Event-Driven Architecture

The event-driven architecture pattern is a popular distributed asynchronous architecture pattern used to produce highly scalable applications. It is also highly adaptable and can be used for small applications and as well as large, complex ones. The event-driven architecture is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.

1. Topology

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, it is important to understand each one to know which is best suited for your particular situation.

1.1. Mediator Topology

The mediator topology is useful for events that have multiple steps and require some level of orchestration to process the event. For example, a single event to place a stock trade might require you to first validate the trade, then check the compliance of that stock trade against various compliance rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker. All of these steps would require some level of orchestration to determine the order of the steps and which ones can be done serially and in parallel.

1.1.1. Components

There are four main types of architecture components within the mediator topology: event queues, an event mediator, event channels, and event processors. 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. The *event mediator* receives the initial event and orchestrates that event by sending additional asynchronous events to *event channels* to execute each step of the process. *Event processors*, which listen on the event channels, receive the event from the event mediator and execute specific business logic to process the event. Below figure (*Figure 1: Event-driven architecture mediator topology*) illustrates the general mediator topology of the event-driven architecture pattern.

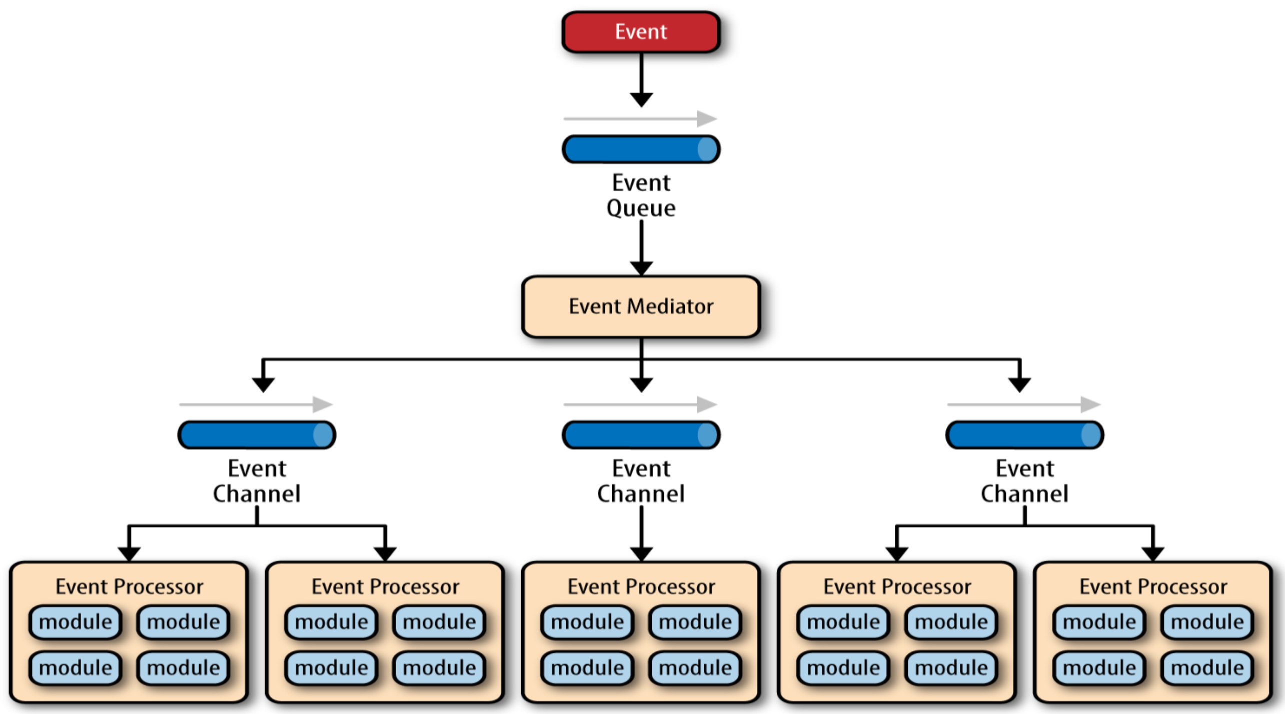

Figure 1: Event-driven architecture mediator topology

There are two types of events within this pattern: an *initial event* and a *processing event*. The initial event is the original event received by the mediator, whereas the processing events are ones that are generated by the mediator and received by the event-processing components.

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.

Event channels are used by the event mediator to asynchronously pass specific processing events related to each step in the initial event to the event processors. The event channels can be either message queues or message topics, although message topics are most widely used with the mediator topology so that processing events can be processed by multiple event processors (each performing a different task based on the processing event received).

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. While the granularity of the event-processor component can vary from fine-grained (e.g., calculate sales tax on an order) to coarse-grained (e.g., process an insurance claim), it is important to keep in mind that in general, each event-processor component should perform a single business task and not rely on other event processors to complete its specific task.

1.1.2. Implementation

It is common to have anywhere from a dozen to several hundred event queues in an event-driven architecture. The pattern does not specify the implementation of the event queue component; it can be a message queue, a web service endpoint, or any combination thereof.

The event mediator can be implemented in a variety of ways. As an architect, you should understand each of these implementation options to ensure that the solution you choose for the event mediator matches your needs and requirements.

The simplest and most common implementation of the event mediator is through open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB. Event flows in these open source integration hubs are typically implemented through Java code or a DSL (domain-specific language). For more sophisticated mediation and orchestration, you can use BPEL (business process execution language) coupled with a BPEL engine such as the open source Apache ODE. BPEL is a standard XML-like language that describes the data and steps required for processing an initial event. For very large applications requiring much more sophisticated orchestration (including steps involving human interactions), you can implement the event mediator using a business process manager (BPM) such as jBPM.

1.1.3. Example

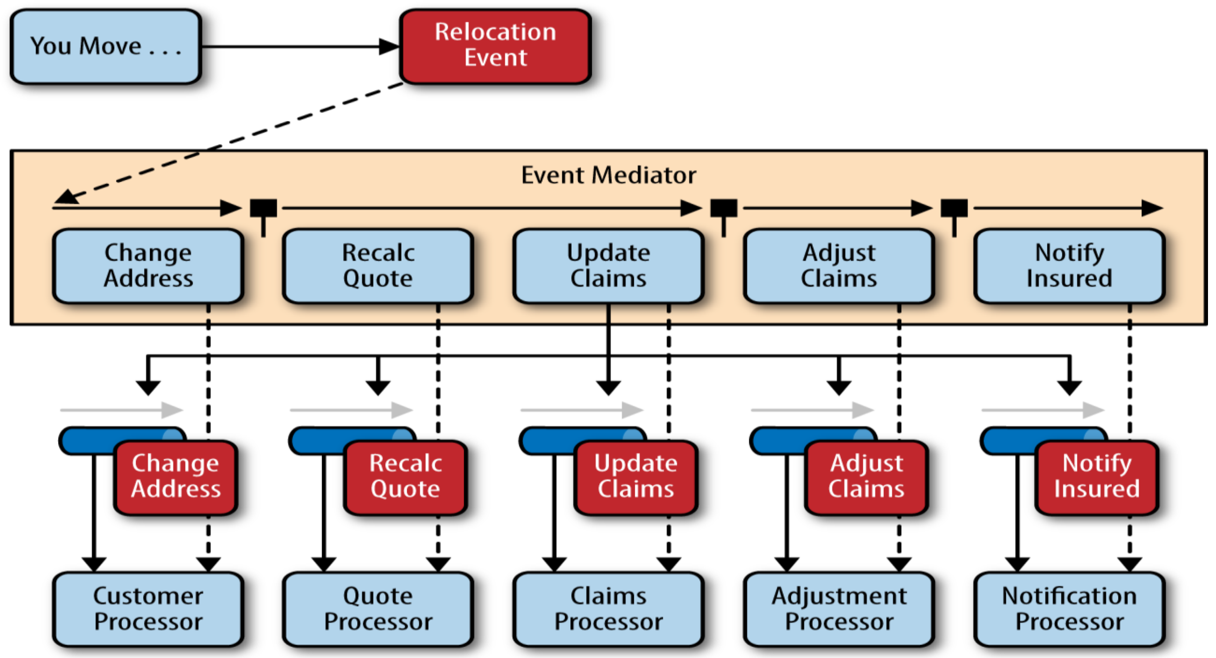

Figure 2: Mediator topology example

To illustrate how the mediator topology works, suppose you are insured through an insurance company and you decide to move. In this case, the initial event might be called something like *relocation event*. The steps involved in processing a *relocation event* are contained within the event mediator as shown in above figure (*Figure 2: Mediator topology example*). For each initial event step, the event mediator creates a processing event (e.g., *change address*, *recalc quote*, etc.), sends that processing event to the event channel and waits for the processing event to be processed by the corresponding event processor (e.g., customer process, quote process, etc.). This process continues until all of the steps in the initial event have been processed. The single bar over the recalc quote and update claims steps in the event mediator indicates that these steps can be run at the same time.

1.2. 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 (e.g., ActiveMQ, HornetQ, etc.). This topology is useful when you have a relatively simple event processing flow and you do not want (or need) central event orchestration.

1.2.1. Components

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 – refer below figure (*Figure 3: Event-driven architecture broker topology*)

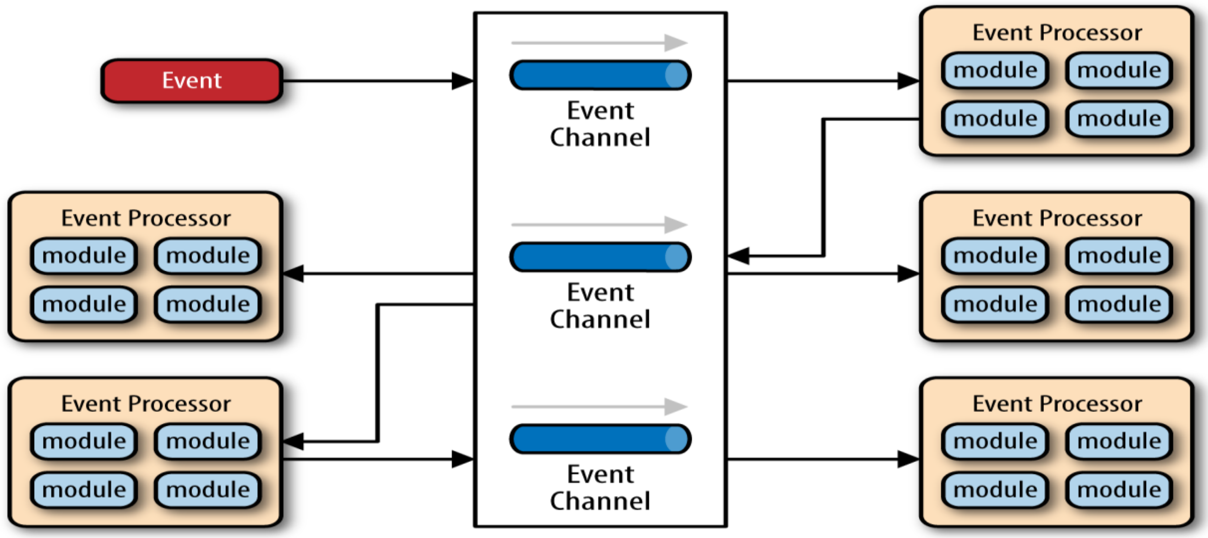

Figure 3: Event-driven architecture broker topology

1.2.2. Example

To illustrate how the broker topology works, we'll use the same example as in the mediator topology (an insured person moves). Since there is no central event mediator to receive the initial event in the broker topology, the customer-process component receives the event directly, changes the customer address, and sends out an event saying it changed a customer's address (e.g., *change address* event*)*. In this example, there are two event processors that are interested in the *change address* event: the quote process and the claims process. The quote processor component recalculates the new auto-insurance rates based on the address change and publishes an event to the rest of the system indicating what it did (e.g., *recalc quote* event). The claims processing component, on the other hand, receives the same *change address* event, but in this case, it updates an outstanding insurance claim and publishes an event to the system as an *update claim* event. These new events are then picked up by other event processor components, and the event chain continues through the system until there are no more events are published for that particular initiating event.

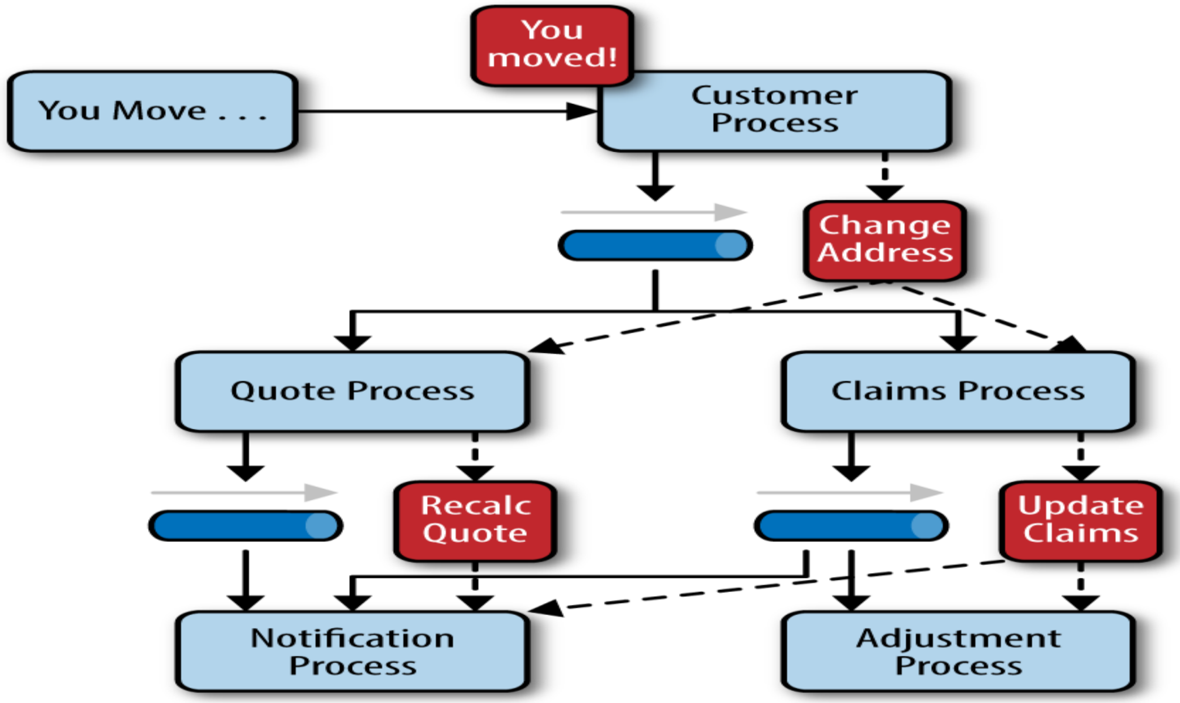

Figure 4: Broker topology example

As you can see from above figure (*Figure 4: Broker topology example*), the broker topology is all about the chaining of events to perform a business function. The best way to understand the broker topology is to think about it as a relay race. In a relay race, runners hold a baton and run for a certain distance, then hand off the baton to the next runner, and so on down the chain until the last runner crosses the finish line. In relay races, once a runner hands off the baton, she is done with the race. This is also true with the broker topology: once an event processor hands off the event, it is no longer involved with the processing of that specific event.

1.2.3. Implementation

For broker component, lightweight message brokers like ActiveMQ, HornetQ etc can be used.

2. Considerations

The event-driven architecture pattern is a relatively complex pattern to implement, primarily due to its asynchronous distributed nature. When implementing this pattern, you 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.

One consideration to take into account when choosing this 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 your application using this pattern, you must continuously think about which events can and can't run independently and plan the granularity of your event processors accordingly. If you find that you need to split a single unit of work across event processors—that is, if you are using separate processors for something that should be an undivided transaction—this is probably not the right pattern for your application.

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. Each event usually has a specific contract associated with it (e.g., the data values and data format being passed to the event processor). 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.

3. Pattern Analysis

The following table contains a rating and analysis of the common architecture characteristics for the event-driven architecture pattern. The rating for each characteristic is based on the natural tendency for that characteristic as a capability based on a typical implementation of the pattern, as well as what the pattern is generally known for.

1.3. Overall agility

Rating: High
Analysis: Overall agility is the ability to respond quickly to a constantly changing environment. Since event-processor components are single-purpose and completely decoupled from other event processor components, changes are generally isolated to one or a few event processors and can be made quickly without impacting other components.

1.4. Ease of deployment

Rating: High
Analysis: Overall this pattern is relatively easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change.

1.5. Testability

Rating: Low
Analysis: While individual unit testing is not overly difficult, it does require some sort of specialized testing client or testing tool to generate events. Testing is also complicated by the asynchronous nature of this pattern.

1.6. Performance

Rating: High
Analysis: While it is certainly possible to implement an event-driven architecture that does not perform well due to all the messaging infrastructure involved, in general, the pattern 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.

1.7. Scalability

Rating: High
Analysis: Scalability is naturally achieved in this pattern through highly independent and decoupled event processors. Each event processor can be scaled separately, allowing for fine-grained scalability.

1.8. Ease of development

Rating: Low
Analysis: Development can be somewhat complicated due to the asynchronous nature of the pattern as well as contract creation and the need for more advanced error handling conditions within the code for unresponsive event processors and failed brokers.