MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Distribution of processing at CERN monitoring

TDK DISSERTATION

Author
Dániel Zolnai

Supervisor
Dr. Péter Ekler

October 26, 2016

Table of contents

Abstract

This dissertation will be about my work done at CERN as a technical student. As part of the monitoring team I was assigned to replace the system responsible for calculating historical and real-time statistics of the jobs submitted to do physics simulation and analysis tasks.
For this work we have chosen to use the Spark distributed processing framework, on which I have implemented three different applications using the Scala programming language. These applications were tested on internal clusters processing the same data as the current live systems, and the results were compared to those.
Furthermore my work also included configuration of virtual machines and additional software, such as Apache Flume, Apache Kafka, Mesosphere Marathon, Mesosphere Chronos, and Docker.

In the first chapter I will give a short introduction on the context of my project, and computing at CERN.
The second chapter will explain the history of running Spark jobs in our team, advancing from the very basic manual submission to the dedicated frameworks.
Chapter 3 is all about collecting the necessary input data for further processing. There will be a detailed intro to the usage of Flume to periodically fetch rows from databases, and forward events to multiple output sources.
Finally, in chapter 4, I will draw a short conclusion about the work covered in this dissertation.

Chapter 1

Introduction

In this chapter I would like to introduce the company I've spent one year at as part of an internship and also the context of the work I've done there.

1.0.1 CERN

CERN (short for *European Organization for Nuclear Research* in French) is a research organization located next to Geneva, on the Switzerland-France border. Its main goal is to host particle accelerators to experiment with high-energy physics. As of now, its main particle collider, the Large Hadron Collider (LHC)[15] is the largest of its kind in the world.
To make the functioning of this giant machine possible, more than ten thousand people work in different departments and groups day by day. The LHC has not been running for many years yet, but it has already built a reputation by being the place where proof of the existence of the Higgs Boson has been reported[11], or by hosting the first website ever[19].
The laboratory will continue to operate for many more years, to provide a platform for future physics experiments, maybe giving a way out of the Standard Model.

1.0.2 CERN computing centre and the Worldwide LHC Computing Grid (WLCG)

CERN runs today two large data centers, one in Geneva and a second in Budapest, providing over 120,000 cores, 180 PB of disk space, 100 PB of tape space, and 150 high performance tape drives to serve CERN's computing and storage needs. This capacity is extended by the Worldwide LHC Computing Grid (WLCG)[3], providing resources from different organizations around the globe, as CERN provides 20% of the total WLCG resources. WLCG is a global collaboration of more than 170 computing centers in 42 countries, linking up national and international grid and cloud infrastructures, which are divided into 3 tiers, with CERN being Tier 0 (in figure 1.1 one can see the Tier 1 sites as well). At any given time, there are typically more than 300,000 concurrent computational jobs running and, on a daily basis, more than two million computational jobs are submitted and multiple terabytes of data are transferred between sites. The Monitoring team of the IT Compute and Monitoring (IT-CM) group is responsible for providing monitoring solutions for these resources.

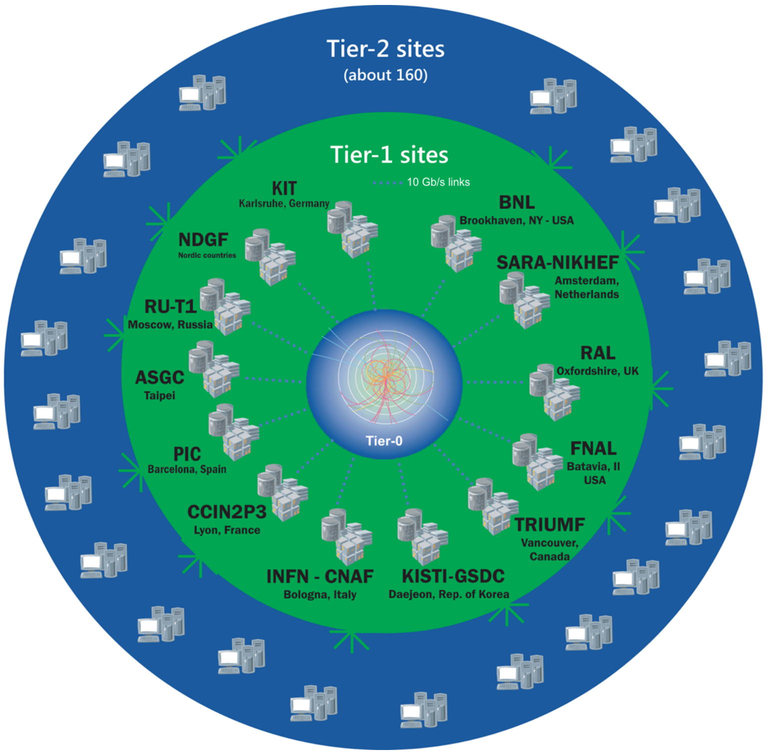


Figure 1.1. The tiered subdivision of the WLCG sites.

1.0.3 ATLAS

ATLAS is a particle physics experiment at the Large Hadron Collider at CERN that is searching for new discoveries in the head-on collisions of protons of extraordinarily high energy.[2]
Two general purpose detectors, *ATLAS (A Toroidal LHC ApparatuS)* and *CMS (Compact Muon Solenoid)* have been built for probing p-p and A-A collisions. The ATLAS detector represents the work of a large collaboration of several thousand physicists, engineers, technicians, and students over a period of fifteen years of dedicated design, development, fabrication, and installation.[12]

1.0.4 PanDA

The Production and Distributed Analysis (PanDA) system has been developed to meet ATLAS production and analysis requirements for a data-driven workload management sys-

tem capable of operating at LHC data processing scale. ATLAS production and analysis place challenging requirements on throughput, scalability, robustness, minimal operational manpower, and efficient integrated data/processing management. PanDA was originally developed for ATLAS and is being extended as a generic high level workload manager usable by non-ATLAS OSG activities as well.[14]
Heterogeneous resources are distributed worldwide at hundreds of sites, thousands of physicists analyze the data remotely, the volume of processed data is beyond the exabyte scale, while data processing requires more than a few billion hours of computing usage per year. The PanDA system was developed to meet the scale and complexity of LHC distributed computing for the ATLAS experiment.[13] ATLAS fully relies on PanDA to handle the experiment's analysis and production workloads, and to date PanDA has successfully delivered without showing scalability issues.[10]

1.1 Monitoring at CERN and WLCG

The Monitoring team is responsible for checking the availability and status of each (virtual) machine in the data centers of CERN and providing monitoring solutions to the LHC experiments that cover all their computing activities in the Worldwide LHC Computing Grid (WLCG), this includes their data distribution and access, computational job processing and availability of their computing sites and services. Not only does it provide software and tools for doing this, it also has to supply dashboards and other interfaces for viewing and processing real-time and historical data. As part of the WLCG operations there are also dashboards maintained where the performance and status of the computing sites or the whole grid altogether can be checked upon. The monitoring team provides generic solutions for a wide variety of hardware that can be used to monitor the nodes of any computer centre and it is not coupled to the CERN data centre itself.

1.1.1 Job Monitoring in WLCG

The highly distributed infrastructure of WLCG is also heterogeneous with multiple middleware flavours, job submission and execution frameworks, and several methods of transporting and accessing the data. The large scale activity and heterogeneity of the WLCG increases the level of complexity of the system and, thus, increases the probability of failures or inefficiencies arising. Effective monitoring is essential for providing a comprehensive way to identify and address any issues within such a highly distributed and heterogeneous infrastructure. It is also a key factor in the effective utilization of the system.[9]

1.1.2 New Monitoring team

Before January 2016, there were two teams working actively on monitoring: the Agile Infrastructure Monitoring team was responsible for the nodes in the CERN Data Centre, and the WLCG Monitoring team for the WLCG infrastructure. In January, upper management decided to merge these two teams, since their operation is similar, and the combined knowledge could be used to provide better solutions. The merged team became the IT Compute and Monitoring group, and was also given the task to iteratively replace older technologies. My work was also part of this replacement project.

Chapter 2

Problem statement

This chapter will be about the original problem and the abstract overview of the proposed solution. The next chapters will give a detailed walkthrough on this solution.

2.1 Introduction of the problem

When the new monitoring team was created by merging two similar teams together, it was given a task: create a new architecture, which would be able to handle data coming from all monitoring activities (WLCG and data center monitoring as well). Learning from the mistakes and experiences the teams have made during the past years with monitoring, the aim was to create an architecture which:

- Is scalable: we knew that the incoming data will continue to grow in the future. We wanted a solution which works in a distributed way, and is not limited to a certain throughput theoretically.
- Has a reduced single point of failure: We knew from experience, that things can go wrong at a lot of places. But most of the times those failures could have been avoided if the service or interface which failed had a fallback. Our goal was to even handle a complete data center shutdown, since we had multiple data centers available where we could deploy virtual machines (in Geneva and Budapest).
- Is maintainable: A frequently occurring problem with our old software was that they were hard to maintain. There were multiple Perl scripts, C++ projects, highly optimized Oracle databases which were hard to work on. Also there were in-house-built software, which could have been replaced with open-source variants, giving us an opportunity to help the open-source community, and also taking some of the maintenance work off our backs.
- Is extensible: We wanted to have an architecture, which would allow other teams to easily interact with it, and even allow users some freedom. For example, a common

request was to create specialized dashboards for users, but if they could do this themselves, that would save time and effort.

The design, implementation and deployment of this new architecture was the work I have been doing while I was a technical student at CERN.

2.1.1 Tasks

A lot of this work was a team effort. But I have been given four main tasks related to this work, which I have completed during my time at CERN. These four tasks were:

1. *Data retrieval and ingestion:* Collecting all the necessary input data from several sources, forwarding them to different storage systems, configuration of Apache Flume.
2. *Real-time enrichment:* Enhancing data taken from the official PanDA database with relevant information fetched from other external sources.
3. *Accounting and validation:* Streaming the output of the previous task, and storing the current state in a global map. Taking and storing snapshots of this global state at regular intervals.
4. *Aggregations and statistics:* Taking the enhanced data and the snapshots, aggregating them together, computing statistics which can be visualized to the end users in the form of dashboards.

Only the first task will be covered in this dissertation. The 3 latter ones are each a Spark job I was able to write and run based on the completed work of the first task. Although these jobs will not be covered, I will describe how we were able to run the Spark framework in the new stack, and how it ties together with the other components.

2.2 Proposed solution

We chose open-source, well-known software for each component. The data ingestion part will be done by deploying Flume agents, fetching the data constantly, and forwarding it to ElasticSearch and HDFS through Kafka, a buffer with a retention period of a few hours.
The other three tasks will all be performed in separate Spark applications. For streaming analytics, Spark will get the data from Kafka. For requesting information older than the previous few hours, HDFS can be used, which also supports parallel access, allowing us to increase performance by simply adding more executors.
Finally, Spark will write its results to Kafka. From there, the messages might be sent to ElasticSearch, depending on the type. Every output will be archived on HDFS.
The results stored in ElasticSearch can also be visualized using Kibana, a platform for

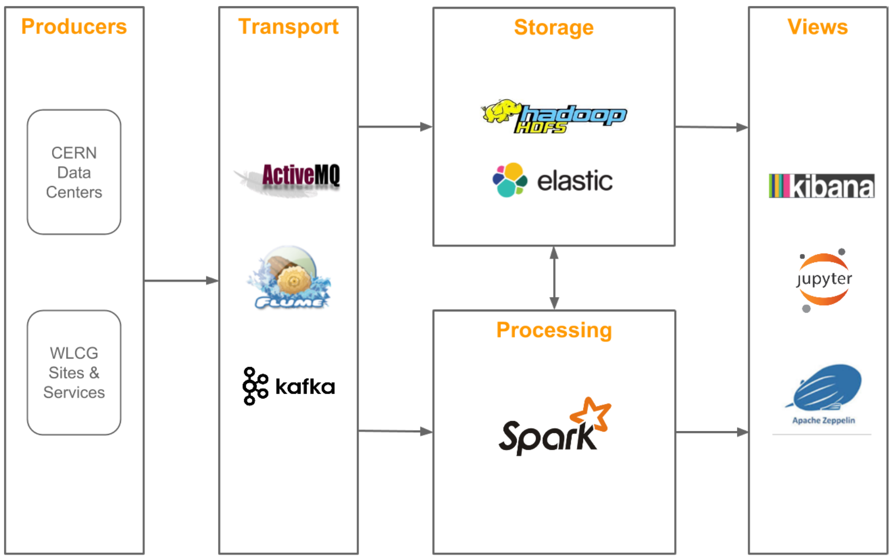


Figure 2.1. The Monitoring architecture.

creating plots and dashboards accessible for the public.
This setup can be seen in figure 2.1. Note that this architecture is not only used for this purpose, but for all the activities in our team. That is why there are some additional technologies presented in the chart.
The overview of the part where we do job monitoring with Spark is described in figure 2.2. The arrows indicate the flow of data, and the grey boxes are the 3 Spark jobs I will be implementing as part of my work.

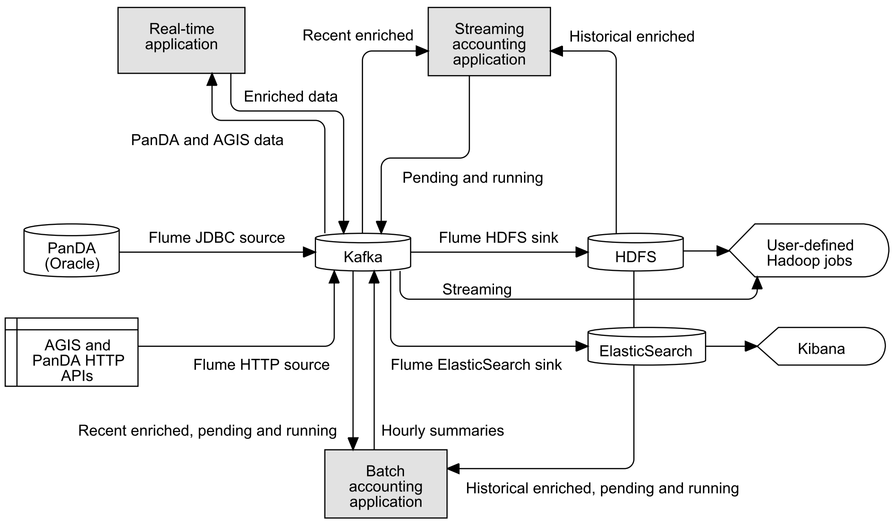


Figure 2.2. Job monitoring data flow diagram.

Chapter 3

Running Spark jobs

In this chapter I will detail the history of how we started using Spark, and how the need for building, submission, and monitoring frameworks were developed in our team. I will introduce these software briefly, and also describe how we had to configure them to cooperate with each other and cope with our requirements.

3.1 The evolution of our executor frameworks

This section will the demand for running Spark applications in our production environment evolved, with high priority being on stability, and for one-click submission.

3.1.1 Running manually

When I came to CERN in September 2015, the team was only running 2 Spark jobs, both once a month. But rather soon, the number of jobs grew. Submitting those two jobs once a month was possible by having the sources of them available on one specific machine, and then adding a cron job which schedules them. Quite soon, we had some Spark Streaming jobs ready for deployment. Of course they ended up in the same machine, but no one knew if they were running fine. They could die and one would notice it only a week thereafter. The same applied to the non-streaming executors that were started by cron. We had no way to know if they were running fine, unless we checked the output data, or the logs of the jobs. Also there would have been loss of data if the streaming job had been terminated.

3.1.2 The Python script

We knew the manual way was not feasible for the long term, so we were in search for something that would schedule the jobs for us and send notifications if there would be any problems. However, we did not find such a thing, so we decided to develop our own software for handling these.

First, we moved all our sources into a git repository, without the built artifacts. At the time we only had Python jobs, which do not have to be compiled into a jar, that simplified things. The first version was a very basic python script, which did the following:

1. Clone the git repository containing the sources every 2 hours.
2. Check if there are new versions available, if yes, recompile the sources, and restart the streaming jobs.
3. Also check for dead streaming jobs. Restart them if there were any.
4. If there are any new jobs, submit them, or kill the ones which are not in the repository any more.
5. Add and remove cron jobs for the scheduled jobs.

As one can see from the list, our first Python script already supported compiling of sources (both SBT and Maven), in case we had Java or Scala jobs to submit.

3.1.3 The spark-job-runner package

The Python script was clearly an improvement compared to the manual way, but it still had some flaws:

- No support for High Availability *(HA)*: If the machine running the script would be having a failure, all jobs would not be running.
- Still no email notifications, if a job dies too frequently: If a streaming job were misconfigurated, it would just be restarted every 2 hours, but it would not produce any actual output.
- The Python script was distributed as a Puppet resource, which is not what Puppet should be used for.

Together with some highly needed bug fixing, I addressed the aforementioned issues in the next iteration of our script. First I converted it to a quite simple RPM package, called the *spark-job-runner*, which just installs our script as a Python module. I also added a way to store the timestamps when a job was restarted. A cron job would run every morning, and send an email report about which jobs had failed the past day. Meanwhile my colleague had just created a small Python module called *master-slave*. It is used to be installed on multiple machines, which would then act in a master-slave hierarchy: of all the machines which are registered to the same path, only one can be the master. If the health checks fail on the machine, or it just does not send any heartbeats, a new slave will be elected as master. The module was aimed to be multipurpose, so it could be reused for different situations. One of these situations was the spark-job-runner. We put multiple machines inside a hostgroup, deployed the master-slave packaged, and only the master was allowed the execute the job runner workflow every two hours.

3.1.4 Problems with the spark-job-runner

The spark-job-runner had been running for quite some time, but it still had problems. To be able to monitor each job, we would submit them using the client deploy, which means that the driver process would stay on the local machine, giving us the ability to monitor the job through its lifetime, and also terminate it easily by just killing the driver process. However, these driver processes consumed a lot of memory, and there was the possibility to not have enough memory on the runner machine to be able to execute all jobs (even with extra large VMs). Also the monitoring consolidation happened at that time, which meant that we had to find a way to run the WLCG and IT monitoring Spark jobs in the same way.

3.1.5 Evaluation of Mesos Chronos and Mesos Marathon

After the migration, one of the first goals was to find a new way to submit Spark jobs. This was required because development began on multiple data processing applications, and by the time those are ready to be deployed, we should have something to submit them to.
Initially, we browsed for alternatives. We were searching for something which should be able to run both streaming and batch jobs. We found *Mesos Marathon* and *Mesos Chronos*. Both of these take a command, which is executed on one of the predefined machines. Marathon continually monitors the command, and expects it to be running forever. If the command exits, Marathon automatically redeploys it on a different executor. Chronos does not monitor the command, but instead has an interface for defining fine-grained execution times for tasks. In figure 3.1 you can see an example architecture, which would benefit from using Chronos. Since it is running in a Mesos container, it can be deployed remotely, while making use of databases and execution frameworks such as MapReduce, Hive, Pig, or even Spark.

The two frameworks perfectly fitted our requirements, but had one limitation: they require the commands and files to be present on the current executor. However, they both support running Docker containers. Here we came up with the idea that we should bundle our jobs inside Docker images.
The reasoning for it was that jobs would be perfectly abstracted from each other. Deployment of the jars and any additional resources would be done with the image. We can use the Docker registry and its tags for having different versions of an application available.
I did the first tests, where I deployed Marathon and deployed the Chronos Docker image inside it. Installing Marathon is as easy as registering the Mesos yumrepo and installing the marathon package directly. Marathon can be started as a service. Marathon requires the Java 8 SDK to be present, and since CentOS 7 comes with Java 7 by default, we had to install a newer version explicitly, and set it to be the default one using *alternatives*.

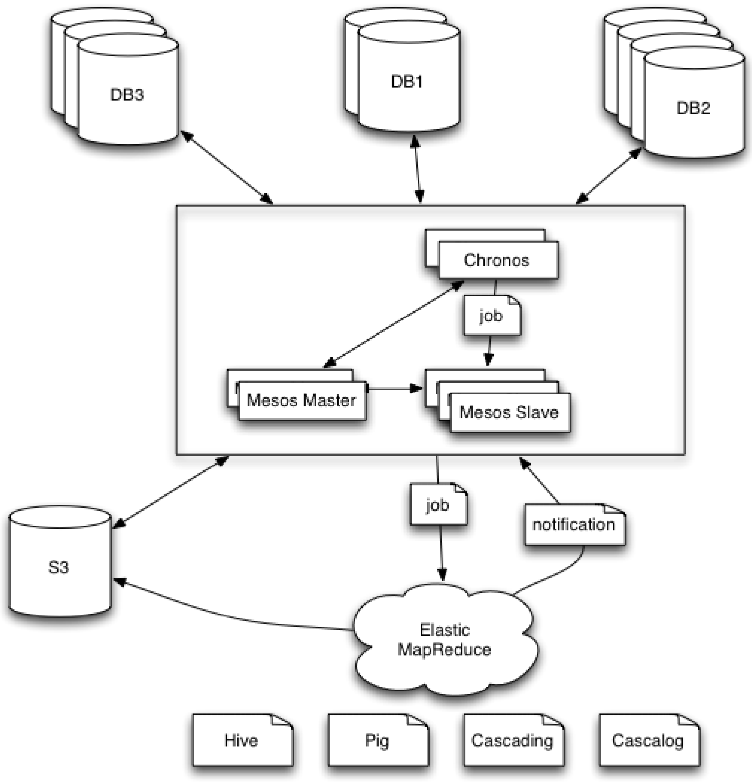


Figure 3.1. Chronos sample architecture[18].

When I instructed Marathon to start the Docker image of Chronos on any of the Mesos slaves, it would complain that none of the workers were able to execute Docker containers. Of course the slaves require the Docker daemon to be running, but that seemed to work fine locally. The solution was to put the text `"docker,mesos"` in the file `/etc/mesos/slave/executors`. This would tell the Mesos slave service that it is able to execute both Mesos and Docker containers. By default, it only accepts Mesos containers.

The next problem I faced was more complicated. Whenever I restarted Docker, and ran the Chronos image, it would work fine for around 30 minutes. After that, connection to the port would be refused, and it would only work again if the Docker service was restarted. The root cause turned to be Puppet. Whenever a Puppet run happens (by default, each hour), the iptables module clears all unknown rules from the INPUT chain. This chain contained rules inserted by Docker, which rules would forward packets to the DOCKER chain in some cases. The solution to this problem was defining the exact same forward rules in the Puppet manifest, so they would not get purged. Finally, there were some simple configuration tweaks required, as inserting our Zookeeper URL into the Marathon configuration files, and increasing the Marathon task timeouts so that an extra-long Docker pull would not time out before it could start the container. The resulting Puppet manifest for the Spark masters can be seen in the appendix, section A.1.

3.2 Selecting our build tool

When the two teams merged, we were using different ways to create Spark jobs. We were using Python and Scala for writing applications, and *Maven* or *SBT* for building. While all of these work fine, it was time to settle on one language and build tool. Scala was quickly selected for the programming language, since the Java and Python API for Spark and often quite behind, and have limited support compared to Scala. Also writing a data processing application in Python for Spark is only preferred if its code can fit on a sheet of paper. Else one is better off with Scala.
Selecting the build tool was not that easy. We decided to make a comparison between all available options, and have a vote after the results are in. The three competitors were Maven, SBT and Gradle.
The points of comparison included: build speed, shading and debugger support, parent project support, and community usage.
It is hard to find a Scala project which is built with Gradle (probably because it is a latecomer), but it surprised us how fast is was. Building an assembly jar was twice as fast as SBT, which finished second. Maven was the only one which did not support iterative compilation, which means that it did a full build even if only one source file changed.
All three supported parent projects and shading. Debugger support was perfect for SBT and Gradle, but Maven was cumbersome in this aspect. The JVM args for an application run could not be injected from the process arguments, but had to be explicitly declared in the `pom.xml`. IDE integration was also limited: *IntelliJ IDEA* was missing the Scala SDK, since that was added as a Maven dependency, instead of providing the one available on the local machine.
Additionally, I did a subjective comparison on how simple the build files are: Gradle won this by being the shortest and easiest to read.
In table 3.1 I listed the points of comparison and the results. Since SBT is also one of the most popular build tool in open-source Spark projects, and performed quite well in our tests, we settled on that.

Point of comparison	Gradle	Maven	SBT
Speed	Fastest	Slow	Fast
Incremental build	Supported	NOT supported	Supported
Build file complexity	Simple	Complicated	Complicated
Shading plugin	shadow	maven-shade-plugin	sbt-assembly
Debugging support	Flawless	JVM args not set	Flawless
Community	Least used	Not updated anymore	Thriving community
Parent project support	Submodules	Parent pom files	Multi-project builds

Table 3.1. Comparison table of Gradle, Maven, and SBT

3.3 Running applications in containers

In this section I would like to give a short introduction on how we managed to package a Spark job, ready for submission. First, let us list the files necessary for a job:

1. The Python script (if written in Python), or jar file (if written in Java or Scala) containing the code written by us.
2. A keytab file for authenticating as the operations account, giving the job access to the Kerberos-secured HDFS and YARN cluster.
3. Docker credentials, allowing the worker access to the private Docker registry.
4. Any additional configuration or supplementary files required by the job itself.

A major issue is that these files have to be available on every worker - since we do not know on which one the job will be started (Marathon and Chronos select one based on the offered resources by Mesos). For one job these might be easy to do, but when one has 10 different jobs to manage on the cluster, this can easily be a hard task: keeping files in sync between machines and job versions can be quite a hassle. One would not also send the keytab and Docker credentials around in the cluster or put them somewhere on the web. And where would the jars or python files be stored?
A lot of these open problems and questions got solved by packaging our jobs in Docker images. It has some minor drawbacks, but its advantages proved to be well-suited for us:

- We can package the job and its supplementary files inside the container. The files of different jobs are clearly separated.
- The keytab and Docker credentials are distributed securely via Puppet, and are mounted by the Docker job.
- Jobs are packaged as-is. We can add additional resources and even RPM packages just for that specific job without bloating the configuration of all the workers.
- Since we use `docker run`, the logs of the application will be available as Docker logs, and can be viewed very easily using `docker logs`. These logs are also stored for exited containers, allowing us to see previous failure causes.

The main drawbacks we see, are:

- An extra layer of abstraction, which makes the jobs a bit less performant. This is not effecting us entirely, since only the driver executor is running in Docker, the other executors are running on the YARN or Mesos cluster, without the extra abstraction.

- Large image sizes can be a problem when distributing an image. Since it does not only contain the job and its files, but also a full fledged operation system, it can be several hundreds of megabytes large. However, since we have a private registry in our own data center, downloading and uploading images this big is still just a few seconds.

- We have to make sure that Docker is configured and running properly. This was mostly a burden in the beginning. For example, we have found a rare case where our job would hang when initiating any network connection, but just on some of the workers. The root cause of this was that those workers had IPv6 addresses for the DNS servers, and Docker was not prepared for that. Using a newer Docker version downloaded from their official yum repos solved this problem (the CentOS repos are months behind in the versions).

3.3.1 Building the Docker image

Our original idea for the base image was to start from a *CERN CentOS 7* image. This was required because it contained the Kerberos configuration to connect to the correct realms. This is not the optimal solution, because the image is quite big compared to an *Arch Linux* image with Java, but it was the perfect image for creating our first images. The initial working Dockerfile was this simple:

```
FROM docker.cern.ch/linuxsupport/cc7-base
MAINTAINER it-dep-cm-mm-monitoring-all@cern.ch
RUN yum -y install java krb5-workstation.x86_64
COPY target/spark-ddm-*-jar-with-dependencies.jar /monit/spark-ddm/
```

When the image was built, it was uploaded to the CERN Docker registry, under the team organization (each Spark application has a different project in the organization). After running jobs have been built with this Dockerfile for some months, we decided to optimize it a bit. Its main problems were:

1. The image was very big (almost 1 Gb), because the official CERN base image was 700 Mb.
2. Mounting the Mesos library was not possible, because it had some `.so` files in the `\usr\lib\` directory.
3. It did not have `python-pip`, which might be necessary for downloading some Python dependencies if we are talking about a Python job. Of course these would only work on the driver node, but that is fine for the preparation phase, or for jobs running only in local mode.

4. Also we wanted to have a base image, which all of our Spark jobs can use, and have fixes for all issues listed above.

A second version of the Dockerfile provided a solution for all of these problems:

```
FROM centos:7.2.1511
MAINTAINER it-dep-cm-mm-monitoring-all@cern.ch
RUN rpm -iUh
↪  http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-7.noarch.rpm
&& rpm -Uh
↪  http://repos.mesosphere.com/el/7/noarch/RPMS/mesosphere-el-repo-7-2.noarch.rpm
&& yum -y install java-1.7.0-openjdk krb5-workstation.x86_64 python-pip
↪  python-devel mesos
&& yum clean all
COPY krb5.conf /etc/
```

As one can see, we have installed `python-pip` and the Mesos package as well. By using the original CentOS 7 image as the base, we shaved off a few megabytes (it takes up 562 Mb), while still adding features such as these new packages. The CERN specific Kerberos config has to be explicitly provided though, since the defaults will not do. This image can be used as a base image for any of our other images.
By building our images with the base image as the foundation, some core components as Java and Mesos will be available inside the container. Others, which are configured via Puppet (Spark and its Hadoop dependencies for example) are made available to the container by mapping those directories to inside the images (using volumes). An overview of how a job is running in a slave and which components are located where can be seen in figure 3.2.

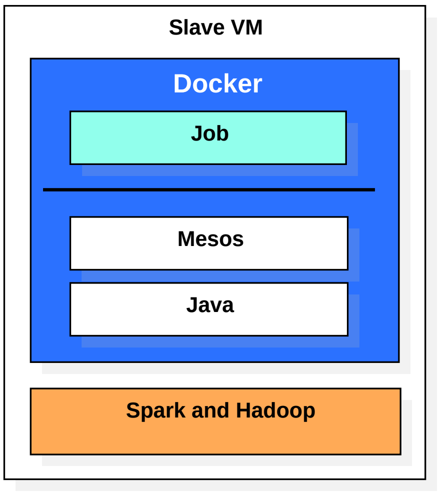


Figure 3.2. Stack overview of a job deployed in a slave

A job image has to be run with some important arguments:

```
docker run --net=host -t -i \
-v /etc/hadoop/conf:/etc/hadoop/conf/ \
-v /usr/lib/spark:/usr/lib/spark \
docker.cern.ch/monitoring-service/spark-ddm:test \
/bin/bash -c "kinit -k -t /etc/hadoop/conf/monitops.keytab
↪  monitops@CERN.CH;/usr/lib/spark/bin/spark-submit --class
↪  ch.cern.monitoring.BatchJob --master yarn-client
↪  /monit/spark-ddm/spark-ddm-0.0.1-SNAPSHOT-jar-with-dependencies.jar
↪  --job-argument value"
```

When the image has succeeded with building, we have to upload it to the CERN private Docker registry. Authentication is handled via our LDAP credentials.

3.3.2 Running the built image

We have decided to use our Mesos worker machines for running the Docker images. After numerous small fixes and fine-tuning we deployed the Puppet configuration in production, which can be viewed in the appendix, section A.2.

As one can see, Chronos is deployed as a service in each worker, instead of launching it as an application from Marathon. This was changed for the reasons below:

1. Running it as a Docker image from Marathon would result in a Docker image running in a Docker image, which could lead to configuration issues.
2. When submitted from Marathon, it is running on one random machine. Only this one machine can submit any task, which could be a bottleneck if there are multiple Chronos submissions running in parallel.
3. If the worker machine running Chronos becomes unresponsive, there us a window of a few minutes where Marathon has to recognize this and redeploy it on a different machine. During this timeframe, a job submission might be missed. When running as a service, the services on each worker communicate with each other via Zookeeper.

After the evaluation, a recurrent question was how we could supply secrets and passwords to the Docker images, without exposing them in Puppet or inside Marathon/Chronos.
In Puppet, we have a module called *teigi*, which securely transports secrets and passwords into files on the VM from an external store.
A secret is first added using the *tbag* command-line utility as a key-value pair. When creating a new secret, one has to define a host or hostgroup to attach it to. It will only be visible for that domain, and it checks in Active Directory if the current Kerberos credentials have access to that domain. We define all our secrets for the `monitoring` hostgroup.
When Puppet runs, the teigi module will access this external store with the key defined in the manifest, and will put its value in a file or fill its placeholder in a template.

We gave the instruction that these secrets should be in separate files such as `/etc/monit/monit_es` or `/etc/monit/monitops.keytab`. The command running inside Docker container has to read these passwords. To solve this, we mount the `/etc/monit` to each container, so it will be visible as a directory there, and either read the files inside the application, or create parameters for it:

```
docker run ... spark-submit application.jar --es-password $(cat
 ↪  /etc/monit/monit_es)
```

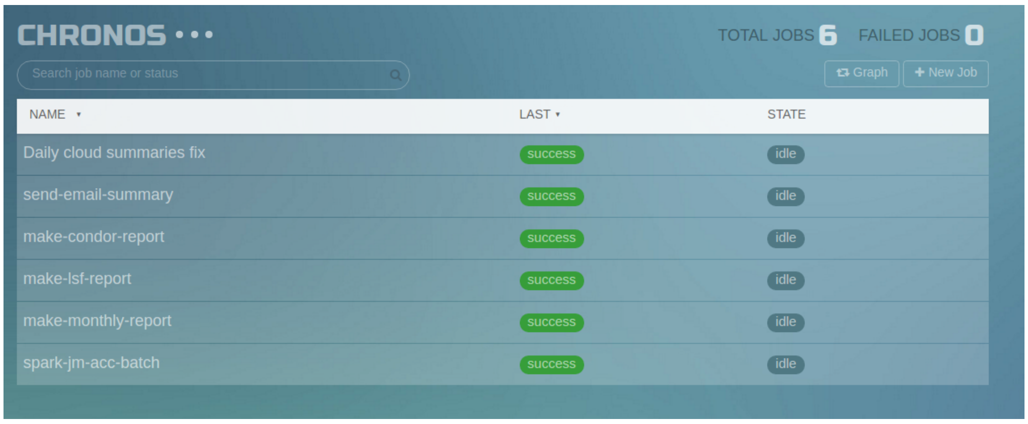
The file `/etc/monit/docker.tar.gz` is the compressed version of the `/.docker` directory after a login with our operations user account to our private Docker registry.
This file is provided to Marathon as a supplementary resource for each job which has to pull an image from our registry. It will recognize the file as the authentication keys for our registry and will use it for authenticating against the private Docker registry.[20]

3.4 Scheduling

The two scheduling types - batch and streaming, are very different. The former has to be submitted at specific times, the latter has to be always running. We solved this by running the two types in two different submission frameworks.

3.4.1 Batch jobs in Chronos

We defined *batch jobs* as applications which do not run for an infinite time, so they are expected to finish sooner or later. Take for an example a data aggregation crawler, which launches when a new day starts at midnight, and takes all files from the last day, and compresses them. This way writing and reading those will be a grade slower, but they will take up a lot less space on our storage systems.
We would just like to run this crawler once a day, exactly at midnight. It is not a viable option to have someone there each night submit the job to the cluster. We could use a cron job, but what if the VM becomes inaccessible while running the job, and it fails?
As I already described, we decided to use Mesos Chronos to do this. It provides a solution for all these problems I mentioned.
Configuring a job in Chronos is a bit more challenging then it initially seems. Although it is possible to create new entries from the web UI, it does not expose all fields (such as Docker image, and other parameters, which are used infrequently), and it has a known bug that changing properties via the web interface can not be trusted[6].
Instead, the REST API has to be used. For an example payload of creating a new application in Chronos, see the JSON body in appendix A.3.
The latest status of the jobs can be inspected on the web interface, of which an example of can be seen in figure 3.3.



NAME	LAST	STATE
Daily cloud summaries fix	success	idle
send-email-summary	success	idle
make-condor-report	success	idle
make-lsf-report	success	idle
make-monthly-report	success	idle
spark-jm-acc-batch	success	idle

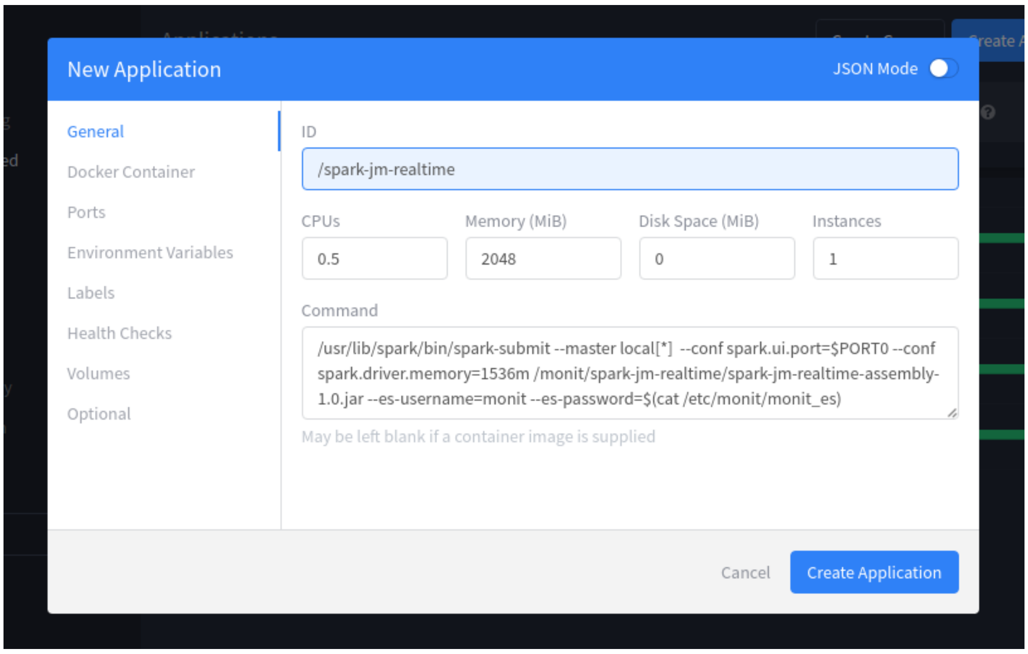
Figure 3.3. Chronos overview of defined jobs.

It is also possible to define dependencies between different jobs. E.g. to state the order of running if two jobs would be submitted at the same times.
One may also notice on the screenshot, that we already have 6 entries in Chronos. This is because we made this service available for other groups inside CERN, so they can run their jobs on our Spark cluster.

3.4.2 Streaming jobs in Marathon

Streaming jobs are expected to run forever. If they would crash for any reason, they should be immediately restarted, and they are required to continue their work where they left off, meaning that the crash should not result in any data loss optimally (delays in the output might happen though).
Our framework for running such jobs is *Mesos Marathon*. Its simple, yet elegant user-interface, together with the plentiful options to schedule a job to the Mesos cluster made us feel confident that Marathon was the right choice for running streaming jobs.
Compared to Chronos, the configuration of a new job is very similar in the terms of fields. However, this can be also done on the web interface, which is quite easy to use. In figure 3.4 one can see how the dialog looks like. On the left menu the switch between the different configuration sections can be done, most of them being optional to fill.

A nice feature of Marathon is the ability to do health checks on applications. This means that at specified intervals, a given command or HTTP request is ran in the Docker container. If the result is different from expected, it will mark the application as unhealthy, and will restart the job.



ID: /spark-jm-realtime

CPUs: 0.5
Memory (MiB): 2048
Disk Space (MiB): 0
Instances: 1

Command:

```
/usr/lib/spark/bin/spark-submit --master local[*]  --conf spark.ui.port=$PORT0 --conf spark.driver.memory=1536m /monit/spark-jm-realtime/spark-jm-realtime-assembly-1.0.jar --es-username=monit --es-password=$(cat /etc/monit/monit_es)
```

May be left blank if a container image is supplied

Figure 3.4. Create application dialog in Marathon.

Chapter 4

Data retrieval and ingestion

This chapter will be about the configuration of Apache Flume, a software which specializes in reading and writing data in a distributed and reliable way. This chapter will go into quite much detail how we achieved to get data flowing between the sources and the targets.

4.1 Objective

The job monitoring aggregation and processing stack is divided in multiple steps. The PanDA database is a highly optimized Oracle database, which is the primary information source of any physics simulation and analysis task to be run within the ATLAS experiment collaboration.
The following external data sources are used for supplying metadata for each entry:

- ATLAS Grid Information System (AGIS) - PanDA queues information[8]
- AGIS - Topology information
- PanDA error codes, statuses and types database available in our ElasticSearch cluster (static, updated manually only)
- Prodsys II database, containing extra data about the specific simulation tasks

4.2 Input sources

In this section I will describe the input sources from the ATLAS Job Monitoring perspective: where they are reading from, in what context the data is relevant to us, and how and where it will end up in our architecture in terms of storage.

4.2.1 PanDA job database

As a start for the conversion to the new architecture, we have to read from the PanDA database. Since all our data collectors and transfer points use Flume to do this, we were looking for a Flume based solution, to keep the collectors on the same base. Thus we were looking for a solution that would make it possible to read from an Oracle database, and create flume events from each row. What is really important, that this type of reading has to be continuous in the sense that it reads in small batches, but only the rows which have changed since the last batch. Meanwhile the CERN IT Database team was adding the finishing touches to their flume JDBC reader, which was exactly developed for the purpose we were looking for. They were also happy that someone aside from them would be testing their software, so we were free to use. It is also available to the public in their GitHub repository[5].

4.2.2 Configuring the Flume agent for reading from PanDA

In this part of the chapter I will explain how we configured the JDBC reader Flume plugin to read from Panda. Although it will contain quite a lot of code, I recommend the reader to read the explanations, because it has some important explanations about how Flume and the reader works. In figure 4.1 one can see how this component is connected to the other parts of the architecture.

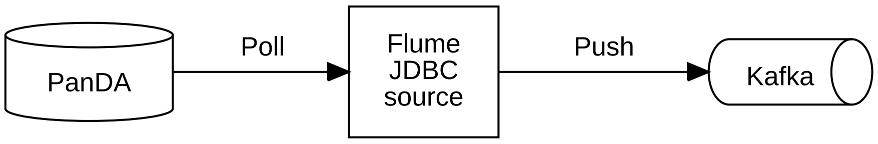


Figure 4.1. Flume JDBC source data flow direction.

First we did some tests by deploying flume-ng-audit-db in a VM, and just trying to understand how it works by using it. Here is a simplified version of our final configuration (with some modified fields such as usernames and URLs for security reasons):

```
reader.query.path = /etc/flume/agent-jdbc/panda.query
reader.connectionUrl = jdbc:oracle:thin:@redacted-host:10000:servicename
reader.username = USERNAME
reader.password = %TEIGI__monit_pandasrc_db_pass__%
batch.size = 40000
batch.minimumTime = 180000
reader.table.columnToCommit = modificationtime
reader.committingFile = /etc/flume/agent-jdbc/query.status
reader.scaleAwareNumeric = true
reader.expandBigFloats = true
```

```
duplicatedEventsProcessor = true
duplicatedEventsProcessor.size = 100000
duplicatedEventsProcessor.header = false
duplicatedEventsProcessor.body = true
duplicatedEventsProcessor.path = /etc/flume-ng/agent-jdbc/last_event.log
interceptors = iproducer iversion itype itypeprefix itimestamp
interceptors.itimestamp.type = timestamp
interceptors.iproducer.type = static
interceptors.iproducer.key = producer
interceptors.iproducer.value = panda
interceptors.itypeprefix.type = static
interceptors.itypeprefix.key = type_prefix
interceptors.itypeprefix.value = raw
interceptors.itype.type = static
interceptors.itype.key = type
interceptors.itype.value = all
interceptors.iversion.type = static
interceptors.iversion.key = version
interceptors.iversion.value = 1
```

I will explain this configuration line-by-line.

```
reader.query.path = /etc/flume-ng/agent-jdbc/panda.query
```

This is an important line in the configuration. This is the path to a file which contains the query which will be executed at the beginning of each batch to gather all the latest data from PanDA. In the next section I will show this query and describe it in detail. The most important thing to know about it, is that it has a **`WHERE`** filter with a template value which is filled by the JDBC reader automatically.

```
reader.connectionUrl = jdbc:oracle:thin:@redacted-host:10000:servicename
```

This is a JDBC URL to the host and port where the database can be accessed. It utilizes the Oracle Thin JDBC client, which should be available on the Java classpath.

```
reader.username = USERNAME
reader.password = %TEIGI__monit_pandasrc_db_pass__%
```

The username - password combination will authenticate the reader to the database. The password is not the password itself, but a special string, which is recognized by the *Teigi* puppet module, and refers to the key `monit_pandasrc_db_pass`. In the actual configuration file, this will be replaced by the password defined under that key in *tbag*[1]. Teigi makes sure that the password is securely transported to the VM where the configuration file is created, and also ensures that only the authorized hosts can use this key (this is done by having attached the key to a specific hostgroup).

```
batch.size = 40000
```

This determines the maximum number of events to be read from the database in one batch. So if the query would return more than 40,000 rows at the beginning of the batch, only the `TOP @BatchSize` would be actually collected by Flume. The rest, which is not collected, will be collected in the next batch - of course the same thing can happen there again. By setting a batch size, we add throttling to Flume, but this comes with the drawback of having increasing delays. We selected this value to be around 2-3 times the count of the usual amount of rows we get.

```
batch.minimumTime = 180000
```

The time in milliseconds to wait between batches. We configured this to be 3 minutes (as it is with the old collector). This interval is small enough to catch the important state changes, but not too frequent, as it could create too high a load on Oracle, if retrieved too often.

```
reader.table.columnToCommit = modificationtime
```

As explained before, the agent reads the rows that have been changed since the latest batch. For this, the agent has to detect which rows are changed and which were there before. This is done by having a column in the table, which is updated when a row is modified. In the panda database, this column is the `modificationtime`. If any property of a job changes, the timestamp on this column is also updated. The agent has a special value under the alias `:committed_value`. This stores the timestamp when the last batch ran. So a really simple SQL query to just get everything from the database would be similar to this one:

```sql
SELECT * FROM panda WHERE :committed_value < modificationtime
```

This collects every row whose `modificationtime` was after the last batch. In practice, the query is a lot more complicated compared to this, but the idea is exactly the same.

```
reader.committingFile = /etc/flume-ng/agent-jdbc/query.status
```

This is the path to the file which stores the latest position of the agent (that is `:committed_value`). Before the agent is started the first time, it is recommended to be populated with a beginning value. We set this to be to the current machine time minus

2 minutes. I noticed that if the time is in the future, the agent did not want to start up, even not hours after the specified timestamp, so we subtract those 2 minutes for safety. The first time it is us who set the timestamp in this file, but after each run the agent will update it with the latest `:committed_value`. One additional thing: when there are more rows returned than the batch size, then there will be rows discarded. If we set the committed value to the current time, the next batch would miss those rows, which is clearly not the intention. This is why the agent does not take the current time for the new `:committed_value`, but the value in the `modificationtime` column of the last row. Because of this, it is highly recommended to order by `modificationtime` in the query.

```
reader.scaleAwareNumeric = true
reader.expandBigFloats = true
```

These two properties were developed by me. We had some problems with numbers displaying in the JSON output. I will demonstrate it with the following example:

```
{"pandaid": 896.0, "milliseconds_since_submission": 1.45678886E9 }
```

The first problem is that the PanDA ID is displayed as a float, but all IDs are actually integers. In the database the type is a `NUMBER(12, 0)` which means that although the `NUMBER` type would allow it by default, but this specific type has disabled fractional numbers for this column. So I added a check in the reader which checks if the numeric type (which is called `NUMBER` in Oracle, but this naming varies per DBMS) has a 0 scale or not. If yes, the read values will be treated as integers for that specific column.

The second problem is that `milliseconds_since_submission` is displayed in an exponential format. This can result in precision loss, and also might be inconvenient to read for humans (although the official JSON format does support it). The cause of this issue is that `Float.toString()` will use the exponent format when the value to convert to string is bigger than a specific number.

```
duplicatedEventsProcessor = true
duplicatedEventsProcessor.size = 100000
duplicatedEventsProcessor.header = false
duplicatedEventsProcessor.body = true
duplicatedEventsProcessor.path = /etc/flume-ng/agent-jdbc/last_event.log
```

The duplicated events processor checks if there are duplicate events. By theory, this may happen when the agent gets restarted, and this will ensure that there will be no distinct events with the same data. For this, the event processor stores the hash string of each event in its buffer, which we have defined to be 100,000 events big. Since a hash string is pretty short (compared to the original data), this file just uses a few megabytes on disk. The header and body properties of the processor define if the headers and/or body should be used when computing the hashes. We set a timestamp header with a Flume interceptor when we get an event in the source (so we know when it has been read from the database), so even though two events are identical in data, they will still not be picked up by the duplicate events processor, because the timestamp, and so the hash will differ. So we disabled the headers for computing the hashes.

```
interceptors.itimestamp.type = timestamp
```

The `itimestamp` interceptor adds a timestamp header to the event, which is the current time in a POSIX time format.

```
interceptors.iproducer.type = static
interceptors.iproducer.key = producer
interceptors.iproducer.value = panda
interceptors.itypeprefix.type = static
interceptors.itypeprefix.key = type_prefix
interceptors.itypeprefix.value = raw
interceptors.itype.type = static
interceptors.itype.key = type
interceptors.itype.value = all
interceptors.iversion.type = static
interceptors.iversion.key = version
interceptors.iversion.value = 1
```

Static interceptors just add an extra header with a static value for a defined key. So the code above will just append these headers to all events:

```
producer: panda
type_prefix: raw
type: all
version: 1
```

The first three headers are always required, and set on all our Flume sources. In section 4.2.4 I will describe how we use headers to route our data. The last header named `version` is only checked by the ElasticSearch sink (as of now), which routes to a different index based on the value.

4.2.3 The query executed by the JDBC agent

The full version of the query can be found in the attachments within the file `panda.query`. This is a simplified version, which uses `SELECT *` instead of selecting the specific columns we want.

```
SELECT * FROM (
      SELECT *
       FROM atlas_panda.JOBSACTIVE4 WHERE  modificationtime >=
       TO_DATE(':committed_value', 'YYYY-MM-DD HH24:MI:SS')
       AND cloud != 'OSG' AND VO NOT IN ('cms','CMS')
UNION ALL
      SELECT *
      FROM atlas_panda.JOBSARCHIVED4 WHERE modificationtime >=
      TO_DATE(':committed_value', 'YYYY-MM-DD HH24:MI:SS')
      AND cloud != 'OSG' AND VO NOT IN ('cms','CMS')
UNION ALL
      SELECT *
      FROM atlas_panda.JOBSDEFINED4 WHERE modificationtime >=
      TO_DATE(':committed_value', 'YYYY-MM-DD HH24:MI:SS')
      AND cloud != 'OSG' AND VO NOT IN ('cms','CMS')
UNION ALL
      SELECT *
      FROM atlas_panda.JOBSWAITING4 WHERE modificationtime >=
      TO_DATE(':committed_value', 'YYYY-MM-DD HH24:MI:SS')
      AND cloud != 'OSG' AND VO NOT IN ('cms','CMS')
) ORDER BY "modificationtime"
```

What this does is ask for all the data in 4 tables at the same time:

- the active jobs table
- the archived jobs table
- the defined jobs table
- the waiting jobs table

All table names end with the number 4. This number indicates the version of the framework. Most queries and table names have a 4 at the end, indicating that there are (or were) also older versions at hand.
Each subquery filters on:

- the *VO (Virtual Organization)* should not be CMS (as of now, we are only monitoring ATLAS jobs using the PanDA database)
- the Cloud should not be OSG (this is because the database is also used by some other experiments, and data could be mixed up otherwise)
- the modification time should be later than the dynamically changing `:committed_value`, which is initialized by the Flume agent.

When the rows returned by the subqueries are unionized, there is a final ordering done based on the modification time. This is important, because the agent will store the `modificationtime` of the last row returned (with the `TOP @BatchSize` added). If the result were not be ordered by this timestamp column, the last row would have a random time between the latest `modificationtime` and `:committed_value`.

4.2.4 Mapping from Flume event headers to output directions

Our architecture is based around three main data storage formats, which are:

- an *ElasticSearch cluster* provided by the ElasticSearch service
- an *HDFS cluster* which is shared with other IT services (each service has its own directory with proper permissions)
- an *Apache Kafka deployment* with multiple broker machines for high availability

Not all input sources route to all outputs, since each data storage format is for different usages:

- ElasticSearch is mostly used to store the documents for displaying Kibana dashboards for users.
- HDFS is used for long-term archival and for supplying data for batch jobs.
- Kafka sends the events as messages to the subscribed consumers, which are the streaming jobs doing real-time analytics.

As noted earlier, three Flume headers are mandatory: `producer`, `type_prefix` and `type`. The producer is the name of the input. For example, `fts` (File Transfer Service) or `panda` (the PanDA database rows). The type prefix is for differentiating between different representations of the same data, which is mostly done by having a `raw` prefix for the data coming from the sources, and using a different prefix for processed data. The type is used for differentiating between event types. *FTS* triggers messages for transfer started, state change, or completed. Each type's message is built up differently, so they have to be under a different type in ElasticSearch too, so we can match the templates to the messages.

Each storage system has a Flume sink. This is where the headers are translated into output directions. Originally, the route to these different sinks were defined in each source. This resulted in channels getting full even if one of the three sinks were down, since the message has to be kept until the sink is back again.
Also the architecture was quite complicated at this point. So the team decided to write to Kafka from all sources, and pull from Kafka inside the sinks. This introduces Kafka as a single point of failure, but the arguments against these are that Kafka is either way a

key point to our infrastructure, and by using its high availability settings (using partitions and replicas), we can ensure that it could handle VM's falling out of the cluster.
Before this change, if we had had 6 sources and 3 sinks, with each source writing to all sinks, there would have been 18 different ways the messages could go. With having Kafka inbetween, that number would be reduced to 9, 6 connections between each source and Kafka, and 3 connections between Kafka and each sink. Figure 4.2 describes how the data comes from Kafka and is split into two streams.

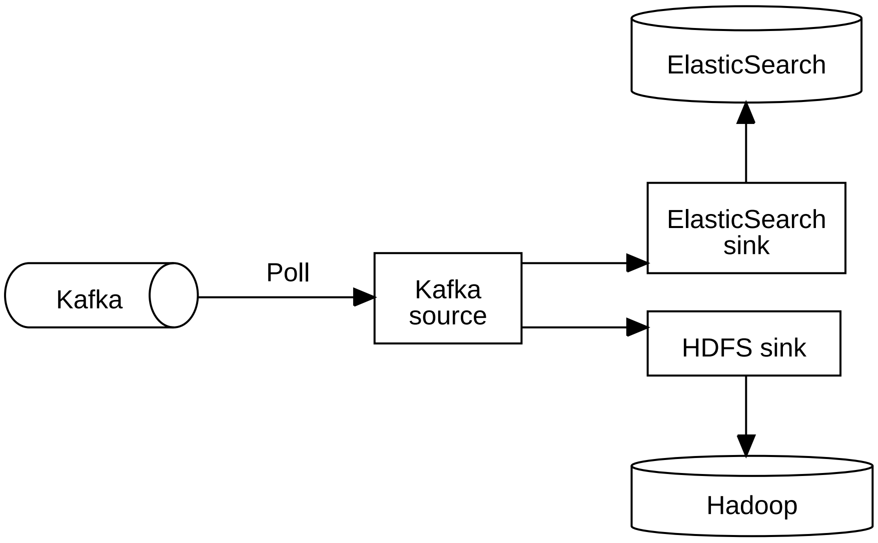


Figure 4.2. Flume Kafka source splitting incoming data to Hadoop and ElasticSearch.

HDFS sink

The HDFS sink dumps all the data into files in HDFS directories. The path for these files can be pretty long, but it is easy to write jobs for. This is the format defined in the Flume configuration:

```
/project/monitoring/archive/%{producer}/%{type_prefix}/%{type}/%Y/%m/%d.tmp/
```

- `/project/monitoring` is our own directory provided by the Hadoop service.
- `/archive` is where we put all the important data. We also have a `/testing` directory on this level to just store temporary files for testing jobs or similar things.
- `/%{producer}` is the header which changes per input source.
- `/%{type_prefix}` is the representation (mostly raw).
- `/%{type}` is the type of data.
- `/%Y` is the year extracted from the timestamp (which is set in the Flume sources).

- `/%m` is the month of the year extracted from the timestamp (as a 2 digit number).
- `/%d.tmp/` is the day of the month extracted from the timestamp (also as a 2 digit number), with .tmp added as a postfix.

Each file is prefixed with the following string: `FlumeData.%{hostname}` The hostname in the prefix ensures that two HDFS sinks do not try to write into the same file, which would drastically reduce performance (because of the waiting for releasing of file locks).

ElasticSearch sink

The 2 parameters for inserting into *ElasticSearch* are the index name and the type name. The index name is parametrized with the producer and the version:

`monit_%{producer}_v%{version}`

The version is only mapped to the destination in ElasticSearch, because Kafka and HDFS do not check if the messages conform to any structure, they just dump the body of the event to its final destination. But in ElasticSearch, we use index templates, which have to match the incoming documents. So we also utilize the version header, to forward the document to a different index in case the version increases.

The type name contains the 3 mandatory headers:

`%{type_prefix}_%{producer}_%{type}`

The index templates had been previously constructed, refined and uploaded to our cluster before production data started coming.

Kafka sink

Kafka calls the queues which store messages for different types in separate topics. The mapping from Flume headers to topic names is pretty simple:

`%{producer}_%{type_prefix}_%{type}`

So for example, a consumer should subscribe to topics such as `panda_raw_all` or `fts_raw_complete`.

4.3 Increasing the reliability of the Flume source

While we had the source deployed in our QA environment, we discovered a few issues, which needed intervention and applying fixes either to the original source code of the Flume source or the hostgroup Puppet configuration.
Firstly, we noticed that after an agent restart, the first database read would only happen

10-30 (sometimes even 60) minutes later. A few minutes delay would be understandable, since it is just supposed to read every 3 minutes, but this was way more than that. The logs were not pointing to anything suspicious, they reported that the source was indeed started, but the database read batch was not executed.
Usually when I do not get enough information from the Flume logs, I attach a debugger with the sources opened, and add some breakpoints. Since I did not know where to put a breakpoint, I restarted the agent, waited a minute to make sure the initialization process surely finishes, and then attached the debugger. After that, I paused all threads, which shows the current stack trace of each thread. There were quite a few of them, but I quickly found the one called `Main`. Checking the stack trace it was waiting on a native method called `getRandom()`, which was inside the Oracle Thin JDBC client implementation. Then it all occurred to me: the Oracle JDBC client uses `/dev/random` for its security model, but `/dev/random` blocks if the entropy pool is empty, which blocks until there is enough entropy. After issuing a fix, which solved this, the agent immediately started fetching the latest data from the database, as expected.
Following to the deployment in QA we executed some simple tests against the Python collector . Since real-time processing step does not contain any post-processing, we were interested to find out if all the data is extracted from the database correctly, and there are no duplicates or missing jobs. We ran the flume sources for multiple hours, and compared the number of documents in ElasticSearch on non-overlapping windows of 10, 30 and 60 minutes.
All numbers were matching, except on the window of 1 hour, where the new cluster contained 1 document less compared to the old one. Using binary search, I pinpointed the exact timestamp at which the difference appeared between the two clusters.
Since the timestamp format declared in Oracle has seconds precision, even a one second bin yields multiple results. I retrieved the documents from both clusters for the one-second bin, and compared the task IDs, which could be found in the body of each document.
When I discovered which specific ID was missing, I did a search in Kafka and HDFS for that specific ID, but there was no result found. To make sure this was not an error in the query itself, I did a search for a different task ID from the same bin, which did return results for both storage systems.
So the specific job update was not present in either of the 3 used storages. This means that the root cause had to be at the source. I had a look at the source code, and inspected the parameters as well. The configuration displayed in section 4.2.2 describes the usage of a duplicated events processor which can store up to 100 000 events. I suspected that this might be the issue, since an incorrectly detected duplicate event might be dropped, while it should not be. I opened the source code of this processor, and checked the implementation of the detection for duplicates.
The processor generates a hash based on the body and or headers of the event (depending on the configuration settings). The hashing is executed using the default Java `String.hashCode()` algorithm, which returns a 32 bit integer, approximately 4.3 billion different combinations. One might think that the probability of a hash collision would be

100 000 divided by this number, thus 0.002%.
But this is not the case. A principle called the *birthday problem* applies here:

> The probability p(n) that in a room of n people, there exists at least a pair that has the same birthday, we ignore the variation in distribution (in reality, not all the dates are equally likely) and assume the distribution of birthdays are uniform around a year of 365 days.[4]

Of course, we are not talking about people and birthdays here. But our hashing problem is very alike: instead of birthdays, we take the hashed value, and instead of 365 possible values, we have 2^{32} combinations (the distinct number of values an integer can take).
If $n = 10^5$ and $k = 2^{32}$, the probability could be calculated based on the following formula:

$$p(n) = \frac{k!}{k^n(k-n)!} = \frac{2^{32}!}{2^{32^{10^5}}(2^{32} - 10^5)!} \tag{4.1}$$

That does not seem like an easy calculation, even for computers. But there is an upper bound formula which works fairly good on larger numbers:

$$\bar{p}(n) = e^{-(n(n-1)/2)/k} = e^{-(10^5(10^5-1)/2)/2^{32}} = 0.3121 \tag{4.2}$$

So the actual probability of no hash collision is around 31%, which will lead to events being dropped even if they are not duplicates.
To fix this, I changed the hashing implementation to MD5, which hashes to a 128 bit value, making the probability of two hashes colliding almost impossible. The change also required to implement hashing of the `Map<String, String>` type, because the headers are stored as key-value pair in this type. I studied the original Java source code to get some ideas, and found that they XOR the hashed value with the hashed key, which makes the following two maps have the same hash, and print this code `true`:

```java
Map<String, String> map1 = new HashMap<String,String>();
Map<String, String> map2 = new HashMap<String,String>();
map1.put("foo", "bar");
map2.put("bar", "foo");
System.out.println(map1.hashCode() == map2.hashCode());
```

In my implementation I added an offset to the value hash before applying the XOR operation, so that it would return different hashes in the case above.
Aside from this patch, I also made some other improvements to the JDBC Flume agent source code, which were all merged in the library and are available to the public on GitHub.[16, 17]

4.4 Kafka, the middle man

As one may have noticed on the architecture diagram 2.2 on page 10, every component in the stack depends in some way on Kafka. I will give a short introduction on Kafka, since its initial release was not so long ago, and it might be unknown to some.
Apache Kafka is a message queue with extra features. Its most prominent feature is that it stores the messages even if no one or everyone consumed it. Consumers can subscribe and receive messages in real-time, but also fetch the messages from the past hour (given that those are still available).
Kafka is supposed to be running in a distributed way, but it can also run on a single machine, or even in-process (this is very useful when Kafka is a unit testing dependency in Java projects). Kafka has a dependency on Zookeeper, which is used to coordinate the distribution of messages, registering the consumers, and discovering other brokers in the same domain.
Storage of messages can be done in-memory and on the hard disk as well. The former is faster, the latter has a lot more space to offer. Usually the speed of the hard disk is more than enough, and the throughput can scale by adding new broker machines.
Kafka tries to solve the usual problems with distributed systems by itself. When the Zookeeper connection URL is set to the same value for different machines, Kafka will take care of:

- Discovery and setup of other brokers
- Load and consumer distribution
- Automatic deletion of data after certain time or space consumed
- Support for streaming data from multiple brokers at the same time

At the monitoring team, we have been using Kafka for a while, although not as a production service in the beginning. But together with the redesign of the architecture, the need came for a temporary storage service, which would be able to store messages for some hours at least. This was required because a processing service might be down, and delayed output is even better than lost output. Another requirement is distributed operation, because if Kafka is inoperable, the whole data flow stops. We wanted to minimize the risk of this.
Although Flume does also fulfill these requirements, Kafka has a big advantage over Flume: it plays really well with other software. It is very easy to connect to Spark or Flume with it, which was very important for us. Also it takes care of distributed read and write, which is very important in the case of Spark. When you read with multiple machines (or executors) from a single source, you are mostly limited to the throughput to that single source. Multiple consumers reading the same data are called consumer groups in Kafka terminology.

When using Kafka, it seems that you are still connecting to a single source, but if the data is stored on multiple Kafka brokers, Kafka will take care of connecting each consumer to a different broker, which will multiply the speed. In figure 4.3 one can see an example where two different consumer groups read from the same two Kafka brokers. In theory, the two groups should finish reading at the same time, since there are only 2 brokers, which will be the bottleneck when reading with 4 consumers.

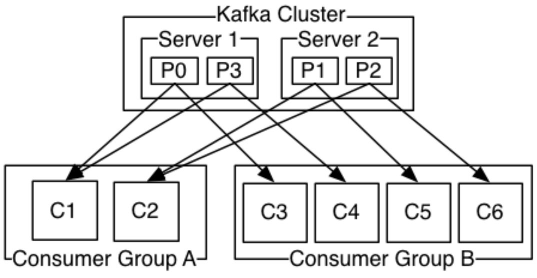


Figure 4.3. Kafka consumer groups reading from two brokers.

Usually one does not use a Kafka setup for just distributing one type of data. To separate messages of different types, so-called *topics* can be used. When uploading data to or downloading data from a broker, you have to define a topic. Similar messages will be in the same topic, which can be referenced by their name. Each topic will have two offsets: the *start offset* and *end offset*. When a message is uploaded to a Kafka topic, it will get an offset value. The first one will get the number 1, the second 2, it always increases by one. The topic's start offset is the offset value for the oldest message it stores, the end offset is the offset of the most recent message. Usually messages are automatically deleted when they are too old, and new ones are coming constantly, so these 2 markers are always increasing.

4.4.1 Setting up and optimizing Kafka for monitoring data

Initially, we started with a Kafka development setup of 3 brokers, which was enough for us for a surprisingly long time. Setting up a Kafka cluster was also pretty easy. It requires Java 8, a Zookeeper URL in the configuration file, and after that its service can be started. If another machine has the same Zookeeper URL, it will automatically join the cluster, and start fetching the messages from the other broker so that the data is more distributed.
The 2 main parameters for scaling are *partitions* and *replicas*. The number of partitions means that how many machines a topic should be distributed onto. The number of replicas is the number of copies to keep in the cluster, so if one machine falls out, another machine can still have the same data, so there is no data loss at all. Each partitions will have their own offsets, but the numbers can differ, since distribution is not always even.

In figure 4.4 one can see an example where one partition has less messages than the other 2.

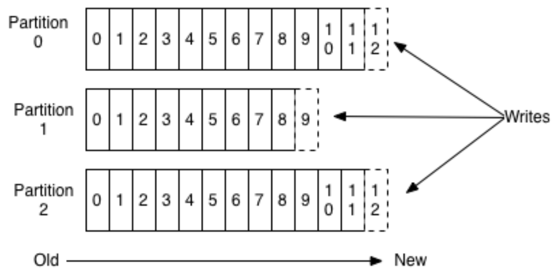


Figure 4.4. Anatomy of a topic.

Replicas copy partitions, which means that offsets should be the same in the case of flawless operation, but there might be times when they are out of sync, which is not a problem, but is usually a forecast that the cluster can't deal with the current throughput or there are communication issues. After a few months, we started seeing issues with our 3 broker cluster. The common problem was that the disks of the brokers got almost full. We fixed this by fine-tuning the log retention time to delete outdated messages more frequently. But this was not enough for long, since this just delayed our problem. We were in need of a cluster which would be able to cope with our present and future throughput. So we designed 2 clusters, a smaller one for development, and a bigger one for production. Its parameters are:

- 16 brokers, each 80 GB of disk space
- 20 partitions per topic
- 3 replicas for each partition
- 12 hour data retention time

Although the number of partitions is bigger than the number of brokers, this is not a problem, some brokers will have 2 partitions instead of 1. Choosing a higher partition number was done to prepare for the future: if we add 4 more machines, the partitions will be rebalanced, and each broker will have 1 partition.
A Kafka cluster like this can be easily used to connect to other applications. Flume has a Kafka sink and source, which support parallel reading by making use of consumer groups, and writing as well (the different machines just write to the same topic). Spark has a Kafka connector, both for streaming and batch usage. The connector automatically scales if it sees that there are multiple brokers and executors available.

Chapter 5

Results and conclusion

In this short chapter I will summarize the state of this solution, and will give a few results based on the latest progress. Also I will give an outline of the work required to make this solution a production-ready architecture.

5.1 Results

In this past year I had the possibility to work at one of the few places in Europe, where I had access to the resources and technical knowledge to create and run truly high-performance distributed applications.
All this could not have been possible without the aid and experience of my colleagues. Many of us worked hard together to build something which we all could be proud of and use with great confidence. In my opinion, the resulting architecture fulfills all our present and future needs - reliability, scalability and maintainability. I had to leave CERN before it was put to test in a production environment, but the initial results while being still in development were promising.
While I was still there, we were able to deliver around 100 GB of data through the system per hour, day and night. We were able to deploy multiple Spark jobs which were running and monitored constantly. All output data is being streamed to HDFS and ElasticSearch, where it is stored in a well-structured format, making it available for future processing or real-time visualizations as well. Since the architecture was designed to be scalable, it could deal with throughputs 10 times more than the current, just by adding more machines to the different hostgroups.
In this short period (as it seemed to me), I developed a fair understanding about how Spark, Flume and Kafka works. I had the chance to put my experiences into some important data management applications, which will be the replacements of some outdated technologies, and an architecture which will hopefully be able to serve the needs of monitoring for the many years to come.

5.2 Future work

The work I did at CERN is by no means completed. Although each of the applications are working as intended in the pre-production environment on actual production data, they can be improved and extended by need. Even though I did my best to finish all of the work, there are still feature requests ready to pick up.
The main directions where the implemented software stack and the Spark jobs could be improved:

- There are still some external data sources which need to be added. These are sites which do not report statistics through PanDA, but have their own interface (for example, the MIRA supercomputer[7], or for a special job type called TZero).
- Create example dashboards in Kibana for the users to explore the options they have. Also provide education if necessary.
- By having not only job monitoring data in ElasticSearch, but also transfer and network statistics, more complex dashboards could be created which combine the data of these monitoring systems, making it easier for the users to have an overview of the job(s) they want to check.
- Although the Spark jobs have been running in pre-production for multiple months, they still need extensive testing to check if they are production-ready. For example, it has to be checked if there is no data lost on a restart, and how they would handle if any of the tasks would get stuck.
- There are multiple ways to reduce resource usage at various places, but we didn't want to do premature optimizations, so this is still left to do when we feel it is time.

As time will go on, the list of these improvements and feature requests will surely grow.
I hope that the results of this 12 month period were useful to the collaboration of the thousands of scientists, engineers, and any other people who, by delivering exceptional work day-by-day, make the existence of CERN a reality today.

List of Figures

Bibliography

[1] CERN Configuration team docs - Adding secrets. `https://configdocs.web.cern.ch/configdocs/secrets/adding.html`. Accessed: 2016-08-15.

[2] ATLAS Experiment - About ATLAS. `http://atlasexperiment.org/`. Accessed: 2016-08-17.

[3] Ian Bird. Computing for the Large Hadron Collider. *Annual Review of Nuclear and Particle Science Vol. 61: 99-118*, 2011.

[4] Birthday problem. `http://www.math.cornell.edu/~mec/2008-2009/TianyiZheng/Birthday.html`. Accessed: 2016-06-18.

[5] Apache Flume JDBC source. `https://github.com/cerndb/flume-ng-audit-db`. Accessed: 2016-04-16.

[6] Changing job config via web UI resets cpu, mem, etc. `https://github.com/mesos/chronos/issues/426`. Accessed: 2016-08-22.

[7] Alessandro Di Girolamo. The atlas distributed computing project for lhc run-2 and beyond. Technical report, ATL-COM-SOFT-2015-149, 2015.

[8] A Anisenkov et al. AGIS: Evolution of Distributed Computing information system for ATLAS. *Journal of Physics: Conference Series, Volume 664*, 2015.

[9] E Karavakis et al. Processing of the WLCG monitoring data using NoSQL. *Journal of Physics: Conference Series, Volume 513, Track 3*, 2014.

[10] Fernando Barreiro Megino et al. PanDA: Exascale Federation of Resources for the ATLAS Experiment at the LHC. *EPJ Web of Conferences, Volume 108, 2016*, 2016.

[11] G Aad et al. Combined Measurement of the Higgs Boson Mass in pp Collisions at sqrt(s) = 7 and 8 TeV with the ATLAS and CMS Experiments. *Physical Review Letters - PRL 114, 191803 (2015)*, 2015.

[12] G Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Physics: Conference Series, Volume 664*, 2015.

[13] K De et al. The future of PanDA in ATLAS distributed computing. *Journal of Physics: Conference Series, Volume 664*, 2015.

[14] T Maeno et al. Overview of ATLAS PanDA Workload Management . *Journal of Physics: Conference Series, Volume 331, Part 7: Distributed Processing and Analysis*, 2011.

[15] Lyndon Evans and Philip Bryant. LHC Machine. *Journal of Instrumentation, Volume 3, August 2008*, 2008.

[16] cerndb/flume-ng-audit-db - [MONIT-20] Improvements to type handling of the JDBC reader . `https://github.com/cerndb/flume-ng-audit-db/commit/81dd4c612199c51bdd448761ba7a43f739a414c9`. Accessed: 2016-06-18.

[17] cerndb/flume-ng-audit-db - [MONIT-228] Add MD5 hashing option . `https://github.com/cerndb/flume-ng-audit-db/commit/e5ee8bbe40dabd7b040597ab5bf48518530c1a99`. Accessed: 2016-06-18.

[18] The Philosophy and design of ScalaTest. `https://mesos.github.io/chronos/`. Accessed: 2016-10-18.

[19] Tim Berners-Lee's proposal. `http://info.cern.ch/Proposal.html`. Accessed: 2016-08-16.

[20] Marathon documentations - Using a Private Docker Registry. `https://mesosphere.github.io/marathon/docs/native-docker-private-registry.html`. Accessed: 2016-08-15.

Appendix

A.1 Mesos master puppet configuration

```
class hg_monitoring::spark::master inherits hg_monitoring::spark::base {

$java_8_path = '/usr/lib/jvm/jre-1.8.0-oracle.x86_64/bin/java'

# Add all Marathon parameters here. For all valid options,
# see the docs:
↪  https://mesosphere.github.io/marathon/docs/command-line-flags.html
$marathon_defaults = {
'MARATHON_TASK_LAUNCH_TIMEOUT' => '360000' # 6 minutes
}

# This node will act as a Mesos master
class { '::mesos::master':
work_dir => '/var/lib/mesos',
options  => {
quorum   => 1,
hostname => $::hostname,
ip       => '0.0.0.0',
},
}

# 5050 is the default port for Mesos masters
firewall { '100 mesos ports':
proto  => 'tcp',
dport  => ['5050'],
action => 'accept',
}

# All Mesos masters have Marathon installed
# The Mesos class already added the Mesos yum repos,
# so we can install it immediately.
package { 'marathon':
ensure   => installed,
provider => 'yum',
}

# The marathon defaults are flattened in a file,
# which will be read when the service starts
file { '/etc/default/marathon':
ensure  => present,
content => template('hg_monitoring/spark/marathon_defaults.erb'),
notify  => Service['marathon'],
}

# Marathon only works on Java 8, so we have to switch from the default
 ↪  Java 7
exec { 'set-java-8-from-alternatives':
command => "alternatives --set java ${java_8_path}",
path    => ['/usr/sbin', '/usr/bin'],
unless  => 'ls -l /etc/alternatives/java | grep 1.8.0',
} ->

service { 'marathon':
ensure     => running,
enable     => true,
hasstatus  => true,
hasrestart => true,
require    => File['/etc/default/marathon'],
}

# 5050 is the default port for Mesos masters
firewall { '110 marathon port':
proto  => 'tcp',
dport  => ['8080'],
action => 'accept',
}
}
```

A.2 Mesos worker puppet configuration

```
  # Monitoring config directory
  # Needs a configuration directory for monit secrets
```

```
file { ['/etc/monit/']:
  ensure => directory,
}

# Keytab file for kerberos autentication
# The monitops_keytab is in tbag at monitoring top level
# Allows us to read from / write to HDFS
teigi::secret { 'monitops_keytab':
  path => '/etc/monit/monitops.keytab',
  mode => '0755',
} ->

# IT es-monit password file
# Allows us to read from / write to elasticsearch
teigi::secret { 'monit_es_password':
  path => '/etc/monit/monit_es',
  mode => '0755',
}

# docker.cern.ch login credentials for the monitops account
teigi::secret { 'monitops_docker_auth':
  path => '/etc/monit/docker.tar.gz',
  mode => '0755',
}

# Applications will be assigned a random port in the offered range from
↪  the
# resources declaration.
# So we will open up the entire port range, and will discover
↪  applications
#  with service discovery
firewall { '200 marathon applications':
  proto  => 'tcp',
  dport  => '7000-11000',
  action => 'accept',
}

# The following rules are required by Docker
# If not defined, they will be purged by puppet,
# and the containers will be unable to connect to

# Required for Docker 1/3
```

```
firewall { '000 forward to docker chain':
  proto       => 'all',
  chain       => 'FORWARD',
  jump        => 'DOCKER',
  source      => '0.0.0.0/0',
  destination => '0.0.0.0/0',
}

# Required for Docker 2/3
firewall { '001 forward accept all with state':
  proto       => 'all',
  chain       => 'FORWARD',
  action      => 'accept',
  state       => ['RELATED', 'ESTABLISHED'],
  source      => '0.0.0.0/0',
  destination => '0.0.0.0/0',
}

# Required for Docker 3/3
firewall { '002 forward accept all':
  proto       => 'all',
  chain       => 'FORWARD',
  action      => 'accept',
  source      => '0.0.0.0/0',
  destination => '0.0.0.0/0',
}

# Some applications are deployed as Docker images
# What we gain with this that we do not need to install every
↪  application
# on all workers, we just pull the image and launch it with the
↪  parameters.
package { 'docker':
  ensure   => installed,
} ->

service { 'docker':
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
}
```

```
# Mesos Chronos
package { 'chronos':
  ensure => installed,
}

firewall { '100 chronos port':
  proto  => 'tcp',
  dport  => 4400,
  action => 'accept',
}

service { 'chronos':
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package['chronos'],
}

# Open firewall to let connection from IT HADOOP analytix in
$hadoop_node_names =
↪  sort(unique(query_nodes("hostgroup~\"hadoop/aimon/datanode\"",
↪  'ipaddress')))
ensure_resource(hg_monitoring::modules::firewall_wrapper,
↪  $hadoop_node_names)

# Disable selinux.
# Needed to mount volumes (with secrets) to containers
$sysconfig_selinux = '/etc/sysconfig/selinux'
augeas { $sysconfig_selinux:
  context => "/files${sysconfig_selinux}",
  changes => ['set SELINUX permissive',],
} ->
exec { "SELinux to permissive mode on ${::fqdn}":
  user    => 'root',
  command => '/usr/sbin/setenforce Permissive',
  unless  => "/usr/sbin/sestatus | /bin/egrep -q\
    '(Current mode:.*permissive|SELinux.*disabled)'",
}
```

A.3 Example job configuration in Chronos

```
{
        "schedule": "R/2016-07-21T00:00:00Z/PT1H",
        "name": "my-job",
        "description": "My job description",
        "ownerName": "Monitoring service",
        "owner": "example@cern.ch",
        "container": {
                "type": "DOCKER",
                "image":
 ↪  "docker.cern.ch/monitoring-service/my-job:latest",
                "network": "HOST",
                "forcePullImage": true,
                "volumes": [
                {
                        "containerPath": "/etc/hadoop/conf",
                        "hostPath": "/etc/hadoop/conf",
                        "mode": "RO"
                },
                {
                        "containerPath": "/usr/lib/spark",
                        "hostPath": "/usr/lib/spark",
                        "mode": "RO"
                },
                {
                        "containerPath": "/etc/monit",
                        "hostPath": "/etc/monit",
                        "mode": "RO"
                },
                {
                        "containerPath": "/usr/lib/hadoop",
                        "hostPath": "/usr/lib/hadoop",
                        "mode": "RO"
                },
                {
                        "containerPath": "/usr/lib/hadoop-mapreduce",
                        "hostPath": "/usr/lib/hadoop-mapreduce",
                        "mode": "RO"
                },
                {
                        "containerPath": "/usr/lib/hadoop-hdfs/",
                                "hostPath": "/usr/lib/hadoop-hdfs/",
                                "mode": "RO"
                        }
                        ]
                },
                "cpus": "0.5",
                "mem": "1536",
                "retries": 1,
                "uris": [
                "file:///etc/monit/docker.tar.gz"
                ],
                "command": "/usr/lib/spark/bin/spark-submit ... "
        }
```