Event-Driven Architecture

March 27, 2023

Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Event-Driven Architecture

Software Architecture

March 27, 2023

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 business process.

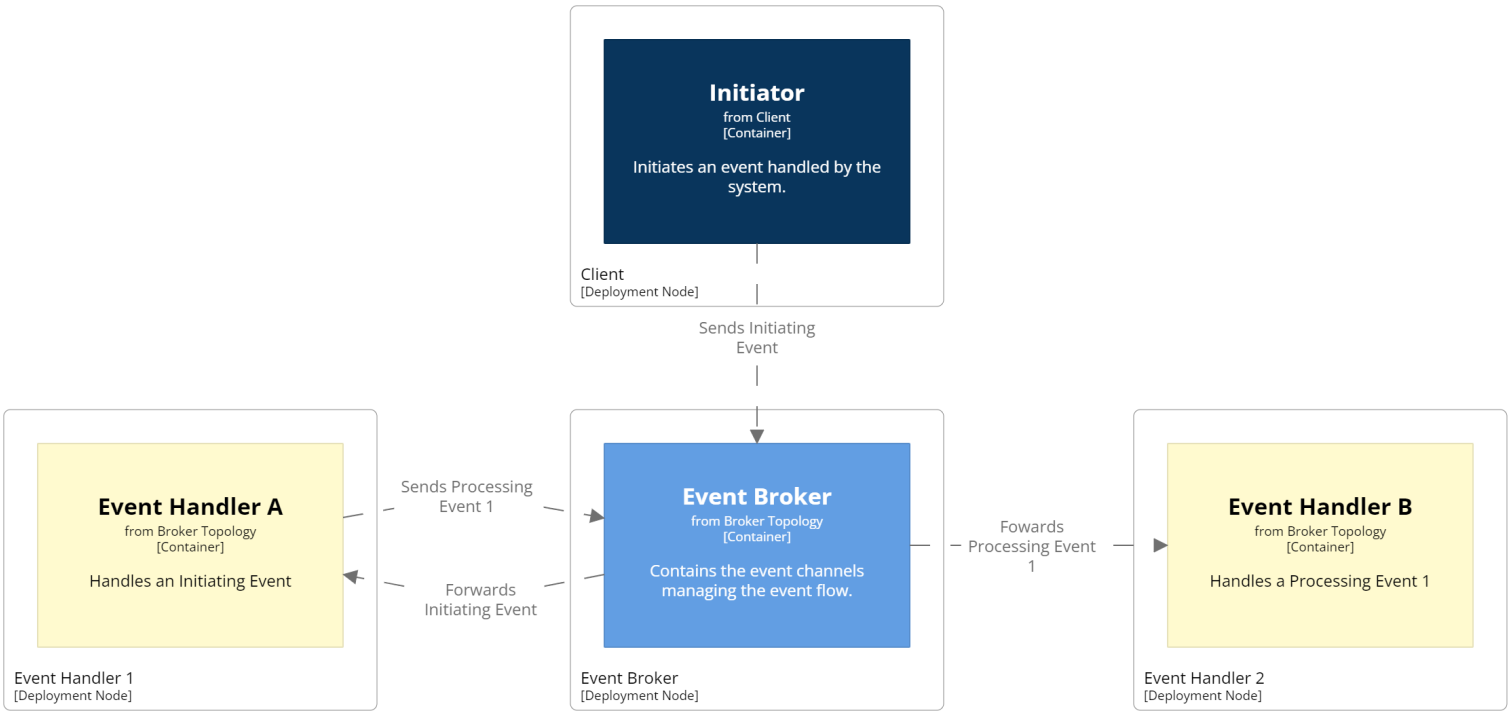

Figure 1: Conceptual deployment structure of an event-driven architecture.

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.

An auction system, as shown in figure 2, could be implemented as an event-driven architecture.

- When a customer bids on an item, they initiate a bid *event*.
- Upon receiving the bid event, the event broker notifies the bid handler, which checks the bid against the current high bid and determines the new high bid.
- The bid handler generates a new high bid event.
- The event broker notifies the page update handler of the high bid event, which sends the new high bid to all pages displaying the item.
- The bid notification handler is also notified of the high bid event and it sends an email to the bidder with the result of their bid.
- And, the rebid notification handler is also notified of the high bid event. If the previous high bidder is no longer the high bidder, they are sent an email asking if they would like to make another bid.

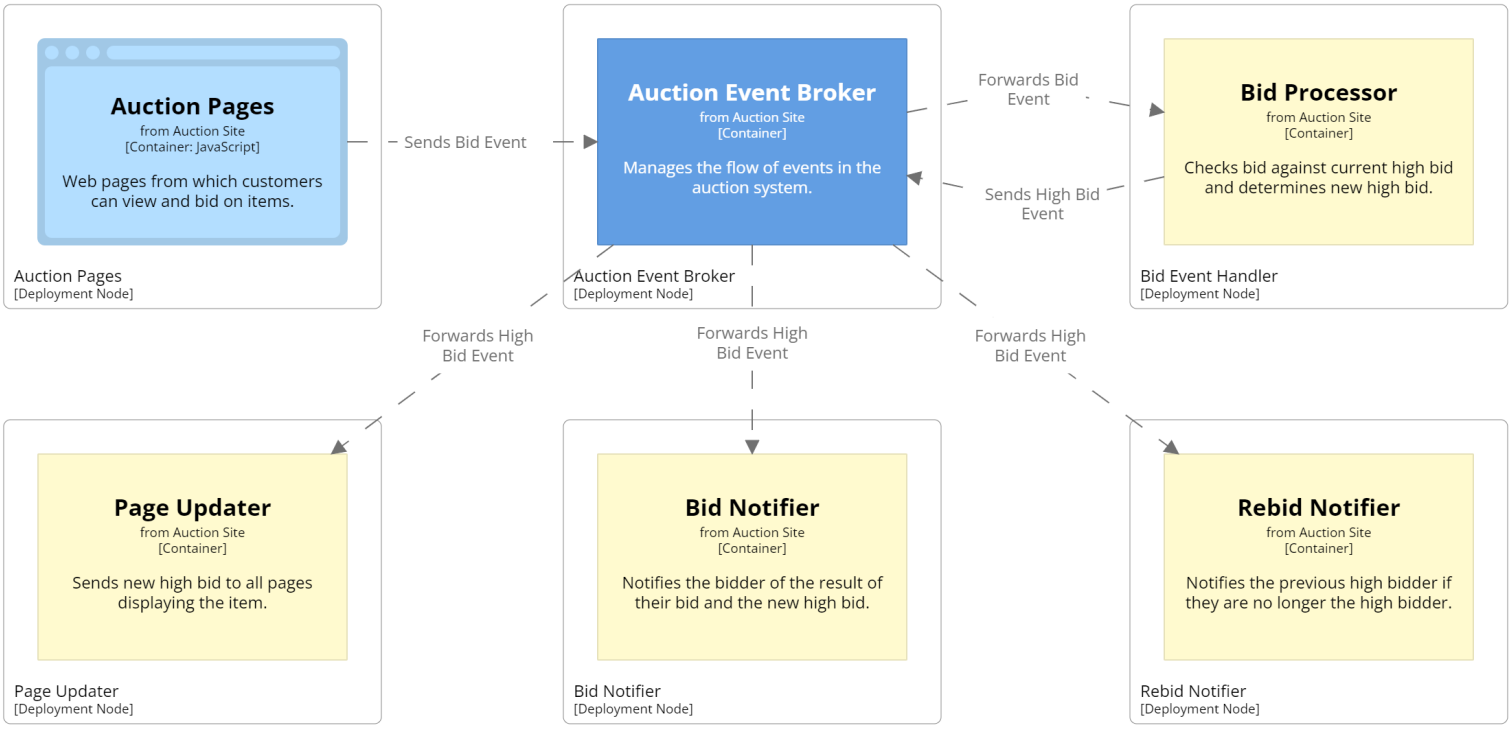

Figure 2: Auction bidding example event-driven architecture.

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

2 Broker Topology

The broker topology consists of five elements.

Initiating Event starts the flow of events.

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

Event Channel holds events waiting to be processed by an event handler.

Event Handler accepts and processes events.

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

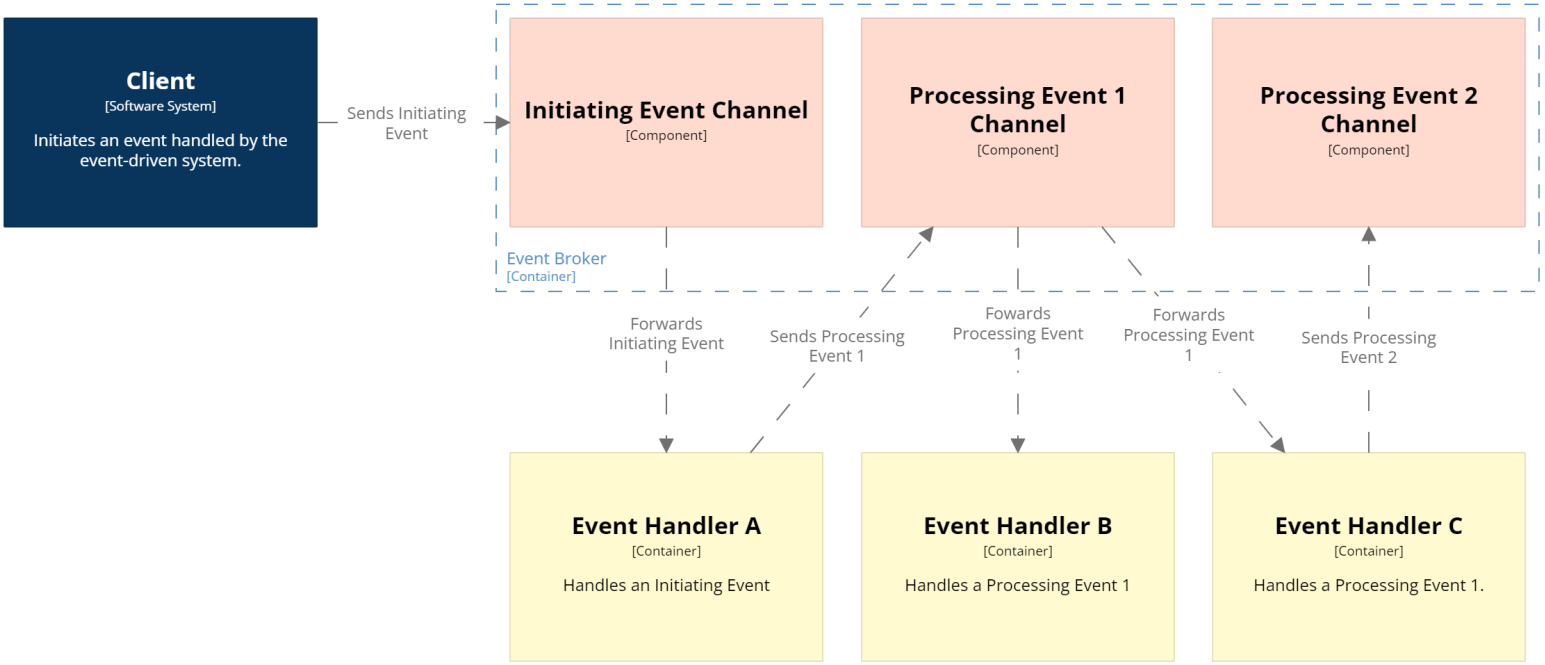

Figure 3: Basic broker topology.

In figure 3, 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 3, 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.

Definition 2. Event Handler Independence Principle

Event handlers should not depend on the implementation of any other event handler.

Dependencies between event handlers limit the processing throughput of the system, and increases the complexity of implementing event handlers. The only dependency should be the event that is processed by the event handler.

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 3. 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 3, 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 3, 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[1] or RabbitMQ[2]) and cloud-computing platforms (e.g. AWS SQS[3], AWS MQ[4] or Google Cloud Pub/Sub[5]) 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.

[1] https://activemq.apache.org/
[2] https://www.rabbitmq.com/
[3] https://aws.amazon.com/sqs/
[4] https://aws.amazon.com/amazon-mq/
[5] https://cloud.google.com/pubsub/docs/overview

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

An alternative to channels using queues to manage events, is to use streams instead. Event sources send events to the event broker and they are added to the appropriate stream. Each channel becomes a stream and events are added to the stream. The observer pattern is used where the channel is the subject and the event handlers are the observers. When an event is added to a stream, the channel notifies all its observers that an event has been added. The event handlers then decide whether to retrieve the event and process it.

The most important difference of using streams versus queues is that the events are stored permanently in the stream. Event handlers can process events at their own pace. Because the events are stored permanently, history is maintained. This allows the system to go back and redo past processing or for an audit to be conducted of what happened during processing.

2.5 Queues vs. Streams

Queues work well when there are known steps in the business process being implemented. Queues ensure that all preceding events in the business process are completed before the next step. Queues can ensure that events are only processed once (known as “exactly once” semantics). An ecommerce website is an example of the type of system suited to using queues. This is because there is a fixed set of rules about which event handlers process which events.

Streams work well when there are multiple event handlers responding to events, particularly when event handlers may choose to ignore some events based on various conditions. Streams mean that the event broker or channels do not need to be notified if an event handler wants to ignore an event. Because

streams store events permanently, data analysis of the system's behaviour can be done by processing past events. Streams work well for social media platforms as monetisation of the platform depends on analysing user behaviour. If user generated events are stored permanently in streams, new analyses can be performed on past data.

Streams also work well for systems that generate a vast number of events, or that have a large number of event handlers. A health monitoring system, taking advantage of the Internet of Things[6] (IoT), would be an example of such a system. There could be a very large number of monitoring devices that generate frequent events to notify the system of physiology changes. Streams make it practical to process different events at an appropriate pace. Some events may need almost immediate processing (e.g. a monitor reporting heart failure). Other events may need periodic processing (e.g. analysing patient blood pressure over a period of time). Streams can accept and store events and different event handlers can be optimised for how they process data in the stream.

The health monitoring system would also have a large number of event handlers, because many devices would process some events (e.g. a heart rate display or a patient alarm). Streams have less overhead to manage their events, so can more easily cope with many event handlers retrieving events for processing.

Streams also enable *event sourcing* as a persistence mechanism. Event sourcing stores every event, and its associated state, rather than storing and updating system state. This makes it easier to analyse how the system is used and what changes happen to the state over time.

3 Mediator Topology

The mediator topology extends the broker topology by adding an *event mediator*. The event mediator manages the flow of events to implement the business process. There are six elements that make up the mediator topology.

Initiating Event starts the flow of events.

Event Queue holds initiating events sent to the event mediator.

Event Mediator coordinates sending events to the event handlers to implement a business process.

Processing Event sent by the event mediator to start the next step in the process.

Event Handler accepts and processes events.

Event Channel holds events waiting to be processed by an event handler.

In figure 4, the *Client* sends the *Initiating Event* to the *Event Queue*. The *Event Queue* notifies the *Event Mediator* that an event has been enqueued. The *Event Mediator* dequeues the next *Initiating Event* and starts the business process to deliver the behaviour for that event. The *Event Mediator* adds a *Processing Event 1* to its channel. The channel notifies the event handlers listening for this type of event. The event handlers retrieve the event from the channel and process it.

When an event handler finishes processing it does not send a processing event to the *Event Mediator* to be handled. Rather, the event handler usually sends an asynchronous message to the *Event Mediator* indicating that the handler has finished processing. When the *Event Mediator* receives a completion message from an event handler, the mediator determines what is the next step in the process and sends a processing event to the appropriate channel to continue the logic.

The event mediator allows the system to implement more sophisticated logic to implement the business process. Based on the result received from an event handler when it finishes processing an event, the mediator can decide which processing event to send next. This allows conditional or iterative logic to be implemented in the business process. Unlike the broker topology, which only has a sequential flow through the business logic based on event handlers sending new events when they finish processing.

[6] https://www.oracle.com/au/internet-of-things/what-is-iot/

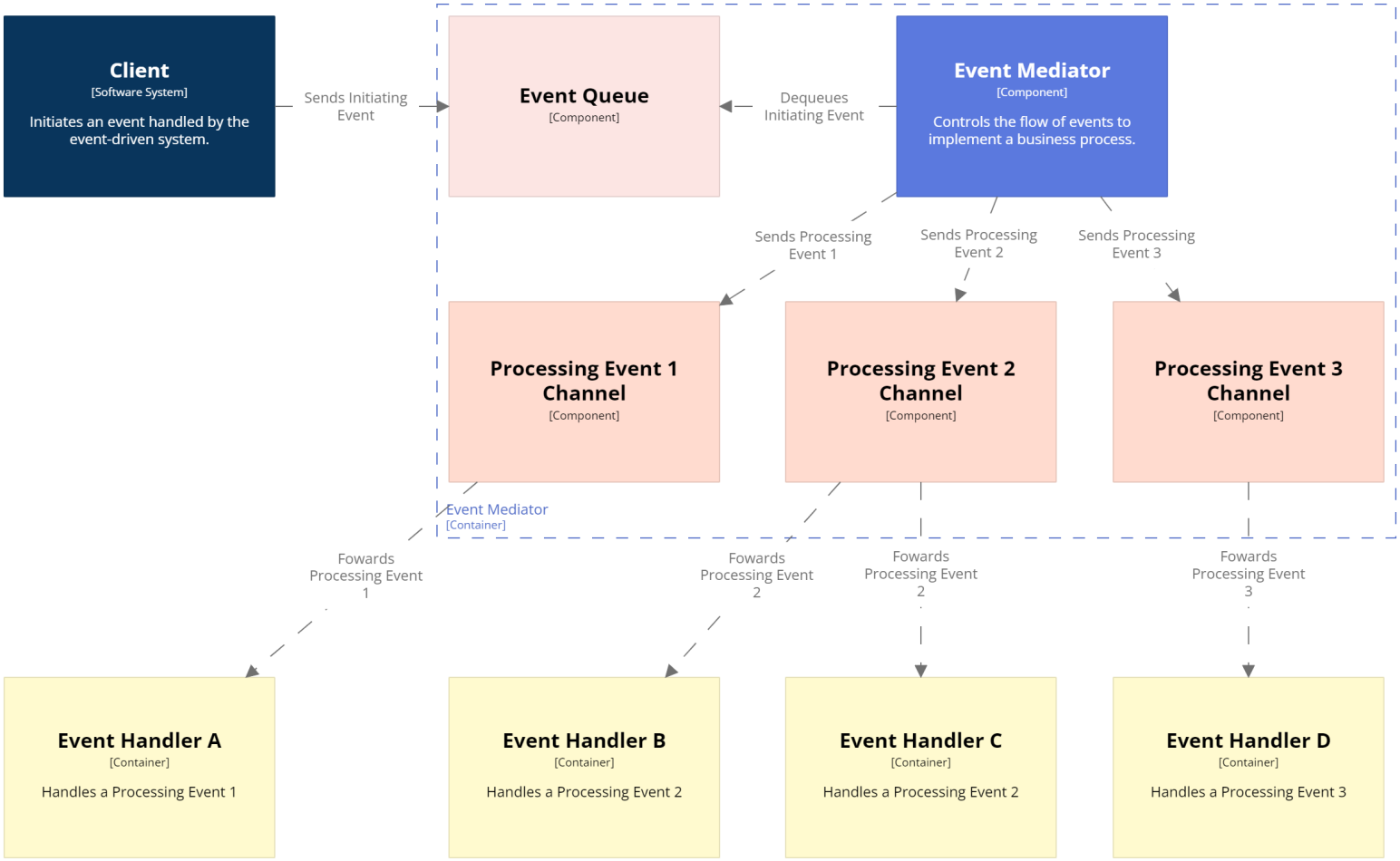

Figure 4: Basic mediator topology.

3.1 Implementation

Commonly, the mediator topology will implement multiple event mediators. Each one deals with a subset of events. These are usually related to a particular part of the problem domain. For example, in the Sahara eCommerce system, there could be event mediators for:

Customer Events – registering, updating profile, viewing order history, ...

Browsing Events – searching, browsing by category, requesting product detail, ...

Order Events – adding product to shopping cart, checking out, cancel order, ...

Inventory Events – viewing inventory levels, reporting on popular products, ordering new stock, ...

This allows different mediators to be scaled independently, based on their load. It also improves reliability, as one mediator can fail but others can continue to operate.

Like with the broker topology, there are many libraries and cloud-computing platforms that support implementing the mediator topology. Different libraries focus on different aspects of event coordination within the event mediator.

If the event coordination and error handling that needs to be implemented are relatively simple, it is often easiest to implement the logic in code. Libraries such as Apache Camel[7] or Spring Integration[8], provide the messaging infrastructure to support this type of implementation.

[7] https://camel.apache.org/
[8] https://spring.io/projects/spring-integration

If the event coordination and error handling are fairly complex, involving a lot of complex conditional processing or dynamic path determination with complex error handling, it is often easier to use a declarative DSL[9] to implement the logic. One such language is Business Process Execution Language[10] (BPEL). It describes the steps involved in processing an event, along with error handling, multicasting and other features. Libraries such as Oracle's BPEL Process Manager[11], provide infrastructure to support the description and execution of the process.

BPEL is not suited to event coordination and error handling that may require human intervention, due to the long run times of this type of processing. A more sophisticated Business Process Management (BPM) execution engine may be more appropriate in this context. These engines use multiple DSLs to provide more sophisticated process automation. Libraries such as jBPM[12] provide the infrastructure that enable this type of event mediator implementation.

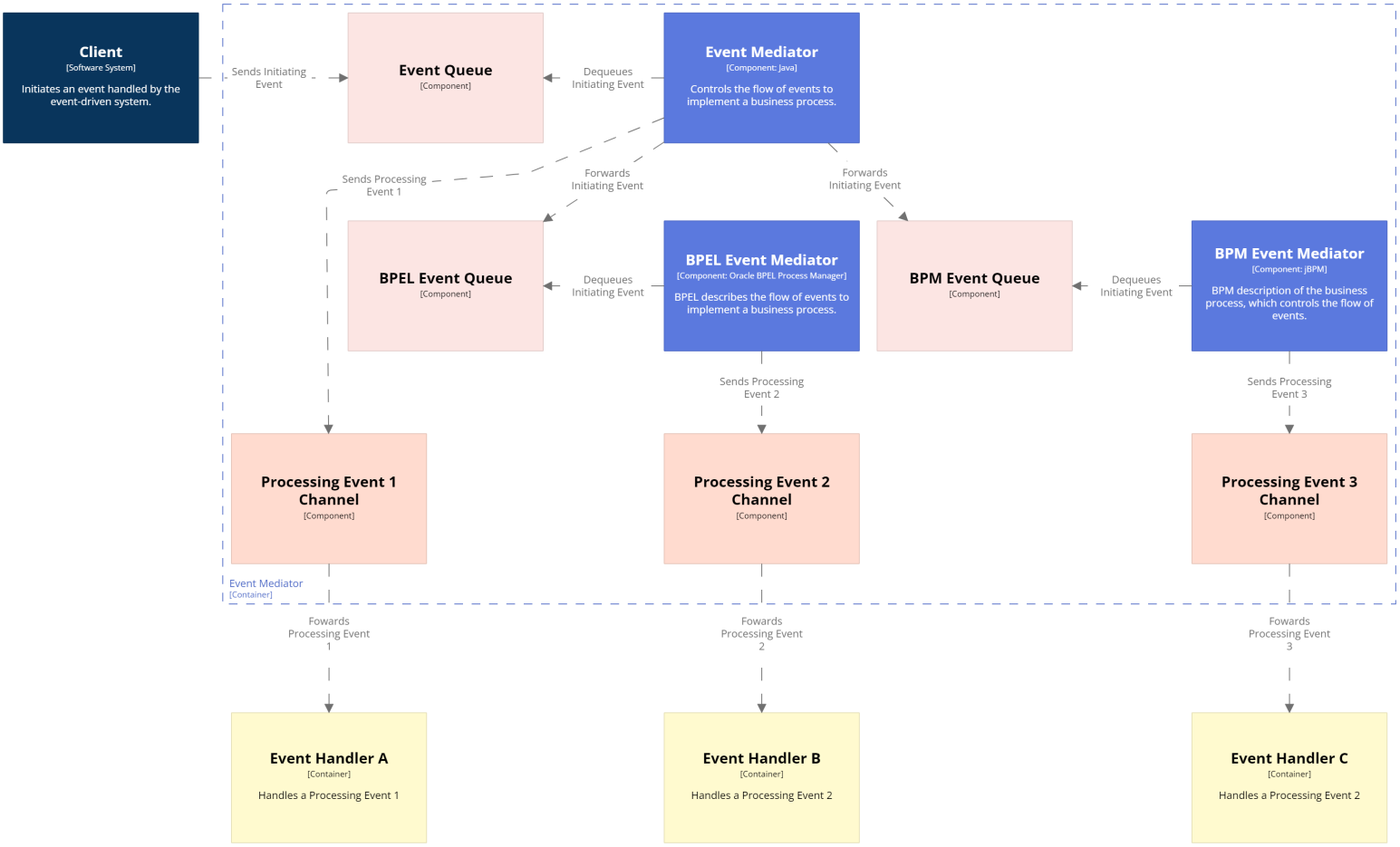

Figure 5: Complex mediator topology.

Figure 5 is an example of extending the idea of implementing multiple event mediators to have different types of mediators based on the complexity of the event. This is because most large systems will have a mix of simple and complex events, and some of the complex events may have long durations. It is usually more efficient in developer time and effort to apply different strategies for different types of events. Describing and verifying the process for a simple event using BPEL or a BPM execution engine may cost more developer effort than implementing the logic programmatically. To minimise coupling, the client still only sends initiating events to a single event queue. An event mediator retrieves the events from the event queue and classifies them, forwarding the events to the appropriate type of mediator.

[9] Domain Specific Language

[10] https://www.oracle.com/technical-resources/articles/matjaz-bpel.html

[11] https://www.oracle.com/middleware/technologies/bpel-process-manager.html

[12] https://www.jbpm.org/

4 Broker vs. Mediator Topologies

One way to think of the differences between the broker and mediator topologies is:

Broker is a *dumb pipe*.

Mediator is a *smart pipe*[13].

An event broker only provides a messaging service to notify event handlers of events. The broker does not control the logic of business process. Consequently, it cannot maintain any state related to the process. It also cannot perform any error handling if an error occurs in the middle of the process.

An event mediator coordinates the implementation of the business process by sending events to start each step of the process. This allows a mediator to implement complex logic. This can include maintaining state for the process and performing error handling. In the auction example, once a bidder wins an auction they have to pay for the item. *Pay for item* would be the initiating event to being the process of paying for the item. *Notification sent* and *order fulfilled* would be other events in the payment process. If the payment failed, the event mediator could store the state of the process and once payment was successfully processed it could restart the process.

A conceptual difference between the broker and mediator topologies is that the broker topology is a pure form of event handling. Events represent things that have happened. Once action has occurred, the event is published and event handlers react to it. In the auction system, events in a broker topology would be *bid placed*, *new high bid*, *payment started*, *payment accepted*, ….

The mediator topology is more like a command driven process. The events are commands sent by the event mediator to perform the next step in the business process. In the auction system, events in a mediator topology would be *place bid*, *notify bidders*, *notify previous high bidder*, *pay for item*, ….

Consequently, in the broker topology, because events represent actions that have occurred, some events can be ignored. In the mediator topology, because events are commands to perform the next step in the business process, they must be executed.

Broker Advantages

- Greater scalability because the broker just distributes event notifications and does not have state, it is easier to federate the broker to provide greater throughput.
- Greater reliability because the broker does not coordinate the business process. If the broker is federated and a broker compute node fails, another node can accept the next processing event and the process can continue.
- Lower coupling because the only dependency between parts of the system are the events.

Mediator Advantages

- Allows for more complex business process logic and error handling.
- Can store process state and restart a process after recovering from an error.

This is not to say that the mediator topology scales poorly, it is just easier to scale to extreme loads with the broker topology. The broker topology takes independence to the extreme to reduce coupling. To provide process control, the mediator topology requires some coupling between the event mediator and the event handlers. The trade-off to consider is whether the business process logic is complex enough that it is easier to implement with the mediator topology.

[13] This is a corruption of the original usage of "smart" pipes, which was related to complex Enterprise Service Buses. Though, using BPEL processors or BPM engines is moving into the same territory of intelligence.

Reliability versus recoverability is another trade-off to consider. If the event broker is federated and there are multiple instances of each event handler, it is extremely unlikely that some part of the system will fail and not be available. But, this does not deal with errors that occur within a business process. The event mediator can recover from errors and complete a business process.

It is possible to have a hybrid architecture that uses both broker and mediator topologies. Event mediators can implement complex business processes in the system. Event brokers can handle events related to simpler business processes.

5 Conclusion

Event-driven architecture enables delivering a highly scalable and extensible distributed system. Implementation is complex, though this is offset to a certain degree by the availability of libraries or services that implement much of the event broker or event mediator behaviour. The event handler cohesion and event handler independence principles help reduce the complexity of the event handlers. It is the event logic coordination, along with asynchronous communication and inevitable communication failures, that adds to the complexity of the architecture.

Testing and debugging asynchronous communication is a difficult task. The scale of most event-driven architecture tends to exasperate this difficulty.

The event-driven architecture can be combined with other architectural styles. This allows other architectures to take advantage of the scalability and responsiveness of the event-driven approach. Two architectures that commonly are combined with the event-driven architecture are space-based and microservices. They are not the only hybrid approaches, many other architectures use or borrow from event-driven (e.g. event-driven microkernel).