Event-Driven Architecture

April 4, 2022

Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Event-Driven Architecture

Software Architecture

April 4, 2022

Richard Thomas

1 Introduction

Event-driven is an asynchronous architectural style. It reacts to events, which is different to the common procedural flow of control of many designs where messages are sent as requests. It is a distributed event handling system, which is conceptually similar to event handling used in many graphical user interface libraries.

Events provide a mechanism to manage asynchronous communication. An event is sent to be handled, and the sender can continue with other tasks while the event is being processed. If necessary, an event handler can send an asynchronous message back to an event initiator indicating the result of the event processing.

Each event handler can be implemented in its own independent execution environment. This allows each type of handler to be easily scaled to handle its load. Asynchronous communication means that event generators do not need to wait for the handler to process the event. Handlers can generate their own events to indicate what they have done. These events can be used to coordinate steps in a complex business process.

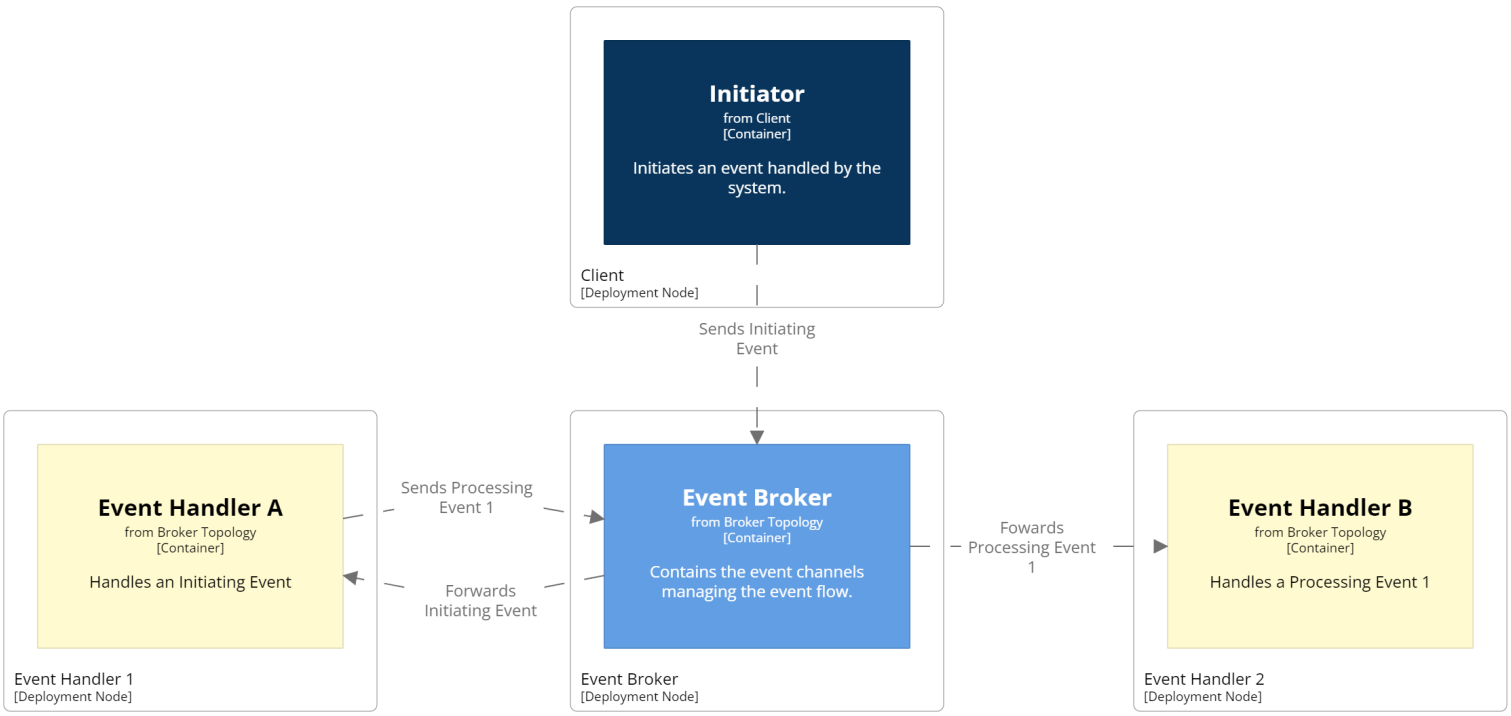

Figure 1: Conceptual deployment structure of an event-driven architecture.

Figure 1 shows the conceptual structure of an event-driven architecture. A client of some form sends the initiating event to a central event broker or mediator. That event is picked up by an event handler that processes the event. That may lead to the event handler generating a processing event to move to the next step of the process. The processing event is sent to the central broker and is picked up by another event handler. This continues until the business process is completed.

There are two basic approaches to implementing an event-driven architecture. The terminology is that these are different *topologies*, as they have different high-level structures. The *broker topology* is the simpler of the two and is optimised for performance, responsiveness, scalability, extensibility, and low coupling. The *mediator topology* is more complex but is designed to provide reliability, process control, and error handling.

2 Broker Topology

The broker topology consists of four elements.

Initiating Event starts the flow of events.

Event Broker has *channels* that receive events waiting to be handled.

Event Handler accepts and processes events.

Processing Event sent by an event handler when it has finished processing an event.

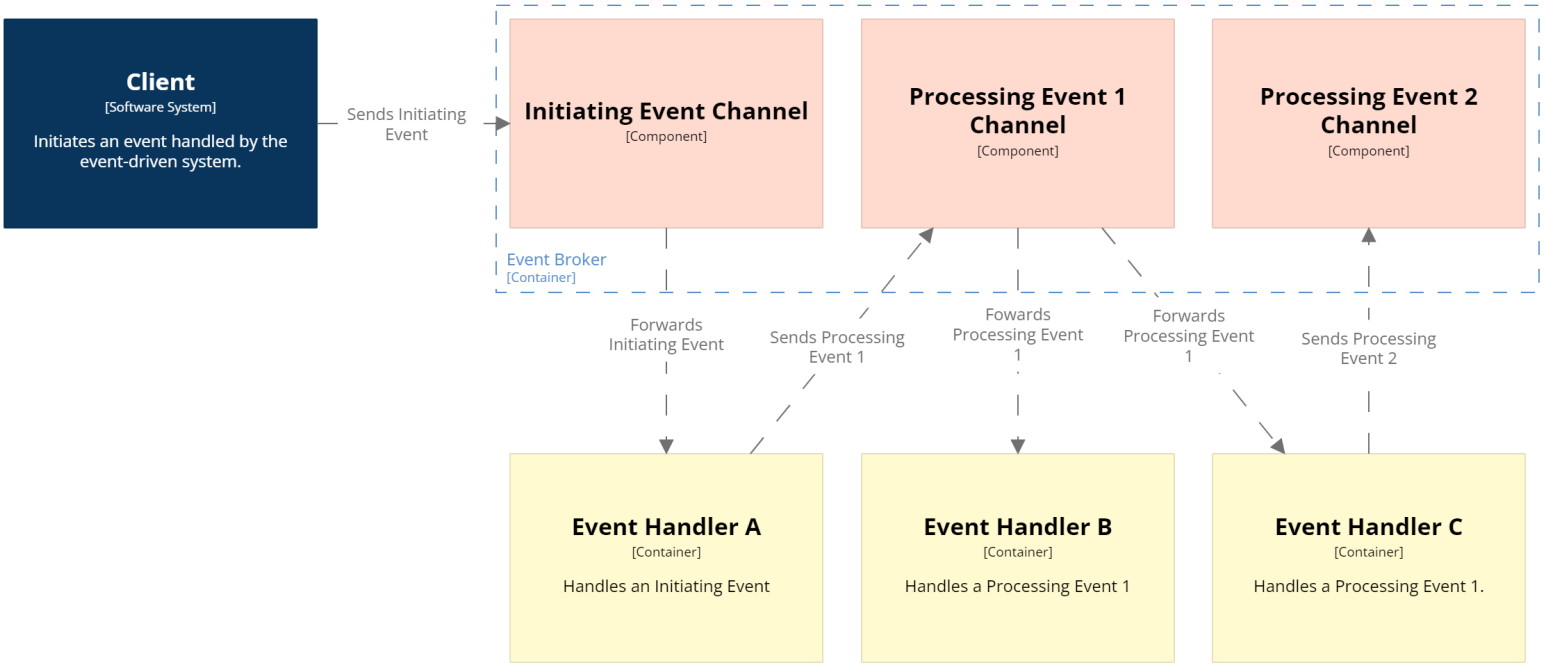

Figure 2: Basic broker topology.

In figure 2, the *Client* sends the *Initiating Event* to the *Initiating Event Channel* in the *Event Broker*. *Event Handler A* accepts the event and processes it. Upon completion of handling the *Initiating Event*, *Event Handler A* sends *Processing Event 1* to the appropriate channel in the event broker. *Event Handlers B* and *C* accept this processing event and perform their actions. When *Event Handler C* finishes processing it sends *Processing Event 2* to its channel. No event handler is designed to accept *Processing Event 2*, so it is ignored.

Different event channels in the event broker provide a simple mechanism to coordinate the flow of events in a business process. As shown in figure 2, there is a separate channel for each type of event. This allows event handlers to register to be notified of only the type of events they can process. This reduces the overhead of broadcasting events to handlers that cannot process them. The consequence is that event sources need to send their events to the correct channel. For a simple broker topology, this could be by sending event messages directly to a channel or, for better abstraction and reduced coupling, the event broker may implement a façade that directs events to the correct channel.

Definition 1. Event Handler Cohesion Principle

Each event handler is a simple cohesive unit that performs a single processing task.

The *event handler cohesion principle* minimises the complexity of each handler, and improves the ability of the system to scale only the tasks that need additional computing resources. It also makes it easier to design each handler to be independent of the other handlers, reducing overall system coupling.

2.1 Extensibility

There may be multiple event handlers for a single type of event, shown by *Processing Event 1* being sent to both *Event Handler B* and *C* in figure 2. This allows more than one action to be performed when an event is sent. This means that new event handlers can be added to the system as it evolves. A new feature can be added by implementing an event handler to do something new when an event is received by the event broker.

In figure 2, when *Event Handler C* finishes processing its event it sends *Processing Event 2* to the event broker. The diagram indicates that the system does not handle the event. The purpose of doing this is to make it easier to extend the system. Currently, the system may not need to do anything when *Event Handler C* finishes processing, but because it sends an event to indicate it has finished it means other tasks can be added to the system following *Event Handler C*. Because event handlers are independent of each other, *Event Handler C* does not need to be modified to cater for this later addition of functionality.

If there are events that are not handled by the system, the event broker façade can ignore them, it does not need a channel to manage them. If the system is extended and needs to process one of the ignored events, the event broker can create a new channel to manage them.

Event Handler B, in figure 2, does not send an event when it finishes processing the event it accepted. This is a valid design choice when implementing the system, if there is nothing foreseeable that might need to know when *Event Handler B* is finished processing. The drawback is that if the system later needs to do something else when *Event Handler B* is finished, it will need to be modified to send an event to the event broker. The tradeoff is reducing asynchronous communication traffic with unnecessary events, versus providing easy extensibility later.

2.2 Scalability

As was described in section 1, the broker topology is optimised for performance. Each event handler is a separate container that can be deployed independently of other handlers. A load balancer and an automated scaling mechanism ensures that each event handler can scale to manage its load.

There may be multiple clients sending events to be processed, and each event handler may itself be a source of events. This requires the event broker and its channels be able to handle the event traffic load. The event broker itself can be deployed on multiple compute nodes, with its own load balancing and auto-scaling. The challenge is to implement this in such a way that the event handlers do not need to know about the event broker's deployment structure.

A simple distributed event broker could deploy each channel on a separate compute node. The event broker façade is deployed on its own node and manages receiving events and sending them to the appropriate channel. The façade also manages how event handlers register to receive notification of events from channels. This works until the event traffic to the façade or a single channel exceeds their capacity.

A more robust approach, which scales to very high traffic levels, is to federate the event broker. This allows the façade and channels to be distributed across multiple nodes but provides an interface that the clients and event handlers can treat as a single access point. The complexity of implementing a federated computing system is beyond the scope of this course. There are several libraries (e.g. ActiveMQ or RabbitMQ) and cloud-computing platforms (e.g. AWS SQS, AWS MQ or Google Cloud Pub/Sub) that provide

this functionality. They can still be used when the system does not need to scale to a federated event broker, as they provide the underlying implementation for the event broker.

2.3 Queues

The other issue that the channels need to manage to allow scalability is holding events until they are processed by an event handler. The simple approach is that a channel implements a queue. Events are added to the end of the queue as they are received. When an event reaches the front of the queue, all the event handlers for the channel are notified that the event is available. The channel queue needs to be configured to cater for different implementation choices. The simple option is that when an event handler accepts the event, it is removed from the queue. This means that only one type of event handler listening to the channel will process the event.

If the system needs all the different types of event handlers to process the event, the queue needs to be configured so that the event is not removed from the queue until all the event handlers have retrieved it. This can be implemented in the queue by having multiple *front of queue pointers*, one for each type of event handler listening to the channel. This allows different event handlers to process through events in the queue at different rates. The queue should be implemented to pass queue size or the amount of time events are in the queue to the auto-scaling mechanism for each event handler. This can be used as part of the logic to determine when to deploy new event handlers, or to scale-back when traffic decreases.

To increase reliability, the queue can be implemented to only remove the event from the queue once the event handler has finished processing it, rather than when it has been received by the handler. This allows another event handler to process the event if the first event handler to accept the event fails. A timeout mechanism can be used to provide this functionality. If the queue does not receive notification that the event has been processed within a certain amount of time, it notifies the event handlers that the event is available.

Another consideration to increase reliability is dealing with when the event broker, or one of its queues, fails. If queues are implemented as in-memory queues, events in the queue will be lost when the event broker or queue fails. Queues can be implemented to persistently store events until they have been processed. A federated event broker will further improve reliability by making it less likely that a single queue will fail between receiving an event and storing it persistently.

2.4 Streams

2.5 Sahara Example

The service-based architecture example for the Sahara eCommerce system is designed to perform synchronous processing of requests. A customer *requests* to view a product's details through a REST API and receives a response with the result. Similarly, a customer adding a product to their shopping cart is another request.

Adding auctions to the Sahara eCommerce system is a simple example that benefits from an event-driven architecture. A customer sends a message to the Sahara eCommerce system initiating the *event* of making a bid. Upon receiving the bid event, the system checks the bid against the current high bid and determines the new high bid. The system then generates a new high bid event. This event triggers actions in the system to update the high bid and notify the bidder of the result of their bid. It may also trigger an action to notify the previous high bidder that they are no longer the high bidder.

Multiple customers could submit bids for the same item at almost the same time (e.g. bid sniping). These bid events are queued to be processed in the order they are received. This queuing mechanism allows event handlers to process messages that arrive faster than the handler can process them. It assumes that the message load will reduce in time and that the event handler will process all messages in the queue.

3 Design Considerations

A service-based architecture is typically used for medium-sized systems.

4 Service-Based Principles

There are a couple of principles which should be maintained when designing a service-based architecture to produce a simple, maintainable, deployable and modular designs.

Definition 2. Independent Service Principle
Services should be independent, with no dependencies on other services.

5 Extensions

There are a few common variations of the service-based architecture to consider.

5.1 Separate Databases

The first variation we will consider is to have separate databases for each service.

6 Conclusion

Service-based architecture is an approach to designing a distributed system that is not too complex. Domain services provide natural modularity and deployability characteristics in the architecture design. Well designed service APIs improve the encapsulation and hide implementation details of the services.