**USING APACHE KAFKA TO BUILD PIPELINES FOR STREAMING DATA**

# Distributed Event Streaming Platform Components

**Event:** They describe an entity’s observable state updates over time.

Example: GPS coordinates of a car, Temperature of a room, Blood pressure of a patient etc…

**Common Event formats:**

* Primitive: int, float, string etc…
* Key-value pair: Key:”car\_id\_1”

Value: (43.82, -79.48)

* Key value pair with timestamp: Key: “patient\_id”

Value: (125, 85)

Timestamp: 2021-07-01 12:00

**Event streaming from one source to one destination:**

**A screen shot of a computer

Description automatically generated**

**Event streaming from many sources to many destinations:**

**A diagram of a event

Description automatically generated**

An event destination can also act as an event source.

In real-world scenario, as data transfer pipelines may be based on different communication protocols such as:

* + FTP: File Transfer Protocol
  + HTTP: HyperText Transfer Protocol
  + JDBC: Java DataBase Connectivity
  + SCP: Secure Copy

**Event Steaming Platform (ESP)**

An ESP acts as a middle layer among various event sources and destinations and provides a unified interface for handling event-based ETL.

A screen shot of a computer

Description automatically generated

As such, all event sources only need to send events to an ESP instead of sending them to the individual event destination.

On the other side, event destinations only need to subscribe to an ESP and just consume the events sent from the ESP instead of the individual event source.

Different ESPs may have different architectures and components.

Common components of ESP systems include:

A screenshot of a computer

Description automatically generated

* **Event Broker:** Designed to receive and consume events. It contains three-sub components:

**A diagram of a process

Description automatically generated**

* + Ingester: It is designed to efficiently receive events from various event sources.
  + Processor: It performs operations on data such as serializing and deserializing, compressing and decompressing, encryption and decryption and so on.
  + Consumption: It receives events from event storage and efficiently distributes them to subscribed event destinations.
* **Event Storage:** Used for storing events being received from event sources. Accordingly, event destinations do not need to synchronize with event sources, and stored events can be retrieved at will.
* **Analytic and Query Engine:** Used for querying and analyzing stored events.

**Common ESP solutions:**

Apache Kafka Amazon Kinesis Apache Flink

Apache storm Apache spark

Logstash in Elastic Stack etc

ESP is needed especially when there are multiple event sources and destinations.

# Apache Kafka Overview

A screen shot of a computer

Description automatically generated

Common use cases:

1. To track **User activities** like keyboard strokes, muse clicks, page views, searches, gestures etc.
2. For all kinds of **metric streaming** such as readings, GPS as well as H/W and S/W monitoring.
3. For **enterprise applications** and **infrastructure** with huge number of **logs,** Kafka can be used to collect and integrate them into a centralized repository.
4. For banks, insurance or FinTech companies, Kafka is also used for payments & transactions.
5. When you want **high throughput** and **reliable data transportation services** among various event sources and destinations.
6. **Data storage** and **movement** to other online or offline databases and backups.
7. **Real-time processing and analytics** including dashboard, ML, AI algorithms and so on.
8. Generating **notifications** such as email, text messages, or data governance and auditing to make sure sensitive data such as bank transactions are complying with regulations.

**Kafka Architecture – A distributed client-server architecture:**

**A screenshot of a computer

Description automatically generated**

* Kafka has a distributed client-server architecture.
* For the server side, Kafka is a cluster with many associated servers called brokers, acting as the event broker to receive, store and distribute events.

All those brokers are managed by another distributed system called **zookeeper** to ensure all brokers work in an efficient and collaborative way.

* Kafka uses **TCP (Transmission Control Protocol)** based on network communication protocol to exchange data between clients and servers.
* For client side, Kafka provides different types of clients, such as **Kafka CLI** which is a collection of shell scripts to communicate with the kafka server, **many high level programming APIs** such as Java, Scala, REST APIs and some specific third party clients made by the kafka community. Accordingly, you would choose different clients based on your requirements.

**Kafka Features:**

* Highly scalable to handle high data throughput and concurrency.
* Kafka cluster normally has multiple event brokers which can handle event streaming in parallel.
* Kafka is very fast and highly scalable.
* It also divides event storage into multiple partitions and replications which makes it fault-tolerant and highly reliable.
* It stores the events permanently.
* Event consumption could be done whenever suitable for consumers without a deadline.
* It is open source, it is free to use and even customize it based on your specific requirements.

Deploying kafka cluster requires extensive efforts for tuning infrastructure and consistently adjusting the configurations, especially for enterprise level deployments.

Several commercial service providers offer an on-demand **ESP-as-a-service** to meet your streaming requirements.

**Some well known ESP providers:**

* Confluent cloud
* IBM Event streams
* Amazon Managed Streaming for Apache Kafka

**Kafka Use Cases:**

* User activity tracking
* Metrics and logs integration
* Financial transaction processing

# Building Event Streaming Pipelines using Kafka

**Broker and Topic:**

* Kafka cluster contains one or many brokers.
* Kafka broker can be considered as a dedicated cluster to receive, store, process and distribute events.
* Brokers are synchronized and managed by another dedicated server called **zookeeper.**

A screenshot of a computer

Description automatically generated

* Brokers manage to save published events into topics and distribute the events to subscribed consumers.
* Like many distributed systems, kafka also implements partition and replications.
* It uses topic partitions and replications to increase fault tolerance and throughput so that event publication and consumption can be done in parallel with multiple brokers.
* Even if some brokers are down, kafka clients are still able to work with the target topics replicated in other working brokers.

Eg:

A screenshot of a computer

Description automatically generated

A log topic has been separated into 2 partitions: 0,1 and a user topic has been separated into 2 partitions: 0,1. And each partition is duplicated into two replications and stored in different brokers. The kafka CLI provides a collection of powerful script files for users to build an event streaming pipeline. The kafka topics script is the one you probably will be using often to manage topics in a kafka cluster.

**Common** **usages**:

A screenshot of a computer

Description automatically generated

Kafka topics script is the one you use to manage topics in a kafka cluster.

In the above diagram:

Step-1: Create a topic: log\_topic—here we are creating a topic with 2 partitions and 2 replicaitons. Many kafka topics needs to be referred to a running kafka cluster with a host and a port.

Step-2: List topics-checking the topics created.

Step-3: Getting more details about the topic.

Step-4: You can delete a topic using delete option.

**Kafka Producer:**

**Features:**

* Client applications that publish events to topic partitions
* An event can be optionally associated with a key.
* Events associated with the same key will be published to the same topic partition.
* Events not associated with any key will be published to topic partitions in rotation.

**Kafka producer in action:**

**A diagram of a company

Description automatically generated**

Suppose you have an event source-1 that produces log entries and event source-2 that produces user activities records. Then you can create a kafka producer to publish log records to log topic partitions and a user producer to publish user activity events to user topic partitions respectively. When you publish events in producers, you can choose to associate events with a key such as UserID or appName.

Similar to Kafka topic CLI, kafka also has a **Kafka Producer CLI** for users to manage producers.

**A close-up of a computer screen

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Kafka consumer:**

Once events are published and properly stored in topic partitions, you can create consumers to read them.

* Consumers are client applications that can subscribe to topics and read stored events.
* Then event destinations can further read events from Kafka consumers.
* Consumers read data from topic partitions in the same order as they are published.
* Consumers also store an offset for each topic partition as the last read position.
* With the offset, consumers are guaranteed to read events as they occur.
* Playback is also possible by resetting the offset to zero. This way the consumer can read all events in the topic partition from the beginning.
* Kafka, producers, and consumers are fully decoupled. As such, producers don’t need to synchronize with consumers, and after events are stored in topics, consumers can have independent schedules to consume them.
* To read published log and user events from topic partitions, you will need to create log and user consumers, and make them subscribe to corresponding topics. Then kafka will push the events to those subscribed consumers. Then, the consumers will further send to event destinations.

A diagram of a website

Description automatically generated

**Kafka Consumer CLI:**

A screen shot of a computer

Description automatically generated

The started consumer will read only the new events, starting from the last partition offset. After those events are consumed, the partition offset for the consumer will also be updated and committed back to Kafka. Very often a user wants to read all events from the beginning, as a playback of all historical events. To do so, you just need to add the ‘from beginning’ option. Now, you can read all events starting from offset 0.

**Example:**

A weather pipeline example:

A diagram of a diagram

Description automatically generated

**Explanation:**

Suppose you want to collect and analyze weather and Twitter event streams, so that you can correlate how people talk about extreme weather on Twitter. Here you can use two event sources: IBM Weather API to obtain real-time and forecasted weather data in JSON format. And Twitter API to obtain real-time tweets and mentions, also in JSON format. To receive weather and twitter JSON data in Kafka, you then create a weather topic and a Twitter topic in a Kafka cluster, with some partitions and replications. To publish weather and Twitter JSON data to the two topics, you need to create a weather producer and a Twitter producer. The event’s JSON data will be serialized into bytes and saved in Kafka topics. To read events from the two topics, you need to create a weather consumer and a Twitter consumer. The bytes stored in Kafka topics will be deserialized into event JSON data. If you now want to transport the weather and Twitter event JSON data from the consumers to a relational database, you will use a DB writer to parse those JSON files and create database records. And then you can write those records into a database using SQL insert statements. Finally, you can query the database records from the relational database and visualize and analyze them in a dashboard to complete the end-to-end pipeline.

# Kafka Streaming Process

**Ad hoc weather stream processing:**

Any applications that are designed to process streams are called streaming applications.

A computer screen shot of a diagram

Description automatically generated

For stream processing applications based on Kafka, a straightforward way is to implement an ad hoc data processor to read events from one topic, process them, and publish them to another topic.

You first request raw weather JSON data from a weather API, and you start a weather producer to publish the raw data into a raw weather topic. Then you start a consumer to read the raw weather data from the weather topic. Next, you create an ad hoc data processor to filter the raw weather data to only include extreme weather events, such as very high temperatures. Such a processor could be a simple script file or an application which works with Kafka clients to read and write data from Kafka. Afterwards, the processor sends the processed data to another producer, and it gets published to a processed weather topic. And finally, the processed weather data will be consumed by a dedicated consumer and sent to a dashboard for visualization.

**Kafka Streams API:**

Ad hoc processors may become complicated if you have many processes. Here is where Kafka stream APIs come into play.

* It is a simple client library to facilitate data processing in event steaming pipelines.
* It processes and analyzes data stored in Kafka topics.
* Thus both i/ps and o/ps of kafka APIs are kafka topics.

A diagram of a company

Description automatically generated

* Each record is processed only once.
* It processes one record at a time.

**Stream Processing Topology:**

Each node 🡪 Stream processor (which receives streams from its upstream processor, performs data transformations such as data mapping, filtering, formatting, and aggregation and produces o/p streams to its downstream processors. Thus the edges of the graph are the I/O streams).

2 types of processors:

**Source processors:** Have no upstream processors. Source processors acts like consumers that consumes streams from kafka topics and forwards the processed streams to its downstream processors.

**Stream processors:** Used for filtering specific streams from the raw streams.

**Sink processors:** Have no downstream processors. It acts like a producer that publishes the received stream to a kafka topic.

Redesigning the ad hoc stream processing using Kafka stream APIs:

A screenshot of a computer

Description automatically generated

**LAB: Objectives:**

* Download and install Kafka
* Start the zookeeper server for kafka metadata management
* Start the kafka message broker service
* Create a topic
* Start a producer
* Start a consumer

**Step**-1: In a new terminal, download kafka using the command **🡪 wget** [**https://archive.apache.org/dist/kafka/2.8.0/kafka\_2.12-2.8.0.tgz**](https://archive.apache.org/dist/kafka/2.8.0/kafka_2.12-2.8.0.tgz)

**Step**-2: Extract kafka from the zip file by running**🡪 tar -xzf kafka\_2.12-2.8.0.tgz** (This creates a new dir ‘kafka\_2.12-2.8.0’ in the current dir)

**Step**-3: Zookeeper is required for kafka to work. Start zookeeper using:

**cd kafka\_2.12-2.8.0**

**bin/zookeeper-server-start.sh config/zookeeper.properties**

ZooKeeper is responsible for the overall management of Kafka cluster. It monitors the Kafka brokers and notifies Kafka if any broker or partition goes down, or if a new broker or partition goes up.

**Step**-4: Start the kafka message broker service using 🡪

**cd kafka\_2.12-2.8.0**

**bin/kafka-server-start.sh config/server.properties**

**Step**-5: Create a topic

**cd kafka\_2.12-2.8.0**

**bin/kafka-topics.sh --create --topic news --bootstrap-server localhost:9092**

**Step-**6: Start a producer **🡪**

**bin/kafka-console-producer.sh --topic news --bootstrap-server localhost:9092**

**Good morning**

**Good day**

**Enjoy the Kafka lab**

**Step**-7: You need consumers to read messages from kafka. Open new terminal.

**cd kafka\_2.12-2.8.0**

**bin/kafka-console-consumer.sh --topic news --from-beginning --bootstrap-server localhost:9092**

A screenshot of a computer

Description automatically generated

# Message Keys and Offset:

* Use message keys to keep message streams sorted in their original publication state/order.
* Use consumer offset to control and track message sequential positions in topic partitions.

1. **wget** [**https://archive.apache.org/dist/kafka/2.8.0/kafka\_2.12-2.8.0.tgz**](https://archive.apache.org/dist/kafka/2.8.0/kafka_2.12-2.8.0.tgz)
2. **tar -xzf kafka\_2.12-2.8.0.tgz**
3. **cd kafka\_2.12-2.8.0**

**bin/zookeeper-server-start.sh config/zookeeper.properties**

**New Terminal:**

1. **cd kafka\_2.12-2.8.0**

**bin/kafka-server-start.sh config/server.properties**

**Create a topic and producer for processing bank ATM transactions**

1. {“atmid”:1, “transid”:100}
2. cd kafka\_2.12-2.8.0
3. bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic bankbranch --partitions 2
4. bin/kafka-topics.sh --bootstrap-server localhost:9092 –list
5. bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic bankbranch
6. bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic bankbranch
7. Produce messages:

**{"atmid": 1, "transid": 100}**

**{"atmid": 1, "transid": 101}**

**{"atmid": 2, "transid": 200}**

**{"atmid": 1, "transid": 102}**

**{"atmid": 2, "transid": 201}**

1. New Terminal 🡪 cd kafka\_2.12-2.8.0
2. bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic bankbranch --from-beginning

**Produce and consume with message keys:**

At this point you have 4 terminals open:

* zookeeper terminal
* kafka server terminal
* producer terminal – ctrl+c to stop the producer
* consumer terminal – ctrl+c to stop the consumer

1. Stop the producer and consumer.
2. Now you can start new producer and consumer, this time using message keys. You can start a new producer with the following message key commands:
   * **--property parse.key=true** 🡪 to make the producer parse message keys
   * **--property key.separator=:** 🡪 define the key separator to be the “:” character

So our message with key now looks like: **1:{“atmid”:1,”transid”:100}**

1. Start a new producer with the message key enabled:

**bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic bankbranch --property parse.key=true --property key.separator=:**

**1:{"atmid": 1, "transid": 102}**

**1:{"atmid": 1, "transid": 103}**

**2:{"atmid": 2, "transid": 202}**

**2:{"atmid": 2, "transid": 203}**

**1:{"atmid": 1, "transid": 104}**

1. Next switch to consumer terminal again and start a new consumer with:

**bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic bankbranch --from-beginning --property print.key=true --property key.separator=:**

This is because each topic partition maintains its own message queue, and new messages are enqueued (appended to the end of the queue)  
as they get published to the partition. Once consumed, the earliest messages will be dequeued and nno longer be available for consumption.

Recall that with two partitions and no message keys specified, the transaction messages were published to the two partitions  
in rotation:

* Partition 0: [{"atmid": 1, "transid": 102}, {"atmid": 2, "transid": 202}, {"atmid": 1, "transid": 104}]
* Partition 1: [{"atmid": 1, "transid": 103}, {"atmid": 2, "transid": 203}]

As you can see, the transaction messages from atm1 and atm2 got scattered across both partitions. It would be difficult  
to unravel this and consume messages from one ATM with the same order as they were published.

However, with message key specified as the atmid value, the messages from the two ATMs will look like the following:

* Partition 0: [{"atmid": 1, "transid": 102}, {"atmid": 1, "transid": 103}, {"atmid": 1, "transid": 104}]
* Partition 1: [{"atmid": 2, "transid": 202}, {"atmid": 2, "transid": 203}]

Messages with the same key will always be published to the same partition, so that their published order will be preserved within the message queue of each partition.

**Show the details of consumer group:**

**bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group atm-app**

# Kafka Python Client

**https://github.com/P-Swarnamayee/ETL-and-Data-Pipelines-with-Apache-Spark-Airflow-and-Kafka/blob/main/kafka%20python%20client.pdf**