Hochschule **Rosenheim**
University of Applied Sciences

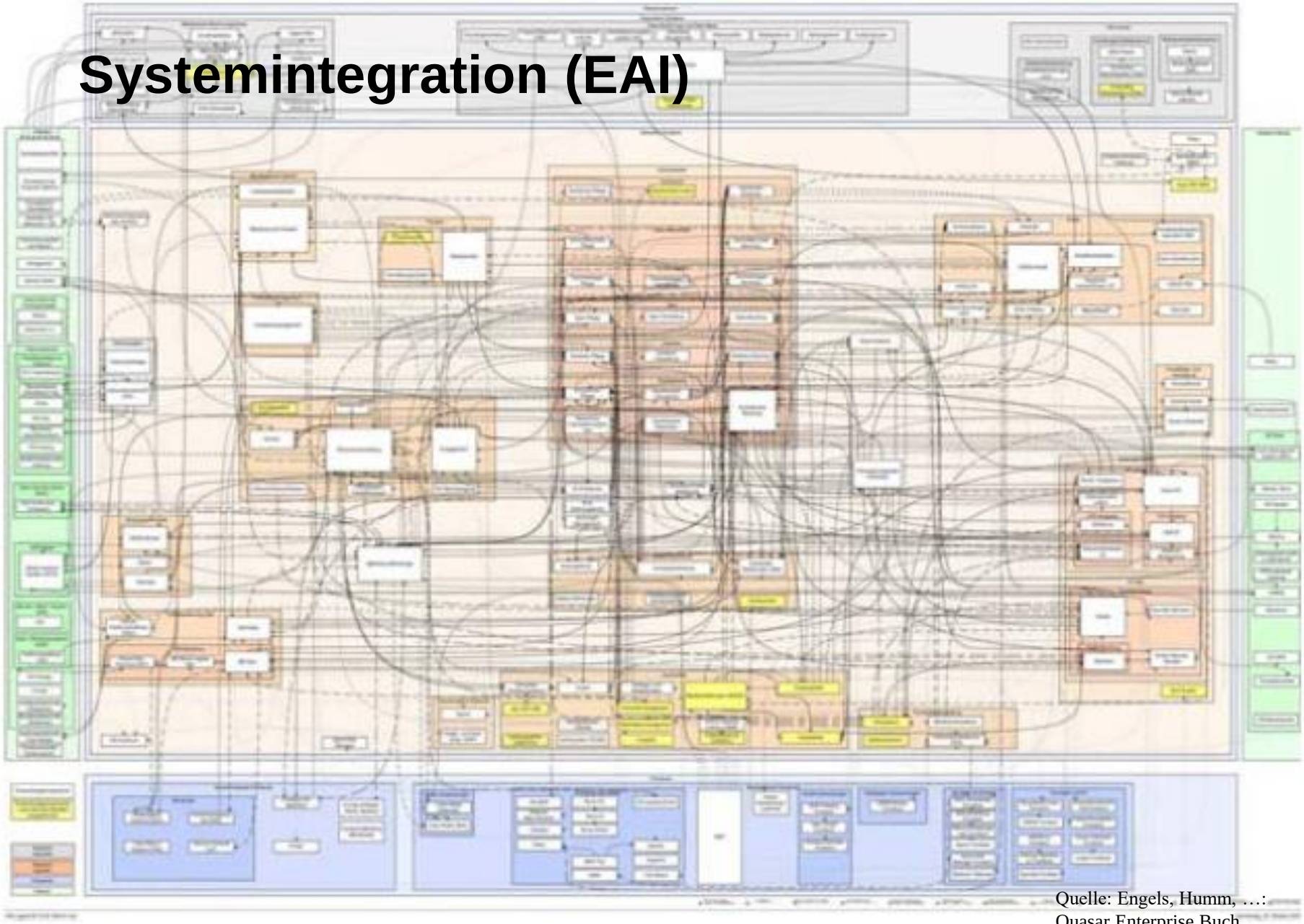
Verteilte Verarbeitung

Kapitel 09

Messaging

Asynchroner Nachrichtenaustausch

Systemintegration (EAI)



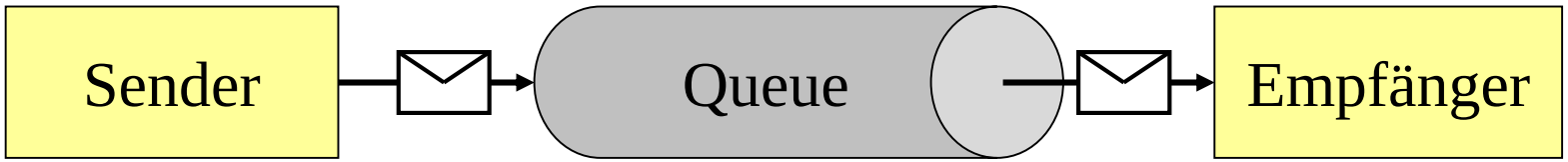
Quelle: Engels, Humm, …: Quasar Enterprise Buch, dpunkt, 2008

Queues und was sie damit machen können

Message Oriented Middleware

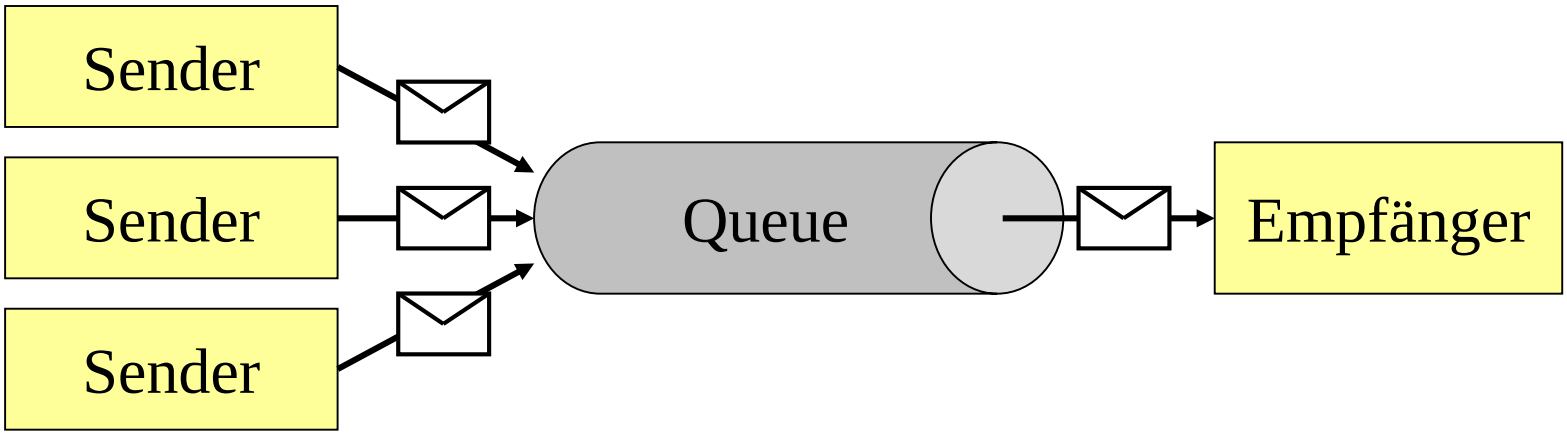
Every DAD needs a MOM

- MOM = Message Oriented Middleware
- Synchrone und ***Asynchrone*** Kommunikation über Nachrichten
  - Asynchron: Laufzeitentkopplung von Sender / Empfänger
  - Nachrichten werden in Warteschlange (Queue) zwischen gesp.
  - Warteschlangen von Infrastruktur/Betriebssystem bereitgestellt
- Warteschlange = FIFO-Prinzip
  - Sender schreibt in Warteschlange
  - Empfänger liest aus Warteschlange
  - Reihenfolge der Nachrichten bleibt erhalten



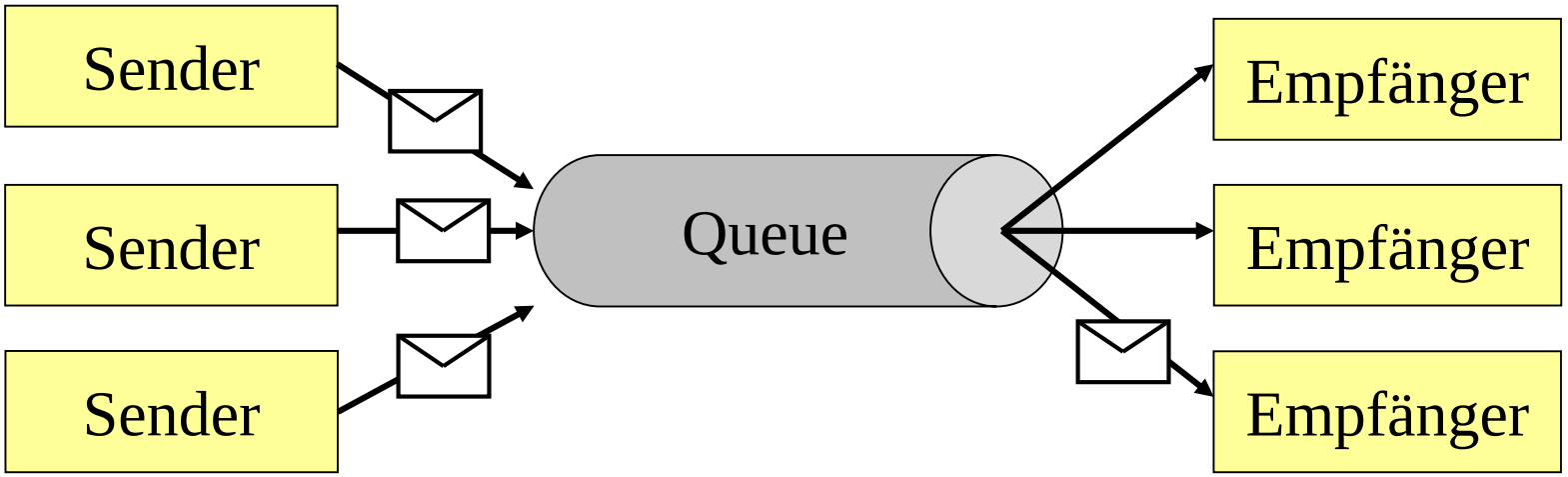
Messaging Konzepte
Point-to-Point (one way)

- Ein oder mehrere Sender, ein Empfänger
  - Kommunikation nur in eine Richtung
- Zeitliche Entkoppelung
  - Sender senden Nachricht in Queue
  - Empfänger holt Nachricht aus Queue zu beliebigem Zeitpunkt
  - Sender und Empfänger müssen ***nicht gleichzeitig online*** sein
- Queue kann transaktionsgesichert/persistent sein (s.u.)



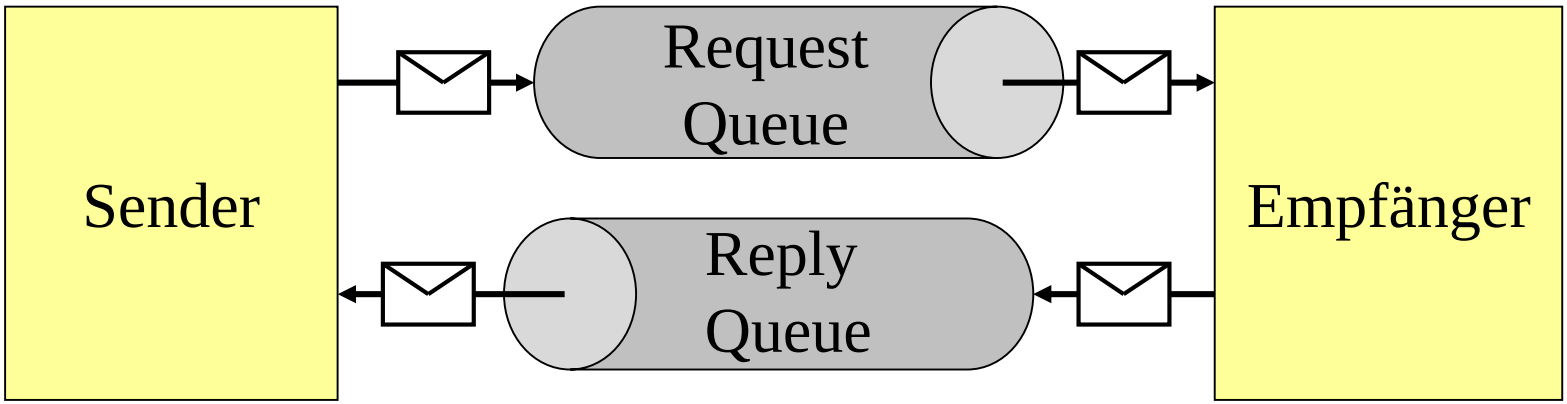
Messaging Konzepte
„Worker“

- Viele Sender, viele gleiche Empfänger (Worker):
  - Verschiedene Sender senden Nachrichten (z.B. Jobs)
  - Viele Empfänger können Nachricht entnehmen (***verbrauchend***)
  - Nur ein Empfänger verarbeitet die Nachricht
- Idee: Skalierbarer Service, da beliebig viele Empfänger start/stopp-bar



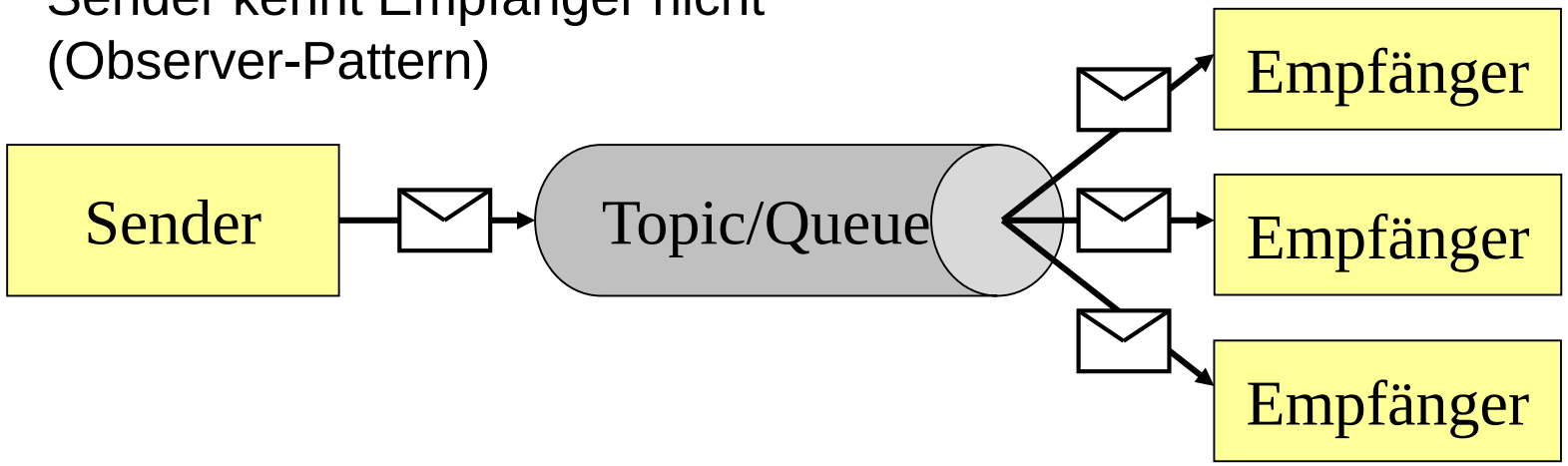
Messaging Konzepte
Request-Reply

- Jeder Knoten ist Sender und Empfänger
- Zwei Queues
  - Request-Queue (Sender -> Empfänger)
  - Reply-Queue (Empfänger -> Sender)
- Damit RPC ähnliche Kommunikation möglich
  - z.B. Technisch Asynchron / Logisch Synchron

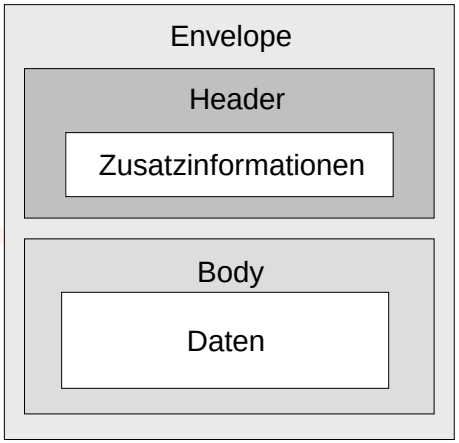


Messaging Konzepte
Publish/Subscribe

- Ein Sender, viele Empfänger:
  - Sender schicken Nachricht an Topic („Thema“)
  - Viele Empfänger melden sich bei Topic an
  - Alle Empfänger erhalten Nachricht aus Topic (***nicht verbrauchend***)
  - Empfänger müssen jedoch online sein
- Sender und Empfänger entkoppelt
  - Sender kennt Empfänger nicht (Observer-Pattern)



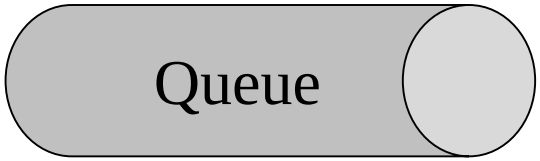
Nachricht/Messages



Bestandteile:

- Header
  - Wird nur von der Middleware/MOM interpretiert
  - Enthält Verwaltungsinformationen / Metainformationen für die Middleware (Sender, Empfänger, Verfallszeit, …)
- Body
  - Wird von der Anwendung interpretiert
  - Enthält Anwendungsdaten
  - ***Datenformat: frei definierbar***: ASCII-Text, XML, Binärdaten, …

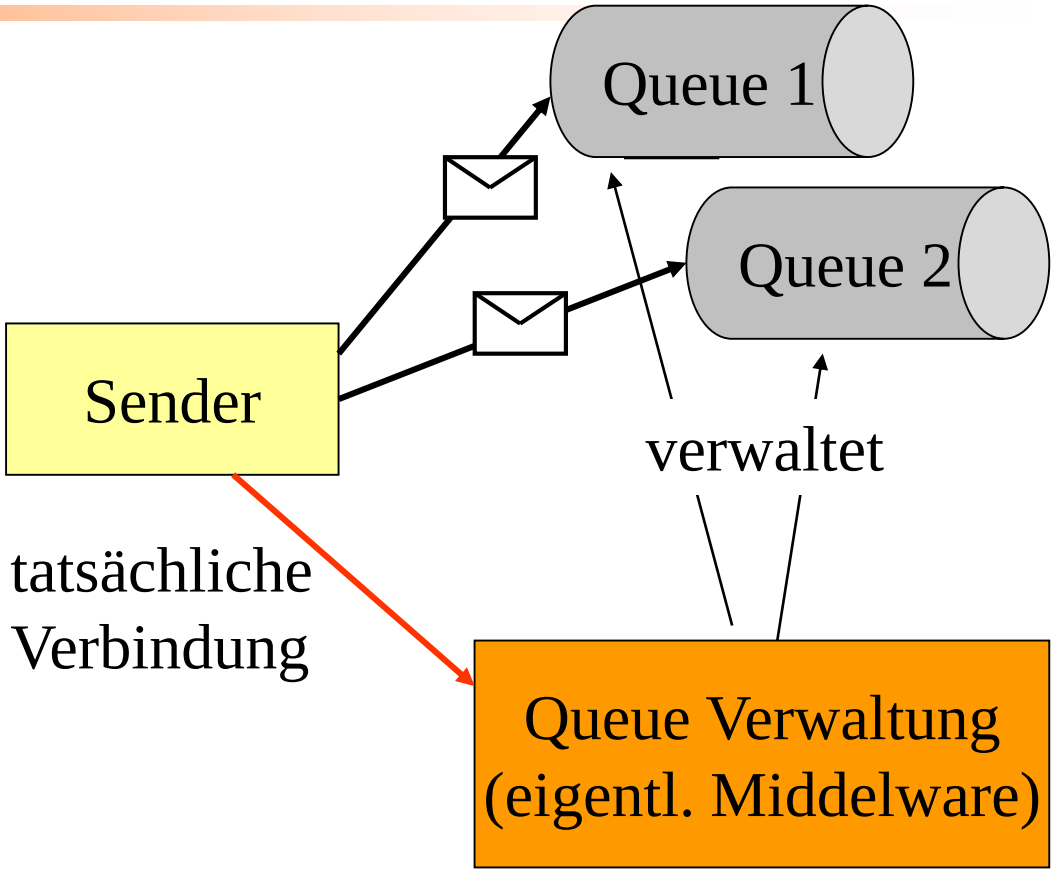
Kanäle / Queues / Topics



- Speicher für Nachrichten
  - Häufig persistent (mit Datenbank)
  - Häufig transaktional (schreiben / lesen jeweils als TXN)
  - idR. FIFO-Prinzip, auch Priority Queue
  - Häufig „Speicher“ beim Sender und „Speicher“ beim Empfänger
- Wird von der MOM verwaltet
  - ist also unabhängig von der Anwendung
  - Kann mit MOM-Tools erstellt / gelesen / gelöscht werden
- Sonderform: Dead Letter Queue (für unzustellbare Nachrichten)

Queue Verwaltung

- = MOM-Server
  Verwaltet Queues
  - Überwacht Q.
  - Überwacht Nachrichten
  - Evtl. mit Persistenz



Wozu brauchen sie Queues und eine nachrichtenorientierte Middleware?

Synchrone Kommunikation
Wie bei RMI, SOAP, REST (Ausnahmen mögl.)

- Nutzer sitzt vor Client, erwartet flüssige Interaktion (-> ***Logisch*** Synchrone Kommunikation)
- Synchron ***= technisches/logisches*** Kommunikationsmodell von Client/Server
  - Z.B. Verteilte Ressourcen (z.B. RESTful WebServices)
- ***Client blockiert so lange, bis Server geantwortet hat (-> Enge Koppelung)***
  - Absturz Server -> Client kann ggf. nicht mehr arbeiten
  - Server langsam -> Client langsam
  - Viele Clients -> Server evtl. verstopft / überlastet
    -> Problem der sog. „Skalierbarkeit“

Asynchrone Kommunikation

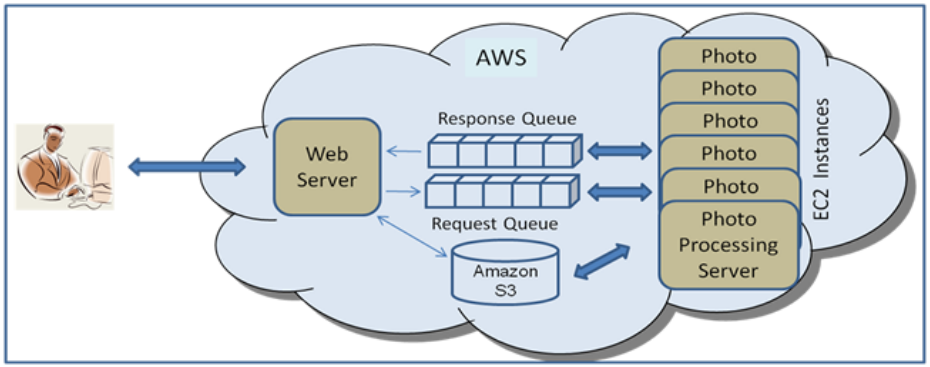
- Verarbeitung kann logisch sogar synchron sein (Nutzer wartet auf Antwort), ***technisch aber asynchron***
- Ein/mehrere Sender und ein/mehrere Empfänger (Client/Server = Sonderfall), d.h. Mehr Freiheit im Design
- ***Entkoppelung von Sender und Empf. über Queue***
- Sender sendet Nachricht und arbeitet weiter (Fire and Forget, ggf. mit Acknowledge)
- ***Empfänger arbeitet Nachricht ab, wenn er Zeit hat***
  - Absturz des Empfängers beeinträchtigt Sender nicht
  - Langsamer Empfänger beeinträchtigt Sender nicht
  - Empfänger kann durch viele parallele Empfänger ersetzt werden (Skalierbarkeit)

Warum MOM?
(insbesondere für Integrationsprobleme)

- MOM erlaubt ***zeitliche Entkoppelung*** von Sender und Empfänger (asynchrone Kommunikation, z.B. für Vertreter-Laptop)
- ***Nachrichtenformat*** kann frei gestaltet werden (z.B. XML, JSON)
- MOM unabhängig von technischen Plattformen
  - Integration verschiedener Plattformen möglich (JavaEE, Host, .NET)
  - MOM unabhängig administrierbar
- Nachrichten unterwegs ***erweiterbar/ transformierbar,*** damit auch technische Entkoppelung, Legacy-Software integrierbar
- ***Workflows*** über Nachrichten formulierbar: Workflow = Weiterleitung, Transformation, Pufferung von Nachrichten
- ***Throtteling*** / Thread Management einfach möglich
- Flexiblere Architekturen, da 1:n und m:n Kommunikationsmodelle (Publisher Subscriber) einfach

Warum asynchron?
(Insbesondere für Verfügbarkeit / Skalierbarkeit)

- Höhere Gesamtverfügbarkeit, Sender kann arbeiten, obwohl Empfänger abgestürzt
- Wenn Empfänger wieder hochfährt, alle Nachrichten noch in Queue, kein synchrones „Retry“
- Bessere (automatische!) Skalierbarkeit

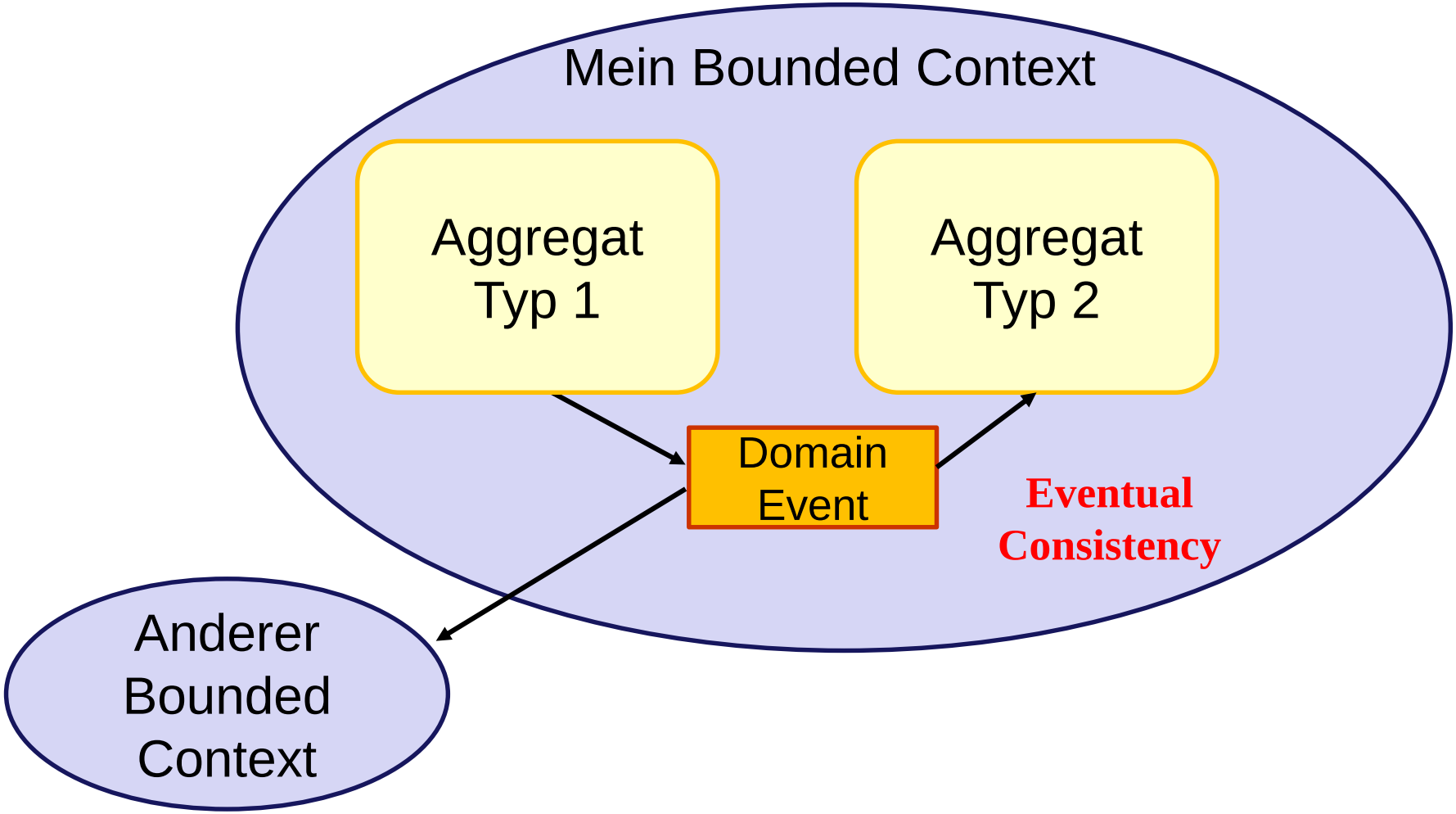


- „Share Nothing“: fast zwingend vorgegeben

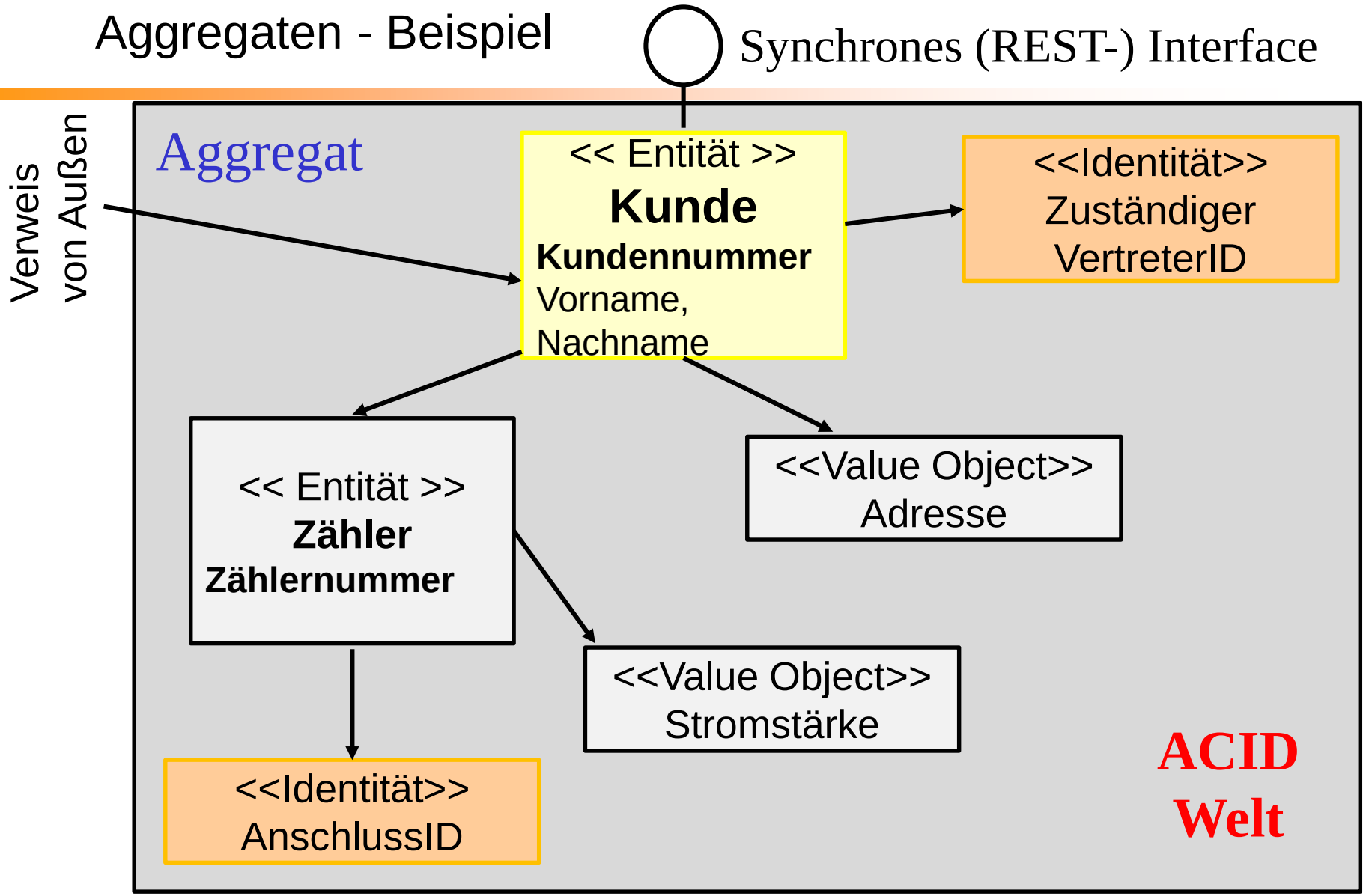
Warum MOM?
Domain Events

- = something happened that domain experts care about
- Aktivitäten in einer Domäne
  = Folge diskreter Ereignisse
  = Ausgeführte Anwendungsfälle / User Storys
- Ereignis, fachlich wichtig
  - = eignes Objekt, das auch gespeichert oder in eine Queue geschrieben werden kann.
  - hat eine fachliche (keine technische) Bedeutung
  - Beispiele: ***KundeAngelegt, KundeGelöscht, KundeGeprüft, FiskalJahrEnde, MonatEnde***
  - Immutable, wird nicht gelöscht oder verändert

Warum MOM?
Zusammenspiel zwischen zwei „Microservices“



Warum MOM? Kommunikation zwischen Aggregaten - Beispiel



MOM Produkte

- Betriebssysteme
  - Microsoft MSMQ
    - Fester Bestandteil seit Windows 2000 / XP
    - Einfacher Zugriff, z.B. über WCF, und andere .NET API
- Eigenständige MOM - Produkte
  - IBM MQSeries (WebSphere MQ)
    - Eines der ältesten MOM Produkte (IBM-Host)
    - Mittlerweile Bestandteil von WebSphere (WebSphere MQ)
  - ***Apache ActiveMQ***
  - ***RabbitMQ (pivotal)***
- JEE Application Server mit integrierter MOM (JMS impl.)
  - z.B. IBM WebSphere, BEA WebLogic, Oracle OAS, …
  - = Message Driven Beans
- EAI-Produkte mit MOM als Kern
  - z.B. TIBCO, WebMethods, SeeBeyond, Vitria

Bevor es losgeht: Standard Protokolle für Asynchrone Middleware

Ziel: Interoperabilität

STOMP vs.

MQTT vs.

AMQP

STOMP

- = **Streaming Text Oriented Messaging Protocol**
- Ähnlich wie HTTP

- *Details fehlen hier noch*

MQTT

- = **Message Queuing Telemetry Transport (MQTT)**
- Etablierter Standard, speziell bei IoT
- Definiert von IBM, derzeit gepflegt über OASIS

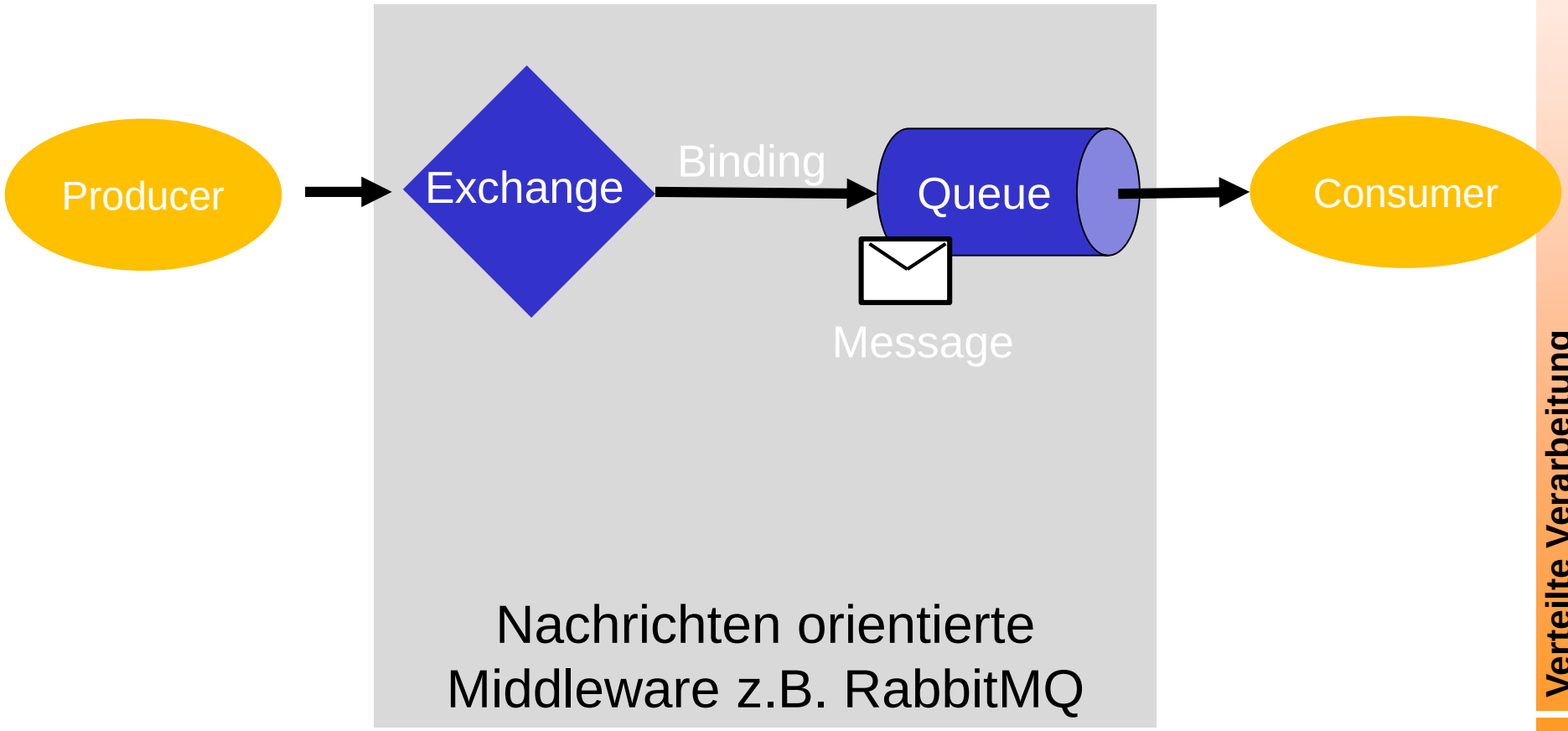
- *Details fehlen hier noch*

AMQP
https://www.amqp.org/

- = **Advanced Message Queuing Protocol**
- Herkunft aus dem Bankenbereich (JPMorgan)
- Implementierungen z.B. RabbitMQ
- Flexible Architektur durch „Exchanges“ und „Bindings“

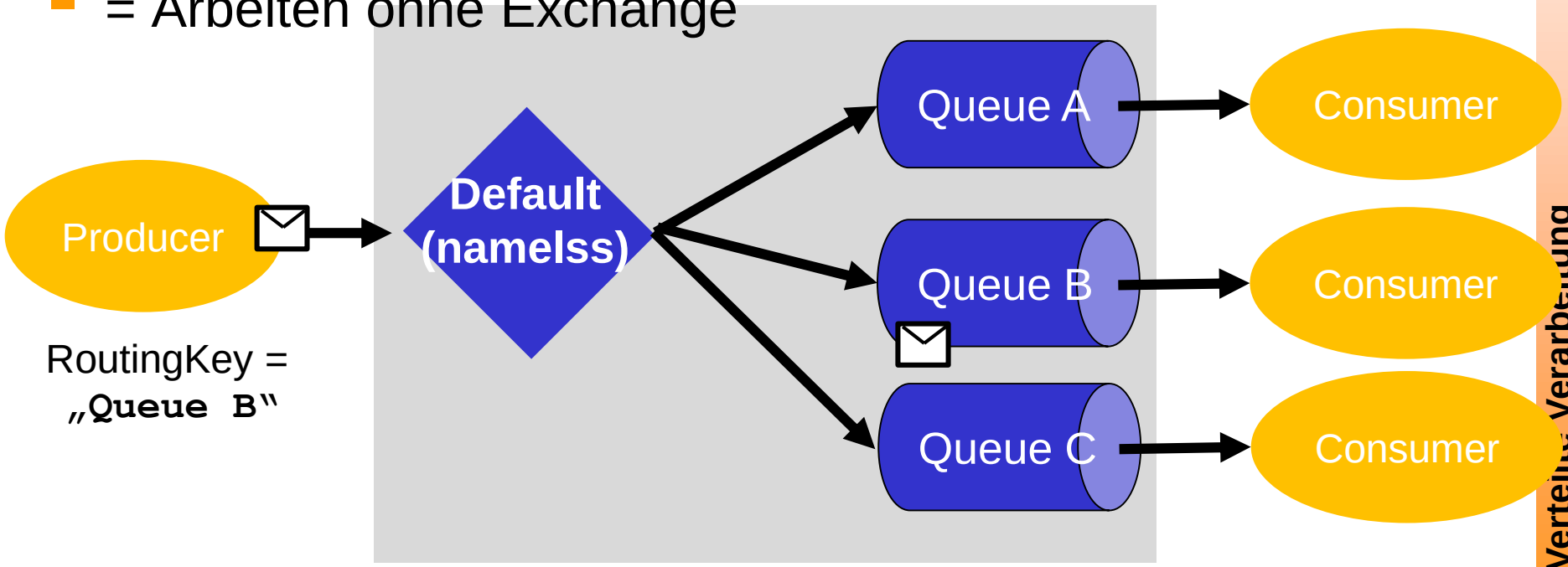
- Siehe: https://www.youtube.com/watch?v=deG25y_r6OY

AMQP Konzepte



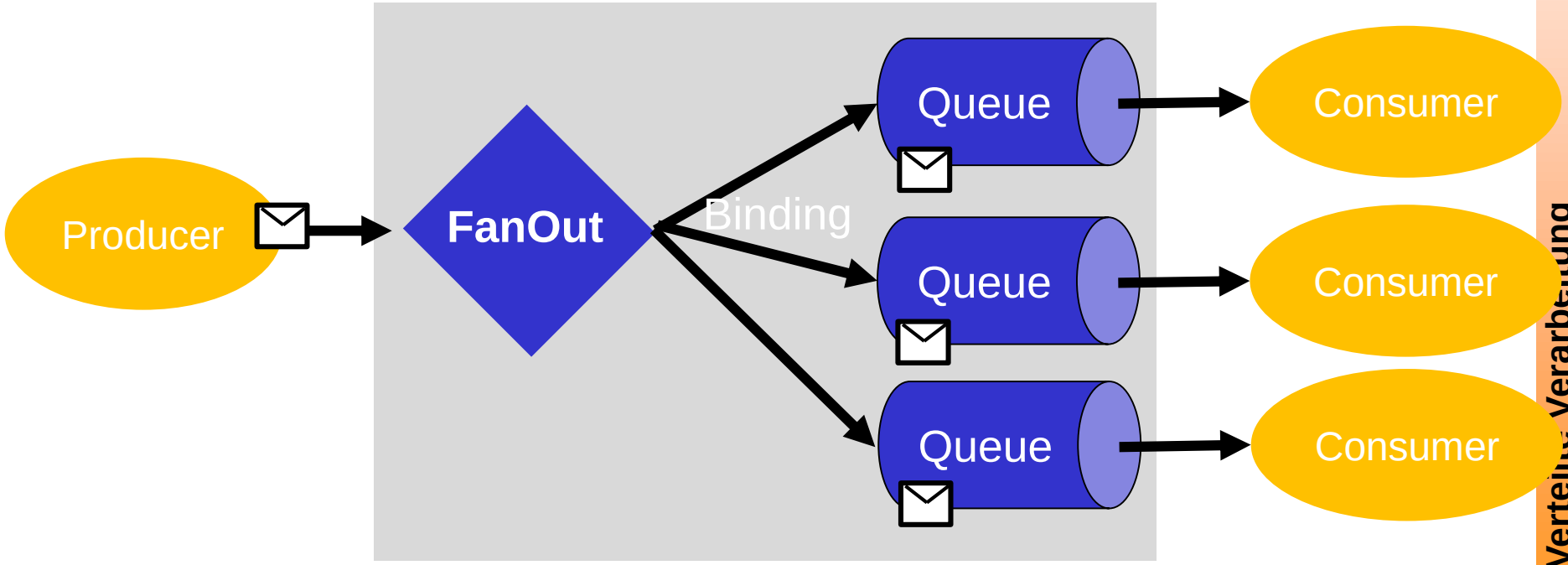
Umsetzung: Direkte Adressierung ohne Exchange (vgl. einfaches Beispiel im Code)

- Namenloser Exchange
  (wenn sie im Code keinen Exchange angeben)
- Queue-Name == Routing Key
- = Arbeiten ohne Exchange



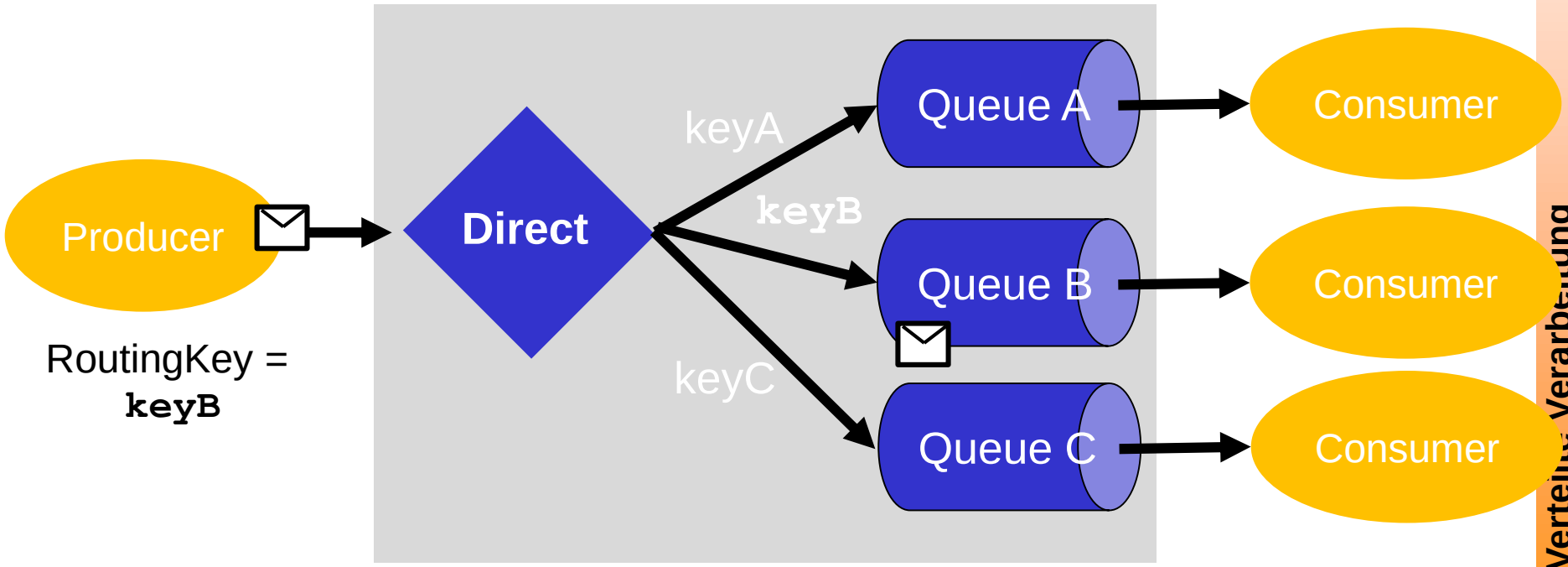
Umsetzung: Publisher Subscriber

- FanOut liefert aus an alle Queues, die über das Binding angemeldet sind
- = asynchrones Client / Server
- Unabhängig vom Binding Key



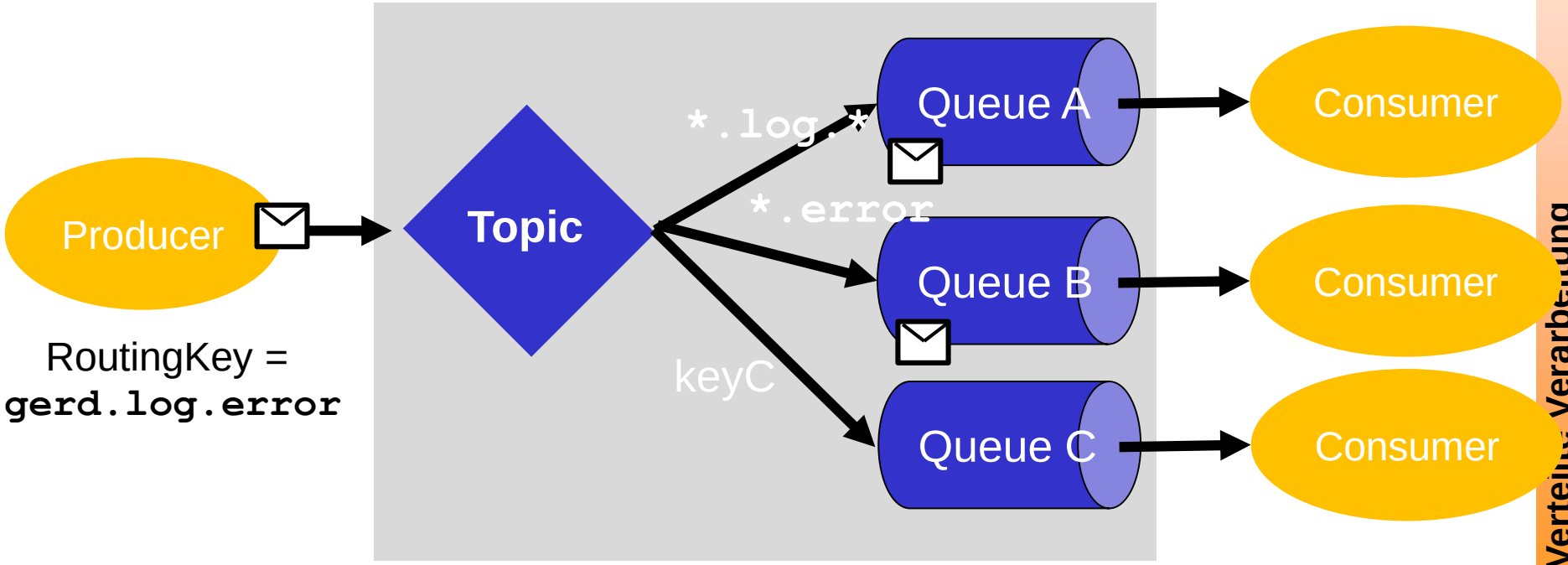
Umsetzung: Direkte Lieferung in Queue

- Direct: Routing Key = Binding Key
- Damit direkt Queue und Consumer adressieren



Umsetzung: Multicasting

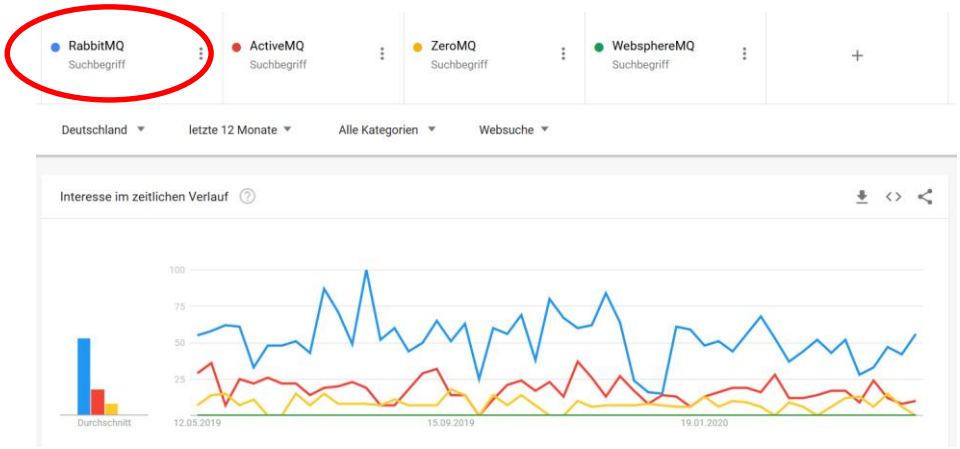
- Topic: Routing Key matched Binding Key
- Teilweises matchen -> Zustellung
- Dazu enthält das Binding wildcards z.B. **`*.log.*`**

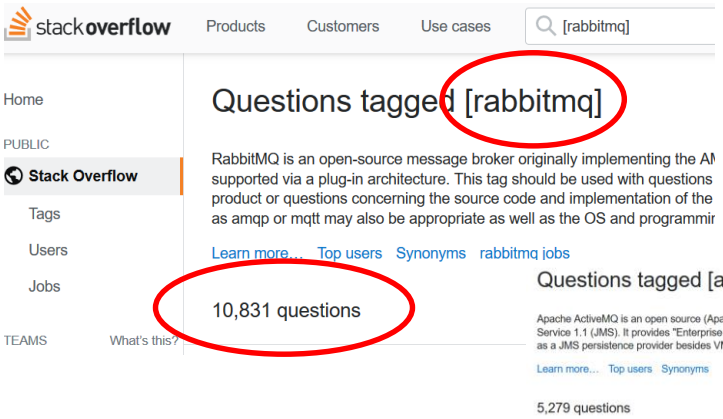



RabbitMQ
by Pivotal

RabbitMQ

- Sehr häufig verwendete Nachrichten Orientierte Middleware





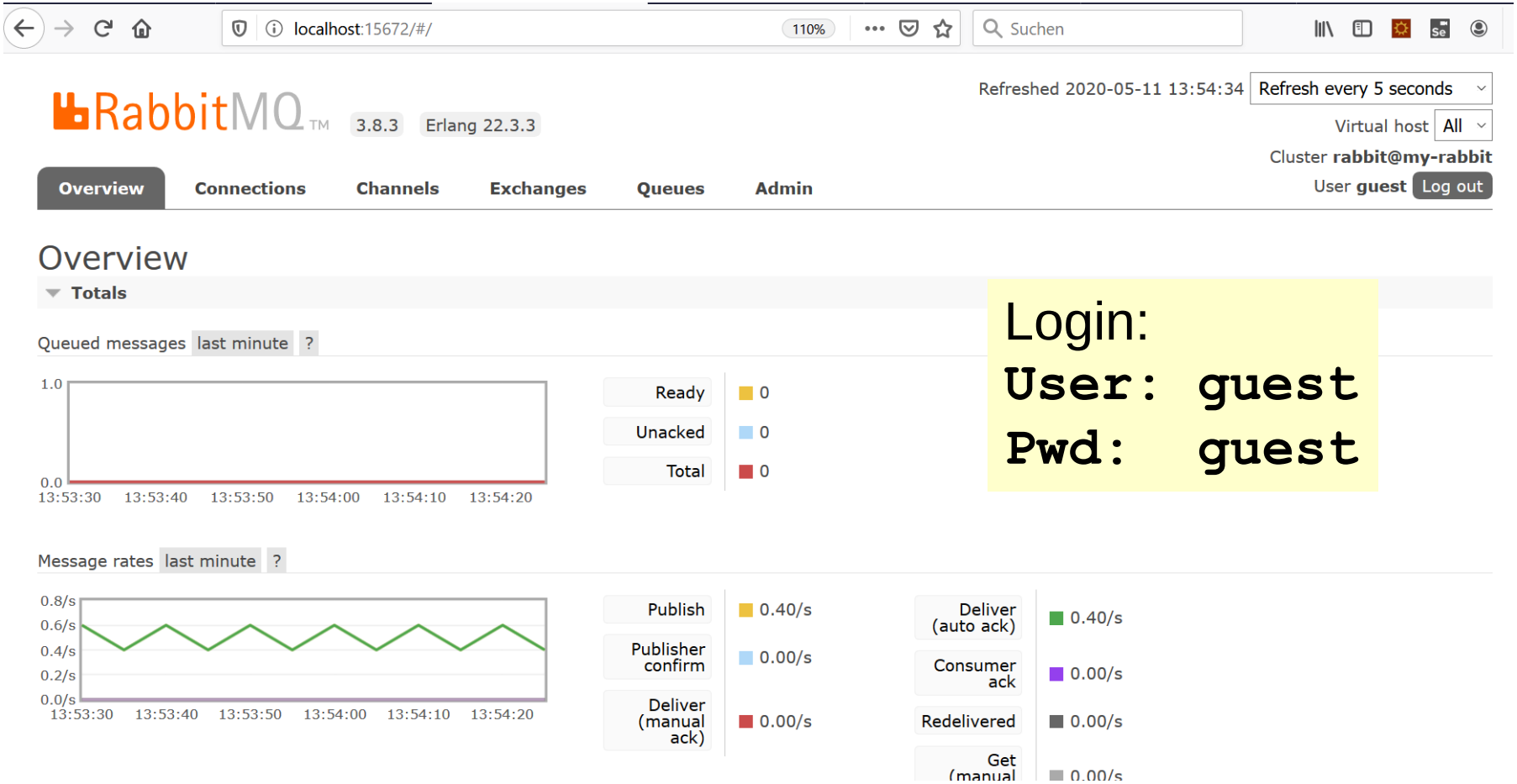
- Open Source (MPL), gepflegt von Pivotal (SpringBoot)
- Adapter für die meisten Programmiersprachen (Python, Go, Java, C#, …), Spring-Integration
- Implementiert Standards wie AMQP, STOMP (Plugin), MQTT (Plugin) oder JMS (siehe unten)

Installation über Docker-Container https://hub.docker.com/_/rabbitmq

```
docker run -d --hostname vv-rabbit --name some-rabbit
     -p 15672:15672 -p 5672:5672 rabbitmq:3-management
```

- RabbitMQ-Image enthält Erlang und andere benötigte Komponenten
- Management Konsole ist dabei (siehe nächste Folie)
- Achtung: Zwei Ports müssen gemapped warden
  a) Port für die Management-Konsole: 15672
  b) Port für den eigentlichen Service: 5672

Management – Konsole
http://localhost:15672



Ergänzung in build.gradle

Home » com.rabbitmq » amqp-client » 5.9.0

RabbitMQ Java Client » 5.9.0

The RabbitMQ Java client library allows Java applications to interface with RabbitMQ.

License	Apache 2.0 GPL 2.0 MPL 1.1
Categories	Message Queue Clients
Organization	VMware, Inc. or its affiliates.
HomePage	https://www.rabbitmq.com
Date	(Apr 14, 2020)
Files	jar (629 KB) View All
Repositories	Central
Used By	**488 artifacts**

Maven | Gradle | SBT | Ivy | Grape | Leiningen | Buildr

```
// https://mvnrepository.com/artifact/com.rabbitmq/amqp-client
compile group: 'com.rabbitmq', name: 'amqp-client', version: '5.9.0'
```

☑ Include comment with link to declaration

Hello World mit RabbitMQ + AMQP
Senden …

```
public class Producer {
  public static void main(String[] args) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setUri("amqp://guest:guest@localhost");
    factory.setConnectionTimeout(100000);
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare("my-queue", true, false,
         false, null);

    String message = "Tachchen, isch bin eine Nachricht";
    channel.basicPublish("", "my-queue",
                  null, message.getBytes());
  }
}
```

Hello World in RabbitMQ + AMQP Empfangen …

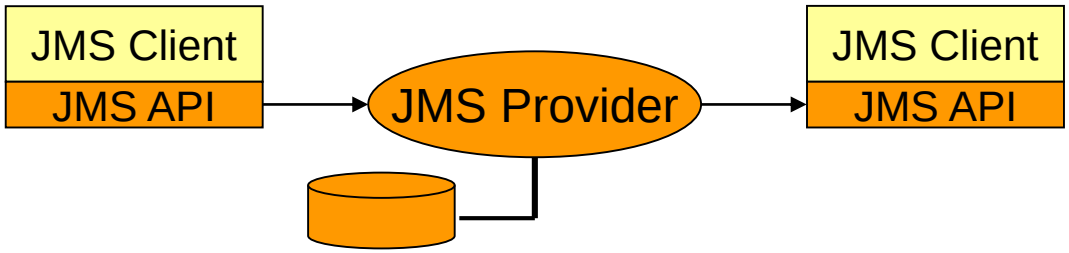
```
public class Consumer {
  public static void main(String[] args) {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setUri("amqp://guest:guest@localhost");
    factory.setConnectionTimeout(100000);
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare("my-queue",
          true, false, false, null);
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
          String message = new String(delivery.getBody(), "UTF-8");
          System.out.println(" [x] Received '" + message + "'");
    };

    channel.basicConsume("my-queue", true,
           deliverCallback, consumerTag -> { });
  }
}
```

JMS

Java Message Service (JMS)

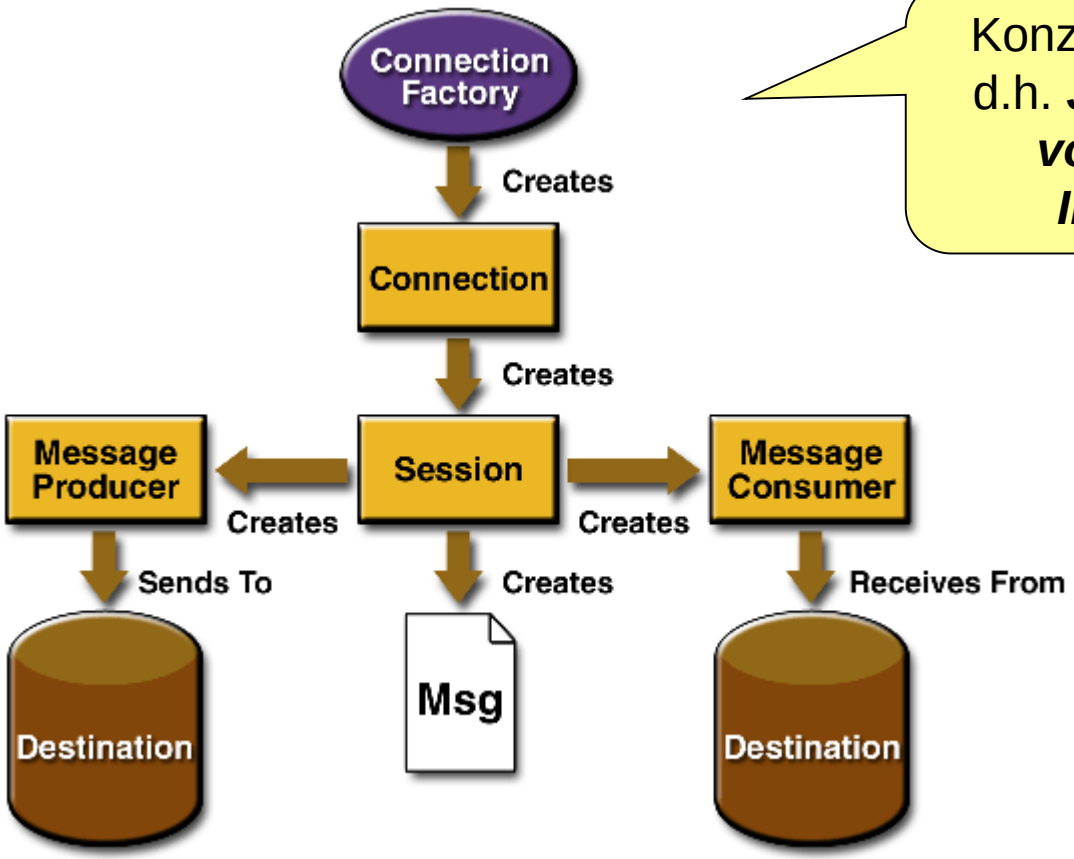
- Teil der Java EE-Plattform (**`javax.jms.`** …)
- JMS unterscheidet zwei Rollen
  - ***JMS Provider***: der jeweilige MOM Server, z.B. ActiveMQ
  - ***JMS Client***: die Java Anwendung
    (Empfänger und Sender von Nachrichten).
- JMS unterstützt alle Messaging Konzepte
  - Point-to-Point
  - Request/Reply
  - Publish and Subscribe
- Implementierungen:
  Java EE Application Server, ActiveMQ, RabbitMQ…

JMS Provider



- JMS Provider = eigener Prozess / Bestandteil BS
- Verwaltet Queues und Topics
  - Queues und Topics werden idR. Vom Administrator angelegt
  - Programmatisches Anlegen ist möglich
- JMS Client: muss nicht in Java implementiert sein

JMS Programmiermodell



Konzept wie JDBC
d.h. ***JMS definiert vorwiegend Interfaces***

[Quelle: JMS-Tutorial, http://java.sun.com/products/jms/tutorial]