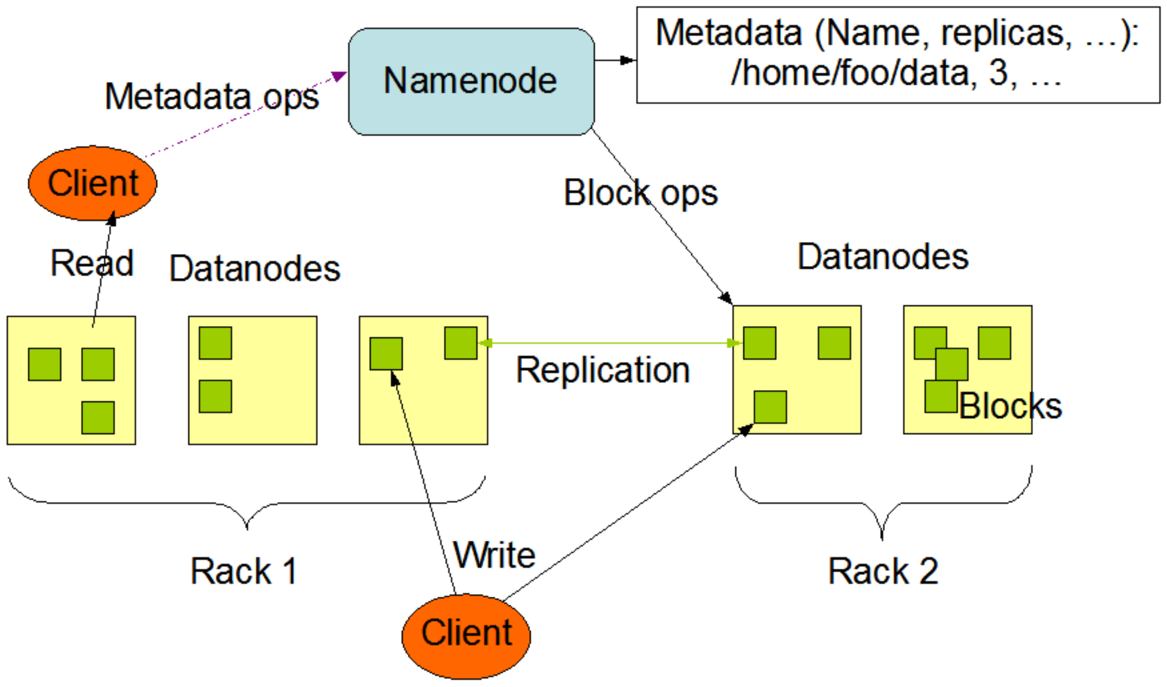


Figure 7: HDFS Architecture

The file content is split into large blocks (typically 128 megabytes), and each block of the file is independently replicated at multiple DataNodes. The blocks are stored on the local file system on the datanodes. The Namenode actively monitors the number of replicas of a block. When a replica of a block is lost due to a DataNode failure or disk failure, the NameNode creates another replica of the block. The NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.

The NameNode does not directly send requests to DataNodes. It sends instructions to the DataNodes by replying to heartbeats sent by those DataNodes. The instructions include commands to: replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, or shut down the node.

5.7 Apache Hive

Apache Hive is the defacto standard for SQL queries over petabytes of data in Hadoop. It is a comprehensive and compliant engine that offers the broadest range of SQL semantics for Hadoop, providing a powerful set of tools for analysts and developers to access Hadoop data.

The HiveQL language requires the same familiar skills and semantics that experienced analysts already understand for database SQL queries, providing a familiar way to make interactive queries. Finally, Apache Hive easily integrates with existing tools using a familiar JDBC interface.

5.7.1 What Hive Does

Hadoop was built to organize and store massive amounts of data. A Hadoop cluster is a reservoir of heterogeneous data, from multiple sources and in different formats. Hive allows the user to explore and structure that data, analyze it, and then turn it into business insight.

5.7.2 How Hive Works

The tables in Hive are similar to tables in a relational database, and data units are organized in a taxonomy from larger to more granular units. Databases are comprised of tables, which are made up of partitions. Data can be accessed via a simple query language, called HiveQL, which is similar to SQL. Hive supports overwriting or appending data, but not updates and deletes.

Within a particular database, data in the tables is serialized and each table has a corresponding Hadoop Distributed File System (HDFS) directory. Each table can be sub-divided into partitions that determine how data is distributed within sub-directories of the table directory. Data within partitions can be further broken down into buckets.

Hive supports primitive data formats such as TIMESTAMP, STRING, FLOAT, BOOLEAN, DECIMAL, BINARY, DOUBLE, INT, TINYINT, SMALLINT and BIGINT. In addition, primitive data types can be combined to form complex data types, such as structs, maps and arrays.

Here are some advantageous characteristics of Hive:

- Familiar Hundreds of unique users can simultaneously query the data using a language familiar to SQL users.
- Fast Response times are typically much faster than other types of queries on the same type of huge datasets.
- Scalable and extensible As data variety and volume grows, more commodity machines can be added to the cluster, without a corresponding reduction in performance.
- Informative Familiar JDBC and ODBC drivers allow many applications to pull Hive data for seamless reporting. Hive allows users to read data in arbitrary formats, using SerDes and Input/Output formats.

5.8 HCatalog

Apache HCatalog is a table and storage management layer for Hadoop that enables users with different data processing tools – Apache Pig, Apache MapReduce, and Apache Hive – to more easily read and write data on the grid. HCatalog's table abstraction presents users with a relational view of data in the Hadoop Distributed File System (HDFS) and ensures that users need not worry about where or in what format their data is stored. HCatalog displays data from RCFile format, text files, or sequence files in a tabular view. It also provides REST APIs so that external systems can access these tables' metadata.

5.8.1 What HCatalog Does

Apache HCatalog provides the following benefits to grid administrators:

- Frees the user from having to know where the data is stored, with the table abstraction
- Enables notifications of data availability
- Provides visibility for data cleaning and archiving tools

5.8.2 How HCatalog Works

HCatalog supports reading and writing files in any format for which a Hive SerDe (serializer-deserializer) can be written. By default, HCatalog supports RCFile, CSV, JSON, and SequenceFile formats. To use a custom format, you must provide the InputFormat, OutputFormat, and SerDe.

HCatalog is built on top of the Hive metastore and incorporates components from the Hive DDL. HCatalog provides read and write interfaces for Pig and MapReduce and uses Hive's command line interface for issuing data definition and metadata exploration commands. It also presents a REST interface to allow external tools access to Hive DDL (Data Definition Language) operations, such as "create table" and "describe table".

HCatalog presents a relational view of data. Data is stored in tables and these tables can be placed into databases. Tables can also be partitioned on one or more keys. For a given value of a key (or set of keys) there will be one partition that contains all rows with that value (or set of values).

5.9 Cloudera

Figure 8: cloudera

Cloudera Inc. is an American-based software company that provides Apache Hadoop-based software, support and services, and training to business customers.

Cloudera's open-source Apache Hadoop distribution, CDH (Cloudera Distribution Including Apache Hadoop), targets enterprise-class deployments of that technology. Cloudera says that more than 50% of its engineering output is donated upstream to the various Apache-licensed open source projects (Apache Hive, Apache Avro, Apache HBase, and so on) that combine to form the Hadoop platform. Cloudera is also a sponsor of the Apache Software Foundation.

5.10 Conclusion

YARN is a completely rewritten architecture of Hadoop cluster. It seems to be a game-changer for the way distributed applications are implemented and executed on a cluster of commodity machines.

YARN offers clear advantages in scalability, efficiency, and flexibility compared to the classical MapReduce engine in the first version of Hadoop. Both small and large Hadoop clusters greatly benefit from YARN. To the end user (a developer, not an administrator), the changes are almost invisible because it's possible to run unmodified MapReduce jobs using the same MapReduce API and CLI.

There is no reason not to migrate from MRv1 to YARN. The biggest Hadoop vendors agree on that point and offer extensive support for running Hadoop YARN. Today, YARN is successfully used in production by many companies, such as Yahoo!, eBay, Spotify, Xing, Allegro, and more.

6 Docker

In this section we will describe what is Docker, what are the advantages of its usage comparing with virtual machines and we will present Docker's main components and architecture. The sources that we used in this section are [9] [10].

Docker is an open platform for developing, shipping, and running applications. Docker is designed to deliver your applications faster. With Docker you can separate your applications from your infrastructure AND treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code.

Docker does this by combining a lightweight container virtualization platform with workflows and tooling that help you manage and deploy your applications.

At its core, Docker provides a way to run almost any application securely isolated in a container. The isolation and security allow you to run many containers simultaneously on your host. The lightweight nature of containers, which run without the extra load of a hypervisor, means you can get more out of your hardware.

Surrounding the container virtualization are tooling and a platform which can help you in several ways:

- getting your applications (and supporting components) into Docker containers
- distributing and shipping those containers to your teams for further development and testing
- deploying those applications to your production environment, whether it be in a local data center or the Cloud

6.1 How is docker different from Virtual Machines?

6.1.1 Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.

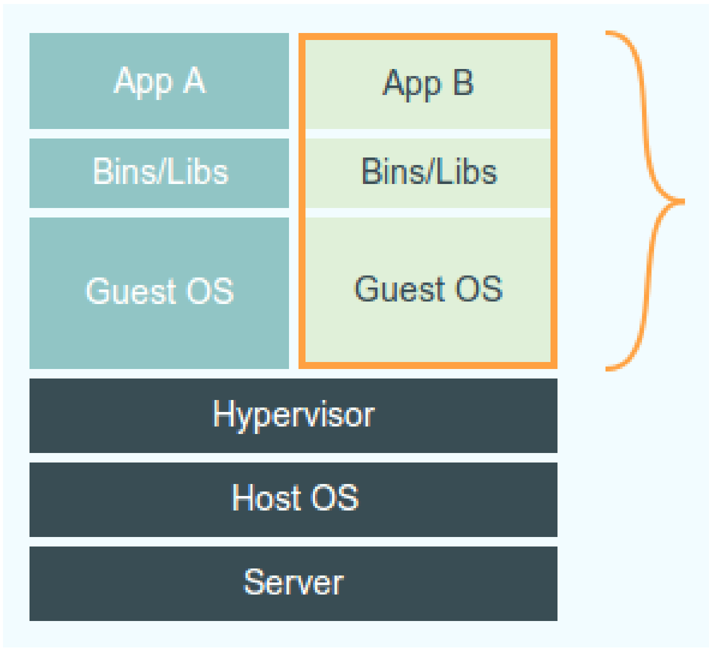


Figure 9: Virtual Machines

6.1.2 Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

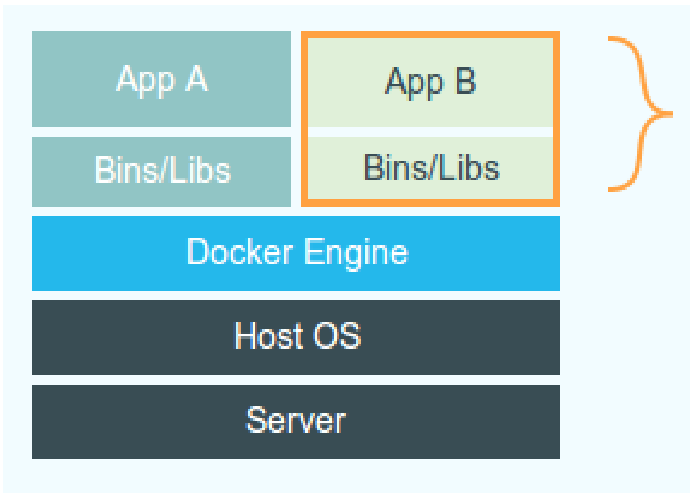


Figure 10: Docker

6.2 Docker architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. Both the Docker client and the daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate via sockets or through a RESTful API.

6.2.1 Docker Daemon

As shown in the diagram above, the Docker daemon runs on a host machine. The user does not directly interact with the daemon, but instead through the Docker client.

6.2.2 Docker Client

As shown in the diagram above, the Docker daemon runs on a host machine. The user does not directly interact with the daemon, but instead through the Docker client.

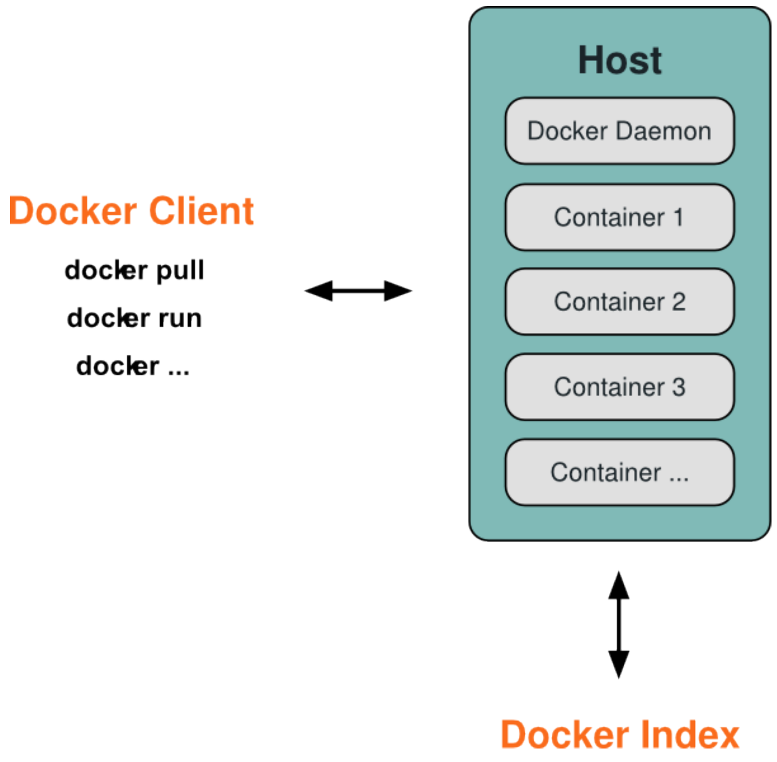


Figure 11: docker-architecture

6.3 Docker inside

To understand Docker's internals, you need to know about three components:

- Docker images
- Docker registries
- Docker containers

6.3.1 Docker images

A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created. Docker images are the build component of Docker.

6.3.2 Docker registries

Docker registries hold images. These are public or private stores from which you upload or download images. The public Docker registry is called Docker Hub. It provides a huge collection of existing images for your use. These can be images you create yourself or you can use images that others have previously created. Docker registries are the distribution component of Docker.

6.3.3 Docker containers

Docker containers are similar to a directory. A Docker container holds everything that is needed for an application to run. Each container is created from a Docker image. Docker containers can be run, started, stopped, moved, and deleted. Each container is an isolated and secure application platform. Docker containers are the run component of Docker.

6.4 Why do we use Docker

6.4.1 Faster delivery of your applications

Docker is perfect for helping you with the development lifecycle. Docker allows your developers to develop on local containers that contain your applications and services. It can then integrate into a continuous integration and deployment workflow.

For example, your developers write code locally and share their development stack via Docker with their colleagues. When they are ready, they push their code and the stack they are developing onto a test environment and execute any required tests. From the testing environment, you can then push the Docker images into production and deploy your code.

6.4.2 Deploying and scaling more easily

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local host, on physical or virtual machines in a data center, or in the Cloud.

Docker's portability and lightweight nature also make dynamically managing workloads easy. You can use Docker to quickly scale up or tear down applications and services. Docker's speed means that scaling can be near real time.

6.4.3 Achieving higher density and running more workloads

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines. This is especially useful in high density environments: for example, building your own Cloud or Platform-as-a-Service. But it is also useful for small and medium deployments where you want to get more out of the resources you have.

7 Overall Solution

Our overall solution could be described by figure 12.

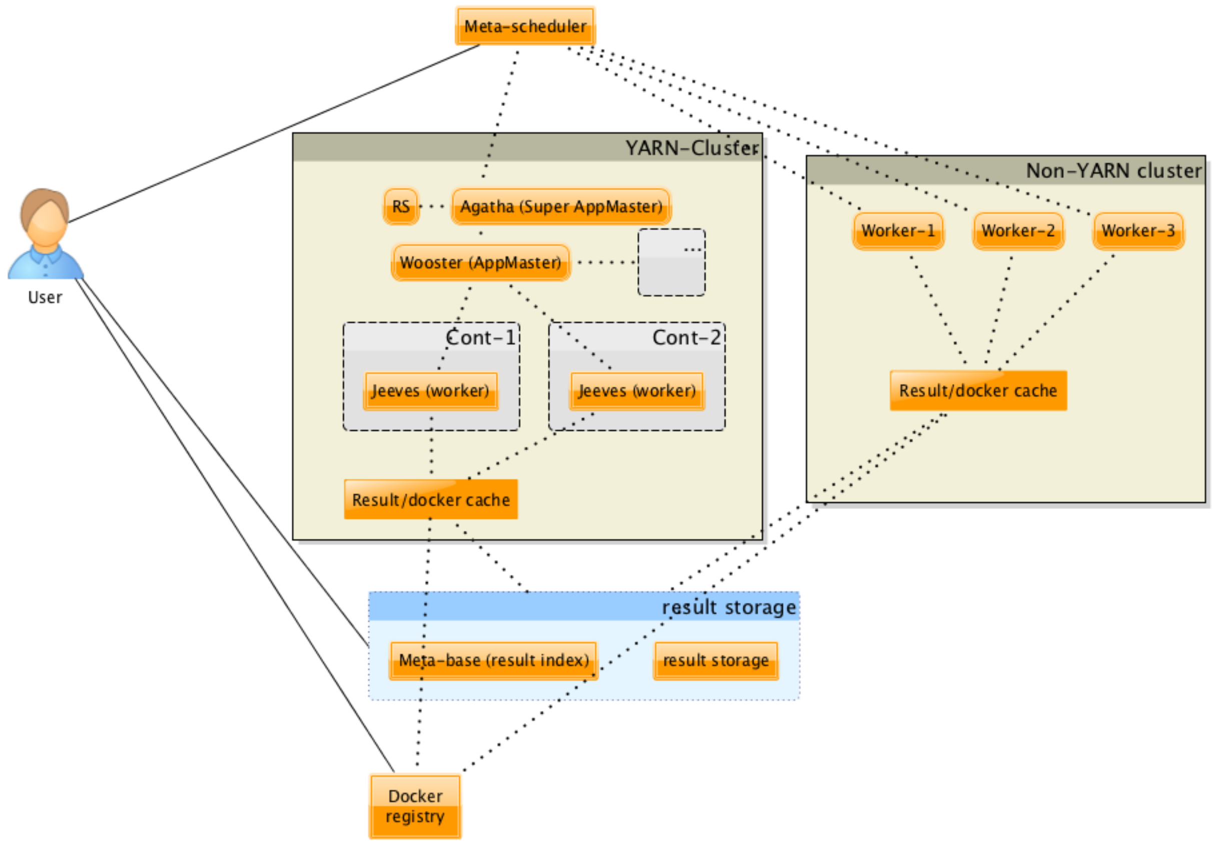


Figure 12: skygrid-architecture

As you can see, our architecture consists of the below main components:

- Client
- Meta-Scheduler
- Yarn Cluster

  - Agatha
  - Wooster
  - Resource Manager
  - Containers (Jeeves)
  - cache
- Non-Yarn Clusters
- Storage
- docker registry

In the next sections we are going to describe all the components and the flow of this architecture.

7.1 Client

We assume that there are many users who have a necessity to compute something to the system. These jobdescriptors are submitted first by a client that could be a form in a browser or a JSON file. In our example, we use the latter way to submit jobs. This file contains specification for a job. For instance, some of the details are responsible for the environment, such as the command, number of the containers, working directory and environments and some other details are responsible for the parameters of the job which could be the number of events that are going to be executed, a set, a range and a random seed.

An instance of a specific job with 1000 events could be like the figure 13 below.

7.2 Meta-Scheduler Service

As a next step the job submitted by the user to the client, is passed to the Meta-Scheduler. Meta-Scheduler is a service that has two main responsibilities:

- Control a queue of all the tasks/jobs that the client submits
- Split these jobs in smaller chunks

The Client can ask Meta-Scheduler about a task status or termination.

7.3 Data Centers

7.3.1 Hadoop YARN

In a sequel, we can observe that there are various clusters in the architecture. In our case, we focused on Hadoop-YARN cluster.

7.3.2 Superior Application Master

Inside Hadoop-YARN cluster, Agatha which is the Superior Application Master and runs always by default in one dedicated machine. Moreover, according to it's internal state is responsible to ask Meta-Scheduler for jobs from the Meta-Scheduler. When it gets a least one job it ask for all the required resources from the Resource Manager. After, Resource Manager returns all the required resources, Agatha has to pass the job to Wooster.

7.3.3 Resource Manager

As we said before, Yarn has a Resource Manager which runs as a master daemon, usually on a dedicated machine, that arbitrates the available cluster resources among various competing applications. The Resource Manager tracks how many live nodes and resources are available on the cluster and coordinates what resources should Agatha get.

7.3.4 Wooster

Then, another Application Master which is called Wooster, is responsible to split the jobs by the number of the containers and start each job inside a container.

7.3.5 Containers

The containers that are created by Wooster are called Jeeves. Depending on the job descriptor, Jeeves is always responsible to perform these steps:

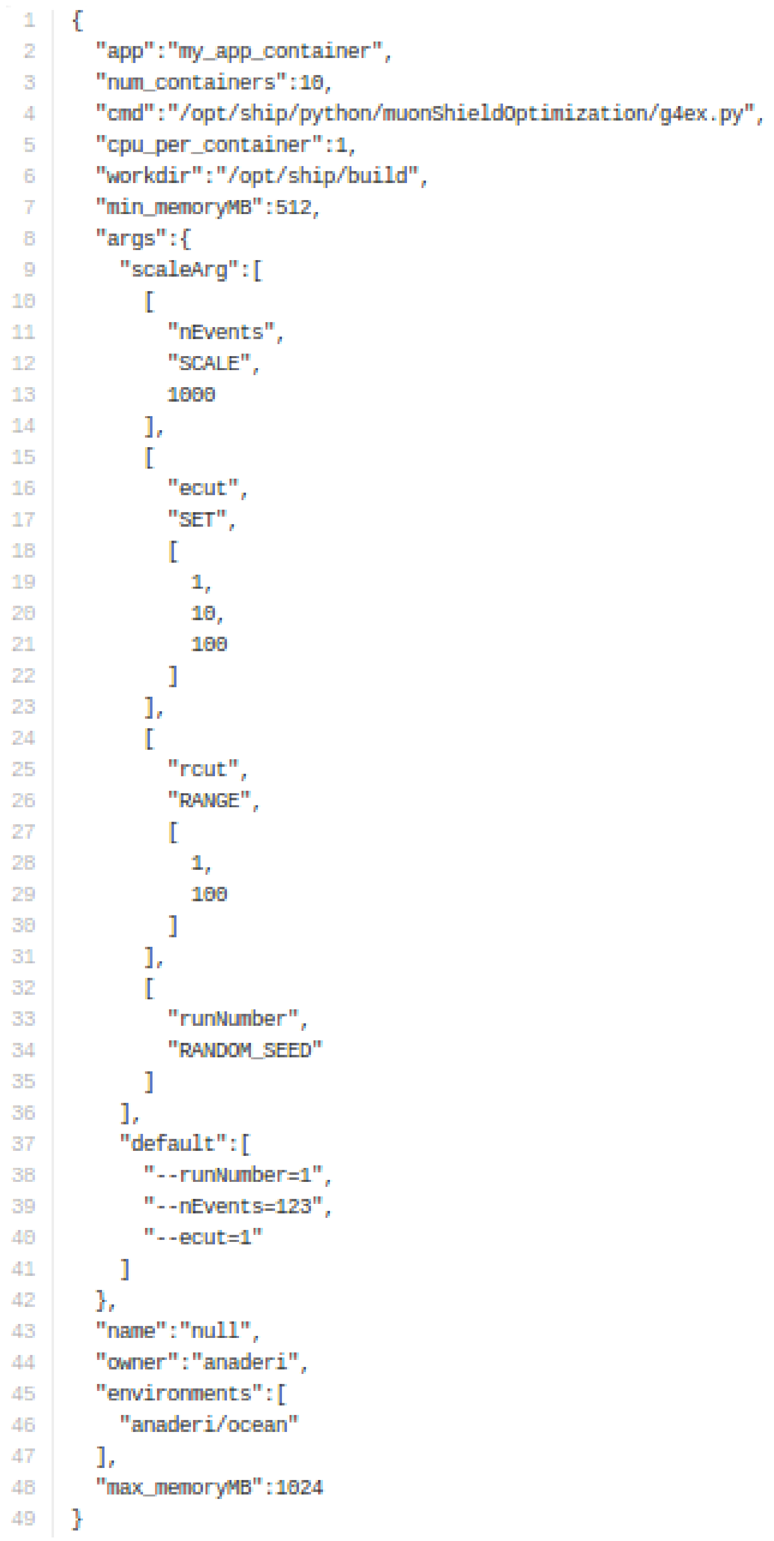
1. Take a jobdescriptor file as input
2. Parse jobdescriptor's file
3. Pull the corresponding image to docker
4. Create the working directory
5. Run this image via docker with the working directory mounted
6. Collect all the files
7. Send these files to a filesystem (HDFS)
8. Report status of job

7.4 HDFS

Finally, files (results of a Jeeves container) are stored to the HDFS outside the Data Center. Moreover, we use a cache database inside Hadoop Yarn and by building another database with meta-information about HDFS which is out of the Data Center, we achieve fast data access to all the files.

This flow has the following advantages:

- Meta-Scheduler has no knowledge about each Data Center computational load. If Application Master asks for a job, it'll get it. If not, Meta-Scheduler does not care.
- There is a bunch of different implementations as for the executors.



```
{
  "app":"my_app_container",
  "num_containers":10,
  "cmd":"/opt/ship/python/muonShieldOptimization/g4ex.py",
  "cpu_per_container":1,
  "workdir":"/opt/ship/build",
  "min_memoryMB":512,
  "args":{
    "scaleArg":[
      [
        "nEvents",
        "SCALE",
        1000
      ],
      [
        "ecut",
        "SET",
        [
          1,
          10,
          100
        ]
      ],
      [
        "rcut",
        "RANGE",
        [
          1,
          100
        ]
      ],
      [
        "runNumber",
        "RANDOM_SEED"
      ]
    ],
    "default":[
      "--runNumber=1",
      "--nEvents=123",
      "--ecut=1"
    ]
  },
  "name":"null",
  "owner":"anaderi",
  "environments":[
    "anaderi/ocean"
  ],
  "max_memoryMB":1024
}
```

Figure 13: job instance

8 GitHub

Figure 14: github

Our project is available on GitHub: https://github.com/anaderi/skygrid

8.0.1 Contributors

- Andrey Ustyuzhanin
- Konstantin Nikitin
- Sarigiannis Dimitrios

8.0.2 Support

Yandex

9 My Contribution

In this chapter I will describe what is my personal contribution to this project so far.

9.1 Test YARN

As we mentioned before, we focus on Hadoop YARN so, my first mission was to test Hadoop Yarn. I created a simple random number generator inside RNG folder to the project on GitHub and I tested it through Vagrant (https://github.com/Cascading/vagrant-cascading-hadoop-cluster) single-node environment so that everyone would be able to test it with the same configuration.

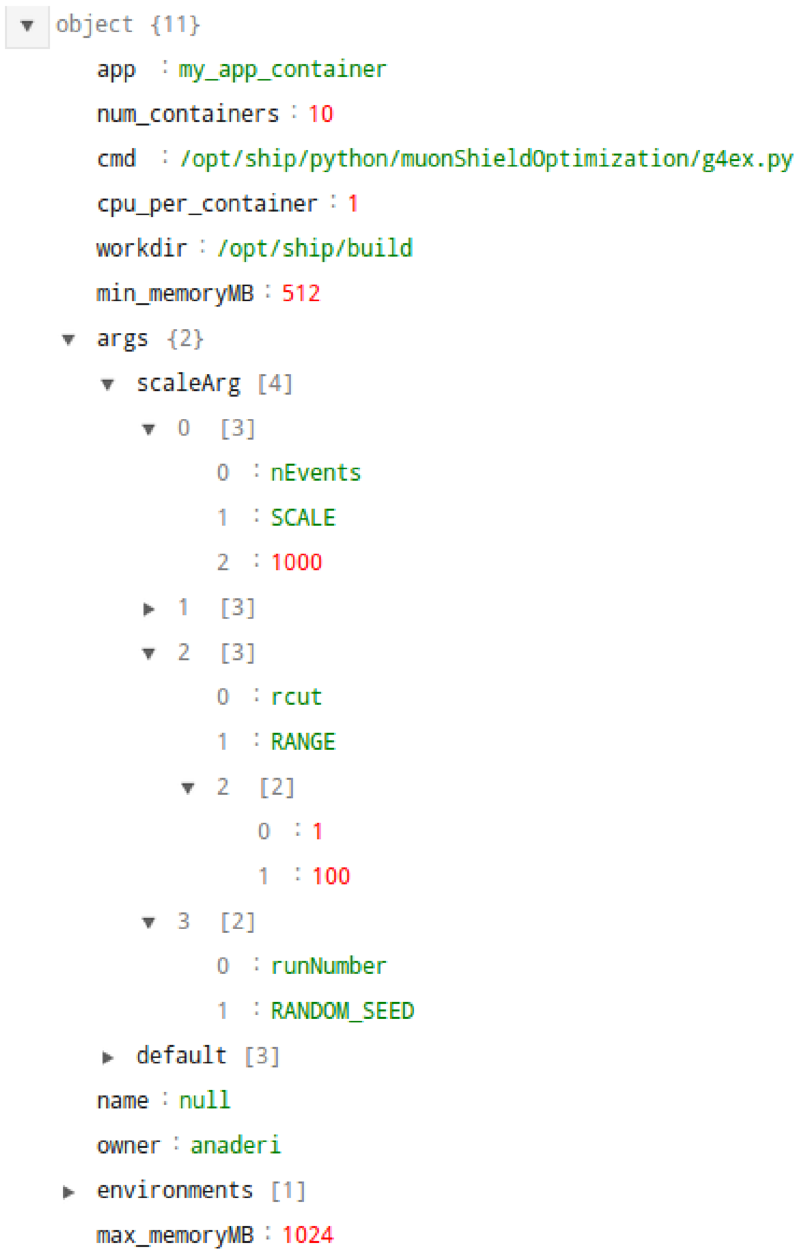
9.2 Split Jobs

The next step was concerning the Meta-Scheduler service and my task was to create some methods for splitting a jobdescriptor JSON file into sub-jobdescriptors in various ways. More specifically, I created a class JobDescriptor that can create object by parsing such JSON and that can split itself into smaller chunks. Splitting job into job chunks of smaller size, after execution of those chunks user will get results of the whole job. There are different strategies implemented for splitting:

1. equal(N), creates N smaller chunks out of job descriptor
2. proportional ($[p_1, p_2, \ldots, p_N]$), creates N chunks of sizes proportional to integers p_i
3. next(p), gives next job chunk of p items

After finishing this task, I also implemented tests for it with JUnit. Below, we will show you a trivial example for equal split method.

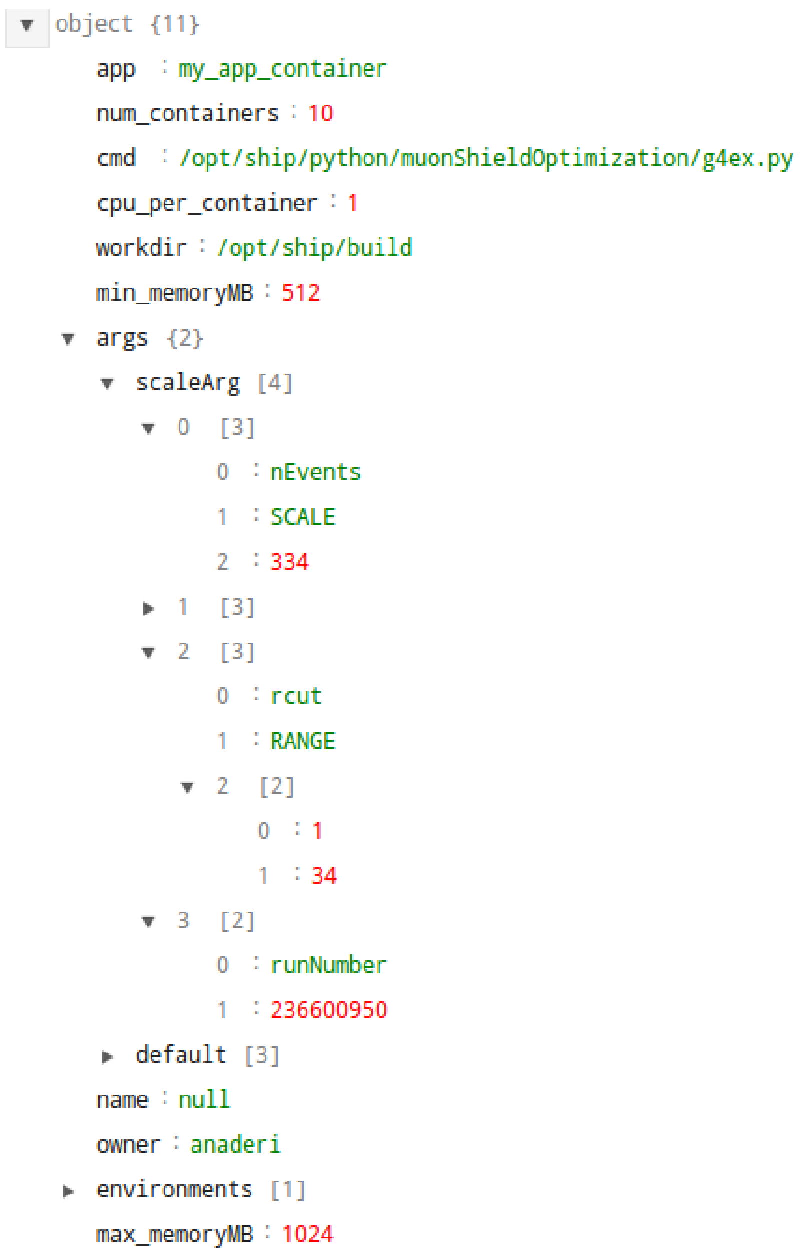
We assume that we have an initial jobdescriptor with specifications like in figure 15:



```
▼ object {11}
    app : my_app_container
    num_containers : 10
    cmd : /opt/ship/python/muonShieldOptimization/g4ex.py
    cpu_per_container : 1
    workdir : /opt/ship/build
    min_memoryMB : 512
  ▼ args {2}
    ▼ scaleArg [4]
      ▼ 0  [3]
           0 : nEvents
           1 : SCALE
           2 : 1000
      ▶ 1  [3]
      ▼ 2  [3]
           0 : rcut
           1 : RANGE
         ▼ 2  [2]
              0 : 1
              1 : 100
      ▼ 3  [2]
           0 : runNumber
           1 : RANDOM_SEED
    ▶ default [3]
    name : null
    owner : anaderi
  ▶ environments [1]
    max_memoryMB : 1024
```

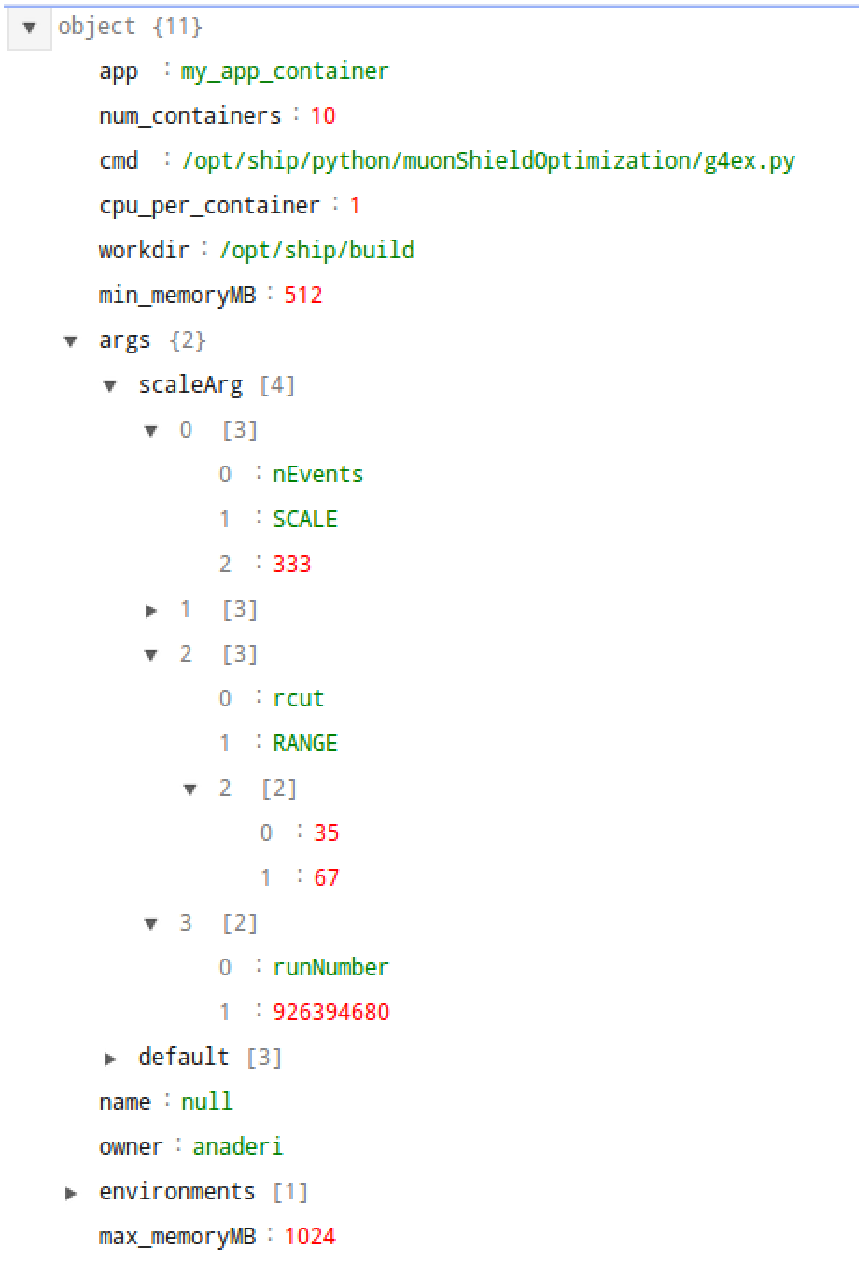
Figure 15: initial jobdescriptor

We want to split this job into 3 sub-jobdescriptors with the best equal way. Its is obvious that, after splitting this jobdescriptor we will have three sub-jobdescriptors (figures 16,17,18).



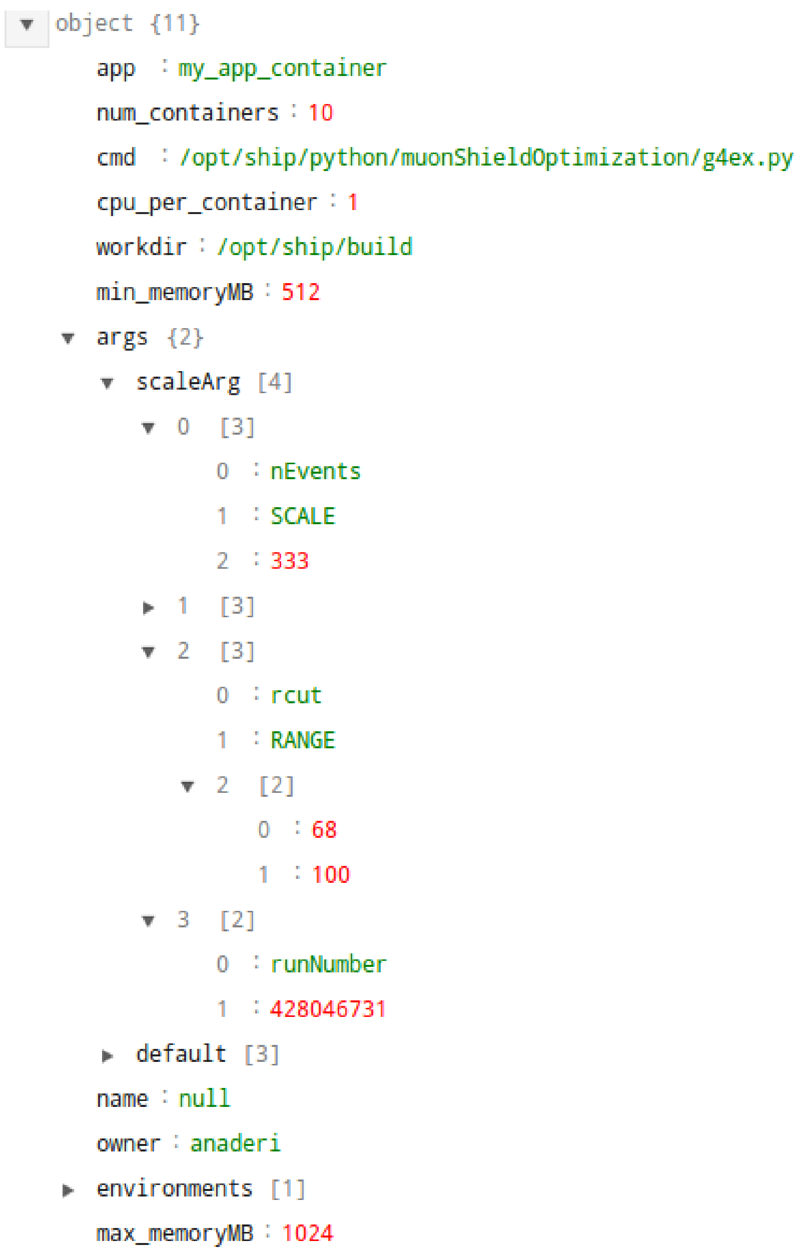
```
▼ object {11}
    app : my_app_container
    num_containers : 10
    cmd : /opt/ship/python/muonShieldOptimization/g4ex.py
    cpu_per_container : 1
    workdir : /opt/ship/build
    min_memoryMB : 512
  ▼ args {2}
    ▼ scaleArg [4]
      ▼ 0 [3]
          0 : nEvents
          1 : SCALE
          2 : 334
      ▶ 1 [3]
      ▼ 2 [3]
          0 : rcut
          1 : RANGE
        ▼ 2 [2]
            0 : 1
            1 : 34
      ▼ 3 [2]
          0 : runNumber
          1 : 236600950
    ▶ default [3]
    name : null
    owner : anaderi
  ▶ environments [1]
    max_memoryMB : 1024
```

Figure 16: sub-jobdescriptor1



```
▼ object {11}
    app : my_app_container
    num_containers : 10
    cmd : /opt/ship/python/muonShieldOptimization/g4ex.py
    cpu_per_container : 1
    workdir : /opt/ship/build
    min_memoryMB : 512
  ▼ args {2}
    ▼ scaleArg [4]
      ▼ 0 [3]
            0 : nEvents
            1 : SCALE
            2 : 333
      ▶ 1 [3]
      ▼ 2 [3]
            0 : rcut
            1 : RANGE
          ▼ 2 [2]
              0 : 35
              1 : 67
      ▼ 3 [2]
            0 : runNumber
            1 : 926394680
    ▶ default [3]
    name : null
    owner : anaderi
  ▶ environments [1]
    max_memoryMB : 1024
```

Figure 17: sub-jobdescriptor2



```
▼ object {11}
    app : my_app_container
    num_containers : 10
    cmd : /opt/ship/python/muonShieldOptimization/g4ex.py
    cpu_per_container : 1
    workdir : /opt/ship/build
    min_memoryMB : 512
  ▼ args {2}
    ▼ scaleArg [4]
      ▼ 0 [3]
          0 : nEvents
          1 : SCALE
          2 : 333
      ▶ 1 [3]
      ▼ 2 [3]
          0 : rcut
          1 : RANGE
        ▼ 2 [2]
            0 : 68
            1 : 100
      ▼ 3 [2]
          0 : runNumber
          1 : 428046731
    ▶ default [3]
    name : null
    owner : anaderi
  ▶ environments [1]
    max_memoryMB : 1024
```

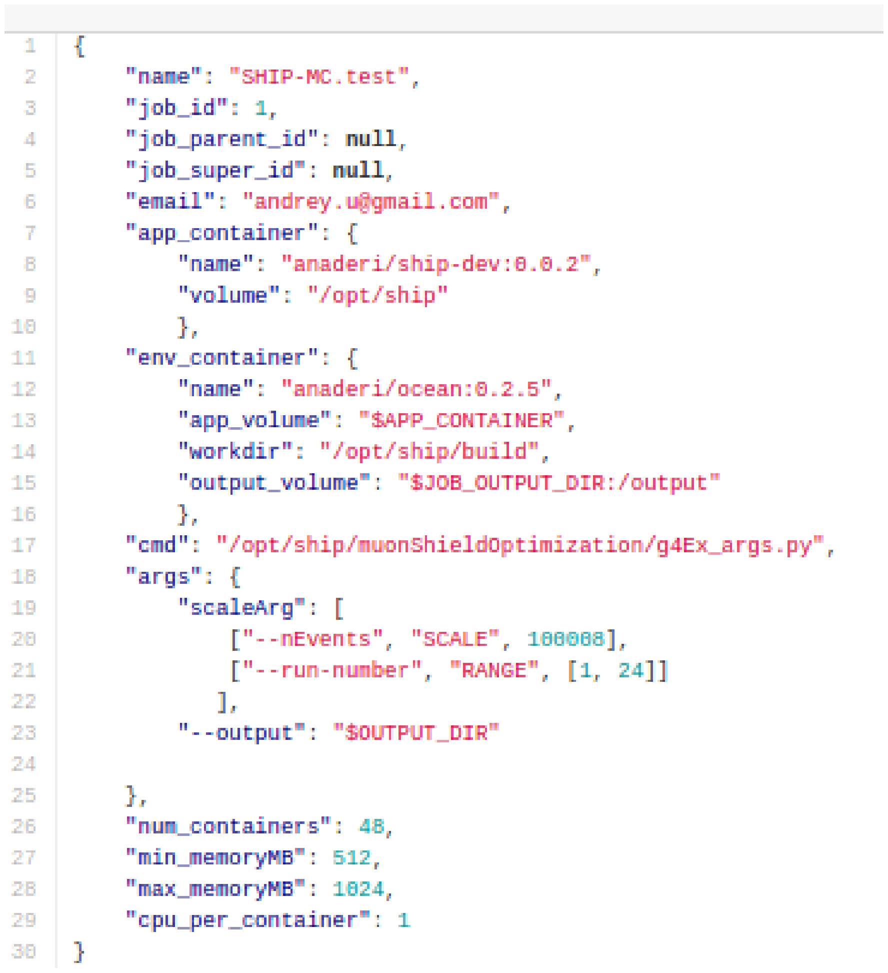
Figure 18: sub-jobdescriptor3

Finally, there are some other challenges to make the idea more intelligent so that MetaScheduler can pass, process and split different fromats of json files as input and output.

For instance, we could execute the same function for two different input files with different formats. In the next two figures we can see 2 different jobdescriptors that could be processed from the same function.

For the first format the splitter has to understand that it must split the jobs by the number of the events and the range. The optimal way to do it is to divide 100008 event by 24 so that 4167 events will take 1 of the 24 range values.

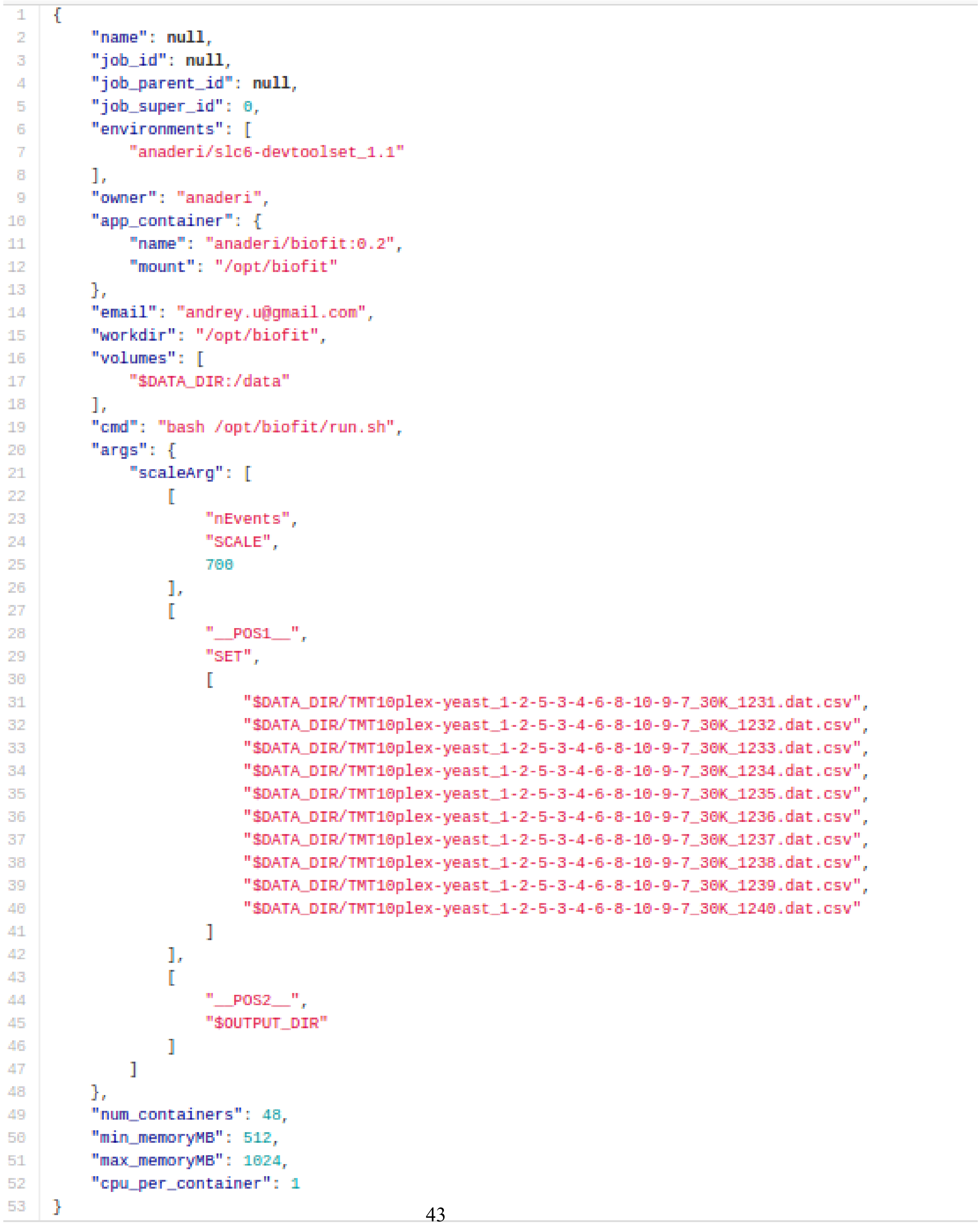
For the first format the splitter has to understand automatically that there are 70 possible sub-jobdescriptors in the optimal way of splitting the jobs. Each sub-jobdescriptor will has one event and one unique set value.



```
{
    "name": "SHIP-MC.test",
    "job_id": 1,
    "job_parent_id": null,
    "job_super_id": null,
    "email": "andrey.u@gmail.com",
    "app_container": {
        "name": "anaderi/ship-dev:0.0.2",
        "volume": "/opt/ship"
        },
    "env_container": {
        "name": "anaderi/ocean:0.2.5",
        "app_volume": "$APP_CONTAINER",
        "workdir": "/opt/ship/build",
        "output_volume": "$JOB_OUTPUT_DIR:/output"
        },
    "cmd": "/opt/ship/muonShieldOptimization/g4Ex_args.py",
    "args": {
        "scaleArg": [
            ["--nEvents", "SCALE", 100000],
            ["--run-number", "RANGE", [1, 24]]
          ],
        "--output": "$OUTPUT_DIR"

    },
    "num_containers": 48,
    "min_memoryMB": 512,
    "max_memoryMB": 1024,
    "cpu_per_container": 1
}
```

Figure 19: format 1



```json
{
    "name": null,
    "job_id": null,
    "job_parent_id": null,
    "job_super_id": 0,
    "environments": [
        "anaderi/slc6-devtoolset_1.1"
    ],
    "owner": "anaderi",
    "app_container": {
        "name": "anaderi/biofit:0.2",
        "mount": "/opt/biofit"
    },
    "email": "andrey.u@gmail.com",
    "workdir": "/opt/biofit",
    "volumes": [
        "$DATA_DIR:/data"
    ],
    "cmd": "bash /opt/biofit/run.sh",
    "args": {
        "scaleArg": [
            [
                "nEvents",
                "SCALE",
                700
            ],
            [
                "__POS1__",
                "SET",
                [
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1231.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1232.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1233.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1234.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1235.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1236.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1237.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1238.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1239.dat.csv",
                    "$DATA_DIR/TMT10plex-yeast_1-2-5-3-4-6-8-10-9-7_30K_1240.dat.csv"
                ]
            ],
            [
                "__POS2__",
                "$OUTPUT_DIR"
            ]
        ]
    },
    "num_containers": 48,
    "min_memoryMB": 512,
    "max_memoryMB": 1024,
    "cpu_per_container": 1
}
```

Figure 20: format 2

9.3 Jeeves

Then, I focused on Jeeves and I created a class Jeeves and CommandExecutor. Jeeves uses CommandExecutor class to execute FairShip software (https://github.com/ShipSoft/FairShip) and run bash and python scripts for event simulation via ocean/anaderi image inside docker's containers. More specifically, the required scripts (inside folder vm) which must be executed in the correct order in order to produce the results are:

1. start.sh
2. build_ ship.sh
3. run.sh 'cd /opt/ship/build; . ./config.sh; macro/run_ simScript.py'

When this process is finished, all the generated results are gathered to build/ship.Pythia8-TGeant4.root file.

9.3.1 Collect Files

My goal after executing Jeeves was to collect all the results and store them to HDFS with 4 different ways. More specifically I had to add support to Jeeves to send results of job via several backends. These backends are:

1. scp
2. cp (local copy)
3. hdfs (copy to hdfs location)
4. http post

In addition to that, I had to create a file as a job output descriptor and store it to meta-data database for faster search of the results.

10 Conclusion

Until now, SkyGrid was used for running jobs on cluster 40 machines (24 CPUs) for biology data processing and for MC generation for SHIP experiment. It takes about 10 days to generate 10^9 events. In this report, we have first explained some of the most important technologies and concepts that we used on this project. Then, in the second part we have described our new idea of building a prototype for the SHiP experiment at CERN with a new architecture on the grid-cloud platform using Hadoop YARN. The most important technologies that we used for this project, were Hadoop YARN and docker.

In conclusion, the most meaningful advantages that we gain with our solution are:

- Run event simulation and event processing using grid interface to cloud technologies
- Provide support for virtualization and hence, ability to run jobs in almost arbitrary environment
- Achieve quick navigation to the output of the jobs using indexes
- Evaluate the system on ShiP-specific tasks (MC production)

References

[1] ShiP experiment
http://ship.web.cern.ch/ship/

[2] Networking and Workload Management
https://twiki.cern.ch/twiki/pub/PanDA/PanDA/WMSnetworking-dec12.pdf

[3] The PanDA Production and Distributed Analysis System
https://twiki.cern.ch/twiki/bin/view/PanDA/PanDA

[4] developerWorks IBM - Hadoop YARN
http://www.ibm.com/developerworks/library/bd-yarn-intro/index.html?ca=dat-

[5] Hortonworks university - Hadoop YARN
http://hortonworks.com/hadoop/yarn/

[6] Hortonworks university - Hadoop Destributed File System (HDFS)
http://hortonworks.com/hadoop/hdfs/

[7] Hortonworks university - Apache Hive
http://hortonworks.com/hadoop/hive/

[8] Wikipedia - Cloudera
http://en.wikipedia.org/wiki/Cloudera

[9] Docker official webpage
https://www.docker.com/whatisdocker/

[10] Understanding Docker
https://docs.docker.com/introduction/understanding-docker/