**SUBMITTING A SPARK(Pyspark) JOB**

This blog is a deep dive from a data engineer perspective on “what really happens behind the scenes” after a submitting a Pyspark job through spark-submit.

Basically while designing a spark application in python , we follow below structure (a standard practice)

* All the shared python modules which contains generic function will be packed together, say shared.zip
* All the ETL jobs which are implementations of business rules will be packed separately, say jobs.zip
* All the open source third party libraries which are not installed on worker nodes of the cluster will be packed as another python package, say libs.zip
* A driver python file which will be entry point to every spark ETL job, say main.py
* And finally a shell script which contains the spark-submit command (and other wrapper processing activities which are pre-requisites before submitting the spark application)
* Any configuration files (like conf.ini, conf.yml etc) will kept separate in a configuration directory

A sample spark submit command with all above resources mentioned will look like this (I am intentionally skipping resource configuration values here, since that will not be in scope of discussion under this blog )

***spark-submit --master <cluster manager> --deploy-mode <deploy mode> --files <configuration files > --py-files jobs.zip, shared.zip, libs.zip main.py <arguments to driver if any>***

***Some higher level idea what a master and deploy-mode is***

***--master:***

Since spark is distributed computing framework , it comes by default with a cluster manage called standalone cluster manager. But in real time use cases, it is more preferred to use more featured and standardized cluster managers such as ‘YARN’ or ‘MESOS’. And most famous among them is YARN. So here we will proceed with **--master** as **yarn**

**--deploy-mode:**

Spark has two deploy modes

* ***client****: driver process runs as a separate JVM on client node (edge node) from where the spark-submit has been triggered. If any failure to client node results in failure of spark application. Usually this mode prefers at the time of development and testing , especially while running spark jobs in jupyter notebooks, spark shells etc.*
* ***cluster*** *: driver process runs in a separate container on one of the data nodes decided by yarn resource manager. Most of the production scenarios use this deploy mode considering high availability of driver (there by spark application ) since if the container in which driver runs fails , resource manager restarts the driver program in a new container.*

These definitions are very high level. We will dive deep into it later.

Client Mode Spark Submit

1. Once we fire the spark-submit with **--deploy-mode client ,**Spark context initialization starts.
   * Spark context initialization includes creation of
     1. Task Scheduler
     2. Scheduler Backend
   * YarnClientSchedulerBackend is the SchedulerBackend for Spark on YARN for client deploy mode.
   * When YarnClientSchedulerBackend starts, it creates a new instance of Client(yarnClient) and executes submitApplication process.
   * submitApplication submits a Spark application to a YARN cluster (i.e. to the YARN ResourceManager) and returns the application’s ApplicationId.
   * When a Spark application is submitted to YARN, it calls the private helper method createContainerLaunchContext that creates a YARN ContainerLaunchContext request for YARN NodeManager to launch ApplicationMaster (in a container).
   * Application master will be launched in one of the worker nodes/data nodes inside a container with following specifications
     + AM memory will be equals [spark.yarn.am.memory](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-settings.html#spark.yarn.am.memory)
     + AM memory overhead will be  [spark.yarn.am.memoryOverhead](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-settings.html#spark.yarn.am.memoryOverhead) (or the maximum of 10% of amMemory and 384 is chosen)
     + Number of cores will be [spark.yarn.am.cores](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-settings.html#spark.yarn.am.cores)
   * When [created](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-applicationmaster.html#creating-instance) ApplicationMaster class is given a [YarnRMClient](https://mallikarjuna_g.gitbooks.io/spark/content/yarn/spark-yarn-applicationmaster.html#client) (which is responsible for registering and unregistering a Spark application).
2. When you do a spark-submit with **--deploy-mode client*,*** *there you are*  submitting the spark application in client mode. This means driver process which is the core of your spark application will be launched in an Isolated JVM in your client machine from where your have submitted the job.
3. Once we fire the spark-submit , it starts a YARN client program first
4. Spark Context will check the version of the spark and will submit the application to cluster manager
5. Now SecurityManager will comes into picture. Security manager controls the access to running application (application UI,history UI etc). This will validate the users permission and access control list defined for the job for the running application. spark.ui.filter and spark.acls.enabled few of the major properties w.r.t SecurityManager. More information can be found here

<https://spark.apache.org/docs/2.0.0/security.html#authentication>

**extra note:**

If we take real time business hadoop clusters , every organization will have some kind of mechanism which controls access to services in a Hadoop cluster. The authentication and authorization to HDFS services are defined through these controls.

One such famous example is Kerberos authentication on HDFS cluster level. To access HDFS paths and all, organization can set Kerberos authentication for users so that access to such clusters will be first validated through Kerberos KDC (key distribution center)

We can integrate this cluster level authentication filter to spark for every submitted spark application through a property called spark.ui.filter and setting spark.acls.enabled to true.

ACLs are access control list which decides which user/group can view or modify the job.

We can set cluster level authentication filter to spark.ui.filter as follows

spark.ui.filter=org.apache.hadoop.security.authentication.server.AuthenticationFilter

And if we are using kerberos , this server level filter should be mapped to kerberos on cluster level configuration.

More information on these properties can be found in below spark documentation :

<https://spark.apache.org/docs/2.0.0/configuration.html#security>

<https://spark.apache.org/docs/latest/security.html#authentication-and-authorization>

1. Once Security Manager authenticate and authorize by setting the access controls to application, Utils class will start the driver service named ‘sparkDriver’ on the client machine (client node/edge node/gateway node from which we are submitting the application). sparkDriver service is a network addressable service which means , it will have a host name and port to address . Host name is defined by property spark.driver.host and port value is defined by spark.driver.port. Host value is the ip address or domain address of the node where the driver service is running. Port is by default a random generated one. But If we want to restrict it to a range , it can be done by setting below properties

spark.driver.port=<some number>

spark.port.maxRetries=<some number>

so the range in which , spark will try to open port for driver process is between spark.driver.port and spark.driver.port + spark.port.maxRetries