Introduction

Dans le cadre de ma formation d'ingénieur en informatique à l'Université de Technologie de Belfort-Montbéliard (UTBM), j'ai effectué mon stage au sein de l'Observatoire Astronomique de Strasbourg. Durant six mois, j'ai eu l'occasion de mettre en pratique mes connaissances acquises en cours et d'acquérir des compétences professionnelles.

J'ai choisi ce stage car le sujet proposé correspond à l'activité que je souhaiterais exercer et implique un travail dans un cadre lié à l'astronomie.

Dans ce rapport je vous présenterais l'Observatoire Astronomique de Strasbourg et ses différentes équipes dont le Centre de Données astronomiques de Strasbourg (CDS) qui en fait partie. Après cela, j'aborderais le déroulement de mon stage et son organisation. Puis je m'étendrais plus en détail sur les aspects techniques du stage.

Table des matières

I. Présentation de l'Observatoire

A) L’Observatoire de Strasbourg

i. Historique

Après la guerre franco-allemande de 1870, l’Alsace-Moselle fut cédée à l’Allemagne. L’empereur Guillaume Ier d’Allemagne veut faire de Strasbourg une vitrine du savoir du peuple Allemand et y construit une université comprenant notamment un observatoire astronomique et un jardin botanique.

Il est inauguré en 1881 et est composé de trois édifices : une grande coupole, un bâtiment des salles méridiennes (c’est à dire aligné sur l’axe nord-sud) comprenant deux coupoles et un bâtiment dédié aux bureaux et servant de résidence. La grande coupole abrite une lunette astronomique de sept mètres de long appelée le Grand Réfracteur construite en 1877.

L’observatoire est initialement utilisé pour l'observation de comètes, de météorites et d'étoiles variables, il sera plus tard utilisé pour la photométrie de nébuleuses et l’observation d'étoiles doubles. Vers les années 1970 l’observatoire commence à se développer vers l'archivage informatique, les observations au sol commençant à montrer leurs limites.

ii. Activité actuelle

L’Observatoire astronomique a aujourd’hui une activité de recherche et emploie 80 personnes en permanence. Ils travaillent au sein de trois équipes principales :

- Hautes Énergies (étude des trous noir, naines blanches, pulsars ...)
- Galaxies
- Centre de Données astronomiques de Strasbourg

iii. Centre de Données astronomiques de Strasbourg

J’ai travaillé au sein de l’équipe du Centre de Données astronomique qui est composé de 30 personnes comprenant des astronomes, des ingénieurs et des documentalistes. Leur mission est la collecte, la distribution et le traitement de données astronomique provenant de télescopes et de satellites. Ces données, appelées catalogues, comprennent aussi bien des observation du 19^{e} siècle jusqu’aux données du satellite

Gaia de l'Agence Spatiale Européenne publiées en septembre 2016. Il est possible d'accéder à ces catalogues à l'aide de différents services développés par le CDS.

Les services

Plusieurs services sont mis à disposition des chercheurs et astronome du monde entier leur permettant d'effectuer des recherches dans ces catalogues de données astronomiques.

Simbad est une base de données référençant la nomenclature et la bibliographie des objets astronomiques connus. Ce service accessible publiquement permet d'accéder aux informations comme les coordonnées, des mesures et des données bibliographiques sur plus de 9 millions d'objets astronomiques. Ces informations sont accessible grâce à un résolveur de noms.

VizieR est un service de catalogues astronomiques qui permet de faire des requêtes sur plus de 16750 catalogues pouvant contenir jusqu'à un milliards d'entrées. Il est possible d'utiliser ce service pour filtrer les résultats d'un catalogue en fonction de critères comme une position du ciel ou différentes données de mesure.

Aladin est un atlas interactif du ciel, il permet un accès à des images astronomiques

issues de différentes missions et couvrant l'ensemble du spectre électromagnétique. Ce service permet de visualiser des résultats de requêtes effectués sur Aladin et VizieR. Ce logiciel initialement disponible comme une applet Java est maintenant aussi disponible sous la forme d'un widget Javascript, ce qui lui permet d'être directement intégré dans une page web.

II. Organisation du stage

A) Objectifs

Pour assurer le fonctionnement des services présentés précédemment et leur efficacité future, le CDS fonctionne sur un mode de recherche et développement. Les données à

traiter ne faisant qu'augmenter au cours des années, il est nécessaire de prévoir l'évolutivité de ces applications. Mon stage s'est donc déroulé dans cette optique d'optimisation et d'amélioration.

Le travail de développement des ingénieurs du CDS se fait actuellement manuellement (compilation, déploiement), mon stage à donc eu comme but de tester un fonctionnement de type DevOps en collaboration avec des ingénieurs afin de rentre le travail de développement plus rapidement disponible et accélérer les tests. Le stage a donc eu pour sujet "L'apport du DevOps au CDS".

L'un des services du CDS permet de réaliser des crossmatch (jointure floue entre deux catalogues de données astronomique), actuellement ce calcul est fait sur une seule machine. L'étude porte donc sur l'utilisation de Spark et Hadoop afin de remplacer cette machine unique par plusieurs machines fonctionnant en cluster. Il aura donc fallu explorer différents moyens afin d'optimiser le temps de calcul de Spark.

J’ai aussi effectué un travail sur le service VizieR. Ce service est déployé sur différents miroirs à travers le monde afin d'assurer un accès rapide quelque soit l'origine géographique des requêtes. Le but à été de déployer ce service en se servant de Docker afin de voir les différents avantages ainsi que les limitations actuelles de la technologie.

B) Planning

Mon stage s'est déroulé en plusieurs étapes :

Exploration des différentes technologies disponible et apprentissage du fonctionnement de Docker, Hadoop et Spark. J'ai eu à ma disposition plusieurs machines de test sur lesquelles j'ai pu déployer un cluster Docker (Docker Swarm). Puis une fois l’environnement Docker prêt j'ai pu développer des image Docker Spark et Hadoop, celles-ci n'ayant pas de version officielle disponible. La partie suivante consistait à continuer le développement d'une application Spark afin d'améliorer ses performances. Notamment par la réécriture en langage Scala et le développement d'une application en python permettant la collocation des données sur le cluster HDFS.

Dans un deuxième temps j'ai pu collaborer avec le développeur principal du service VizieR. J'ai donc développé une image docker permettant de déployer rapidement les

mises à jour du service. Il a aussi fallu repenser certains aspect de l'application qui n'est pas prévue pour Docker au départ.

Vers la fin du stage l'IN2P3 (Institut national de physique nucléaire et de physique des particules) nous a donnée accès à un cluster de 9 machines plus performantes que ce dont disposait le CDS. Une fois leur environnement maîtrisé (configuration du cluster, déploiement des images, mise en place d'un tunnel SSH) j'ai pu effectuer des tests de l'application développée.

C) Méthode de travail

La méthode de travail était de type recherche et développement. C'est à dire que les objectifs à réaliser n'avaient pas de planning précis et demandais un certain travail de recherche. Notamment sur l'utilisation de technologies récentes telles que Docker et Spark, le CDS ne disposait d'aucun expert ce qui m'a demandé une certaine autonomie dans mon travail face aux problèmes que j'ai pu rencontrer.

Tout au long du stage des réunions étaient régulièrement organisées afin de mettre au point les différents avancements et discuter des problèmes rencontrés. A partir de ces réunions des objectifs suivant étaient définis.

D) Description de l'existant

i. Docker

La technologie Docker et plus globalement le système DevOps n'était pas utilisé à mon arrivé à l'observatoire.

ii. Crossmatch

Une première expérimentation avait été faite lors d'un stage précédent sur l'utilisation de Spark pour effectuer les crossmatch mais n'est pas utilisé en production.

Actuellement, un programme Java développé par François-Xavier Pineau fonctionne sur une seule machine puissante, ce serveur est mis à disposition sur le portail du CDS.

iii. VizieR

Ce service est déployé manuellement (installation de paquets et compilation de dépendances) sur les différents miroirs. Ces machines fonctionnent sur des systèmes

différents donc avec des versions de librairies différentes ce qui complique la tache d'installation.

E) Technologie utilisée

i. Docker

Logiciel libre permettant de démarrer des applications dans des conteneurs et de les déployer. Cette technologie permet d'avoir un environnement similaire à un hyperviseur de machines virtuelles mais avec un impact beaucoup plus faible sur le système hôte

ii. Hadoop

Framework libre comprenant plusieurs outils facilitant l'exécution de calculs distribués. Les principales fonctions sont le système MapReduce (implémentation du design patern du même nom permettant l'exécution de calcul en parallèle) et le système de stockage distribué HDFS (Hadoop Distributed File System)

iii. Spark

Framework open source présenté comme le successeur de MapReduce optimise les différentes étapes du calcul en stockant les résultats intermédiaire en mémoire ce qui accélère le calcul. Il dispose aussi d'un langage SQL propre lui permettant d'effectuer des requêtes complexes sur des données.

La méthode utilisée par Spark pour distribuer le calcul est de découper les données en partitions qui seront envoyées sur les différents nœuds. Lorsqu'un nœud à besoin de données qui ne se trouve pas dans les partitions qu'il a à sa disposition il va aller les chercher directement sur le nœud qui les possède, ce comportement est appelé shuffle.

iv. Scala

Le framework Spark est principalement développé en Scala mais propose des APIs en Java et Python. Cependant ces APIs sont moins évoluées que l'API Scala qui permet un accès à plus de méthodes. Ce

langage sera donc utilisé pour le développement de l'application Spark pour le crossmatch.

v. Apache HTTP Server

L'application VizieR utilise le serveur web Apache notamment pour l'utilisation des CGI (Common Gateway Interface) qui permettent d'afficher un contenu dynamique sans utiliser de PHP, cela par le biais de scripts Bash, Python ou de programmes C.

III. Travail réalisé

Dans cette dernière partie je vais détailler les différentes taches qu'il m'a été donné de réaliser en collaboration avec différents ingénieurs du CDS ainsi que d'autres centres de recherche.

A) Utilisation de Docker

Il existe différentes technologie de conteneurisation comme OpenVZ, LXC ou encore Rkt. Mon choix s'est posé sur Docker car il permet la création simple d'images que l'on peut ensuite déployer. En comparaison avec LXC et OpenVZ son utilisation est plus légère car chaque conteneur ne démarre que l'application et non tout un système de processus (initd ou systemd).

i. Fonctionnement

Le fonctionnement de ce système de conteneurs est divisé en plusieurs parties :

Les images sont des systèmes de fichiers contenant un système complet (gestionnaire de paquets, librairies ...). Il est possible de récupérer des images déjà existantes, par exemple il existe des images Ubuntu, Debian, CentOS mais aussi des images de systèmes préinstallés comme Apache, OpenJDK ou encore Tomcat. Ces images sont composées de plusieurs couches (ou layers) qui correspondent aux différentes étapes ayant permis la création de l'image, chaque couche possède un hash permettant de les identifier et contient uniquement les modifications opérées depuis la couche précédente. Ainsi, moins une image possède de layers moins elles pèsera lourd.

Lors du démarrage d'un **conteneur**, Docker se base sur l'outil RunC qui va se charger de créer un environnement isolé dans lequel l'application sera exécutée. Il est important de noter que lors de l'exécution, l'image de base n'est pas dupliquée mais

est directement montée sur le système de fichier de l'hôte. Docker utilise ensuite un système de Copy-on-Write, c'est à dire que lorsque l'application exécutée dans le conteneur voudra modifier un fichier contenu dans l'image, ce fichier modifié sera stocké sur le disque donnant ainsi l'illusion que le contenu de l'image est accessible en écriture.

Il est aussi possible de **créer des images** en partant des images de base précédemment citées. Cela se fait à l'aide d'un Dockerfile, un fichier qui peut s'apparenter à un script. Il faudra donc indiquer les différentes étapes de la création de l'image :

- L'image de base (Debian, Ubuntu, Java ...)
- Les variables d'environnement
- Les commandes à exécuter (installation de paquets, compilations ...)
- Les fichiers à copier depuis la machine hôte (configurations ...)
- La commande à exécuter par défaut au lancement du conteneur

Lors de l'exécution de plusieurs conteneurs, ils sont tous exécutés dans des environnements isolés (au niveau des processus, du système de fichier et du réseau). Ainsi pour permettre à deux conteneur de communiquer entre eux il est possible de créer des **réseaux Docker**, c'est à dire des réseaux uniquement accessible par les conteneurs désignés leur permettant ainsi de communiquer. Pour notre application avec Apache Spark cela permet aux différents nœuds workers qui exécutent les taches de communiquer avec le nœud master qui orchestre le travail.

Docker met aussi à disposition un outil de déploiement de conteneur, **Docker Swarm**. Cet outil se configure en quelque commandes consistants en la création du cluster (ce qui va générer une clé permettant de chiffrer les échanges entre les nœuds et permet d'authentifier les nouveaux nœuds à ajouter) et à la connexion des différentes machines. Les nœuds sont ensuite synchronisés par le système de consensus Raft qui leur permet à tous d'avoir les mêmes informations. Il est ensuite possible de démarrer des conteneurs sous la forme de services, c'est à dire que l'on peut spécifier différentes contraintes comme le nombre d'instances, les ports à rediriger et les réseaux à connecter. Concernant la redirection de ports il est intéressant de noter que Docker Swarm met en place un réseau dit ingress, ce réseau permet à un service d'être accessible depuis l'IP de n’importe-quel nœud même si celui-ci n'est en réalité exécuté que sur un seul nœud.

Docker Machines est une application permettant la création automatique de machines virtuelles afin de créer rapidement un cluster et de pouvoir tester le fonctionnement de Docker Swarm en local. Ce système m'a permit de tester rapidement le fonctionnement de Docker Swarm au départ.

Il est possible de déployer ses images en utilisant le service **Docker hub** qui est accessible gratuitement depuis leur site internet. Afin de ne pas dépendre de services tiers il est possible d'installer localement un **Docker registry** qui permet d'obtenir des fonctionnalités similaires au Docker Hub mais hébergées localement. Ce système est particulièrement utile en combinaison avec Docker Swarm car il permet un accès rapide aux images depuis les différents nœuds.

ii. Problèmes rencontrés

Au début de mon stage le système Docker Swarm était à ses début, j'ai ainsi rencontré plusieurs bugs (comme par exemple un malfonctionnement du DNS intégré au réseau Docker ce qui empêchait dans certains cas les conteneurs de communiquer entre eux) que j'ai pu rapporter à l'équipe de développement de Docker par le biais d'issue GitHub. Les développeurs pouvaient ainsi directement discuter avec les utilisateurs ayant rencontrés les mêmes problèmes.

iii. Création d'images Spark et Hadoop

Afin de pouvoir utiliser Spark avec Docker j'ai du créer les images, Apache ne distribuant pas d'images officielles. J'ai donc créé plusieurs images dont notamment des images de base Hadoop et Spark qu'il est possible de réutiliser afin de déployer les différents services tels que HDFS et les Spark Workers et Master.

J'ai construit ces images en partant de l'image `openjdk:8-alpine`, c'est à dire une image Alpine Linux (une distribution légère avec un gestionnaire de paquet plus efficace que Debian et CentOS, ce qui permet d'obtenir des images finales plus petites et une construction plus rapide) avec la librairie Java OpenJDK préinstallée dans sa version 8.

Exemple avec l'image `hadoop-base`

```
# On récupère l'image de base
FROM openjdk:8-alpine

# Puis on définit les différentes variables d'environnement qui seront disponnibles lors de la création de l'image et l'exécution du conteneur
ENV HADOOP_VERSION="2.7.3"
```

```
ENV JAVA_HOME /usr/lib/jvm/default-jvm
ENV HADOOP_HOME /app/hadoop-$HADOOP_VERSION
ENV HADOOP_CONF_DIR $HADOOP_HOME/etc/hadoop
ENV HADOOP_PREFIX /app/hadoop-$HADOOP_VERSION
ENV YARN_CONF_DIR $HADOOP_CONF_DIR
ENV CLUSTER_NAME CDS_HDFS

# On indique ensuite toutes les commandes qui permettront la création de l'image. Toutes ces commandes sont regroupées sous un bloc RUN, ceci afin de réduire le nombre de layers
RUN \

# La commande apk est le gestionnaire de paquet de Alpine Linux, il est possible de lui passer des options afin de réduire la taille de l'image : --no-cache permet de ne pas garder les paquets une fois installés et --virtual permet de définir un groupe de paquets que l'on pourra manipuler en bloc.

apk add --no-cache --virtual=build-dependencies \
        curl \
        tar && \

mkdir -p \
    /app && \

# On utilise ensuite la commande curl associés aux pipes UNIX afin de télécharger et de décompresser la dernière version de Hadoop

curl -sL --retry 3 \
"http://archive.apache.org/dist/hadoop/common/hadoop-${HADOOP_VERSION}/hadoop-${HADOOP_VERSION}.tar.gz" \
| gunzip \
| tar -x -C /app/ && \
rm -rf ${HADOOP_HOME}/share/doc && \

# Une fois les différentes étapes terminées il est possible de nettoyer les résidus et les paquets inutiles au fonctionnement de l'image

apk del --purge \
        build-dependencies && \
rm -rf /tmp/*

# Pour terminer tous les fichiers de configuration sont copiés depuis le système de fichier de l'hôte

COPY conf/core-site.xml /app/hadoop-${HADOOP_VERSION}/etc/hadoop/
COPY conf/hdfs-site.xml /app/hadoop-${HADOOP_VERSION}/etc/hadoop/
COPY conf/yarn-site.xml /app/hadoop-${HADOOP_VERSION}/etc/hadoop/
```

Cette image servira ensuite de base à la création d'une image Spark (qui dépend de Hadoop) et qui est construite de la même manière.

Bonnes pratiques pour faciliter la création d'images Docker

Les images précédemment décrites n'ont pas été construite de cette façon dès le début, il a d'abord fallu que je comprenne comment obtenir des images Docker optimisées.

Comme on a pu le voir, les images Docker sont composées de plusieurs couches, chacune contenant les différences avec la couche précédente. Ainsi lors de la modification d'un Dockerfile (par exemple pour mettre à jour un fichier de configuration) il n'est pas nécessaire de ré-exécuter toutes les commandes, les lignes n'ayant pas changées correspondront à des couches déjà existantes qui seront réutilisées comme un cache. Cependant un nombre important de couches va augmenter la taille de l'image puisqu'il faudra stocker toutes les petites modifications entre chaque couche, y compris les fichiers supprimés.

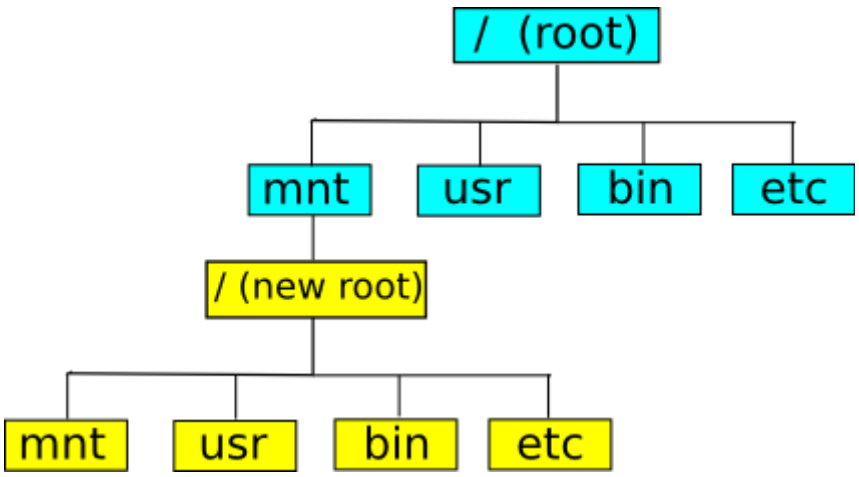
iv. La sécurité de Docker

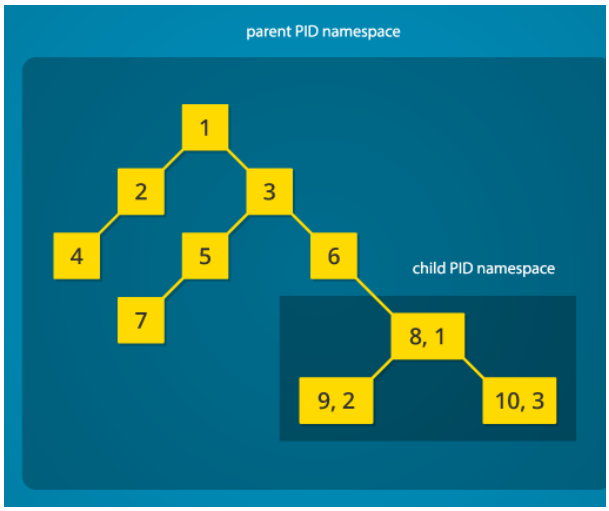
Afin de cerner le niveau de sécurité de Docker j'ai effectué une recherche sur le fonctionnement de Docker à bas niveau. Le but étant de savoir si on laisse une personne tierce exécuter une application dans un conteneur, cette personne pourrait-elle compromettre la machine hôte (volontairement ou non).

Lors du lancement d'un exécutable dans un conteneur celui ci est isolé à l'aide de fonctionnalités du noyau Linux, les Namespaces et les Cgroups.

Les namespaces sont divisés en plusieurs groupes qui permettent d'isoler chaques composantes du système Linux :

- Le système de fichiers
- La table des processus
- Le réseau
- Les points de montage





Les Cgroups permettent de limiter au au contraire de réserver des ressources sur le système, comme de la RAM ou du temps CPU. Cette fonctionnalité est utile dans le

cas ou on permet à d’autres utilisateurs de démarrer des conteneurs sur notre système, afin d’empêcher une utilisation abusive.

v. Persistance des données

Cette problématique est essentielle quand on utilise Docker. Une des règles principales est qu’un conteneur ne doit pas avoir d’état, c’est à dire que la suppression d’un conteneur et sa recréation ne doit pas modifier son comportement. Pour servir des sites web ou encore pour les services workers de Spark cela ne pose aucun problème, la configuration étant incluse dans l’image.

Cependant pour faire fonctionner HDFS il est nécessaire de stocker les fichiers et les métadonnées autre part que dans le conteneur. Autrement, à chaque redémarrage du service toutes les données seraient perdues. Dans le cas de conteneurs fonctionnant sur une seule machine il suffit de monter un dossier depuis l’hôte à l’intérieur du conteneur. Dans le cas d’un cluster Docker Swarm on ne peut pas savoir sur quel nœud quel service va s’exécuter il faut donc avoir un accès aux données depuis tous les nœuds. Pour cela j’ai utilisé un système de fichiers distribués GlusterFS adapté pour un usage serveur (la ou HDFS est conçu spécifiquement pour effectuer des calculs distribués). Ce système va permettre d’avoir un point de montage identique à toutes les machines, il suffit ensuite de monter ce chemin dans un service Docker afin d’assurer la persistance des données. Dans mon cas ce système était utilisé pour stocker les métadonnées du Namenode HDFS.

A titre d’information, il est à noter que Docker vient de racheter la Start Up française Infinit spécialisée dans le stockage distribué du même type que GlusterFS. Il est donc possible de voir apparaître cette fonction nativement dans une future version de Docker.

vi. Spark et Hadoop

Il est maintenant possible de créer les différents services qui vont permettre au cluster Spark / HDFS de fonctionner. Voici un exemple de commande permettant de démarrer les Spark Workers sur toutes les machines :

`docker service create --mode global --name spark-worker --network spark-net localhost:5000/spark-worker:2.1.0`

Lors du déploiement de Spark et Hadoop plusieurs problématiques se sont posées, comme l'accès aux interfaces web et l'upload de fichier vers le cluster. En effet, les

applications étant isolées du système il n'est pas possible d'accéder directement aux ports ou au système de fichier.

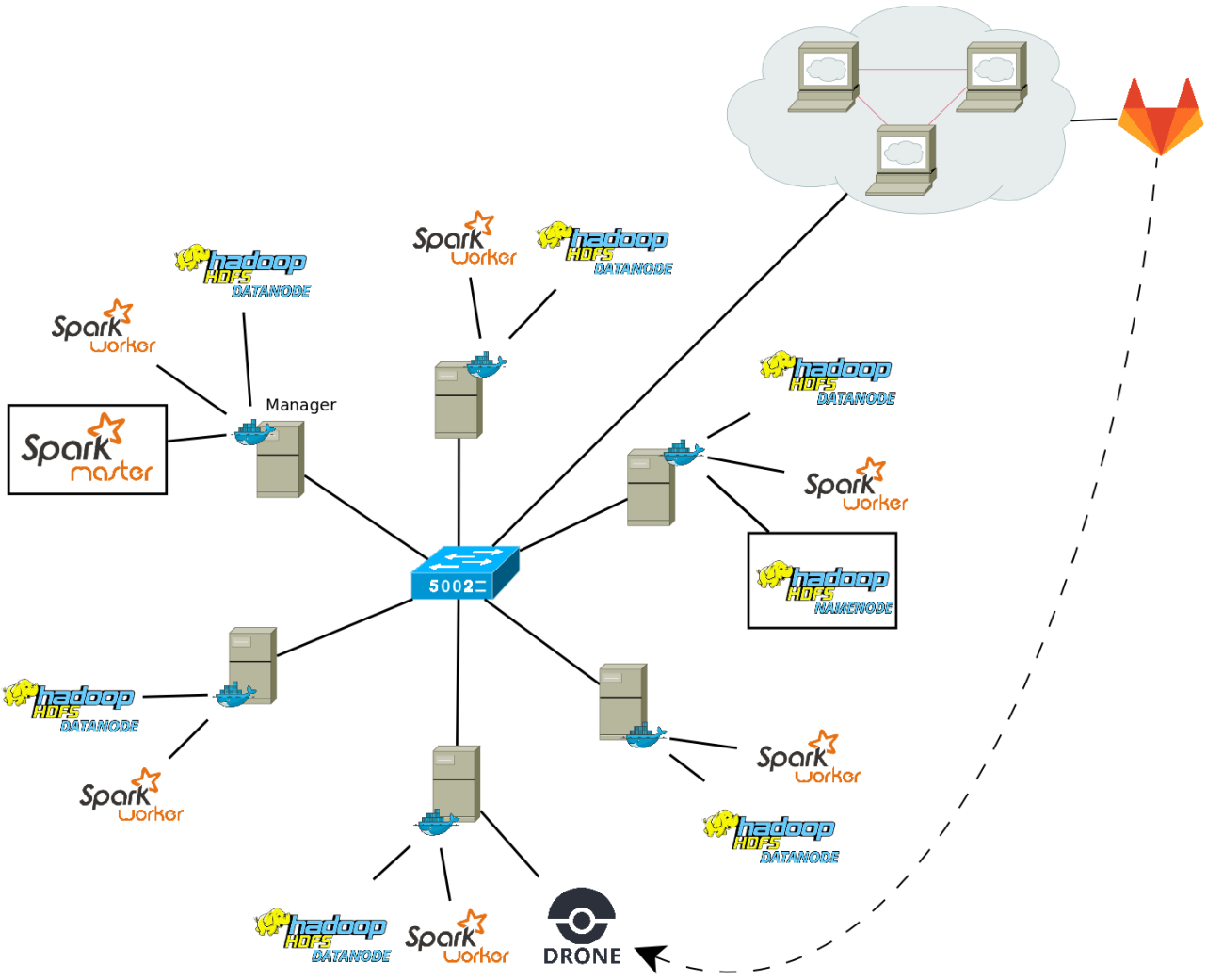
Accès à l'interface web de Spark

Pour l'accès aux interfaces web, le plus évident serait de rediriger les ports en utilisant les fonctions de Docker. Cependant en procédant de cette manière cela crée une carte réseau virtuelle supplémentaire visible par le conteneur. Par exemple en considérant l'un des Spark Workers il y aura une IP dédiée à la redirection du port et une IP dédiée au réseau Docker interne assurant la communication entre les conteneurs. Ceci pose problème à Spark qui gère mal les différentes IP et empêche donc les nœuds de se connecter entre eux.

Une première solution aura été de paramétrer un reverse proxy Nginx, c'est à dire un serveur web qui va écouter sur un port de la machine hôte et rediriger les requêtes vers le réseau Docker. Bien que fonctionnelle, cette solution n'est pas très évolutive, à chaque ajout ou suppression de service à rediriger il faudra modifier la configuration du serveur Nginx et le redémarrer.

Une autre solution beaucoup plus flexible est l'utilisation d'un proxy Squid. Le principe est similaire à la différence qu'il n'est pas nécessaire de configurer chaque service et qu'une configuration coté client est nécessaire pour rediriger le trafic vers ce proxy.

Voici à quoi ressemble le système mis en place :



On peut voir sur ce schéma les six nœuds de notre cluster reliés entre eux par un switch. Sur chaque machine Docker est installé et est configuré en mode Swarm. Ceci permet de démarrer les différents services tels que le Sparm Master et le HDFS Namenode qui orchestrent leur technologie respective. Le service Drone est connecté à GitLab par le biais de WebHooks qui lui permet de recevoir des commandes à chaque commit effectué.

Premiers tests de Spark et HDFS

Une fois un cluster fonctionnel mis en place j'ai pu commencer des tests pratiques sur Spark et HDFS. Pour cela il a fallu trouver un moyen de transférer des fichiers entre la machine hôte et le cluster HDFS / Spark.

Une solution est de démarrer un conteneur attaché au réseau Docker sur lequel sera monté un dossier de la machine hôte contenant, par exemple, les fichier à uploader ou le fichier JAR à exécuter. Une fois ce conteneur démarré en mode interactif (c'est à

dire avec un accès à la ligne de commande) un environnement Spark / Hadoop est disponible et un accès direct au cluster est disponible (Spark Master, HDFS Namenode).

vii. Jenkins / Drone

Au cours de mes développements j'ai effectué des recherches sur les systèmes de compilation automatisés. Les deux systèmes que j'ai testé sont Jenkins et Drone, j'ai pu faire fonctionner ces deux services en combinaison avec GitLab (un service de gestion de version Git) ce qui permet d'automatiser les compilations à chaque commit.

Dans le cas de Jenkins il est nécessaire de configurer individuellement chaque projet à l'aide d'une interface web. Il est possible de définir les commandes à lancer, un intervalle pour vérifier les mises à jour ou encore la façon de gérer le résultat de la compilation.

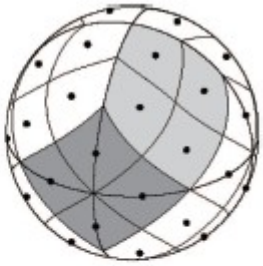
En comparaison, la configuration des projets avec Drone se fait à l'aide d'un fichier de configuration placé à la racine du projet dans Git. Les compilations sont effectuées dans des conteneur Docker ce qui permet de mieux gérer l'environnement de compilation et de partir d'un système propre à chaque compilation (pas de restes d'ancienne compilations). Ce fonctionnement avec Docker permet aussi à Drone de créer des images Docker et de les envoyer sur un registry nativement.

J’ai donc au final mis en place un système fonctionnant avec Drone car il permet un fonctionnement avec Docker ce qui évite d’avoir à gérer l’installation des dépendances.

B) Optimisations Crossmatch

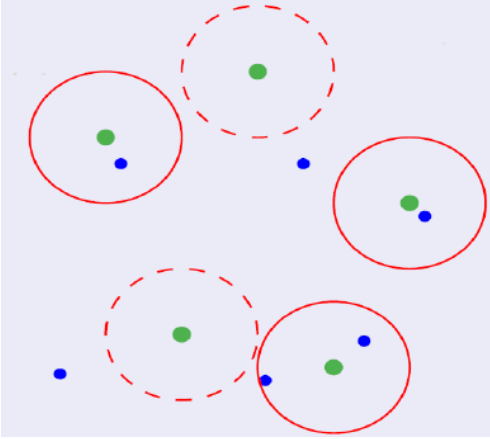
i. Fonctionnement du Crossmatch Spark

L’exécution du crossmatch se déroule en deux parties : un préprocess qui va parser des fichiers CSV contenant les données et les enregistre dans un format binaire en rajoutant une colonne correspondant à l’index healpix de chaque objet astronomique.



Cet index healpix est calculé en fonction de la position de l’objet astronomique dans le ciel. Chaque numéro healpix correspond à une zone illustrés ci-contre. Ce système permet de réduire les calculs de jointure et de distribuer plus simplement le calcul. En effet, si on considère deux catalogues sur lesquels on souhaite

effectuer un crossmatch il sera possible de considérer chaque zone healpix séparément, chaque zone pouvant être traitée par un thread différent.



La seconde partie est la jointure en elle-même. La difficulté est qu'il n'y aura pas de correspondance exacte entre les tables puisqu'on considère des mesures instrumentales. La technique consiste donc a comparer les distances entre chaque objet et de considérer un certain seuil sous lequel les deux objets seront liés dans une table de jointure.

Sur le schéma ci-contre, les points verts correspondent aux objet du catalogue A et les points bleu les objets du catalogue B. On peut voir que le seuil de distance est illustré à l'aide d'un cercle tracé autour des objets du catalogue A.

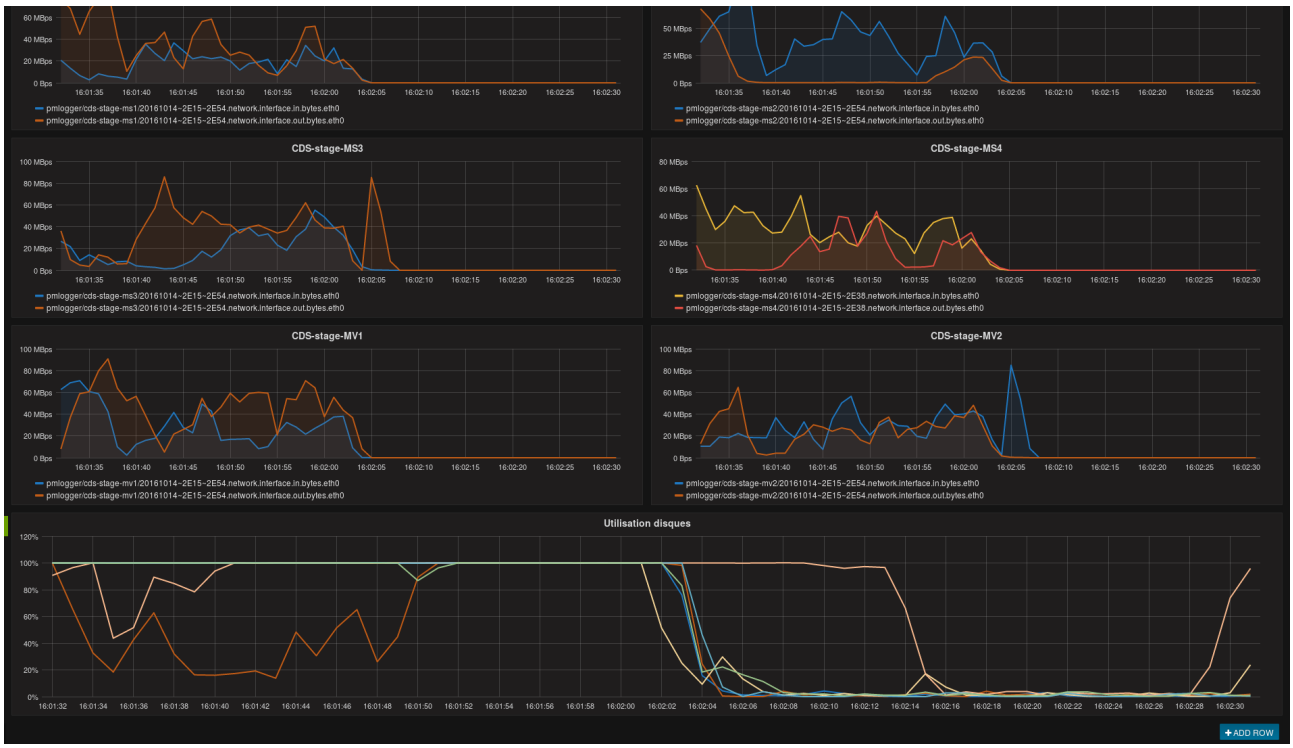
Étant donné qu'il faut comparer tous les objets entre eux on comprend l'intérêt d'effectuer un premier tri à l'aide des indices healpix.

ii. Monitoring des taches

Afin de visualiser l'utilisation de Spark il est possible d'utiliser l'interface web intégrée a Spark qui est accessible dans notre configuration en passant par le proxy Squid.

On peut y voir les différents workers ainsi que leur statut et la charge de travail qui leur est attribué. Cette interface permet aussi de consulter l'évolution d'une application en cours d'exécution et une liste de celles qui sont terminés.

Un autre outil que j'ai utilisé pour monitorer l'utilisation du cluster est Graphana qui est une interface web open source permettant de visualiser différentes composantes des machines comme l'utilisation CPU, RAM et réseau



Capture d’écran de Graphana illustrant l’activité réseau des six machines et la charge processeur en temps réel.

iii. Développement en Java

Problèmes de version Spark

La dernière version de Spark au moment de mon stage était la 2.0, or l’application développée auparavant au CDS était conçue pour la version 1.5 ce qui a posé des problème de compatibilité. En effet, certaines méthode ayant changées il a fallu que je fasse des recherche dans la documentation officielle de Spark (très peu explicite) afin de trouver ce qui n’allait pas. L’erreur provenait au final d’une méthode qui prenait un type différent d’objet en argument.

Résolution de bugs Docker

Au cours de mes tests de Spark dans Docker il m’est arrivé plusieurs fois d’avoir des problème de connexion entre les nœuds. Après une recherche parmi les différentes cause possibles, l’origine du problème venais du résolveur de noms intégré à Docker. Pour communiquer entre eux les services connectés eu réseau Docker communiquent entre eux en faisant référence directement au nom du service, par un exemple un spark worker se connectera a spark-master qui devra être résolu par l’IP réelle du service. Il se trouve donc que dans les premières version de Docker ce système de DNS avait tendance à ne pas fonctionner.

Docker étant un projet open source j'ai pu créer une issue GitHub sur laquelle j'ai pu déclarer ce bug et donner différentes information sur la façon de le reproduire. Ce bug fut résolu au bout de quelques semaines. J'ai ainsi pu découvrir comment s'organise un projet open source de grande ampleur.

Optimisation de l'image Docker

L'image Docker présentée précédemment n'est pas celle que j'ai utilisé dès le début. Comme expliqué il existe différents moyen d'optimiser une image Docker que j'ai pu découvrir au fil de mon développement et de mes tests. J'ai donc trouvé différents moyens d'obtenir des images plus petites et d'automatiser la création des images à l'aide d'un Makefile.

iv. Réécriture du projet Spark en Scala

Le framework Spark est écrit en Scala mais propose aussi une API Java. Le problème est que cette API propose moins de fonctionnalités comme par exemple la sauvegarde de résultats en format Parquet.

Le format Parquet est est format binaire permettant de compresser des tables par colonne. Ce système est intéressant si on possède des tables comprenant plusieurs millions de lignes avec des valeurs susceptibles de se répéter, comme par exemple des position d'objet astronomiques. Pour tester l'efficacité de ce format il a donc fallu réécrire le code Java en Scala.

Cette conversion ne se fait pas de manière triviale, bien que basé sur le Java, le langage Scala possède ses spécificité comme l'utilisation de variables sans type un peu dans le même esprit qu'en Python. Ce langage étant nativement compatible avec les librairies Java il est tout à fait possible d'importer les mêmes librairies de fonctions. L'API Spark fonctionne elle aussi de manière différente, la ou les données sont accessibles par un objet RDD en Java il faut utiliser un Dataset en Scala. La documentation nous informe que le Dataset utilise en fait des RDD mais permet une utilisation plus haut niveau. Un Dataset permet de simplifier certaines étapes comme l'ajout de colonnes qui se fait avec une méthode la ou avec l'API Java il faut passer par la création d'un nouvel objet.

Cependant une des méthodes de l'API Java n'a pas d'équivalence exacte avec l'API Scala, il s'agit de la méthode de répartition. C'est à dire qui va hasher une colonne donnée afin de placer les lignes possédants la même valeur dans cette colonne dans la même partition. La différence est que la fonction de hash utilisée est différente en

Java et en Scala. En Java le hash correspondant a un nombre entier est le nombre entier lui même modulo le nombre de partitions désiré, ainsi pour six partitions, la partition 1 contiendra les indices 1, 7, 13 … et ainsi de suite. Alors qu'en Scala cette continuité n'est pas garantie et il n'est pas possible de connaître à l'avance dans quels partition seront placés tels indices. Ceci posera problème lors de la conception du programme de colocation des données.

v. Collocation des données

Afin d'accélérer les calculs de Spark j'ai développé un programme permettant la collocation des données.

Le fonctionnement normal du preprocessing ne permet pas de garantir que les mêmes partitions seront stockées sur les mêmes Datanode HDFS. Au moment du crossmatch, quand il faudra accéder à ces données, il faudra transférer ces données par le réseau. Ceci ne pose aucun problème dans le cas ou le réseau est plus ou aussi rapide que le stockage. C'est le cas par exemple d'un réseau Gigabit (128 Mo/s) et d'un stockage des données sur disque dur (150 Mo/s en lecture séquentielle). Dans ce cas si on parvient a lire un maximum de données en local le gain ne sera pas très important. Contrairement a un stockage de type SSD (> 400 Mo/s même en lecture aléatoire) ou le gain sera bien plus élevé.

Les machines à ma disposition fonctionnant avec des SSD, j'ai développé un programme en langage Python chargé de répartir les partitions de sortes à ce que les mêmes numéro de partition se retrouvent sur les mêmes machines.

Fonctionnement

Le programme commence par interroger l'API du Namenode HDFS afin d'obtenir la liste des partitions et leur emplacement. Cette API n'est pas prévue pour ce genre d'utilisation et fournie une sortie formatée pour être lisible. Il n'existe pourtant pas d'autre moyen de récupérer ces informations ce qui aurait été pratique, avec une API de type JSON ou XML. Cette sortie est donc parsée à l'aide d'expressions régulières afin d'extraire les différentes informations et de les stocker dans un tableau Python.

Ensuite les services HDFS sont stoppés afin d'empêcher la modification du contenu des Datanode pendant l'exécution. Un serveur Web léger est ensuite démarré sur chaque nœud afin de rendre les partition accessible via une URL, cette solution permet d'aller plus vite que scp ou un rsync par SSH car il n'est pas nécessaire de

créer une connexion SSH pour chaque transfert, on peut donc utiliser un simple curl pour récupérer les partitions.

À chaque machine est ensuite associé un nombre, le numéro de partition modulo le nombre de machines nous donne donc la machine où placer la partition. Une boucle va ensuite itérer sur chaque partitions et vérifier qu'elle se trouve sur le bon nœud.

À la fin du programme les services sont redémarrés. Il n'est pas nécessaire de modifier la configuration du Namenode (le nœud qui orchestre HDFS) car les Datanodes vérifient les blocs à leur disposition et le rapportent au Namenode.

Tests

Une fois le programme fonctionnel j'ai testé l'efficacité de la méthode. D'après les résultats visibles ci-dessous on obtient un gain sur les temps de lecture par rapport au temps médian de 30 %. On peut aussi observer que la durée du crossmatch ne change pas, cela s'explique car à cette étape les partitions sont lues depuis la RAM au niveau des Spark Workers, ces données étant répartie correctement à ce moment la.

vi. Test de YARN

L'un des outils fournis par Hadoop et YARN. Cet outil permet de déployer un cluster Spark au moment de l'exécution d'une application et de le stopper une fois le programme terminé. Cette solution permet théoriquement d'indiquer à Spark l'emplacement des partitions et ainsi d'optimiser le lancement des taches.

En pratique ce système utilise plus de RAM, notamment car il nécessite l'exécution d'un service tournant en arrière plan. Ceci réduit donc les performances de Spark car il n'est plus possible de lui allouer toute la RAM.

vii. Tests sur un cluster de l'IN2P3

Au cours de mon stage l'IN2P3 à mis à disposition un cluster de 9 machines puissantes nous permettant d'effectuer des tests de plus grande ampleur. Ces machines comprenaient chacune 24 threads ainsi que 64 Go de RAM pour un total de 216 threads et 576 Go de RAM. Il nous a donc été possible de tester des crossmatch sur des fichiers contenant un milliard de lignes (impossible sur le cluster Spark utilisé au CDS pour des limitations techniques).

J'ai donc pu déployer un cluster Docker Swarm sur lequel j'ai démarré mes images. Cela m'a permis de vérifier que ces images fonctionnent indépendamment du

système hôte. Les machines du CDS fonctionnant sur Ubuntu et les machines de l'IN2P3 sont des machines virtuelles sous CentOS.

Performances

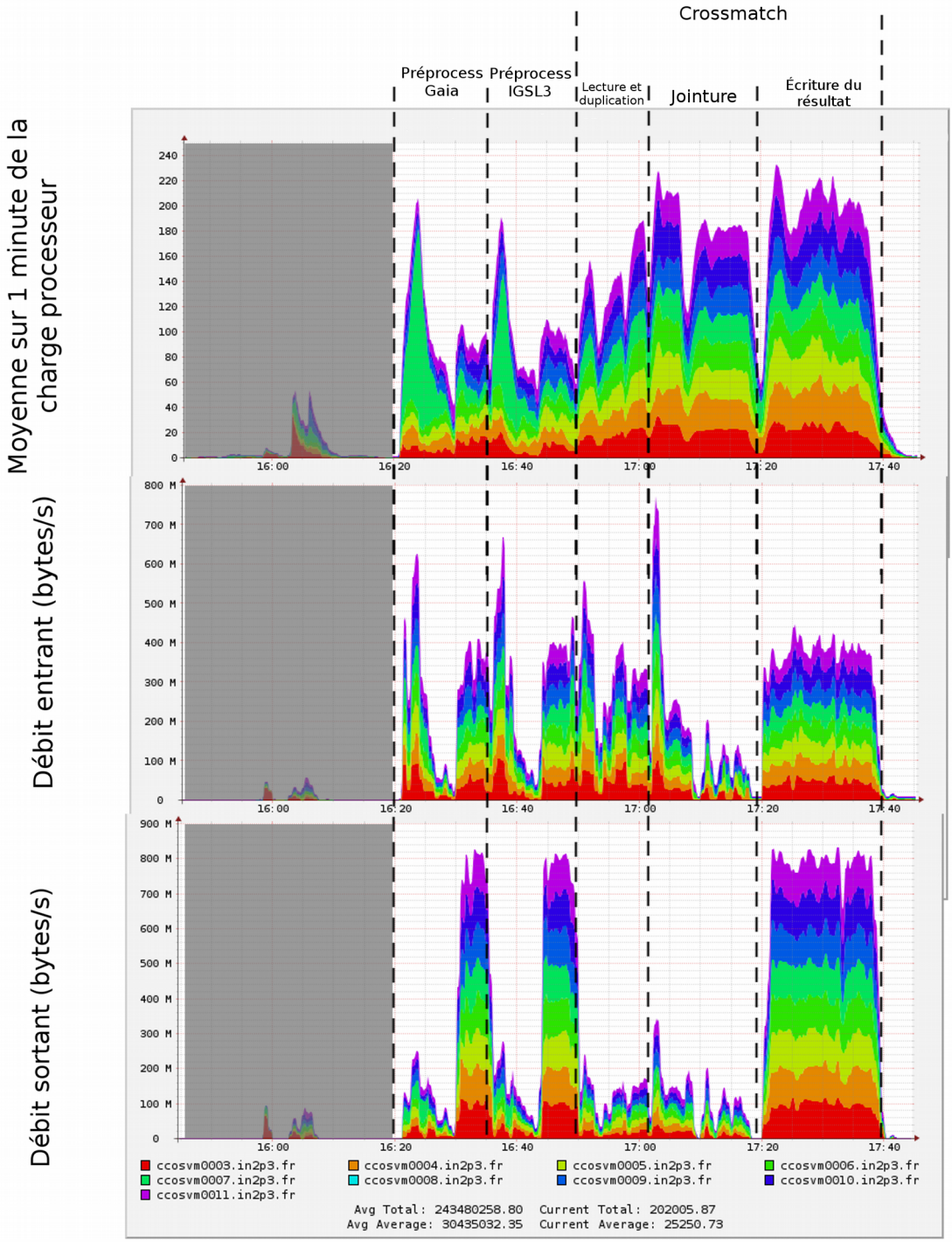
Chaque machine mis a disposition est équivalente à celle utilisée actuellement seulle pour faire les crossmatch en production. Nous avont donc voulu savoir par quel facteur le fait d'ajouter 8 machines diminuerais le temps d'execution. Il reste cependant une incertitude, les machines étant virtualisés il ne nous a pas été possible de connaître l'architecture exacte des processeurs utilisés ce qui pourrait avoir un impact sur les performances.

Après avoir lancé un test de crossmatch sur deux fichier contenant environ 1 milliard de lignes chacun, la différence de temps est d'un facteur 2. C'est à dire qu'il a fallu deux fois moins de temps à 9 machine pour effectuer le même calcul qu'une machine seule a effectué avec un code natif.

Cette différence peut sembler plutôt faible en considérant l'investissement nécessaire pour acheter 8 machines. Mais cela pourrait venir d'un facteur limitant, cela peut être le fonctionnement interne de Spark qui a forcément un impact sur les performances comparé à du code natif, ou encore le réseau qui est bien moins rapide que l'accès au disque locaux.

Afin de trouver une réponse j'ai observé l'utilisation des différentes composantes au cours du crossmatch.

Crossmatch entre Gaia et IGSL3



Ce graphique illustre les différentes étape du crossmatch afin de pouvoir déterminer les facteurs limitants. On peut observer par exemple que durant la jointure le réseau

ne fonctionne que durant un cours instant au début ce qui correspondrait au shuffle (redistribution des données)

C) Dockerization VizieR

Un autre projet réalisé durant ce stage à été la création d'une image Docker contenant le service VizieR, ceci afin d'automatiser les mises a jour des diverse dépendances et d'automatiser le déploiement du service sur différents serveurs.

Ce travail a été réalisé en collaboration avec Gilles Landais qui est actuellement le développeur principal de VizieR.

Définition des objectifs

La première étape dans la création de cette image à été de discuter des possibilités de Docker avec Gilles Landais et de définir quels sont les problèmes à résoudre. Au lancement de ce projet j'étais le seul a connaître le fonctionnement de Docker, il a donc fallu expliquer les différents aspects de ce logiciel afin de pouvoir discuter des aspects techniques du travail.

Les objectifs à remplir ont donc été :

- Le déploiement de VizieR sur les différents miroirs
- L'automatisation de l'installation (notamment la compilation et l'installation des dépendances)

Tests locaux

Une fois les objectifs définis j'ai pu tester une installation étape par étape du service sur mon poste. Cela m'a permis de lister les différentes dépendances nécessaires et les commandes permettant la compilation des différentes dépendances. Une configuration spécifique d'Apache à été a prendre en compte puisque le service utilise différentes composantes comme les CGIs.

L'installation se déroule en plusieurs étapes :

- Transfert des différentes dépendances et des scripts CGI par Rsync ou scp
- Installation des dépendances avec le gestionnaire de paquets
- Compilation des dépendances uniquement disponible sous forme de sources (typiquement les programmes développés en interne au CDS)
- Copie des fichier CGI et configuration de Apache

- Démarrage de Apache

Création d'une image

L'installation du service maîtrisée j'ai pu créer une image en me basant sur les notes de l'installation précédente. Dans un premier temps cette image ne sera pas optimisée pour Docker afin d'obtenir rapidement un exemple fonctionnel. Par exemple les fichiers sources seront copiés depuis la machine hôte ainsi que les différents scripts CGI. Ce qui résulte évidement en une image très lourde et peu efficace à déployer, cependant rapide à créer.

Il reste encore à résoudre les problèmes lié à l'exécution du service dans Docker. Principalement des paquets manquants, car l'image de base Docker ne comprend souvent que le strict minimum contrairement à une distribution de bureau standard. Des programmes comme gcc, make ou rsync ne sont par exemple pas installés par défaut dans l'image de base Apache httpd.

Amélioration de la création de l'image

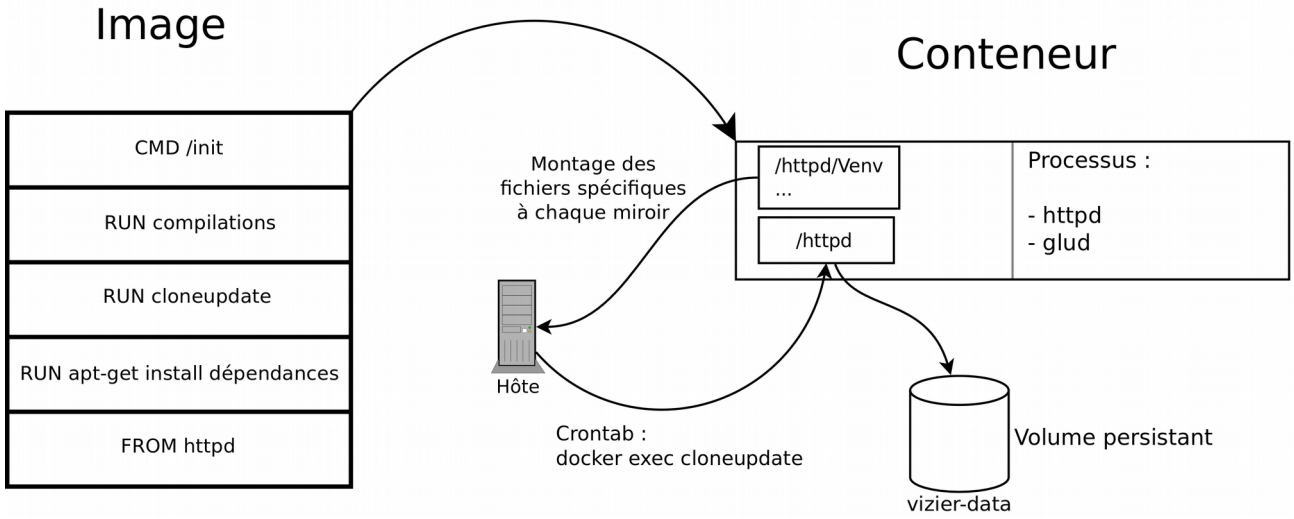
L'image étant fonctionnelle il est maintenant possible d'améliorer sa création.

La première tache fut de récupérer les sources des différents programme depuis CVS, qui est un système de gestion de version utilisé avant Git. Dans ce cas, contrairement à l'image Spark il est possible de tirer parti du cache en séparant chaque dépendance dans une couche. Cela permet en cas de mise à jour d'un programme de n'avoir à recompiler que les suivants (une modification du Dockerfile entraîne l'invalidation du cache lié à cette ligne et à toutes les lignes suivantes)

Ensuite, concernant les scripts CGI et les différents fichiers statiques il n'est pas intéressant de les placer dans une couche séparée de l'image (tous les fichiers combinés font environ 300 Mo). Car en cas de mise à jour d'un seul de ces fichier il faudrait renvoyer l'intégralité de cette couche sur l'ensemble des miroirs, ce qui ne serais absolument pas efficace étant donné que ces fichiers peuvent être modifiés tous les jours. La solution retenue a donc été de stocker ces fichiers dans un volume Docker, c'est à dire un point de montage entre la machine hôte et le conteneur ce qui permettra de conserver le contenu de ce dossier entre les exécutions de ce conteneur. La mise à jour de ces fichiers sera assurée par un script déjà utilisé pour mettre à jour les miroirs actuels, ce script sera exécuté à chaque démarrage du conteneur et par un service cronjob exécuté depuis le système hôte.

Cependant j’ai du adapter certains scripts afin qu’ils fonctionnent dans mon installation Docker (principalement des chemins à corriger). Il a donc fallu créer des script sed afin de remplacer ces chemins automatiquement après l’application de mises à jour.

L’une des fonctionnalités de VizieR est de pouvoir accéder aux grand catalogues (les catalogue comprenant plusieurs millions de lignes et provenant d’observation récentes de satellites). Ces catalogues sont accessibles par le biais d’exécutables spécifiques à chaque catalogue. La commande à exécuter diffère si le catalogue est accessible localement ou si il faut chercher les données de ce catalogue sur un autre miroir. Le fonctionnement passait donc par un script exécuté régulièrement par un cronjob qui vérifiait si ces catalogues étaient accessibles ou non puis créait des liens pointant vers la bonne commande. Le problème de ce système est qu’il nécessite un processus en arrière plan (crond) en plus de apache, or les conteneurs Docker sont optimisés pour l’utilisation d’un seul processus (et ses enfants) par conteneur. J’ai donc développé un script bash qui va vérifier la présence de ces catalogue au moment de l’exécution des requêtes.



Le schéma ci dessus illustre le fonctionnement de l’image Docker de VizieR. A gauche l’image est représenté par un empilement des différentes couches présentes et à droite le conteneur et les différentes relations avec le système hôte (points de montage, commandes)

Interface de gestion

Ce système fonctionne très bien quand on maîtrise la ligne de commande Docker. Cependant, il peut arriver que la maintenance de VizieR soit temporairement confié à d’autre personnes qui ne connaissent pas forcément l’environnement Docker. J’ai donc étudié les différents moyen de mettre à disposition une interface claire

permettant à n’importe qui de visualiser l’état des services et donner la possibilité de les redémarrer simplement à l’aide de boutons. Il existe différentes interfaces web encore en développement dont Docker Datacenter (logiciel propriétaire développé par Docker), Portainer et Rancher pour citer les plus complètes et avec des développeurs actifs. Mon choix s’est porté sur Portainer car il proposait le fonctionnement le plus basique, sans services supplémentaires. La plupart des fonctionnalité des l’API Docker sont disponible depuis l’interface Portainer ce qui permet d’effectuer les mêmes actions en ligne de commande que sur l’interface web.

Conclusion

Durant ce stage j’ai pu découvrir différents aspect de la vie professionnelle, autant sur le côté technique que sur le côté humain.

J’ai ainsi pu mettre en application des technologies comme Docker que je n’avais jusqu’alors utilisé qu’à très petite échelle. L’équipe du Centre des Données astronomique de Strasbourg m’a très bien accueilli et m’a soutenu durant le développement de ces projets.

Comme aspect critique du logiciel Docker, je pense qu’il n’est pas encore prèt à être utilisé à grande ampleur. Notamment par la nécessité d’avoir des systèmes récents et à jour pour pouvoir l’utiliser.

Plusieurs perspectives sont maintenant possibles. Comme par exemple l’utilisation de Docker dans le processus de développement des autres applications du CDS. Les travaux réalisé ayant mis en place des bases réutilisable pour d’autres projets.

Des tests seraient aussi envisageable sur une architecture de type cloud, c’est à dire en démarrant une centaine de petits nœuds à la demande, afin d’évaluer la rentabilité d’un tel système.

Bibliographie

Sites officiels de l'Observatoire astronomique de Strasbourg

Site principal : https://astro.unistra.fr

Portail du CDS : http://cds.u-strasbg.fr

Technologies utilisées

Docker : https://www.docker.com

Hadoop : https://hadoop.apache.org

Spark : https://spark.apache.org

Scala : http://www.scala-lang.org

Apache : https://www.apache.org

GlusterFS : https://www.gluster.org

Drone : http://try.drone.io

Jenkins : https://jenkins.io

Sites utilisés

Base de donnée de questions : http://stackoverflow.com

Registry officiel Docker : https://hub.docker.com

Mots clefs

Fonction publique - Astronomie - Recherche - Informatique - Bases de données

TREHIOU Paul Rapport de stage ST40 - A2016

Résumé

Ce rapport résume 6 mois de stage passé au Centre de Données astronomique de Strasbourg, qui a pour mission de stocker des informations sur des observations astronomiques et de proposer des services pour effectuer des traitements

En collaboration avec mes tuteurs de stage, j'ai mis en place des services en utilisant Docker, j'ai aussi du effectuer des développements autour d'une application de cross-match permettant d'effectuer une jointure entre deux tables de données astronomiques. J'ai finalement aussi développé une image Docker du service VizieR permettant des mises à jour et un déploiement plus facile.

Ce rapport décrit donc les différentes taches et objectifs de mon stage et les difficultés rencontrées.

Observatoire astronomique de Strasbourg
11 rue de l'Université
67000 Strasbourg