

Datos

Profesores de Prácticas: Sergio Villar, Carlos Montemuiño

Material de las prácticas

Para las prácticas, en las máquinas Linux, la documentación será los *HOWTOs* y *manpages* del sistema. La presente guía es orientativa (no funcionará un copy paste sin saber lo que se hace).

En las prácticas se usarán máquinas virtuales (*Virtual Machines VM*) con OpenNebula. A OpenNebula se puede acceder desde todas las máquinas del laboratorio. Existen varios templates con sistema operativo Linux ya instalado.

Para acceder al entorno de prácticas se ha habilitado un acceso remoto a los servicios OpenNebula en esta dirección:

<http://nebulacaos.uab.cat:8443>

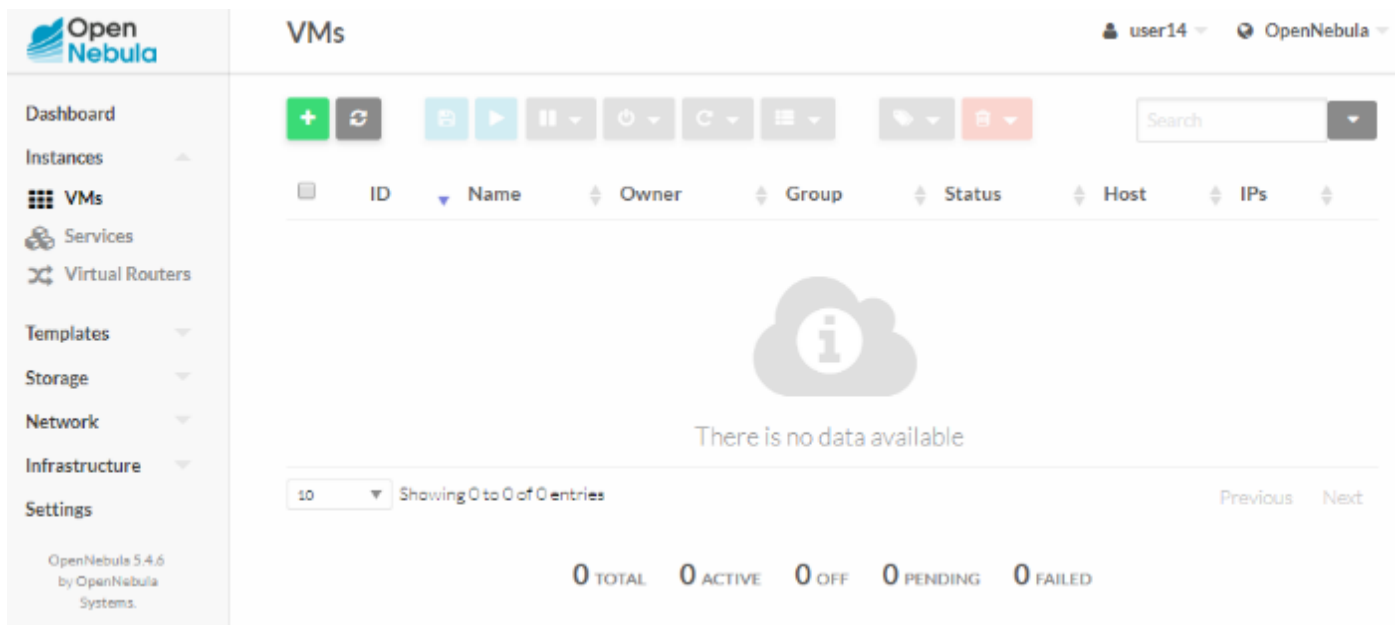
La documentación básica sobre el entorno Open Nebula y cómo utilizarlo puede consultarse en el manual de Operación: <http://docs.opennebula.org/5.10/operation/index.html>

1. Descripción del entorno

Para la realización de esta práctica diseñaremos un entorno con una única VM con una instancia de CentOS donde se irán instalando los programas necesarios.


2. Creación del entorno y las máquinas virtuales

Después de acceder a OpenNebula, en el menú de la izquierda se debe seleccionar *Instances* y en el desplegable seleccionar *VMs*. Así nos aparecerá una pantalla similar a la siguiente:




En esta pantalla, se debe clicar el botón verde (+) para crear una máquina y aparecerán los diferentes templates disponibles; de la lista se debe seleccionar el template Centos7-X

El nombre de la máquina virtual (VM name) será *Master* y no se deberá cambiar ningún otro campo.

En el apartado *Network* se debe añadir una interface de red con el botón . La interfaz debe ser conectada a la red con *Id 0* y *Name Internet*.

En este momento ya podemos crear la máquina virtual con el botón “CREATE” del inicio de la página. Como resultado, podemos ver la máquina Master creada en nuestra lista de VMs.

3. Acceso a las máquinas virtuales

Ahora entramos a la interfaz gráfica de la máquina *Master* pulsando el icono  de la derecha. En ese momento accederemos al interface de usuario de la máquina Master para terminar de configurarla. Para entrar en la máquina se debe utilizar el usuario *root* con la contraseña *NebulaCaos* seleccionando la opción *¿No está en la lista?* para poder entrar con el usuario *root*.

A continuación, crearemos un nuevo usuario llamado *alumno* en cada una de las máquinas virtuales. El usuario *alumno* debe ser creado mediante el comando *adduser*.

```
adduser alumno
```

Este comando nos creará el usuario y su correspondiente home, reduciendo así la cantidad de pasos a realizar para crear un usuario si se compara con el comando *useradd*. A continuación, le asignamos una contraseña mediante el comando *passwd*:

```
passwd alumno
```

Seguidamente, tenemos que darle permisos de sudo al nuevo usuario creado (`alumno ALL=(ALL:ALL) ALL`).

Tenemos dos alternativas:

- De forma apropiada con el comando visudo: <https://www.golinuxcloud.com/add-user-to-sudoers/>
- De forma más “temeraria” cómo lo hicimos en la práctica 1, modificando el fichero `/etc/sudoers`.

Ahora, para comprobar si los cambios se han llevado a cabo correctamente, abrimos un nuevo terminal. En el nuevo terminal cambiamos de usuario e intentamos actualizar la lista de paquetes disponibles en nuestra distribución Linux.

```
su - alumno
```

```
sudo yum -y update gedit java-1.8.0-openjdk wget
```

Nota:

También podéis hacer un logout del usuario “root” y hacer logging con el usuario “alumno”.

En cualquier caso, si el “yum update” se os queda bloqueado por mucho tiempo (> 2 minutos) por un “lock”, podéis probar de hacer lo siguiente con el usuario “root”:

```
systemctl stop packagekit
systemctl mask packagekit
killall -9 yum
yum remove -y PackageKit*
reboot
```

Referencia: <https://www.thegeekdiary.com/centos-rhel-7-how-to-enable-or-disable-automatic-updates-via-packagekit/>

4. Preparación del entorno de trabajo

Instalación de *Spark*. Primero, necesitaremos instalar los siguientes paquetes: Java JDK, Scala y Spark. Para instalarlos en la maquina local realizaremos los siguientes pasos:

Paso 1: Verificar en entorno OpenJDK.

Después de entrar al sistema con el usuario "alumno", vamos a verificar que el entorno de ejecución Java es 1.8 o posterior:

```
java -version
```

La salida debería reflejar una versión similar a la siguiente:

```
openjdk version "1.8.0_262"
OpenJDK Runtime Environment (build 1.8.0_1262-b10)
```

Opcionalmente, podríamos agregar las variables de entorno "JAVA_HOME" y "JRE_HOME" en el siguiente fichero de configuración situado en la carpeta home del usuario

```
gedit ~/.bashrc
```

```
JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk
```

```
JRE_HOME=/usr/lib/jvm/jre
```

Ahora **abrimos una nueva terminal** y evaluamos el valor de nuestras variables de entorno:

```
echo $JAVA_HOME  
echo $JRE_HOME
```

Paso 2: Instalar Scala

Ahora vamos a instalar el ultimo RPM de Scala en la rama 2.12 desde la web oficial, siendo en este momento la 2.12.12:

```
cd ~  
wget https://downloads.lightbend.com/scala/2.12.12/scala-2.12.12.rpm  
sudo yum install scala-2.12.12.rpm
```

Verificamos la instalación comprobando la versión de Scala:

```
scala -version
```

Y la salida debería ser algo muy parecido al siguiente mensaje:

```
Scala code runner version 2.12.12 -- Copyright 2002-2020, LAMP/EPFL  
and Lightbend, Inc.
```

Paso 3: Instalación de SPARK

Esta vez, nos vamos a basar en la distribución 3.0.1 de Spark desde la propia web del proyecto Apache:

```
cd ~  
wget http://www-eu.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-  
hadoop2.7.tgz  
tar -xzf spark-3.0.1-bin-hadoop2.7.tgz
```

Ahora creamos una nueva variable en *.bashrc* y modificamos la variable *PATH* para que apunte a nuestra versión local de *Spark*

```
gedit ~/.bashrc
```

y añadimos estas variables:

```
SPARK_HOME=/home/alumno/spark-3.0.1-bin-hadoop2.7
```

```
PATH=$PATH:$SPARK_HOME/bin
```

Ahora **abrimos una nueva terminal** y comprobamos que *Spark* se haya instalado bien iniciando su propia consola interactiva

```
cd $SPARK_HOME
./sbin/start-master.sh
spark-shell
```

Con `:help` podemos ver la lista de comandos disponibles y con `:quit` salimos de la consola *Spark*

Comprobamos que los servicios arrancados por *Spark* están bien abriendo un navegador y comprobando que existe la página: `http://localhost:8080`

Si todo está bien, ejecutamos una consulta a *Spark* para comprobar el buen funcionamiento del sistema. Lo que vamos a hacer es crear una secuencia de números. Esta estructura (`'data'`) se dice que está en el `'Driver'`, es decir. Luego, la copiamos a una estructura de datos distribuidas (RDD: *Resilient Distributed Dataset*) para que podamos operarla en paralelo (`'distData'`). Por último, le aplicamos la operación `'reduce'`, tomando los dos primeros elementos y los sumamos, y luego sumamos el resultado al tercer elemento, etc., hasta que llegamos al último elemento.

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at
parallelize at <console>:28

scala> distData.reduce((a, b) => a + b)
res1: Int = 15
```

5. Sesión interactiva con Spark

La primera abstracción que vamos a utilizar en *Spark* es una colección distribuida de elementos llamada *Dataset*. Los *datasets* pueden ser creados desde archivos HDFS (con el formato Hadoop InputFormat) o transformando otros *datasets* que ya tengamos en el momento. Vamos a crear nuestro primer *dataset* a partir del texto del archivo README que está en la carpeta de instalación de *Spark* (en `$SPARK_HOME`). Nuestro *dataset* se va a llamar `distReadme` y cada elemento de este *dataset* será una línea del fichero fuente.

```
scala> val distReadme = sc.textFile("README.md")
distReadme: org.apache.spark.rdd.RDD[String] = README.md
MapPartitionsRDD[1] at textFile at <console>:27
```

Ahora lo primero que vamos a hacer es contar las líneas de nuestro *dataset* y guardar este valor en una variable `"linesCount"`.

```
scala> val linesCount=distReadme.count  
linesCount: Long= 108
```

Después vamos a imprimir el primer elemento de nuestro *dataset* y guardarlo en otra variable

```
scala> val header=distReadme.first()  
header: String = # Apache Spark
```

Ahora vamos a empezar a aplicar transformaciones al dataset. El primero será un filtro. Vamos a llamar a la función `filter` para que nos devuelva un nuevo dataset donde sólo tengamos las líneas que contengan la palabra "Spark". ¿Cuántas líneas son?

```
scala> val linesWithSpark = distReadme.filter(line =>  
line.contains("Spark"))  
linesWithSpark: org.apache.spark.rdd.RDD[String] =  
MapPartitionsRDD[2] at filter
```

Acabamos de crear un nuevo dataset, *linesWithSpark*, que contienen solo aquellas líneas con la palabra "Spark". Para ello, hemos usado una función anónima del lenguaje de programación Scala cuando hemos escrito el símbolo "`=>`". Éste define una función sin nombre que obtiene un String y retorna un valor cierto cuando el String contiene "Spark" y un valor falso cuando no.

La función `filter` evalúa la función anónima (la de la flecha) para cada elemento del dataset y genera uno nuevo que contiene solo aquellos elementos donde la función anónima ha devuelto un valor cierto.

Por ejemplo:

Entrada	<code>=> line.contains("Spark")</code>	dataset de salida
this line is boring	false	
this line is also boring	false	
this line contains Spark	true	this line contains Spark
this line contains also Spark	true	this line also contains Spark

Para contar las líneas, podemos usar `count`, como en el ejemplo anterior:

```
scala> linesWithSpark.count  
res0: Long = 19
```

Típicamente, se encadenan las transformaciones (`filter`) y las acciones (`count`) en una misma línea sin usar variables:

```
scala> val numSpark=distReadme.filter(line =>  
line.contains("Spark")).count()
```

Cuando vamos a usar funciones en el código, podemos definir las del siguiente modo:

```
scala> val isSpark = (line : String) => line.contains("Spark")
isSpark: String => Boolean =<function>
```

Y, a partir de ese momento, usar esa función en lugar de las funciones anónimas como en el ejemplo anterior:

```
scala> val linesWithSpark2 = distReadme.filter(isSpark)
linesWithSpark2: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[5] at filter
scala> linesWithSpark2.foreach(x => println(x))
```

5. Transformaciones de datos con Spark

Si echamos un vistazo al mensaje que devuelve *Spark* se puede comprobar que *linesWithSpark* es una *dataset* especial: RDD (Resilient Distributed Dataset). Estas estructuras son colecciones de elementos donde se puede operar en paralelo en el cluster Spark donde tenemos distribuidos los datos con alta tolerancia a fallos.

Existen dos tipos de operaciones con estructuras RDD: las transformaciones y las acciones.

Transformaciones: producen un nuevo RDD al aplicar una serie de manipulaciones a un RDD fuente. Por ejemplo, al aplicar un filtro a nuestra lista de líneas.

Acciones: realizan un cierto cómputo sobre los elementos de un RDD y devuelven un cierto resultado. Como contar el número de elementos de un RDD.

Vamos a estudiar tres transformaciones básicas:

a) MAP

map: aplica una función a cada elemento de un RDD para producir un nuevo RDD.

Ejemplo: `distReadme.map(s => s.length)` toma como entrada un RDD con Strings y retorna un RDD con la longitud de las Strings.

b) FLATMAP

flatMap: similar a map, pero cada elemento a la entrada puede generar varios elementos en el RDD de salida. Ejemplo: `distReadme.flatMap(linea => linea.split(" "))` toma como entrada un String como

"Hola, estoy usando Spark ahora"

Y devuelve como resultado un RDD con los elementos:

("Hola,", "estoy", "usando", "Spark", "ahora")

c) REDUCEBYKEY

reduceByKey: cuando se le llama en un conjunto de datos tipo (clave, valor) o (C, V), devuelve un RDD con parejas (C,V) donde los valores para cada clave se han agregado usando una función del tipo (V, V) => V.

Ejemplo: `parejas.reduceByKey((a, b) => a+b)` procesa parejas (C, V) de forma que todas las parejas con la misma clave son transformadas a una única nueva pareja con la misma clave pero con la suma de todos los valores V. (Spark, 1) y (Spark, 15) dan como resultado una única pareja (Spark, 16)

Ahora vamos a ver en un ejemplo cómo contar las palabras de nuestro archivo de prueba:

```
val text_file= sc.textFile("README.md")
val lines = text_file.flatMap(line => line.split(" "))
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

O lo que es lo mismo:

```
counts=text_file.flatMap(line
=>line.split(" ").map(w=>(w,1)).reduceByKey((a, b) => a+b ))
```

¿Por qué no funciona el comando anterior? ¿Cuál es el error?

Ejercicios:

Definiendo nuestro vector de entrada:

```
scala> val data=Array(1,5,10,11,12,21,50,51,99)
```

Describe qué transformación o transformaciones hay que usar para resolver las siguientes preguntas:

1. Calcular el cuadrado de cada número de nuestro array data
2. Convertir el Array de enteros data a un Array de String y luego almacenar el inverso de cada String (reverse) en un nuevo RDD
3. Crear un nuevo RDD con los primeros 5 elementos del RDD anterior
4. Crear un nuevo RDD a partir de nuestro fichero README.md donde solo se almacenen las palabras de un texto sin repetición o duplicados. ¿Cuántas palabras son? ¿Qué porcentaje del total representan?

Transformaciones útiles: map, flatMap, filter, reduceByKey

Acciones útiles: count, collect, first, seq, top, distinct, reverse, take

La lista de transformaciones y acciones que se pueden usar en Spark es:

- <https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations>
- <https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions>