QAWARE SOFTWARE ENGINEERING

Hochschule München University of Applied Sciences

Cloud Computing

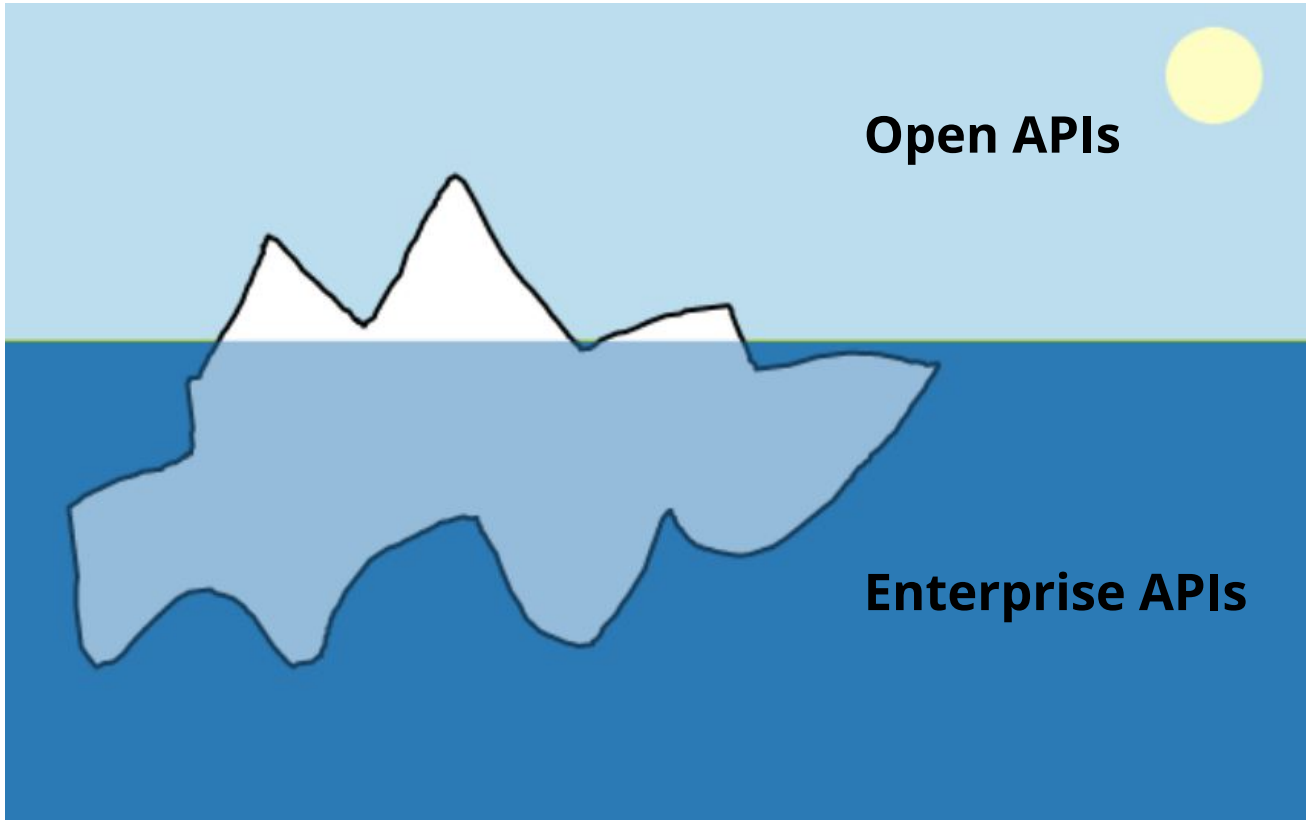
Kommunikation

Bernhard Saumweber

2022-03-24

“Man kann nicht nicht kommunizieren”

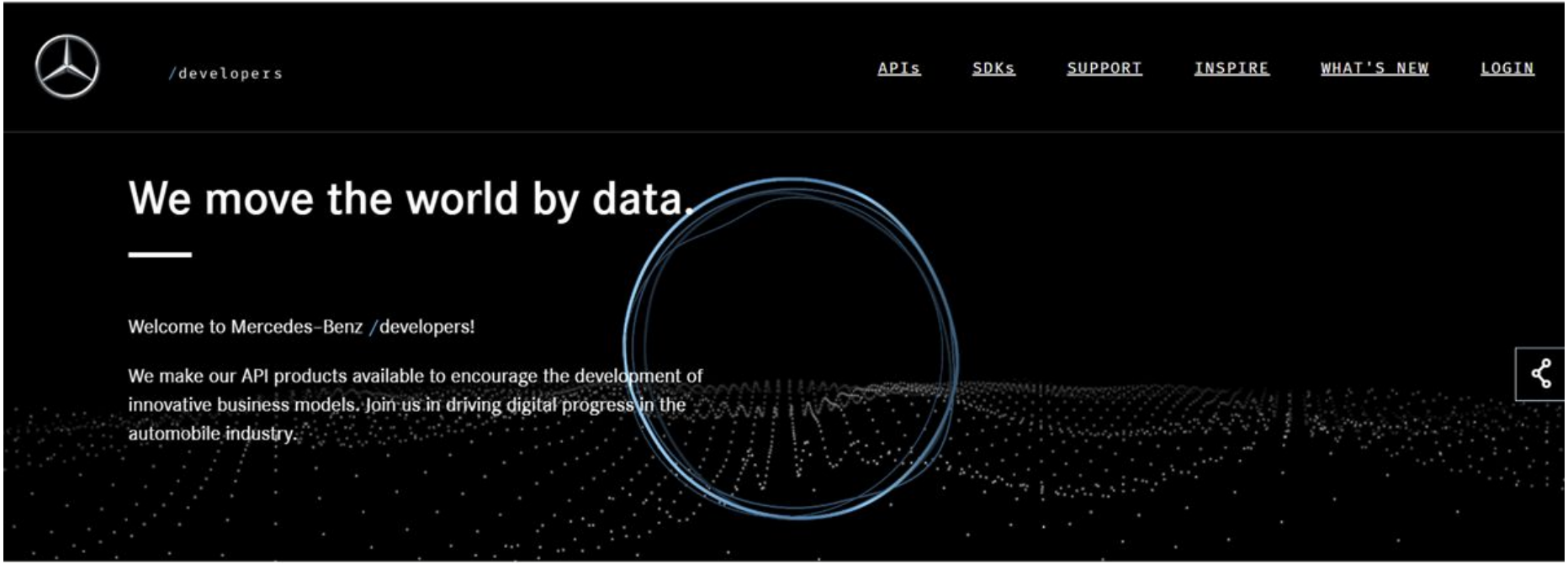
- Paul Watzlawik



- Nur ein Bruchteil der bestehenden APIs ist öffentlich aufrufbar.
- Mit APIs lässt sich Geld verdienen.
- Bestimmte APIs müssen veröffentlicht werden.
- Nutzer öffentlicher APIs entwickeln (mit etwas Glück) innovative Produkte.

https://joshdata.me/iceberger.html , inspired by https://twitter.com/GlacialMeg/status/1362557149147058178

Beispiel: Mercedes Benz Developer Portal



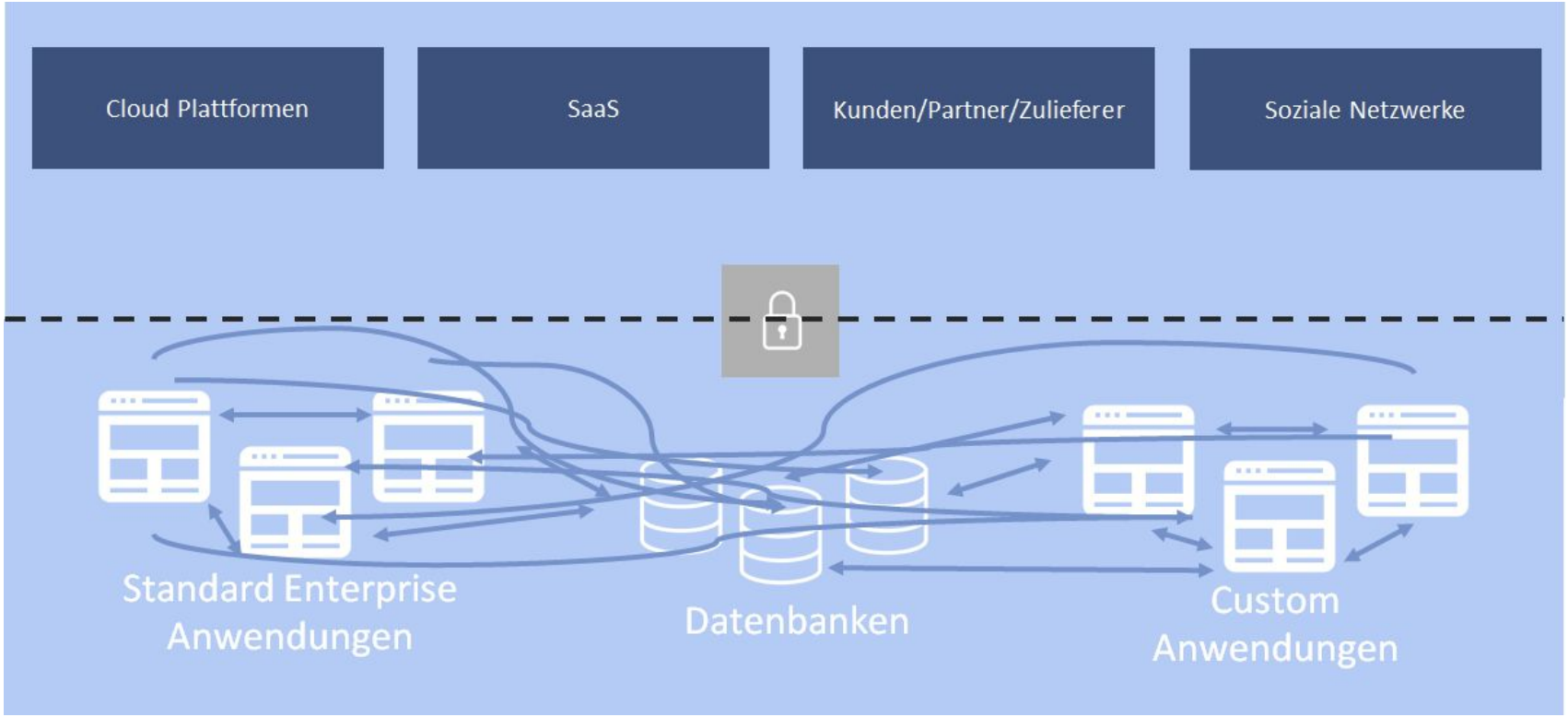
https://developer.mercedes-benz.com/

Was hat das mit Cloud Computing zu tun?

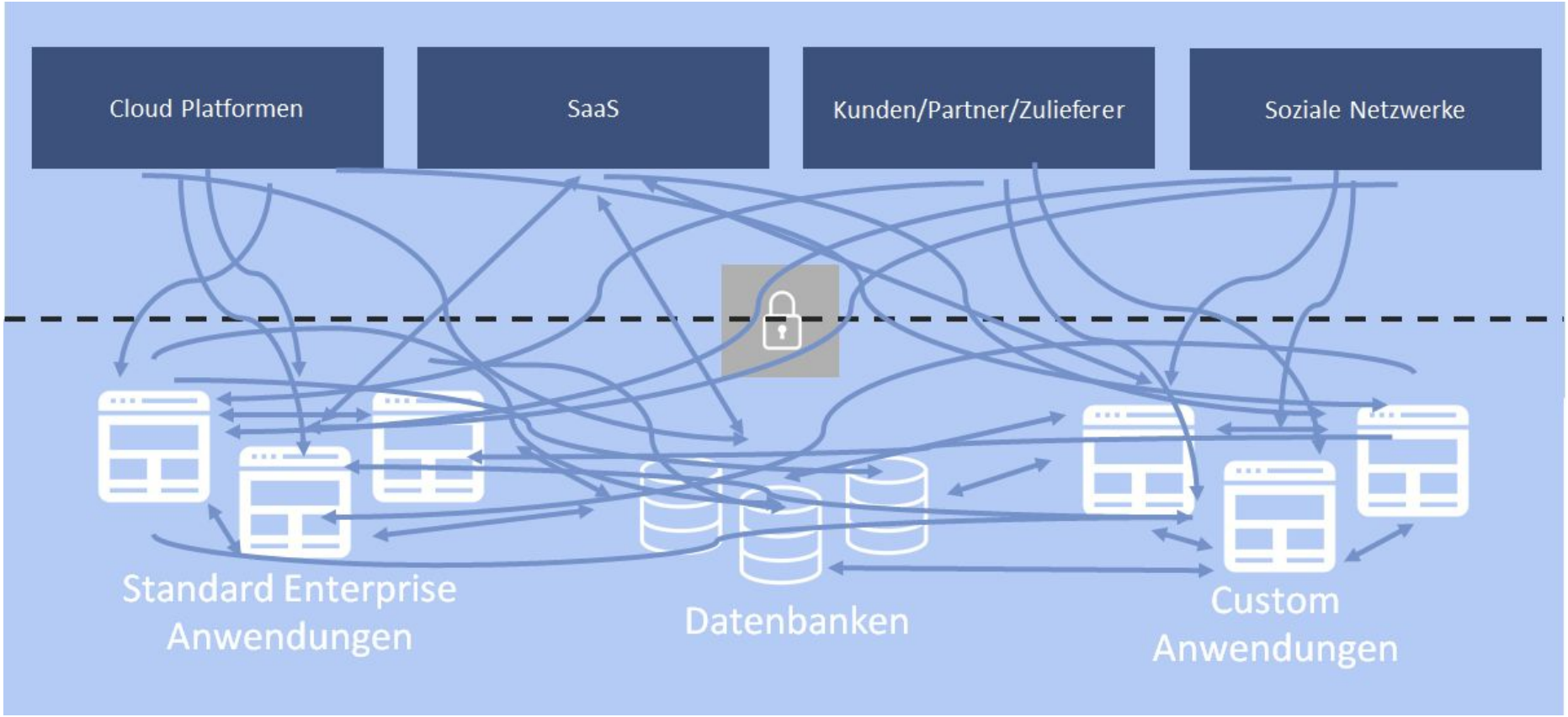
„Früher“ lief Enterprise Software hinter einer Firewall

„Früher“ lief Enterprise Software hinter einer Firewall.
Die Komponenten kommunizieren intern.

Heute geht die Vernetzung noch deutlich weiter, auch über die Grenzen des Unternehmensnetzwerks hinaus:

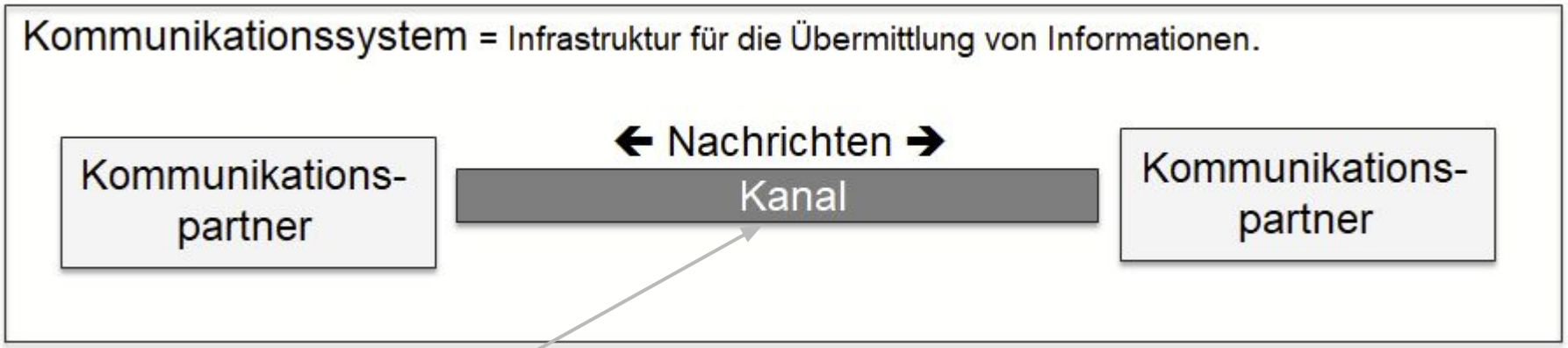


Aber eigentlich sieht es so aus:



Kommunikationsmodelle

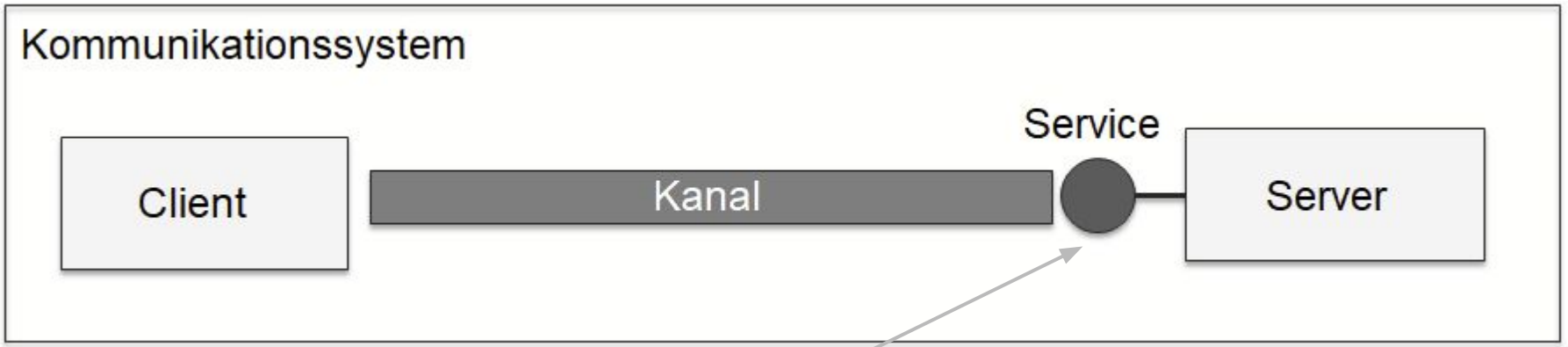
Ein allgemeines Kommunikationsmodell im Internet. Angelehnt an das Modell von Shannon/Weaver.



Typische Kanaleigenschaften:

- Richtung
- Datenformat/Codierung
- Synchronität (synchron/asynchron)
- Zuverlässigkeit und Garantien
- Sicherheit
- Performance (Latenz, Bandbreite)
- Overhead (Nutzlast / Gesamtlast)

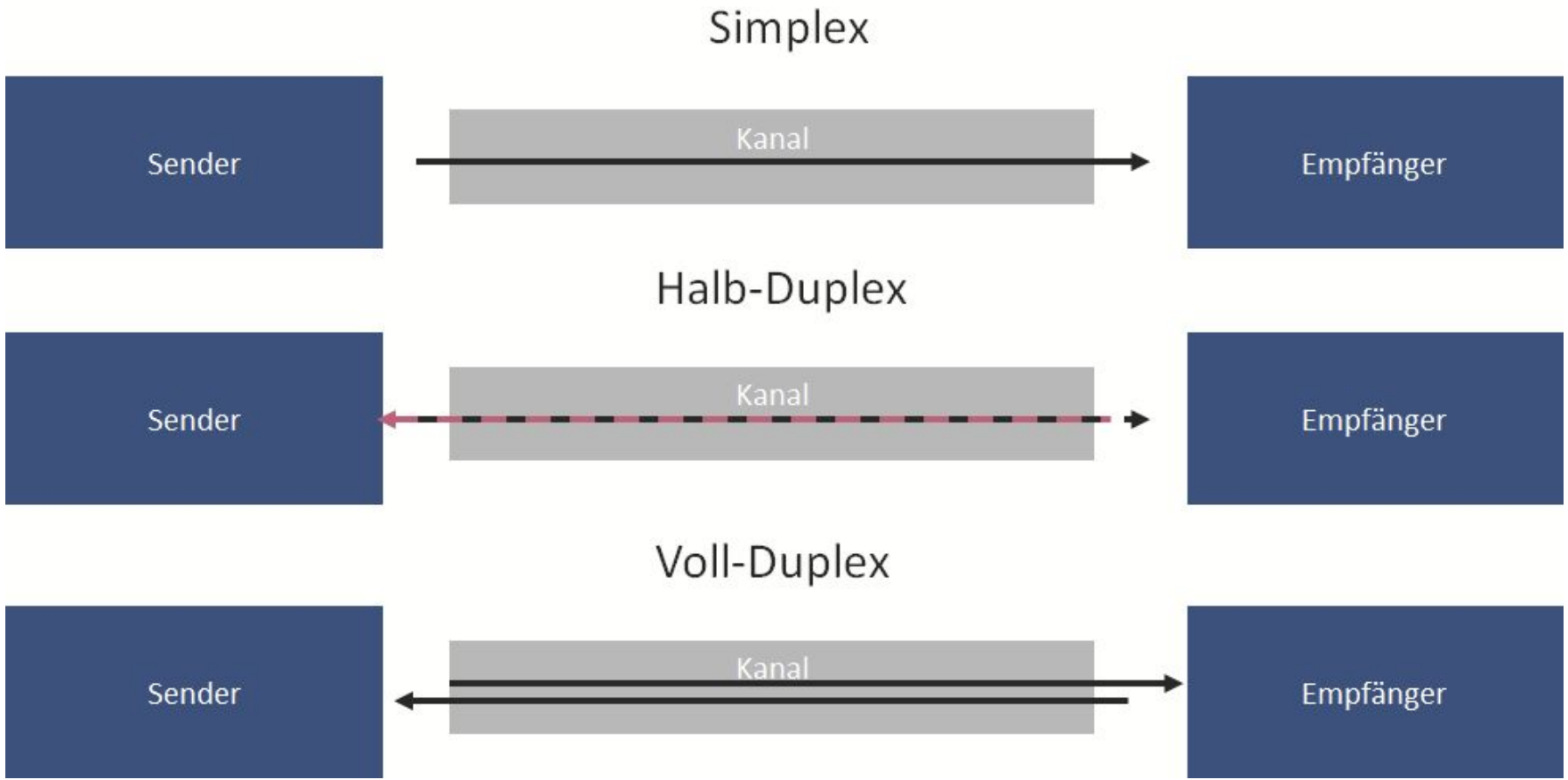
Service-Orientierung in einem Kommunikationssystem: Client-Server-Kommunikation über Services



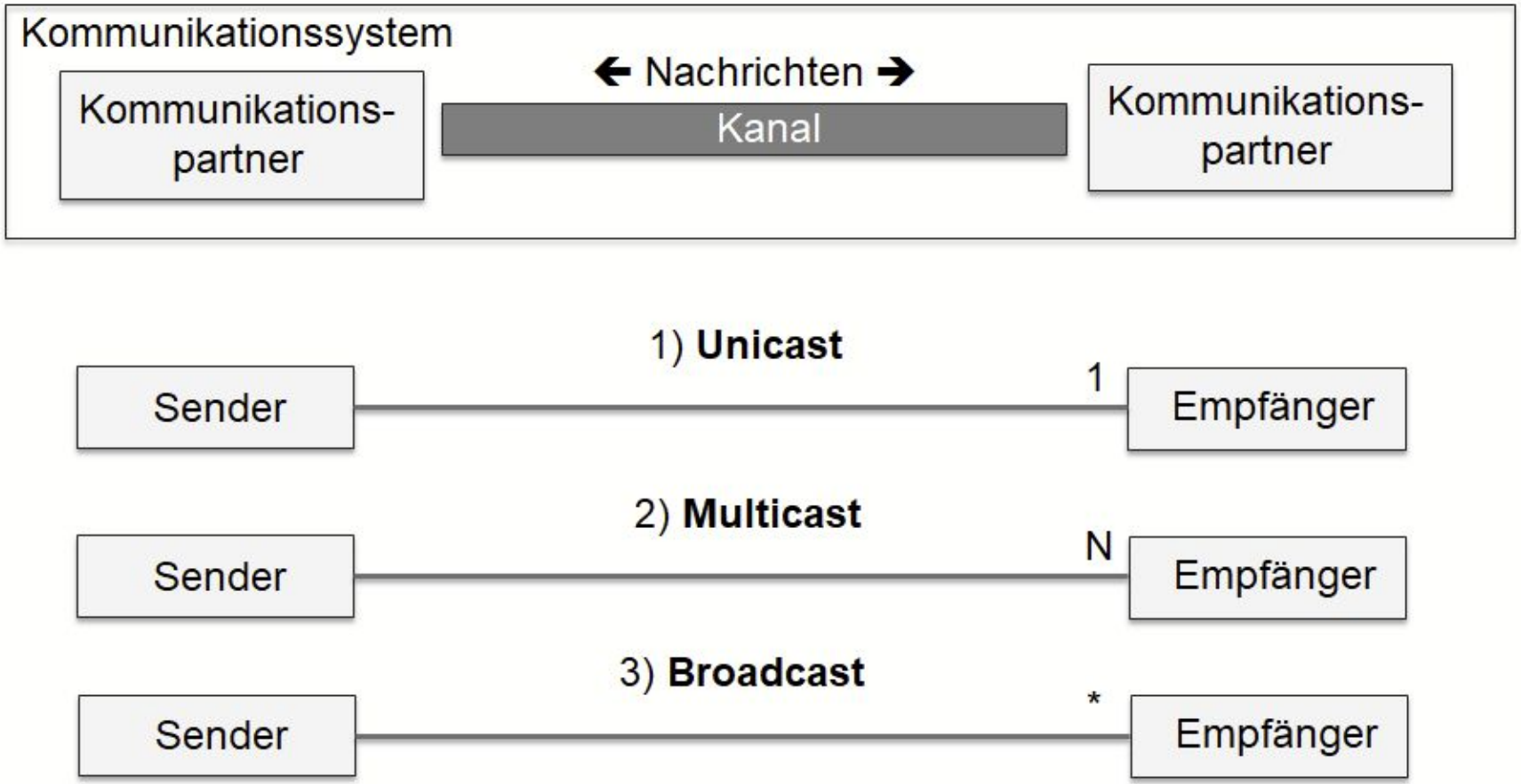
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

Eine **Serviceschnittstelle** ist ein Vertrag zwischen Nutzer und Anbieter über Syntax und Semantik der Service-Nutzung und enthält optional Zusicherungen in Hinblick auf den **Quality of Service**.

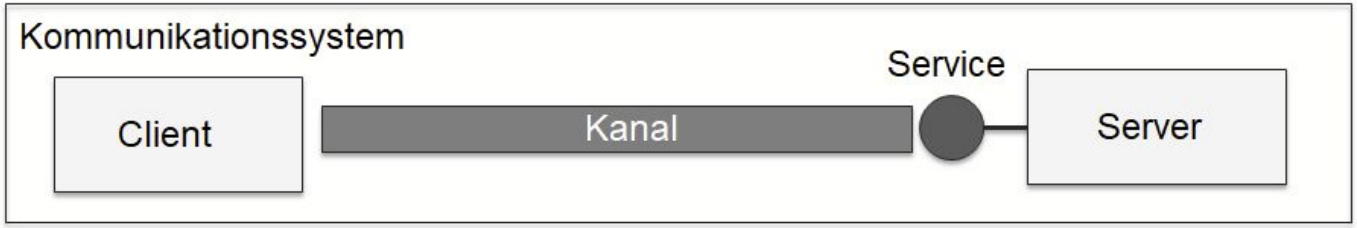
Nutzungsmuster des Kanals (Richtung)



Kardinalität der Empfänger einer Nachricht.



Wer beginnt mit der Kommunikation?



Client beginnt die Kommunikation.

Request/Response

Anfrage mit der Bitte um Antwort.

Benachrichtigung in eine Richtung.

Push

Server beginnt die Kommunikation.

Peer-to-Peer

Beide Kommunikationspartner können die Kommunikation beginnen (Ambivalenz der Rollen Client und Server).

Nachrichtenaustausch in beide Richtungen.

HTTP

Kommunikationsmuster in HTTP

- **Request/Response**: Klassisch HTTP. Client schickt Request, Server antwortet mit Response. Client schickt wieder Request, etc.
- **Push**: Long Polling. Klassisches HTTP, Verbindung wird offen gehalten, Server sendet Antwort später und beendet den Request, es wird danach sofort wieder ein neuer Request aufgebaut.
- **Push**: Server Sent Events (SSE). Client schickt Request, Server antwortet mit Response. Verbindung bleibt offen. Nun kann Server aber auch weitere Nachrichten, ohne Requests vom Client, an den Client schicken.
- **Push**: HTTP/2 Server Push. Server verschickt mit Response weitere Resourcen (z.B. CSS) vorab mit, ohne dass diese explizit angefordert wurden.
- **Peer-to-Peer**: Websockets. Client schickt Request, Server antwortet mit Upgrade-Response. Nun kann der Server Nachrichten an den Client schicken, und der Client kann Nachrichten an den Server schicken (bidirektionaler Kanal)

Ein Beispiel für eine HTTP-Kommunikation.

Client:

```
GET / HTTP/1.1
Host:
www.example.com
```

Server:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 155
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
...
```

Übertragungsformate

JSON

```
{
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Deckung": 2e+6,
  "Währung": "EURO",
  "Inhaber": {
    "Name": "Mustermann",
    "Vorname": "Max",
    "männlich": true,
    "Hobbys": [ "Reiten", "Golfen", "Lesen" ],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
```

- JSON = JavaScript Object Notation (Daten pur).
- Auch in Binärcodierung (BSON – Binary JSON).
- MIME-Typ: *application/json*
- Schema-Sprachen: JSON Schema (http://json-schema.org)

Datentypen:

- Nullwert: null
- bool'scher Wert: true, false
- Zahl: 42, 2e+6
- Zeichenkette: "Mustermann"
- Array: [1,2,3]
- Objekt mit Eigenschaften: {"Name":"Mustermann"}

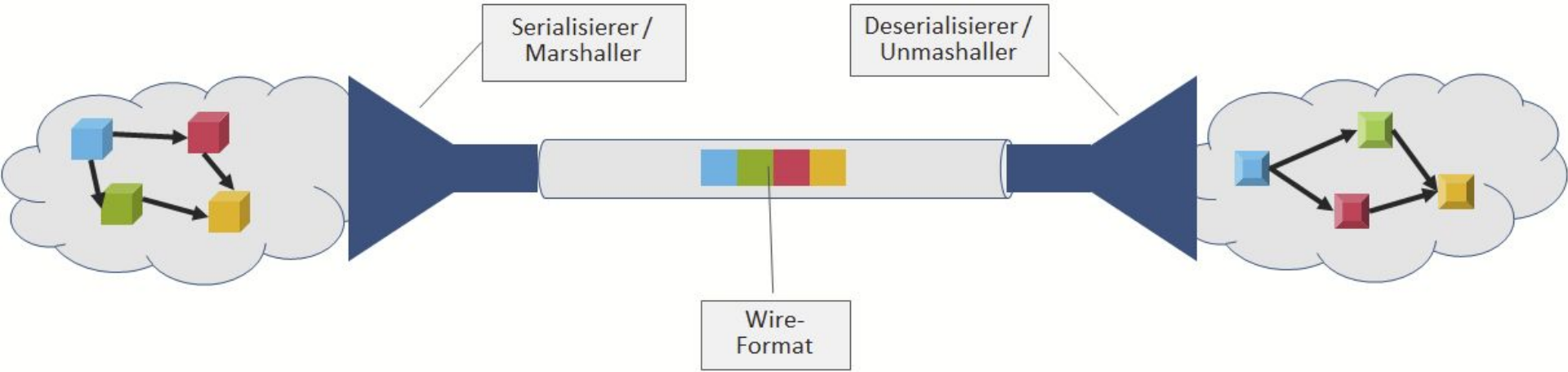
Protocol Buffers

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- 2008 von Google veröffentlicht
- Sprachneutral
- Platformunabhängig
- Binärkodiert - speichereffizient und schnell
- Protobuf-Source wird von Protobuf-Compiler in Programmiersprache übersetzt

Für die meisten Schnittstellen (JSON, ProtoBuf) benötigen wir Serialisierung und Deserialisierung.

Die **Serialisierung** ist […] eine Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.



Marshalling (englisch marshal ‚aufstellen', ‚ordnen') ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht

Einordnung der Performance

Je höher, desto schneller.

Benchmark	Mode	Cnt	Score	Error	Units
jsonDeserialize	thrpt	20	2969647,037	± 26838,990	ops/s
jsonSerialize	thrpt	20	4126749,377	± 40462,442	ops/s
protoBufDeserialize	thrpt	20	6992710,402	± 254170,355	ops/s
protoBufSerialize	thrpt	20	22019405,324	± 741610,436	ops/s
xmlDeserialize	thrpt	20	577380,698	± 9450,761	ops/s
xmlSerialize	thrpt	20	1578672,085	± 40488,728	ops/s

Service-orientierte Request-Response-Kommunikation mit REST

https://youtu.be/SLwpqD8n3d0

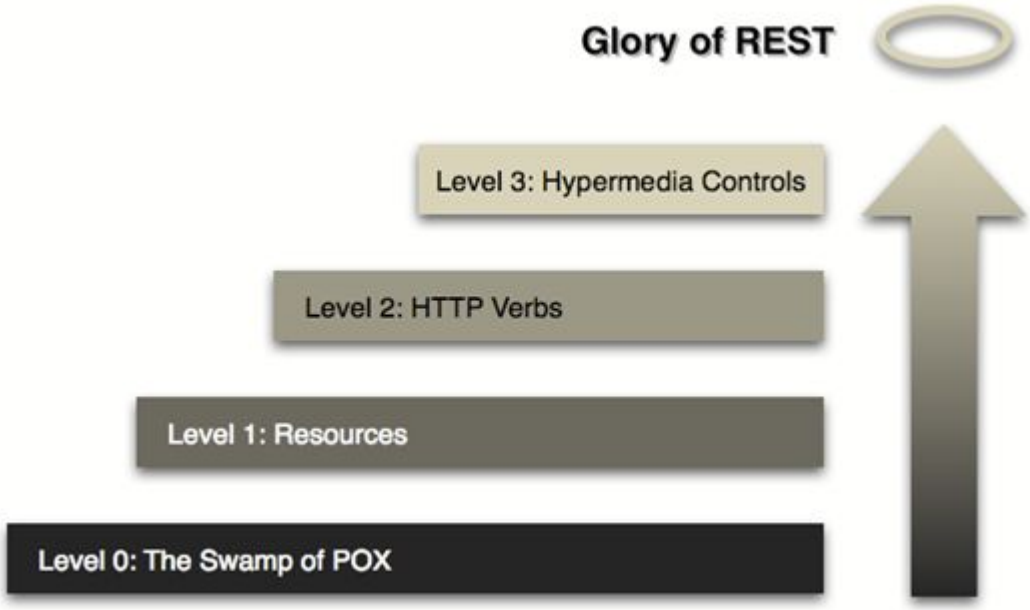
REST ist ein Paradigma für Anwendungsservices auf Basis des HTTP-Protokolls.

- REST ist eine Paradigma für den Schnittstellenentwurf von Internetanwendungen auf Basis des HTTP-Protokolls (Verben).
- Dissertation von Roy Fielding: „Architectural Styles and the Design of Network-based Software Architectures", 2000, University of California, Irvine.

Grundlegende Eigenschaften:

- **Alles ist eine Ressource**: Eine Ressource ist eindeutig adressierbar über einen URI, hat eine oder mehrere Repräsentationen (XML, JSON, bel. MIME-Typ) und kann per Hyperlink auf andere Ressourcen verweisen. Ressourcen sind, wo immer möglich, hierarchisch navigierbar.
- **Uniforme Schnittstellen**: Services auf Basis der HTTP-Methoden (POST = erzeugen, PUT = aktualisieren oder erzeugen, DELETE = löschen, GET = abfragen). Fehler werden über die HTTP Codes zurückgemeldet. Services haben somit eine standardisierte Semantik und eine stabile Syntax.
- **Zustandslosigkeit**: Die Kommunikation zwischen Server und Client ist zustandslos. Ein Zustand wird im Client nur durch URIs gehalten.
- **Konnektivität**: Basiert auf ausgereifter und allgegenwärtiger Infrastruktur: Der Web-Infrastruktur mit wirkungsvollen Caching- und Sicherheitsmechanismen, leistungsfähigen Servern und z.B. Web-Browser als Clients.

Mit dem REST Maturity Model kann bewertet werden, wie RESTful ein HTTP-basierter Service ist.



https://martinfowler.com/articles/richardsonMaturityModel.html

Wie sieht es jeweils aus, wenn eine Bestellung **geändert** wird?

http://www.service.de/customers/4711/orders
Eingabe: *Order* per PUT
Ausgabe: *HTTP-Code (200, 500)* + *Link auf geänderte Order*

http://www.service.de/customers/4711/orders
Eingabe: *Order* per PUT
Ausgabe: *HTTP-Code (200, 500)*

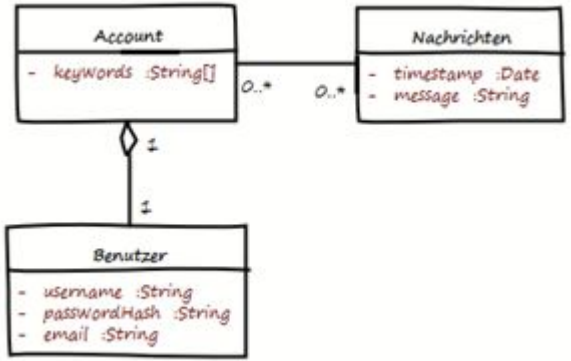
http://www.service.de/customers/4711/orders
Eingabe: *Order* per POST
Ausgabe: *UpdateOrderConfirmation (OK)*

http://www.service.de/updateOrderService
Eingabe: *UpdateOrderRequest* per POST
Ausgabe: *UpdateOrderConfirmation (OK)*

Für die Entwicklung von REST-APIs gibt es zwei Ansätze:

Am Anfang:

- Anwendungsfälle erheben
- Entitätenmodell erstellen
- Nun gibt es zwei Möglichkeiten:



Top-Down-Ansatz:

- REST-Schnittstelle definieren (z.B. in OpenAPI)
- REST-Schnittstelle zu Code generieren (z.B. mit dem OpenAPI Generator)
- REST-Schnittstelle implementieren

Bottom-Up-Ansatz:

- REST-Schnittstelle implementieren
- Definition der REST-Schnittstelle generieren (z.B. mit Swagger)

REST-API mit JAX-RS

Request Path

```
@Path("/hello/{name}")
public class HelloWorldResource {
   @GET
   @Produces(MediaType.APPLICATION_JSON)
   public ResponseMessage getMessage(
           @QueryParam("salutation") @DefaultValue("Hallo") String salutation,
           @PathParam("name") String name
   ) {
       return new ResponseMessage(Instant.now().toString(), salutation + " " + name);
   }
}
```

HTTP Verb

HTTP Content-Type

REST-API mit Spring Web MVC

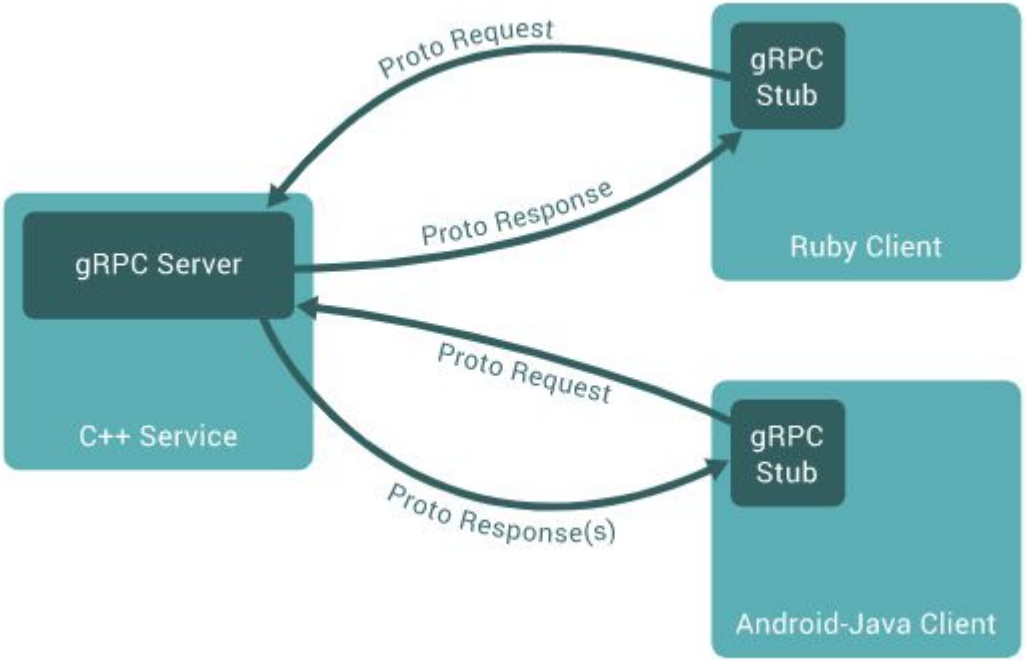
```
@RequestMapping("/hello/{name}")
public class HelloWorldController {

   @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
   public ResponseMessage getMessage(
           @RequestParam(name = "salutation", defaultValue = "Hallo") String salutation,
           @PathParam("name") String name
   ) {
       return new ResponseMessage(Instant.now().toString(), salutation + " " + name);
   }
}
```

Service-orientierte Request-Response-Kommunikation mit gRPC

gRPC

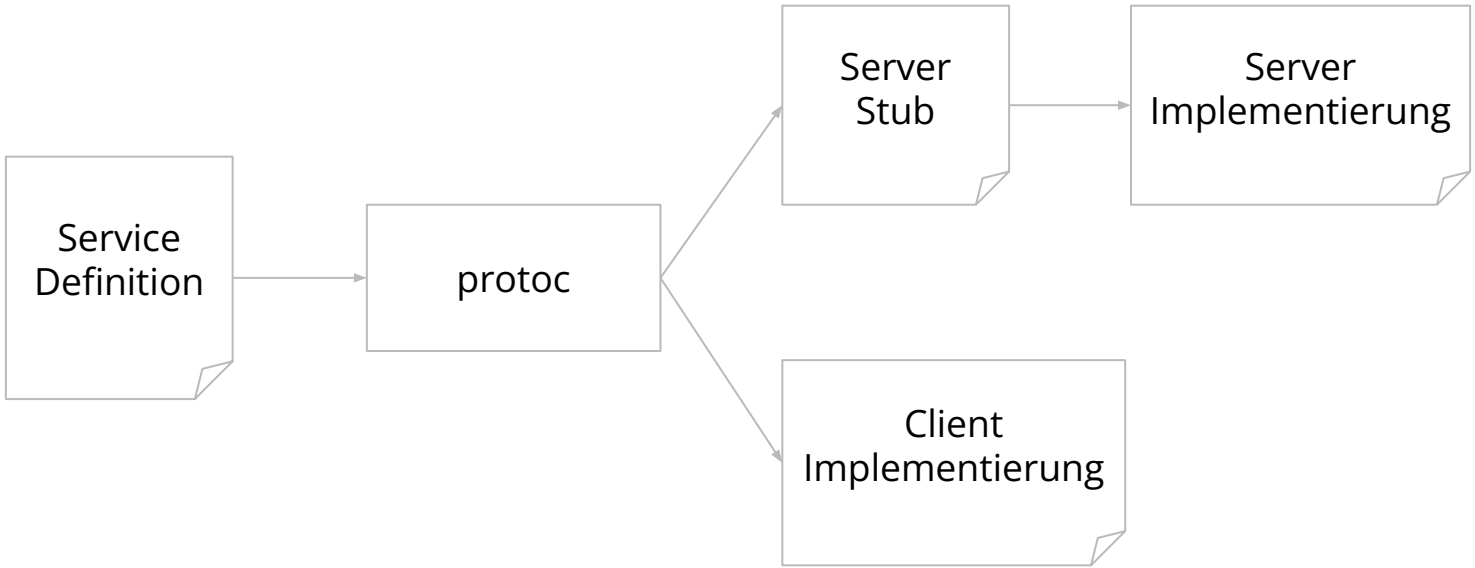
- Entwickelt von Google 2015
- HTTP/2 als Transportprotokoll
- Protocol Buffers als Serialisierungsformat
- Kommunikationsarten
  - Request-Response
  - Client-side Streaming
  - Server-side Streaming
  - Bidirectional Streaming
- Generator für Server- und Client-Implementierungen
  - Official: C, C++, C#, Dart, Go, Java, Kotlin, Node.js, ObjectiveC, PHP, Python, Ruby

gRPC

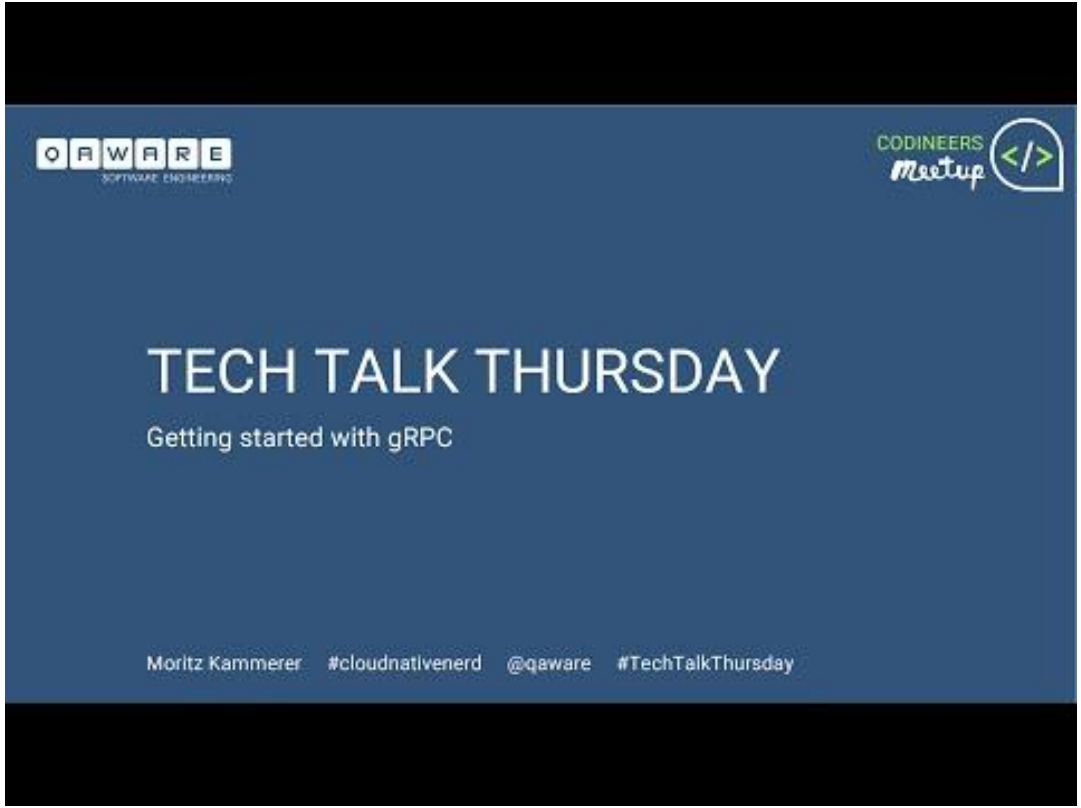


https://grpc.io/docs/what-is-grpc/introduction/

gRPC Workflow



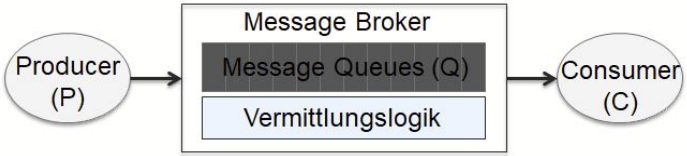
Demo



https://www.youtube.com/watch?v=Qw8bPpw8h-g

Flexible Kommunikationsmuster mit Messaging

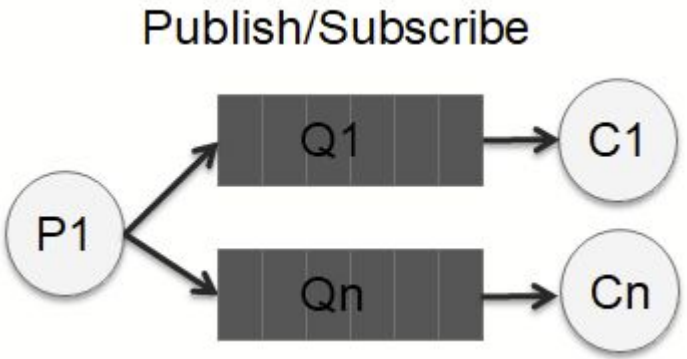
Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch

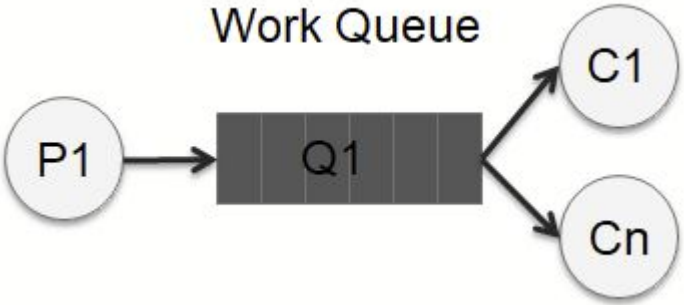


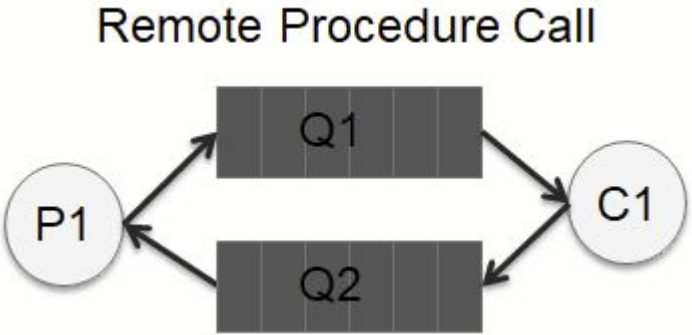
Entkopplung von Producer und Consumer

- Serviceschnittstelle: Format
- Messagebroker: keine Einschränkung für Format
- Sende- und Empfangszeitpunkt können beliebig lange auseinander liegen
- Horizontale Skalierbarkeit: Kann an mehrere Consumer ausgeliefert werden
- Lastverteilung: z.B. Auslieferung an Consumer, der am wenigsten zu tun hat
- Lastspitzen: Auslieferung der Nachricht kann verzögert werden, wenn alle Consumer überlastet sind
- Konfigurationsoptionen:
  - TTL der Nachricht
  - Zustellgarantie (min. 1 mal, exakt 1 mal, keine Garantie)
  - Transaktionalität
  - Priorität der Nachricht
  - Einhaltung der Reihenfolge

Messaging erlaubt mehrere Kommunikationsmuster





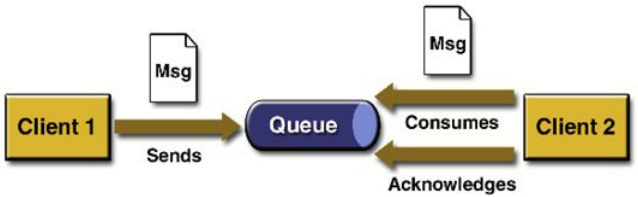


JakartaEE (JavaEE) Standard: JMS

JMS = Jakarta Messaging API (Java Messaging Service). Standardisierte API im Rahmen der Java-Enterprise-Edition-Spezifikation. Standardisiert nicht das Messaging-Protokoll.

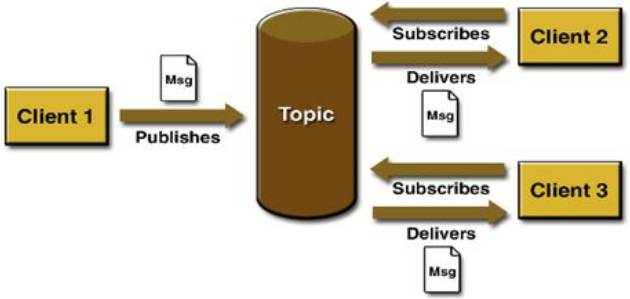
- 2002-2013: Version 1.1. Sehr stabil und weit verbreitet in der Java-Welt.
- Seit Mai 2013: Version 2.0 als Teil der JEE 7 Spezifikation
- An Version 3.0 wird gerade gearbeitet

Message Passing



- Ein Consumer pro Nachricht
- Erhalt der Nachricht wird bestätigt

Publish / Subscribe



- Mehrere Consumer pro Nachricht

AMQP: Standardprotokoll für Messaging

Problem: Message Broker waren proprietär und inkompatibel zueinander. Messaging über Firmengrenzen hinweg sehr schwierig

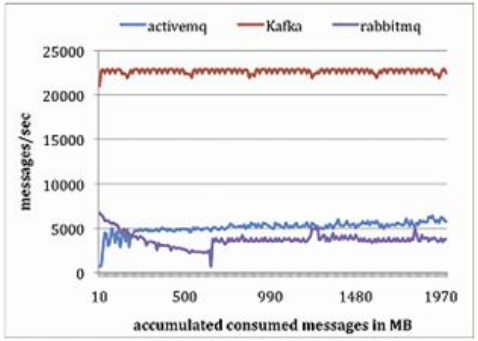
Lösung: AMQP. Standardisiert das Protokoll für Messaging. Version 1.0 seit Ende 2011.

- Standardisiert ein Netzwerk-Protokoll für die Kommunikation zwischen den Clients und den Message Brokern.
- Standardisiert ein Modell der verfügbaren APIs und Bausteine für die Vermittlung und Speicherung von Nachrichten (Producer, Exchange, Queue, Consumer).
- Unterstützung aller bekannter Messaging-Muster.

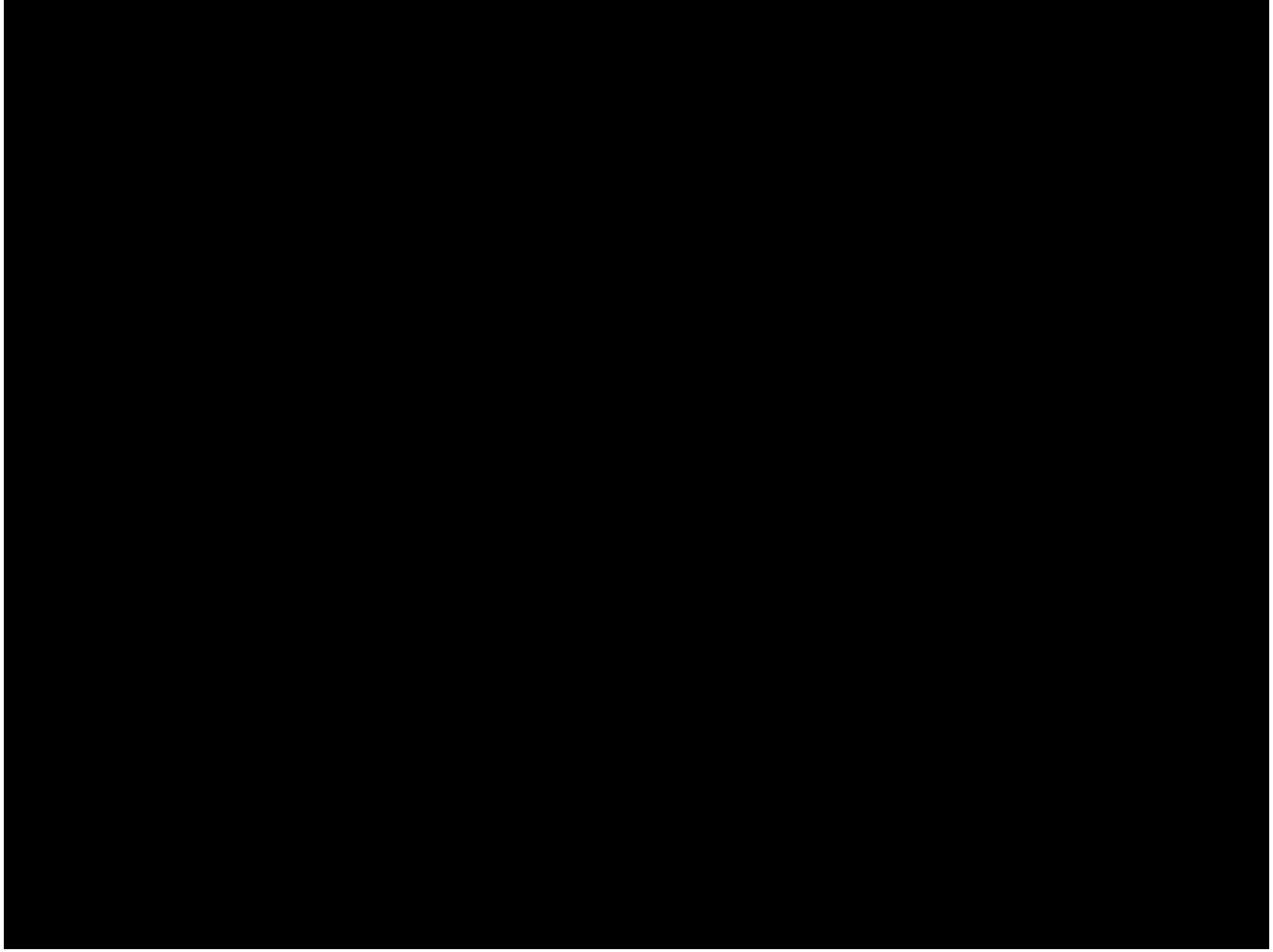
Im Standardisierungsgremium sind u.A. Cisco, Microsoft, Red Hat, Deutsche Börse Systems, IONA, Novell, Credit Suisse, JPMorganChase.

Apache Kafka

- Entwickelt bei LinkedIn und 2011 als Open Source Projekt veröffentlicht. Seit 2012 bei Apache Foundation.
- Kafka hat sich zum de-facto Standard in der Cloud für Messaging entwickelt, da Kafka hochgradig verteilbar und deutlich schneller als vergleichbare Lösungen ist:



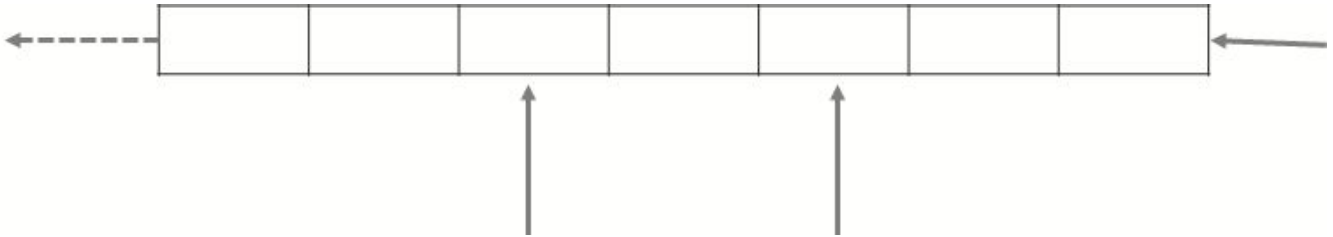
- Kafka ist so schnell, da es Betriebssystem-Mittel intelligent nutzt, ein effizientes Codierungsformat für Nachrichten besitzt und den Auslieferungszustand in den Clients hält.
- Kafka ist in Java und Scala geschrieben.
- Die Kafka API ist proprietär und orientiert sich an keinem Messaging-Standard.



https://www.youtube.com/watch?v=k-7lz6Ex354

Kafka basiert auf dem Konzept eines Event-Logs. Jeder Consumer hat einen eigenen Lese-Zeiger im Log.

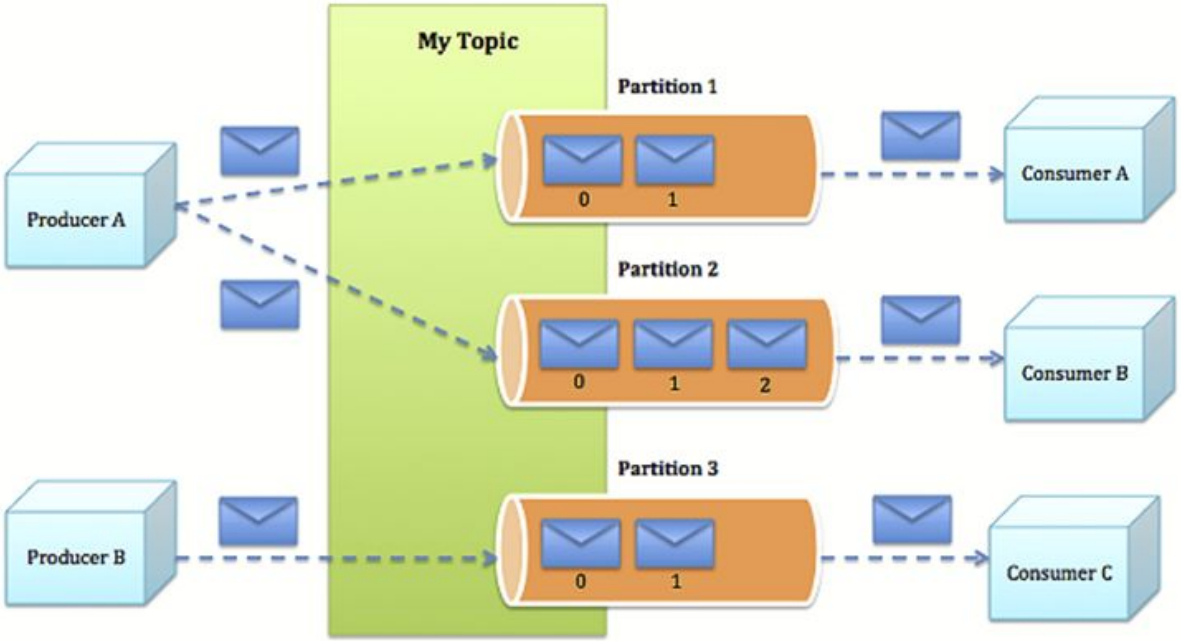
Alte Events werden gemäß definierter Kriterien gelöscht (z.B. Alter, max. Topic-Größe)



Neue Events werden immer am Ende des Topics angehängt

Zeiger auf letztes gelesenes Event eines Clients (verwaltet der Client selbst, Offset kann in Kafka gespeichert werden).

Der Event-Log in Kafka ist hochgradig verteilt.

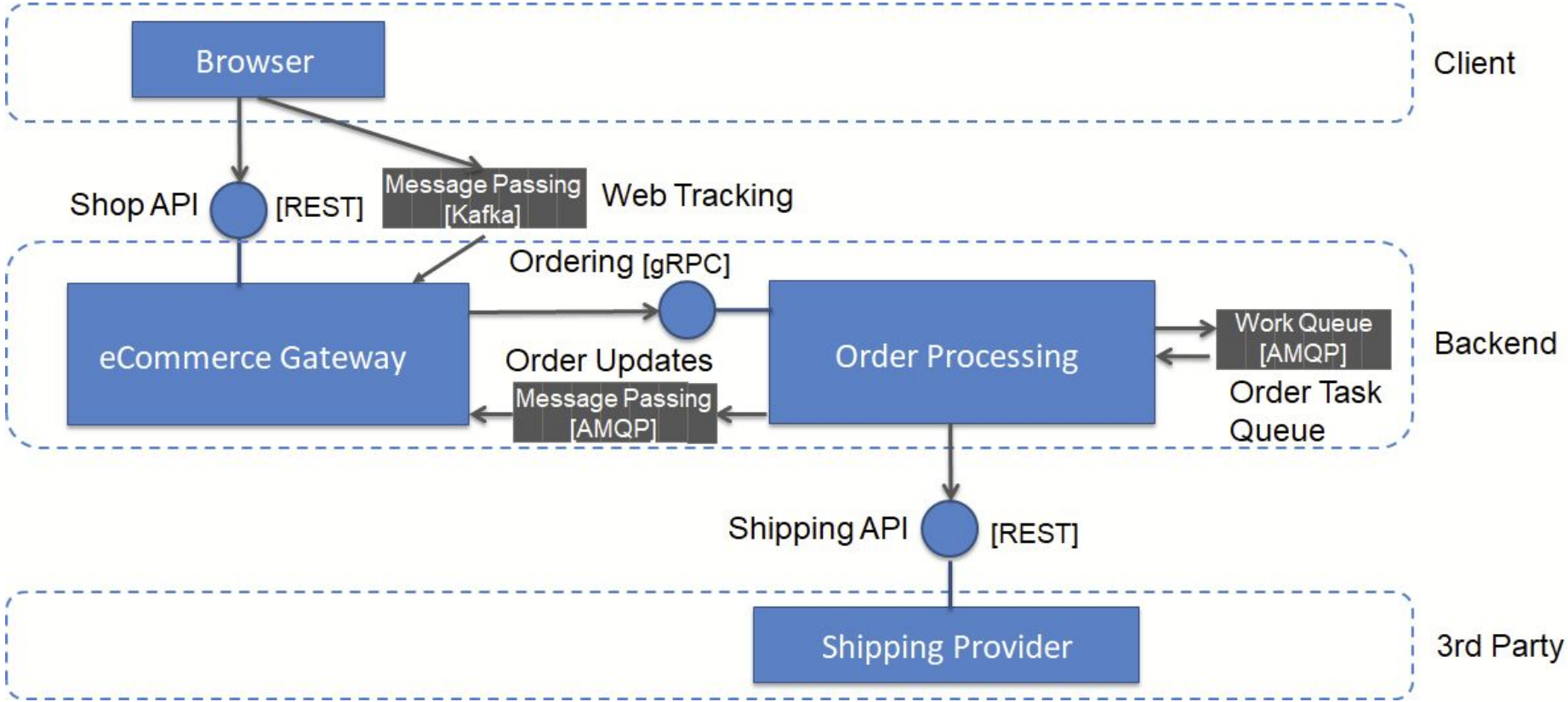


- Die Events in einem Topic werden aufgeteilt in Partitionen
- Die Partitionen werden verteilt auf die verfügbaren Broker-Instanzen
- Partitionen werden zur Fehlertoleranz repliziert

siehe:

- http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node
- http://www.infoq.com/articles/apache-kafka

Beispielhafte Architektur


Browser
Client
Shop API
[REST]
Message Passing
[Kafka]
Web Tracking
Ordering [gRPC]
eCommerce Gateway
Order Updates
Message Passing
[AMQP]
Order Processing
Work Queue
[AMQP]
Order Task
Queue
Backend
Shipping API
[REST]
Shipping Provider
3rd Party

Literatur

Literatur

Bücher:

- Patterns of Enterprise Application Architecture, Martin Fowler, 2002
- Computer Networks, Andrew Tanenbaum, 2010
- Inter-Process Communication, Hephaestus Books, 2011

Internet:

- Dissertation von Roy Fielding zu REST

  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RESTful Webservices

  http://www.ibm.com/developerworks/webservices/library/ws-restful
- Richardson Maturity Model

  https://martinfowler.com/articles/richardsonMaturityModel.html