Real-time data processing with

# kafka

TOBIAS FISCHER

# Apache Kafka

OVERVIEW

- Distributed event streaming platform
- Open Source 

- Written in Java and Scala
- Key Capabilities
  - high throughput and low latency
  - fault tolerance
  - scalability
  - data integrity

# Event Streaming

WHAT IS IT?

Processing, transmitting and storing data in real time

- data is viewed as continuous streams of events
- from various sources (databases, sensors, mobile devices, cloud services, ...)
- processed in real time or afterwards

→ right data is available to right application at the right time

Kafka combines three key functions for event streaming:

- **Publish** (write) and **Subscribe** (read) to events
- Store Streams
- Process Streams

# Functionality

SERVER & CLIENTS

**TCP** is used for communication

**Server/Broker**

- distributed cluster
- manage data storage layer
- Kafka Connect: integration of external systems

**Client**

- event stream processing (read/write)
  - resilient to network disruption and hardware failures

# Functionality

EVENTS

*Notifications of something noteworthy*

→ Kafka's read/write operations

Example Event:

- **Key**: ”john.doe@example.com”
- **Value**: ”Hello World!”
- **Timestamp**: ”01.01.1970 00:00:00”
- (Optionally) Metadata Headers

# Functionality

PRODUCERS & CONSUMERS

**Producers**: publish events to Kafka

**Consumers**: subscribe and process those events

*Decoupled Architecture*

- producers and consumers can operate independently
  → high scalability (without the need to wait for each other)
- enhanced by Kafka's ability to guarantee data integrity
  → ensure events are only processed once

# Functionality

TOPICS

Events organized and persistently stored within topic

- more or less similar to folder in filesystem
  - events acting as the files within that folder
- Kafka relies heavily on the filesystem

Events are persisted

- unless other traditional messaging systems
- stored for a predefined period of time
- performance remains consistent regardless of data size

# Functionality

PARTITIONS

Division of the topics into several partitions

- enhance scalability and concurrency

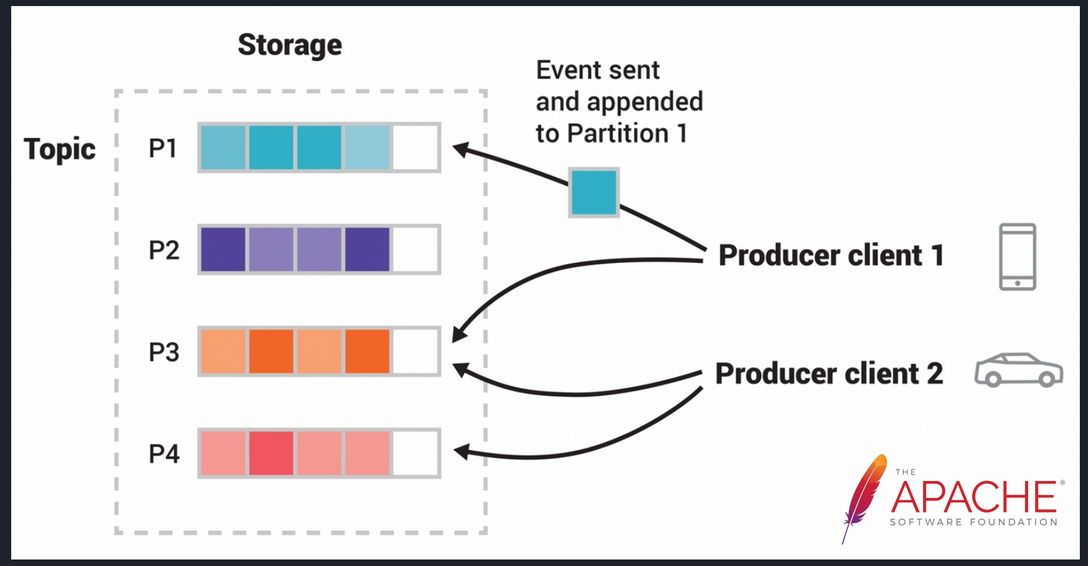
New event published to a topic

→ appended to one of corresponding partitions

Events with the same key get into the same partition

- data consistency
- efficient retrieval based on the key

Events sorted by time



# Functionality

REPLICA

Topics can be replicated across multiple brokers

→ after failure/maintenance of the broker, the data remains available

Every partition has exactly one **partition leader**

- handles all the read/write requests of that partition
- followers replicate the leader
- if leader fails - one follower becomes new leader
- leaders are balanced over cluster (leader for some partitions)

# Kafka APIs

**Language-independent protocol**

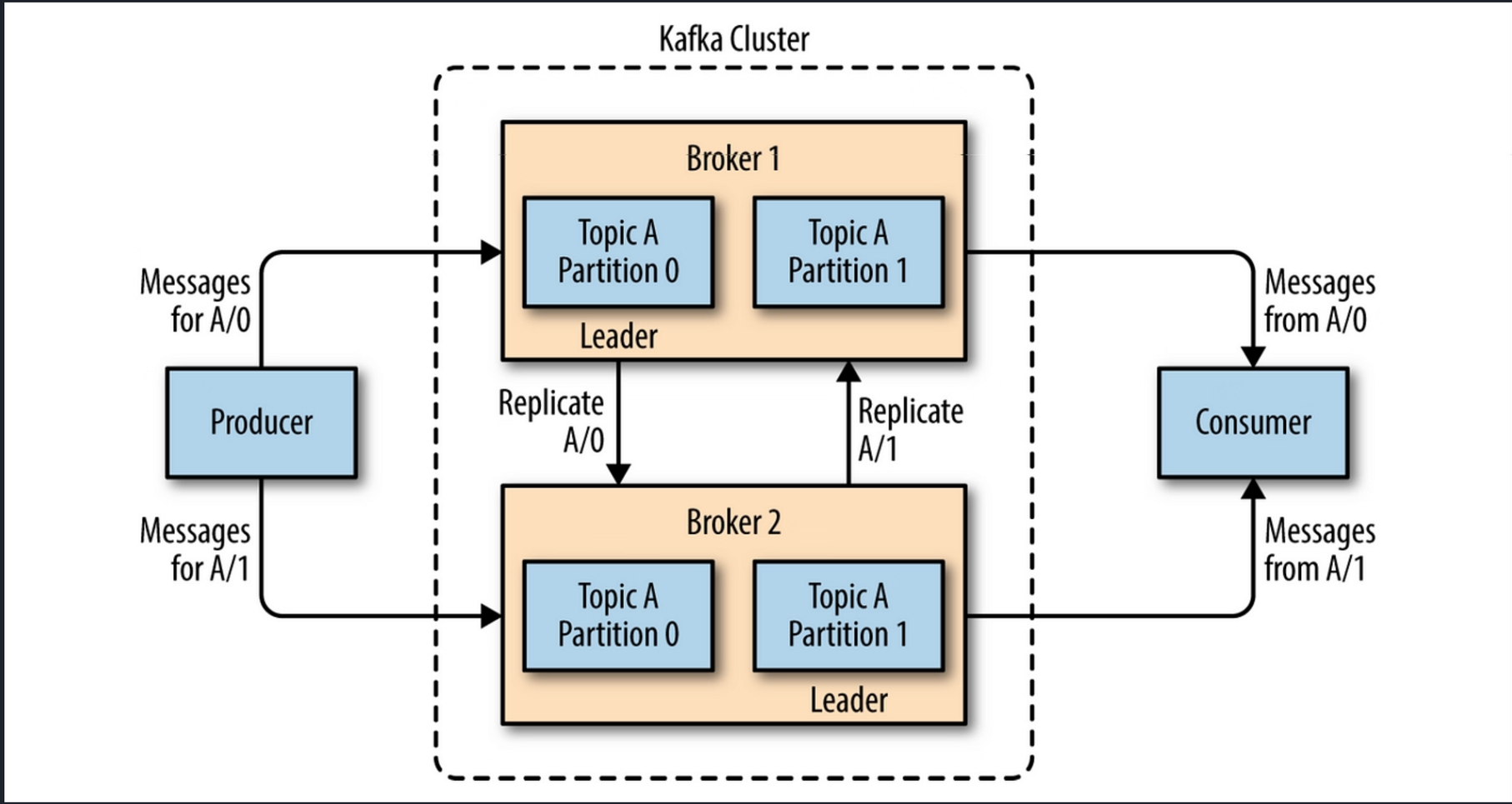
- Main Kafka project only maintains Java clients
- Others available as part of other independent open source projects

**5 core APIs**

- Producer API: send data streams to Kafka topics
- Consumer API: read data streams from Kafka topics
- Streams API: transform data streams from one topic to another
- Connect API: connectors to external systems
- Admin API: manage and monitor Kafka cluster components

# Architecture

BIG PICTURE



# Practical Example (1)

REAL-TIME ENERGY DATA STREAMING

**Smart Meter Adapter** of Österreichs Energie

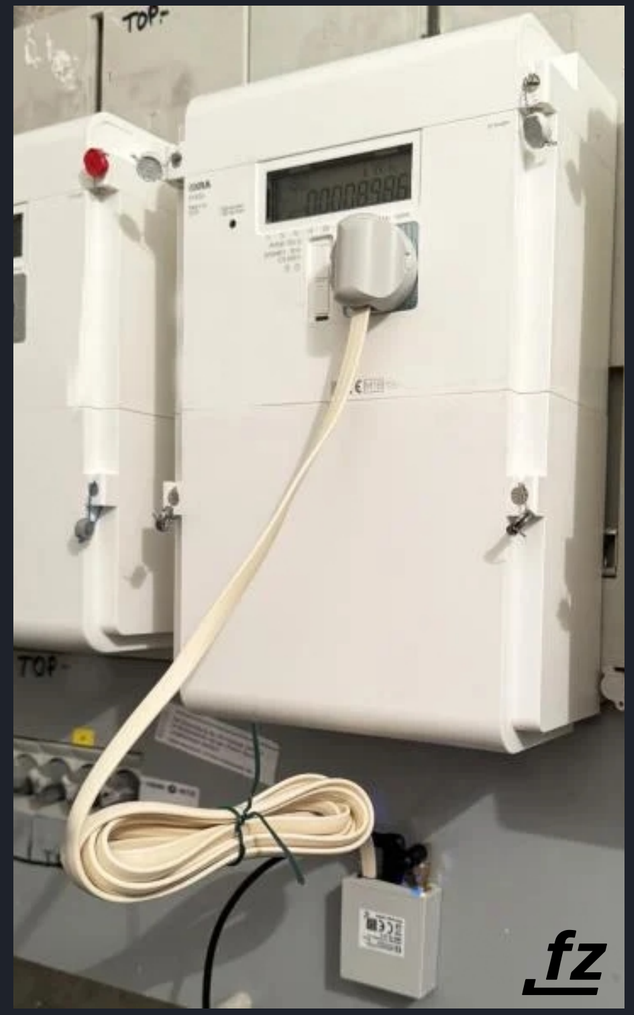
- read real-time energy measurements
- compatible with all smart meters on the Austrian market
- **Interfaces**: *MQTT*, REST-API and Modbus TCP

**Goal**

- local device subscribes via MQTT to measurements
- data is streamed via Kafka for further use in a large scaled application

Implementation in **Kotlin**

- Kafka clients are available in Java



# Practical Example (2)

SETUP MQTT

**Setup MQTT Broker**

Eclipse Mosquitto (open source MQTT broker)

0. Install Mosquitto
1. Create new config file

```
listener 1883
allow_anonymous true
```

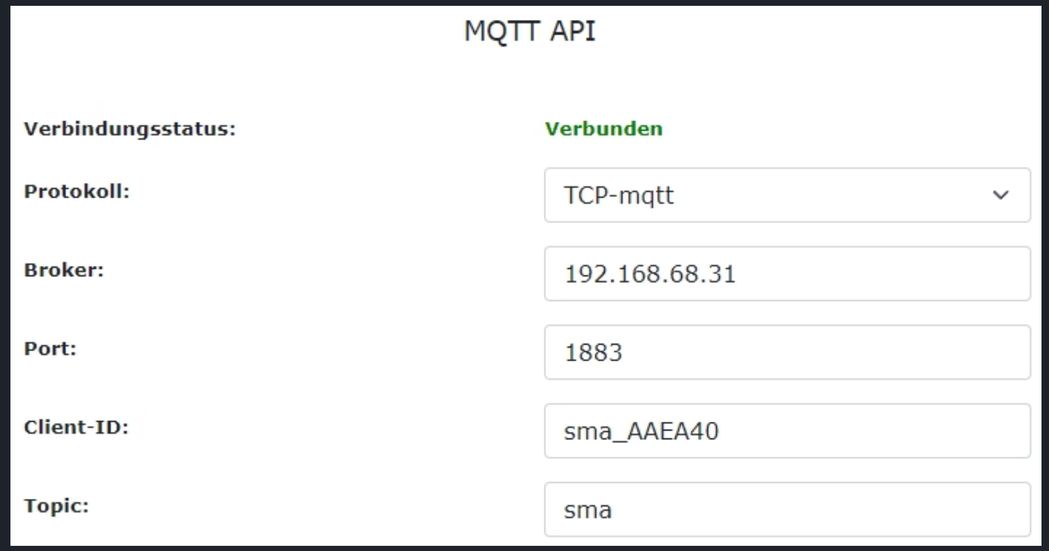
2. Start broker with created config file

```
$ mosquitto -c mosquitto.conf
```

**Connect Smart Meter Adapter**

Measurements published to MQTT broker

3. Setup MQTT connection on Adapter



# Practical Example (3)

SETUP KAFKA WITH KRAFT (Apache Zookeeper was used before KRaft)

0. Download Kafka
1. Generate a Cluster UUID

```
$ KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

2. Format Log Directories

```
$ bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
```

3. Start the Kafka Server

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

# Practical Example (4a)

KAFKA PRODUCER

*SmaMqttConsumer* subscribes to MQTT topic "sma"

- uses org.eclipse.paho.client.mqttv3

*SmaKafkaProducer* writes events to Kafka

- uses org.apache.kafka.clients.producer

```
val smaKafkaProducer = SmaKafkaProducer()
SmaMqttConsumer().subscribe {
    println("MQTT: $it")
    smaKafkaProducer.produce(it, it.toSmaMeasurement())
}
```

# Practical Example (4b)

KAFKA PRODUCER

```
class SmaKafkaProducer {
    private val producer: Producer<String, String>

    init {
        val properties = Properties().apply {
            setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, SmaKafkaConstants.BOOTSTRAP_SERVERS)
            setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer::class.java.name)
            setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer::class.java.name)
        }

        producer = KafkaProducer(properties)
    }

    fun produce(json: String, smaMeasurement: SmaMeasurement){
        val record = ProducerRecord(SmaKafkaConstants.SMA_TOPIC, smaMeasurement.name, json)
        producer.send(record)
    }
}
```

# Practical Example (5a)

KAFKA CONSUMER

```
class SmaKafkaConsumer {
    private val consumer: Consumer<String, String>

    init {
        val properties = Properties().apply {
            setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, SmaKafkaConstants.BOOTSTRAP_SERVERS)
            setProperty(ConsumerConfig.GROUP_ID_CONFIG, UUID.randomUUID().toString())
            setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer::class.java.name)
            setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer::class.java.name)
            setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest")
        }
        consumer = KafkaConsumer(properties)
    }
    …
```

# Practical Example (5b)

KAFKA CONSUMER

```
    …
    fun subscribe(onRecordArrived: (String?, String?, String?) → Unit){
        consumer.seekToBeginning(consumer.assignment());
        consumer.subscribe(listOf(SmaKafkaConstants.SMA_TOPIC))

        while (true) {
            val records = consumer.poll(Duration.ofSeconds(1))

            for (record in records) {
                onRecordArrived(record.topic(), record.key(), record.value())
            }
        }
    }
}
```

# Conclusion

DRAWBACKS?

Kafka offers **significant benefits**

- it's important to consider its limitations before adoption

**Drawbacks**

- Complexity
- Overhead
- Limited Management and Monitoring Tools

Real-time data processing with

# kafka

TOBIAS FISCHER

tformatix/saap-apache-kafka