MODULE - 6

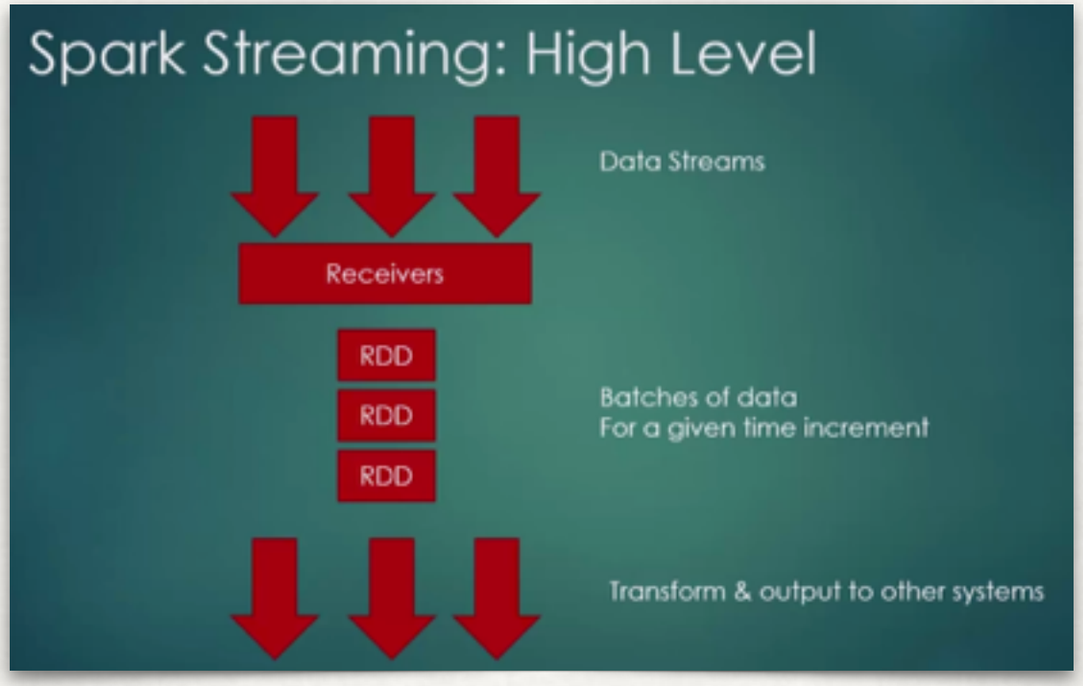
SPARK STREAMING

WHY SPARK STREAMING???

Why Spark Streaming?

- "Big data" never stops!
- Analyze data streams in real time, instead of in huge batch jobs daily
- Analyzing streams of web log data to react to user behavior
- Analyze streams of real-time sensor data for "Internet of Things" stuff

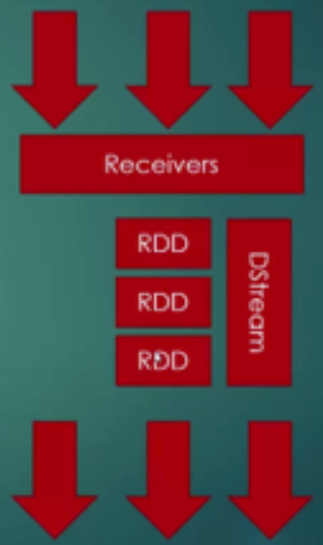
SPARK STREAMING: HIGH LEVEL



DSTREAMS

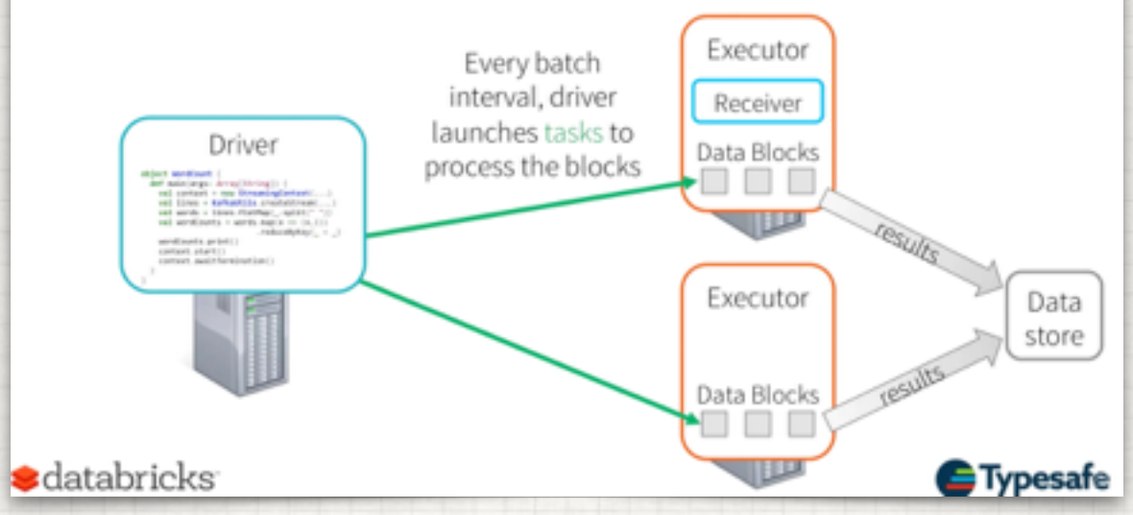
DStreams (Discretized Streams)

- Generates the RDD's for each time step, and can produce output at each time step.
- Can be transformed and acted on in much the same way as RDD's
- Or you can access their underlying RDD's if you need them.



WORK CAN BE DISTRIBUTED

Spark Streaming Application: Process data



wordcount.scala | PrintTweets.scala | Utilities.scala

```scala
import org.apache.spark.streaming.twitter._
import org.apache.spark.streaming.StreamingContext._
import org.apache.log4j.Level
import Utilities._

/** Simple application to listen to a stream of Tweets and print them out */
object PrintTweets {

  /** Our main function where the action happens */
  def main(args: Array[String]) {

    // Configure Twitter credentials using twitter.txt
    setupTwitter()

    // Set up a Spark streaming context named "PrintTweets" that runs locally using
    // all CPU cores and one-second batches of data
    val ssc = new StreamingContext("local[*]", "PrintTweets", Seconds(1))

    // Get rid of log spam (should be called after the context is set up)
    setupLogging()

    // Create a DStream from Twitter using our streaming context
    val tweets = TwitterUtils.createStream(ssc, None)

    // Now extract the text of each status update into RDD's using map()
    val statuses = tweets.map(status => status.getText())

    // Print out the first ten
    statuses.print()

    // Kick it all off
    ssc.start()
    ssc.awaitTermination()
  }
```

Windowed Transformations

- Allow you to compute results across a longer time period than your batch interval
- Example: top-sellers from the past hour
  - You might process data every one second (the batch interval)
  - But maintain a window of one hour
- The window "slides" as time goes on, to represent batches within the window interval

TYPE OF INTERVALS IN WINDOWING

Batch interval vs. slide interval vs. window interval

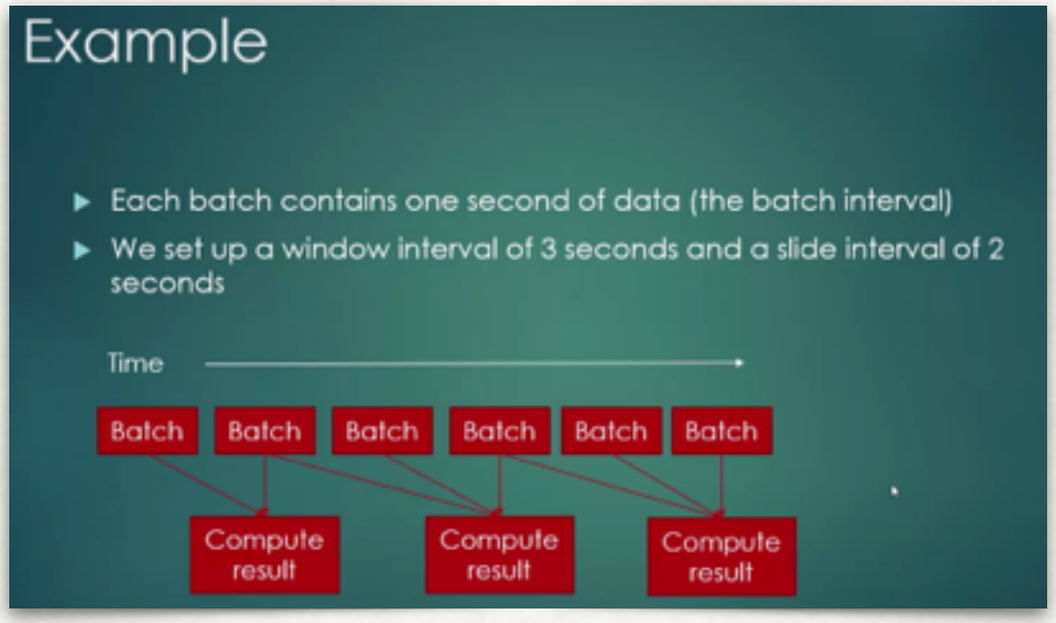
- The batch interval is how often data is captured into a Dstream
- The slide interval is how often a windowed transformation is computed
- The window interval is how far back in time the windowed transformation goes

1 Sec

5 Mins

6 Mins

Capture - 1 Sec = Batch Interval
Process Data - 2 Sec = Slide Interval
Data considered for processing - 3 Sec = Window Interval



www.KAGGLE.com- Project Collaboration

Transformations on Window Operations

Transformation	Meaning
window(windowLength, slideInterval)	Return a new DStream which is computed based on windowed batches of the source DStream
countByWindow(windowLength, slideInterval)	Return a sliding window count of elements in the stream
reduceByWindow(func, windowLength, slideInterval)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using func.
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window
countByValueAndWindow(windowLength, slideInterval, [numTasks])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window

StreamingContext ==> SparkContext

reduceByKey(_ * _) (x,1)(x,2)(y,1)(y,3) => (x,2)(y,3)

reduce(_ + _) (1,2,3,4,5,6,7,8, 300) => (36)

Windowed transformations: code

- The batch interval is set up with your SparkContext, as usual:

```
val ssc = new StreamingContext("local[*]", "PopularHashtags",
Seconds(1))
```

- You can use reduceByWindow() or reduceByKeyAndWindow() to aggregate data across a longer period of time!

```
val hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow(
(x,y) => x + y, (x,y) => x - y, Seconds(300), Seconds(1))
```

COMMON TRANSFORMATIONS

Common stateless transformations on DStreams

- Map
- Flatmap
- Fliter
- reduceByKey

FAULT - TOLERANT

Spark Streaming is Fault-Tolerant

- Incoming data is replicated to at least 2 worker nodes
- A checkpoint directory can be used to store state in case we need to restart the stream (to HDFS, S3, etc.)
  - Just use ssc.checkpoint() on your StreamingContext
  - Checkpoints are required if you are using stateful data

LIMITS

But there are limits

- What if your receiver fails?
- What if your driver script fails?

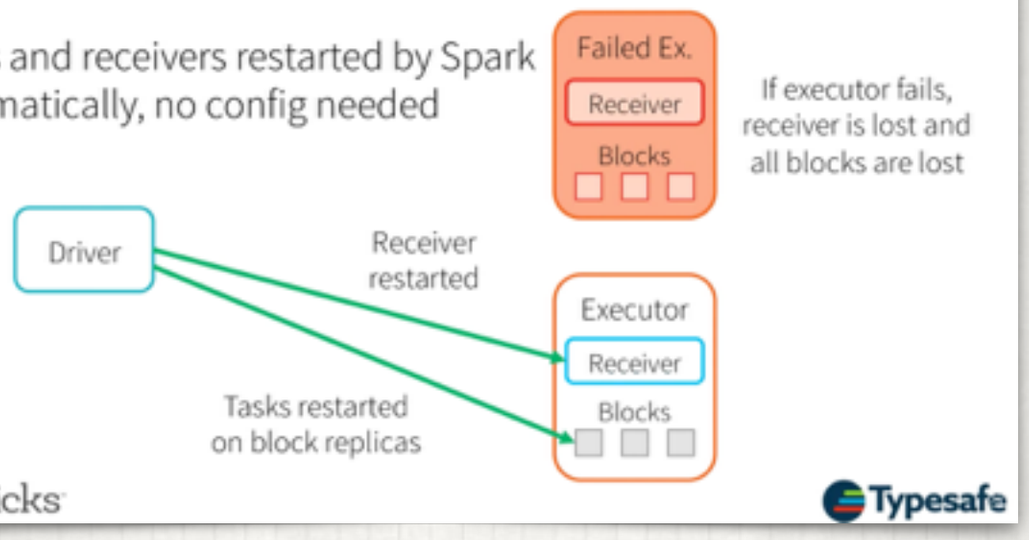
WHAT IF RECEIVER FAILS??

Dealing with receiver failure

- Some receivers are better than others.
- If your Twitter receiver fails for example, that data is just plain lost.
- Same with configurations of Kafka or Flume that only "push" data to Spark
- But receivers based on replicated, reliable data sources are more resilient
  - HDFS
  - Directly-consumed Kafka
  - Pull-based Flume

What if an executor fails?

Tasks and receivers restarted by Spark automatically, no config needed



databricks

Typesafe

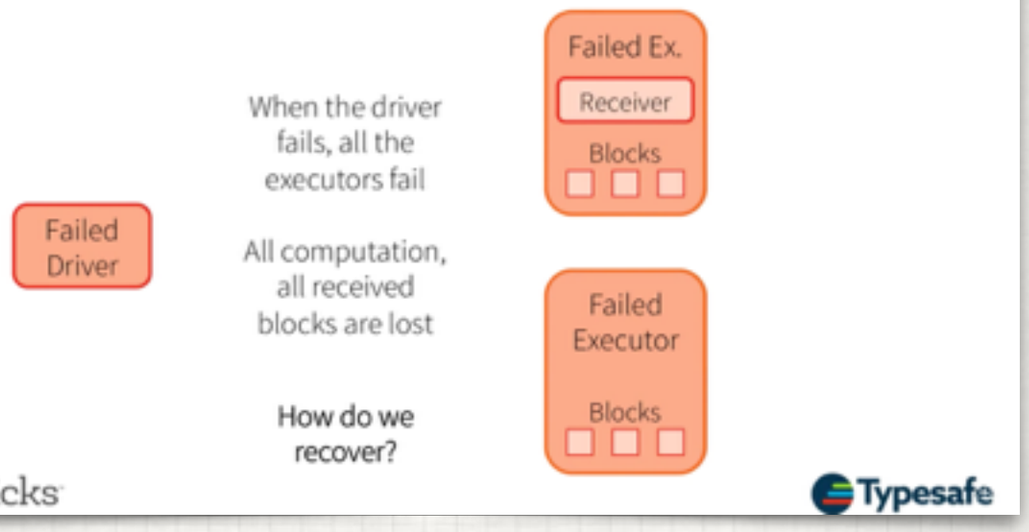
DRIVER FAILURE

ssc.checkpoint(<>)

Dealing with driver script failure

- Although the data worker nodes deal with is replicated, your driver node can be a single point of failure.
- Instead of working with the StreamingContext you created directly, use the one returned by StreamingContext.getOrCreate()
  - …and use a checkpoint directory on a distributed file system.
  - ssc.getOrCreate(checkpointDir, <function that creates a new StreamingContext>)
  - Either gets a StreamingContext from the checkpoint directory, or creates a new one.
- Then if you need to restart your driver script, it can pick up from the checkpoint directory.
- Still need to monitor the driver node for failure, and restart the script if it does.
  - Zookeper and Spark's built-in cluster manager can do this for you (use –supervise on spark-submit)

What if the driver fails?



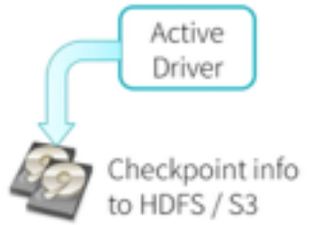
databricks

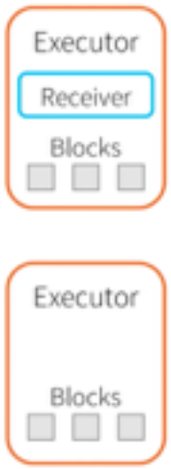
Typesafe

Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage





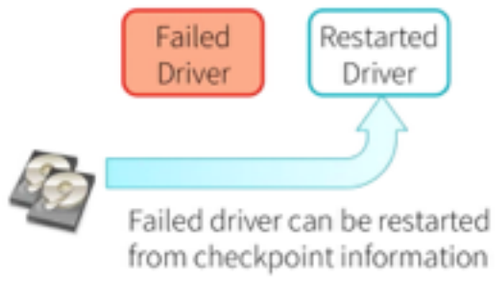
databricks

Typesafe

Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



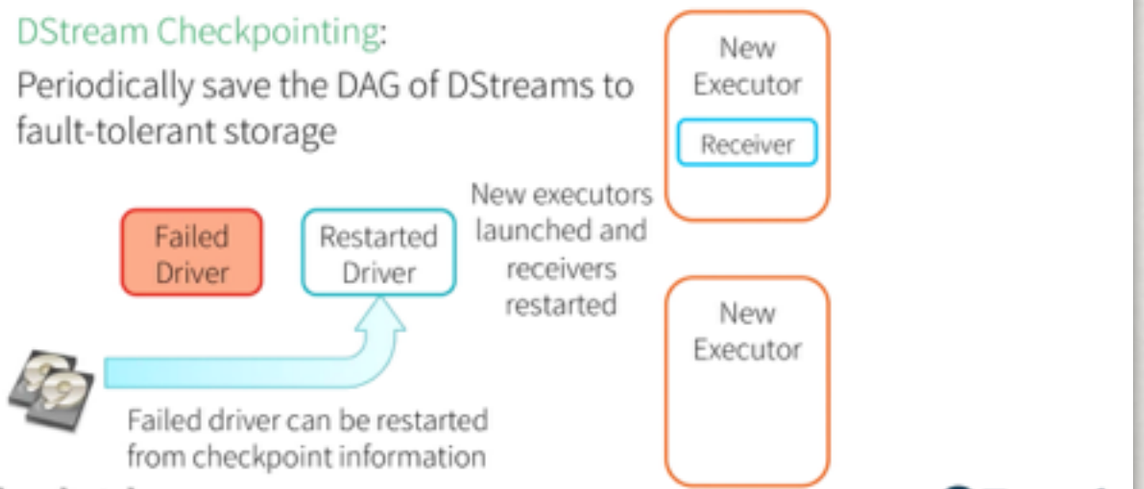
databricks

Typesafe

Recovering Driver w/ DStream Checkpointing

DStream Checkpointing:

Periodically save the DAG of DStreams to fault-tolerant storage



databricks

Typesafe

Recovering Driver w/ DStream Checkpointing

1. Configure automatic driver restart
   All cluster managers support this

2. Set a checkpoint directory in a HDFS-compatible file system
   ```
   streamingContext.checkpoint(hdfsDirectory)
   ```

3. Slightly restructure of the code to use checkpoints for recovery

Configurating Automatic Driver Restart

Spark Standalone – Use spark-submit with "cluster" mode and "--supervise"

See http://spark.apache.org/docs/latest/spark-standalone.html

YARN – Use spark-submit in "cluster" mode

See YARN config "yarn.resourcemanager.am.max-attempts"

Mesos – Marathon can restart applications or use the "--supervise" flag.

Restructuring code for Checkpointing

Create + Setup

```
val context = new StreamingContext(...)
val lines = KafkaUtils.createStream(...)
val words = lines.flatMap(...)
...
```

```
def creatingFunc(): StreamingContext = {
   val context = new StreamingContext(...)
   val lines = KafkaUtils.createStream(...)
   val words = lines.flatMap(...)
   ...
   context.checkpoint(hdfsDir)
}
```

Put all setup code into a function that returns a new StreamingContext

Start

```
context.start()
```

```
val context =
StreamingContext.getOrCreate(
  hdfsDir, creatingFunc)
context.start()
```

Get context setup from HDFS dir OR *create* a new one with the function

databricks

Typesafe

1) #RELIANCEJIO XYZ GH;K;KFL;;KF; KAS;FK;SF #TRUMP #PMOINDIA

2) #PMOINDIA, #RELIANCEJIO, #PMOINDIA => MAP(_,1) ==>

3) (#PMOINDIA,1) (#RELIANCEJION,1) (#PMOINDIA,1) ==>

4) REDUCEBYKEYANDWINDOW ==========>
((#PMOINDIA,2)
(#RELIANCEJIO,1))

5) (6,#TRUMP) (2,#PMOINDIA) (1,#RELIANCEJIO)

6) POPULAR TOPICS IN LAST 60 SECONDS (25 TOTAL)
7) (6,#TRUMP)
(2,#PMOINDIA)
(1,#RELIANCEJIO)

8) POPULAR TOPICS IN LAST 10 SECONDS (10 TOTAL)
9) (6,#SALMANKHAN)
(2,#PMOINDIA)
(1,#RELIANCEJIO)

A FEW SUPPORTED ALGORITHMS

What is Logistic Regression?

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, the logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

What is Naive Bayes algorithm?

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

DEFINITIONS OF A FEW TERMS

Spark ML Concepts

edureka!

DataFrame	Spark ML uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions
Transformer	A Transformer is an algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer which transforms DataFrame with features into a DataFrame with predictions
Estimator	An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model
Pipeline	A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow
Parameter	All Transformers and Estimators now share a common API for specifying parameters

FEATURE VECTORS

In pattern recognition and machine learning, a **feature vector** is an n-dimensional **vector** of numerical **features** that represent some object. Many algorithms in machine learning require a numerical representation of objects, since such representations facilitate processing and statistical analysis.

DEFINITIONS OF A FEW TERMS

LABELS

Supervised **learning** is the **machine learning** task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised **learning**, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal).

THE KAFKA MESSAGING QUEUE

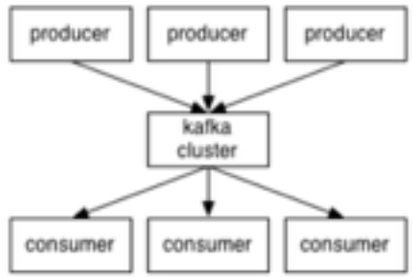
WHAT IS APACHE KAFKA?

Apache Kafka differs from traditional messaging system in:

- It is designed as a distributed system which is very easy to scale out.
- It offers high throughput for both publishing and subscribing.
- It supports multi-subscribers and automatically balances the consumers during failure.
- It persist messages on disk and thus can be used for batched consumption such as ETL, in addition to real time applications.

KAFKA COMPONENTS

- A stream of **messages** of a particular type is defined as a **topic.** A **Message** is defined as a payload of bytes and a **Topic** is a category or feed name to which messages are published.
- A **Producer** can be anyone who can publish messages to a Topic.
- The published messages are then stored at a set of servers called **Brokers or Kafka Cluster**.
- A **Consumer** can subscribe to one or more Topics and consume the published Messages by pulling data from the Brokers.



SAMPLE CODES

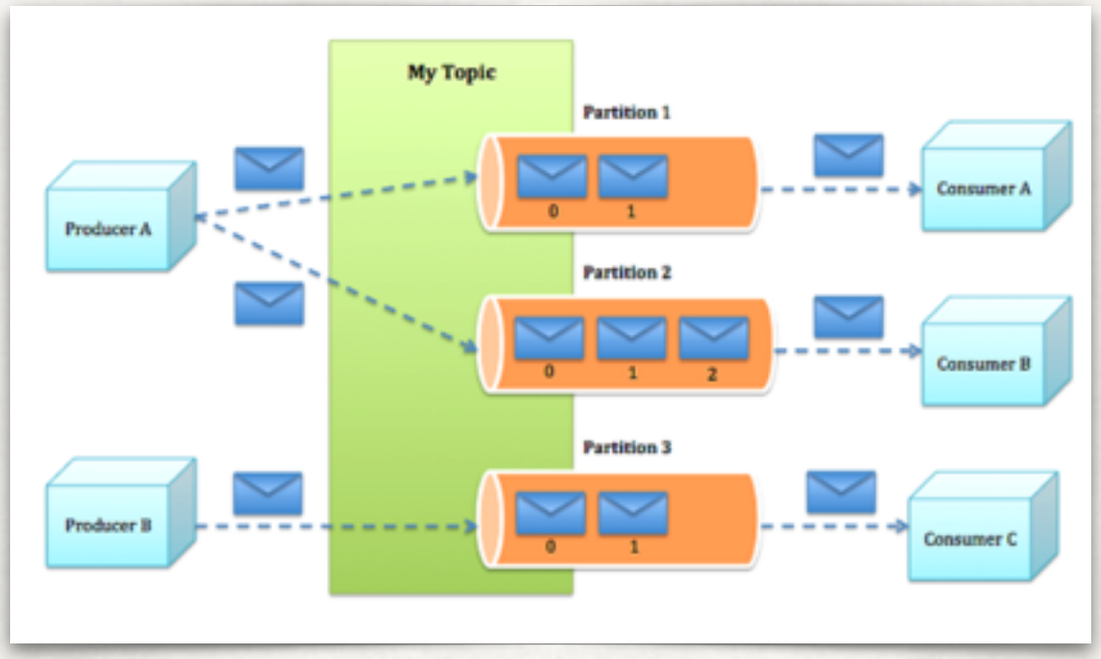
Sample producer code:

```
producer = new Producer(…);
message = new Message(“test message str”.getBytes());
set = new MessageSet(message);
producer.send(“topic1”, set);
```

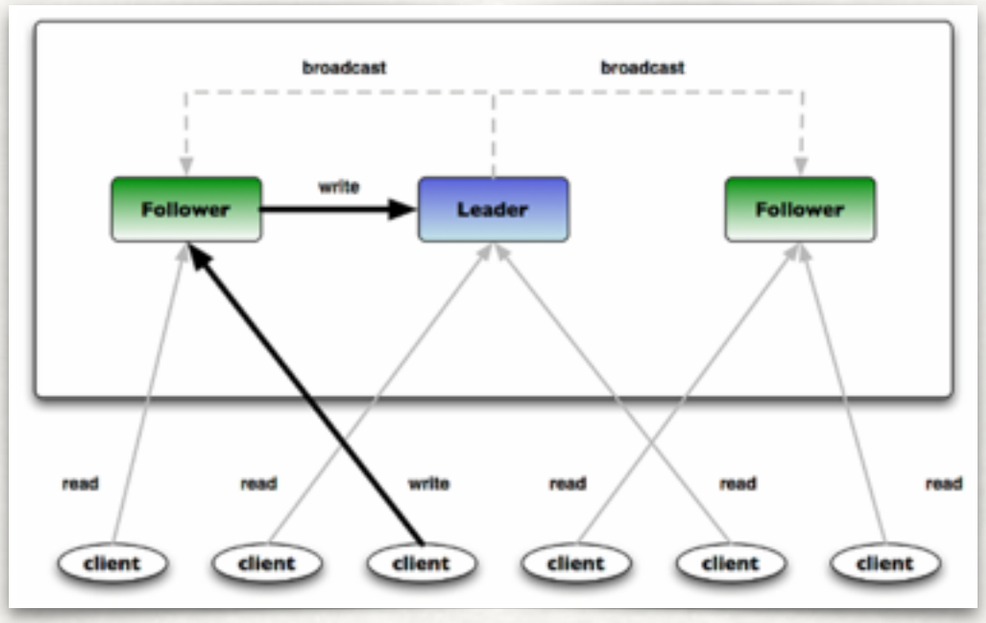
Sample consumer code:

```
streams[] = Consumer.createMessageStreams(“topic1”, 1)
for (message : streams[0]) {
bytes = message.payload();
// do something with the bytes
}
```

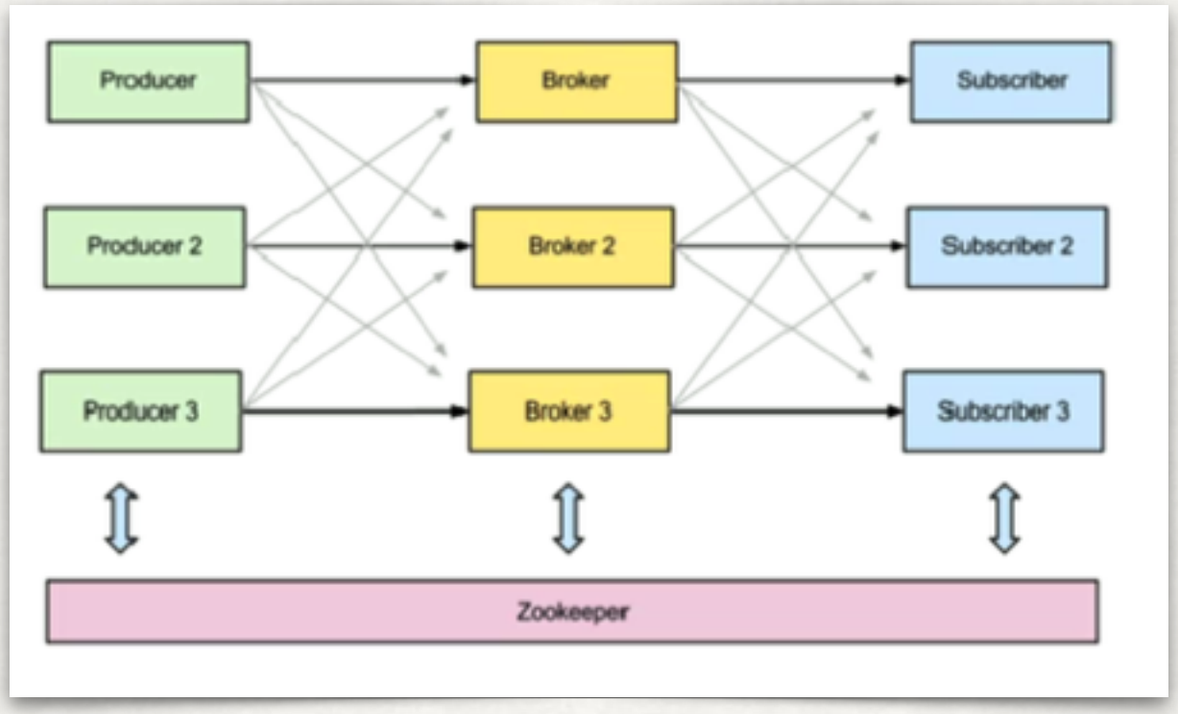
KAFKA ARCHITECTURE



ZOOKEEPER ENSEMBLE



KAFKA WITH ZOOKEEPER



TRADITIONAL KAFKA BASED APPLICATION FLOW