International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 06 | June 2020 www.irjet.net p-ISSN: 2395-0072

BUILDING AN EVENT-DRIVEN MESSAGING BROKER

Gowthami T P[1], Dr. Deepamala N[2]

[1]Department of Computer Science and Engineering, R V College of Engineering, Bengaluru, Karnataka, India
[2]Assistant Professor, Department of Computer Science and Engineering, R V College of Engineering, Bengaluru, Karnataka, India

---

Abstract – *In traditional message processing, a component creates a message then sends it to specific destination. The receiver, which has been sitting idle and waiting, receives the message and acts accordingly. When the message arrives, the receiver performs a single process, and then deletes the message. But message broker use a store and forward system where events travel from broker to another broker until they reach the specific consumer. Message broker eliminate the inefficiencies linked with the traditional polling based communication mechanism and make the process of data exchange simple and reliable. The event-driven architecture consists of two main topologies, the mediator and the broker. In this paper the difference between those two topologies, their drawbacks and advantages are enhanced.*

1. INTRODUCTION

Nowadays, about 12 billion smart machines are connected to the Internet. Considering about 7 billion people on the planet, we have almost one-and-a-half device per person. With technological development, our everyday life becomes more and more digitized. As a result of this digitization, software developers face the problem of successful data exchange. To overcome that, we have special message brokers. They make the process of data exchange simple. We use message broker in event-driven applications because.

1. To control the data feeds to the system For example, the number of registrations in any system.
2. To put data to several applications and avoid direct usage of their API.
3. To implement time-bound request/reply interface

In event-driven architecture, when a service performs some piece of work that other services might be interested in, that service produces an *event*-a record of the performed action. Other services consume those events so that they can perform any of their own tasks needed as a result of the event. Unlike with REST, services that create requests do not need to know the details of the services consuming the requests.

Events can be published in a variety of ways. For example, they can be published to a queue that guarantees delivery of the event to the appropriate consumers, or they can be published to a pub/sub model stream that publishes the event and allows access to all interested parties.

Uses of event-driven architecture is,

- **Asynchronous** – event-based architectures are asynchronous without blocking.
- **Loose Coupling** – services don't need knowledge of, or dependencies on other services.
- **Easy Scaling** – Since the services are decoupled under an event-driven architecture, and as services typically perform only one task, tracking down bottlenecks to a specific service, and scaling that service becomes easy.
- **Recovery support** – An event-driven architecture with a queue can recover lost work by "replaying" events from the past.

2. Types of Message Broker

2.1.1 Message Broker RabbitMQ:

RabbitMQ is known as a traditional message broker written in the Erlang, which is suitable for a wide range of projects. It is successfully used both for development of new startups and notable enterprises. RabbitMQ perfectly works with Java, Spring, .NET, PHP, Python, Ruby, JavaScript, Go, Elixir, Objective-C, Swift and many other technologies. The numerous plugins and libraries are the main advantage of the software.

Advantages :

- Suitable for many programming languages and messaging protocols.
- Modern in-built user interface.
- Can be used on different operating systems and cloud environments.
- Scales to around 500,000+ messages per second.
- Gives an opportunity to use various developer tools.

Disadvantages :

- Needs Erlang
- Minimal configuration that can be done through code
- Issues with processing big amounts of data

2.1.2 Message Broker Apache Kafka :

Kafka is a powerful event streaming platform capable of handling trillions of messages a day. Kafka is useful both for storing and processing historical data from the past and for real-time work. With the help of Apache Kafka, you can successfully create event-driven applications and manage complicated back-end systems.

The Apache Kafka has become popular largely due to its compatibility. We can use Apache Kafka with a wide range of systems. They are:

- web and desktop custom applications
- microservices, monitoring and analytical systems
- any needed sinks or sources

Advantages :

- Fault-tolerance and reliable solution.
- Suitable for real-time processing.
- Powerful event streaming platform.
- Good scalability and Multi-tenancy.

Disadvantages:

- Lack of ready to use elements.
- The absence of complete monitoring set.
- Dependency on Apache Zookeeper.

3. Methodology

The event-driven architecture pattern consists of two main topologies, the mediator and the broker. The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator, Because the architecture characteristics and implementation strategies differ between these two topologies.

1. Mediator topology :

The mediator topology is useful for events that have multiple steps and require some level of orchestration to process the event. There are three main types of architecture components within the mediator topology:

I. **Event queues** - The event flow starts with a client sending an event to an *event queue*, which is used to transport the event to the event mediator.

II. **Event mediator** - The event-mediator component is responsible for orchestrating the steps contained within the initial event. For each step in the initial event, the event mediator sends out a specific processing event to an event channel, which is then received and processed by the event processor. It is important to note that the event mediator doesn't actually perform the business logic necessary to process the initial event; rather, it knows of the steps required to process the initial event.

III. **Event processors** - The event processor components contain the application business logic necessary to process the processing event. Event processors are self-contained, independent, highly decoupled architecture components that perform a specific task in the application or system.

2. Broker Topology

The broker topology differs from the mediator topology in that there is no central event mediator; rather, the message flow is distributed across the event processor components in a chain-like fashion through a lightweight message broker. This topology is useful when you do not want central event orchestration.

There are two main types of architecture components within the broker topology: a ***broker* component** and an ***event processor* component**. The broker component can be centralized or federated and contains all of the event channels that are used within the event flow. The event channels contained within the broker component can be message queues, message topics, or a combination of both. This is shown in the figure-1.

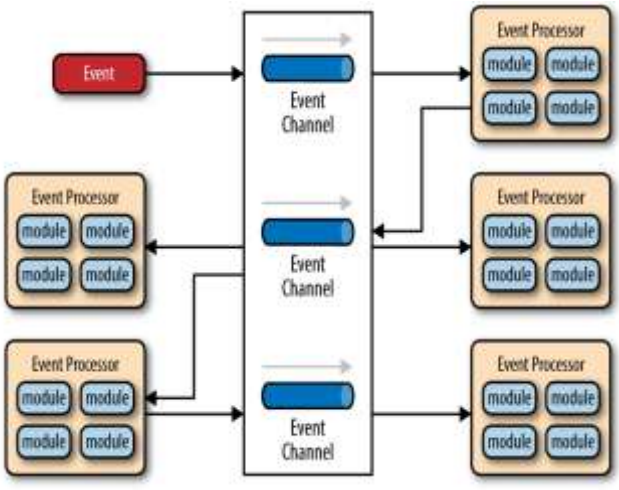

Fig 1 : Event-driven broker topology

As we can see from the diagram, there is no central event-mediator component controlling and orchestrating the initial event; rather, each event-processor component is responsible for processing an event and publishing a new event indicating the action it just performed.

For example, an event processor that balances a portfolio of stocks may receive an initial event called stock split. Based on that initial event, the event processor may do some portfolio rebalancing, and then publish a new event to the broker called rebalance portfolio, which would then be picked up by a different event processor. Note that there may be times when an event is published by an event processor but not picked up by any another event processor. This is common when you are evolving an application or providing for future functionality and extensions.

4. Consideration and Analysis

One consideration to take into account when choosing the architecture pattern is the lack of atomic transactions for a single business process. Because event processor components are highly decoupled and distributed, it is very difficult to maintain a transactional unit of work across them. For this reason, when designing our application using the pattern, we must continuously think about which events can and can't run independently and plan the granularity of our event processors accordingly.

Perhaps one of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and governance of the event-processor component contracts. Analysis of the common architecture characteristics for the event-driven architecture patterns are

1. Overall agility - Ability to respond quickly to a constantly changing environment.

2. Performance - The pattern should achieves high performance through its asynchronous capabilities; in other words, the ability to perform decoupled, parallel asynchronous operations outweighs the cost of queuing and dequeuing messages.

3. Scalability - Each event processor can be scaled separately, allowing for fine-grained scalability.

4. Ease of development - Need for more advanced error handling conditions within the code for unresponsive event processors and failed brokers.

While implementing the pattern, we must address various distributed architecture issues, such as remote process availability, lack of responsiveness, and broker reconnection logic in the event of a broker or mediator failure.

It is vitally important when using this pattern to settle on a standard data format (e.g., XML, JSON, Java Object, etc.) and establish a contract versioning policy right from the start.

5. CONCLUSIONS

In this paper we studied about different brokers and topologies used to implement the event-driven messaging broker. There are two messaging pattern

- Queuing
- pub/sub.

Both of them have some pros and cons. The advantage of the first pattern is the opportunity to easily scale the processing. On the other hand, queues aren't multi-subscriber. The second model provides the possibility to broadcast data to multiple consumer groups.

Being a broker-centric program, RabbitMQ gives guarantees between producers and consumers. If we choose this software, we should use transient messages, rather than durable. But Apache Kafka combines those two patterns, getting benefits of both of them. Implementing the message broker with the pub/sub in queuing Kafka is the more efficient broker for the event-driven large projects.

6. REFERENCES

[1] F. Yang, X. Ye, and Y. Zhang, ''DZMQ: A decentralized distributed messaging system for realtime Web applications and services,'' in Proc. 11th IEEE Web Inf. Syst. Appl. Conf. (WISA), Sep. 2014, pp. 165–171.

[2] Z. Wang, W. Dai, and F. Wang, ''Kafka and its using in high-throughput and reliable message distribution,'' in Proc. 8th Int. Conf. Intell. Netw. Intell. Syst. (ICINIS), 2015, pp. 117–120.

[3] S. Vinoski, ''Advanced message queuing protocol,'' IEEE Internet Comput., vol. 10, no. 6, pp. 87–88, Jun. 2006.

[4] M. Rostanski, K. Grochla, and A. Seman, ''Evaluation of highly available and fault tolerant middleware clustered architectures using RabbitMQ,'' in Proc. Federated Conf. IEEE Comput. Sci. Inf. Syst. (FedCSIS), Sep. 2014, pp. 879–884.

[5] J. Kreps, N. Narkhede, and J. J. Rao, ''Kafka: A distributed messaging system for log processing,'' in Proc. NetDB, 2011, pp. 1–7.

[6] J. Gascon-Samson, F. Garcia, B. Kemme, and J. Kienzle, ''Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud,'' in Proc. 35th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS), Jun. 2015, pp. 486–496.