Event-Driven Architecture with Apache Kafka

Ellmer Benjamin
Mobile Computing Master
University of Applied Sciences Upper Austria
Campus Hagenberg
St.Georgen/Gusen, Oberösterreich
benjamin.ellmer@yahoo.com

***Abstract*—The Event-Driven Architecture and Apache Kafka play an important role in the software development industry. Using them together leads to extremely flexible and highly scalable software architectures that are able to process massive loads of data with an incredible efficiency. Especially regarding Big Data and Internet of Things the Event-Driven Architecture tends to become a state-of-the-art pattern. Furthermore, the use of the Microservice pattern enhances the Event-Driven Architecture leading to fully decoupled and highly coherent services. In addition, using the Apache Kafka Message broker ensures mechanisms to achieve high performance, fault tolerance and much more.**

This paper will explain the key concepts of both technologies and show an example application, which demonstrates how good Apache Kafka and Event-Driven Architectures work together. The main focus of this paper is providing the necessary information that the readers of this paper are able to design an application using the Event-Driven Architecture and Apache Kafka. Furthermore, the readers should understand why and when to use this technology stack and for which situations it is more appropriate to use an alternative architecture or and alternative messaging system. Moreover, the example project should provide a good starting point for all kinds of applications. The use-cases provided in the Related Work section should inspire the readers to adapt this sample architecture depending on the specific requirements of their applications.

***Index Terms*—Event-Driven Architecture, Apache Kafka, Mircoservices, IoT, Big Data**

I. Introduction

Event-Driven Architectures are gaining significant attention in the recent years. They are providing a way to build scalable, resilient and very flexible systems that are able to handle high volumes of data [13]. The idea to build a system that reacts to events became established in the second half of the 80s [6]. With the high magnitudes of data in the 20th century this style of building applications became popular under the term Event-Driven Architectures. The use of event messages provides a way to fully decouple the functional units from each other. This decoupling of functional units that are commonly called services leads to significant benefits compared to traditional architectures like the monolithic architecture. It allows to independently deploy and scale services, increase the performance using asynchrony and isolate failures [13].

The most crucial part of an Event-Driven Architecture is the messaging system that is used to communicate the events to all components. One of the most popular choices for such a messaging system is Apache Kafka, which was originally developed by LinkedIn to address their needs for monitoring activity stream data and operational metrics. Now it is maintained by the Apache foundation, and it is used as messaging platform by many big players like LinkedIn, Twitter Foursquare and Datashift [7]. Because of its high throughput, low latency and fault tolerance it perfectly fits to the Event-Driven Architecture [2]. It is able to handle thousands of messages per second in a high-performance and efficient way.

All of these characteristics make the combination of the Event-Driven Architecture and Apache Kafka to a compelling technology stack. This paper explains the fundamentals of both technologies and provides a demonstrative example architecture to give an idea how the structure of an application could look like.

II. Related Work

Regarding the concrete architecture of an Event-Driven Architecture with Apache Kafka, there is no state-of-the-art structure, that can be used for every kind of application. The needed components and their interactions are tightly coupled to the use-case and the specified requirements.

Khriji et al. [9] have designed the so called REDA architecture which is focused on cost-effective measuring of data from wireless sensor nodes and storing this data in the cloud using Apache Kafka. In their paper they demonstrated a cheaply hosted architecture that guarantees a high data availability by handling about 8000 messages per second with very low latency. In contrast to the project of this paper, their architecture includes an MQTT (Message Queuing Telemetry Transport) broker, that collects and transfers the data to the Kafka Message Broker. Since their architecture focuses on IoT, where the devices oftenly have constrained resources it makes sense to add the MQTT Broker between the sensors and the Kafka Message Broker. Furthermore, Khriji et al. used Microservices developed in Java Spring Boot, same as the project, which is introduced in this paper. This once demonstrations how the Event-Driven Architecture with the Microservice Pattern has emerged to a compelling paradigm [9], [10].

The Event-Driven Architecture of Rocha et al. [14] is very similar to the architecture of the project, which is shown in this paper. The purpose of their system, is monitoring the energy consumption in a manufacturing ecosystem. In their implementation they have Java Programs, that collect the data from the machine and transfer the data to the Kafka Message

Broker. They decided for this structure without an intermediate program between the broker and the machine so the latency of the maintaining data flow keeps low. Furthermore, they have compared Apache Kafka with the message brokers Pulsar, NATS Streaming, RabbitMQ and Apache ActiveMQ. Regarding throughput, scalability, latency and integration, they came to the result that Apache Kafka is the best fitting broker for their project.

Because of the scalability, availability and the performance of Apache Kafka, it is oftenly used in the context of Big Data [10], [17]. Zhelev et al. [17] have researched the challenges, advantages and potential problems concerning big data stream processing. They came to the conclusion that Event-Driven Architectures with Microservices are the preferred technique for implementing modern scalable cloud solutions. Laigner et al. [10] agree with Zhelev et al. and therefore describe the steps of converting a Monolithic Big Data System to an Event-Driven Architecture with Microservices. By converting the legacy monolithic system into an Even-Driven Architecture with the Microservice pattern, they achieved easier maintenance and fault isolation. Furthermore, they were able to enable a reactive programming model with high-cohesive components. One of their main takeaways is, that defining Microservices too early tends to yield to wrong definitions of the service boundaries. Newman [12] and Fowler [6] have the same opinion and recommend to start with monolithic systems first and then break down the service into Microservices. Nevertheless, the project which is reviewed in this paper is very minimal and is only used for demonstrations, therefore it will embrace the Microservice Pattern from the beginning.

III. Event-Driven Architecture

The Even-Driven Architecture is a software architecture pattern, which is intended for highly scalable applications. It is highly adaptable, therefore it can be used for large applications and for small applications as well [13]. Mostly it is used to design applications with high performance requirements, that must be able to handle high loads of transactions. An Event-Driven Architecture consists of a number of highly decoupled and distributed components that asynchronously receive and produce events. A component that produces an event does not depend on the availability of the component that receives the event. This leads to the highest possible decoupling of the components that interact with each other [11].

A. Topologies

According to Richards [13] the Event-Driven Architecture consists of two topologies, the Mediator Topology and the Broker Topology.

The Mediator Topology is used, when events consist of multiple steps, that require some level of orchestration to process the events. The components of the Mediator Topology are event queues an event mediator, event channels and event processors. First the initial event is put into the event queue, from where the event mediator receives it. Then the event mediator step by step creates multiple processing events based on the initial event and sends them to the event channels. The event processors that contain the business logic receive the events from the event channels and handle the events until all processing events are processed.

The Broker Topology is used for simple events that do not need orchestration. The events are distributed across the event processor components in a chain like fashion. The Broker topology consists of two architectural components, the broker and the event processors. Instead of having a mediator that takes care of the orchestration, each event processor is responsible for processing and publishing events. The events are published and received by the event channels that are provided by the broker [13].

Nowadays, the Event-Driven Architecture is mostly associated with the Broker Topology and this paper will mainly focus on the Broker Topology using Apache Kafka as the message broker.

B. Concepts

Each application has different requirements and therefore requires a different architecture, but overall most Event-Driven Architectures follow these principles:

- **Loose coupling**: The event producers and the event consumers do not know from each other. Therefore, new producers or new consumers can be added or removed without changing other components [8].
- **Asynchrony**: The Event-Driven Architecture embraces an asynchronous communication style. An event can be processed by any order and by any time enabling the processing of large data volumes [13].
- **Resilience**: Events are queued and can be processed later making the architecture resilient for demand spikes. Furthermore, multiple instances of event processors can be deployed, therefore one failing component does not lead to a complete system failure [2].
- **Microservices**: The Event-Driven Architecture helps to prevent the code from getting too complex by producing single purpose components, which are oftenly called Microservices [8].

C. Motivations and Challenges

The reason why the Event-Driven Architecture became one of the most popular architecture patterns is because it brings lots of benefits compared to other architectures. The following reasons are some of the motivations that could justify to choose the Event-Driven Architecture over other architectures:

- Changes are isolated to one or a few event processors, which leads to a good ability to adapt to a changing environment [13].
- The components of the Architecture are easy to deploy, because of the decoupled nature of the components [13].
- The ability to work asynchronously and parallel outweighs the costs of queuing and dequeuing messages, which leads to a high performance [13].
- Each event processor can be scaled independently, which leads overall to a high scalability [13].

- Depending on the system the building and maintaining effort could be reduced by removing the persistence logic and re-running previous event, when the application has to restart [5].

Of course the Event-Driven Architecture does not only bring benefits with it, but it also leads to certain challenges. Those challenges have to be taken into consideration when choosing the optimal architecture for an application. The following reasons are some of the downsides that could argue not to use the Event-Driven Architecture:

- Testing of the interactions of the components becomes very hard because of the asynchronous nature of the pattern and because a specialized testing tool has to be used to generate events [13].
- The development becomes more complex because of the asynchronous nature and because of the additional error handling that is needed [13].
- The collaboration of the events is very implicit which makes it hard to debug and understand errors [5].
- Incorrect information is very hard to handle and require additional mechanisms or manual work [5].

IV. Apache Kafka

Apache Kafka is an open-source messaging system which focuses on persistent messaging, high throughput, multiple client support and real time processing. The communication in Apache Kafka is done using a publish-subscribe manner [7]. Furthermore, Apache Kafka provides sophisticated streaming capabilities and is able to deal with large amounts of data. The streams library is designed for processing and analyzing data that is stored in Apache Kafka in a more efficient way than using consumers [9].

To support all kinds of use cases Apache Kafka was built with the goal to achieve the following characteristics [2]:

- High throughput to support high volume event streams.
- Low latency delivery.
- Ability to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.
- Support partitioned distributed real-time processing of data feeds to create new derived feeds.
- Guarantee fault tolerance in case of machine failure.

A. Architectural Components and Concepts

Zookeeper is a centralized service for maintaining configuration information, providing distributed synchronization and providing group services [16]. Apache Kafka uses the Zookeeper services and therefore a Zookeeper instance is necessary to run a Kafka **server** [14]. A Kafka **cluster** can be one or multiple servers and the servers that form the storage layer are called **brokers**. When one of multiple servers fails, another server takes over the work to ensure continuous operations without any data loss. Data is transferred between the clients using **messages**, which are also called **events** or **records**. A message consists of a key, a value a timestamp and optional metadata. Messages are organized using **topics**, which are used to durably store events with familiar categories. **Producers** publish events to topics and Apache Kafka takes care to deliver the events to the **consumers** that subscribed to certain topics. Topics can be supplied and consumed by multiple producers and multiple consumers [2]. Furthermore, Apache Kafka provides the possibility to define **consumer groups**. Messages are then transferred to only one consumer of the consumer group [16]. The topics are separated into **partitions** which can be located on different Kafka Brokers. The partition is decided based on the key of the message and messages with the same key will always be stored in the same partition. Regarding fault-tolerance and availability, topic partitions can be replicated among geo-regions or datacenters so that always multiple brokers have a copy of the data [2].

B. Consumption in Depth

When it comes to messaging systems one of the most important questions is how to inform the consumers about new data. In general two approaches exist to tackle this question, push and pull. With a pull system the consumers always have to ask somebody if new data exists. With a push system a function of the consumers gets called every time when new data is available. Apache Kafka decided to use a pull system, where the producers push the data to the broker and the consumers pull it from the broker. This approach has the advantage that the consumers can control the data rate and when the data rate gets higher than the maximum of the consumer he just falls behind [2].

Using this approach it is essential to keep track of what has already been consumed by a consumer or a consumer group. The common technique for this problem is using acknowledgements and saving a flag whether a message was only sent or also consumed. This mechanism is harmful to the performance because the broker has to save a state for each message. Furthermore, it leads to multiple message consumption when the consumer fails after processing a message but before sending the acknowledgement. Therefore, Apache Kafka uses message offsets to handle this problem. The message offset is just a single integer that stores the offset of the next message that has to be consumed. This offset is stored for each partition and a partition is always consumed by only one consumer of a consumer group. Using this approach Apache Kafka has a very cheap equivalent to acknowledgements, and additionally it provides the possibility to re-consume messages in certain cases [2].

C. Message Delivery

The Kafka Message Broker has three different configurations regarding the message delivery [2].

- At most once - Messages may be lost but are never redelivered.
- At least once - Messages are never lost but may be redelivered.
- Exactly once - Each message is delivered once and only once

It uses an approach where messages that are published are being committed to the log. After committing a message, it can

not be lost as long as the broker who replicates the partition of the message keeps alive. A case where the producer fails to receive the response indicating that a message was committed has three different solutions regarding to the configuration. When the configuration is set to at most once, the producer will not resend the message, and will not know if it was delivered successfully or not. When the configuration is set to at least once, the producer will resend the message until he gets a response that the message was successfully committed to the log. When the configuration is set to exactly once, the broker assigns each producer an ID and deduplicates the messages using a sequence number that is sent by the producer along with every message [2]. This shows the adapdability of the Apache Kafka, making it configurable for all kinds of applications, regardless whether it is a huge IoT application where multiple message deliveries are no problem, or whether it is a critical financial application where multiple message deliveries result in critical problems.

D. Alternatives

Apache Kafka is one of the most used messaging systems, still it is not perfect suited for all cases. This section includes the summarized comparisons of the most popular alternatives to Apache Kafka.

Apache Pulsar is an open-source distributed pub/sub messaging system which is built on top of BookKeeper, a distributed storage system [4]. Pulsar has very similar throughputs to Apache Kafka but when the number of publishers and subscribers increases or when the size of messages increases, it can not scale its output as well as Apache Kafka. Furthermore, it needs Apache Zookeeper and Apache BookKeeper while Apache Kafka needs only Zookeeper [14].

NATS Streaming is a lightweight open-source cloud-based messaging system that supports publisher-subscriber, request-reply and message queue models [15]. NATS Streaming has very similar throughputs to Apache Kafka, but it has a severe problem with latency because NATS Streaming always needs to connect to the server before any action [14]. Furthermore, scaling is not well-supported, but it can be achieved using mechanisms provided by the Go language, that was used to implement NATS Streaming [15].

RabbitMQ is an open-source traditional messaging middleware that implements the Advanced Message Queuing Protocol [4]. RabbitMQ has very low latency for low throughput, but when the number of publishers and subscribers is increased, and more information is sent the throughput cannot keep up. Therefore, RabbitMQ is more suited for low scale applications but not suitable on a bigger scale like Apache Kafka [14].

Overall it is valid to say, that Apache Kafka is superior to all the presented alternatives regarding performance. It has the best throughput and offers the lowest end-to-end latencies. Furthermore, Apache Kafka provides the best value of the compared systems regarding costs per written byte [4]. Still there are use cases where Apache Kafka might not be the best solution, especially when features are needed that are not implemented by Apache Kafka [15].

V. Sample Architecture

This section describes and shows the architecture of a sample application, which is used to demonstrate the use of the Event-Driven Architecture and Apache Kafka. The architecture of this sample application can not be simply reused for each kind of application. Instead, it should inspire to adapt parts, depending on the requirements of the specific use cases. The application currently consists of six components:

The **Edge Device Simulator** is a Java Application that simulates data produced by production machines which are called Equipments. An Edge Device Simulator simulates multiple Edge Devices and an Edge Device has one to multiple Equipments connected. The simulation program sends the production status of the machines in a configurable time interval to the Kafka Message Broker. Each instance of an Edge Device Simulator is a Producer with a unique Producer-ID. Depending on the use case lots of production properties can be published, but for the sake of simplicity the Equipment Status only includes the fields *jobPercentFinished*, *producing* and a *timestamp*.

The **Kafka Message Broker** is the messaging system of the application. It currently has only one topic the *EquipmentStatus* topic, that contains the data produced by the running Edge Device Simulators. The *EquipmentStatus* topic is configured to have two partitions, for the case that another Kafka broker instance is added or for the case that consumer groups with multiple consumers exist. The retention period of the topic is set to the default value, which is 168 hours (one week). This means the events will disappear after one week and can then not be consumed by the services anymore.

The **Timescale Service** is the service that is responsible to store the Equipment Status data into the Timescale Database. The data is consumed using the Apache Kafka Streams API, and then it is persisted using Spring Data JPA (Java Persistence API). The service was built with Java Spring Boot, because the Spring framework provides an easy integration of Kafka streams and JPA. Furthermore, the Spring framework provides convenient deployment capabilities using docker. For larger volumes of data it would be possible to run multiple instances of the Timescale Service. As long as the services are using the same consumer group each instance would get its own partition(s) assigned, and the data would not be stored multiple times.

The **Timescale Database** is used to persistently store all entries in the *equipment_status* table. From the database the long term trends and historical data can be derived. TimescaleDB is a time-series database built on top of PostgreSQL. More specifically timescale is an extension for the PostgreSQL database that improves the regular PostgreSQL database in three key areas: Better performance at scale, lower storage costs and features that speed up the development time [3]. Especially the better performance at scale perfectly fits to the technology stack which is presented in this paper.

Grafana is used to visualize the data, which is stored in the Timescale Database. The Grafana open source software is

a tool that enables querying, analyzing, visualizing, exploring metrics, logs and traces of data wherever it is stored [1]. The reason why Grafana was chosen for this application, is because it can easily integrate data from a Timescale Database and provides an easy but sophisticated way to visualize the data.

The **Telegram Monitoring Service** is a Spring Boot service that hosts a Telegram Monitoring Bot for Equipment Status updates. Using the Telegram Bot, it is possible to subscribe and unsubscribe to status updates of certain Equipments. The reason why Telegram was chosen is, because it provides a simple but yet powerful API for Java Spring Boot. Same as the Timescale Service the Telegram Monitoring Service consumes the data using the Apache Kafka Streams API.

One might argue that it would make sense to merge the Telegram Monitoring Service and the Timescale Service to one service to prevent duplication of code. But this approach would couple two services that are meant to do completely different tasks. It would break the ability to independently scale the services or run multiple instances of the service, because only one instance of a Telegram Bot can be running at a time. Furthermore, changes to the Timescale Service would require to redeploy the logic of the Telegram Monitoring Service vice versa.

A diagram of the architecture of this application is shown in Figure 1. The diagram shows the previously described components and their interactions. This architecture should provide a good starting point for highly scalable applications. Of course the underlying technologies like Java Spring Boot, TimescaleDB or Telegram can easily be substituted by other technologies depending on the requirements.

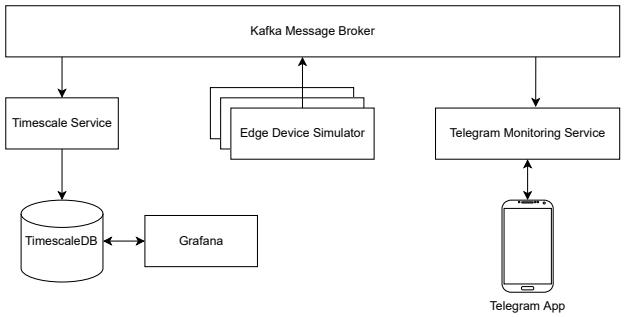

Fig. 1. Architecture of the Sample Application.

VI. Discussion

This paper explained the fundamentals of the Event-Driven Architecture and Apache Kafka. The focus was showing how good this compelling technologies fit together and enable to build highly scalable and high throughput architectures. Additionally, alternative message broker systems were compared to Apache Kafka. An important question that could not be answered in this paper is, when it makes sense to choose an alternative messaging system instead of Apache Kafka. For smaller applications with only few components alternative messaging systems like RabbitMQ are sufficient. But where is the sweet spot?

Furthermore, the Event-Driven Architecture might be too much overhead for some applications. Especially regarding very small applications the expense of deploying all components including multiple services and a messaging system might exceed the benefits.

VII. Conclusion

The motivations, challenges and concepts presented in this paper highlighted that Apache Kafka is a powerful tool for building high scale applications using the Event-Driven Architecture. The ability to handle high volumes of data, the fault tolerance the low latency and the support for real-time processing make it an ideal technology stack for many use cases. Even if there are other messaging systems that can be used for Event-Driven Architectures, the unique combination of Apache Kafka's features and its growing ecosystem make it an attractive opinion for many organizations.

References

[1] Grafanadocs. https://grafana.com/docs/grafana/latest/introduction/.

[2] Kafka 3.3 documentation. https://kafka.apache.org/33/documentation.html.

[3] Timescaledocs. https://docs.timescale.com/timescaledb/latest.

[4] Which is the fastest? https://www.confluent.io/blog/kafka-fastest-messaging-system/.

[5] Martin Fowler. Focusing on events. https://martinfowler.com/eaaDev/EventNarrative.html, 2006-01-25.

[6] Martin Fowler. Miroservice premium. https://martinfowler.com/bliki/MicroservicePremium.html, May 2015.

[7] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[8] Shelton Graves. Event-driven on azure: Part 1 – why you should consider an event-driven architecture. https://techcommunity.microsoft.com/t5/apps-on-azure-blog/event-driven-on-azure-part-1-why-you-should-consider-an-event/ba-p/2106983, 2021-01-28.

[9] Sabrine Khriji, Yahia Benbelgacem, Rym Chéour, Dhouha El Houssaini, and Olfa Kanoun. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *The Journal of Supercomputing*, 78(3):3374–3401, 2022.

[10] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. From a monolithic big data system to a microservices event-driven architecture. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 213–220. IEEE, 2020.

[11] Behrooz Malekzadeh. Event-driven architecture and soa in collaboration-a study of how event-driven architecture (eda) interacts and functions within service-oriented architecture (soa). Master's thesis, 2010.

[12] Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2021.

[13] Mark Richards. *Software architecture patterns*, volume 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ..., 2015.

[14] Andre Dionisio Rocha, Nelson Freitas, Duarte Alemão, Magno Guedes, Renato Martins, and José Barata. Event-driven interoperable manufacturing ecosystem for energy consumption monitoring. *Energies*, 14(12):3620, 2021.

[15] T Sharvari and K Sowmya Nag. A study on modern messaging systems-kafka, rabbitmq and nats streaming. *CoRR abs/1912.03715*, 2019.

[16] Khin Me Me Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[17] Svetoslav Zhelev and Anna Rozeva. Using microservices and event driven architecture for big data stream processing. In *AIP Conference Proceedings*, volume 2172, page 090010. AIP Publishing LLC, 2019.