# LAB 2. Implementando un nodo con Multi-Broker.

## I. INTRODUCCIÓN.

En este segundo LAB vamos a realizar una configuración un poco más interesante en la que seguiremos teniendo un único nodo, pero varios Brokers.

El esquema del laboratorio que montaremos será el siguiente:

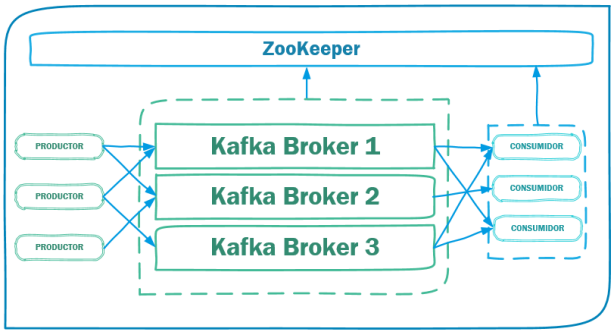


*Ilustración 1 Esquema general de un solo nodo y multi-broker.*

***Recursos utilizados para el laboratorio***:

- Máquina virtual Ubuntu 17.04 (2G RAM y 40 HD en thin provisioning).
- Java 1.8.0.131
- Confluent-oss-3.2.1-2.11

## II. Primeros pasos. Preparando el entorno.

Si ya has realizado el primer LAB no debería de hacerte falta, pero por si acaso, lo volvemos a poner….

Lo primero que vamos a hacer es instalar java ya que es el requisito principal que deberemos cumplir para poder levantar nuestro entorno de pruebas. En mi caso he optado por instalar la variante de java de Oracle, se puede usar cualquier variante siempre que sea > 1.7.

```
# add-apt-repository ppa:webupd8team/java
# apt-get update
# apt-get install Oracle-java8-installer
```

Comprobamos que todo ha ido bien mediante:

```
# java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

## III. Instalación inicial.

Para llevar a cabo la instalación vamos a utilizar un empaquetado de Apache Kafka creado por Confluent, empresa fundada por los desarrolladores originales.

Si no lo tenemos creado procedemos a crear el usuario "***kafkauser***".

```
$ sudo adduser --home /kafka kafkauser
$ sudo passwd kafkauser
```

Una vez creado el usuario, descargamos el software de Confluent:

```
$ wget http://packages.confluent.io/archive/3.2/confluent-oss-3.2.1-2.11.tar.gz
```

Una vez descargado, vamos a descomprimir el producto:

```
kafkauser@SERVER1:~$ tar xvf confluent-oss-3.2.1-2.11.tar.gz
kafkauser@SERVER1:~$ mv  confluent-3.2.1 confluent
```

Con estos sencillos pasos ya estamos preparados para comenzar a jugar.

## IV. Levantando el entorno.

Primeramente, vamos a modificar los ficheros de configuración de Kafka y ZooKeeper para que apunten a nuestro directorio "***tmp***". Para ello:

Para ZooKeeper modificamos:

**./etc/kafka/zookeeper.properties**

```
dataDir=/kafka/tmp/zookeeper
```

Para Apache Kafka modificamos:

**./etc/kafka/server.properties**

```
log.dirs=/kafka/tmp/kafka-logs
```

Una vez modificados pasamos a hacer unas copias de los ficheros de configuración de Kafka de la siguiente forma:

```
kafkauser@SERVER1:~/confluent/etc/kafka$ cp server.properties server1.properties
kafkauser@SERVER1:~/confluent/etc/kafka$ cp server.properties server2.properties
kafkauser@SERVER1:~/confluent/etc/kafka$ cp server.properties server3.properties
```

A continuación, vamos a editar los ficheros creados para cambiar las siguientes variables:

**server1.properties**

broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/kafka/tmp/kafka-logs1

**server2.properties**

broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/kafka/tmp/kafka-logs2

**server3.properties**

broker.id=3
listeners=PLAINTEXT://:9095
log.dirs=/kafka/tmp/kafka-logs5

La propiedad broker.id es el nombre único y permanente de cada uno de los nodos del clúster. En este caso tenemos que modificarlos porque nos encontramos en el mismo nodo. Si estuvieran en diferentes servidores nos haría falta cambiarlo.

A continuación, pasamos a levantar todos los componentes para ver cómo funcionan. Como en el LAB anterior, levantaremos en primer lugar ZooKeeper mediante:

kafkauser@SERVER1:~/confluent$ ./bin/zookeeper-server-start -daemon ./etc/kafka/zookeeper.properties

Comprobamos que está funcionando correctamente:

kafkauser@SERVER1:~/confluent$ echo "ruok"|nc localhost 2181
imok

Con ZooKeeper funcionando, procedemos a levantar los diferentes Brokers:

kafkauser@SERVER1:~/confluent$ ./bin/kafka-server-start -daemon etc/kafka/server1.properties
kafkauser@SERVER1:~/confluent$ ./bin/kafka-server-start -daemon etc/kafka/server2.properties
kafkauser@SERVER1:~/confluent$ ./bin/kafka-server-start -daemon etc/kafka/server3.properties

Comprobamos que los procesos están levantados:

```
kafkauser@SERVER1:~/confluent/etc$ netstat -putan|grep ::909
(No todos los procesos pueden ser identificados, no hay información de propiedad del proceso
 no se mostrarán, necesita ser superusuario para verlos todos.)
tcp6       0      0 :::9093                 :::*                    ESCUCHAR    18791/java
tcp6       0      0 :::9094                 :::*                    ESCUCHAR    18861/java
tcp6       0      0 :::9095                 :::*                    ESCUCHAR    18924/java
```

kafkauser@SERVER1:~/tmp$ ps -ef|grep server[123].proper|wc -l
3

Aquí también podemos utilizar zktop para ver las conexiones que se han abierto de los Kafka a ZooKeeper.

kafkauser@SERVER1:~/zktop$ ./zktop.py

```
Ensemble -- nodecount:28 zxid:0x2d sessions:4

ID SERVER          PORT M    OUTST    RECVD     SENT CONNS MINLAT AVGLAT MAXLAT
0  localhost       2181 S        0     2112     2116     4      0      0     18

CLIENT           PORT S I   QUEUED    RECVD     SENT
127.0.0.1       55420 0 0        0        1        0
0:0:0:0:0:0:0:1 33600 0 1        0      657      657
127.0.0.1       55338 0 1        0      752      757
127.0.0.1       55342 0 1        0      659      659
```

*Ilustración 2 zktop de los tres Brokers conectados a ZooKeeper.*

Parece que todo está levantado correctamente. Podemos ver como se han creado los espacios temporales de almacenamiento en el directorio ***/tmp***.

```
kafkauser@SERVER1:~/tmp$ tree
.
├── kafka-logs1
│   ├── cleaner-offset-checkpoint
│   ├── meta.properties
│   ├── recovery-point-offset-checkpoint
│   └── replication-offset-checkpoint
├── kafka-logs2
│   ├── cleaner-offset-checkpoint
│   ├── meta.properties
│   ├── recovery-point-offset-checkpoint
│   └── replication-offset-checkpoint
├── kafka-logs3
│   ├── cleaner-offset-checkpoint
│   ├── meta.properties
│   ├── recovery-point-offset-checkpoint
│   └── replication-offset-checkpoint
└── zookeeper
    └── version-2
        └── log.1
```

*Ilustración 3 Espacio temporal después de levantar los tres Brokers y ZooKeeper.*

Ahora que tenemos todo el entorno funcionando, vamos a crear una nueva cola con factor de replicación dos. Voy a poner la salida completa ya que es bastante interesante e instructivo ver cómo está funcionando.

kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --create --zookeeper localhost:2181 --replication-factor 2 --partitions 1 --topic TestLAB2

Recordamos que el ***número de particiones*** determinará el paralelismo que se puede alcanzar en el lado del consumidor. Por esta razón, deberemos escoger el número de particiones en base a la forma en que nuestros datos serán consumidos. El ***factor de replicación*** determinará el número de réplicas de la Topic presentes en el clúster. Como máximo sólo puede existir una por Broker. En nuestro ejemplo podríamos tener un factor de replicación de tres.

Salida del log:

[2017-06-01 23:41:23,033] INFO Created log for partition [TestLAB2,0] in /kafka/tmp/kafka-logs2 with properties



```
{compression.type -> producer, message.format.version -> 0.10.2-IV0, file.delete.delay.ms -> 60000, max.message.bytes -> 1000012, min.compaction.lag.ms -> 0, message.timestamp.type -> CreateTime, min.insync.replicas -> 1, segment.jitter.ms -> 0, preallocate -> false, min.cleanable.dirty.ratio -> 0.5, index.interval.bytes -> 4096, unclean.leader.election.enable -> true, retention.bytes -> -1, delete.retention.ms -> 86400000, cleanup.policy -> [delete], flush.ms -> 9223372036854775807, segment.ms -> 604800000, segment.bytes -> 1073741824, retention.ms -> 604800000, message.timestamp.difference.max.ms -> 9223372036854775807, segment.index.bytes -> 10485760, flush.messages -> 9223372036854775807}. (kafka.log.LogManager)
[2017-06-01 23:41:23,038] INFO Created log for partition [TestLAB2,0] in /kafka/tmp/kafka-logs3 with properties {compression.type -> producer, message.format.version -> 0.10.2-IV0, file.delete.delay.ms -> 60000, max.message.bytes -> 1000012, min.compaction.lag.ms -> 0, message.timestamp.type -> CreateTime, min.insync.replicas -> 1, segment.jitter.ms -> 0, preallocate -> false, min.cleanable.dirty.ratio -> 0.5, index.interval.bytes -> 4096, unclean.leader.election.enable -> true, retention.bytes -> -1, delete.retention.ms -> 86400000, cleanup.policy -> [delete], flush.ms -> 9223372036854775807, segment.ms -> 604800000, segment.bytes -> 1073741824, retention.ms -> 604800000, message.timestamp.difference.max.ms -> 9223372036854775807, segment.index.bytes -> 10485760, flush.messages -> 9223372036854775807}. (kafka.log.LogManager)
[2017-06-01 23:41:23,040] INFO Partition [TestLAB2,0] on broker 3: No checkpointed highwatermark is found for partition TestLAB2-0 (kafka.cluster.Partition)
[2017-06-01 23:41:23,036] INFO Partition [TestLAB2,0] on broker 2: No checkpointed highwatermark is found for partition TestLAB2-0 (kafka.cluster.Partition)
[2017-06-01 23:41:23,120] INFO [ReplicaFetcherManager on broker 2] Removed fetcher for partitions TestLAB2-0 (kafka.server.ReplicaFetcherManager)
[2017-06-01 23:41:23,144] INFO Truncating log TestLAB2-0 to offset 0. (kafka.log.Log)
[2017-06-01 23:41:23,216] INFO [ReplicaFetcherThread-0-3], Starting  (kafka.server.ReplicaFetcherThread)
[2017-06-01 23:41:23,230] INFO [ReplicaFetcherManager on broker 2] Added fetcher for partitions List([TestLAB2-0, initOffset 0 to broker BrokerEndPoint(3,SERVER1,9095)] ) (kafka.server.ReplicaFetcherManager)
```

Ahora podemos comprobar cómo ha ido la creación de la nueva Topic:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 --topic TestLAB2
Topic:TestLAB2  PartitionCount:1  ReplicationFactor:2
        Configs:
        Topic: TestLAB2  Partition: 0  Leader: 3
        Replicas: 3,2  Isr: 3,2
```

Viendo estos datos podemos saber que el Leader es el Broker 3 por lo que es el responsable de todas las lecturas y escrituras para la partición dada.

El número de réplicas es dos que se encuentran en los Brokers 2 y 3. Y que las dos réplicas están vivas ya que el ISR así lo indica.

Vamos a ver qué pasa si matamos el nodo marcado como Leader.

```
kafkauser@SERVER1:~/confluent$ kill -9 $(ps -ef|grep server3| grep -v grep|awk ' { print $2 } ')
```

En cuestión de segundos veremos que se produce el takeover:

```
Topic:TestLAB2  PartitionCount:1  ReplicationFactor:2
        Configs:
        Topic: TestLAB2  Partition: 0  Leader: 2
        Replicas: 3,2  Isr: 2
```

Si levantamos de nuevo el Broker 3 veremos cómo las réplicas sincronizadas (ISR) se recuperan:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 --topic TestLAB2

Topic:TestLAB2  PartitionCount:1  ReplicationFactor:2
        Configs:

        Topic: TestLAB2  Partition: 0  Leader: 2
        Replicas: 3,2  Isr: 2,3
```

Y si le dejamos un tiempo prudencial, volverá a ser el líder de nuevo:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 --topic TestLAB2
Topic:TestLAB2  PartitionCount:1  ReplicationFactor:2
        Configs:
        Topic: TestLAB2  Partition: 0  Leader: 3
        Replicas: 3,2  Isr: 2,3
```

Si matamos los dos nodos que contienen la Topic nos aparecerá algo parecido a esto:

```
kafkauser@SERVER1:~/confluent$ kill -9 $(ps -ef|grep server2| grep -v grep|awk ' { print $2 } ')
kafkauser@SERVER1:~/confluent$ kill -9 $(ps -ef|grep server3| grep -v grep|awk ' { print $2 } ')

kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 --topic TestLAB2
Topic:TestLAB2  PartitionCount:1  ReplicationFactor:2
        Configs:
        Topic: TestLAB2  Partition: 0  Leader: -1
        Replicas: 3,2  Isr:
```

La creación de las particiones también la podemos ver a nivel del directorio ***tmp***:



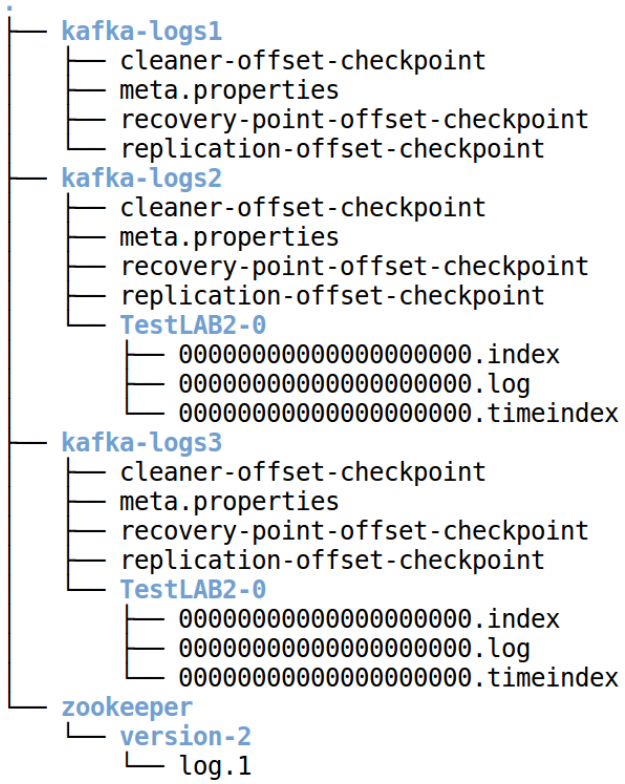


*Ilustración 4 Distribución de la Topic TestLAB2.*

Vemos que se corresponde con la salida que nos proporciona el comando.

## V. Pruebas con el consumidor y el productor.

Ahora que tenemos listo el entorno pasamos a crear algunos mensajes de prueba y consumirlos para ver como funciona.

Primeramente, levantaremos el producto y lanzaremos algunos mensajes mediante:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-console-producer --broker-list localhost:9093,localhost:9094,localhost:9095 --topic TestLAB2
```

Como curiosidad, de esta forma de invocar el consumidor, podemos decir que desde la versión 0.7 de apache kafka se ha eliminado la opción de invocar a Zookeeper para descubrir los Brokers. En las versiones actuales deberemos usar el parámetro ***"--broker-list"***.

Como podemos apreciar, ***kafka-console-producer***, es un cliente simple que nos permite probar de forma sencilla nuestro entorno. Debido a esta importancia vamos a enumerar algunos de los parámetros de entrada que soporta:

- ***--broker-list***. Especifica la lista de Brokers separa por comas.
- ***--topic***. Nombre de la cola Topic.
- ***--sync***. Especifica que el mensaje debe ser mandado de forma síncrona.
- ***--compression-code***. Especifica el códec usado para la compresión del mensaje. Puede ser, none, gzip, snappy ó lz4. Si no se especifica se enviará por defecto con gzip.
- ***--message-send-max-retries***. Esta propiedad determinará el número de reintentos máximo que podemos hacer para mandar un mensaje.

Ahora vamos a lanzar el consumidor mediante el siguiente comando:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-console-consumer --bootstrap-server localhost:9093,localhost:9094,localhost:9095 --from-beginning --topic TestLAB2
```

En este caso podríamos haber invocado a ZooKeeper pero nos dará una advertencia como la siguiente:

*"Using the ConsoleConsumer with old consumer is deprecated and will be removed in a future major release. Consider using the new consumer by passing [bootstrap-server] instead of [zookeeper]."*

Parece ser que la idea de kafka es ir quitando poco a poco las dependencias que tiene de ZooKeeper. Por lo que nos invita a comenzar a utilizar el parámetro ***"--bootstrap-server".***

Como en el laboratorio anterior hemos procedido a lanzar el consumidor con la opción ***--from-beginning*** por lo que nos mostrará todos los mensajes que contenga la Topic indicada, en ese momento. Si quitamos ese parámetro sólo recibiremos los mensajes desde el offset indicado del grupo de consumidores al que pertenezcamos.

El consumidor que nos proporciona Confluent para hacer pruebas también puede recibir parámetros que nos pueden ayudar a realizar pruebas interesantes:

- ***--fetch-size***. Este parámetro especifica el tamaño máximo de datos que nos podemos traer en una petición simple. El parámetro se especifica en bytes.
- ***--autocommit.interval***. Especifica el intervalo en el que el offset es guardado. El tiempo se especifica en ms.
- ***--max-messages***. Especifica el número máximo de mensajes que el consumidor consumirá antes de salir. Si no se especifica el valor por defecto será unlimited.
- ***--skip-message-on-error***. Si ocurre un error durante el procesamiento de mensajes, el sistema no debe parar. Simplemente debe desechar el mensaje.



Bueno ahora tenemos todo levantado así que debemos comprobar si funciona correctamente:

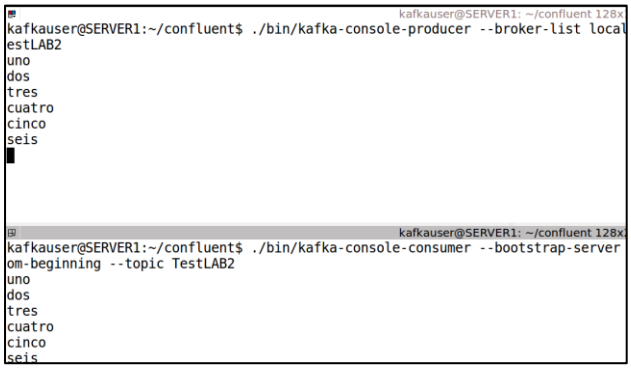


*Ilustración 5 Productor/consumidor funcionando.*

Ahora vamos a ver qué pasa con los consumidores/productores cuando algunos de los nodos caen. Primeramente, vamos a recordar cómo se ha creado nuestra Topic TestLAB2.

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 -topic TestLAB2
Topic:TestLAB2 PartitionCount:1 ReplicationFactor:2
        Configs:
        Topic: TestLAB2 Partition: 0 Leader: 3
        Replicas: 3,2 Isr: 2,3
```

Comprobamos que el factor de replicación que pusimos era dos y que el líder es el nodo 3, así que vamos a parar el nodo tres para ver qué sucede.

```
kafkauser@SERVER1:~/confluent$ kill -9 $(ps -ef|grep server3|grep -v grep|awk '{ print $2 }')
```

Una vez lo matamos vemos que el productor ni se ha enterado y que el consumidor empieza a dar algunos warnings de que no puede realizar el auto-commit del offsets:

```
[2017-06-07 11:59:27,743] WARN Auto-commit of offsets {TestLAB2-0=OffsetAndMetadata{offset=10, metadata=''}} failed for group console-consumer-32667: Offset commit failed with a retriable exception. You should retry committing offsets. (org.apache.kafka.clients.consumer.internals.ConsumerCoordinator)
```

Mientras nosotros seguimos produciendo mensajes y vemos como el consumidor sigue recibiéndolos sin problema:

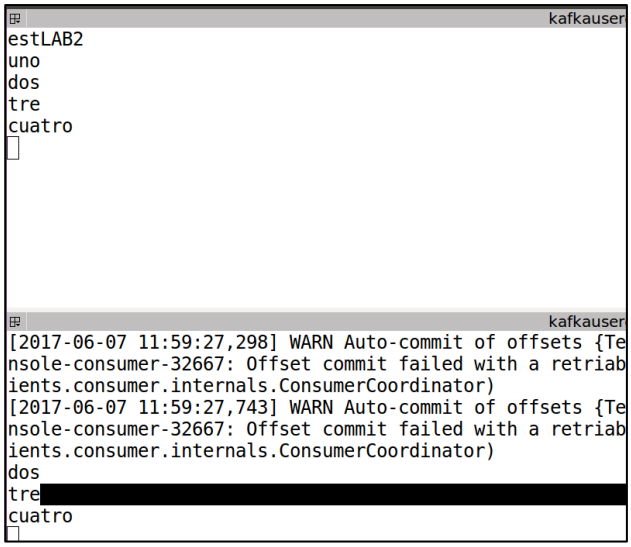


*Ilustración 6 Consumidor/productor después de matar el proceso lider.*

Si comprobamos como ha quedado la Topic configurada vemos lo siguiente:

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 -topic TestLAB2
Topic:TestLAB2 PartitionCount:1 ReplicationFactor:2
        Configs:
        Topic: TestLAB2 Partition: 0 Leader: 2
        Replicas: 3,2 Isr: 2
```

Ha tomado el control el Broker2 pasando a ser líder. Si matásemos el proceso dos los productores/consumidores fallarían ya que establecimos la replicación de la Topic TesLAB2 en dos.

Si levantamos de nuevo el Broker que hemos matado vemos como el entorno se recupera.

```
kafkauser@SERVER1:~/confluent$ ./bin/kafka-server-start -daemon ./etc/kafka/server3.properties

kafkauser@SERVER1:~/confluent$ ./bin/kafka-topics --describe --zookeeper localhost:2181 -topic TestLAB2
Topic:TestLAB2 PartitionCount:1 ReplicationFactor:2
        Configs:
        Topic: TestLAB2 Partition: 0 Leader: 3
        Replicas: 3,2 Isr: 2,3
```