A distributed pub/sub platform: Apache Kafka (Part 1)

Prof. Carlo Ferrari
Michele Stecca, Ph.D.
Request-Response model
Request-Response: some problems

• Bad for multiple receivers

• Tightly coupled

• Client is waiting for response

• Web Service chaining
Event-driven model
Event-driven model: some considerations

• Loosely coupled

• Works even though client(s) is(are) offline

• Message delivery issues
Software platforms for event-driven apps
Event-driven model

Some use cases:
• Managing “Events”
  • Anything happened (or didn’t happen)
  • A change in the state
  • A condition that triggers a notification
• IoT (Internet of Things) data source
• Change Data Capture (CDC) for databases
• Near real-time data processing
• …
Why Web APIs?

Web APIs as “an internal bus” for Microservices
Point to Point Pattern

- A *point-to-point* channel delivers a message to exactly one of the consumers that is reading from the channel.
- Each service depends on other services directly.
- Services are directly coupled to each other APIs.
- Services know about and understand their dependencies.

Source: https://medium.com/@knoldus/distributed-messaging-patterns-reactive-architecture-c7f0ce989fe0
Why event-driven platforms?

Event-driven (a.k.a., Message Queue, Pub/Sub) as “an internal bus” for Microservices
Publish/Subscribe Pattern

- Services publish messages to a common message bus.
- Other services subscribe to the messages.
- The publishing service has no knowledge of the subscribing services.
- Subscribing services also has no knowledge of the publishing services.
- Services are completely decoupled as they have no knowledge of each other.
- Services are coupled to only the message format.
Apache Kafka

- Distributed publish-subscribe messaging system
- Designed for processing of real time activity stream data (log, metrics, collections, social media streams,.....)
- Does not use JMS (Java Messaging Service) API and standards
- Kafka maintains feeds of message in topics
- Initially developed at Linkedin, then part of Apache
- Current commercial version by Confluent
Main Components

- A **Producer** is an entity/application that publishes data to a Kafka cluster, which is made up of **brokers**.
- A **Broker** is responsible for receiving and storing the data when a producer publishes.
- A **Consumer** then consumes data from a broker at a specified offset, i.e. position.
Pub/Sub paradigm
A **Topic** is a category/feed name to which records are stored and published.

Each **Topic** is divided into **partitions**.

Each partition is an ordered, immutable sequence of messages that is **continually appended to**.

**The message order is only guaranteed inside a partition** (i.e., the FIFO property is only guaranteed inside a partition).

A message in a partition is identified by a sequence number called **offset**.

**Consumers** subscribe to topics.

Consumers with different group-id receive all messages of the topics they subscribe to. They consume the messages at their own speed.

Consumer offsets are persisted by Kafka with a commit/auto-commit mechanism.
Log Anatomy

```
Data Source
  | writes
  ↓
Log
  | reads
  ↓
  0 1 2 3 4 5 6 7
  | reads
  ↓
Destination 1 (offset 4)
Destination 2 (offset 7)
```
Log Anatomy – Producer side

A topic in Kafka is broken into multiple partitions

Partitions are the way that Kafka provides scalability and redundancy
Log Anatomy – Consumer side

Partition

0 1 2 3 4 5 6 7

Consumer-1 (offset 3)
Consumer-2 (offset 5)

Each consumer has its own view about the partition.
Kafka Messages

• A message contains
  - key-value pair (the key is optional)
  - timestamp
  - headers (optional)

• All data is stored in Kafka as byte arrays
• Producer provides serializers to convert the key and value to byte arrays
• Key and value can be any data type
The partitioning strategy is specified by the Producer

- Default strategy is a hash of the message key
  \[ hash(key) \% number\_of\_partitions \]

- If a key is not specified, messages are sent to Partitions on a round-robin basis

- Developers can provide a custom partitioner class
Consumers sharing the same group-id will be assigned to one (or several) partition of the topics they subscribe. They only receive messages from their partitions. So a constraint appears here: the number of partitions in a topic gives the maximum number of parallel consumers.

The assignment of partitions to consumer can be automatic and performed by Kafka (through Zookeeper). If a consumer stops polling or is too slow, a process call “re-balancing” is performed and the partitions are re-assigned to other consumers.
Consumers and Consumer Groups

When a consumer group consumes the partitions of a topic, Kafka makes sure that each partition is consumed by exactly one consumer in the group.
Consumers and Consumer Groups
Consumers and Consumer Groups

Topic with 3 partitions

Consumer Group

Idling consumer