Procesado de datos

5. Fuentes de datos avanzadas para streaming

Boni García

http://bonigarcia.github.io/
boni.garcia@uc3m.es

Departamento de Ingeniería Telemática
Escuela Politécnica Superior

2020/2021

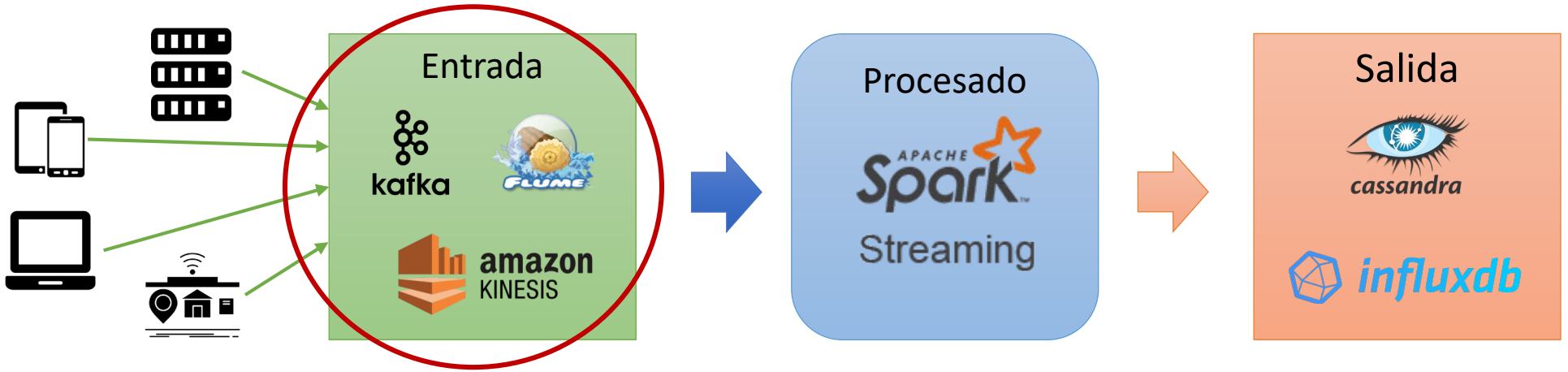
uc3m | Universidad Carlos III de Madrid

Contenidos

Contenidos

1. Introducción

- **Spark Streaming** es una extensión del framework Spark que permite procesar **datos en streaming** (en tiempo real)
- Spark Streaming implementa un data pipeline en diferentes etapas:
  1. Recepción de datos de entrada (data ingestion) de diferentes tipos fuentes:
     - Básicas: sistema de ficheros y sockets TCP (vistos en el tema anterior)
     - Avanzadas: Apache Kafka, Apache Flume, y Amazon Kinesis (los vemos en este tema)
  2. Procesado con Spark Streaming usando una técnica llamada micro-batching
  3. Entrega de resultados para almacenamiento o visualización



1. Introducción - Dependencias externas

- Las fuentes de datos avanzadas están disponibles en Spark Streaming a través de **clases de utilidad**
- Estas clases no están disponibles por defecto en Spark, y habrá que usarlas a través de **dependencias externas**
- Típicamente, estas dependencias estarán desarrolladas en Scala, y empaquetadas a través de **sbt** (https://www.scala-sbt.org/)
- Estas dependencias se distribuyen como ficheros JAR (Java Archive) disponibles en repositorios públicos de artefactos binarios Java, como Maven Central (https://search.maven.org/)

1. Introducción - Dependencias externas

- Cada dependencia tiene unas **coordenadas** que lo identifica unívocamente:
  - `groupId`: Identificador de la organización
  - `artifactId`: Identificador del proyecto
  - `version`: Versión del artefacto (normalmente usando versionado semántico)
- La siguiente tabla resume las dependencias que vamos a necesitar para usar las fuentes avanzadas de datos en Spark Streaming

Fuente de datos	groupId	artifactId	version
Amazon Kinesis	`org.apache.spark`	`spark-streaming-kinesis-asl_2.11`	`2.4.7`
Apache Flume	`org.apache.spark`	`spark-streaming-flume_2.11`	`2.4.7`
Apache Kafka	`org.apache.spark`	`spark-streaming-kafka-0-8_2.11` `spark-sql-kafka-0-10_2.11`	`2.4.7`

1. Introducción - Dependencias externas

- Hay varias formas de resolver estas dependencias en nuestros programas Python:

1. Mediante línea de comandos y el script `spark-submit`:

```
$ spark-submit --packages groupId:artifactId:version program-name.py <args>
```

2. Incluir las coordenadas de la dependencia usando el parámetro de configuración `spark.jars.packages` en el contexto o sesión Spark (separadas por comas si hay más de una dependencia):

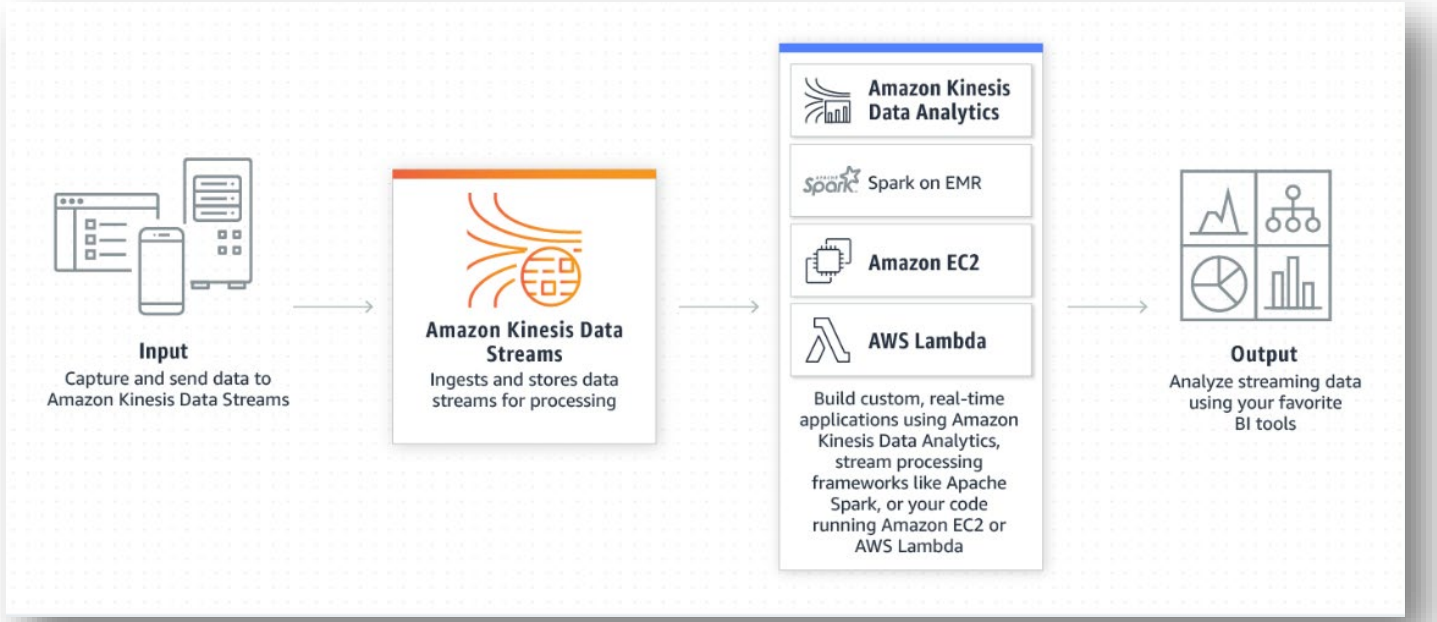
```
sc = SparkContext(master="local[*]",
                  appName="Kafka-DStream-StdOut",
                  conf=SparkConf()
                  .set("spark.jars.packages", "org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.7"))
```

Por simplicidad, en los ejemplos de clase vamos a usar esta solución. En proyectos reales, esta solución no sería demasiado buena, ya que estamos especificando la configuración de dependencias en el propio código de nuestra aplicación (*hardcoded*)

Contenidos

2. Amazon Kinesis

- **Amazon Kinesis** es una plataforma gestionada (*managed*) para la recolección, análisis, y procesado de datos en tiempo real (streaming)



amazon KINESIS

https://aws.amazon.com/kinesis/

2. Amazon Kinesis

- En PySpark, podemos crear **DStreams** procedente de Amazon Kinesis usando la clase de utilidad **`KinesisUtil`**, con los siguientes parámetros:
  - Contexto de streaming
  - Nombre de aplicación
  - Nombre de flujo
  - URL del flujo
  - Nombre de región AWS (por ejemplo us-east-2, eu-west-1, etc.)
  - Posición inicial dentro del flujo de datos
  - Intervalo en el que Kinesis guarda datos en el flujo (típicamente será el mismo intervalo usado para los micro-batches)

```
from pyspark.streaming.kinesis import KinesisUtils

stream = KinesisUtils.createStream(streamingContext, [Kinesis app name], [Kinesis stream name],
[endpoint URL], [region name], [initial position], [checkpoint interval])
```

https://spark.apache.org/docs/latest/api/python/pyspark.streaming.html

2. Amazon Kinesis

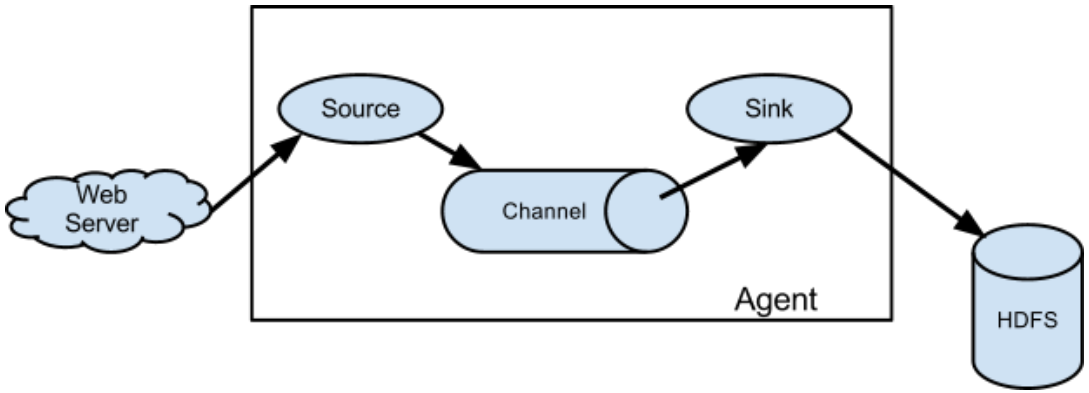
Este ejemplo recibe un flujo de texto procedente de Amazon Kinesis y cuenta las palabras que se van recibiendo en cada segundo


```
import sys

from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kinesis import KinesisUtils, InitialPositionInStream

if __name__ == "__main__":
    if len(sys.argv) != 5:
        print(f"Usage: {sys.argv[0]} <app-name> <stream-name> <endpoint-url> <region-name>",
              file=sys.stderr)
        sys.exit(-1)

    # Local SparkContext and StreamingContext (batch interval of 1 second)
    sc = SparkContext(master="local[*]",
                      appName="Kinesis-DStream-StdOut",
                      conf=SparkConf()
                      .set("spark.jars.packages", "org.apache.spark:spark-streaming-kinesis-asl_2.11:2.4.7"))
    ssc = StreamingContext(sc, 1)

    # 1. Input data: create a DStream from Kinesis
    appName, streamName, endpointUrl, regionName = sys.argv[1:]
    stream = KinesisUtils.createStream(
        ssc, appName, streamName, endpointUrl, regionName, InitialPositionInStream.LATEST, 1)

    # 2. Data processing: word count
    count = (stream.flatMap(lambda line: line.split(" "))
             .map(lambda word: (word, 1))
             .reduceByKey(lambda x, y: x + y))

    # 3. Output data: show result in the console
    count.pprint()

    ssc.start()
    ssc.awaitTermination()
```

Contenidos

3. Apache Flume

- **Apache Flume** es un sistema open-source desarrollado en Java para la recolección de grandes cantidades de logs de forma distribuida, eficiente y fiable
- Un **agente** Flume es un proceso (JVM) que redirige datos de entrada de tipo texto (logs) desde una entrada (**source**), por ejemplo, un servidor web, hacia una salida (**sink**), por ejemplo, un sistema de ficheros HDFS o una conexión TCP, a través de un canal (**channel**) determinado, por ejemplo, en memoria



https://flume.apache.org/

3. Apache Flume - Componentes

- Algunas de las entradas de datos (**source**) posibles en Flume son:

Entrada	Tipo	Descripción
Apache Avro	`avro`	Formato de serialización de datos Apache Avro (https://avro.apache.org/)
Apache Thrift	`thrift`	Lenguaje de definición de interfaces (https://thrift.apache.org/)
Exec	`exec`	Salida de un comando Unix por su salida estándar (stdout)
JMS	`jms`	Cola de mensajes Java (Java Message Service), como ActiveMQ (https://activemq.apache.org/)
Directorio	`spooldir`	Se vigila el contenido de un directorio en el que se van añadiendo datos (spooling directory)
Kafka	`org.apache.flume.source.kafka.KafkaSource`	Cola de mensajes Apache Kakfa (https://kafka.apache.org/)
NetCat	`netcat` `netcatudp`	Socket TCP o UDP
Generador de secuencia	`seq`	Generador de secuencia que empieza en 0 y se va incrementado en el tiempo (usado para pruebas)

https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#flume-sources

3. Apache Flume - Componentes

- Algunas de las salida de datos (**sink**) posibles en Flume son:

Salida	Tipo	Descripción
HDFS	`hdfs`	Sistema de ficheros HDFS (Hadoop Distributed File System)
Hive	`hive`	Base de datos Apache Hive (https://hive.apache.org/)
Logger	`logger`	Fichero de log
Avro	`avro`	Formato de serialización de datos Apache Avro (https://avro.apache.org/)
Apache Thrift	`thrift`	Lenguaje de definición de interfaces (https://thrift.apache.org/)
Kafka	`org.apache.flume.sink.kafka.KafkaSink`	Cola de mensajes Apache Kakfa (https://kafka.apache.org/)
Elasticsearch	`org.apache.flume.sink.elasticsearch.ElasticSearchSink`	Base de datos NoSQL (https://www.elastic.co/elasticsearch/)
Fichero	`file_roll`	Fichero en el que se van escribiendo los datos de salida
HTTP	`http`	Endpoint HTTP que espera datos a través de POST
Null	`null`	Se descartan los datos (`/dev/null`)

https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#flume-sinks

3. Apache Flume - Componentes

- Algunos de los canales (**channels**) posibles en Flume son:

Canal	Tipo	Descripción
Memoria	`memory`	Cola en memoria
Apache Derby	`jdbc`	Base de datos Java embebida Apache Derby (https://dbdb.io/db/derby)
Kafka	`org.apache.flume.source.kafka.KafkaChannel`	Clúster Kafka (https://kafka.apache.org/)
Fichero	`file`	Sistema de ficheros
Memoria y sistema de ficheros	`SPILLABLEMEMORY`	Memoria (principal) y sistema de ficheros en caso de desbordamiento (overflow)

https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#flume-channels

3. Apache Flume - PySpark

- En PySpark, se crearán DStreams procedente de Apache Flume usando la clase de utilidad **`FlumeUtils`**, con los siguiente parámetros:
  - Contexto de streaming
  - Nombre de máquina (o dirección IP) donde se ejecuta Apache Flume
  - Puerto por donde enviará datos Apache Flume

```
from pyspark.streaming.flume import FlumeUtils

stream = FlumeUtils.createStream(streamingContext, hostname, port)
```

- Para la última versión de Spark, habrá que usar la dependencia `org.apache.spark:spark-streaming-flume_2.11:2.4.7`

https://spark.apache.org/docs/latest/api/python/pyspark.streaming.html

3. Apache Flume - PySpark

Este ejemplo recibe un flujo de texto en formato Avro procedente de Apache Flume y lo muestra por pantalla

```
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.flume import FlumeUtils

# Local SparkContext and StreamingContext (batch interval of 1 second)
sc = SparkContext(master="local[*]",
                  appName="Flume-DStream-StdOut",
                  conf=SparkConf()
                  .set("spark.jars.packages", "org.apache.spark:spark-streaming-flume_2.11:2.4.7"))
sc.setLogLevel("ERROR")
ssc = StreamingContext(sc, 1)

# 1. Input data: create a DStream from Apache Flume
stream = FlumeUtils.createStream(ssc, "localhost", 4444)

# 2. Data processing: get first element
lines = stream.map(lambda x: x[1])

# 3. Output data: show result in the console
lines.pprint()

ssc.start()
ssc.awaitTermination()
```

Apache Avro utiliza JSON para el formato de los datos, que después se serializan (se envían a través de la red) usando un formato binario más compacto

3. Apache Flume - PySpark

En primer lugar, ejecutamos nuestro programa Python, esperando recibir datos por el puerto 4444 de la máquina local

```
$ python flume-dstream-stdout.py
```

1

```
-------------------------------------------
Time: 2020-03-26 19:21:02
-------------------------------------------

-------------------------------------------
Time: 2020-03-26 19:21:03
-------------------------------------------
0
1
2
3
4
5
6
7
8
9
...
```

Cuando ambos procesos establezcan un conexión por el puerto 4444, se empezarán a mostrar trazas por la salida

UNIX-like

```
$ bin/flume-ng agent -n a1 -conf-file conf/flume-conf.properties
```

Windows

```
$ bin\flume-ng agent -n a1 -conf-file conf\flume-conf.properties
```

2

En segundo lugar, ejecutamos Flume (descargado de https://flume.apache.org/download.html) usando la siguiente configuración

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

a1.sources = r1
a1.sources.r1.type = seq
a1.sources.r1.channels = c1

a1.sinks = k1
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = localhost
a1.sinks.k1.port = 4444
```

3. Apache Flume - PySpark

UNIX-like

```
$ bin/flume-ng agent -n a1 -conf-file conf/flume-conf2.properties
```

2

Windows

```
$ bin\flume-ng agent -n a1 -conf-file conf\flume-conf2.properties
```

```
$ python flume-dstream-stdout.py
```

1

```
-------------------------------------------
Time: 2021-04-15 11:54:02
-------------------------------------------
hello world

-------------------------------------------
Time: 2021-04-15 11:54:03
-------------------------------------------

-------------------------------------------
Time: 2021-04-15 11:54:04
-------------------------------------------
hi hi hi hi
```

Ahora modificamos la configuración de Flume para “escuchar” la entrada de datos por una conexión TCP (socket)

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

a1.sources = r1
a1.sources.r1.channels = c1
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 9999

a1.sinks = k1
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = localhost
a1.sinks.k1.port = 4444
```

3

```
$ nc localhost 9999
hello world
OK
hi hi hi hi
OK
```

3. Apache Flume - PySpark

UNIX-like

```
$ bin/flume-ng agent -n a1 -conf-file conf/flume-conf3.properties
```

2

Windows

```
$ bin\flume-ng agent -n a1 -conf-file conf\flume-conf3.properties
```

```
$ python flume-dstream-stdout.py
```

1

```
-------------------------------------------
Time: 2021-04-13 19:23:38
-------------------------------------------
top - 19:23:37 up  6:15,  1 user,  load average: 0,62, 0,34, 0,36
Tasks: 196 total,   1 running, 195 sleeping,   0 stopped,   0 zombie
%Cpu(s): 65,6 us,  6,2 sy,  0,0 ni, 28,1 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :   3936,1 total,    265,7 free,   2235,8 used,   1434,7 buff/cache
MiB Swap:    923,3 total,    914,2 free,      9,0 used.   1399,5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  39647 user      20   0 2397364  77352  28836 S 112,5   1,9   0:01.30 java
   1042 root      20   0 1016844 150020  55996 S   6,2   3,7   4:50.93 Xorg
  39356 user      20   0 3622504 413908  35472 S   6,2  10,3   0:10.31 java
...
```

Ahora modificamos la configuración de Flume para “escuchar” la salida del comando `top -b`

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

a1.sources = r1
a1.sources.r1.channels = c1
a1.sources.r1.type = exec
a1.sources.r1.command = top -b

a1.sinks = k1
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = localhost
a1.sinks.k1.port = 4444
```

Contenidos

1. Introducción
2. Amazon Kinesis
3. Apache Flume
4. Apache Kafka
   - Pub/Sub
   - Arquitectura
   - PySpark
   - Python
5. Resumen

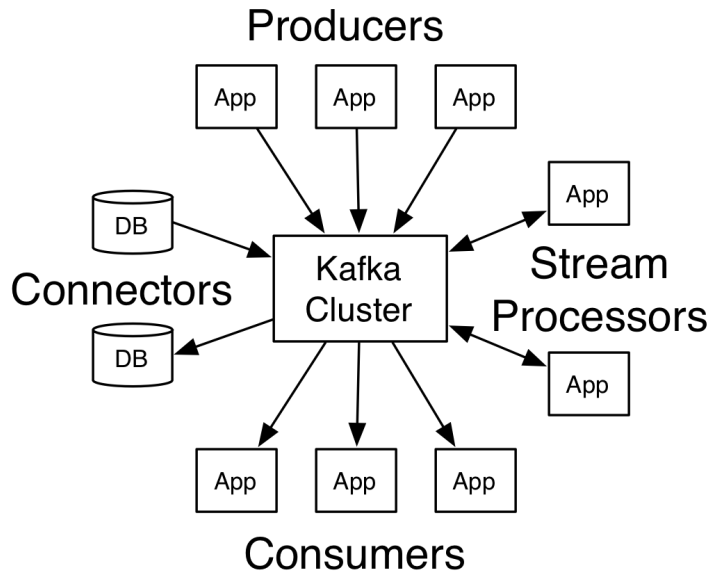
4. Apache Kafka

- **Apache Kafka** esta una plataforma de streaming distribuida, de alto rendimiento, escalable, y con tolerancia de fallos. Tiene 3 capacidades principales:
  - Sistema de mensajería (message broker) distribuido basado en el modelo publicación-subscripción (publish-subscribe, pub/sub)
  - Almacenamiento de datos
  - Procesado en tiempo real
- Kafka fue creado por LinkedIn en 2010 y posteriormente fue donado como proyecto open-source a la Apache Software Foundation
- Kafka está desarrollado en Java y Scala

https://kafka.apache.org/

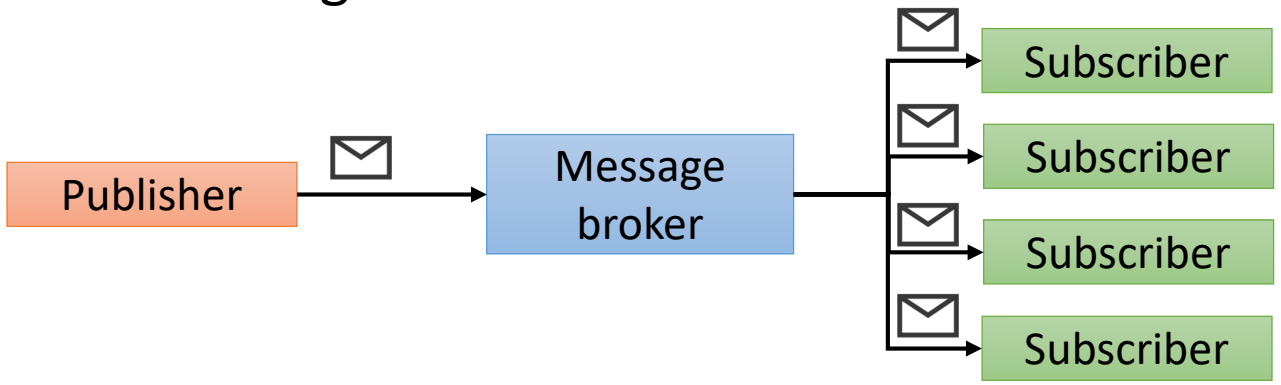
4. Apache Kafka

- Kafka se ejecuta sobre un **clúster** (conjunto de máquinas)
  - Aunque para pruebas, podemos ejecutar Kafka en una sola máquina
- Se ofrecen 4 tipos de APIs sobre un clúster Kafka:
  - Producer API: Usado por aplicaciones que generan datos, llamadas productores (*producers*)
  - Consumer API: Usado por aplicaciones que reciben datos , llamadas consumidores (*consumers*)
  - Streams API: Usado por aplicaciones que procesan datos, llamados procesadores (*stream processors*)
  - Connector API: Usado para integración con otros componentes, llamados conectores (*connectors*)



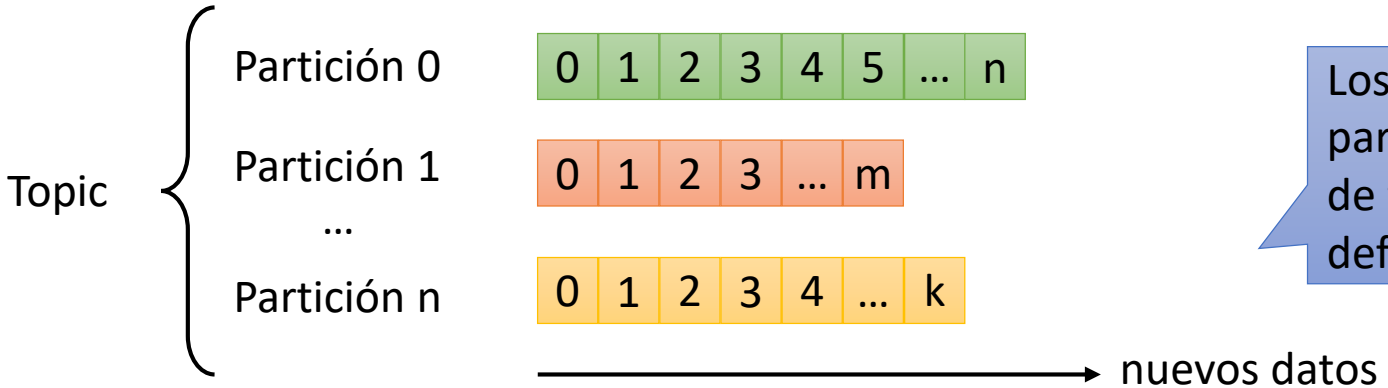
4. Apache Kafka - Pub/Sub

- El modelo publicación-subscripción (publish-subscribe, pub/sub) es un patrón de diseño que permite a una aplicación (publisher) anunciar mensajes (o eventos) a otras aplicaciones (subscribers) de forma **asíncrona** (para evitar acoplamiento en la comunicación)
  - Consiste en introducir un componente intermedio (**message broker**) entre publisher y subscriber(s) que gestione esta comunicación asíncrona
  - Cada subscriber pedirá ser avisado por el message broker cuando se reciba un nuevo mensaje/evento (*eventListener*)
  - La recepción de nuevo mensaje/evento del publisher en el en bróker activará los eventListeners registrados



4. Apache Kafka - Arquitectura

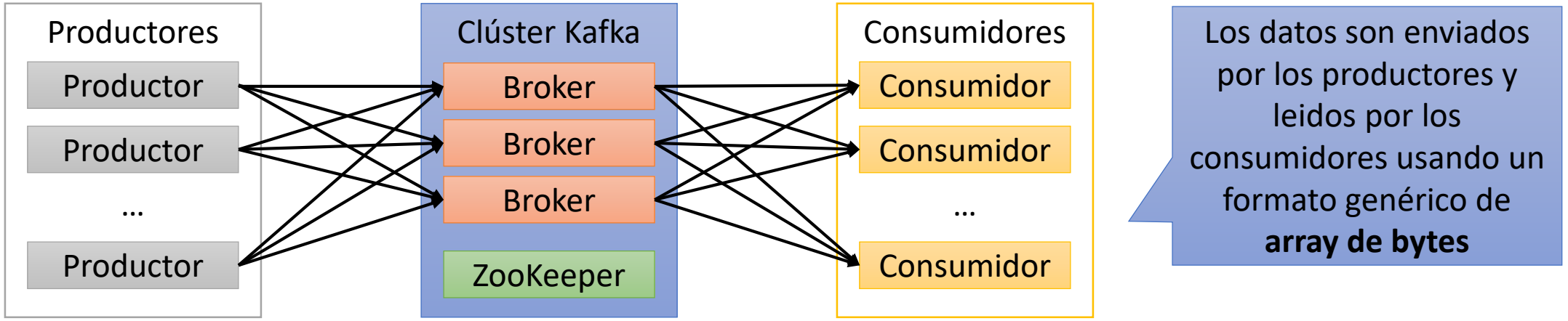
- Un clúster Kafka almacena los datos usando una abstracción llamada **topic**
  - Un topic es el nombre de una categoría en el que los datos son publicados y consumidos
- Cada topic se divide en **particiones** (para lograr escalabilidad horizontal)
  - Cada partición contiene una parte de los datos del topic en forma de colección inmutable de elementos llamados **offset** (identificados de forma única)
  - En cada offset hay registro que consiste en una tupla de 3 campos: clave, valor, y timestamp
  - Los productores escriben datos en esos offset, que puede ser leído por diferentes consumidores
  - Una partición puede ser **replicada** (para conseguir tolerancia a fallos), de esta forma hablamos de particiones líder (con datos originales) y replicas (copias)



Los datos son persistentes en las particiones durante un periodo de tiempo configurable, por defecto 168 horas (1 semana)

4. Apache Kafka - Arquitectura

- Un clúster Kafka tendrá servidores (también llamados **brokers**) que hacen de intermediarios entre productores y consumidores
  - Cada broker almacena un número de topics
- Para la coordinación entre brokers, existirá un servidor especial que ejecuta una instancia de **ZooKeeper** (https://zookeeper.apache.org/)
  - ZooKeeper es un sistema centralizado de configuración y sincronización para servicios distribuidos



4. Apache Kafka - PySpark

- PySpark ofrece soporte experimental que permite crear objetos DStreams con datos procedente de Apache Kafka usando la clase de utilidad **`KafkaUtils`**, al menos con los siguiente parámetros:
  - Contexto de streaming
  - Nombre de máquina (o dirección IP) y puerto donde escucha ZooKeeper (por ejemplo `localhost:2181`)
  - Cadena que identifica de consumidor (se pueden usar diferentes identificadores para usar varios consumidores en paralelo del mismo topic)
  - Topic y particiones de Kafka a consumir, en formato diccionario, por ejemplo: `{"test-topic": 1}`

```
from pyspark.streaming.kafka import KafkaUtils

stream = KafkaUtils.createStream(streamingContext, zookeeper, consumer-id, topic)
```

- Para la última versión de Spark, habrá que usar la dependencia `org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.7`

https://spark.apache.org/docs/latest/api/python/pyspark.streaming.html

4. Apache Kafka - PySpark

Este ejemplo recibe un flujo de texto procedente de Apache Kafka y cuenta la ocurrencia de las palabras

Por defecto, el método `createStream` usa funciones para decodificar los array de bytes de clave y valor procedente de Kafka a cadenas de texto UTF-8

```
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

# Local SparkContext and StreamingContext (batch interval of 5 seconds)
sc = SparkContext(master="local[*]",
                  appName="Kafka-DStream-StdOut",
                  conf=SparkConf()
                  .set("spark.jars.packages", "org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.7"))
ssc = StreamingContext(sc, 5)

# 1. Input data: create a DStream from Apache Kafka
stream = KafkaUtils.createStream(
    ssc, "localhost:2181", "spark-streaming-consumer", {"test-topic": 1})

# 2. Data processing: word count
count = (stream.map(lambda x: x[1])
         .flatMap(lambda line: line.split(" "))
         .map(lambda word: (word, 1))
         .reduceByKey(lambda x, y: x + y))

# 3. Output data: show result in the console
count.pprint()

ssc.start()
ssc.awaitTermination()
```

4. Apache Kafka - PySpark

- Pasos para ejecutar el ejemplo anterior:

1. Ir al directorio donde tenemos la distribución de Apache Kafka:
   - Descargado de https://kafka.apache.org/downloads

```
$ cd kafka_2.11-2.4.1
```

2. Arrancar Apache ZooKeeper (en el puerto 2181 por defecto):

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```
UNIX-like

```
$ bin\windows\zookeeper-server-start.bat config\zookeeper.properties
```
Windows

3. En otra consola, arrancamos un broker de Kafka (en el puerto 9092 por defecto):

```
$ cd kafka_2.11-2.4.1
```

```
$ bin/kafka-server-start.sh config/server.properties
```
UNIX-like

```
$ bin\windows\kafka-server-start.bat config\server.properties
```
Windows

4. Apache Kafka - PySpark

4. En una tercera consola, creamos el topic (si no existe previamente), en el ejemplo se llama `test-topic`:

```
$ cd kafka_2.11-2.4.1
```

UNIX-like

```
$ bin/kafka-topics.sh -create -zookeeper localhost:2181 -replication-factor 1 -partitions 1 -topic test-topic
```

Windows

```
$ bin\windows\kafka-topics.bat -create -zookeeper localhost:2181 -replication-factor 1 -partitions 1 -topic test-topic
```

5. Podemos usar esa misma consola para añadir datos al topic:

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test-topic
```

UNIX-like

```
$ bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test-topic
```

Windows

6. Por último, en otra consola, ejecutamos el ejemplo:

```
$ python kafka-dstream-stdout.py
```

4. Apache Kafka - PySpark

- También es posible consumir datos de Kafka con Spark Streaming a través de la API de DataFrames (datos estructurados)
- Para ello, en primer lugar hay que crear una sesión Spark especificando el uso de la dependencia `org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.7`

```
spark = (SparkSession
         .builder
         .master("local[*]")
         .appName("Kafka-DataFrame-StdOut")
         .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.7")
         .getOrCreate())
```

https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html

4. Apache Kafka - PySpark

- Usando la sesión Spark previamente creada, podemos crear objetos de tipo DataFrame en streaming para leer datos de Kafka especificando los siguiente parámetros:
  - Formato: `kafka`
  - Opción `kafka.bootstrap.servers`: lista de brokers Kakfa (separados por comas) en formato host:puerto
  - Opción `subscribe`: nombre del topic

```
df = (spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("subscribe", "test-topic")
      .load())
```

El DataFrame recibido de Kakfa tendrá las siguientes columnas: `key`, `value`, `topic`, `partition`, `offset`, `timestamp` y `timestampType`

4. Apache Kafka - PySpark

Este ejemplo recibe un flujo estructurado procedente de Apache Kafka y lo muestra por pantalla

Podemos cambiar el formato el array de bytes de clave-valor procedente de Kafka a otro tipo (casting)

```
from pyspark.sql import SparkSession

# Local SparkSession
spark = (SparkSession
         .builder
         .master("local[*]")
         .appName("Kafka-DataFrame-StdOut")
         .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.7")
         .getOrCreate())

# 1. Input data: DataFrame from Apache Kafka
df = (spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("subscribe", "test-topic")
      .load())
df.printSchema()

# 2. Data processing: read value
values = df.selectExpr("CAST(value AS STRING)", "timestamp")

# 3. Output data: show result in the console
query = (values
         .writeStream
         .outputMode("append")
         .format("console")
         .start())

query.awaitTermination()
```

4. Apache Kafka - PySpark

- Para ejecutar el ejemplo `Kafka-DataFrame-StdOut`, se siguen los mismos pasos explicados antes
- La salida de los pasos 5 y 6 sería como sigue:

```
$ bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test-topic
>Hello world
```

```
$ python kafka-dataframe-stdout.py
...
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

-------------------------------------------
Batch: 0
-------------------------------------------
+-----------+--------------------+
|      value|           timestamp|
+-----------+--------------------+
|Hello world|2020-04-30 17:04:...|
+-----------+--------------------+
```

4. Apache Kafka - Python

- Kafka proporciona una API Python para crear productores, consumidores, procesadores, y conectores
- Para poder usarla, en primer lugar hay que instalar la dependencia

```
$ pip install kafka-python
```

- Para crear un **productor** de datos, necesitamos crear un objeto de tipo `KafkaProducer` especificando uno (o varios) brokers Kafka
- Se pueden mandar mensajes de forma asíncrona usando el método `send` especificando el nombre del topic y clave y valor en array de bytes

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=["localhost:9092"])
producer.send("my-topic", value=b"bar", key=b"foo")
```

https://kafka-python.readthedocs.io/en/master/index.html

4. Apache Kafka - Python


```python
from kafka import KafkaProducer
from random import randrange
import time

producer = KafkaProducer(bootstrap_servers=["localhost:9092"])
startTime = time.time()
waitSeconds = 1.0

while True:
    randomInt = randrange(100)
    print("Sending random number to Kafka", randomInt)
    producer.send("test-topic", str(randomInt).encode())

    # Wait a number of second until next message
    time.sleep(waitSeconds - ((time.time() - startTime) % waitSeconds))
```

Este programa Python crea un productor de datos que genera enteros aleatorios entre 0 y 100 y los manda a un bróker Kafka en local cada segundo usando como nombre de topic `test-topic`

Se reciben los datos en un programa Spark que procesa los datos realizando un filtrado a aquellos enteros mayores de 50, y los muestra por pantalla

```python
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

# Local SparkContext and StreamingContext
sc = SparkContext(master="local[*]",
                  appName="Kafka-DStream_RandomInt-StdOut",
                  conf=SparkConf()
                  .set("spark.jars.packages", "org.apache.spark:spark-
streaming-kafka-0-8_2.11:2.4.7"))
sc.setLogLevel("ERROR")
ssc = StreamingContext(sc, 1)

# 1. Input data: create a DStream from Apache Kafka
stream = KafkaUtils.createStream(
    ssc, "localhost:2181", "spark-streaming-consumer", {"test-topic": 1})

# 2. Data processing: filter numbers > 50
higher50 = (stream.map(lambda x: x[1])
            .filter(lambda x: int(x) > 50))

# 3. Output data: show result in the console
higher50.pprint()

ssc.start()
ssc.awaitTermination()
```

4. Apache Kafka - Python


```python
from kafka import KafkaProducer
from random import randrange
import time
import json

producer = KafkaProducer(bootstrap_servers=["localhost:9092"])
startTime = time.time()
waitSeconds = 1.0

while True:
    randomInt1 = randrange(100)
    randomInt2 = randrange(100)
    msg = [{"randomInt": randomInt1}, {"randomInt": randomInt2}]
    print("Sending JSON to Kafka", msg)
    producer.send("test-topic", json.dumps(msg).encode())

    # Wait a number of second until next message
    time.sleep(waitSeconds - ((time.time() - startTime) % waitSeconds))
```

Este programa Python crea un productor de datos que genera datos en formato JSON y los manda a un bróker Kafka en local cada segundo usando como nombre de topic `test-topic`

Se reciben los datos en un programa Spark que “parsea” estos mensajes JSON y suma los valores enteros recibidos

```python
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
import json

def load_json(msg):
    try:
        return json.loads(msg)
    except Exception:
        print("Exception parsing JSON", msg)
        return {}

def sum_values(list):
    sum = 0
    for i in list:
        sum += i.get("randomInt")
    return sum

…

# 2. Data processing: sum receiver integer values
out = (stream
       .map(lambda x: load_json(x[1]))  # parse JSON of Kafka stream value
       .filter(lambda x: len(x) > 0)  # filter out non-json messages
       .map(lambda j: sum_values(j))  # sum each randomInt received
       )

# 3. Output data: show result in the console
out.pprint()

ssc.start()
ssc.awaitTermination()
```

Contenidos

5. Resumen

- Las fuentes avanzadas de datos en Spark Streaming son 3: **Amazon Kinesis**, **Apache Flume**, y **Apache Kafka**
- El uso de estas fuentes está disponibles en Spark a través de **dependencias externas** que habrá que especificar en nuestro programa
- Apache Flume permite la recolección de logs mediante agentes distribuidos usando un esquema **source→channel→sink**
- **Apache Kafka** es una plataforma de streaming que implementa un sistema de mensajería distribuido basado en el modelo **pub/sub**
- Kafka proporciona diferentes APIs para crear **productores**, consumidores, procesadores, y conectores
- Podemos usar **PySpark** en Python para leer datos en streaming procedentes de Kafka tanto con la API de **DStream** como **DataFrame**