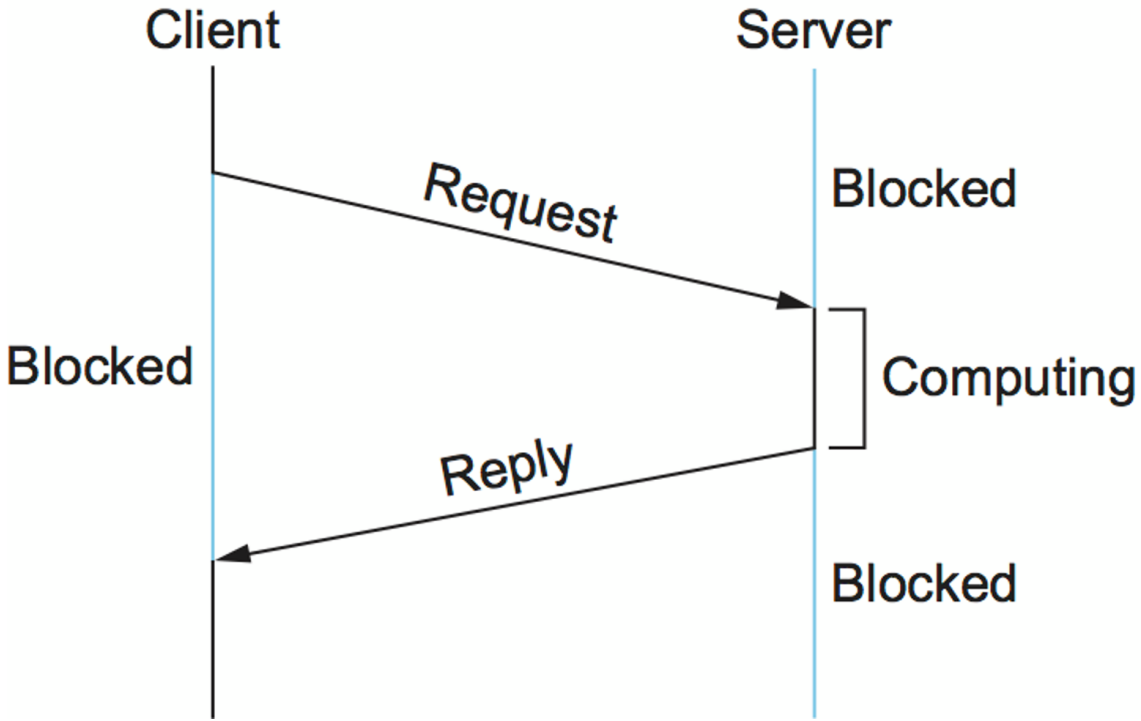
Communication Patterns

BIF5-SWKOM

Communication

FH University of Applied Sciences TECHNIKUM WIEN

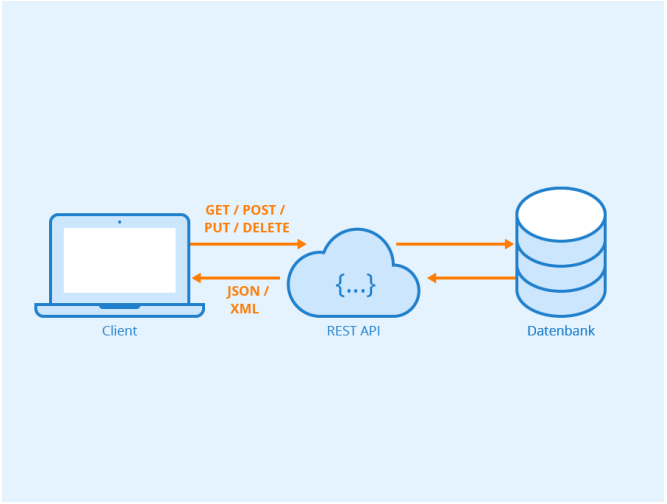
Communication 1 – Remote Procedure Call (RPC)



Communication 2 - REST

- request/response style
- Client/server
- Stateless
- Core abstraction of REST is a resource

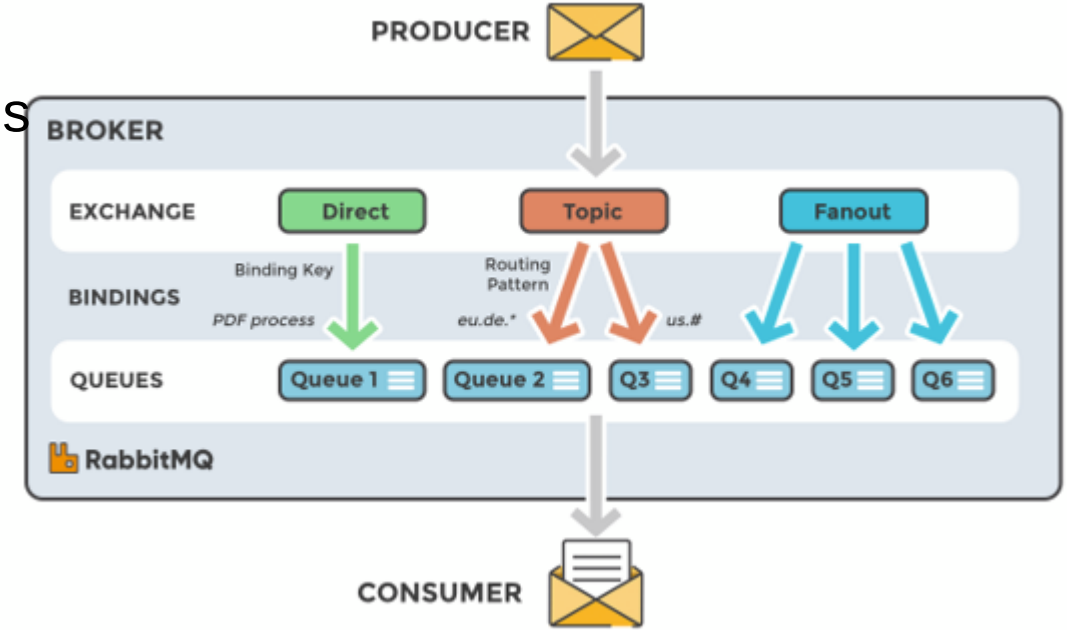
…see next chapter



Communication 3 – Broker (RabbitMQ)

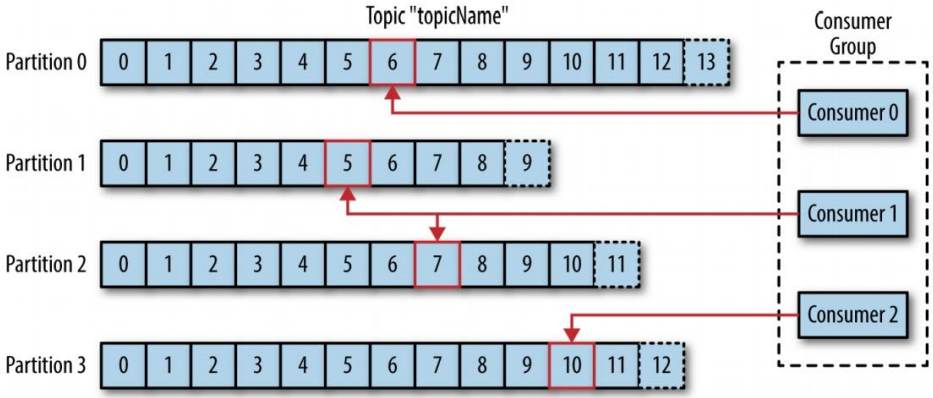
- Broker acts as a kind of intermediary → thus decoupling

- Producer: generates msgs
- Consumer: consumes msgs
- Exchanges:
  - Direct
  - Topic
  - Fanout
- Queue



Communication 3 – Broker (Kafka)

- Message remains after "consume".
  - Consuming means with Kafka: Offset (similarly File-Descriptor) will be pushed further.
  - E.g. reading a topic is reading like from a file with events
- With Kafka we talk about topics (instead of queues)

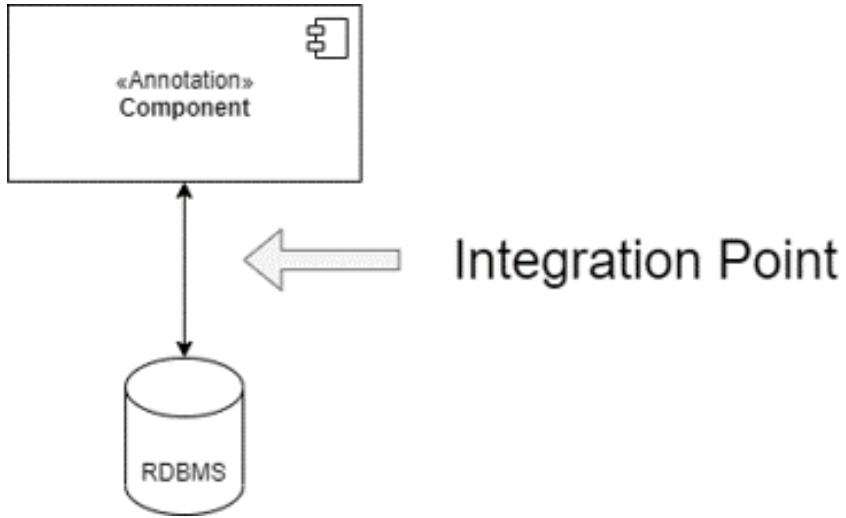


Integration Points

External communications should be viewed critically, they can fail.

Countermeasures:

- Circuit Breaker
- Timeouts
- Decoupling Middleware
- Handshaking



Testing & Integration Phase

Further Reading:

- **Chaosmonkey**: https://github.com/Netflix/chaosmonkey
- **Mock vs Stub**: Mock Object at XUnitPatterns.com and Test Stub at XUnitPatterns.com respectively
- **TestPyramid** (martinfowler.com)
- **Pact** | Microservices testing made easy
- **Gatling** - Professional Load Testing Tool

Maintenance

A distinction is made between two terms:

- Maintenance
- Maintainability

How to design maintainability for components?

- Reduce size
- Increasing cohesion
- Reducing coupling

Event-Driven Architecture

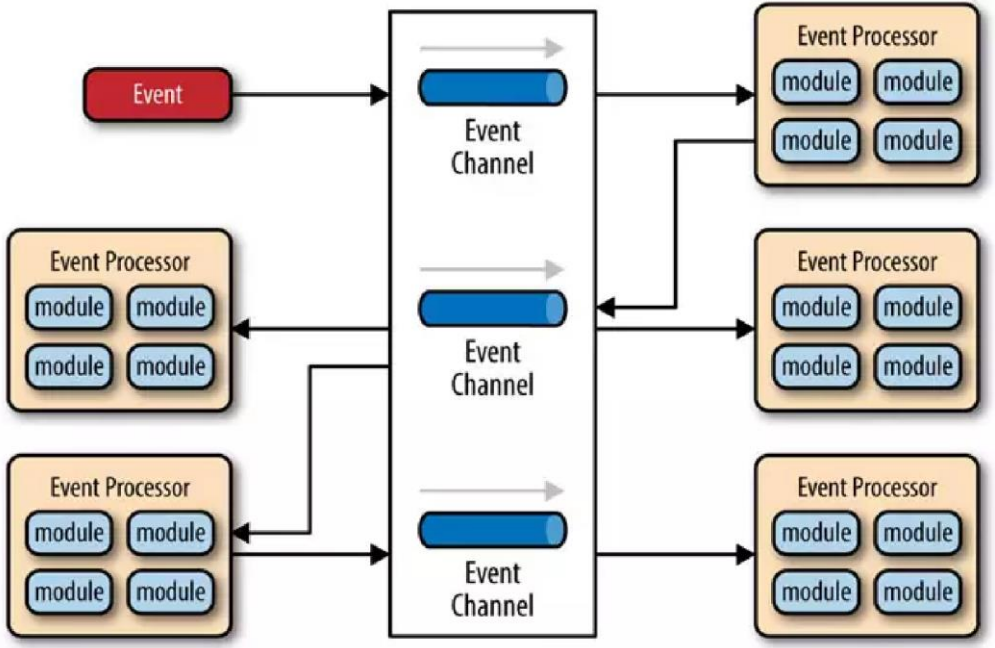
FH University of Applied Sciences TECHNIKUM WIEN

Event-Driven Architecture

- Event-Driven Architecture emphasizes the production, detection, and consumption of events to enable loose coupling and asynchronous communication between components (no direct communication between processes)

Key Elements:

- Events: occurrences or changes, trigger actions or reactions
- Producers: Components that generate events and publish them
- Consumers: Components that subscribe to events and react to them by performing specific actions or processing the event data.
- Event Bus: acts as a central hub or message broker
- Event Handlers: responsible for processing and reacting to specific events they have subscribed to.



Event-Driven Architecture

Benefits:

- Flexibility and Agility
- Scalability
- Modularity and Reusability
- Event Traceability and Monitoring

Considerations:

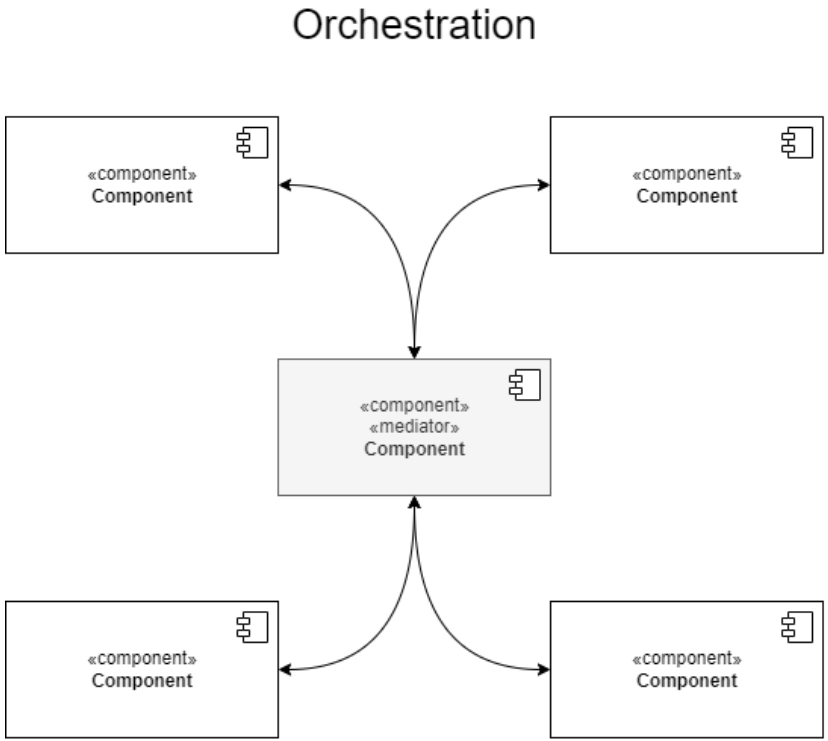
- Event Schema and Versioning
- Eventual Consistency
- Event Message Reliability

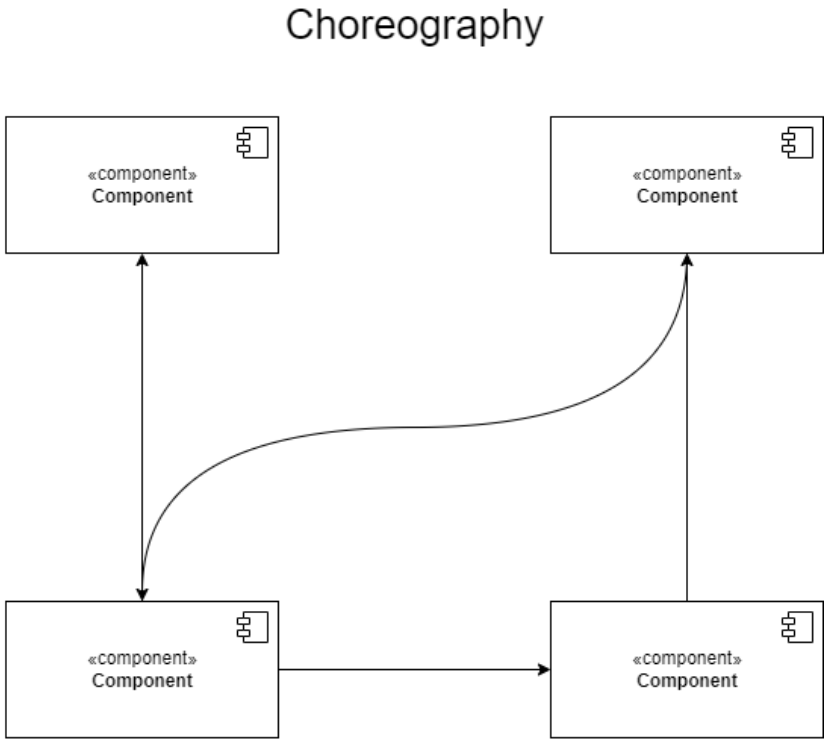
The event-driven architecture pattern is commonly used in systems that require real-time processing, event-driven workflows, and reactive behavior. It is widely employed in domains such as event-driven microservices, event sourcing, and real-time analytics.

Event-Driven Architecture

- Events are key point
- Events are objects, which represent something that had happened in the past.
- So there are immutable - because you cant change the past.
- They common styles are Orchestration (performed by a Mediator) and Choreography

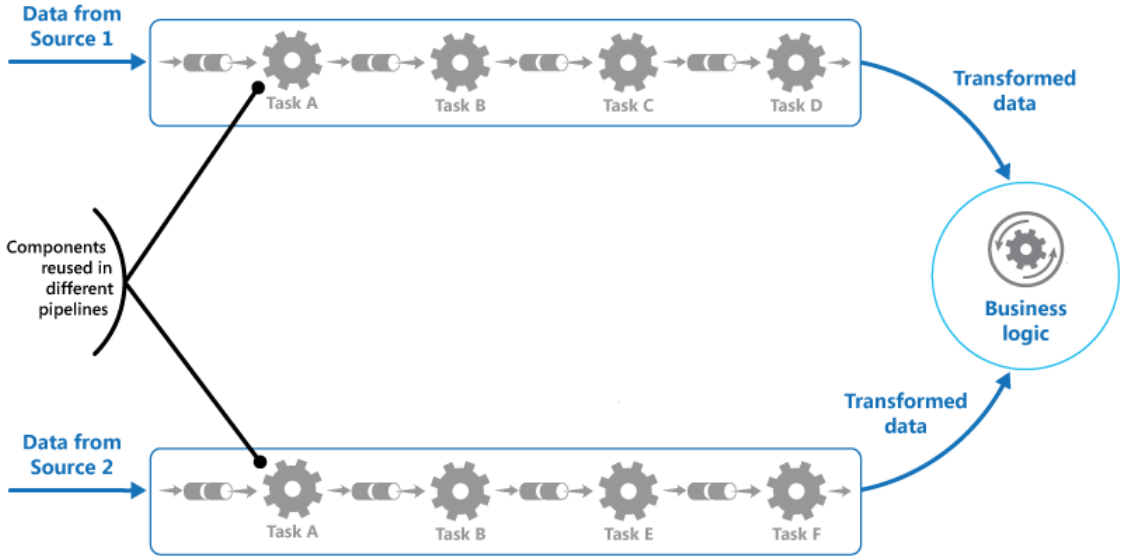
Event-Driven Architecture





Pipes-and Filters

- A variety of tasks of varying complexity needed to be performed
- Break down the processing into separate components, each performing a single task
- Combine tasks

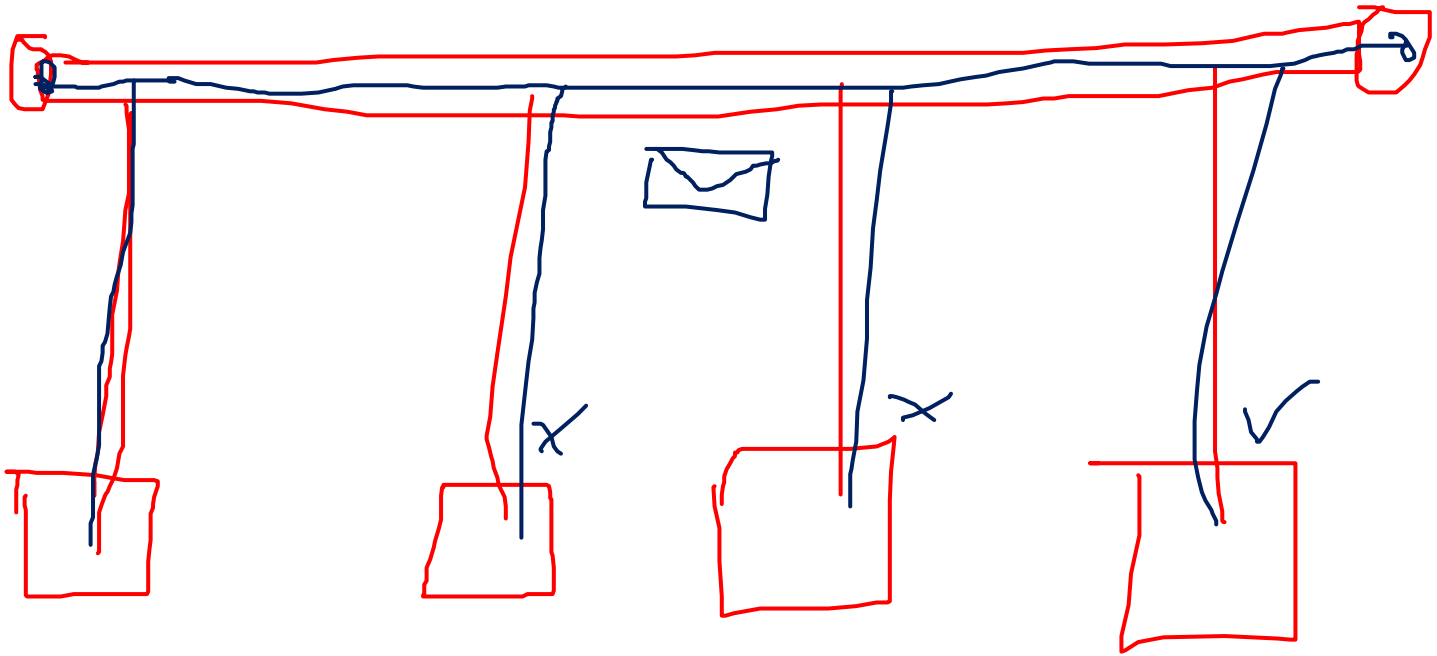


Message-Bus

FH University of Applied Sciences TECHNIKUM WIEN

Message Bus Architectural Style

- Ability to receive & send messages over one or more channels
- Applications don‘t need to know each other
- Pluggable architecture: Systems attach and detach
- Asynchronously by design
- Common Bus = Router, Publish/Subscribe patterns



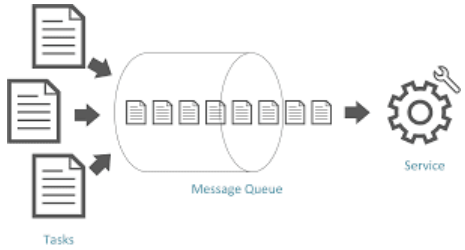
Message Bus Architectural Style

- Variations:
  - Intra-System Message Bus (e.g. CAN)
  - Enterprise Service Bus (ESB)
  - Internet Service Bus (ISB)

- Pros:
  - Extensibility
  - Low complexity
  - Flexibility
  - Loose Coupling
  - Scalability
  - Application simplicity

Message Queue

- sequential list of items that are waiting to be handled
- once an item is executed it can send a confirmation response and then is deleted
- a message is the data that is sent between the sender application and the receiver application

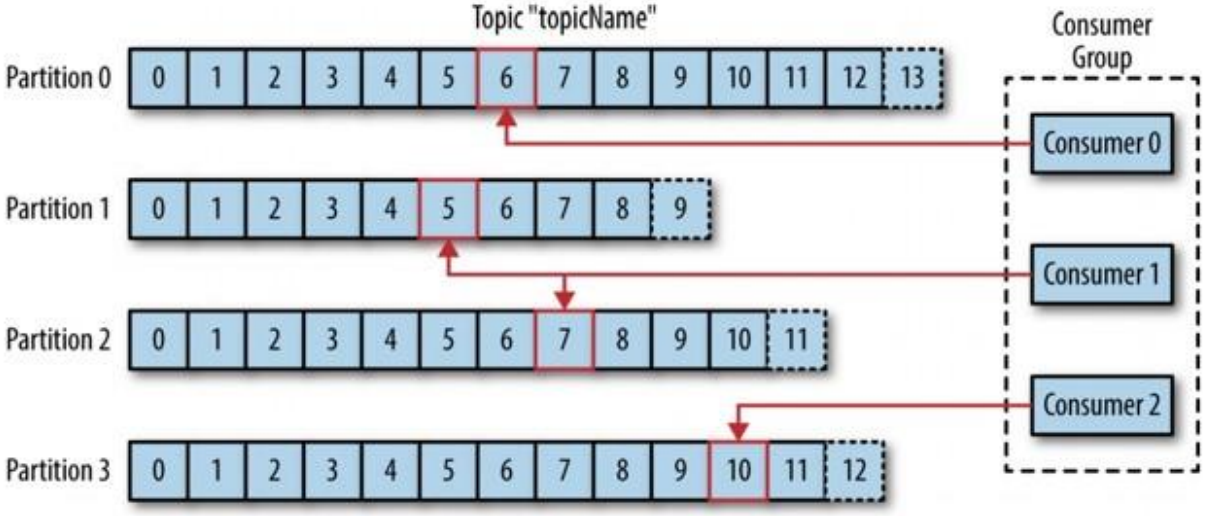


Message Brokers

- Message brokers are frequently used nowadays
- they decouple components (in terms of time, among other things).
  - I.e. sender and receiver do not have to be available or ready at the same time.
  - I.e. the receiver can control its own pace of processing and does not really have to generate back-pressure.
- In addition, brokers can perform other tasks such as filtering and transforming (simply or with the help of multiple information sources).

Kafka

- Messages organized into topics.
- Topics can be "orders", "temperature_values" and so on.
- Separate topics into several partitions. Partitions help you the scale - e.g. move them onto a other separate host.
- The order of messages is only guaranteed inside one partition.

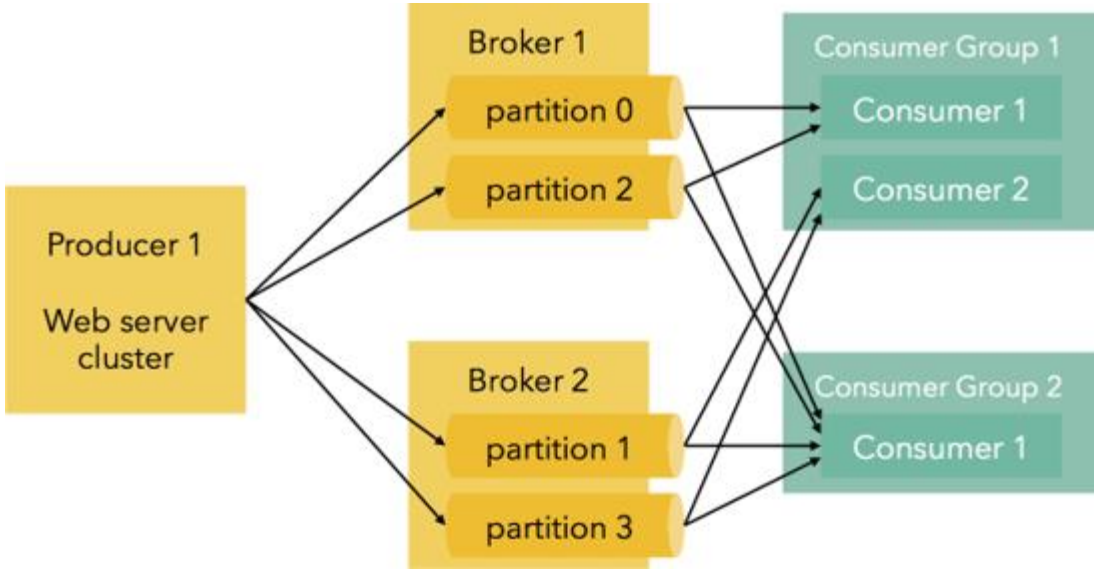


Kafka

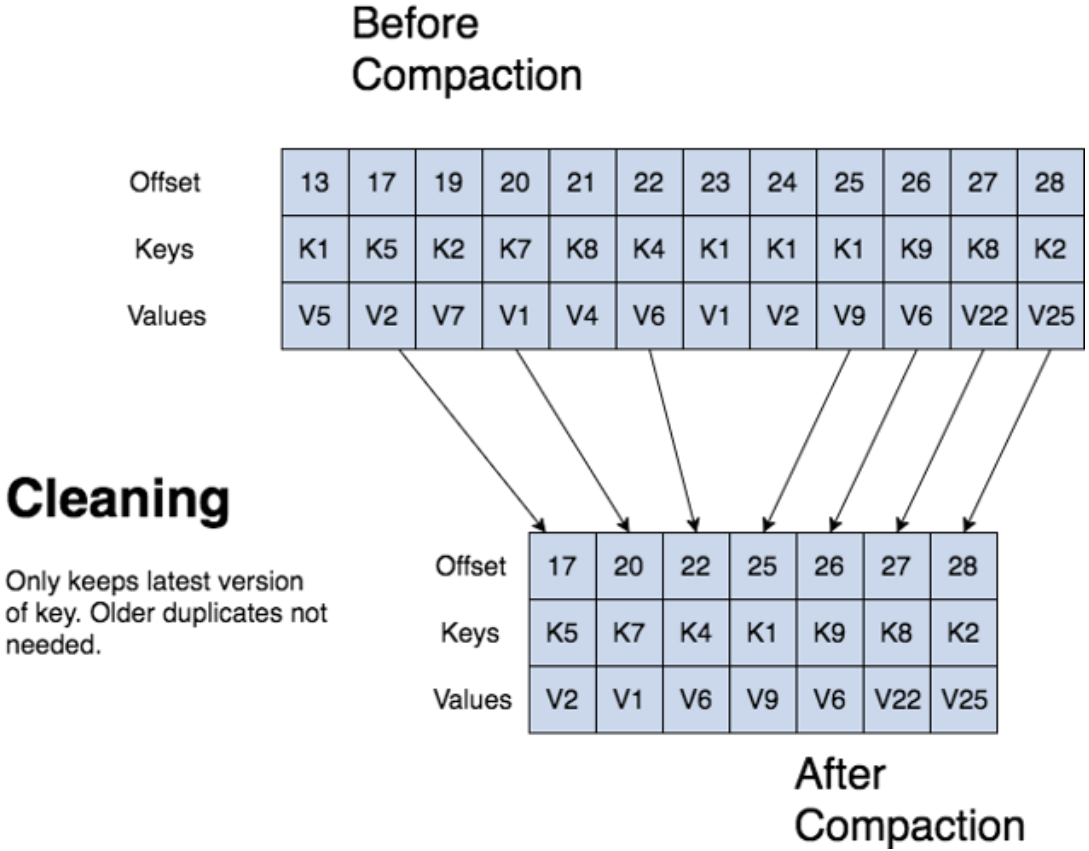
- If you consume a message, Kafka moves to the next one and remembers the position (important for if the consumer crashes). But: the message is not deleted after consumption
- Messages are cleaned up by time (aka retention - e.g. after 24 hours) or by key (aka topic compaction - messages with the same key will be cleaned up - so in the end there is only one message left. Remark: There can be also 2-3 messages left with the same key - so there is no guarantee, that a new consumer will see exactly one message with key "X")

Kafka

- Kafka allows the consumer to run on multiple hosts to scale. They just have to join the same consumer group and Kafka organizes, that partitions X is only consumed by one consumer inside a group.



Kafka – log compaction



RabbitMQ

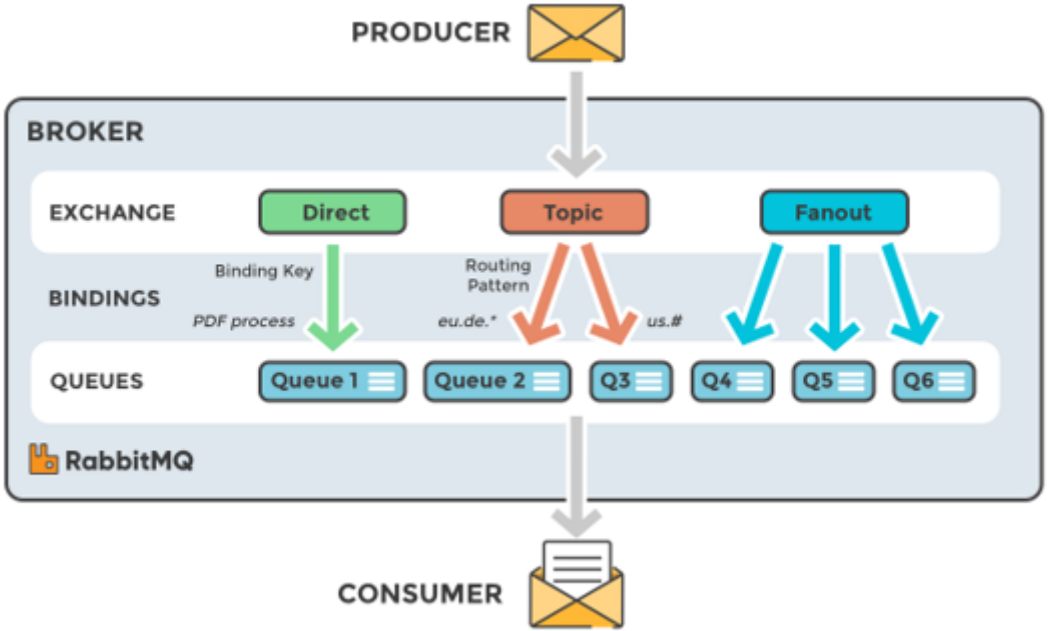
- open-source message broker
- supports several messaging protocols
- libraries for most modern languages
- lightweight
- used by many companies

RabbitMQ

RabbitMQ is another popular broker, mainly used, if the non-functional requirements are not so challenging.

- **Producer**: Generates message - Consumer: Consumes message
- **Exchanges**: Serves only for the forwarding of messages - thus no messages are remembered.
- **Direct**: Forwards message to a specific queue.
- **Topic**: Each message can have a routing key Forwarding based on the routing key by pattern matching: somekey.*.foo.*.bar.#
- **Fanout**: Forward to any connected queue

RabbitMQ



- If the consumer reads the message from the queue, it disappears.

Kafka vs. RabbitMQ

Situation	Kafka	RabbitMQ
Message Konsumieren	Position wird weiter geschoben (Nachricht bleibt aber weiterhin da)	Nachricht verschwindet nach dem ACK
Message Routing	Kafka Streams (externe App)	Build-In: Topic Exchange mit Routing-Key
Admin UI	Einige 3rd Party – like „UI for Apache Kafka“	Build-In (sehr gut zum Debuggen)
Komplexität Verständnis	Schwer	Leicht
Performance	Sehr Hoch (Append-Log)	Mittel
Message Priority	Nein	Ja
Message Größe	Idealerweise 1MB	Idealerweise bis zu 128MB
Protokoll	Binär	Text-Basiert (Level-7 Firewall?)