Leveraging On-Prem Spark Cluster Option Analysis

How can we operate a spark cluster installed on-prem from clients like airflow and JupyterLab running inside an AKS cluster in the cloud.

There are two facet to this problem; how to develop the code of the ingestion jobs and how to schedule these jobs. Ideally the former would be done interactively that is having the ability to print results and easily get at execution errors. The latter does not necessitate interactivity no data will be brought back to the client but having access to execution logs is important.

The initial idea was to leverage the spark-submit in `--deploy-mode cluser` for Airflow and to use the `client` mode for JupyterLab interactive jobs.

Although it possible to stage and submit Java/Scala jobs in cluster mode it's not possible with python jobs.

We then look at Livy which enables submitting python jobs to a server. Although it requires some code to submit, monitor and acquires the logs of a remotely executed pyspark job it's possible to do it.

However when it comes time to do interactive jobs in JupyterLab it's another story. There is a project called spark-magic which helps submitting your queries to Livy and someone what hides the fact that you are using the Livy REST api to talk to spark it's still different. For example creating your spark context is done differently.

We then looked at exposing the ports required for a spark driver to stay inside JupyterLab and inside the Airflow client. We were able to expose ports via the nginx ingress which means the same IP use for all of Hogwarts and thus easily accessible can be given to an on-prem spark executor so it can call back to the driver. This solution has the advantage of not requiring any changes in the way we work with JupyterLab or Airflow.

	submit driver in cluster mode	Livy / Spark-magic	reachable driver client mode
feasibility	no	yes	yes
Airflow stays the same	no	no	yes
JupyterLab works the same	yes	no	yes
complexity to realize	n/a	medium	low
performance concerns	no	no	don't believe so

We don't foresee performance issues with a client driver mode solution. The reasoning is that the executors do not send data to the client unless the user ask for data to be transferred to the client for example to print a dataframe or to convert it to a pandas dataframe. Ingestion jobs mostly exchange data between the executors and write them to disk.

Even if a user asks to retrieve data from the server it's believe that transfer would still be quite performant.

These are the options when using spark-submit. You can stage jars, files and run in client or cluster mode.

```
--py-files PY_FILES         Comma-separated list of .zip, .egg, or .py files to place
                              on the PYTHONPATH for Python apps.
--files FILES               Comma-separated list of files to be placed in the working
                              directory of each executor. File paths of these files
                              in executors can be accessed via SparkFiles.get(fileName).
--archives ARCHIVES         Comma-separated list of archives to be extracted into the
                              working directory of each executor.
--deploy-mode DEPLOY_MODE   Whether to launch the driver program locally ("client") or
                              on one of the worker machines inside the cluster ("cluster")
                              (Default: client).

```

You can also monitor a driver you submitted to the cluster using these commands

```
spark-submit --status [submission ID] --master [spark://...]
spark-submit --kill [submission ID] --master [spark://...]
```

However in order to be able to use these management commands you need to use the REST endpoint to submit your job. To do so you need to set `spark.master.rest.enabled` to true in your spark master.

When the master starts you should see this in the logs

```
21/09/30 05:33:12 INFO Utils: Successfully started service on port 6066.
21/09/30 05:33:12 INFO StandaloneRestServer: Started REST server for submitting applications on port 6066
```

Need to change submission port to 6066 in order to use HTTP post submission

```
./bin/spark-submit --master spark://Jean-Claudes-MacBook-Air.local:6066 --deploy-mode cluster --class org.apache.spark.examples.JavaSparkPi  --verbose


```

```
21/09/30 05:38:06 DEBUG RestSubmissionClient: Response from the server:
{
  "action" : "CreateSubmissionResponse",
  "message" : "Driver successfully submitted as driver-20210930053806-0003",
  "serverSparkVersion" : "3.1.2",
  "submissionId" : "driver-20210930053806-0003",
  "success" : true
}
21/09/30 05:38:06 INFO RestSubmissionClient: Submission successfully created as driver-20210930053806-0003. Polling submission state...
21/09/30 05:38:06 DEBUG RestSubmissionClient: Response from the server:
{
  "action" : "SubmissionStatusResponse",
  "driverState" : "SUBMITTED",
  "serverSparkVersion" : "3.1.2",
  "submissionId" : "driver-20210930053806-0003",
  "success" : true
}

```

using curl to get the status

```
$ ipconfig getifaddr en0
10.0.0.14
$ curl <http://10.0.0.14:6066/v1/submissions/status/driver-20210930054249-0004>
{
  "action" : "SubmissionStatusResponse",
  "driverState" : "SUBMITTED",
  "serverSparkVersion" : "3.1.2",
  "submissionId" : "driver-20210930054249-0004",
  "success" : true
}

```

The `--status` command does not return the logs of the driver.

You can try to configure log4j to send logs to a known location. However even when you do so you will not see stdout/stderr of your pyspark. Only what you write to logging.

Suppose you have a `driver-log4j.properties` file like this writing to a well known location.

```
log4j.rootCategory=INFO,FILE
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n
log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.File=/tmp/SparkDriver.log
log4j.appender.FILE.ImmediateFlush=true
log4j.appender.FILE.Threshold=debug
log4j.appender.FILE.Append=true
log4j.appender.FILE.MaxFileSize=500MB
log4j.appender.FILE.MaxBackupIndex=10
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

```

You can place the `driver-log4j.properties` in a central location, or stage it.

```
./bin/spark-submit --master spark://Jean-Claudes-MacBook-Air.local:6066 --deploy-mode cluster --class org.apache.spark.examples.JavaSparkPi \
--conf spark.driver.extraJavaOptions="-Dlog4j.debug -Dlog4j.configuration=file://./driver-log4j.properties"  \
--files ./conf/driver-log4j.properties \
./examples/jars/spark-examples_2.12-3.1.2.jar
```

But still this does not give you stdout/stderr.

The best way to get at the driver logs is probably via the Spark UI or the history server UI. Both have the driver log even after the driver has finished executing the job.

Logs from Spark UI

On the main spark UI page you see the submitted drivers (running and completed)

In both cases you can obtain the Worker they are running on. for example worker-20210930060341-10.0.0.14-54163

When you click on the link it takes you to the worker for example http://10.0.0.14:8081/
On the worker UI page you see the Finished Drivers with their stdout/stderr links

```
<http://10.0.0.14:8081/logPage/?driverId=driver-20210930060836-0003&logType=stdout>
<http://10.0.0.14:8081/logPage/?driverId=driver-20210930060836-0003&logType=stderr>
```

Logs from History server

Accessible via history server (stdout and stderr of driver)

```
<http://10.0.0.14:8081/logPage/?driverId=driver-20210930060836-0003&logType=stdout>
<http://10.0.0.14:8081/logPage/?driverId=driver-20210930060836-0003&logType=stderr>
```

But all of this is in vain because the fundamental issue with spark-submit is that it does not suppor python submissions.

Java submission work in cluster mode

```
./bin/spark-submit --master spark://Jean-Claudes-MacBook-Air.local:7077
 --deploy-mode cluster --class org.apache.spark.examples.JavaSparkPi
 ./examples/jars/spark-examples_2.12-3.1.2.jar
```

Python submission do not work in cluster mode

```
./bin/spark-submit --master spark://Jean-Claudes-MacBook-Air.local:7077
--deploy-mode cluster examples/src/main/python/pi.py
exception in thread "main" org.apache.spark.SparkException:
Cluster deploy mode is currently not supported for python applications on standalone clusters.
```

Livy

We thus look at Livy as a means to submit python jobs to a server sitting inside the same cluster as the spark master and spark workers.

But this introduces a brand new server with it's own REST API and thus you have to change your client to interface with it. So it would require a fair amount of work in Airflow.

You still have the issue of packaging the python client code so it can be staged into the server.

You need to build Livy from source because they only build it for spark 2.4. It is supposed to work with spark 3 but I did not finished building it and trying it out because I looked at how Livy would be use in the JupyterLab context.

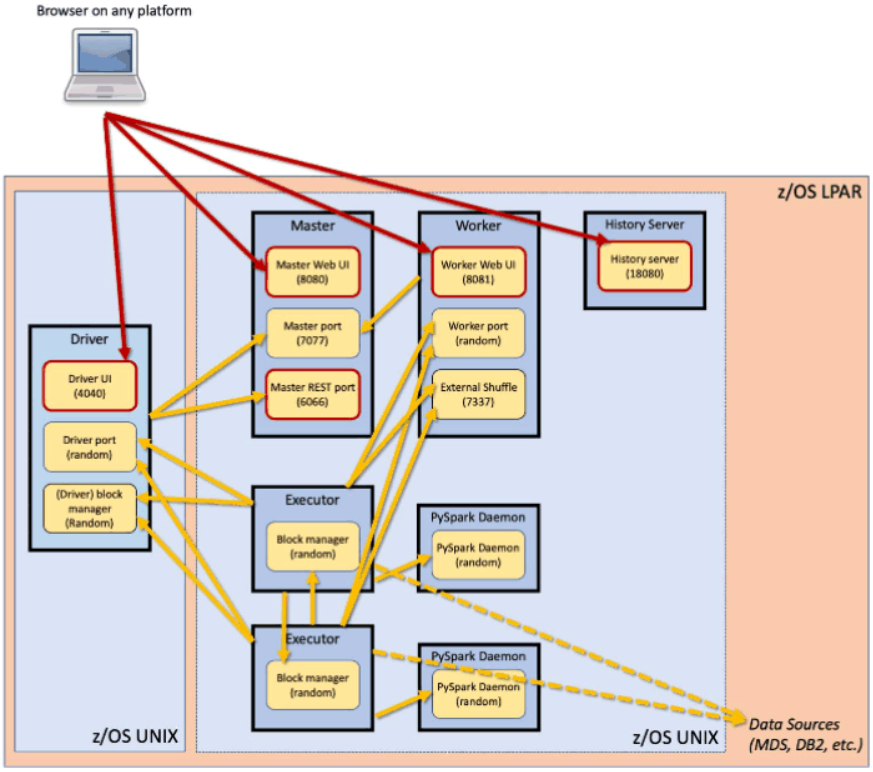
In JupyterLab you need to use spark-magic if you want to have anything resembling how we use JupyterLab with a client side driver.

Everything is based on magic. To run your code on the server-side you place it in a `%%spark` cell.
To see the logs you can issue a `%%info`. To create a spark context you use `%manage_spark` which lets you see your existing spark context on the server and brings up a UI to configure and create new spark context. They have utilities to send files to the server or send local Dataframe to the server.
So although they have all the commands and utilities to work with a remote spark context in JupyterLab you have to be aware that you are a client and that everything is really happening on the server that is where the pyspark driver and the python kernel are running. So the user experience is very different than having the spark driver in your client.

Exposing Spark Driver to On-Prem Spark Cluster

This article clearly explains all the ports used by spark and how to configure them.

Configuring networking for Apache Spark

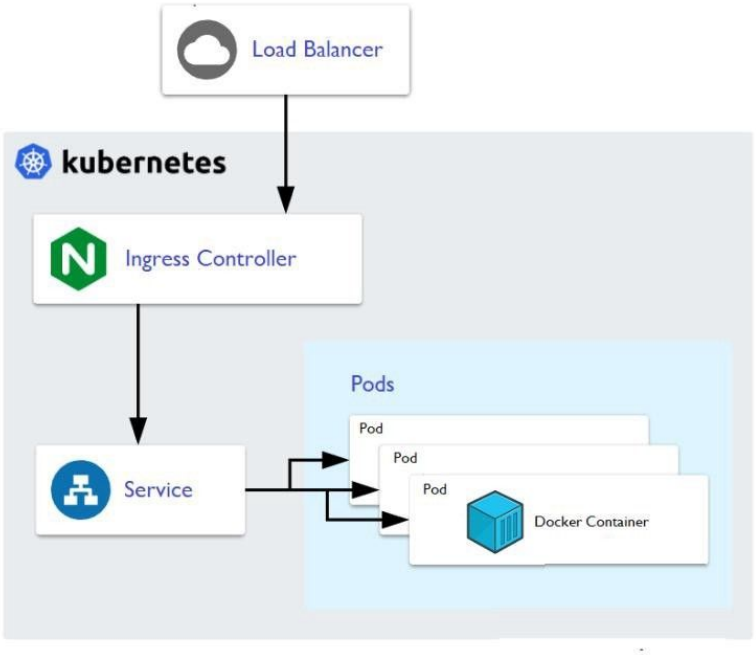


Topology of Incoming TPC Connection

The diagram above clearly shows that the Executors need to make a connection back to the driver on 2 different ports. Additionally if we want to reach the application UI, we need to reach the driver UI port. The 3 ports to expose are

- spark driver UI
- spark driver control plane
- spark driver data plane (block manager)

We can achieve this by placing a kubernetes service `spark-driver-service` in front of the PODs we want to host a spark client driver. Then configure the nginx-ingress to forward connection to certain ports to this `spark-driver-service`. When a POD starts a spark driver client it thus opens a port to listen to. Kubernetes service are smart enough to handle the fact that it is routing connections into multiple PODs only one of which is actually listening. This is the same scenario as fronting a farm of webservers and having some of these webservers being down.



We already have a reachable nginx-ingress

We introduce a new service named `spark-driver-service`

Which is configured to front a selected set of PODs using Kubernetes `selector` and `matchExpressions`

Service in front of JupyterLab pod

```
kubectl -n jupyhub apply -f .\service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: spark-driver-service
spec:
  ports:
  - name: spark-ui-0
    port: 11000
    protocol: TCP
  - name: spark-driver-0
    port: 22000
    protocol: TCP
  - name: spark-blockmanager-0
    port: 33000
    protocol: TCP
  - name: spark-ui-1
    port: 11001
    protocol: TCP
  - name: spark-driver-1
    port: 22001
    protocol: TCP
  - name: spark-blockmanager-1
    port: 33001
    protocol: TCP
  selector:
    matchExpressions:
    - key: hub.jupyter.org/username
      operator: In
      values:
        - jupyter-user-A-pod
        - jupyter-user-B-pod
        - jupyter-user-C-pod
```

tcp-serices config map

```
kubectl -n kubeprod edit configmap tcp-services-67c1890

data:
  11000: "jupyhub/spark-driver-service:11000"
  22000: "jupyhub/spark-driver-service:22000"
  33000: "jupyhub/spark-driver-service:33000"
  11001: "jupyhub/spark-driver-service:11001"
  22001: "jupyhub/spark-driver-service:22001"
  33001: "jupyhub/spark-driver-service:33001"
```

```
kubectl -n kubeprod patch configmap tcp-services  --patch '{"data":{"33000":"jupyhub/spark-driver-service:33000"}}'
```

nginx ingress

```
kubectl -n kubeprod edit service nginx-ingress

spec:
  ports:
  - name: spark-driver-ui-0
    port: 11000
    protocol: TCP
    targetPort: 11000
  - name: spark-driver-0
    port: 22000
    protocol: TCP
    targetPort: 22000
  - name: spark-block-manager-0
    port: 33000
    protocol: TCP
    targetPort: 33000
  - name: spark-driver-ui-1
    port: 11001
    protocol: TCP
    targetPort: 11001
  - name: spark-driver-1
    port: 22001
    protocol: TCP
    targetPort: 22001
  - name: spark-block-manager-1
    port: 33001
    protocol: TCP
    targetPort: 33001
```

```
kubectl patch service ingress-nginx-controller -n kube-system
 --patch "$(cat nginx-ingress-svc-controller-patch.yaml)"
```

listen for incoming connections in JupyterLab terminal

```
$ python -m http.server 11000
Serving HTTP on 0.0.0.0 port 11000 (<http://0.0.0.0:11000/)> ...
```

from client outside the cluster

```
curl <http://10.162.231.8:11000/>
```

any of the domain URL also work because they resolve to that IP

```
curl <http://spark.hogwarts.pb.azure.chimera.cyber.gc.ca:11000/>
```

Test setting custom callback ports but still using our POD IP address

```
./bin/spark-submit \
--conf spark.driver.bindAddress=0.0.0.0 \
--conf spark.ui.port=11000 \
--conf spark.driver.port=22000 \
--conf spark.driver.blockManager.port=33000 \
--master spark://ver-1-spark-master-svc.spark:7077 \
examples/src/main/python/pi.py
```

Ask executors to connect back via the nginx ingress controller IP

```
./bin/spark-submit \
--conf spark.driver.host=10.162.231.8 \
--conf spark.driver.bindAddress=0.0.0.0 \
--conf spark.ui.port=11000 \
--conf spark.driver.port=22000 \
--conf spark.driver.blockManager.port=33000 \
--master spark://ver-1-spark-master-svc.spark:7077 \
examples/src/main/python/pi.py
```