
La captura de datos en tiempo real con Kafka y Flume

PID_00276696

Guillermo Argüello González

Tiempo mínimo de dedicación recomendado: 3 horas



Guillermo Argüello González

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé Ribalta

Primera edición: marzo 2021
© de esta edición, Fundació Universitat Oberta de Catalunya (FUOC)
Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Guillermo Argüello González
Producción: FUOC

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea este eléctrico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita del titular de los derechos.

Índice

Introducción.....	5
1. Flume.....	7
1.1. ¿Qué es Flume?	7
1.2. La arquitectura de Flume	8
1.3. La definición de los agentes de Flume	9
1.3.1. La definición del flujo	9
1.3.2. La configuración individual de los componentes	10
1.3.3. La configuración de los múltiples flujos en un agente	12
1.4. Flume <i>sources</i>	12
1.4.1. Exec <i>source</i>	13
1.4.2. Kafka <i>source</i>	13
1.4.3. Twitter 1 % <i>firehose source</i>	14
1.5. Flume <i>channels</i>	15
1.5.1. <i>Memory channel</i>	15
1.5.2. <i>File channel</i>	16
1.6. Flume <i>sinks</i>	17
1.6.1. HDFS <i>sink</i>	17
1.6.2. Hive <i>sink</i>	18
1.6.3. HBase <i>sink</i>	19
1.6.4. Elasticserach <i>sink</i>	19
1.6.5. Ejemplo práctico	20
2. Kafka.....	22
2.1. ¿Qué es Kafka?	23
2.2. La arquitectura y los componentes de Kafka	23
2.3. La instalación básica	24
2.4. Los componentes	24
2.4.1. Bróker	24
2.4.2. Zookeeper	25
2.4.3. <i>Producers</i>	25
2.4.4. <i>Topics</i>	26
2.4.5. <i>Consumers</i>	29
2.5. La tolerancia a los fallos	32
2.5.1. La tolerancia a los fallos al producir	32
2.5.2. La tolerancia a los fallos al consumir	33
2.6. La configuración	34
2.6.1. La profundidad histórica	34
2.6.2. La compactación	34
2.6.3. Las cuotas	35

2.6.4. La seguridad	35
2.7. Ejemplo completo	35
2.8. Los casos de uso	37
Bibliografía	39

Introducción

En este módulo vamos a ver dos de las herramientas más importantes y utilizadas para la captura y preprocesamiento de datos en tiempo real: **Flume** y **Kafka**. Como veréis, las dos tecnologías son radicalmente diferentes, con sus ventajas e inconvenientes. Contextualizado con el módulo «Patrones de capturas de datos dinámicos», las dos herramientas se podrían clasificar como patrones de publicación/suscripción. Es decir, para leer los datos capturados por Flume y Kafka, se debe establecer un canal de comunicación estable con los citados sistemas. Aunque en general parecen muy similares, veremos que en realidad son muy distintos y están pensados con objetivos diferentes:

- Kafka es un sistema de mensajería productor/suscriptor de propósito general. No está específicamente diseñado para entornos *big data*, aunque es ampliamente utilizado en ellos debido a sus características y a su robustez. Sin embargo, Flume es parte del ecosistema Hadoop y se usa para recopilar, agregar y mover eficientemente grandes cantidades de datos a un almacenamiento de datos como HDFS, S3 o Hbase.
- Ambos están pensados para un distinto número de accesos concurrentes. Aumentar el número de consumidores en Kafka es muy sencillo y no afecta al rendimiento. Sin embargo, aumentar el número de consumidores en Flume implica cambiar su topología y también genera indisponibilidad en el sistema.
- En cuanto a la replicación y la tolerancia a los fallos, ambos aseguran la tolerancia a los fallos, pero Kafka replica los eventos en su clúster, mientras que Flume no lo hace. En este caso, Kafka es mucho más robusto a las caídas parciales del sistema.

Ambas herramientas son muy potentes y no tenemos que considerar que una es mejor que la otra, sino que simplemente tienen objetivos distintos. Elegiremos Kafka o Flume dependiendo de nuestro caso de uso, del resto de herramientas usadas y del ecosistema en el que vayamos a trabajar. En general, utilizaremos Flume cuando trabajemos con datos no relacionales y los queramos ingresar en un ecosistema Hadoop. Kafka tiene un propósito general y es que puede ser utilizado o no en entornos *big data* y es especialmente útil cuando queremos conectar múltiples sistemas.

El uso más habitual de Flume es leer **en tiempo real** información de diferentes fuentes, como las redes sociales (Twitter, Facebook...), los *logs* de páginas web, de procesos y servidores, así como todo tipo de ficheros y, después, llevar esta información a Hadoop (HDFS, Hive, Hbase...) para el posterior procesamiento y análisis de esos datos.

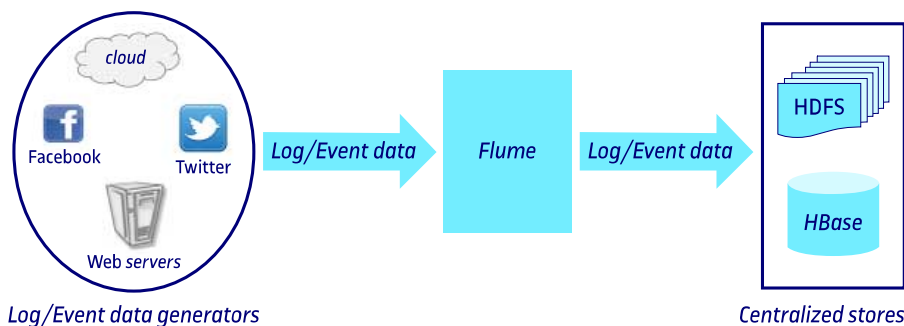
Sin embargo, Kafka tiene un uso más abierto, ya que puede usarse como cola de mensajería entre los servicios de una arquitectura de microservicios. Otro uso clásico es el de herramienta de monitorización con al que se recogen *logs* de varios sistemas dentro de una aplicación, agregándolos y centralizándolos para un procesamiento posterior. Kafka, como veremos con la arquitectura Kappa, incluso puede usarse como mecanismo de *storage* ya que es capaz de almacenar gran cantidad de datos.

1. Flume

En este apartado vamos a presentar la primera de las herramientas de captura de datos en tiempo real que estudiaremos: Apache Flume.

1.1. ¿Qué es Flume?

Apache Flume comenzó a desarrollarse en 2010 como aplicación para la recogida de *logs*. Surge para cubrir una necesidad parecida a la que propició el nacimiento de Kafka. Aun y sus inicios, como sus fuentes de datos de entrada son configurables, también se ha propiciado su utilización para recoger y transportar datos de las redes sociales, las webs o los correos electrónicos, entre otros.



Fuente: https://www.tutorialspoint.com/apache_flume/apache_flume_introduction.htm

Un posible caso de uso de Flume es leer Twitter en tiempo real filtrando los tuits que tengan unas determinadas palabras clave. Por ejemplo, cuando una empresa lanza una nueva campaña publicitaria, se recogen todos los tuits que contengan el nombre de la empresa antes, durante y después del lanzamiento. Estos tuits se podrían almacenar en HDFS o en un lago de datos. Posteriormente, con estos datos se podría hacer un análisis de sentimientos para ver si el recibimiento de la campaña en redes ha sido positivo o negativo y qué impacto ha tenido.

Las principales características y ventajas de Flume son las siguientes:

- La capacidad de ingestar datos prácticamente en tiempo real de diferentes fuentes en un sistema de almacenamiento distribuido como HDFS o Hbase, permitiendo así el procesamiento en un ecosistema Hadoop.
- Las fuentes de datos que puede leer son diversas: *logs*, redes sociales, webs...
- La capacidad de soportar el movimiento de grandes volúmenes de datos.
- Flume es escalable horizontalmente, es decir, podemos añadir nuevos nodos, igual que en un clúster Hadoop, hasta satisfacer los requerimientos de procesamiento definidos.

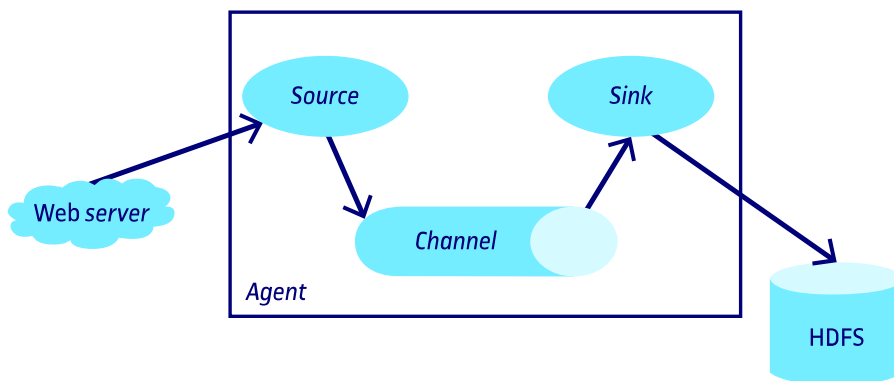
Los principales inconvenientes que presenta Flume son los siguientes:

- No garantiza al 100 % la entrega de mensajes únicos: pueden aparecer mensajes duplicados en cualquier momento durante el funcionamiento de un agente. Garantiza la semántica *at-least-once*.
- El rendimiento depende en gran medida del canal que estemos utilizando, por lo que la escalabilidad, aunque es posible, se ve afectada por este factor.

Con estas características en mente, a continuación vamos a estudiar su arquitectura y sus componentes principales. Primero, vamos a presentar los principales componentes de la arquitectura de Flume y en los siguientes apartados profundizaremos en ellos.

1.2. La arquitectura de Flume

En la siguiente imagen podemos ver un diagrama de la arquitectura básica de Flume. En ella, primeramente, tenemos un generador de datos (en este caso, un servidor web) cuyos datos generados pretenden ser recolectados por lo que conocemos como **agente** de Flume. Finalmente, vemos el tipo de unidad en la que se van a dejar los datos (en este caso, en un sistema de ficheros HDFS). En el proceso de ingesta, procesado y *storage* de datos llamaremos **evento** a la unidad básica que transportará Flume.



Fuente: <https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#>

Como vemos en el diagrama, el agente de Flume está compuesto por tres piezas básicas: **sources** o **fuentes**, **channels** o **canales** y **sinks** o **sumideros**.

La **fente** es el primer componente de un agente Flume. Este recibe los datos de las fuentes y los envía a uno o más canales dentro de Flume. Soporta muchos tipos de fuentes distintas, cada una de ellas diseñada para recibir los datos de un generador de datos diferente. Por ejemplo, existen fuentes Twitter, Facebook, ficheros Avro...

El **canal** es un almacenamiento temporal de datos, recibe los datos y los retiene hasta que son consumidos por los **sinks**. Los tipos de canales más habituales son **en memoria** y **en fichero**.

Por último, el **sumidero** es el encargado de almacenar la información en un repositorio de datos centralizado como puede ser HDFS o Hbase. Este consume los eventos de los canales y los deja en el destino definido. Como ya hemos dicho varias veces, lo más habitual es que el *sink* sea HDFS.

Un agente de Flume no tiene por qué tener solo una fuente, un canal y un sumidero, como el del ejemplo, sino que puede tener múltiples componentes de cada tipo que en conjunto forman un único agente. Por ejemplo, podemos tener un agente diseñado para leer simultáneamente información relativa a una empresa en diversas redes sociales. Este agente podría estar compuesto por dos fuentes: una que reciba datos de Twitter y otra de Facebook; dos canales diferentes, uno en memoria y otro en fichero y, para acabar, con un único *sink* que escribe todos los datos a HDFS para su posterior análisis en un lago de datos.

A continuación, vamos a ver en detalle y con ejemplos concretos, cada una de estas tres piezas que acabamos de presentar y, por último, veremos algunos detalles sobre la configuración de Flume.

1.3. La definición de los agentes de Flume

Para definir un flujo de datos y un agente de Flume necesitaremos generar un fichero de configuración que, en general, es bastante sencillo e intuitivo. A través de este fichero vamos a tener acceso a la mayor parte de las funcionalidades implementadas que veremos en los próximos apartados.

1.3.1. La definición del flujo

Para definir el flujo de un agente, primeramente, se definen las fuentes, los sumideros y los canales. A continuación, se unen las fuentes y los sumideros a través de un canal. Una fuente puede estar conectada a múltiples canales, mientras que un sumidero solo puede estar conectado a un canal. El formato del fichero para definir el flujo es el siguiente:

```
# list the sources, sinks and channels for the agent
<Agent>.sources = <Source>
<Agent>.sinks = <Sink>
<Agent>.channels = <Channel1> <Channel2>

# set channel for source
<Agent>.sources.<Source>.channels = <Channel1> <Channel2> ...

# set channel for sink
<Agent>.sinks.<Sink>.channel = <Channel1>
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

Por ejemplo, si tenemos un agente que se llama `agent_foo`, que lee información de una fuente externa tipo avro y envía la información a HDFS utilizando un canal en memoria, la configuración quedaría de la siguiente forma:

```
# list the sources, sinks and channels for the agent
agent_foo.sources = avro-appserver-src-1
agent_foo.sinks = hdfs-sink-1
agent_foo.channels = mem-channel-1

# set channel for source
agent_foo.sources.avro-appserver-src-1.channels = mem-channel-1

# set channel for sink
agent_foo.sinks.hdfs-sink-1.channel = mem-channel-1
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

En las tres primeras líneas definimos los nombres del agente y los de sus componentes. La sintaxis es simple. En primer lugar, tenemos `agent_foo` que es el nombre de nuestro agente y, seguido de un punto, el componente del agente que estamos definiendo. En este caso una *source*, un canal y un *sink*. Después del igual, tenemos el nombre que le vamos a dar a cada uno de ellos. Este nombre va a ser simplemente un identificador textual (lo define el programador) del componente que estamos definiendo.

A continuación, asignamos el canal que va a utilizar la *source* y el *sink*. Para definirlo utilizamos el nombre del agente, indicamos si es la *source* o el *sink*, el nombre de la *source* o del *sink* que hemos definido anteriormente y, por último, la palabra reservada *channel* que nos sirve para indicar que estamos definiendo el canal para este componente. Después del igual indicamos el nombre del canal que hemos definido anteriormente. A partir de aquí, ya tenemos la *source* y el *sink* conectados a través de un canal. Notad el plural y el singular en las *sources* (que pueden ser múltiples) y en los *sinks* (que son únicos por canal). Aun así, con esto solo hemos definido el flujo, pero tenemos que configurar de qué tipo será cada componente y sus propiedades. Esto es lo que vamos a ver a continuación.

1.3.2. La configuración individual de los componentes

Una vez definido el flujo, debemos configurar de cada uno de los componentes del agente Flume. Estas configuraciones se realizan en el mismo fichero y a continuación de las definiciones anteriores. Lo haremos con la siguiente sintaxis:

```
# properties for sources
<Agent>.sources.<Source>.<someProperty> = <someValue>

# properties for channels
<Agent>.channel.<Channel>.<someProperty> = <someValue>

# properties for sinks
<Agent>.sources.<Sink>.<someProperty> = <someValue>
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

Las *sources*, los *channels* y los *sinks* tienen sus propios valores que configurar y necesitamos definirlos individualmente. En el subapartado anterior vimos cómo definir un flujo de la *source* `avro-AppSrv-source` al `hdfs-Cluster1-sink` a través del canal de memoria `mem-channel-1`. Ahora vamos a ver un ejemplo de la configuración de cada uno de esos componentes.

```
agent_foo.sources = avro-AppSrv-source
agent_foo.sinks = hdfs-Cluster1-sink
agent_foo.channels = mem-channel-1

# set channel for sources, sinks

# properties of avro-AppSrv-source
agent_foo.sources.avro-AppSrv-source.type = avro
agent_foo.sources.avro-AppSrv-source.bind = localhost
agent_foo.sources.avro-AppSrv-source.port = 10000

# properties of mem-channel-1
agent_foo.channels.mem-channel-1.type = memory
agent_foo.channels.mem-channel-1.capacity = 1000
agent_foo.channels.mem-channel-1.transactionCapacity = 100

# properties of hdfs-Cluster1-sink
agent_foo.sinks.hdfs-Cluster1-sink.type = hdfs
agent_foo.sinks.hdfs-Cluster1-sink.hdfs.path = hdfs://namenode/flume/webdata

#...
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

Como se puede ver en la anterior imagen, después de la definición del agente, tenemos las propiedades de cada componente, donde indicamos qué tipo de fuente es (en este caso, `avro`) y en qué máquina se encuentra. Aunque en este ejemplo la IP corresponde a la misma máquina, en general en el *bind* pondremos la IP del sistema que emite los datos. Indicamos que el canal está en memoria y la capacidad de este (estos puntos son importantes y los vamos a ver en detalle a continuación). Por último, se define el *sink* (en este caso es HDFS), lo que quiere decir que todos los eventos que capture nuestro agente de Flume se van a almacenar en la carpeta *webdata* de nuestro HDFS. Más adelante entraremos en el detalle sobre algunas de estas propiedades.

1.3.3. La configuración de los múltiples flujos en un agente

Como se ha comentado, un único agente puede tener varios flujos. Por ejemplo, dos fuentes y canales diferentes y un único *sink*. Definir más de un componente es tan sencillo como definirlos separados por espacios:

```
# list the sources, sinks and channels for the agent
<Agent>.sources = <Source1> <Source2>
<Agent>.sinks = <Sink1> <Sink2>
<Agent>.channels = <Channel1> <Channel2>
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

De esta forma podemos definir distintos flujos dentro de nuestro agente:

```
# list the sources, sinks and channels in the agent
agent_foo.sources = avro-AppSrv-source1 exec-tail-source2
agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2
agent_foo.channels = mem-channel-1 file-channel-2

# flow #1 configuration
agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-1
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-1

# flow #2 configuration
agent_foo.sources.exec-tail-source2.channels = file-channel-2
agent_foo.sinks.avro-forward-sink2.channel = file-channel-2
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

En este ejemplo estamos definiendo dos fuentes, dos canales y dos *sinks* y, a continuación, se definen dos flujos independientes. El primer flujo va del `avro-AppSrv-source1` al `hdfs-Cluster1-sink1` pasando por el `mem-channel-1`. El segundo va del `exec-tail-source2` al `avro-forward-sink2` pasando por el `file-channel-2`.

1.4. Flume sources

En este apartado vamos a ver los tipos más frecuentes de fuentes de Flume. Las *sources* son las piezas del agente de Flume que reciben información de fuentes externas y las envían a uno o más canales. Como hemos visto anteriormente, las fuentes de las que se puede recibir información son variadas: webs, redes sociales, ficheros, etc. Este generador de información externo enviará la información a Flume en un formato que será reconocible y aceptado por el agente de Flume. Vamos a ver, a continuación, los tipos de fuentes más utilizados e importantes, aunque hay muchos más que se pueden consultar en la documentación oficial.

Enlace recomendado

Podéis acceder al siguiente enlace de la documentación oficial para consultar los tipos de fuentes: <https://bit.ly/3bze3r8>.

1.4.1. Exec source

Esta es una fuente muy útil para monitorizar procesos. Lo que hace es ejecutar un comando en una máquina Unix y recoger toda salida que se produce para que se pueda procesar en otro sitio. A continuación, vamos a ver las propiedades que podemos configurar en este tipo de fuente:

Nombre de la propiedad	Valor por defecto	Descripción
channels (obligatoria)	—	El nombre del canal en el que vamos a escribir los datos leídos.
type (obligatoria)	—	Es el tipo de componente y tiene que ser <code>exec</code> .
command (obligatoria)	—	El comando que queremos ejecutar.

En la siguiente imagen podemos ver un ejemplo de configuración de un canal de tipo ejecución:

```
a1.sources.tailsource-1.type = exec
a1.sources.tailsource-1.shell = /bin/bash -c
a1.sources.tailsource-1.command = for i in /path/*.txt; do cat $i; done
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

En este caso tenemos un agente que se llama `a1` para el que definimos una fuente que se llama `tailsource-1`. Le indicamos el tipo de fuente (`exec`) y el comando que queremos ejecutar.

1.4.2. Kafka source

Esta fuente lee datos de un *topic* Kafka (en el próximo apartado se profundizará en Kafka). A continuación, vamos a ver las propiedades que podemos configurar en este tipo de fuente:

Nombre de la propiedad	Valor por defecto	Descripción
channels (obligatoria)	—	El nombre del canal en el que vamos a escribir los datos leídos.
type (obligatoria)	—	Es el tipo de componente y tiene que ser <code>org.apache.flume.source.kafka.KafkaSource</code> .
kafka.bootstrap.servers (obligatoria)	—	Lista de brókeres en los que está corriendo el Kafka del que queremos leer.
kafka.topics (obligatoria)	—	Lista separada por comas de los <i>topics</i> de Kafka que va a leer el <i>source</i> .
kafka.topics.regex (obligatoria)	—	Regex para indicar el conjunto de <i>topics</i> que queremos leer o que usaremos. Esta propiedad o la anterior.

Nombre de la propiedad	Valor por defecto	Descripción
batchSize	1000	Número de mensajes máximo por <i>batch</i> .

En la siguiente imagen podemos ver un ejemplo de configuración de un canal de Kafka:

```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.batchSize = 5000
tier1.sources.source1.batchDurationMillis = 2000
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics = test1, test2
tier1.sources.source1.kafka.consumer.group.id = custom.g.id
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

En este caso tenemos un agente que se llama `tier1`, para el que definimos una fuente que se llama `source1` y un canal que se llama `channel1`. Le indicamos el tipo de fuente (Kafka), los servidores en los que se encuentra Kafka (en este caso, `localhost:9092`) y los *topics* (en este caso, `test1` y `test2`).

1.4.3. Twitter 1 % *firehose source*

Esta fuente de datos de Flume lee datos de Twitter con el filtro que le indiquemos, toma un 1 % de los mismos de forma aleatoria, los transforma a formato Avro y los envía al canal para su posterior procesamiento. Estas son las propiedades que podemos configurar en este tipo de fuente:

Nombre de la propiedad	Valor por defecto	Descripción
channels (obligatoria)	—	El nombre del canal en el que vamos a escribir los datos leídos.
type (obligatoria)	—	Es el tipo de componente y tiene que ser <code>org.apache.flume.source.twitter.TwitterSource</code> .
consumerKey (obligatoria)	—	Credenciales de la cuenta de Twitter.
consumerSecret (obligatoria)	—	Credenciales de la cuenta de Twitter.
accessToken (obligatoria)	—	Credenciales de la cuenta de Twitter.
accessTokenSecret (obligatoria)	—	Credenciales de la cuenta de Twitter.
maxBatchSize	1000	Número de mensajes máximo por <i>batch</i> .

En la siguiente imagen podemos ver un ejemplo de configuración de un canal de Twitter:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.twitter.TwitterSource
a1.sources.r1.channels = c1
a1.sources.r1.consumerKey = YOUR_TWITTER_CONSUMER_KEY
a1.sources.r1.consumerSecret = YOUR_TWITTER_CONSUMER_SECRET
a1.sources.r1.accessToken = YOUR_TWITTER_ACCESS_TOKEN
a1.sources.r1.accessTokenSecret = YOUR_TWITTER_ACCESS_TOKEN_SECRET
a1.sources.r1.maxBatchSize = 10
a1.sources.r1.maxBatchDurationMillis = 200
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#configuration>

En este caso, tenemos un agente que se llama `a1` para el que definimos una fuente que se llama `r1` y un canal que se llama `c1`. A continuación, le indicamos el tipo de fuente (Twitter), el canal en el que tiene que escribir, que es `c1` y, por último, configuraríamos nuestras claves de acceso a Twitter y el tamaño máximo del *batch*.

1.5. Flume channels

Los canales son las piezas intermedias en los agentes de Flume que están entre las fuentes y los sumideros. Las fuentes escriben la información que reciben en los canales y esta información reside temporalmente en los canales hasta que la consumen los sumideros. La forma más intuitiva de entender los canales es como repositorio de almacenamiento temporal (*buffer*), hasta que la información es consumida. Los canales también son los encargados de asegurar que no hay pérdida de información en los agentes de Flume. La definición o configuración óptima de los canales es compleja y se tienen que considerar el tipo de información que va a transmitir el canal (y su tamaño), el flujo que va a tener que soportar y las características de los *sinks* y las *sources*. A continuación, vamos a ver los tipos de canales más utilizados en Flume, pero hay muchos más.

Enlace recomendado

Para consultar los tipos de canales y ver todas sus propiedades, se puede consultar la documentación oficial de Flume en el siguiente enlace: <https://bit.ly/3aTcKnJ>.

1.5.1. Memory channel

Cuando utilizamos este tipo de canal, los eventos se almacenan en una cola de tamaño configurable en la memoria principal (RAM). Utilizaremos este tipo de canal cuando nos interesa un mayor rendimiento y podemos asumir perder algunos eventos en caso de que el agente falle. Por ejemplo, si estamos leyendo datos de Twitter para hacer un estudio de mercado para nuestra empresa, quizá consideremos que no es completamente necesario tener el 100 % de los mensajes generados. Sin embargo, sí que queremos que la información nos llegue lo más rápido posible. Estas son las propiedades que podemos configurar en este tipo de canal.

Nombre de la propiedad	Valor por defecto	Descripción
<code>type</code> (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>memory</code> .
<code>transactionCapacity</code>	100	El número de eventos que el canal va a leer del <code>source</code> o va a enviar al <code>sink</code> por transacción.

Nombre de la propiedad	Valor por defecto	Descripción
capacity	100	Máximo número de eventos que almacenará el canal.

En la siguiente imagen podemos ver un ejemplo de configuración de un canal de Flume en memoria:

```
a1.channels = c1
a1.channels.c1.type = memory
a1.channels.c1.capacity = 10000
a1.channels.c1.transactionCapacity = 10000
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-channels>

En este caso tenemos un agente que se llama `a1` para el que definimos un canal que se llama `c1`; en la segunda línea definimos que el tipo de canal es en memoria y en las dos últimas definimos el número máximo de eventos que va a tener el canal y el número máximo por transacción. Si el canal está al máximo de capacidad, cualquier intento de insertar un nuevo mensaje por parte de una *source* en el canal será fallido con un mensaje `ChannelException` al menos hasta que un *sink* lea tantos eventos como los que estamos intentando insertar. Antes de fallar, el canal espera un máximo de `keep-alive` segundos, durante los cuales reintentará escribirlos. Si no lo consigue, fallará definitivamente y todos los eventos que se generen hasta que volvamos a levantar el agente se perderán.

1.5.2. File channel

Cuando utilizamos este tipo de canal, los eventos se guardan en algún sistema de ficheros intermedio, desde donde, posteriormente, los consume el *sink*. Utilizamos este tipo de canal cuando queremos asegurarnos de que ningún evento se pierde, aunque afectemos al rendimiento del agente. Suponemos que estamos utilizando Flume para ingestar en HDFS todas las operaciones bancarias que hacen nuestros clientes. En este caso, nos tenemos que asegurar de que no se pierde ningún evento y es aconsejable utilizar un canal en fichero. Los eventos no serán borrados del disco hasta que el sumidero no los lea correctamente. Estas son las propiedades más importantes que podemos configurar en un *file channel*.

Nombre de la propiedad	Valor por defecto	Descripción
type (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>file</code> .
maxFileSize	2146435071	Máximo tamaño en <i>bytes</i> que puede tener un único evento.
dataDirs	<code>~/flume/file-channel/data</code>	Lista de directorios separados por comas en donde se almacenarán los datos de los eventos.

Nombre de la propiedad	Valor por defecto	Descripción
capacity	1000000	Máximo número de eventos que almacenará el canal.

En la siguiente imagen podemos ver un ejemplo de configuración de un canal de Flume en fichero:

```
a1.channels = c1
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
a1.channels.c1.dataDirs = /mnt/flume/data
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-channels>

En este caso tenemos un agente que se llama `a1`, para el que definimos un canal que se llama `c1`. En la segunda línea definimos que el tipo de canal es el fichero y, en las dos últimas, el directorio de los `checkpoints` y de los datos.

1.6. Flume sinks

Los *sinks* o sumideros de Flume son los componentes de los agentes de Flume encargados de enviar toda la información recogida a un sistema de almacenamiento centralizado. Consume eventos de los canales y empuja (*hace push*) estos eventos al lugar de almacenamiento (*sink*). Algunos *sinks*, como el específico de Spark, permiten elegir cómo consumir la información: modo *push* o *pull*. Un *sink* solo puede leer de un único canal, pero varios *sinks* pueden leer de un mismo canal. Vamos a ver los tipos de sinks más utilizados en detalle.

1.6.1. HDFS sink

Este componente escribe los eventos en HDFS. Los formatos de ficheros que admite este componente son de tipo texto y de tipo secuencia y ambos pueden ser comprimidos. Según el número de eventos, el tamaño de los eventos o el tiempo, podemos configurar que el componente dé por terminada la escritura de un fichero y empiece con el siguiente. Para configurarlo utilizaremos las siguientes propiedades:

Nombre de la propiedad	Valor por defecto	Descripción
type (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>hdfs</code> .
channel (obligatorio)	—	El nombre del canal del que vamos a leer.
hdfs.path (obligatorio)	—	El directorio de HDFS en donde se van a escribir los datos.

Vamos a ver un ejemplo de fichero de configuración:

Ved también

Para más información sobre cómo consumir la información, revisad los módulos «Patrones de capturas de datos dinámicos» y «Arquitecturas *big data* para el procesado de datos en flujo».

Enlace recomendado

Para ver todos los detalles de los *sinks*, se puede consultar la documentación oficial en el siguiente enlace: <http://flume.apache.org/FlumeUserGuide.html#flume-sinks>.

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-channels>

En este ejemplo tenemos un agente que se llama `a1`, un *sink* que se llama `k1` y un canal que se llama `c1`. Posteriormente, definimos el *sink* como HDFS, le decimos que lea del canal `c1` y, por último, la ruta en la que se van a almacenar los eventos.

1.6.2. Hive *sink*

El *sink* de Hive recoge los eventos de un canal y los deja directamente en una tabla Hive. Los eventos tienen que estar en ficheros de texto delimitados o en formato JSON para poder utilizar este componente. También permite utilizar las funcionalidades de Hive como particiones. Algunas de las propiedades de configuración son las siguientes:

Nombre de la propiedad	Valor por defecto	Descripción
<code>type</code> (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>hive</code> .
<code>channel</code> (obligatorio)	—	El nombre del canal del que vamos a leer.
<code>hive.metastore</code> (obligatorio)	—	La URI del <i>metastore</i> de Hive (por ejemplo, <code>thrift://a.b.com:9083</code>).
<code>hive.database</code> (obligatorio)	—	La base de datos de Hive donde se encuentra la tabla que queremos escribir.
<code>hive.table</code> (obligatorio)	—	La tabla de Hive que queremos escribir.

Vamos a ver un ejemplo de fichero de configuración:

```
a1.channels = c1
a1.channels.c1.type = memory
a1.sinks = k1
a1.sinks.k1.type = hive
a1.sinks.k1.channel = c1
a1.sinks.k1.hive.metastore = thrift://127.0.0.1:9083
a1.sinks.k1.hive.database = logsdb
a1.sinks.k1.hive.table = weblogs
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-sinks>

En este ejemplo tenemos un agente que se llama `a1`, un *sink* que se llama `k1` y un canal en memoria que se llama `c1`. Posteriormente, definimos el *sink* como Hive, le decimos que lea del canal `c1` y, por último, el *metastore* de Hive, el nombre de la base de datos y el nombre de la tabla.

1.6.3. HBase sink

Este sumidero escribe la información en HBase. Utiliza todas las funcionalidades de seguridad y de consistencia de la base de datos. A continuación, veremos los parámetros necesarios para su configuración:

Nombre de la propiedad	Valor por defecto	Descripción
type (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>hbase</code> .
channel (obligatorio)	—	El nombre del canal del que vamos a leer.
table (obligatorio)	—	El nombre de la tabla de Hbase en la que vamos a escribir.
columnFamily (obligatorio)	—	La familia de columnas en Hbase en la que vamos a escribir.

Vamos a ver un ejemplo de fichero de configuración:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hbase
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase.RegexHbaseEventSerializer
a1.sinks.k1.channel = c1
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-sinks>

En este ejemplo tenemos un agente que se llama `a1`, un *sink* que se llama `k1` y un canal que se llama `c1`. Posteriormente, definimos el *sink* como `hbase`, le decimos que lea del canal `c1` y, por último, la tabla, la familia de columnas y el serializador que utilizaremos para escribir.

1.6.4. Elasticsearch sink

Este es el último tipo de *sink* que vamos a presentar. Se encarga de escribir los datos en un clúster de Elasticsearch, que es una base de datos orientada a textos y deja la información lista para que se vea en Kibana, la herramienta de visualización de Elasticsearch. Veamos, a continuación, sus propiedades más importantes:

Nombre de la propiedad	Valor por defecto	Descripción
type (obligatoria)	—	Es el nombre del tipo de componente y tiene que ser <code>elasticsearch</code> .
channel (obligatorio)	—	El nombre del canal del que vamos a leer.
hostNames (obligatorio)	—	Lista separada por comas de servidores y puertos en los que se encuentra instalado Elasticsearch.
indexname	flume	El índice de <i>elastic</i> en el que queremos escribir.

Vamos a ver un ejemplo de fichero de configuración:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = elasticsearch
a1.sinks.k1.hostNames = 127.0.0.1:9200,127.0.0.2:9300
a1.sinks.k1.indexName = foo_index
```

Fuente: <http://flume.apache.org/FlumeUserGuide.html#flume-sinks>

En este ejemplo tenemos un canal que se llama `a1`, un *sink* que se llama `k1` y un canal que se llama `c1`. Posteriormente, definimos el *sink* como `elasticsearch`, listamos los servidores en los que está instalado Elasticsearch junto a sus puertos y le indicamos el índice en el que queremos escribir, que es `foo_index`.

Hemos visto solo cuatro ejemplos de sumideros que Flume tiene preconfigurados. Estos cuatro son los más utilizados e importantes porque corresponden a los cuatro sistemas de almacenamiento más extendidos en la actualidad. Sin embargo, existen muchos más *sinks* disponibles y muchas más propiedades con las que podemos hacer la ingesta de datos más flexible.

1.6.5. Ejemplo práctico

Para acabar con el apartado de Flume vamos a ver un ejemplo práctico sencillo de todo lo que hemos visto hasta ahora. Vamos a crear un agente de Flume que reciba todo lo que le pasemos por la consola y lo deje en ficheros en HDFS. Para ello utilizaremos el tipo de *source* `netcat` <https://www.tutorialspoint.com/apache_flume/apache_flume_netcat_source.htm>, canal en memoria y *sink* HDFS. Lo primero que tenemos que hacer es crear un fichero de configuración que contenga todo lo que queremos hacer con la sintaxis de Flume, como se puede ver a continuación:

```
flume.conf
1 ExampleAgent.sources = netcat1
2 ExampleAgent.channels = MemChannel
3 ExampleAgent.sinks = HDFSSink
4
5 # SOURCE CONFIGURATION
6 # -----
7 ExampleAgent.sources.netcat1.type = netcat
8 ExampleAgent.sources.netcat1.channels = MemChannel
9 ExampleAgent.sources.netcat1.bind = localhost
10 ExampleAgent.sources.netcat1.port = 44444
11
12 # CHANNEL CONFIGURATION
13 # -----
14 ExampleAgent.channels.MemChannel.type = memory
15 ExampleAgent.channels.MemChannel.byteCapacity = 6912212
16 ExampleAgent.channels.MemChannel.capacity = 100
17 ExampleAgent.channels.MemChannel.transactionCapacity = 100
18
19 # SINK CONFIGURATION
20 # -----
21 # SINK hdfs
22 ExampleAgent.sinks.HDFSSink.channel = MemChannel
23 ExampleAgent.sinks.HDFSSink.type = hdfs
24 ExampleAgent.sinks.HDFSSink.hdfs.path = hdfs://Cloudera01:8020/user/garguello/flume_example
25 ExampleAgent.sinks.HDFSSink.hdfs.fileType = DataStream
26 ExampleAgent.sinks.HDFSSink.hdfs.writeFormat = Text
27 ExampleAgent.sinks.HDFSSink.hdfs.batchSize = 100
28 ExampleAgent.sinks.HDFSSink.hdfs.rollSize = 0
29 ExampleAgent.sinks.HDFSSink.hdfs.rollCount = 10000
30
```

En este ejemplo tenemos un agente que se llama `ExampleAgent`, un *sink* que se llama `HDFS Sink`, una *source* que se llama `netcat1` y un canal que se llama `MemChannel`; esto es lo que definimos en las tres primeras líneas. A continuación, definiremos cómo queremos que sea cada uno de estos componentes:

- *Source*: indicamos el tipo, `netcat`; el canal en el que tiene que dejar la información: `MemChannel` y, por último, las dos últimas líneas de la configuración de la *source* indican la máquina y el puerto en los que van a escuchar los mensajes.
- *Channel*: indicamos que queremos que sea en memoria y la capacidad que va a tener.
- *Sink*: indicamos qué canal tiene que escuchar: `MemChannel`; el tipo, `HDFS` y la ruta en la que queremos que Flume deje los mensajes.

Una vez tenemos el fichero de configuración en nuestra máquina, solo tenemos que ejecutar el agente de Flume mediante el siguiente comando:

```
$ flume-ng agent --conf conf --conf-file /home/garguello/flume.conf --name ExampleAgent -Dflume.root.logger=DEBUG,console
```

Una vez ejecutado este comando, el agente queda activo y esperando mensajes de forma indefinida hasta que lo paremos. Vamos a enviar algunos mensajes de prueba y a comprobar que los envía a HDFS de forma correcta. Mediante el comando `netcat` de unix, enviamos mensajes al agente:

```
garguello@Cloudera01:~$ nc localhost 4444
Hola
OK
Prueba
OK
Ejemplo Flume
OK
```

Vamos a ver si los mensajes se han enviado correctamente a HDFS a la ruta que habíamos definido en el fichero de configuración:

```
garguello@Cloudera01:~$ hdfs dfs -ls /user/garguello/flume_example
Found 1 items
-rw-r--r-- 3 garguello garguello 26 2020-09-20 20:30 /user/garguello/flume_example/FlumeData.1600626589805
garguello@Cloudera01:~$ hdfs dfs -cat /user/garguello/flume_example/FlumeData.1600626589805
Hola
Prueba
Ejemplo Flume
```

Vemos que Flume ha creado un fichero en la ruta que le indicamos en el fichero de configuración y, si inspeccionamos el contenido del fichero, obtenemos exactamente los mismos mensajes que escribimos a través del `netcat`.

2. Kafka

Kafka fue diseñado e implementado originalmente en LinkedIn para resolver los problemas que tenían al tratar la gran cantidad de información generada por la aplicación (*logs*, *likes*, comentarios...).

Tengamos en cuenta que hay que contextualizar el nacimiento de Kafka a finales de la década de los 2000. Por aquel entonces, las colas de mensajería solo escalaban verticalmente y la garantía de entrega primaba sobre el rendimiento. Esto implicaba que los sistemas tradicionales no tenían capacidad para procesar la gran cantidad de datos que empezaban a generar. Además, no se aseguraba que los mensajes llegasen a los consumidores si estos no estaban disponibles y los receptores, a veces, se veían saturados por la cantidad de información que enviaba el emisor. Estas limitaciones no cumplían con los requerimientos que tenían los creadores de Kafka (y, en general, los que se presuponen para un sistema *big data* en *streaming*). La administración de LinkedIn requería crear una tecnología con los siguientes objetivos:

- Desacoplar a los productores y consumidores utilizando un modelo *push-pull* para asegurar la tolerancia a los fallos.
- Tener un escalado horizontal que permitiera mejorar la robustez del sistema ante posibles caídas.
- Permitir múltiples consumidores proporcionando persistencia dentro del sistema, mejorando así la eficiencia cuando los mensajes son consumidos por varios consumidores.
- Optimizar el rendimiento para permitir el procesamiento de una gran cantidad de mensajes por segundo.

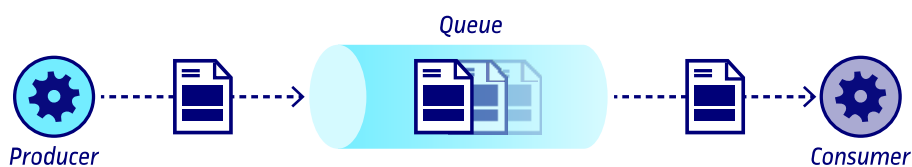
En el año 2011, se publicó Kafka como software libre y al año superó la etapa de incubación de la Apache Software Foundation. Posteriormente, los ingenieros de LinkedIn que lideraron el proyecto, fundaron en 2014 la empresa Confluent, centrada en dar soporte y formación en Kafka. Cuando Kafka se publicó era una cola de mensajería con mejoras sobre las anteriores, pero no ha dejado de evolucionar desde entonces hasta convertirse en una plataforma de transmisión de alto rendimiento y gran potencia utilizada por empresas tan conocidas como Netflix, Microsoft o Airbnb.

2.1. ¿Qué es Kafka?

Apache Kafka se describe oficialmente como una plataforma de *streaming* distribuida y diseñada para publicar, almacenar, procesar y consumir flujos de datos en tiempo real. En otras palabras, se trata de un sistema de mensajería distribuida, altamente escalable, construido con el objetivo de permitir publicar y consumir grandes volúmenes de datos con baja latencia.

2.2. La arquitectura y los componentes de Kafka

Antes de entrar de lleno en los componentes de Kafka vamos a repasar algunos conceptos previos, que nos ayudarán a entender su arquitectura.



Fuente: <https://aws.amazon.com/es/message-queue/>

Ved también

Algunos conceptos previos ya los hemos visto en el módulo «Patrones de capturas de datos dinámicos».

Empecemos por el elemento central y seguramente el más importante: la **cola de mensajería**. Este instrumento permite la comunicación entre las diferentes partes del sistema y ofrece un espacio de almacenamiento temporal en el que se acumulan los mensajes para ser leídos. La cola dispone de conectores que permiten a los generadores y consumidores escribir y leer mensajes respectivamente. Podemos imaginarlo como nuestro buzón de correos, donde el cartero va a dejar las cartas y nosotros vamos a buscarlas. La cola de mensajes va a funcionar de forma muy similar, pero para almacenar mensajes. En concreto, definimos el **mensaje** como la unidad de datos con la que vamos a trabajar, que puede ser una petición, una respuesta, un *log*, un mensaje de error, datos, etc.

Por ejemplo, en un entorno bancario tendríamos un productor que generaría un mensaje con la información de cuánto dinero se ha gastado con nuestra tarjeta, dónde y en qué.

En el módulo «Patrones de capturas de datos dinámicos» nos centramos en definir *mensajes* como un par «clave, valor», donde el valor en general es un número. En muchas aplicaciones este valor debe tener una estructura definida para garantizar la correcta propagación de la información. Por ejemplo, podemos considerar que nuestra información viene codificada en un fichero con formato JSON, XML, CSV, Avro. En este caso, esta información vendría estructurada mediante un **esquema**, aunque no sea totalmente necesario. En este punto hace falta recordar que el protocolo de comunicación tiene que establecerse entre el productor y el consumidor de datos. En este caso, Kafka

solo actúa como intermediario de los datos, de la misma forma que lo hace la empresa de correos al recoger y entregarnos una carta. En algunos casos, la información puede intercambiarse de forma cifrada para ocultarla.

Como vemos en el esquema superior en la arquitectura entran dos componentes adicionales: el productor (nuestro compañero que nos envía una carta), que genera un mensaje y el consumidor (nosotros que recibimos la carta). A continuación, vamos a ver en detalle estos dos componentes.

2.3. La instalación básica

Para dar nuestros primeros pasos con Kafka, la forma más sencilla de instalarlo es en una máquina virtual, como por ejemplo, en la máquina virtual de Cloudera que podemos obtener aquí: <<https://www.cloudera.com/downloads/cdp-data-center-trial.html>>.

La máquina virtual por defecto no trae el servicio de Kafka, pero podemos instalarlo siguiendo unos pasos que se detallan aquí: <<https://blog.clairvoyantsoft.com/installing-apache-kafka-on-clouderas-quickstart-vm-8245d8d0ebe5>>.

2.4. Los componentes

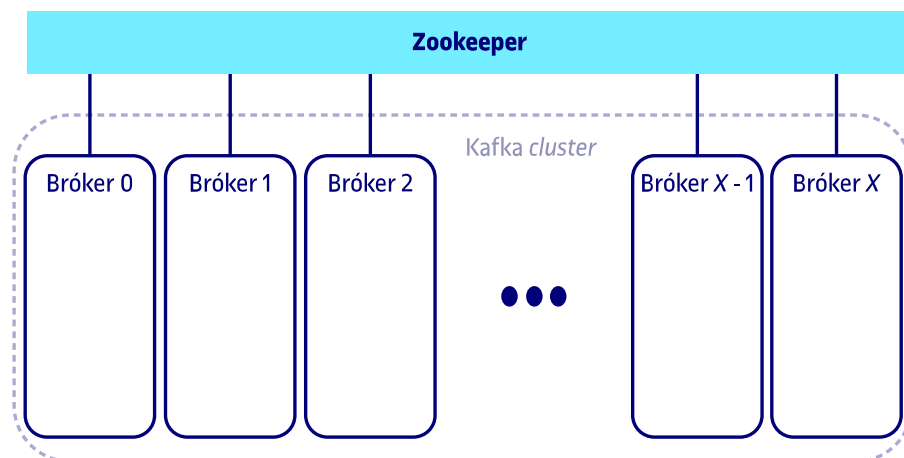
A continuación, pasaremos a ver los componentes más importantes de Kafka. Todos los componentes están muy relacionados entre ellos y no se entienden los unos sin los otros. Para entenderlos hay que verlos como partes de todo el conjunto. Podemos dividir los componentes de Kafka en dos grupos diferentes: los pertenecientes a la arquitectura interna y los que nos ofrecen una interfaz para su uso. La arquitectura interna está compuesta por el *zookeeper* y los brókeres; y los componentes de la interfaz son el *producer*, los *topics* y los *consumers*. A continuación, vamos a ver el papel de cada uno de ellos.

2.4.1. Bróker

Como ya se ha visto en el módulo «Arquitecturas *big data* para el procesamiento de datos en flujo», uno de los principales problemas de un sistema *big data* es mantener la robustez y la tolerancia a los fallos. Para garantizar estas dos características, Kafka puede ejecutarse en una o en varias máquinas. La ejecución distribuida (varias máquinas) de Kafka permite guardar el estado del sistema y los mensajes para garantizar que no se va a perder ningún mensaje que no haya sido consumido. En este caso, Flume no garantiza estas características.

En proyectos e instalaciones reales, lo más habitual es que Kafka esté instalado en un clúster y mayoritariamente la instalación en una sola máquina queda relegada a la ejecución de tests de validación. En un entorno distribuido, cada uno de los servidores que lo componen es lo que llamamos **bróker**. El conjunto de todos los brókeres será lo que conocemos como **clúster de Kafka**. Cla-

ramente, estos servidores no funcionan de manera descoordinada, sino que se coordinan mediante otro elemento fundamental, Zookeeper. A continuación, podéis ver un esquema de la estructura de Kafka:



Fuente: <https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-2-2/>

2.4.2. Zookeeper

Zookeeper es una herramienta *open-source* de Apache para la gestión de clústeres y está orientada a proporcionar un servicio de coordinación para aplicaciones distribuidas. Zookeeper funciona de manera individual y no está atado al desarrollo de Kafka. Debemos entender que la ejecución y el almacenamiento distribuido de datos es una necesidad intrínseca de las herramientas de *big data* y está relacionada con la escalabilidad horizontal. Zookeeper ofrece herramientas para la coordinación de las diferentes máquinas que componen el sistema. Por lo tanto, muchos otros servicios en el mundo *big data* usan *zookeeper* para funciones coordinativas.

No es necesario profundizar mucho en el conocimiento de Zookeeper para poder entender Kafka, simplemente necesitamos saber que será el coordinador del sistema compuesto por los brókeres y que guardará toda la metainformación. Por ejemplo, cuando queramos ver qué tópicos tenemos creados, se lo preguntaremos a Zookeeper que tendrá toda la metainformación de nuestro sistema Kafka.

Ejemplos

Podéis ver algunos ejemplos de Zookeeper en:

<https://zookeeper.apache.org/doc/r3.6.1/zookeeperUseCases.html>.

Enlace recomendado

Para profundizar sobre Zookeeper, se puede consultar la extensa documentación oficial de Apache: <https://zookeeper.apache.org/doc/current/index.html>.

2.4.3. Producers

Un productor Kafka es un cliente que envía mensajes de forma remota a los brókeres. El cliente envía y hace un *push* ('empuje', en castellano) de la información hasta el servidor Kafka o bróker. Los productores de información pueden ser cualquier tipo de sistema que genere y envíe información. Como ya hemos visto, es necesario que la información generada tenga el formato clave o valor.

Como ejemplo podemos imaginar que tenemos un sensor que, cada minuto, mide la temperatura y la humedad ambiental. Cada minuto nuestro sensor genera dos mensajes de tipo clave o valor, en el que la clave es la fecha de lectura y el valor (la información) es,

por un lado, la temperatura y, por otro, la humedad. Así, cada minuto se nos van a generar dos mensajes del tipo `<04-04-2020 08:00:00, 15 C>`, `<04-04-2020 08:00:00, 84 %>`. Como hemos dicho, los mensajes también pueden tener estructura interna. Por ejemplo, este mismo mensaje podría tener formato JSON: `<04-04-2020 08:00:00, {temperatura: 15, humedad: 84}>`. Insistimos en que el protocolo de comunicación está definido entre el productor y el consumidor de los datos.

Ejemplo práctico

Kafka tiene la funcionalidad de consola `kafka-console-producer` para inicializar un productor de mensajes. Al ejecutar la siguiente sentencia, todo lo que escribamos en pantalla será añadido secuencialmente a la cola Kafka.

```
$ kafka-console-producer --broker-list Cloudera01:9092 --topic uocDemoTopic
```

```
[cloudera@quickstart config]$ kafka-console-producer --broker-list localhost:9092 --topic uocDemoTopic
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.21.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hola
segundo mensaje
```

El productor publica estos mensajes en un **topic** de Kafka que es el siguiente componente de la arquitectura que vamos a estudiar.

2.4.4. Topics

Los *topics* o temas son la forma en que los mensajes que genera el productor se categorizan en Kafka. Como analogía, podemos pensar en una tabla de una base de datos y sus registros: las tablas serían los *topics* y, los registros de esta tabla, los mensajes que genera el productor. Los mensajes que se producen se van colocando de forma ordenada en el *topic*. Cada uno de estos mensajes dentro del *topic* va a tener un identificador único, que es lo que conocemos como *offset*. Veremos más adelante el detalle de los *offsets*. Dentro de Kafka, lo habitual es que existan varios *topics* (no hay limitación). Con el ejemplo que poníamos antes del sensor que recoge datos de temperatura y humedad, podríamos tener dos *topics* en Kafka, uno al que el sensor le envía los datos de temperatura y otro al que le envía los datos de la humedad. A menos que se defina algún criterio de persistencia, el *topic* almacena de forma permanente los mensajes, aunque es habitual establecer un criterio de persistencia para no saturar la memoria, que suele ser por tiempo (una semana, un mes...) o por tamaño (por ejemplo, cuando el *topic* supera 1GB).

Crear un *topic* es muy sencillo con la funcionalidad de Kafka `kafka-topics`. Solo tenemos que indicarle el nombre, que en nuestro caso será `uocDemoTopic`, el número de particiones y el factor de replicación. De momento pondremos los dos últimos valores como 1.

```
$ kafka-topics --create --zookeeper Cloudera01:2181/kafka --  
topic uocDemoTopic --partitions 1 --replication-factor 1
```

Si ejecutamos el comando vemos que se crea el *topic*:

```
[cloudera@quickstart config]$ kafka-topics --create --zookeeper localhost:2181/kafka --topic uocDemoTopic --partitions 1 --replication-factor 1  
SLF4J: Class path contains multiple SLF4J bindings.  
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.21.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.  
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]  
Created topic "uocDemoTopic".
```

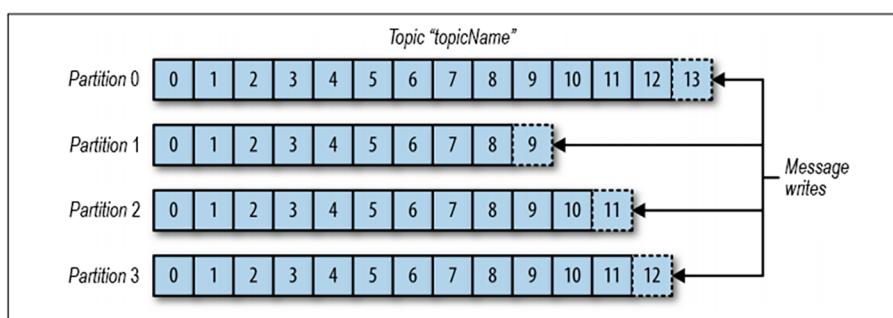
Para asegurarnos de que el *topic* se ha creado, vamos a listar todos los *topics*. Para ello, ejecutamos el siguiente comando:

```
$ kafka-topics --list --zookeeper Cloudera01:2181/kafka
```

```
[cloudera@quickstart config]$ kafka-topics --list --zookeeper localhost:2181/kafka  
SLF4J: Class path contains multiple SLF4J bindings.  
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.21.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.  
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]  
uocDemoTopic
```

Para crear el *topic*, hemos visto que hemos tenido que introducir dos parámetros obligatorios: las particiones y los factor de replicación. Vamos a ver, a continuación, qué son estos conceptos.

Los *topics* se pueden dividir o fragmentar en partes y cada una de estas partes es lo que conocemos como **partición**. El número mínimo de particiones es 1, lo que quiere decir que el *topic* no está particionado. Volviendo a la analogía de la base de datos, las particiones equivaldrían a tener toda la tabla (*topic*) subdividida en tablas más pequeñas (particiones). Esta sería la representación de un *topic* con cuatro particiones:

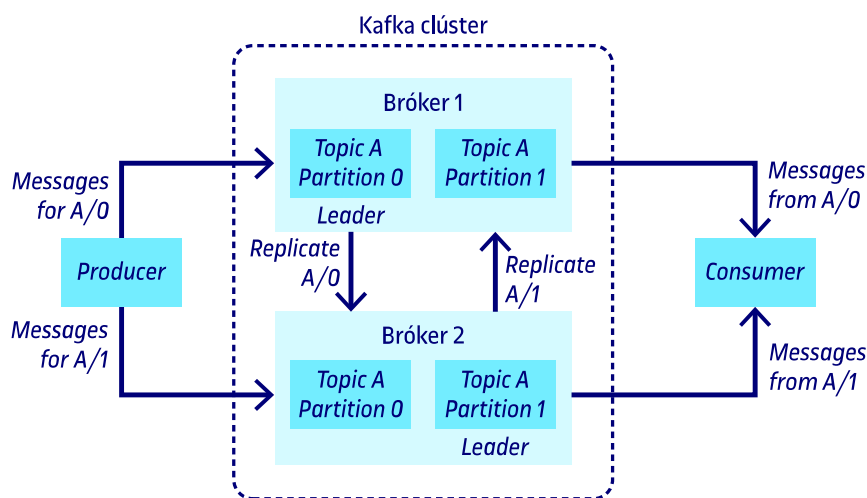


Fuente: Neha Narkhede; Gwen Shapira; Todd Palino (2017). *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale* (pág. 7). O'Reilly Media, Inc.

Los mensajes se van escribiendo de forma ordenada al final de cada partición del *topic*; así, el mensaje 0 es el más antiguo de la partición y, el último número, el más reciente.

Las particiones son la forma que tiene Kafka de proveer redundancia y escalabilidad ya que cada partición puede estar alojada en un servidor diferente, lo que significa que un *topic* puede ser escalado horizontalmente a través de

muchos servidores. A su vez, cada partición puede estar replicada en varios servidores, lo que asegura la replicación de la información. Al número de veces que una partición está replicada lo llamamos **factor de replicación**. Cada partición tiene un servidor que actúa como líder (*leader partition*) y cero o más que actúan como seguidores (*follower partition* o *partitions*). El líder maneja todas las peticiones de lectura y escritura y los seguidores solo replican al líder en caso de que este falle. Si esto sucede, un seguidor lo reemplazará y tomará su rol para ser un nuevo líder. Cada servidor actúa como líder de algunas particiones y como seguidor de otras. Un *topic* con factor de replicación N , tolerará $N-1$ fallos en los servidores sin que se pierdan mensajes.



Fuente: Neha Narkhede; Gwen Shapira; Todd Palino (2017). *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale* (pág. 8). O'Reilly Media, Inc.

Las particiones provocan que la ordenación temporal de los mensajes no se pueda asegurar a nivel de todo el *topic*. Los mensajes están ordenados temporalmente dentro de una partición, pero no dentro de todo el *topic*. Si miramos la imagen anterior, Kafka no asegura que el mensaje 9 (mensaje con *offset* 9) de la partición 1 haya llegado antes que el mensaje 13 (mensaje con *offset* 13) de la partición 0. Sin embargo, dentro de una misma partición sí que están completamente ordenados.

Pero ¿cómo y quién decide qué mensaje va a qué partición? Este es un trabajo del productor. Hay veces que no nos importa qué mensaje esté en qué partición, porque todos los mensajes son independientes entre sí y no hay relación entre los datos; en ese caso, el productor no hará nada y Kafka asignará una partición a cada mensaje mediante un algoritmo balanceador de tipo *round-robin*. Sin embargo, hay ocasiones en las que nos interesa que ciertos mensajes vayan siempre a la misma partición. Por ejemplo, si estamos leyendo operaciones de tarjetas de crédito, nos interesa que las operaciones del mismo cliente vayan siempre a la misma partición para asegurarnos de que siempre leemos los eventos de nuestro cliente en orden cronológico; si estuviesen en

Enlace recomendado

Podéis consultar más información sobre un algoritmo balanceador de tipo *round-robin* en el siguiente enlace:
<https://www.udg.co.cu/cmap/sistemas_operativos/planificacion_cpu/round_robin/Round%20Robin.html>.

distintas particiones, no podríamos asegurar esto. El procedimiento por el que asignamos a un determinado mensaje una determinada partición se conoce como *partitioner*.

Por último, hay que destacar que podemos tener entre 1 y n productores en nuestra arquitectura Kafka. Varios productores pueden escribir en el mismo *topic*. Volviendo al ejemplo del sensor, si en vez de tener un único sensor tenemos varios, representando a distintas estaciones meteorológicas, cada uno de ellos será un *producer* que dejará la información en el mismo *topic* y, como acabamos de ver, cada sensor siempre dejará la información en la misma partición para que esté ordenada cronológicamente.

Hasta ahora tenemos un *producer* que genera unos mensajes y los escribe en un *topic* particionado. A continuación, vamos a presentar la siguiente pieza clave de la arquitectura: el consumidor de la información.

2.4.5. Consumers

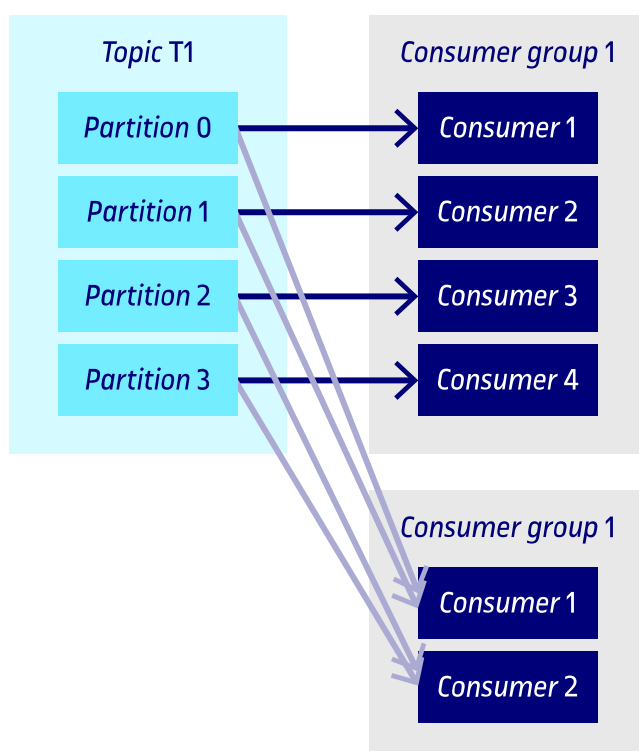
Un consumidor Kafka es un cliente que recibe mensajes desde el clúster de Kafka. Estos mensajes se obtienen «cogiendo» (realizando un *pull*) mensajes. El consumidor **se suscribe a uno o varios *topics***, de los que lee la información en el orden en que se ha producido. Podemos tener varios consumidores distintos leyendo del mismo *topic*. Insistimos en una de las claves de Kafka: una vez que el mensaje es leído por un consumidor, no se borra del bróker en el que reside, salvo que haya algún criterio de borrado definido. Esto es lo que permite tener diferentes consumidores y capacidades de reproceso.

Para cada uno de los *topics* a los que se suscribe, el consumidor lleva un registro de cuál es el último mensaje que ha leído de cada *topic*. Como hemos dicho antes, esto es lo que conocemos como *offset*. Cuando el consumidor decida, podrá guardar la información del mensaje que ha leído en un sistema externo. En las nuevas versiones de Kafka, se utiliza un *topic* exclusivo de Kafka (*_consumer_offsets*) en el que los consumidores guardan esta información para, en el caso de parar el cliente o producirse un error, saber qué mensaje es el último que se había leído y poder retomar la lectura en el lugar en el que se había quedado. Este proceso de guardar el *offset* por el que va el consumidor se denomina *commit*.

Imaginemos que tenemos una aplicación que lee mensajes de un *topic* de Kafka, los valida de alguna forma y los escribe en otro sistema de almacenamiento. En nuestro ejemplo de los sensores, por ejemplo, el consumidor recibirá cada minuto la medición de la temperatura, validará que el valor es un número y que puede corresponder a una temperatura atmosférica y, si pasa la validación, lo escribe en una base de datos relacional. Nuestro consumidor puede funcionar perfectamente recibiendo datos cada minuto, pero ¿qué pasaría si nuestro sensor empieza a enviar mensajes cada segundo? Muy probablemente, nuestro consumidor empezaría a recibir más información de la que es capaz

de procesar. Surge de manera natural la necesidad de escalar el consumo de los *topics*. Al igual que muchos productores escriben en el mismo *topic*, tendremos múltiples consumidores leyendo el mismo *topic*. Es lo que conocemos como **grupos de consumidores**, que trabajan juntos para consumir los mensajes de un *topic*.

En la siguiente figura tenemos como ejemplo un único *topic* con cuatro particiones que son consumidas por dos grupos de consumidores. El grupo 1 tiene cuatro consumidores y cada uno de los cuatro consumidores solo lee una de las particiones del *topic*. Mientras que el grupo 2, solo tiene dos consumidores, por lo que el primer consumidor del grupo lee dos particiones y el segundo, otras dos.



Fuente: Neha Narkhede; Gwen Shapira; Todd Palino (2017). *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale* (págs. 64-65). O'Reilly Media, Inc.

Es importante recalcar que una partición **solo puede ser leída por un consumidor de cada grupo**. Varios consumidores de distintos grupos pueden leer la misma partición, pero solo uno de cada grupo puede leerla.

El principal objetivo de la partición de los *topics* es aumentar el rendimiento al consumir, ya que podremos tener diferentes consumidores del grupo leyendo particiones distintas del mismo *topic* a la vez.

Cada consumidor se encarga de leer una o varias particiones, como hemos visto. Cuando añadimos un nuevo consumidor, este va a empezar a leer una o varias particiones que antes estaba leyendo otro consumidor. De la misma forma, cuando un consumidor deja de funcionar o si se añaden nuevas particiones al *topic*, otros tienen que hacerse cargo del trabajo que estaba haciendo.

Este cambio en las particiones que lee un consumidor es lo que conocemos como **balanceo**. El balanceo es fundamental porque es lo que proporciona al grupo de consumidores una alta disponibilidad y una escalabilidad horizontal. El encargado de hacer el trabajo de balanceo, es decir, de asignarle a cada consumidor del grupo la partición que tiene que leer, es uno de los brókeres que conocemos como *group coordinator*.

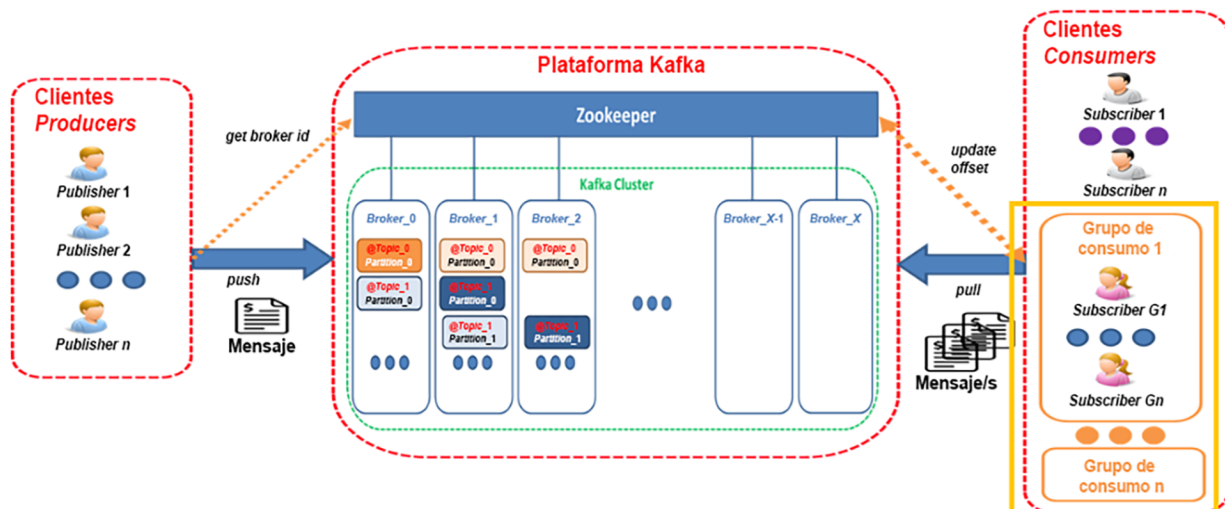
Kafka tiene la funcionalidad de consola `kafka-console-consumer` para crear nuevos consumidores. Si ejecutamos el siguiente comando, empezaremos a consumir todos los mensajes del *topic* `uocDemoTopic`, en el que en el apartado de los productores dejamos dos mensajes:

```
$ kafka-console-consumer --new-consumer --bootstrap-server
localhost:9092 --topic uocDemoTopic
```

```
[cloudera@quickstart ~]$ kafka-console-consumer --new-consumer --bootstrap-server localhost:9092 --topic uocDemoTopic
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.21.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/kafka/libs/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hola
segundo_mensaje
^CProcessed a total of 2 messages
```

Llegados a este punto hemos presentado los elementos más importantes que componen el ecosistema Kafka. En la siguiente figura podemos verlos todos juntos. Observamos que tenemos el *zookeeper* como coordinador y x brókeres. Dentro de la plataforma tenemos dos *topics*, el `topic_0` y el `topic_1`. El `topic_0` tiene una única partición que se llama `partition_0` y que está replicada tres veces en los tres primeros brókeres. El `topic_1` consta de dos particiones, `partition_0` y `partition_1`, que en este caso están replicadas solo dos veces. Las particiones que están resaltadas en un color más oscuro son las *leader partition* que manejan todas las peticiones de lectura y escritura.

En este diagrama encontramos que nuestra plataforma tiene n productores de mensajes. Estos *producers* generarán mensajes que **empujarán** a los *topics* y particiones según estén definidos. Por último, tenemos diferentes tipos de consumidores de la información. Por un lado, tenemos n suscriptores independientes, que consumirán toda la información de uno o de los dos *topics* ellos solos. Por otro lado, tenemos n grupos de consumidores, formados cada uno de ellos por G_n consumidores. Estos consumidores de grupo actuarán conjuntamente para leer la información de los *topics*.



Fuente: <https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-3-conceptos-basicos-extra/>

2.5. La tolerancia a los fallos

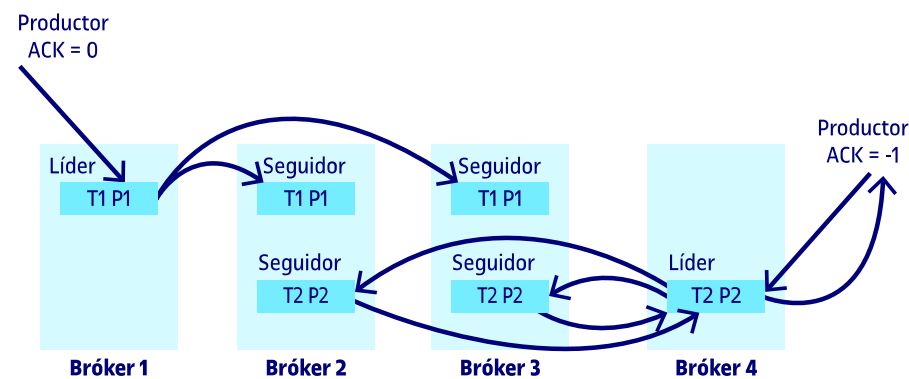
Como decíamos al principio, uno de los puntos clave de Kafka es la tolerancia a los fallos que tendremos tanto al producir mensajes como al consumirlos.

2.5.1. La tolerancia a los fallos al producir

Los productores nos dan la posibilidad de asegurarnos que los mensajes han sido recibidos gracias a la variable de configuración `request.required.acks`. Un ACK (del inglés, *acknowledgement*) es un mensaje, un acuse de recibo, que el destino de la comunicación envía al origen de esta para confirmar la recepción de un mensaje. Lo podemos configurar de tres formas:

- **ACK = 1.** Nos asegura que la replicación líder recibe los datos, pero sin confirmar la escritura a los *followers* o a las réplicas. Si el *leader* falla después de enviar el ACK, entonces el mensaje puede perderse.
- **ACK = -1.** Nos asegura que todas las réplicas reciben los datos. La partición líder espera a que el mensaje sea escrito en todos los *followers* o las réplicas antes de mandar el ACK al productor. Necesitamos que el *topic* tenga al menos una réplica para poder usarlo.
- **ACK = 0.** El productor no espera respuesta de confirmación de ningún bróker. El mensaje se pierde si la partición *leader* se cae.

La petición de confirmación tiene una espera y genera un retardo. Por lo tanto, es necesario estudiar en qué casos es necesario y en cuáles no.



Fuente: Elaboración propia

2.5.2. La tolerancia a los fallos al consumir

El mecanismo que utilizan los consumidores para evitar perder mensajes ya lo hemos visto con anterioridad y es lo que conocemos como *offset* y *commit*. Cuando un consumidor se cae, su *offset* indicará por qué mensaje se ha quedado, impidiendo así la pérdida de mensajes. Sin embargo, esto puede generar que en alguna ocasión procesemos mensajes que ya habíamos leído previamente. Por ejemplo, si tenemos un consumidor leyendo mensajes y procesándolos, pero justo deja de funcionar antes de realizar el *commit*, cuando se vuelva a levantar va a empezar a leer los mensajes desde el último *commit* realizado, procesando los mensajes que ya había leído con anterioridad.

En muchas ocasiones, por el hecho de que el consumidor haya recibido un dato, no queremos marcarlo como leído correctamente. Un ejemplo habitual, es cuando estamos utilizando Kafka para procesar mensajes y escribir estos mensajes en una base de datos relacional. En este caso, no queremos hacer el *commit* hasta que nuestro mensaje no se haya insertado correctamente en la base de datos. Para poder realizar esto, tenemos que desactivar el *commit* automático con la propiedad `enable.auto.commit = false` y realizar un *commit* manual cuando los datos se hayan insertado en la base de datos.

Debemos tener en cuenta que el *offset* es un recurso que los consumidores tendrán en memoria y solo consultarán a *zookeeper* o a Kafka cuando se inicie o rebalancee un consumidor.

Podemos estar sin realizar el *commit* horas sin perder ningún dato, ya que los consumidores llevarán su propio registro de cuál es el último *offset* que han leído, pero en el caso de que caiga y no realicemos el *commit*, perderemos el registro y cuando el consumidor se levante empezará a leer por el último mensaje que haya registrado un *commit*.

2.6. La configuración

2.6.1. La profundidad histórica

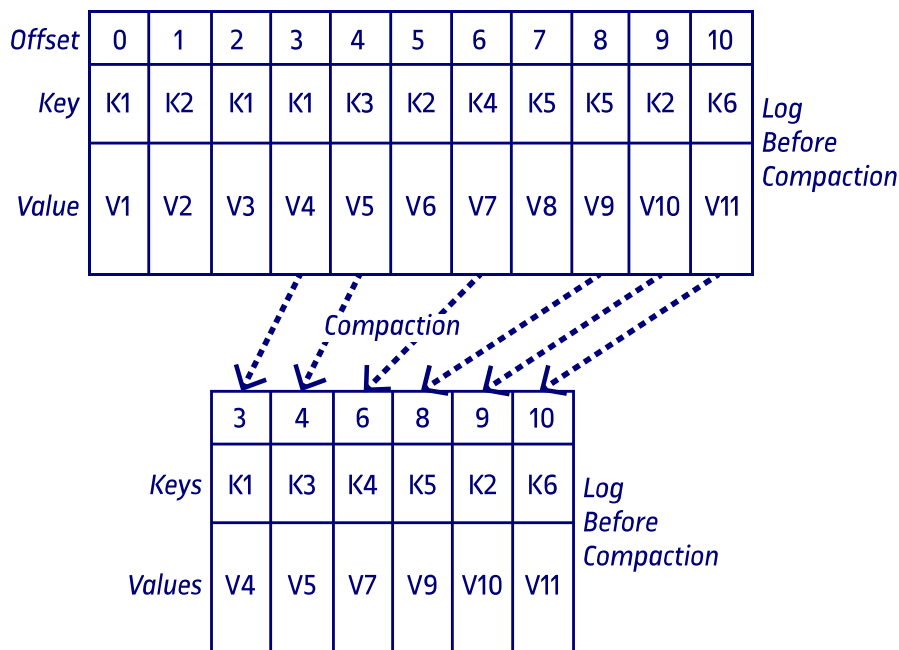
Cuando escribimos un mensaje en la partición de un *topic*, estamos escribiendo información en el bróker (servidor) en el que esta partición está alojada; es decir, estamos escribiendo en el disco duro de una máquina. En este sentido, es importante señalar que el rendimiento de la aplicación será mejor si cada partición está en un servidor diferente para asegurar la escritura secuencial en el disco. Los discos duros tienen una capacidad limitada, por lo que es fundamental gestionar la profundidad histórica que queremos almacenar en nuestros brókeres de Kafka. Para ello, tenemos dos propiedades que podemos modificar:

- `log.retention.hours`: es el tiempo que Kafka retendrá los mensajes antes de borrarlos. Por defecto son 168 horas o, lo que es lo mismo, siete días.
- `log.segment.bytes`: es el tamaño máximo de una partición antes de que se empiecen a borrar los mensajes más antiguos. Por defecto es 1 GB.

Esta configuración se establece a nivel de clúster, pero se puede modificar para cada uno de los *topics*.

2.6.2. La compactación

Hay casos en los que no podemos eliminar datos antiguos por el mero hecho de serlo. Imaginemos que estamos utilizando Kafka para almacenar las direcciones de nuestros clientes. Si nuestro cliente no cambia de dirección en años, aun así queremos seguir teniendo esa información. En estos casos, lo que se utiliza es la compactación. Kafka se queda con el mensaje más reciente por clave. En nuestro ejemplo, la clave sería el cliente y el valor, la dirección. Cuando un cliente cambia de dirección y aplicamos la compactación, el mensaje con su dirección antigua se borrará y solo almacenaremos la nueva. Como podemos ver en la siguiente figura, al aplicar la compactación nos quedamos con el par clave valor más reciente, es decir, con el mayor *offset*.



Fuente: <https://programmer.group/analysis-of-kafka-log-compaction.html>

2.6.3. Las cuotas

Kafka tiene la habilidad de establecer cuotas a los productores y consumidores de la información para evitar que un cliente monopolice los recursos del clúster. Se configura el tráfico que un cliente específico puede tener con un bróker en *bytes* por segundo. Cuando un bróker identifica que un cliente está excediendo su cuota, ralentiza sus respuestas al cliente hasta que se vuelve a situar por debajo de la cuota permitida.

2.6.4. La seguridad

La seguridad es siempre una preocupación en cualquier entorno de datos. A efectos de seguridad, Kafka permite encriptar la información en el movimiento entre la fuente y Kafka y entre Kafka y el destino. También se puede configurar la autenticación de los usuarios con Kerberos y, a nivel de red, con TLS. Además, se pueden configurar permisos a nivel de *topic* (los ACL). Por último, Kafka provee de un sistema de auditoría de acceso que permite monitorizar los accesos a todos los topics.

2.7. Ejemplo completo

Hasta ahora hemos visto cómo se puede interactuar con Kafka mediante la línea de comandos. Esta forma de interacción no es la habitual, aunque es muy útil para depurar problemas o hacer pequeños tests. En el ejemplo que vamos a ver a continuación, pasamos a una forma más estándar de interacción, mediante un lenguaje de programación. Para ello, vamos a ver un ejemplo sencillo, pero completo, del uso de Kafka con la Api de Python. Podéis encontrar

el ejemplo en los ficheros adjuntos a este documento. En primer lugar, crearemos un *topic*; a continuación, crearemos un productor que escriba unos datos de prueba en este *topic* y, por último, un consumidor que los lea.

Creamos un *topic* con nombre `garguelloTopicTest`, una partición y 1 como factor de replicación y comprobamos que está correctamente creado:

```
!kafka-topics --create --zookeeper localhost:2181/kafka --topic garguelloTopicTest --partitions 1 --replication-factor 1
Created topic "garguelloTopicTest".

!kafka-topics --list --zookeeper localhost:2181/kafka
__consumer_offsets
garguelloTopic
garguelloTopic2
garguelloTopicTest
uocDemoTopic
uocDemoTopic2
uocDemoTopic3
uocDemoTopic4
```

Vemos que nuestro *topic* se ha creado correctamente. Ahora vamos a crear un productor que escriba en este *topic* los números naturales hasta el cien.

```
from kafka import KafkaProducer
import numpy as np

#Definimos el productor
producer = KafkaProducer(bootstrap_servers='Cloudera02:9092')

#Bucle para enviar los primeros 100 numeros naturales al topic garguelloTopicTest
for i in range(1, 100):
    producer.send('garguelloTopicTest',value=bytes(str(i),'utf-8'))
```

Una vez que lo ejecutamos, ya tendremos en el *topic* los datos. Por último, vamos a crear un consumidor para comprobarlo:

```
from kafka import KafkaConsumer

#Definimos el consumidor
consumer = KafkaConsumer('garguelloTopicTest',bootstrap_servers='Cloudera02:9092',
                          auto_offset_reset='smallest', consumer_timeout_ms =10000)

#Para cada mensaje leído del consumidor, mostramos por pantalla toda su información
for message in consumer:
    print(message)
```

Cuando lo ejecutamos, vemos como el consumidor lee todos los mensajes que había en el *topic*, ya que le hemos indicado que el *offset* fuese el más pequeño. Obtenemos el siguiente resultado por la pantalla y vemos que, a parte del dato, que es el *value*, tenemos más información asociada como el *timestamp* o la partición.

```

ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=0, timestamp=1596472009606, timestamp_type=0, key=None, value=b'1', checksum=-905532249, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=1, timestamp=1596472009606, timestamp_type=0, key=None, value=b'2', checksum=1393556765, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=2, timestamp=1596472009607, timestamp_type=0, key=None, value=b'3', checksum=-875911499, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=3, timestamp=1596472009607, timestamp_type=0, key=None, value=b'4', checksum=1437479702, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=4, timestamp=1596472009607, timestamp_type=0, key=None, value=b'5', checksum=581501824, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=5, timestamp=1596472009607, timestamp_type=0, key=None, value=b'6', checksum=-1147121094, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=6, timestamp=1596472009607, timestamp_type=0, key=None, value=b'7', checksum=-861445460, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=7, timestamp=1596472009607, timestamp_type=0, key=None, value=b'8', checksum=1545109309, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=8, timestamp=1596472009607, timestamp_type=0, key=None, value=b'9', checksum=723472299, serialized_key_size=-1, serialized_value_size=1)
ConsumerRecord(topic='garguelloTopicTest', partition=0, offset=9, timestamp=1596472009607, timestamp_type=0, key=None, value=b'10', checksum=2137755231, serialized_key_size=-1, serialized_value_size=2)

```

2.8. Los casos de uso

Kafka tiene infinidad de usos prácticos y es una herramienta cada vez más utilizada. Tiene aplicaciones como el agente de mensajería que mejoran el rendimiento de las aplicaciones tradicionales, como por ejemplo, en las aplicaciones que necesitan enviar notificaciones a los usuarios (correos electrónicos, sms...).

También es comúnmente usado en aplicaciones para procesar métricas y *logs* y tiene la habilidad de recopilar mensajes similares de diferentes fuentes. Por ejemplo, en el seguimiento de la actividad de las webs: monitorización y agregación de métricas y estadísticas sobre búsquedas o accesos, etc. O para monitorizar el estado de los servidores de una empresa, recogiendo *logs* y procesándolos en tiempo real.

Kafka nos permite el procesamiento de datos en tiempo real (*stream processing*). Una de las aplicaciones de Kafka más actuales e interesantes está en el ámbito del internet de las cosas (*Internet of Things*). Muchas industrias tratan de predecir a partir de sensores en sus máquinas cuándo van a necesitar mantenimiento. Arreglar una máquina antes de que se estropee por completo puede ahorrar enormes costes a muchas empresas manufactureras.

Bibliografía

Narkhede, Neha; Gwen Shapira; Todd Palino (2017). *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

Shreedharan, Hari (2014). *Using Flume: Flexible, Scalable, and Reliable Data Streaming*. O'Reilly Media, Inc.

