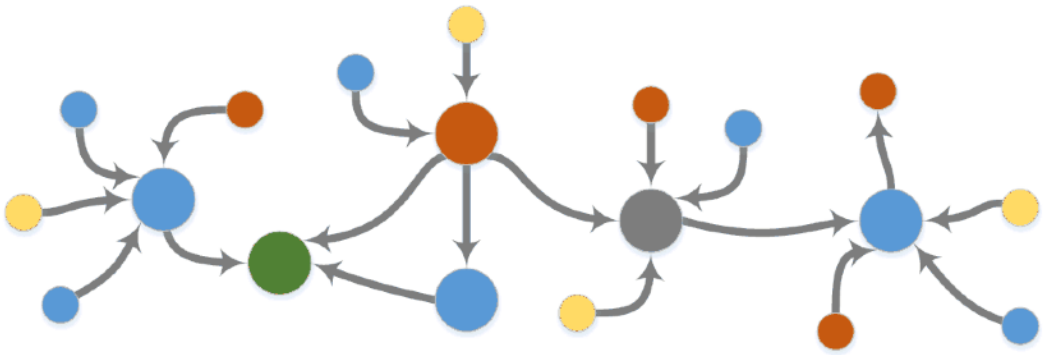
Ingegneria e Scienze Informatiche
Università di Bologna, Cesena Campus

Introducing Kafka,
a distributed streaming platform



final project for the course of **Distributed Systems**
held by **Prof. Andrea Omicini** during
the academic year 2016/17

presented by Giacomo Bartoli

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.

taken from the Reactive Manifesto (https://www.reactivemanifesto.org)

1. Introduction

People, companies, daily activities can be considered today as data producers. As someone goes to the grocery or even grab his smartphone for checking the weather, data are produced and probably stored. Today, everything is a data source: online activities, financial trading, IoT sensors generate data at an ever-increasing rate. This clearly rises an issue: how to store and handle these data? Moreover, what if these data does not come in batch (batch processed) but they are a continuous flow of micro data? This is the idea of stream of data. Hence, we need architectures for the ingestion phase. These architectures must be flexible, scalable, fast and resilient, proving support for multiple reading/writing of data while hiding failures to the users.
The amount of generated data and the need of handling both structured and unstructured data is pushing to the limit traditional infrastructures.
In this context, stream processing plays an important role. Just think about all the real time systems, where data should be delivered, analyzed and process at almost the same time they are produced. The latter case can be considered as pervasive. In fact, it can be applied to the industry (we are currently moving towards what is known as Industry 4.0), but also to social networks and analytics.
The following examples show the ubiquitousness of such real time systems:

- An industrial power plant is controlled by different sensors and the alarm system must be triggered when the total sum of data produced by sensors is greater than a certain threshold.
- Multimedia systems (ex: Youtube, Netflix)
- Real-time analytics for web marketing, meaning tracking the visitor's behaviour on a website and based on its clicks let specific ads appear.
- Gaming (actions taken by the player must trigger other CPU's actions)
- Self-driving cars: the vehicle must continuously detects what is around.

All these examples point out that is not sufficient to process streams of data. It is necessary to store them, ingest them and even combining them.
Apache Kafka is the right framework for handling streaming of data facing all the typical issues of a distributed system.

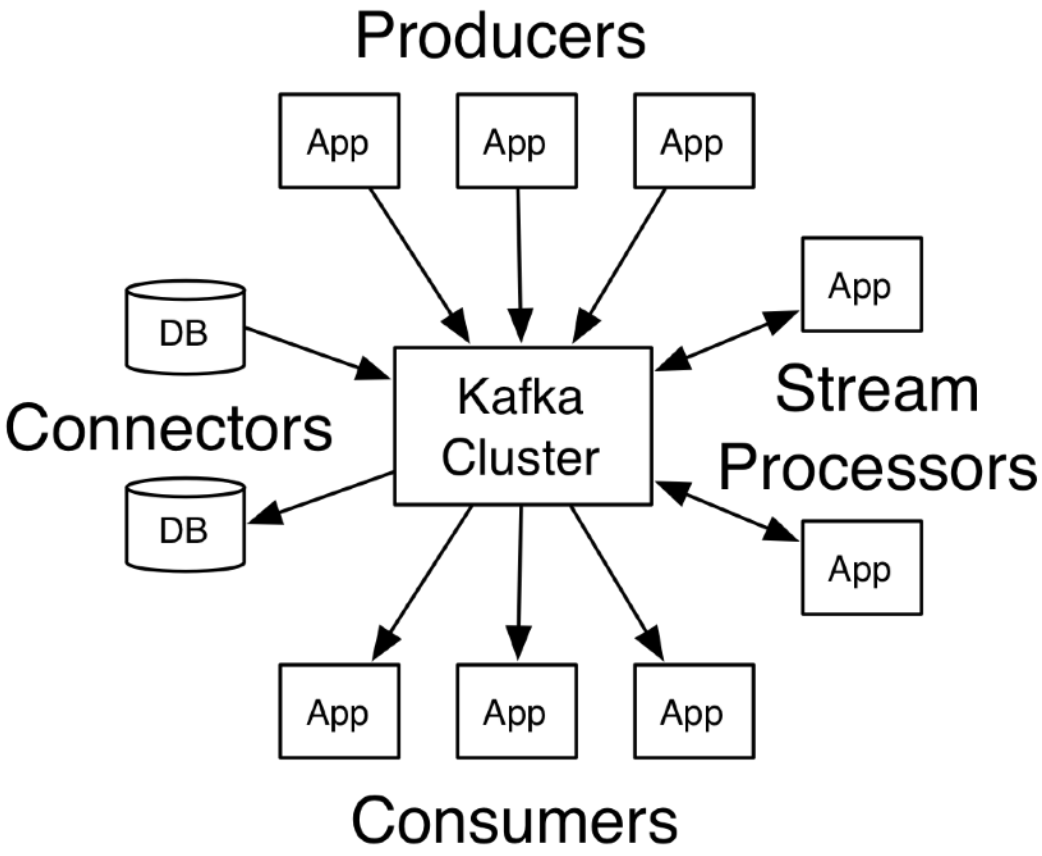
2. Overview

A distributed streaming platform is made essentially by three key capabilities:
- It is possible to publish and subscribe to streams of records.
- It is possible to store streams of records in a fault-tolerant way.
- It is possible to process streams of records.

These three capabilities make it clear that this is the right pathway for building real-time applications. Especially, we will be able to design two broad classes:

- Design real-time streaming data pipelines that ingest and produce streams of data in a reliable way.
- Design real-time streaming applications that transform or react when a message is published on the channel (as we will see the channel is called ‘topic’ in Kafka).

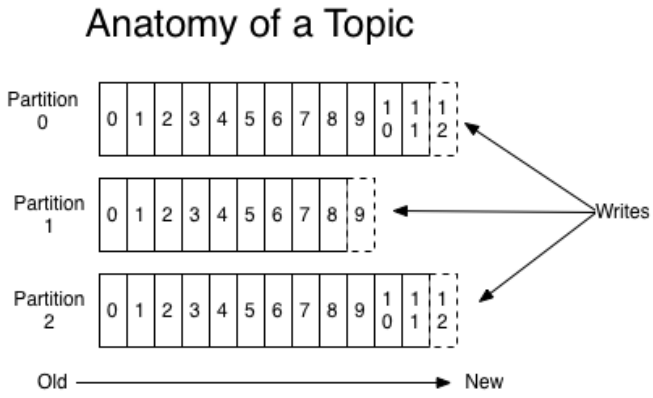
Let’s take a quick overview of how Kafka’s framework is composed:



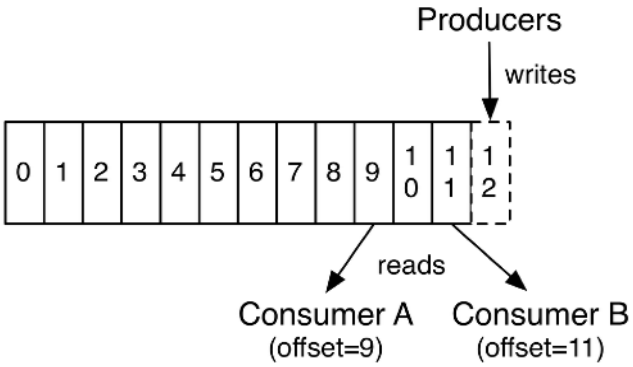
There are four core components:
- Producers: allow an application to publish a stream of records to one or more Kafka topics. Please notice that Kafka topics are the equivalent of a channel in a publish/subscribe architecture.
- Connectors: allow building and running reusable producers or consumers that connect Kafka topics to existing applications. For instance, a connector to a relational database might capture every change to a table.
- Consumers: allow an application to subscribe to one or more topics and process the stream of records produced.
- Stream processors: allow an application to act as a stream processor. They consume the input stream from n topics and produce an output stream to other output topics.

3. Topics

A topic is where records are published. Topics in Kafka are always multi-subscriber. Thus, a topic can have from zero to n consumers that subscribe to the data written to it. The main idea behind is very close to the publish/subscribe channel used in the ESB (Enterprise Service Bus), where the architecture promotes agility and flexibility among applications.



A partitioned log is maintained for each topic into the Kafka cluster (recap: see previous schema). A partition is an ordered, immutable sequence of records that is continuously appended. Each record is assigned to a sequential id called 'offset'. In this way it's easy to identify each record within the partition.
Thanks to a configurable retention period, the cluster is able to retain all published records despite the fact that they have been consumed or not. For example, a record is available for consumption two days after its publication if the retention period is set to two days. After that it will be discarded to free up space. Does the retention period affect Kafka's performances? Depending on the data size performances are constant, so storing data for a long period is not such a big deal.



The offset is handled by the consumer. This means that the consumer can consume records in any order. For instance, a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from it.

Why does Kafka use partitions? There are several purposes.
First of all, the log is able to scale beyond a size that will fit on a single server. A topic is made by different partitions so that it can manage an arbitrary amount of data. Plus,

partitioning enables parallelism. Consumers can read data at the same time, even from the same partition but with different offsets.
Concerning fault tolerance and replication, partitions are replicated through a number of server. That number is, of course, a parameter that can be set.

4. Leader and followers

In a bigger scenario, we can consider a Kafka Broker as a single node in the cluster. The Broker is responsible for maintaining high availability and consistency in the cluster. In order to do so, Kafka totally relies on the concept of replication. In fact, partitions are the way Kafka provides redundancy and scalability. Achieving high availability in Kafka is done through redundant copies of each partition across the cluster. Each copy is called replica. A replica can be:

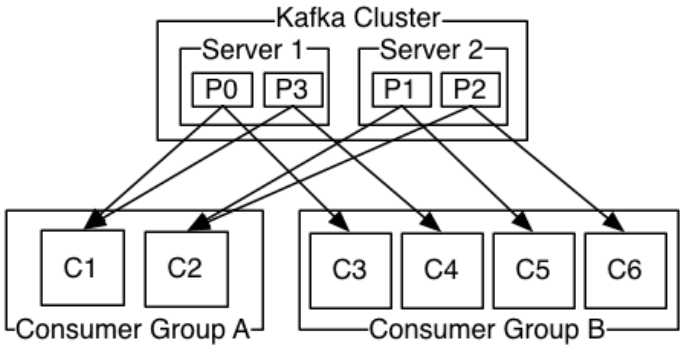
- leader: is responsible for reading and writing data in a particular partition. Each partition has only one leader. All read and write requests for a particular partition will always be served by a specific replica and those replicas are called Leaders.
- follower: as the name suggests, follower just keep himself updated with the leader by fetching messages so that in case of leader's failure any follower can act as a leader.

Leader is also responsible for make sure followers are all up to date. To do this, each follower sends a fetch request that contains the current offset. The rule is that followers cannot request random offsets like consumers, but they they always must request the next offset. Ex: if the follower has currently offset 5 then it can only ask for offset 6. Any follower which is not requesting a new message from the leader within 10 seconds or is not able to catch up the latest message is considered out of sync. Replicas that are in sync are called In Sync Replicas (ISR).

5. Producers and Consumers

Producers are responsible for choosing which record to assign to which partition inside the topic. They can both adopt a round-robin algorithm or apply to more advance behaviours depending on the key in the record. For sure, a round-robin approach is guaranteed to balance load.
On the other hand, consumers are clustered into consumer groups. Each record published to a topic is delivered to one consumer instance, which contains all subscribing consumer groups. Consumer instances can be even on separate machines.

Ex:



- 4 partitions, 1 topic
- 2 servers
- 2 consumer groups, group A is made of 2 instances while group B has 4.

Why does each group have many instances? It's about scalability and fault tolerance.
This architecture is just like a traditional publish/subscribe semantic. The only difference is that the subscriber here is a cluster of consumers instead of a single process.
The way consumption is implemented in Kafka is dividing up the partitions over the consumer instances such that each one is the exclusive consumer of a fair share of partitions. If new instances join the group they will take over some partitions from other members of the group. If an instance dies, its partitions will be distributed to the remaining instances. Kafka provides a total order over records within a partition, not between different partitions in a topic.
What if an application requires a total order over records? It can be achieved with a topic with only one partition. However, in that case only one consumer can process data per consumer group.

Kafka gives the following guarantees:
- Messages sent by a producer to a particular topic partition will be appended in the same order as they are sent.
- A consumer instance sees records in the order they are store in the log.
- For a topic with replication factor n, there can be at most n-1 failures without losing any records committed to the log.

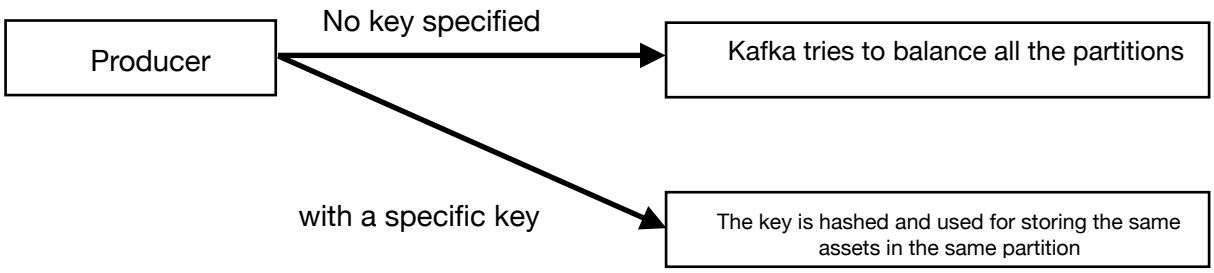
6. Kafka vs traditional message passing systems

Traditional enterprise systems have two models: queuing and publish/subscribe. In a queue, a pool of consumers may read from a server and each record goes to one of them. In a publish-subscribe pattern the record is broadcasted to all consumers. So, queuing allows you to scale your processing because data processing is split over multiple consumer instances. On the other hand, queues are not multi subscribers, once one process reads the data then it's gone. Publish subscribe can broadcast data to different processes, but there is no way of scaling processing since every message goes to every subscriber. Moreover, a queue hands out records in order they have been stored but in case of multiple consumers records are delivered asynchronously, hence there is no guaranteed the final delivery can be different. Ordering of records is lost in the presence of parallel consumption. A workaround is mutual exclusion: one process consumes from a queue. For sure, this cancels parallelism while processing data.
What makes Kafka different is the fact that it combines both systems. Thanks to the partitioning, which enables parallel consumption, Kafka is able to assure both ordering and load balancing over a pool of consumer processes. This is realized by assigning the partitions in the topic to the consumers in the consumer group such that each partition is consumed by exactly one consumer in the group. The consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. However, there is one constrain: the number of

consumer instances in a consumer group must be less or equal to the number of partitions.

6A. Load balancing

Given a topic made by N partitions, Kafka tends to distribute all records produced by producers equally amongst all the partitions. The only way to change this is to overwrite the default behaviour. So, for a specific topic the workload can be parallelized just by using partitions. Make sure to set the right number of partitions because this isn't held automatically by Kafka. The good news is that there is no need to install, configure any other plugin for assuring load balance. Here, the developer has the chance to decide in which topic records will be stored.
What I found interesting is that fact it is possible not to specify in which partition on the topic records will written, and in this case Kafka will try to assure load balancing automatically, or even exploit the interface for semantic partitioning by allowing the user to decided a key to partition. The key is used by an hash function before storing data. This is useful because if the key chosen is a user id then all data for a given user would be sent to the same partition:



6B. Asynchronous send

It's even possible to store temporarily a fixed number of message or to wait no longer than a certain latency bound (both kb or ms). This mechanism is configurable and it represents a trade off: a better throughput for a smaller amount of additional latency.

6C. Consumer position

Another challenging task is to keep track of what records have been consumed or not.
Traditionally this is done by keeping metadata about what messages were consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally or it may wait for acknowledgement from the consumer. This seems clever because after that the broker knows what is consumed and it can delete that record, reducing the data size. However, there might be still some issues. If the broker records a message as consumed immediately every time it is handed out over the network, what happen if the consumer fails then? The message is clearly not delivered. To solve this, many systems add an acknowledgement feature which consists of marking messages as sent when they are sent. The broker waits for a specific ACK from the consumer before

mark the message as consumed. Despite this strategy, this is not the best solution yet. It can happen that the consumer fails before sending back the ACK. Another minor problem is about performances: the broker must keep multiple states about every single message (first to lick it so it is not given out a second time and then to mark it as consumed so that it can be deleted).
Kafka handles this differently. Topic is divided into a set of totally ordered partitions, each of which is consumed by exactly one consumer within each subscribing consumer group at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.
There is a side benefit of this decision. A consumer can deliberately rewind back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

7. Data storage

Replication is enabled by default in order to assure fault-tolerance. Producers wait on acknowledgement meaning that a write isn't considered complete until it is fully replicated and guaranteed to persist even if the server written to fails. Despite this, Kafka will perform the same whether storing 50 KB or 50 TB of data. Given that is the client that controls read position, Kafka can be considered as distributed file system addressed to high-performance, low-latency, log storage, replication and propagation.
Let's dive into the replication design mechanism.
It is known that disk are typically slow, this is why is difficult to offer great performances while reading data and storing a huge volume of them. The point is that there a divergence while reading data linearly and writing data randomly. A linear read over a traditional hard disk is up to 600MB/sec. A random write, due to security reasons, is about 100k/sec. The gap is clearly meaningful. In order to compensate this divergence, modern operating systems make a massive use of the cache, because these reads and writes are the most predictable patterns. A modern OS will happily divert all free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice. Plus, we are working on the top of the JVM, which means that the memory overhead of objects is very high and the garbage collection becomes increasingly fiddly and slow as the in-heap data increases.
Kafka adopts an inverted design: instead of maintaining in-memory data and move them to the file system when there is no space anymore, Kafka writes data to a persistent log on the file system without necessarily flushing to disk. This just means that it is transferred into the kernel's pagecache.

7A. Replication

Kafka replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: slaves are inactive, throughput is heavily impacted, it requires manual configuration, etc. Kafka is meant to be used with replication by default—in fact we implement un-replicated topics as replicated topics where the replication factor is one.
The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All reads and writes go to the leader of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log—all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).
Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.
As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive". For Kafka node liveness has two conditions

1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)
2. If it is a slave it must replicate the writes happening on the leader and not fall "too far" behind

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" nodes. If a follower dies, gets stuck, or falls behind, the leader will remove it from the list of in sync replicas. The determination of stuck and lagging replicas is controlled by the replica.lag.time.max.ms configuration.
In distributed systems terminology we only attempt to handle a "fail/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). Kafka does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).
We can now more precisely define that a message is considered committed when all in sync replicas for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the acks setting that the producer uses. Note that topics have a setting for the "minimum number" of in-sync replicas that is checked when the producer requests acknowledgment that a message has been written to the full set of in-sync replicas. If a less stringent acknowledgement is requested by the producer, then the message can be committed, and consumed, even if the number of in-sync replicas is lower than the minimum (e.g. it can be as low as just the leader).
The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.
Kafka will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

After a brief overview concerning the Kafka infrastructure, main components, capabilities and functionalities it's time to move to the project. Here, we are going to follow a bottom-up approach starting from scratch and, iteratively, increase the complexity of the project.

8. Setup

Kafka relies on ZooKeper that is a centralized service for distributed systems to a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. This means that before running Kafka we have to install Zookeper.
From the command line:

```
> brew install zkserver
```

and then it's easy to run the service:

```
> zkserver start
```

After that it is sufficient to connect to the kafka website (https://kafka.apache.org), download the zip and extract it. Move the the kafka directory and then:

```
>./bin/kafka-server-start.sh /config/server.properties
```

Now services are up and running.
Given that Java and IntelliJ are already installed on your computer, just open IntelliJ and create a new maven project. Dependencies, held by Maven, are listed below:

```
    <groupId>groupId</groupId>
    <artifactId>kafka</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
```

```
        </plugins>
    </build>

    <dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka_2.10</artifactId>
        <version>0.8.2.1</version>
        <exclusions>
            <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-kafka</artifactId>
        <version>0.10.0-beta1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.storm</groupId>
        <artifactId>storm-hdfs</artifactId>
        <version>0.10.0-beta1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>1.0.0</version>
    </dependency>
    </dependencies>
</project>
```

Please pay attention to use this specific component's version. I had some issues that led to an unknown exception while running the program due to some conflicts. Only after some days I realized that I had to use these exact versions.
Let IntelliJ finish the wizard setup for the project. The initial building will probably require few minutes depending on the network speed.
In the mid time, we can create the first topic from the command line:

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

Note that the topic called 'TOPIC_NAME' has just 1 partition and the replication factor is set to 1. It is possible to change these parameters inside the server.properties file located into the kafka directory.
To check if the topic has been correctly created:

```
> ./bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Then the terminal will show the list of available topics on kafka.

8A. Engineering the distributed system

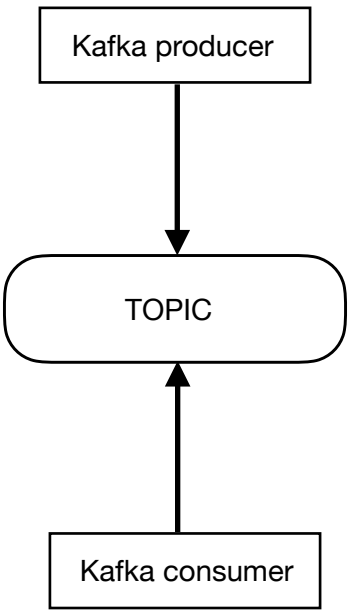
A small recap of the previous phase:

- we setup ZooKeeper, on which Kafka is based on.
- we setup the Kafka server
- we create our first topic
- we create an IntelliJ project for starting develop using Kafka

Next steps are:

- create a producer that produces a stream of data into our topic
- create a consumer that consumes data from that topic
- incrementally increase the complexity of the system by adding producers, consumers and even more topics.

Thus, the first schema that can summarise a simple producer/consumer architecture is:



Now, the point is: which kind of data do we use as stream?
It's easy to implement a function that generates random numbers continuously, or even random strings. However, in this case we are going to exploit Twitter API for create a stream of tweets. Given a specific keyword, the code will produce a stream of tweets (id, value, hashtag) that are going to be stored into the topic.
First, it is necessary to add the following dependencies to Maven (pom.xml):

```
<dependency>
        <groupId>org.twitter4j</groupId>
        <artifactId>twitter4j-core</artifactId>
```

```
        <version>3.0.3</version>
     </dependency>
    <dependency>
        <groupId>org.twitter4j</groupId>
        <artifactId>twitter4j-stream</artifactId>
        <version>3.0.3</version>
    </dependency>
    <dependency>
        <groupId>org.twitter4j</groupId>
        <artifactId>twitter4j-async</artifactId>
        <version>3.0.3</version>
     </dependency>
```

On Twitter's website, API keys are needed: consumerKey, consumerSecret, accessToken, accessTokenSecret. So, let's create a Java class with all these parameters:

```
public class Configuration {

    static String consumerKey = "YOUR_CONSUMER_KEY_HERE";
    static String consumerSecret = "YOUR_CONSUMER_SECRET_HERE";
    static String accessToken = "YOUR_ACCESS_TOKEN_HERE";
    static String accessTokenSecret = "YOUR_ACCESS_TOKEN_SECRET_HERE";
    static String topicName = "TOPIC_NAME";
    static String hashtag = "HASHTAG";
}
```

TOPIC_NAME is simply the name of the topic we created during the setup.
HASHTAG is a string used as key for retrieving tweets from Twitter. For instance, given 'love' as hashtag the program will produces a stream of data made of all the tweets related to the word 'love'.

The full code comprehensive both of the tweet's generation and producing tweets on the topic can be found at this link: https://github.com/giacomobartoli/kafka/blob/master/src/main/java/KafkaTwitterProducer.java. However, I would like to focus on some parts that I consider relevant.

First, a queue is created. Here's where all the tweets will be stored. Later on, a TwitterStreamer is created using the parameters contained into the Configuration class. Then, all tweets are filtered by keyword (hashtag) and added to the queue:

```
TwitterStream twitterStream =new TwitterStreamFactory(cb.build()).getInstance();
        StatusListener listener = new StatusListener() {
        @Override
            public void onStatus(Status status) {
                queue.offer(status);
            }

        twitterStream.addListener(listener);
        FilterQuery query = new FilterQuery().track(keyWords);
        twitterStream.filter(query);
```

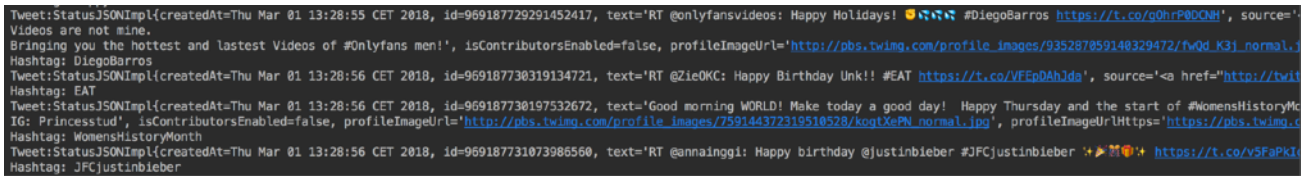
After retrieving tweets, they are added to the topic:

```
Producer<String, String> producer = new KafkaProducer<String, String>(props);
        int i = 0;
        int j = 0;

       while (true) {
            Status ret = queue.poll();

            if (ret == null) {
                Thread.sleep(100);
                // i++;
            } else {
                for (HashtagEntity hashtage : ret.getHashtagEntities()) {
                    System.out.println("Tweet:" + ret);
                    System.out.println("Hashtag: " + hashtage.getText());
                    ProducerRecord r = new ProducerRecord<String, String>(topicName, j++,
ret.getText());
                    producer.send(r);

                     }
            }
        }
```

Each tweet represents a record inside the Kafka's topic, stored as a tupla <T, P, K, V>, where:

- T is the topic's name
- P is the only partition we created during the setup
- K is the offset
- V is the real tweet, to be intended as a string made of 256 characters.

Running the code, this is the output of the console for the keyword 'happy':



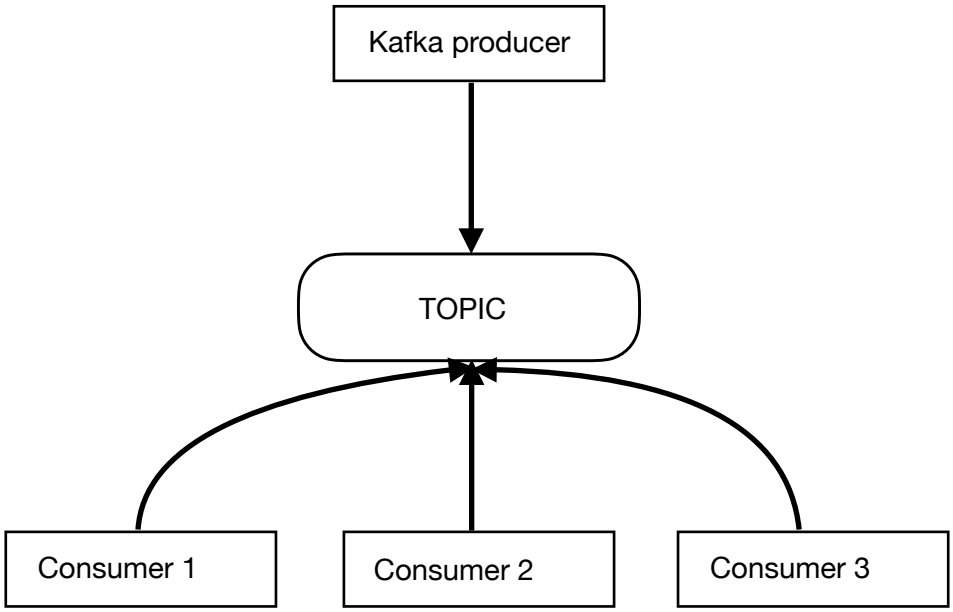
All the tweets are now stored into the topic, but there are no consumers yet.
Let's now create a consumer by adding the following class to the project:

```
public class SingleConsumer {

    public static void main(String[] args) throws Exception {

        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "tweetsConsumer");
        props.put("enable.auto.commit", "false");
        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);

        consumer.subscribe(Arrays.asList("tweets2"));
```

```
        int i = 0;

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(1000);
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("Record: offset = %d, key = %s, value = %s\n",
record.offset(), record.key(), record.value());
            }

        }
    }
}
```

Here things are pretty simple. First, some properties are necessary to setup the Kafka consumer (see props object). Then, a new KafkaConsumer is created and loop is consuming records from the topic until 1000 units. Each record is printed as <offset, key, value>.
Again, running the program we can see that it clearly works as expected.

The following outline aims to increase the complexity of the system. One producer is still producing data on the topic, but, on the other side, there are multiple consumers:



According to this schema, the producer remains the same. No changes to the code are needed.
In this scenario, each consumer is modelled through a thread. Kafka follows the task/ executors pattern for concurrency programming: each computation is a thread and they are all added to the pool of thread. In this way we're sure that threads will be executed according to the CPU availability.

This is how the single thread is designed:

```
public class ConsumerLoop implements Runnable {
    private final KafkaConsumer<String, String> consumer;
    private final List<String> topics;
    private final int id;

    public ConsumerLoop(int id, String groupId, List<String> topics) {
        this.id = id;
```

```
        this.topics = topics;
        this.consumer = new KafkaConsumer(props);
    }

    @Override
    public void run() {
        try {
            consumer.subscribe(topics);
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Long.MAX_VALUE);
                for (ConsumerRecord<String, String> record : records) {
                    Map<String, Object> data = new HashMap();
                    data.put("partition", record.partition());
                    data.put("offset", record.offset());
                    data.put("value", record.value());
                    System.out.println(this.id + ": " + data);
                }
            }
        } catch (WakeupException e) {
        } finally {
            consumer.close();
        }
    }

    public void shutdown() {
        consumer.wakeup();
    }
}
```

This is essentially the same code as the KafkaProducer class. The only difference is that this class implements a Runnable interface, because of thread definition. So, each consumer subscribe to the topic's list. In this case the list will be made by just one topic. Now, let's create a new MultipleConsumer class to handle all the consumers:

```
public class MultipleConsumer{

    private static String groupId = "tweetsConsumer";
    private static int N_CONSUMERS = 3;

    public static void main(String[] args) {
        List<String> topics = Arrays.asList("MY_TOPIC_LIST");
        final ExecutorService executor = Executors.newFixedThreadPool(N_CONSUMERS);

        final List<ConsumerLoop> consumers = new ArrayList();
        for (int i = 0; i < N_CONSUMERS; i++) {
            ConsumerLoop consumer = new ConsumerLoop(i, groupId, topics);
            consumers.add(consumer);
            executor.submit(consumer);
        }

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                for (ConsumerLoop consumer : consumers) {
                    consumer.shutdown();
                }
                executor.shutdown();
                try {
                    executor.awaitTermination(5000, TimeUnit.MILLISECONDS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

A consumer is created exploiting the class previously created. All the consumers are submitted to the ThreadPool. Running the code we got the following output:

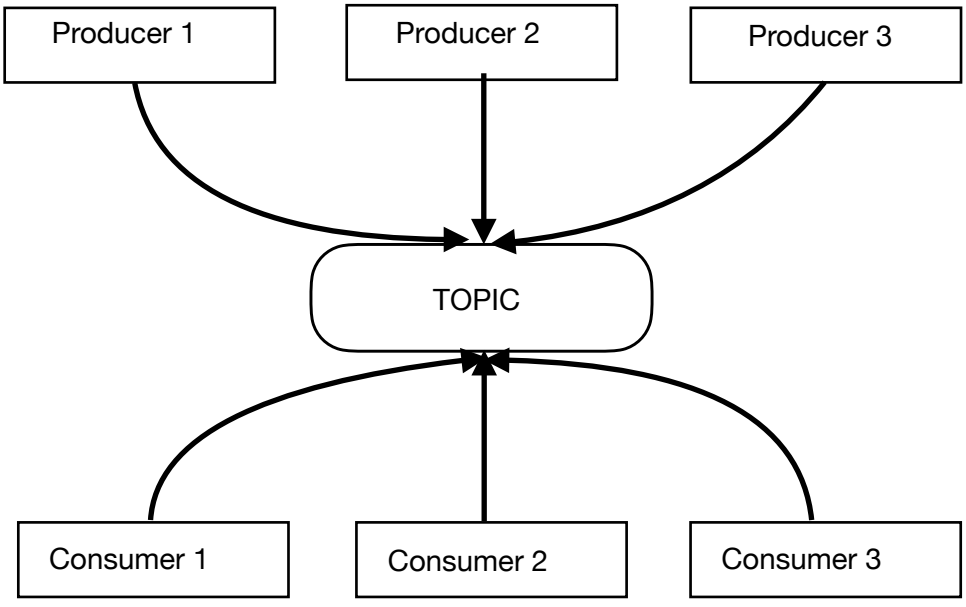
```
0: {partition=0, offset=271, value=RT @luyixing98: I have never been this happy on seeing
youngjae 🤣🤣🤣🤣🤣
#GOT7onKnowingBros  #1nOnlyUGOT7xHyolyn @GOT7Official #GOT7  #갓세븐 #Ey…}
1: {partition=0, offset=202, value=@RanveerOfficial  @deepikapadukone happy holi 😍😍
#deepveer
#HappyHoli https://t.co/ZHqlz0NHuh}
```

Why do multiple consumers are retrieving data from the same partition? As stated in the introduction, a topic is partitioned to enable parallelism and enhancing performances. That's actually kind of useless for a distributed system. So, to fix it let's open the server.properties file and set the parameter `num.partitions = 3` for example. At this point a restart is needed both for the Kafka server and the Zookeeper server. After that, re-running the program this is what is printed on the console:

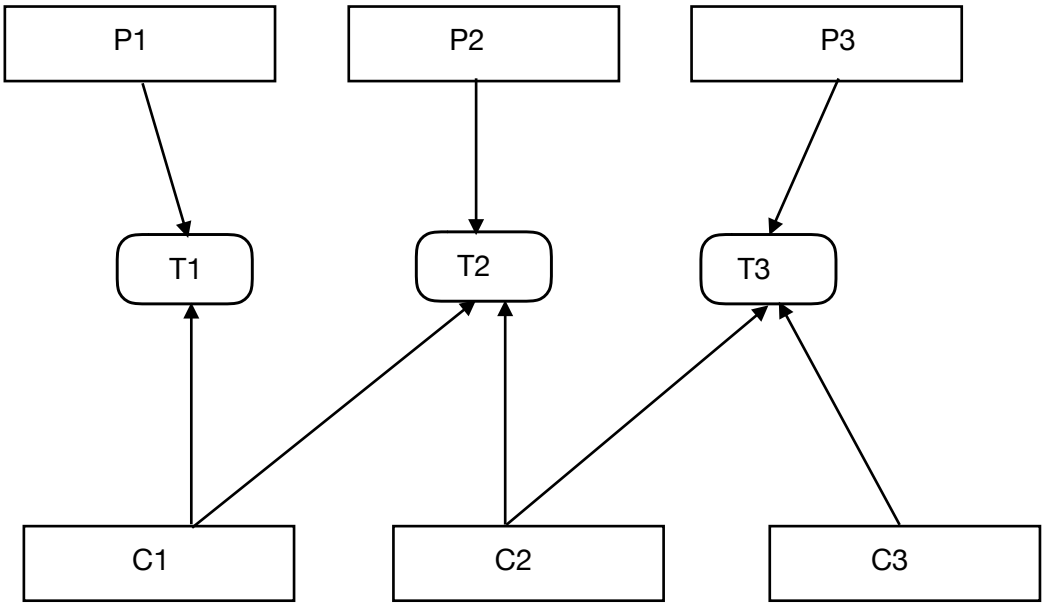
```
1: {partition=1, offset=198, value=Pls Be Good & Damn... Be Beautiful! 🌺🌷🌺 Happy New
Month! #hellomarch ☀️☀️☀️#goodmorning #mood #collage… https://t.co/auQfbviwz6}

2: {partition=2, offset=204, value=RT @___edzzzzzk: Please happy ending for CoPple and
MalTris!!! Wala sana mamatay! Huhuhu

#LaLunaLastSacrifice}
0: {partition=0, offset=266, value=RT @tobennietoben: @EXOL_BD @weareoneEXO Happy
together

Tweet, Retweet, Quote and Reply with
#EXOL #iHeartAwards #BestFanArmy @weareoneE…}
2: {partition=2, offset=205, value=RT @tobennietoben: @EXOL_BD @weareoneEXO Happy
together
```

Data are written from all the partitions. Make sure that number of consumers is greater or equal to the number of partitions. At leat one consumer for partition is needed.

In the same way it's easy to implement multiple Producers: one producer as a thread and then all the producers are taken as input for the bag of threads:



It's important to understand if this kind of architecture is really useful: what's the sense of multiple producers writing on the same topic? what happened if one them fails? Moreover, how to handle maintenance? This questions make it clear that a multiple producers architecture must be split: 1 producer for one topic and, if more producers are needed, each one publishes streams of data on a different topic. Consumers can still read from topics they subscribed.
The new architecture for multiple producer and multiple consumers must have different topics for each producers:



C1 reads data from T1 and T2.
C2 reads data from T2 and T3.
C3 reads data just from T3.

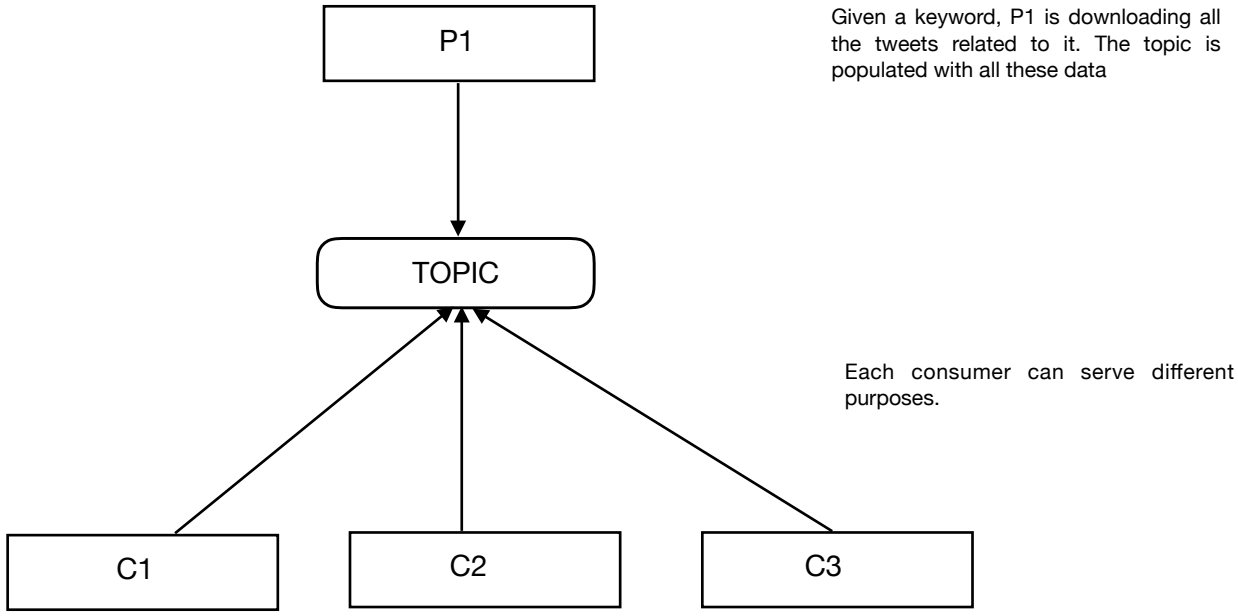
Each topic should be partitioned according to the number of consumers. Number of partitions determines the parallelism grade on Kafka. In general, the more partitions there are in a Kafka cluster, the higher the throughput one can achieve. It is possible to set the number of partitions for each topic when it's created or using the 'alter command':

```
./bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic MY_TOPIC --partitions N
```

The number of partitions can always be increased but never decrease it.
A good practice is to setup the number of partitions as:
#partitions = #consumer + N, where N a factor that depends on how the system is going to evolve through time. It's to important to set N not to be too much greater than the number of consumers because otherwise data will become available after a long time. Keep in mind that each message is replicated through all the partitions of a topic and until this process is finished consumers cannot start reading.

8B. Exploiting Kafka for real time analysis

Right now, the system is built and stable and we find out that a good practice is to assign just one producer to a topic with multiple consumers. It's a key value for software engineering: in this way it's easier to maintain and evolve the architecture. So, let's now focus on the purpose to give a concrete aim to this project. The idea is to retrieve all the tweets from Twitter related to a specific keyword and then use multiple consumers to analyse them and applying some techniques for providing some insights.
At this point, we only need to add some methods to the consumers because they can already consume all the tweets from a certain topic:



The real time factor raises one issue: it is not possible to process data when they are all downloaded because they are streams of data, they are continuously downloaded with no ending. Then, it's important to decide a range of time. Each action must be performed considering data retrieved in a specific range of time.
It's easy to write an algorithm that count the most popular hashtags in the last 60 seconds. This part is written in Scala. In fact, Kafka is built on the top of the JVM:

```
// Map each topic to a thread
val topicMap = topics.split(",").map((_, 1)).toMap

// Map value from the Kafka message (k, v) pair
val lines = KafkaUtils.createStream(ssc, "localhost:2181", group, topicMap).map(_._2)

// Finding hashtags
val hashTags = lines.flatMap(_.split(" ")).filter(_.startsWith("#"))

// Get the top hashtags over the previous 60/10 sec window
val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(60))
       .map { case (topic, count) => (count, topic) }
       .transform(_.sortByKey(false))
```

```
// Print popular hashtags

topCounts60.foreachRDD(rdd => {
       val topList = rdd.take(10)
       println("\nPopular topics in last 60 seconds (%s total):".format(rdd.count()))
       topList.foreach { case (count, tag) => println("%s (%s tweets)".format(tag,
       count))}
})
```

Let's see how the system provides some insights, specifically we want to know the most popular hashtags on Twitter given a keyword, that can even be an hashtag as well.

Keyword: #Elezioni4Marzo2018
Results:
#Elezioni4Marzo2018 (10 tweets)
#Berlusconi (2 tweets)
#Salvini, (2 tweets)
#Casapound, (2 tweets)
#Femen (2 tweets)
#elezioni (2 tweets)
#Macron (2 tweets)
#femen (1 tweets)
#BERLUSCONI (1 tweets)
#italian… (1 tweets)

Keyword: Donald Trump
Results:
#Trump (25 tweets)
#almalki (10 tweets)
#TheResistance (8 tweets)
#News (8 tweets)
#SundayMorning (6 tweets)
#WeLoveTrumpCrumbs (6 tweets)
#FBRParty (4 tweets)
#fbr (4 tweets)
#Trump's (4 tweets)
#Lunacy (4 tweets)

Keyword: Space X
Results:
#domains (15 tweets)
#defenses (12 tweets)
#defense (12 tweets)
#nasa (12 tweets)
#spacex (12 tweets)
#americanmade (7 tweets)
#usa (7 tweets)
#s… (7 tweets)
#deficit (5 tweets)
#overseas!! (5 tweets)

Keyword: Good Morning
Results:
#개학과개강을축하해 (24 tweets)
#좋은인연이생기길 (24 tweets)
#잘자요 (24 tweets)

#정답을알려줘셀카의신예성신 (24 tweets)
#어느새3월!!! (24 tweets)
#아무말트윗 (24 tweets)
#해외팬들은goodmorning (24 tweets)
#월간보라알 (24 tweets)
#goodmorning (18 tweets)
#coffee (8 tweets)

Keyword: coffee
Results:
#coffee (8 tweets)
#coffelover (7 tweets)
#servicewelt (7 tweets)
#selfmade (7 tweets)
#instafood… (7 tweets)
#edna (7 tweets)
#handmade (7 tweets)
#progressivehouse… (4 tweets)
#Ibiza (4 tweets)
#deephouse (4 tweets)

8C. Evaluating the distributed system

Evaluating the system through the following criteria:

- **scalability**: it's easy to add as many consumers as needed. Just keep in mind the rules given at section 8A: one producer for topic with multiple consumers.
- **openness**: consumers, producers and even topics can be added at any time. The system is totally open.
- **fault tolerance**: what happen when a fail occurs? Kafka automatically reassigns consumers to other partitions if available. Given a number of partition as number of consumers + N, we can handle at most 9 consumers failure. Even the topic can fail. In this case, Kafka provides mechanism for disaster recovery, copying the topic periodically on another broker. This requires more bandwidth, but it's definitely worth it. What if the producer fails? Well, the system will keeps working until data can be consumed. However, I would dare to say that this is the critical part.
- **cooperation**: we really did not considered criteria. Each consumers can compute and provide results independently, but what if a consumer needs results from other consumer's activity for its own computation? This lead us to face a problem of concurrency, aiming to guarantee synchronisation among different consumers.

9. Possible extensions

According to the "multiple topic/single producer" architecture, it becomes simple to add new functionalities. To do so, it is sufficient to add a new producer and define its methods. Moreover, what would make the system user friendly is the possibility to have a web GUI. For example, an user could connect to a website and insert a keyword. After that, Producers are triggered and start producing data on the topics while consumer would

analyse data for extract insights. Another thread should update the GUI showing the results that will be constantly updated.

10. Conclusion

To sum up, Kafka seems to be an optimal solution for real time processing streams of data. Actually, we did only exploit producers and consumers but the framework offers also other interesting features such as Data Storage and Stream Processors. However, Kafka performs excellently for scalability and openness. As every software architecture it is fundamental to take right decisions during the design phase. Later on, huge refactories can be tricky. Each component can be written both in Java or Scala, so this should make things easier for developers considering that these are amongst the most adopted programming languages in the world.

The core of Kafka is the idea of topics partitioning. This is clever because it promotes both parallelism and scalability. Personally, I guess in few years other technologies will design system sharing the same concept.

As a software developer, I felt that what is really missing is a tool that can show to the user the current status of the system. For instance, if we want to know how many partitions are there in the topic, how many topics are available, what's the current offset and so on we must type hundreds of different commands on the terminal. Wouldn't be nice to have a single command showing all the info? Or even better a graphic tool?
This makes it clear that Kafka is not a good solution for Agile development. In fact, here rapid prototyping is a key factor and, in order to achieve this, developers need tools and frameworks that can be setup and controlled very quickly. However, Kafka is still at version 1.0.0. and many companies are shifting their software architecture trough the use of Kafka as framework for handling streams of data. I feel confident in saying that this lack of features will be filled in the short term.

11. References

1. *Personal Notes taken during lectures of Distributed Systems* (ay 2016/17)
2. *Kafka documentation*: https://kafka.apache.org/documentation
3. *STData Labs*, http://stdatalabs.blogspot.in
4. *Twitter API*, https://developer.twitter.com/docs
5. *Publishing with Kafka at NYT*, Borgen Svingen
6. *Kafka inside Keystone pipeline*, Netflix tech blog