FHV
Vorarlberg University
of Applied Sciences

Application Integration and Security

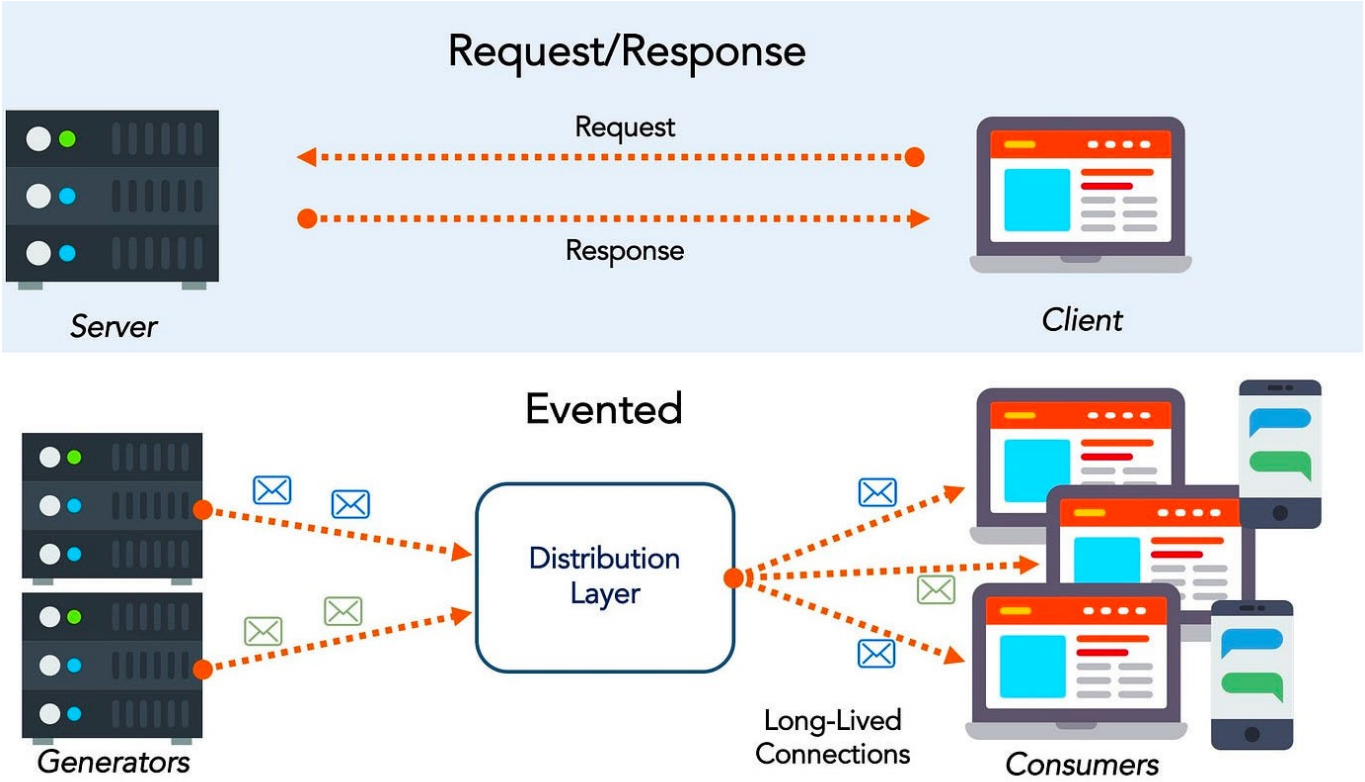
Valmir Bekiri

Philipp Scambor

Event based integration

Distributed Systems
Application integration

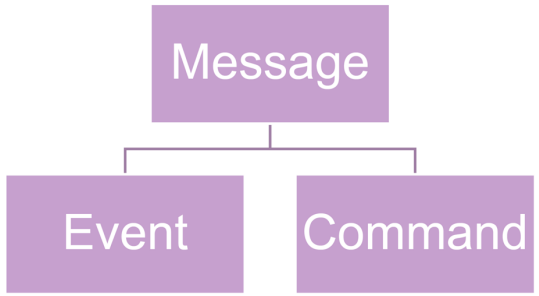
- Application Integration in distributed systems takes place via **various technologies/protocols**
  - **Synchronous communication**
    Request-Reponse based
    RPC, gRPC, SOAP, REST, GraphQL
  - **Asynchronous communication**
    Event based
    MQTT, AMQP
- **MQTT** Message Queuing Telemetry Transport
  lightweight, publish-subscribe based protocol for resource-constrained devices
- **AMQP** Advanced Message Queuing Protocol
  standard for passing business messages between applications or organizations



https://medium.com/analytics-vidhya/apache-kafka-architecture-getting-started-with-apache-kafka-771d69ac6cef

Distributed Systems
Event-based Integration

- Events/messages can be created, sent and consumed within a monolithic system, e.g. AuctionItemSold, but event-based communication is very common in **distributed systems**.
- Instead of creating dependencies defined in the source code (ClassA -> ClassB), eventbased systems communicate with each other via events. There are
  - **Producers** (event source – publisher, producer) of events: generates a message/event
  - **Receivers** (event sinks – consumer, subscriber) for events: read and process the event
- The connection between source and sink can be established & terminated during runtime
- A message can generally have two characteristics:
  - **Event**: Describes an event that has already happened
    e.g. CustomerCreated, MailSend, PaymentReceived
  - **Command**: Describes a work order that needs to be completed
    e.g. CreateCustomer, SendMail, RequestPayment

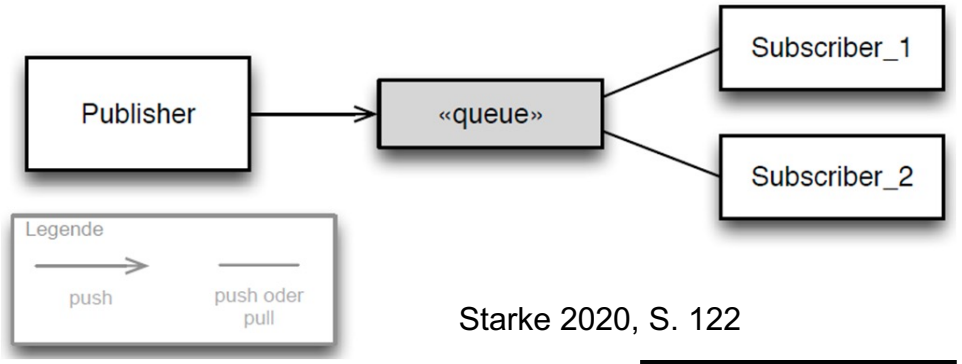


Starke 2020, S. 121

Event-based Integration

Basic concepts - Types

- **Broadcast (unbuffered)**
  - Messages are published on a local network.
  - All consumers connected to the network read these messages and filter them according to their relevance. For example, the header fields of the protocol could be read.
- **Event queue (buffered)**
  - There is a **mediator / broker** between the sender and receivers that takes care of the connection.
  - **Messages** are **sent to a queue**, where they can be read synchronously or asynchronously.
  - **“Fire-and-Forget” Principle**: producers do not know whether the messages has been received or processed by the consumer(s). Even the order of consumption cannot be ensured.
  - Several **consumers** can **listen to a queue** and process the same message.
  - Queues can create a buffer, especially with high throughput.

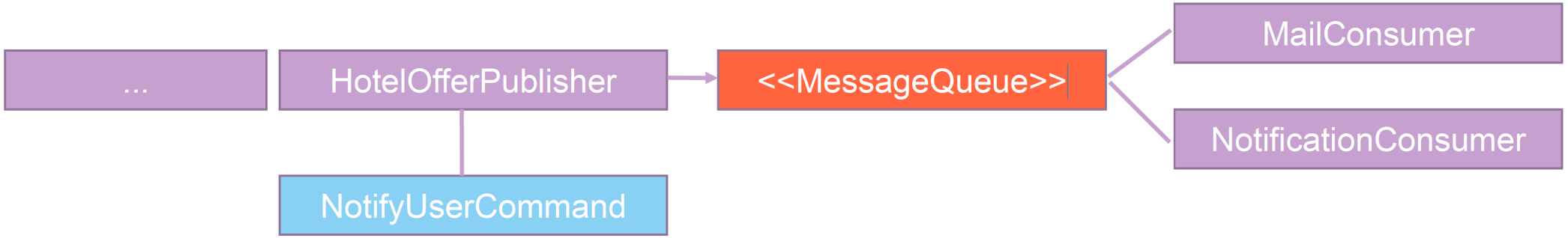


Starke 2020, S. 122

Event-based Integration

Example: Hotel booking system

- Each user is informed about the latest hotel offers once a day.
- Each user can specify in their user profile whether they want to receive these via email or app push notification
- More than 1 000 000 messages are generated daily.



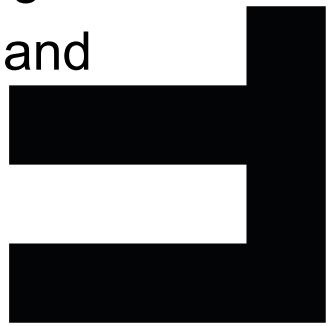
Event-based Integration

Basic concepts - Evaluation

- **Advantages**
  - Technology independent (e.g., producer = WebApp, consumer = Console Application on local machine)
  - Programming language independent
  - Producer and consumer are independent of each other => high level of decoupling and isolation
  - State independent: producer can publish an event even though the consumer is offline
  - Availability and robustness can be significantly increased, especially with a high number of messages
- **Disadvantages**
  - Complex to implement and operate
  - High integrity and reliability must be ensured
  - Asynchronous programming is very complex, as no direct response is possible
    - Troubleshooting is time-consuming
    - Transactions are difficult to implement

Event-based integration and Reactive systems

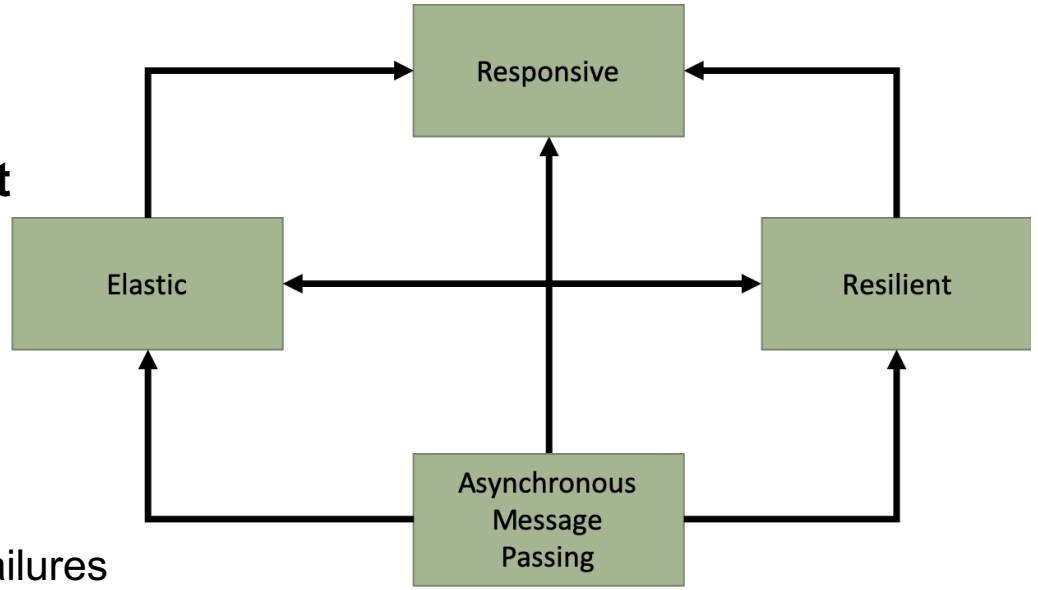
- Instead of request-response based integration, **event-based integration turns things** around. Consuming services no longer initiate requests and then wait for responses.
- **Consuming services turn to event-emitting services** that generate events to communicate that something has happened and expect other services to know how to process these events.
- **Event-emitting services** do not specify what needs to be done but **expect that other services will process them in accordance with the requirements**. Therefore, event based systems are asynchronous by nature and usually more evenly distributed.
- Consequently, event-based integration usually results in a strong decoupling of emitting and consuming services (basis for microservice architectures and **reactive systems**) and simplifies horizontal scaling, maintenance, reliability and resilience.



Event-based Integration

Reactive systems – Architectural style

- The term **reactive systems** was coined to describe an architecture style to deliver **responsive and reactive applications, at the system level**.
- It is designed to enable applications composed of **multiple microservices** working together as a single unit to **better react to their surroundings and one another**
- Main characteristics:
  - **Responsive**: a reactive system needs to handle requests in a reasonable time
  - **Resilient**: a reactive system must stay responsive in the face of failures
  - **Elastic**: a reactive system must stay responsive under various loads (scale both up and down)
  - **Message driven**: components from a reactive system interact using asynchronous message passing to enable loose coupling, isolation and location transparency.

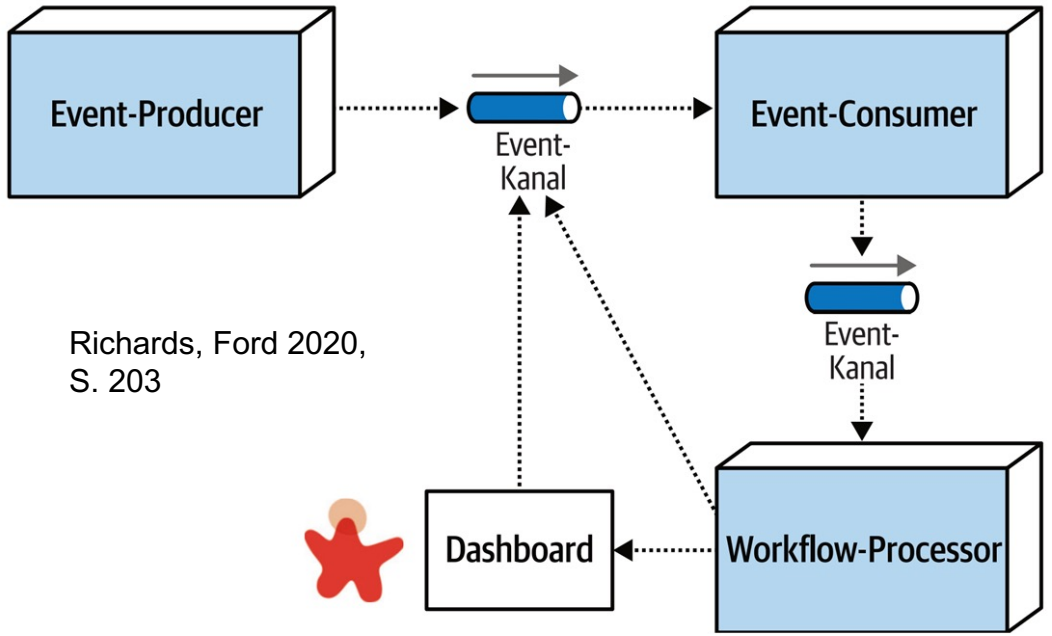


https://developer.ibm.com/articles/reactive-systems-getting-started/

Event-based Integration

Reactive systems – Error Handling

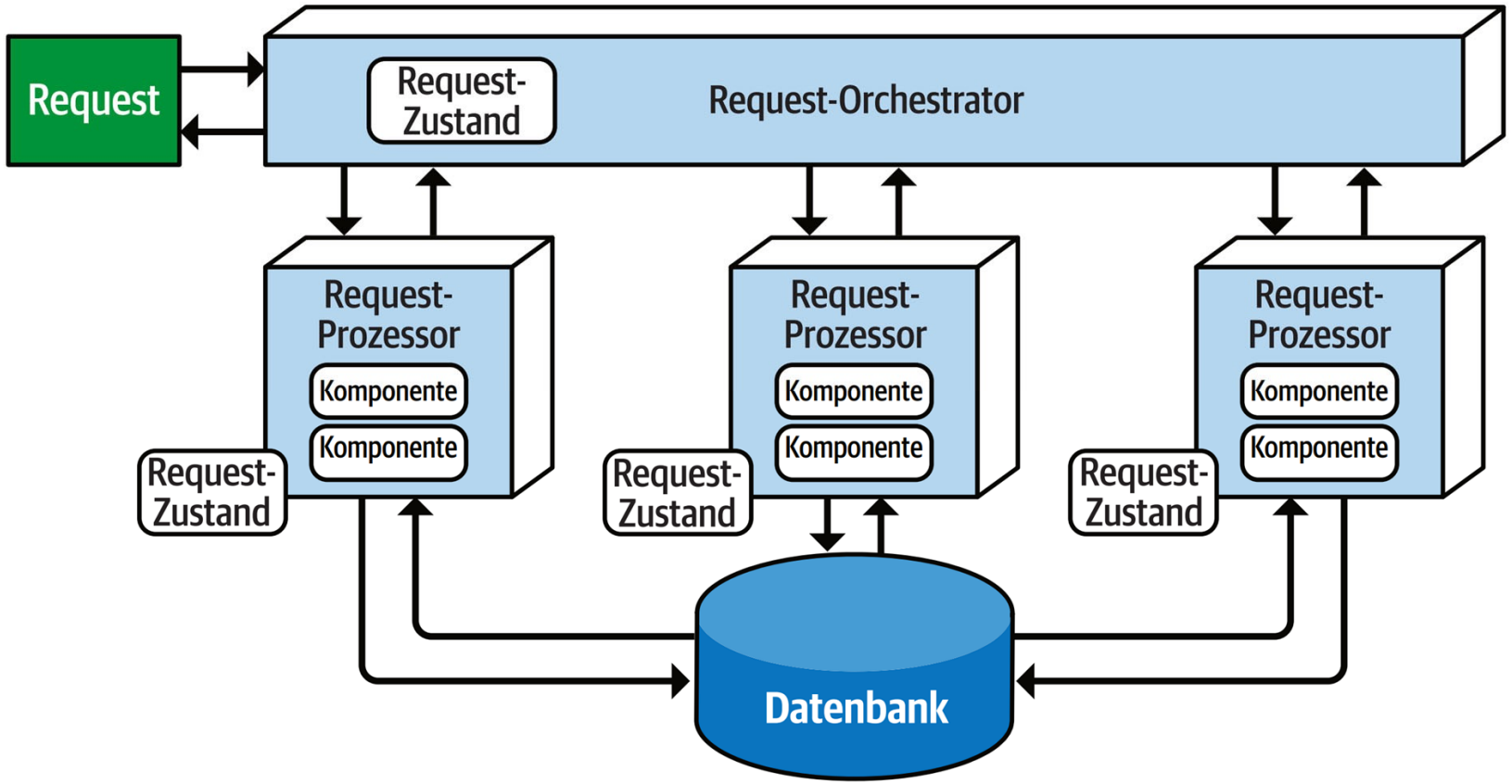
- Asynchronous communication leads to **reactive architectures**, which are decoupled, easier to extend, maintain and scale.
- Main problem: error handling, which increases complexity => Solution: **workflow event pattern**.
- If the workflow processor receives an error, it tries to find out what is wrong with the message: static-deterministically (rule-based) or statistically (AI/ML). In both cases, the workflow processor attempts to solve the error without human intervention and sends the event with the adapted data back to the message queue.
- Consumer tries to process it again, if problem not solved workflow processor forwards the message to another queue, which ultimately ends up in a so-called "dashboard".



Richards, Ford 2020, S. 203

Request-based Architecture

Request-Orchestrator

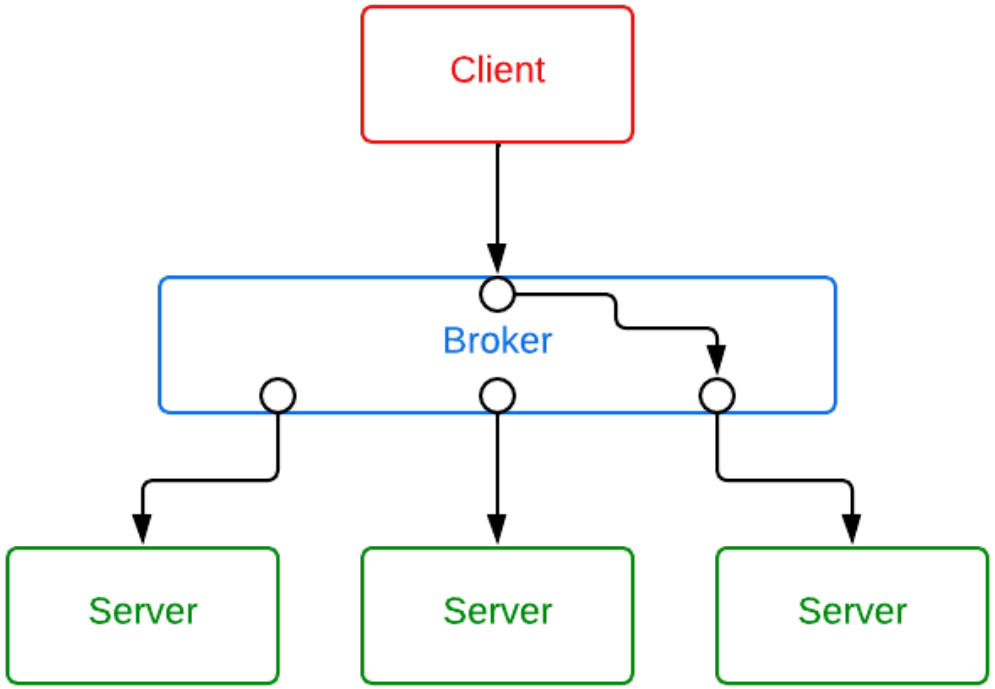


Richards, Ford 2020, S. 183

Event-based Integration

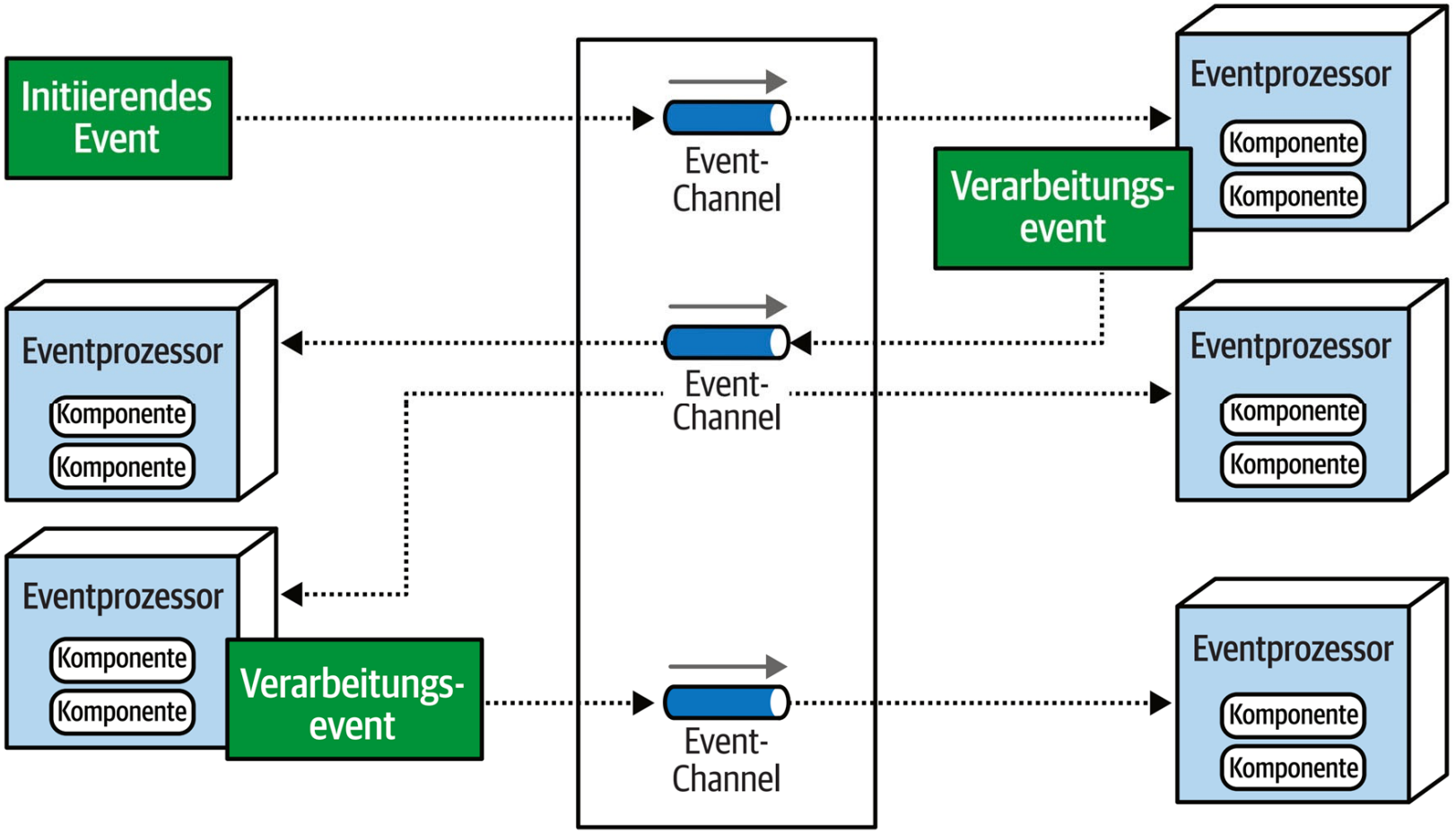
Broker Pattern

- Broker systems are used to structure distributed systems whose components have little or no coupling
  - The broker is responsible for communication between the components
  - Servers tell the broker which services and properties they have
  - Clients request a specific service from the broker
  - The broker forwards the information according to the properties of the server and the requirements of the client



Event-based Architecture

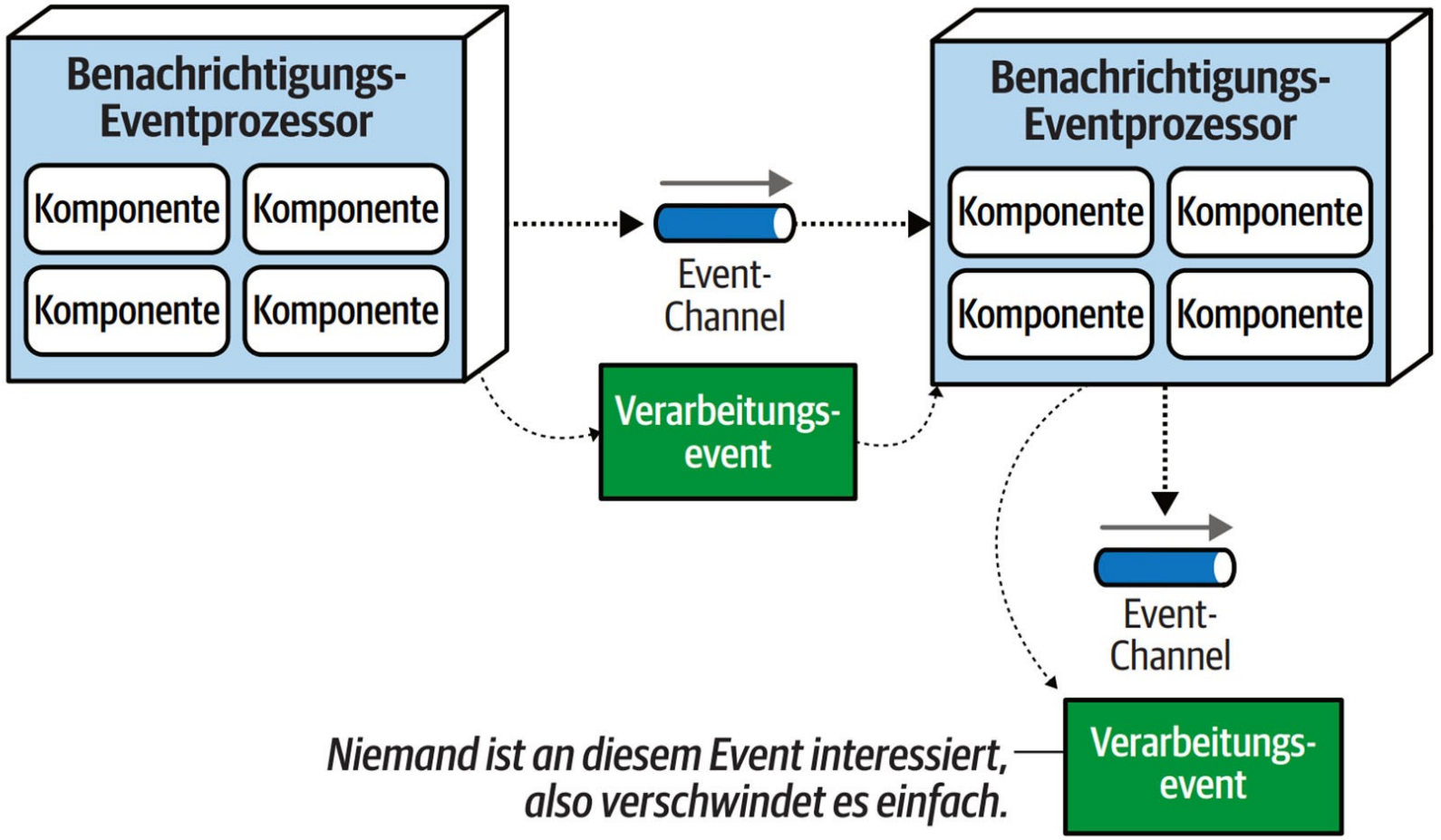
Broker Topology - Components and Workflow



Richards, Ford 2020, S. 185

Event-based Architecture: Broker Topology

Components and Workflow

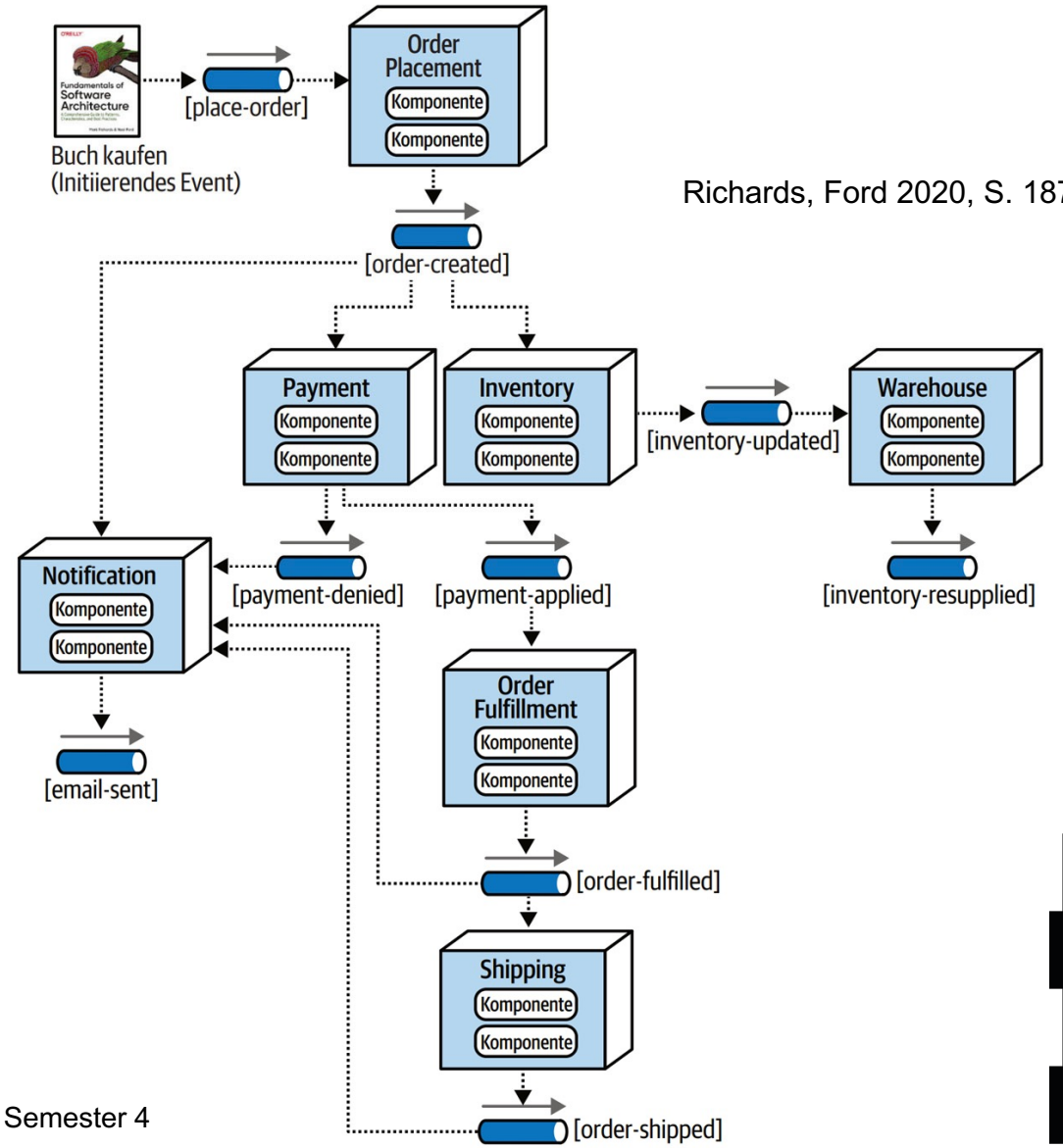


Richards, Ford 2020, S. 186

Event-based Architecture: Broker Topology

Example: Commerce

- **Process flow using Broker Pattern**
  - Initially, the ***OrderPlacement*** event processor receives the initiating event [place-order], which inserts the order into a database table and returns the order ID to the customer. Finally, it then informs the rest of the system that it has created an order using the [order-created] processing event.
  - Three other event processors (**Notification, Payment, Inventory**) are "interested" in this event. All three event processors carry out their tasks in parallel.
  - The **Notification** event processor sends the email to the customer and generates a new processing event [emailsent]. No other event processors listen for this event. => illustrates the architectural extension option!
  - The Inventory event processor decrements the stock for the ordered book and announces this action via the processing event [inventory-updated]. …



Richards, Ford 2020, S. 187

Event-based Architecture: Broker Topology

Analysis

- **Process Flow**
  A command always results in an event until nobody reacts to the event anymore.
- **Decoupling of the components**
  The downstream, consuming component is strongly decoupled from the producing system.
- **Asynchronous communication**
  The producing system does not have to worry about the availability of the consuming system.
- **High configuration effort**
  The topics and data formats used must be coordinated between the components.
- **Transactions are complex**
  Although they are possible (saga pattern), they are technically difficult to implement (fire-and-forget).
- **High performance**
  The consumers determine how quickly the messages are retrieved from the topic.
- **Error handling**
  Any technical or functional errors that occur must also be handled via messages, which is complex.

Event-based Architecture: Broker Topology
Advantages and disadvantages

Advantages

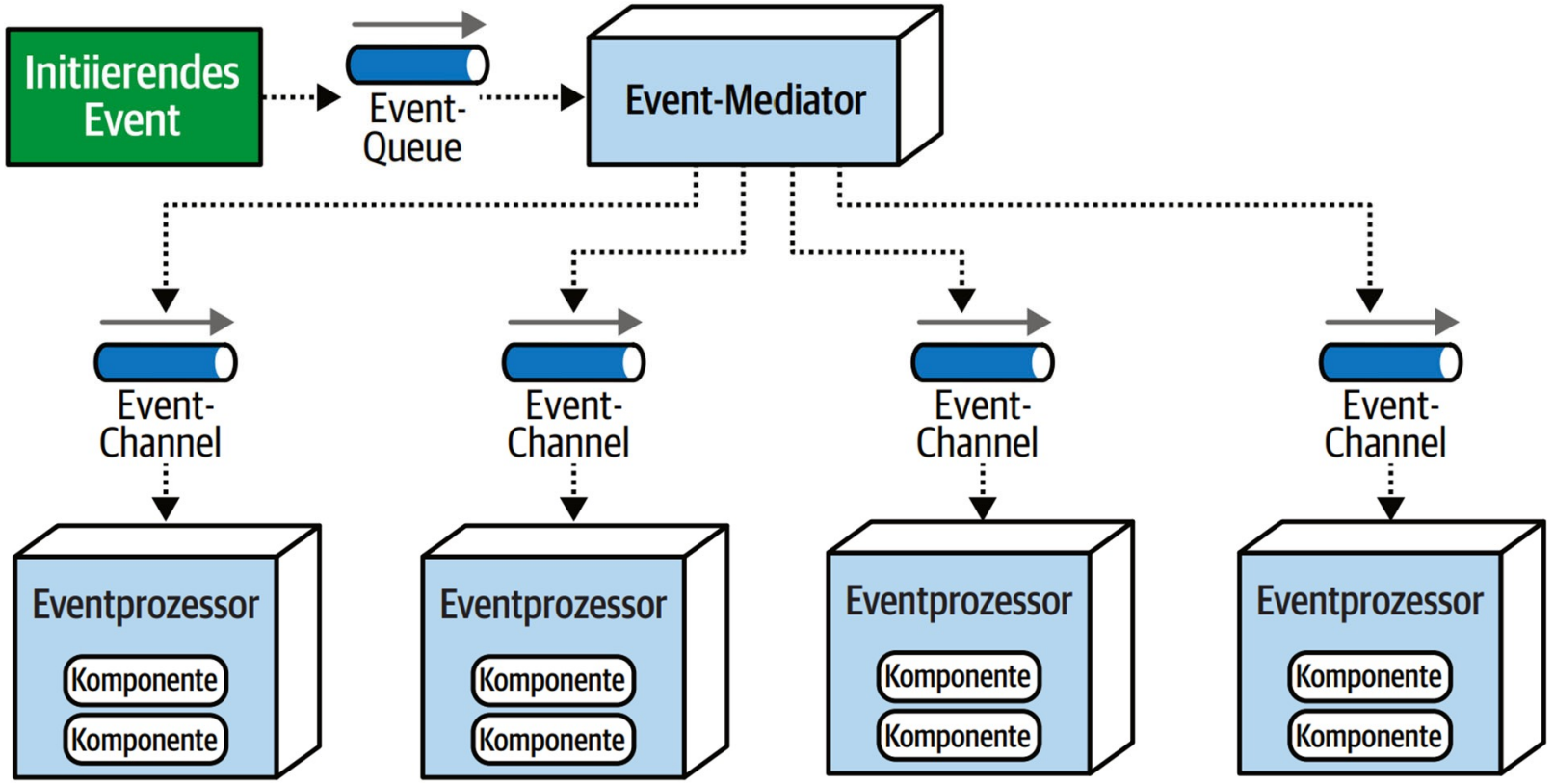
- Highly decoupled event processors
- High scalability
- High responsiveness
- High performance
- High fault tolerance

Disadvantages

- Little control over the workflow
- Complicated error handling
- Poor recoverability
- No possibility for a restart
- Data inconsistency

Event-based Architecture

Mediator Topology - Components and Workflow

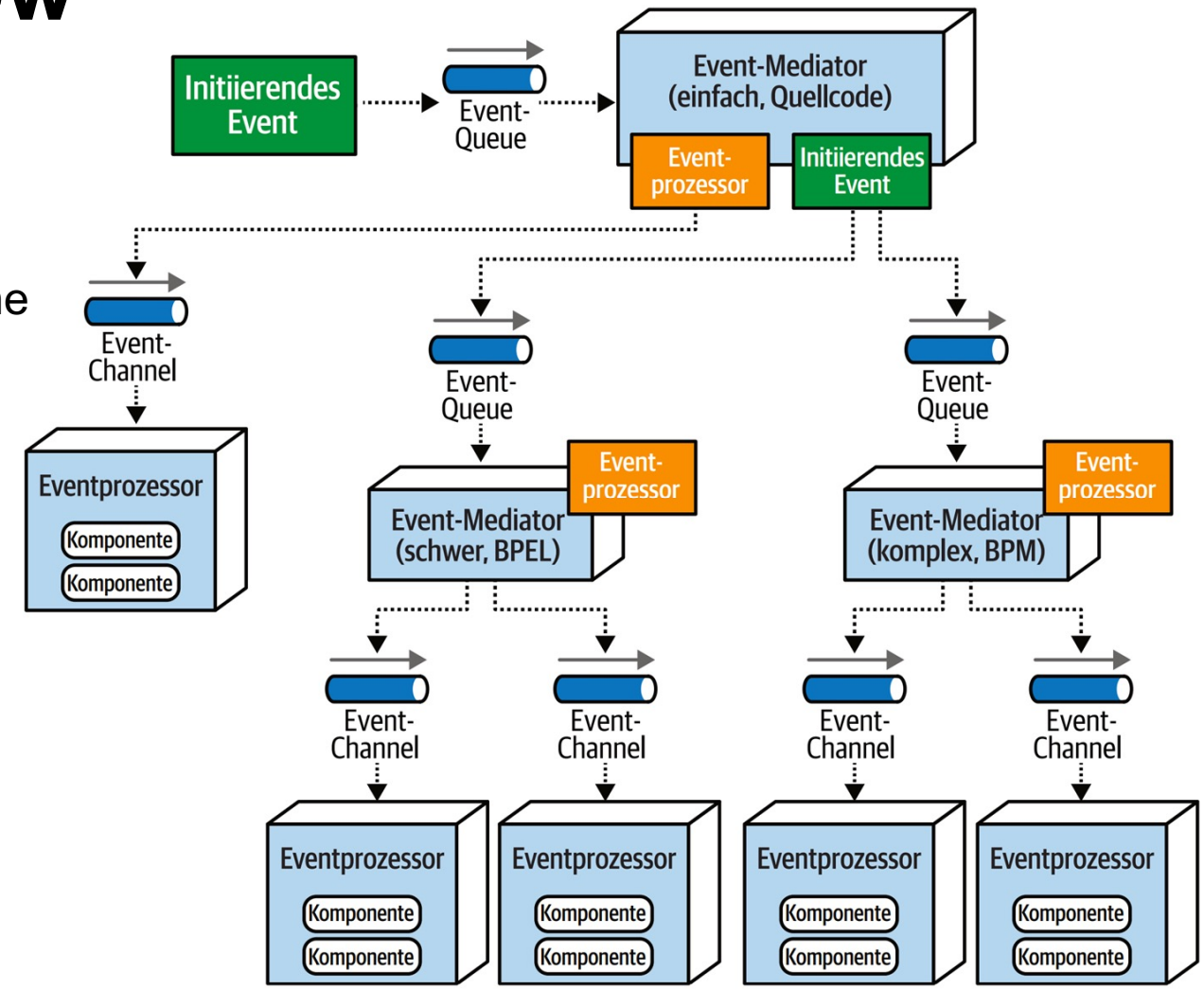


Richards, Ford 2020, S. 190

Event-based Architecture: Mediator Topology

Components and Workflow

- **Initiating Event**
  is passed to an event queue
- **Event-Mediator**
  takes the event from the event queue and perform the orchestration of the workflow
  - generates and sends a processing event to an event channel if the event workflow is simple
  - or if the incoming event is classified as "difficult" or "complex", it forwards the initiating event to an appropriate mediator (BPEL or BPM).
- **Event-Processors**
  listening to channels, process events and notify the mediator if their work is done

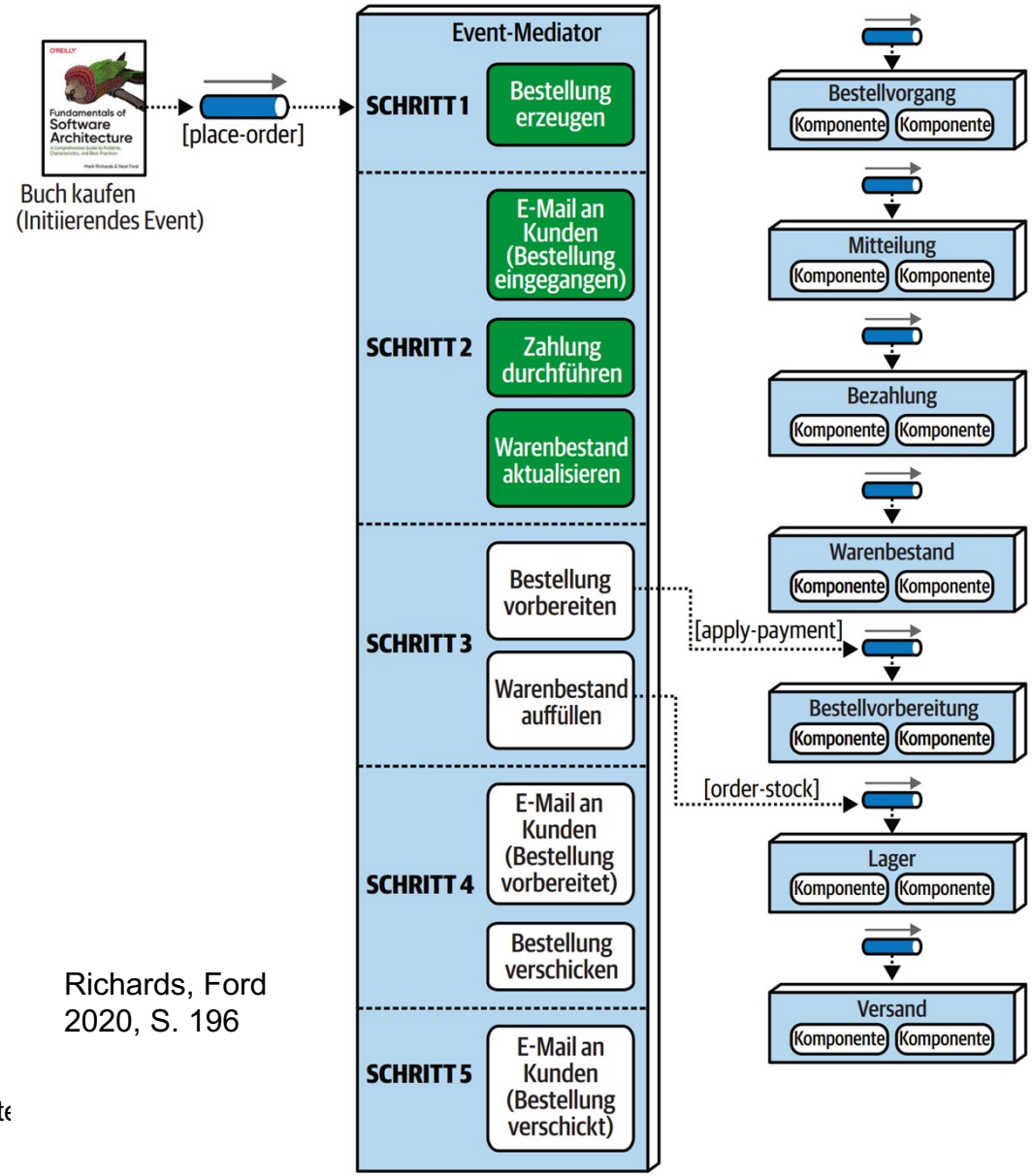


Richards, Ford 2020, S. 192

Event-based Architecture: Mediator Topology

Example: Commerce

- **Process flow using Mediator Pattern**
  - The same **initiating event** [place-order] is passed to the **customer-event-queue** for further processing. The **customer mediator** accepts the initiating event and starts to generate processing events. Finally, it generates a [create-order] event and sends this message to the [order-placement-queue] = step 2.
  - Step 2 simultaneously generates three messages: [email-customer], [apply-payment], [adjust-inventory]
  - Step 3 [prepare-order] must therefore first be fully processed and confirmed before the customer can be informed in step 4 [send-order] that the order is ready for dispatch.
- Be aware: the events within steps 2, 3 and 4 run in parallel, while those from step to step run serially

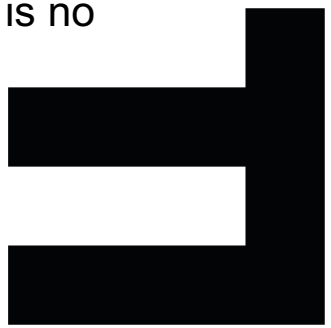


Richards, Ford 2020, S. 196

Event-based Architecture: Mediator Topology

Analysis

- **Process Flow**
  Mediator has the knowledge and control over the workflow.
- **Immediate error handling in combination with a tighter coupling of components**
  Mediator manage the states of the events and thus creates the possibilities for error handling, recoverability and restarting. This leads to a tighter coupling of components.
- **Lower Performance**
  As the mediator controls the event processing, the performance decreases slightly.
- **Processing Events**
  Importance and purpose of processing events changes. In the Mediator topology, the events have the function of commands (things that must happen), in contrast to events in the Broker topology (things that have already happened). In addition, a processing event must be processed in the mediator topology (as a command), whereas it can be ignored in the broker topology (reaction).
- **Workflow difficult to model**
  It is difficult to model dynamic processing of a complex workflow declaratively. For such cases, a combination of broker and mediator topology is usually used. E.g. when an item is no longer in stock.



Event-based Architecture: Mediator Topology

Advantages and disadvantages

Advantages

- Control over the workflow
- Improved Error handling
- Recoverability
- Restart capabilities
- Better data consistency

Disadvantages

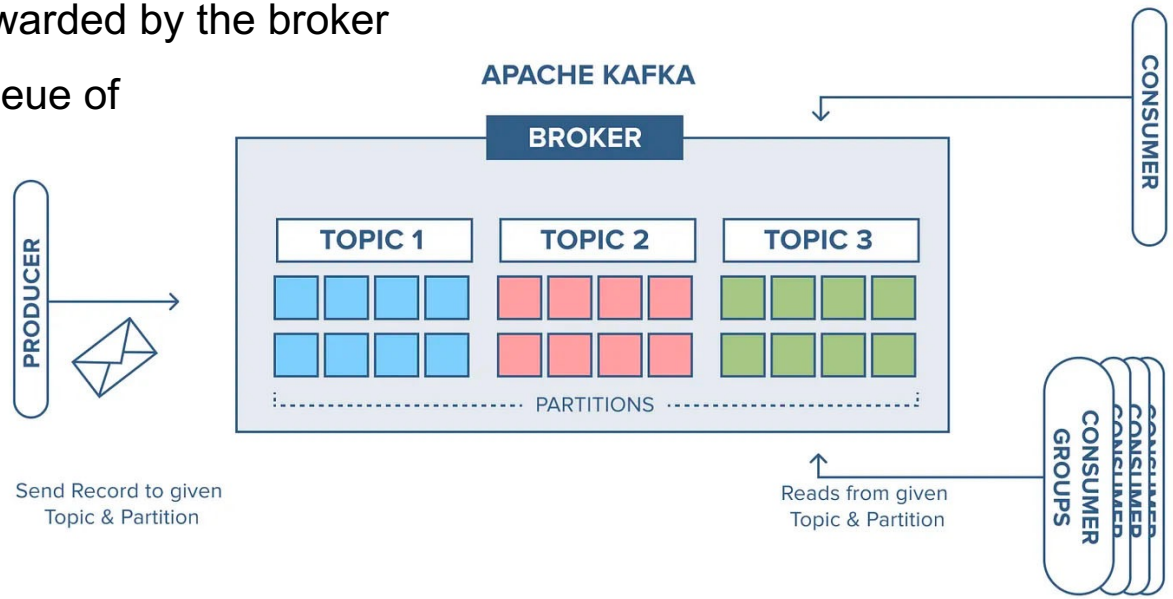
- Higher coupling of event processors
- Lower scalability
- Poorer performance
- Lower fault tolerance
- Modelling of complex workflows

Messaging protocols MQTT and AMQP

Message Broker

Rabbit MQ, Apache Kafka, MuleSoft Anypoint Platform, IBM MQ, AWS MQ, …

- A **message broker** is a **middleware that receives messages** and **forwards** them **to one or more recipients**. The message broker therefore takes on a mediating role by decoupling the sender and receiver of messages from each other, in which communication takes place asynchronously
  - **Producer**: publishes messages that are received and forwarded by the broker
  - **Consumer**: Retrieves the messages from the broker's queue of the broker which are relevant for him.
  - **Topic/Queue***: Holds/saves the received messages according to the FIFO (first-in-first-out) principle. Consumers can register for topics (subscribe).
  - **Exchanger**: Ensures the distribution of the received messages within the broker.



*Be aware: From a technical point of view, queues represent a 1..1 relationship (1 producer->1 consumer), whereby several consumers can register on topics (1..n)

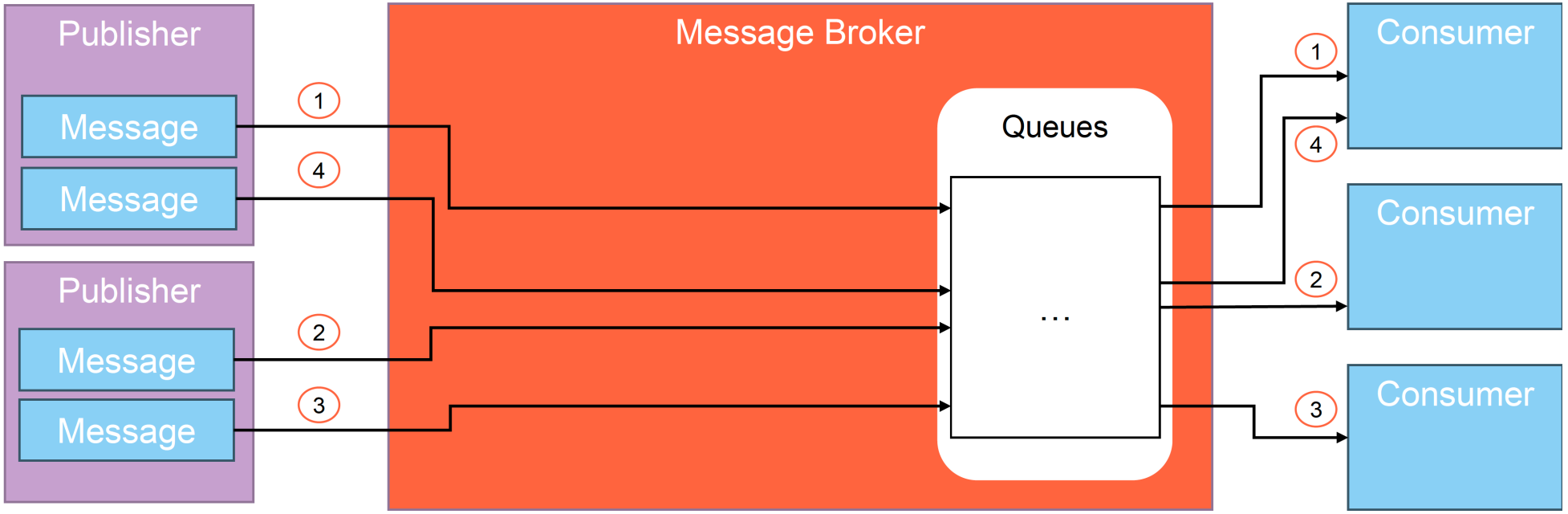
Advanced Message Queueing Protocol (AMQP)

- AMQP is an open standard for passing messages between applications or organizations. It reliably transmits instructions; feeds business processed with the needed information; connects systems.
- The main components of AMQP are (Ackermann 2018, p. 583)
  - **Producer/Publisher**: Component that sends messages to the messaging system
  - **Connection**: TCP connection between an application and the messaging system
  - **Channel**: virtual connection within a connection
  - **Message**: data sent from the producer to the consumer via the messaging system
  - **Exchange**: Message distributor that receives messages from producers and forwards them to queues according to certain criteria
  - **Binding**: connection between queue and exchange
  - **Queue(s)**: Message store that manages messages according to the FIFO principle
  - **Routing Key**: key used by an exchange to decide which queue(s) a message should be forwarded to
  - **Consumer/Subscriber**: Component that processes messages from the messaging system

Advanced Message Queueing Protocol (AMQP)

Queue

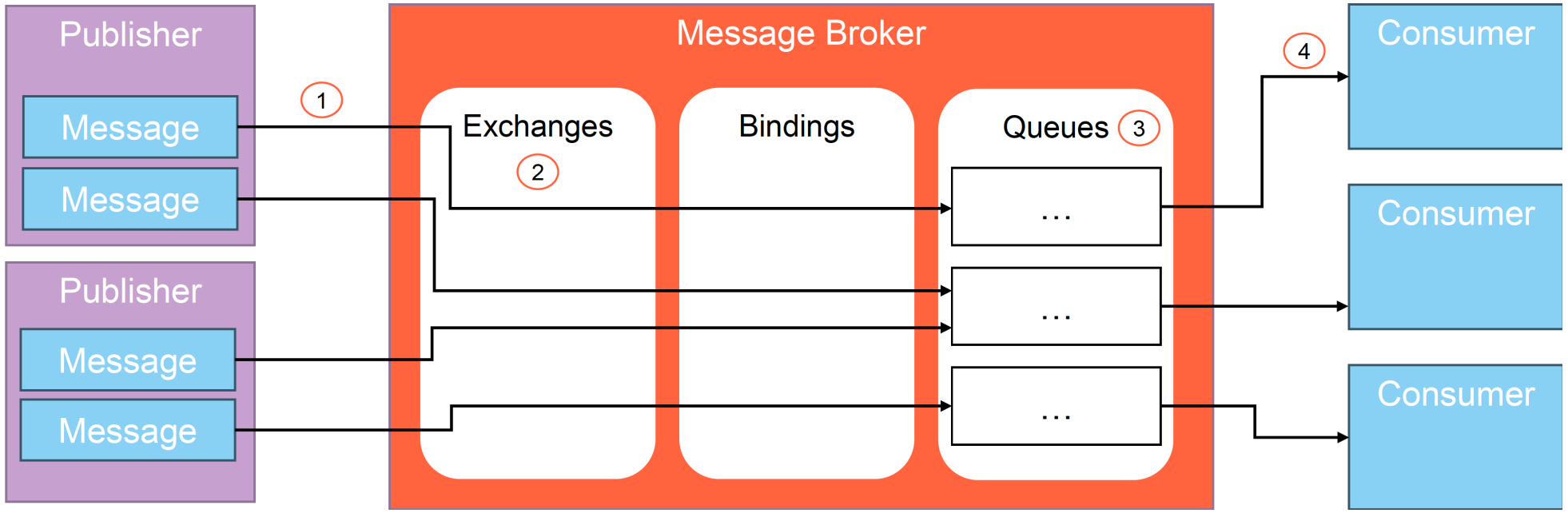
- Both **producers** and **consumers** must have established a **TCP** connection to the messaging system to be able to send or receive messages. **Producers send messages** to queues or to queues via exchanges (see later). Subscribers can retrieve messages from the queues, which are forwarded according to the **FIFO principle**.



Advanced Message Queueing Protocol (AMQP)

Exchanges

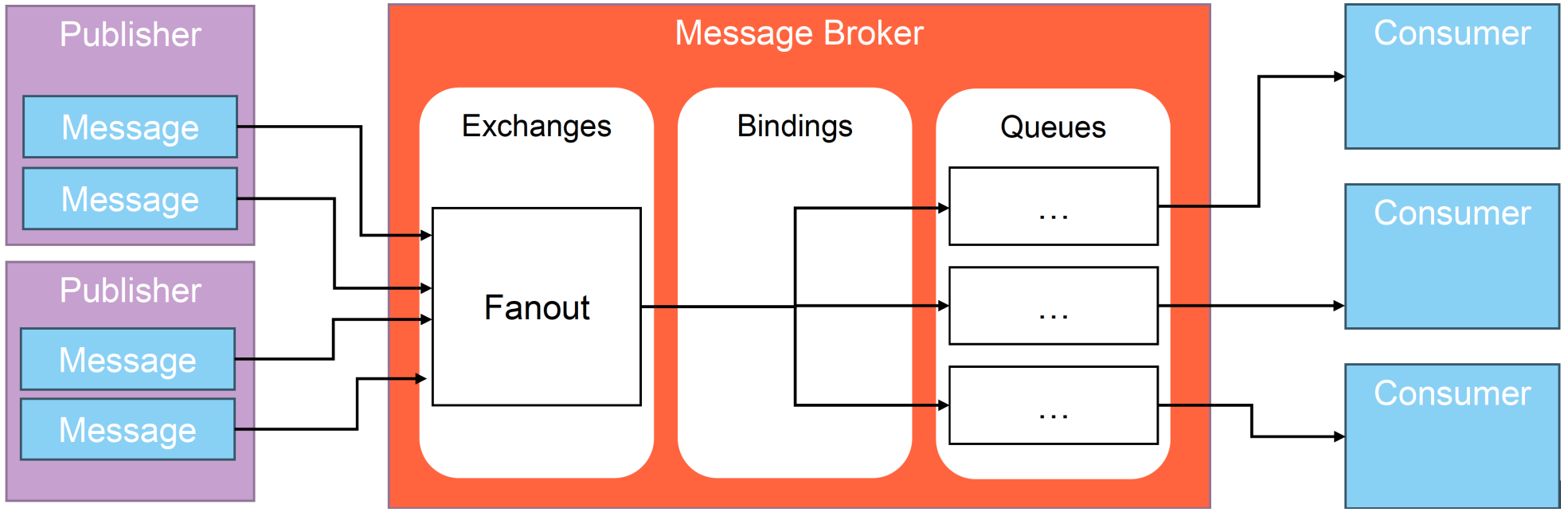
- **Producers** can also send messages to **exchanges**. These are responsible for distributing messages to message **queues** according to certain rules. In other words, queues are bound to exchanges = **message binding. Subscribers** can **retrieve messages** from queues (FIFO).



Advanced Message Queueing Protocol (AMQP)

Exchanges: Fanout

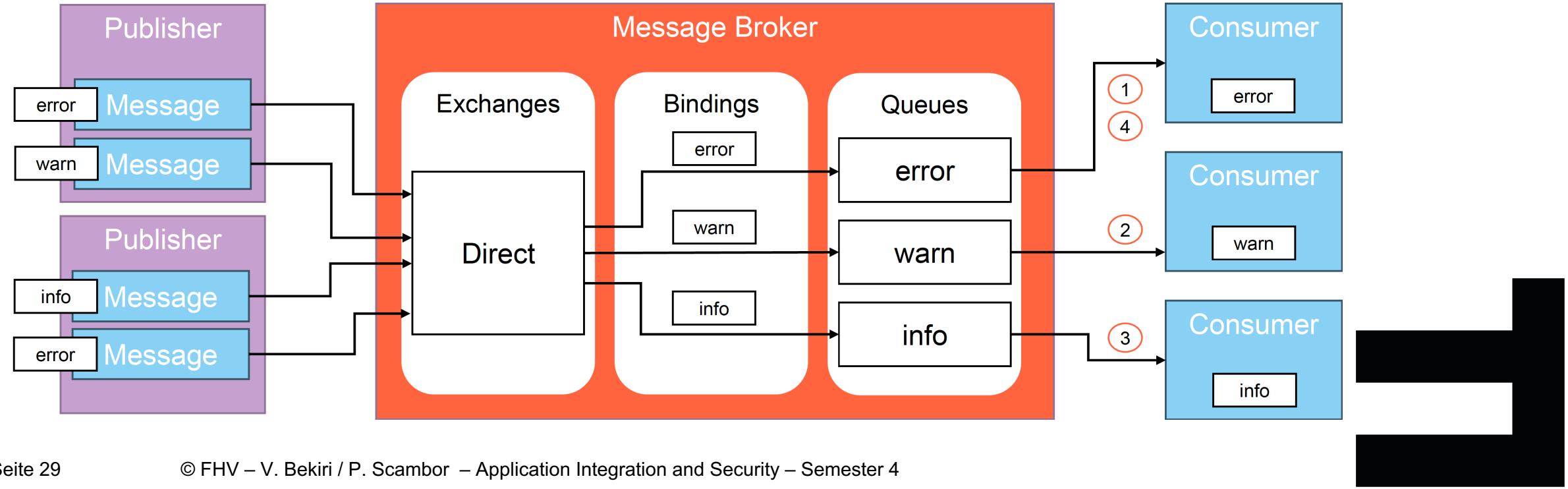
- There are four different **Types** of **Exchanges**: Fanout, Direct, Topic, Header
- **Fanout Exchanges** (publish/subscribe pattern): incoming **messages** are **sent to all queues** bound to this exchange. Consumers register for a specific event, and all are notified when the event occurs.



Advanced Message Queueing Protocol (AMQP)

Exchanges: Direct

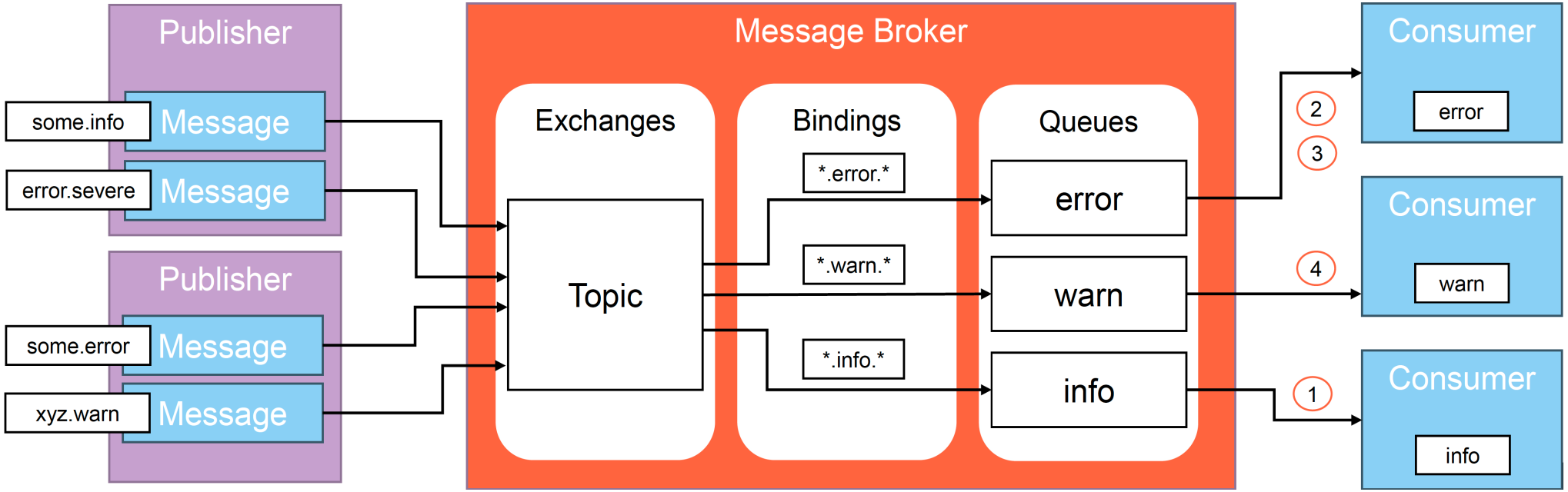
- **Direct Exchanges**: Publisher sends a **routing key** with the message, which is used to decide which queue the message is sent to. Message is **only forwarded** to the consumer if the **routing key** of the message **corresponds** to the routing key defined in the binding between an exchange and a queue.



Advanced Message Queueing Protocol (AMQP)

Exchanges: Topic

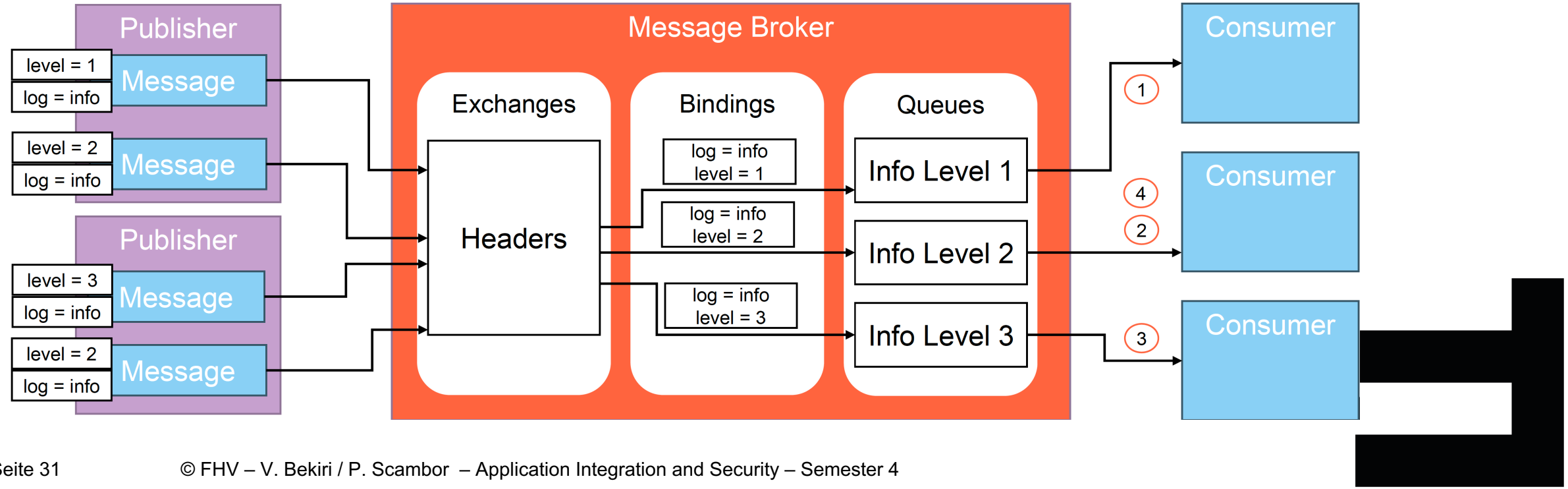
- **Topic Exchanges**: more flexible. The principle is like direct exchanges, as a **routing key** is used to decide which queue a message is distributed to. In contrast to direct exchanges, it is possible to define **wildcards** or **routing patterns**. So, the routing key must not exactly match the key.



Advanced Message Queueing Protocol (AMQP)

Exchanges: Header

- **Header Exchanges**: are the **most flexible** variant. **Routing is** not **based on** a single routing, but on **header information** that can be sent with a message. They **allow several routing keys** to be used and are applied when not all parameters can be integrated in one routing key.



Message Queuing Telemetry Transport (MQTT)

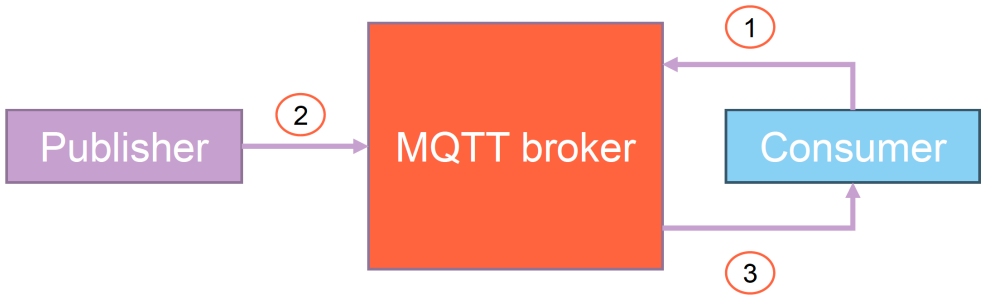
- **MQTT** is a lightweight, publish-subscribe based messaging protocol designed for resource-constrained devices and low-bandwidth, high-latency, or unreliable networks. It is **widely used** in **Internet of Things (IoT)** applications, providing efficient communication between sensors, actuators, and other devices. E.g. Smart-Home/City, eHealth, Industry 4.0, …
- MQTT is characterized by the following features (Ackermann 2018, p. 598)
  - **Lightweight**: even devices with limited resources can easily use MQTT (e.g. Arduino, ...)
  - **Supports various Service Qualities**: to ensure transmission even in unstable networks (QoS)
  - **Service-Efficient transmission**: use of a binary format to enable communication with low bandwidths
  - **Session awareness**: messages are temporarily stored if a connection is lost and sent later
  - **Data agnostic**: supports the transmission of different formats: Text, binary or object messages

https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

Message Queuing Telemetry Transport (MQTT)

- **MQTT** implements the Publish-Subscribe Message Pattern: **Publishers** send messages, one or more subscribers receive messages. However, publishers and subscribers do not communicate directly with each other, but are decoupled from each other via an **MQTT broker**. The broker uses so-called topics to ensure that messages reach the right recipient(s).
- **Topics** are simple character strings: that can have a hierarchical structure and use topic levels ("home/garage/temperatureSensor"). Subscribers register for one or more topics.
- Topics can also contain **wildcards**: allowing subscribers to flexibly define which topics they are interested in. E.g. all sensor data "home/" or only in light sensors "home/#/lightSensor".
- **Workflow**

  1. **Consumer** subscribes to a topic
  2. **Publisher** publish a message for a topic
  3. **MQTT broker** forwards the message to all consumers who have subscribed to this topic



Message Queuing Telemetry Transport (MQTT)

- **Quality of Service** determines the transmission quality of messages respectively how often a message is sent. To accomplish this, the consumer or publisher must confirm receipt of message.
- **Service levels**
  - **Level 0:** no delivery guarantee: ”Fire and forget”
  - **Level 1:** delivery guaranteed, but duplicates possible, number of deliveries at least once
  - **Level 3:** delivery guaranteed, no duplicates, number of deliveries exactly once
- Further features of MQTT
  - **Last Will and Treatment:** if a client unexpectedly disconnects, it can send a message (last will) to other clients
  - **Retained Messages:** enables subscribers who register for the topic to receive the last measured value. This ensures that no message is missed.
  - **Persistent Sessions:** if a client loses its connection and reconnects, it will receive all missed messages. This is particularly useful in the event of frequent disconnections.

Event-based Communication using RabbitMQ in .NET 8

Event-based Communication using RabbitMQ

Code example: AMQP Console Application

- The implementation for using a message broker like RabbitMQ for AMQP is relatively straightforward (https://www.rabbitmq.com/tutorials/tutorial-one-dotnet). However, all technical details (topology, protocol) as well as the architecture and the message structure and its structure must be conceptualized in advance.
- **Steps to develop** an **application** (see code details on the links)
  - **Register a free CloudAMQP** (https://www.cloudamqp.com) or use another Message broker
  - Create two Console Applications (Sender and Receiver)
  - Install RabbitMQ package (NuGet or PM Console): `RabbitMQ.Client`
  - Implement Sender using https://www.rabbitmq.com/tutorials/tutorial-one-dotnet#sending
  - Implement Receiver using https://www.rabbitmq.com/tutorials/tutorial-one-dotnet#receiving

Event-based Communication using RabbitMQ

Code example: AMQP Sender

```
// RabbitMQ URI Spec: https://www.rabbitmq.com/docs/uri-spec
string url = "amqps://user:password@host/vhost";
var factory = new ConnectionFactory { Uri = new Uri(url) };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.QueueDeclare(
  queue: "hello", durable: false, exclusive: false,
  autoDelete: false, arguments: null
);

string message = "Hello World!";

while (!string.IsNullOrEmpty(message)) {
    var body = Encoding.UTF8.GetBytes(message);
    channel.BasicPublish(
      exchange: "", routingKey: "hello", basicProperties: null, body: body
    );
    Console.WriteLine(" [x] Sent: {0}", message);
    Console.Write("Next Message (press Enter to exit): ");
    message = Console.ReadLine();
}
```

Event-based Communication using RabbitMQ

Code example: AMQP Receiver

```
// RabbitMQ URI Spec: https://www.rabbitmq.com/docs/uri-spec
string url = "amqps://user:password@host/vhost";
var factory = new ConnectionFactory { Uri = new Uri(url) };

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.QueueDeclare(
  queue: "hello", durable: false, exclusive: false,
  autoDelete: false, arguments: null
);

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) => {
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine(" [x] Received {0}", message);
};

channel.BasicConsume(queue: "hello", autoAck: true, consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");
Console.ReadLine();
```

Event-based Communication using RabbitMQ

Code example: MQTT Console Application

- The implementation for using a message broker like RabbitMQ for MQTT is straightforward. As MQTT is a lightweight protocol there is no configuration to be done upfront unlike for AMQP.
- **Steps to develop** an **application**
  - **Register a free CloudAMQP** (https://www.cloudamqp.com) or use another Message broker
  - Create two Console Applications (Publisher and Subscriber)
  - Install MQTTnet package (NuGet or PM Console): `MQTTnet`
  - Implement Publisher (next slide)
  - Implement Subscriber (slide after next one)

Event-based Communication using RabbitMQ

Code example: MQTT Publisher

```
var mqttFactory = new MqttClientFactory();

var mqttClient = mqttFactory.CreateMqttClient()
var mqttClientOptions = new MqttClientOptionsBuilder()
    .WithTcpServer("hawk.rmq.cloudamqp.com", 8883)
    .WithProtocolType(ProtocolType.Tcp)
    .WithTlsOptions(o => o.UseTls())
    .WithCredentials("username","password")
    .Build();

await mqttClient.ConnectAsync(mqttClientOptions, CancellationToken.None);

var applicationMessage = new MqttApplicationMessageBuilder()
    .WithTopic("samples/temperature/living_room")
    .WithPayload("19.5")
    .Build();

await mqttClient.PublishAsync(applicationMessage, CancellationToken.None);
await mqttClient.DisconnectAsync();

Console.WriteLine("MQTT application message is published.");
```

Event-based Communication using RabbitMQ

Code example: MQTT Subscriber

```
var mqttFactory = new MqttClientFactory();
var mqttClient = mqttFactory.CreateMqttClient();

var mqttClientOptions = new MqttClientOptionsBuilder()
    .WithTcpServer("hawk.rmq.cloudamqp.com", 8883)
    .WithProtocolType(ProtocolType.Tcp).WithTlsOptions(o => o.UseTls())
    .WithCredentials("username","password").Build();

mqttClient.ApplicationMessageReceivedAsync += e =>
{
    Console.WriteLine("_____________________________________");
    Console.WriteLine("Received message");
    Console.WriteLine("Topic: " + e.ApplicationMessage.Topic);
    Console.WriteLine("Message: " + Encoding.UTF8.GetString(
      e.ApplicationMessage.Payload
    ));

    return Task.CompletedTask;
};

await mqttClient.ConnectAsync(mqttClientOptions, CancellationToken.None);

var mqttSubscribeOptions = mqttFactory
    .CreateSubscribeOptionsBuilder()
    .WithTopicFilter("samples/temperature/living_room").Build();

await mqttClient.SubscribeAsync(mqttSubscribeOptions, CancellationToken.None);

Console.WriteLine("MQTT client subscribed to topic.");
Console.WriteLine("Press enter to exit.");
Console.ReadLine();
```

Literatur

- Ackermann, Philip (2018): Professionell entwickeln mit JavaScript: Design, Patterns, Praxistipps. 2., aktualisierte und erweiterte Auflage. Rheinwerk Verlag, Bonn.
- Ford, Neal; Richards, Mark (2020): Fundamentals of Software Architecture: An Engineering Approach. A Comprehensive Guide to Patterns, Characteristics, and Best Practices. Illustrated Edition. O'Reilly UK Ltd.
- Starke, Gernot (2020): Effektive Softwarearchitekturen: Ein praktischer Leitfaden. 9., überarbeitete Edition. Carl Hanser Verlag GmbH & Co. KG.
- Kratze, Nane (2022): Cloud-Native Computing: Software Engineering von Diensten und Applikationen für die Cloud. Carl Hanser Verlag, München.

That‘s it ☺
…for now