

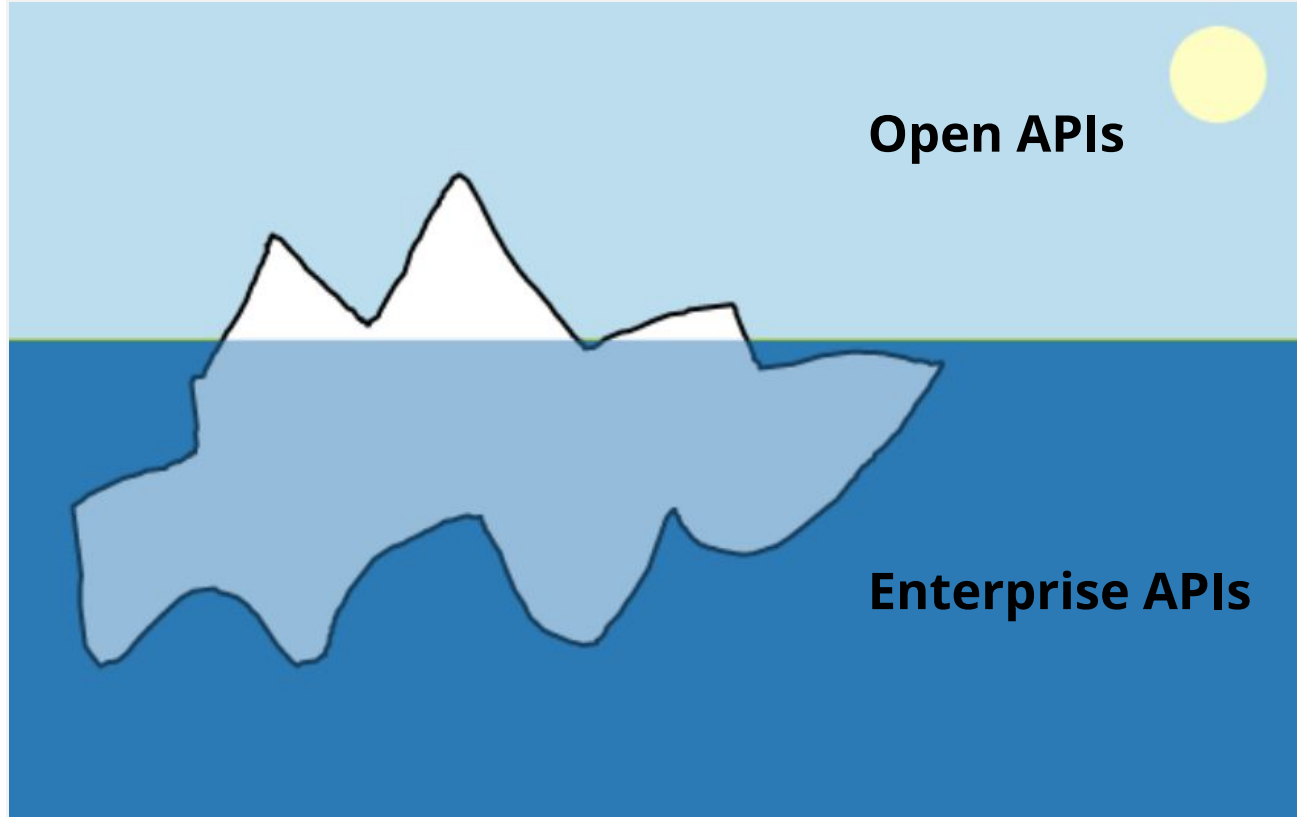
# Cloud Computing Kommunikation



QA|WARE

“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.

# Beispiel: Mercedes Benz Developer Portal



/developers

[APIs](#)

[SDKs](#)

[SUPPORT](#)

[INSPIRE](#)

[WHAT'S NEW](#)

[LOGIN](#)

## We move the world by data.

---

Welcome to Mercedes-Benz /developers!

We make our API products available to encourage the development of innovative business models. Join us in driving digital progress in the automobile industry.



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



QA|WARE

Was hat das mit Cloud Computing zu tun?

# „Früher“ lief Enterprise Software hinter einer Firewall

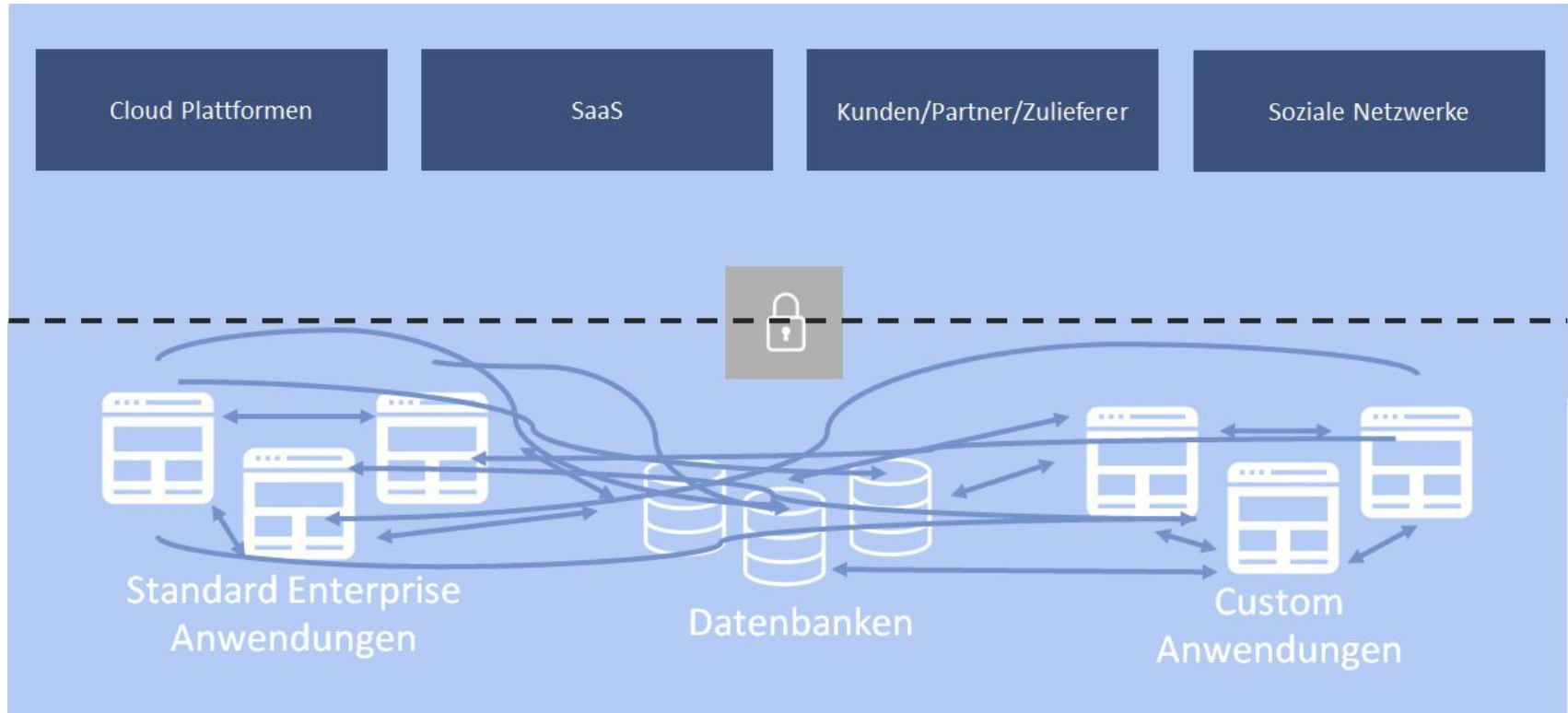


**„Früher“ lief Enterprise Software hinter einer Firewall. Die Komponenten kommunizieren intern.**



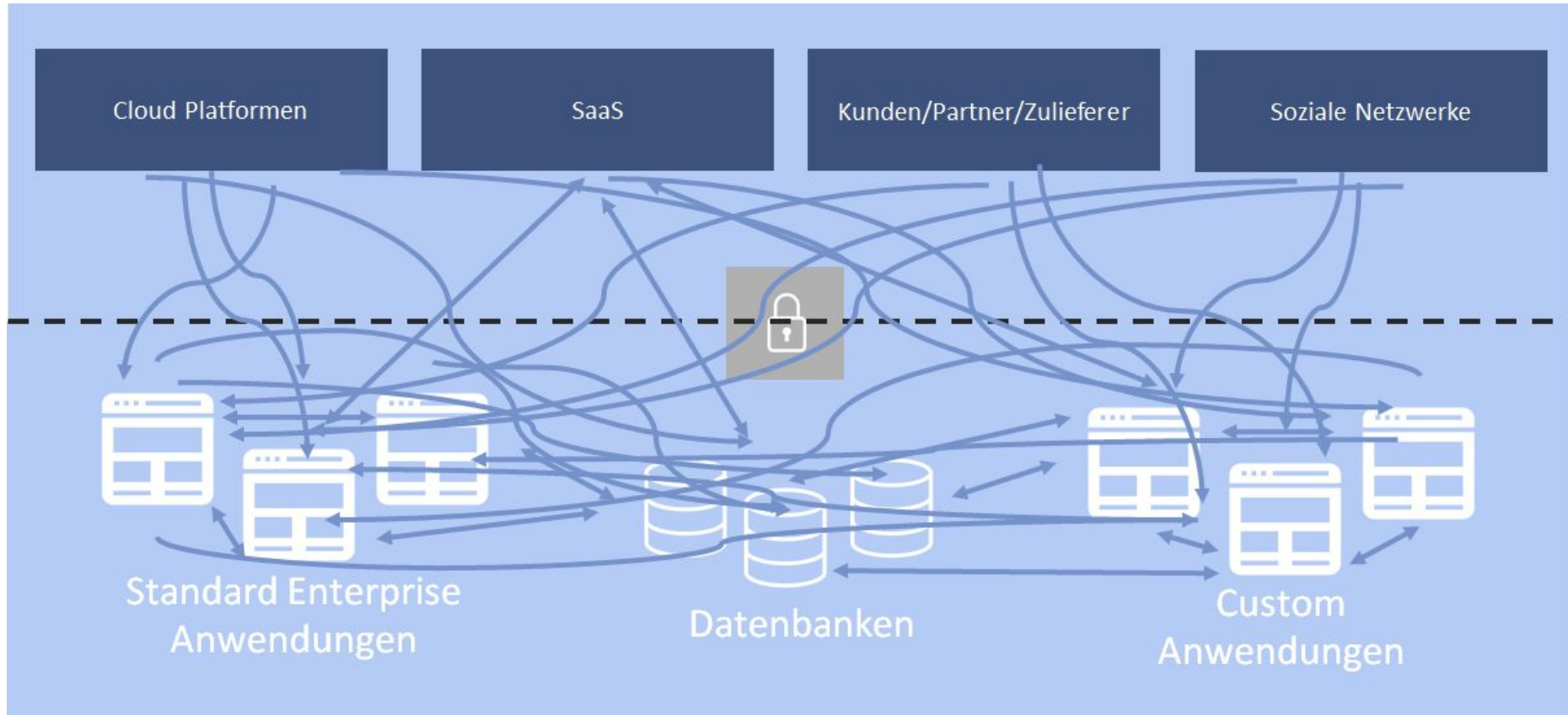


# Heute geht die Vernetzung noch deutlich weiter, auch über die Grenzen des Unternehmensnetzwerks hinaus:





## Aber eigentlich sieht es so aus:





QA|WARE

# Kommunikationsmodelle

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

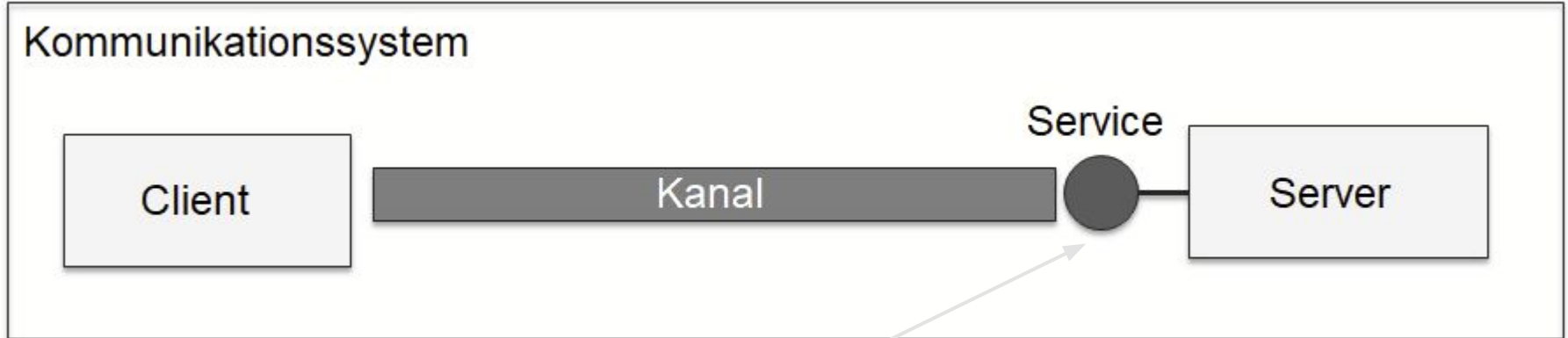
Kommunikationssystem = Infrastruktur für die Übermittlung von Informationen.



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

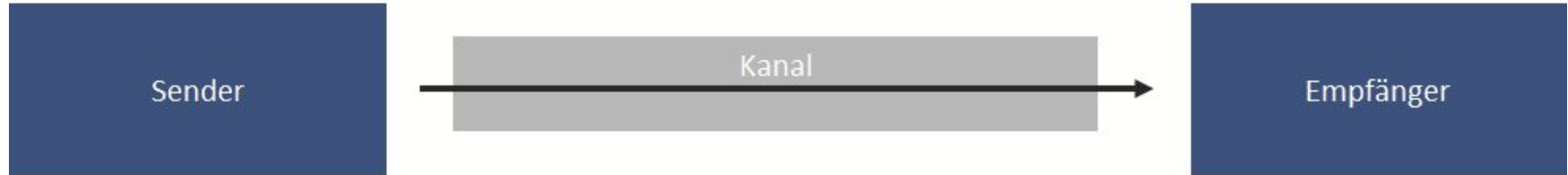


Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

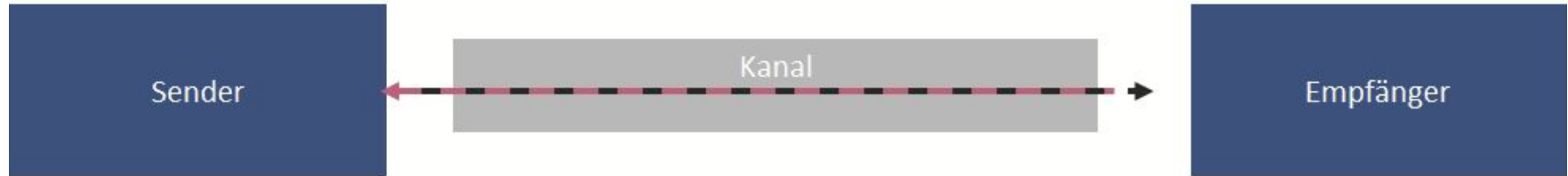
Eine **Serviceschnittstelle** ist ein Vertrag zwischen Nutzer und Anbieter über Syntax und Semantik der Service-Nutzung und enthält optional Zusicherungen in Hinblick auf den **Quality of Service**.

# Nutzungsmuster des Kanals (Richtung)

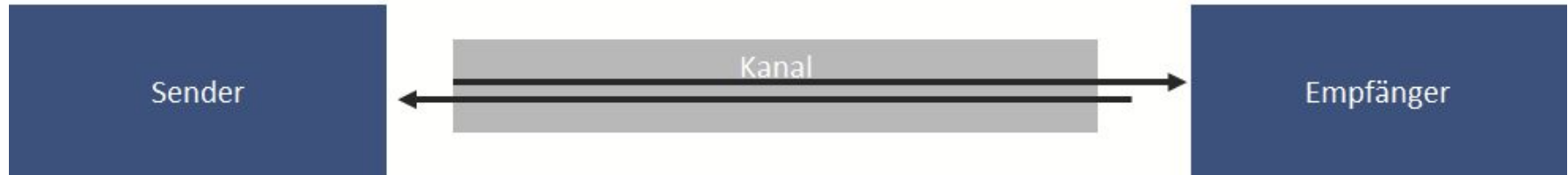
Simplex



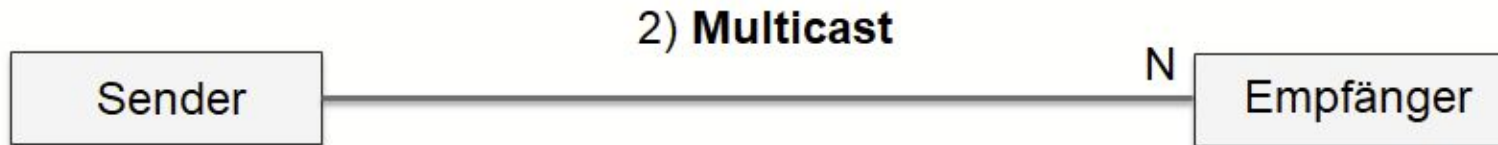
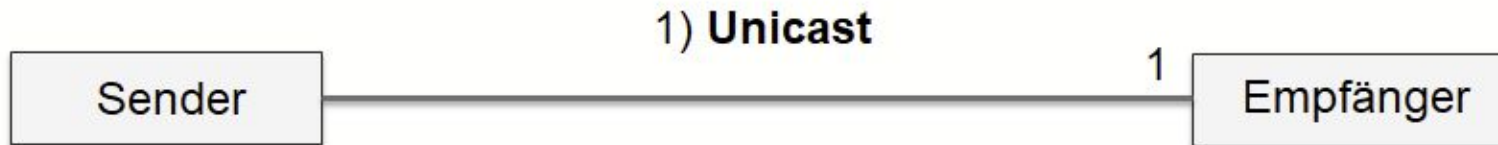
Halb-Duplex



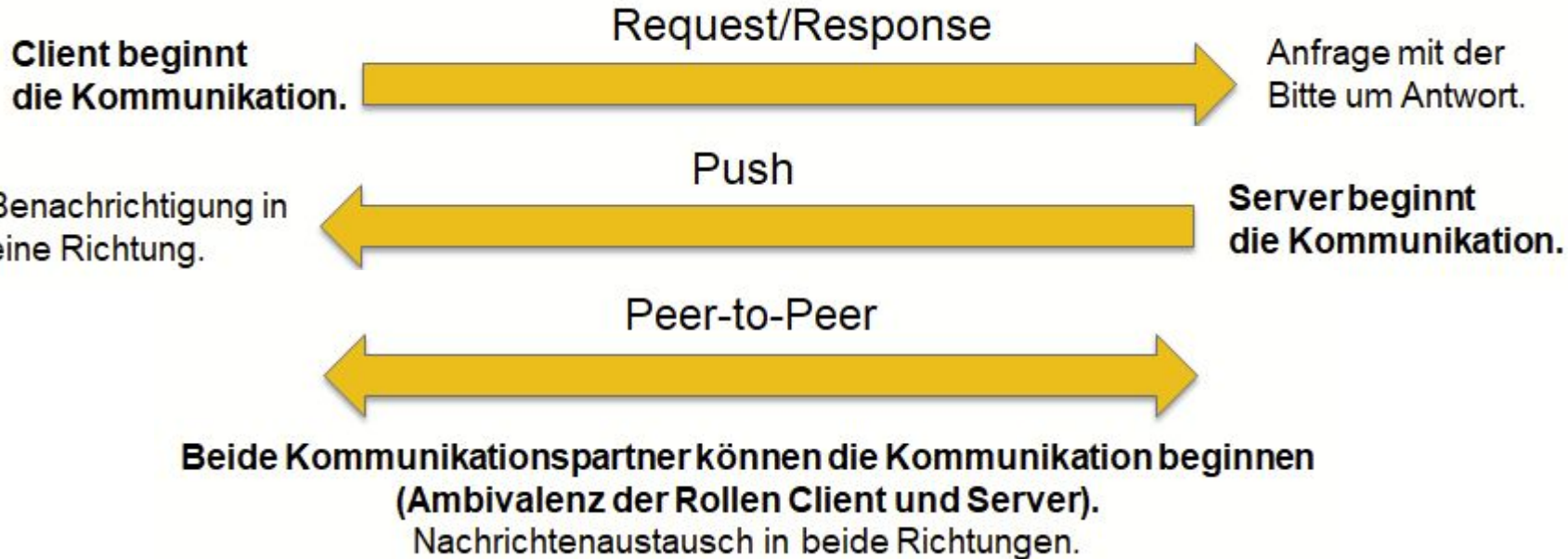
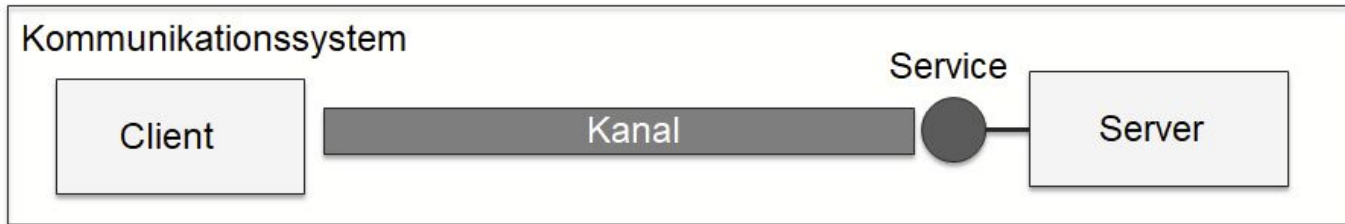
Voll-Duplex



# Kardinalität der Empfänger einer Nachricht.



# Wer beginnt mit der Kommunikation?







QAWARE

# HTTP

# Basis aller Cloud-Kommunikationstechnologien ist TCP und teilweise HTTP.

| HTTP (Hypertext Transfer Protocol) |  |
|------------------------------------|--|
| <b>Familie:</b>                    | Internetprotokollfamilie   |
| <b>Einsatzgebiet:</b>              | Datenübertragung, Hypertext u. a.  |
| <b>Port:</b>                       | 80/TCP   |
| HTTP im TCP/IP-Protokollstapel:    |  |
| <b>Anwendung</b>                   | <b>HTTP</b>  |
| <i>Transport</i>                   | TCP  |
| <i>Internet</i>                    | IP (IPv4, IPv6)  |
| <i>Netzzugang</i>                  | Ethernet Token Bus Token Ring FDDI ...   |
| <b>Standards:</b>                  | RFC 1945 <a href="#">↗</a> (HTTP/1.0, 1996)<br>RFC 2616 <a href="#">↗</a> (HTTP/1.1, 1999) |

## TCP

- Ab 1973 entwickelt und 1981 standardisiert.
- Zuverlässige Voll-Duplex Ende-zu-Ende Verbindung.
- Ein Endpunkt ist eine IP + Port.

## HTTP

- HTTP 1.0: 1989 am CERN entwickelt
- HTTP 1.1: Connection Pooling / Keep-Alive, HTTP-Pipelining, Methoden PUT und DELETE
- HTTP 2.0: Binär-Stream, Multiplexing, Verschlüsselung als Standard, div. Performance-Optimierungen, Push
- HTTP 3.0: Gerade in Entwicklung. Verwendet QUIC als Transport-Layer (das wiederum UDP verwendet)

# Kommunikationsmuster in HTTP

- **Request/Response:** Klassisch HTTP. Client schickt Request, Server antwortet mit Response. Client schickt wieder Request, etc.
- **Push:** Server Sent Events (SSE). Client schickt Request, Server antwortet mit Response. Nun kann Server aber auch weitere Nachrichten, ohne Requests vom Client, an den Client schicken
- **Push:** HTTP/2 Server Push. Analog zu Server Sent Events
- **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>

...



QA|WARE

# 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
- Plattformunabhängig
- Binärkodiert - speichereffizient und schnell
- Protobuf-Source wird von Protobuf-Compiler in Programmiersprache übersetzt





Format

# JSON

# Protobuf



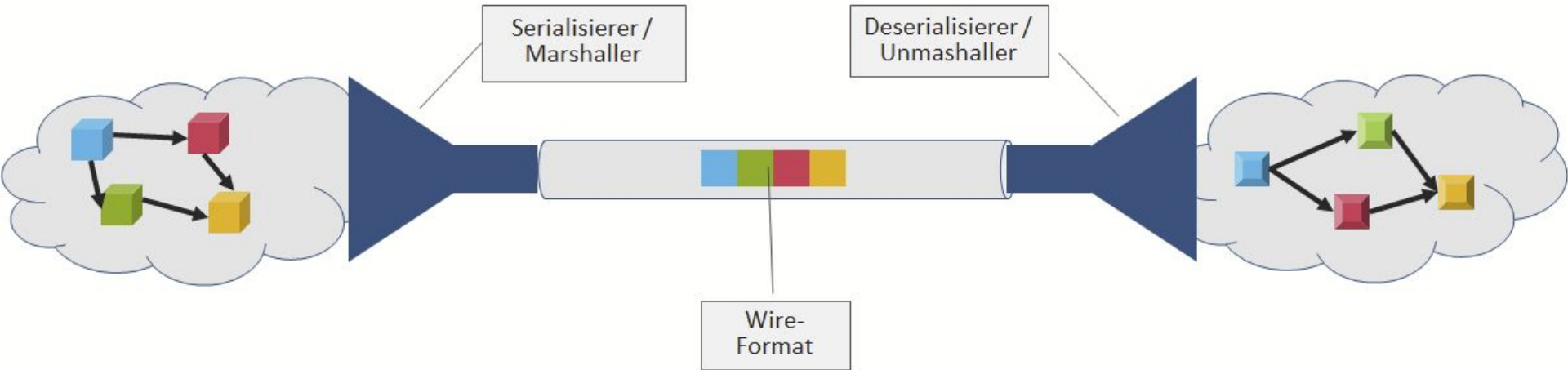
0:34 / 4:59



<https://www.youtube.com/watch?v=uGYZn6xk-hA>

# 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 |



QA|WARE

# Service-orientierte Request-Response-Kommunikation mit REST

# What are REST APIs?

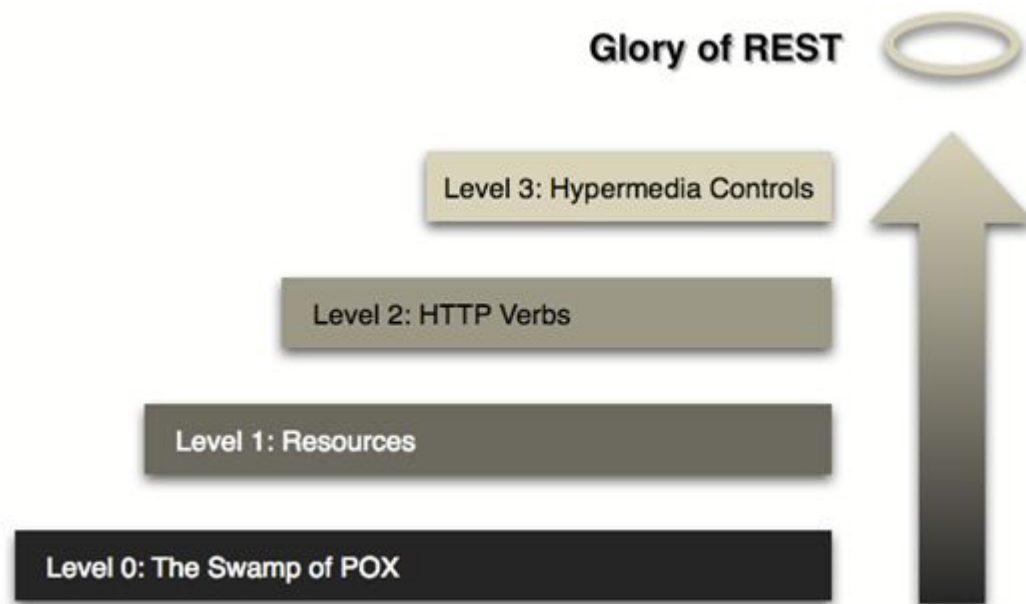
# REST ist ein Paradigma für Anwendungsservices auf Basis des HTTP-Protokolls.

- 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.

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



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:

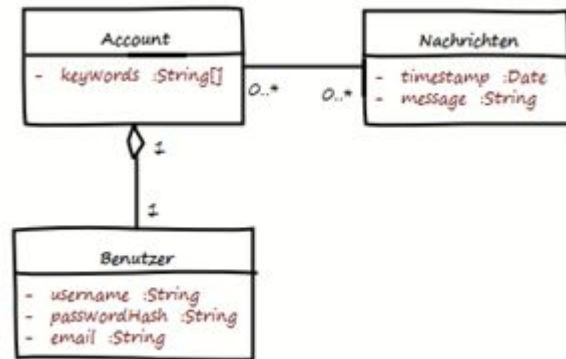
- Anwendungsfälle erheben
- Entitätenmodell erstellen
- Nun gibt es zwei Möglichkeiten:

## 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`

HTTP Verb

`@Produces("application/json")`

HTTP Content-Type

`public ResponseMessage getMessage({`

`@DefaultValue("Hallo") @QueryParam("salutation") String salutation,`

`@PathParam("name") String name) throws IOException {`

`ResponseMessage response = new ResponseMessage(new Date().toString(), salutation + " " + name);`

`return response;`

`}`

`}`

# 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);

    }

}
```



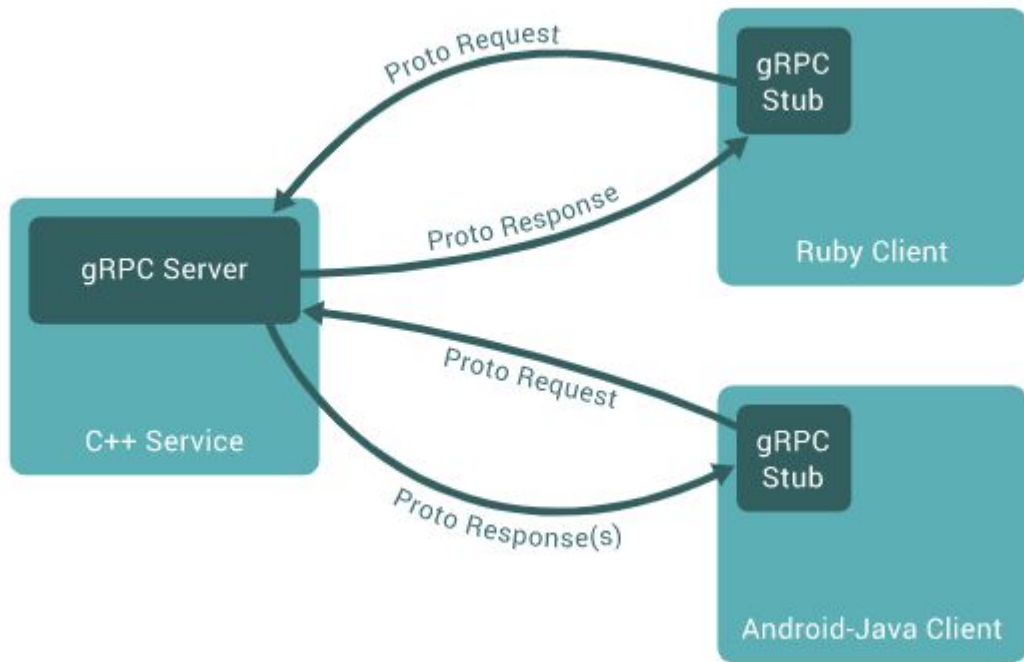
QA|WARE

# Service-orientierte Request-Response-Kommunikation mit gRPC

# gRPC

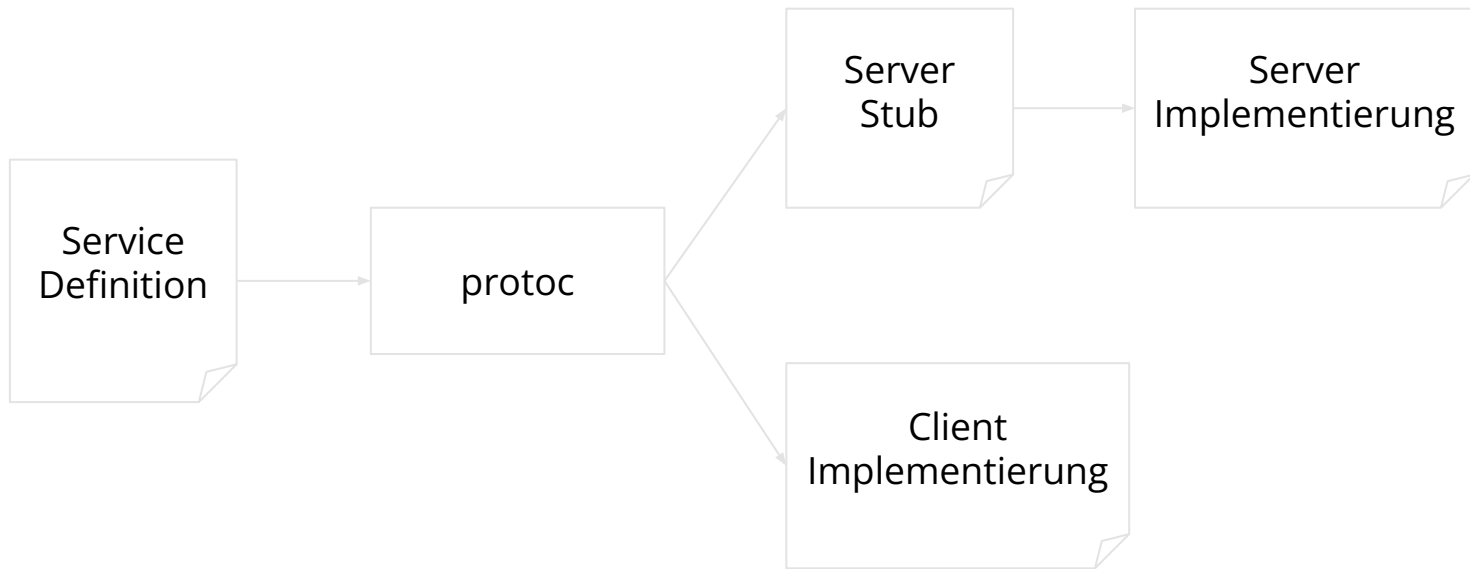
- Entwickelt von Google 2015
- HTTP/2 als Transportprotokoll
- Protocol Buffers als Serialisierungsformat
- Kommunikationsarten
  - Request-Response
  - Client-side Streaming
  - Server-side Streaming
  - Bidirectional Streaming
- Generator für Server- und Client-Implementierungen
  - Official: C, C++,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

# gRPC



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

# gRPC Workflow





# Demo



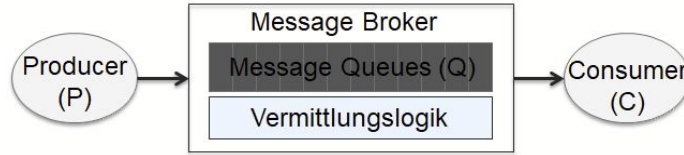
<https://www.youtube.com/watch?v=Qw8bPpw8h-g>



QA|WARE

# Flexible Kommunikationsmuster mit Messaging

# Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch



## Entkopplung von Producer und Consumer

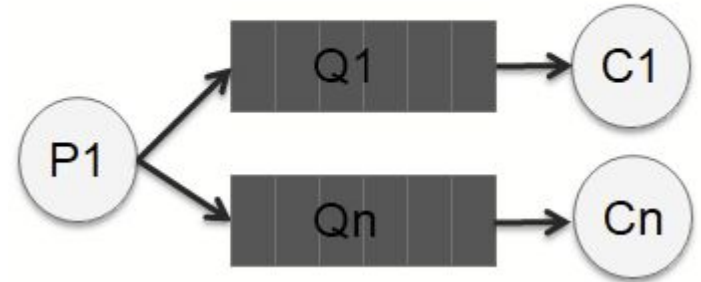
- Serviceschnittstelle: Format
- Messagebroker: keine Einschränkung für Format
- Vorteile
  - Sende- und Empfangszeitpunkt können beliebig lange auseinander liegen
  - Horizontale Skalierbarkeit: Kann an mehrere Consumer ausgeliefert werden
  - Lastverteilung: z.B. Auslieferung an Consumer, der am wenigsten zu tun hat
  - Lastspitzen: Nachrichtenauslieferung kann verzögert werden, wenn 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

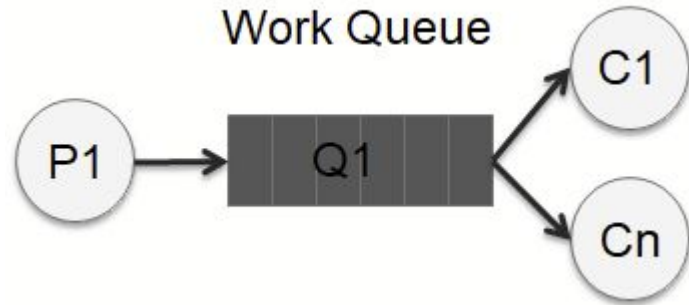
Message Passing



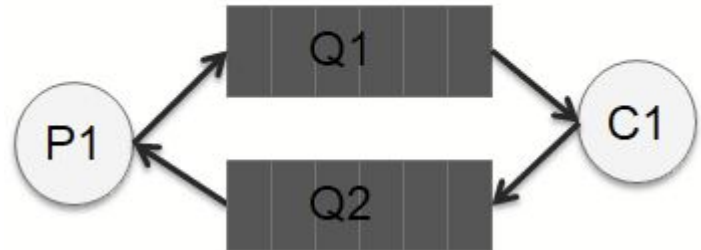
Publish/Subscribe



Work Queue



Remote Procedure Call

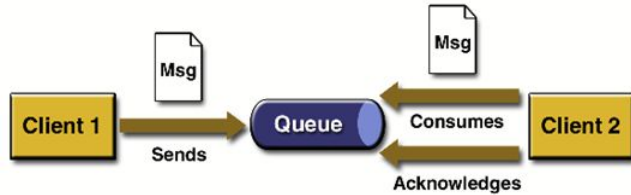


# JakartaEE (JavaEE) Standard: JMS

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

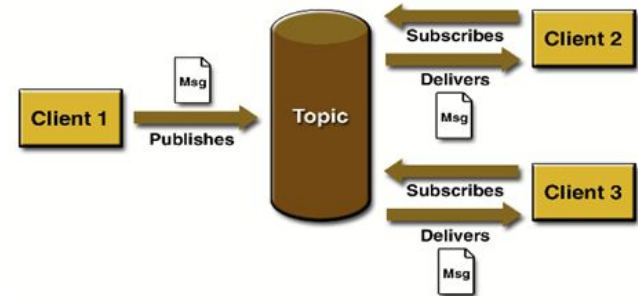
- 2002-2013: Version 1.1. Sehr stabil und weit verbreitet in der Java-Welt.
- Seit Mai 2013: Version 2.0 als Teil der JEE 7 Spezifikation
- An Version 3.0 wird gerade gearbeitet

## 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



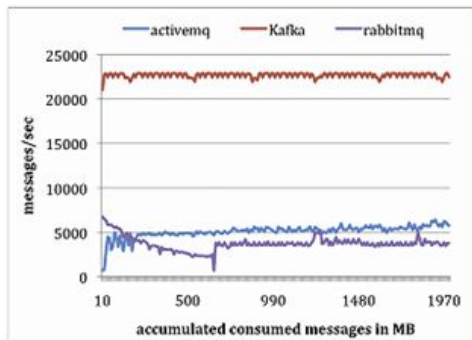
**Lösung:** AMQP. Standardisiert das Protokoll für Messaging. Version 1.0 seit Ende 2011.

- 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:



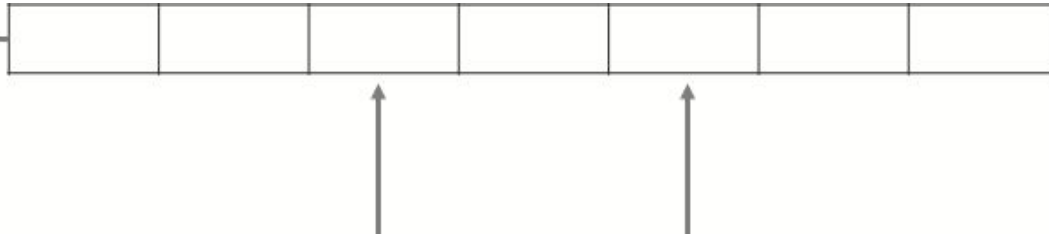
- Kafka ist so schnell, da es Betriebssystem-Mittel intelligent nutzt, ein effizientes Codierungsformat für Nachrichten besitzt und den Auslieferungszustand in den Clients hält.
- Kafka ist in Java und Scala geschrieben.
- Die Kafka API ist proprietär und orientiert sich an keinem Messaging-Standard.

<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.

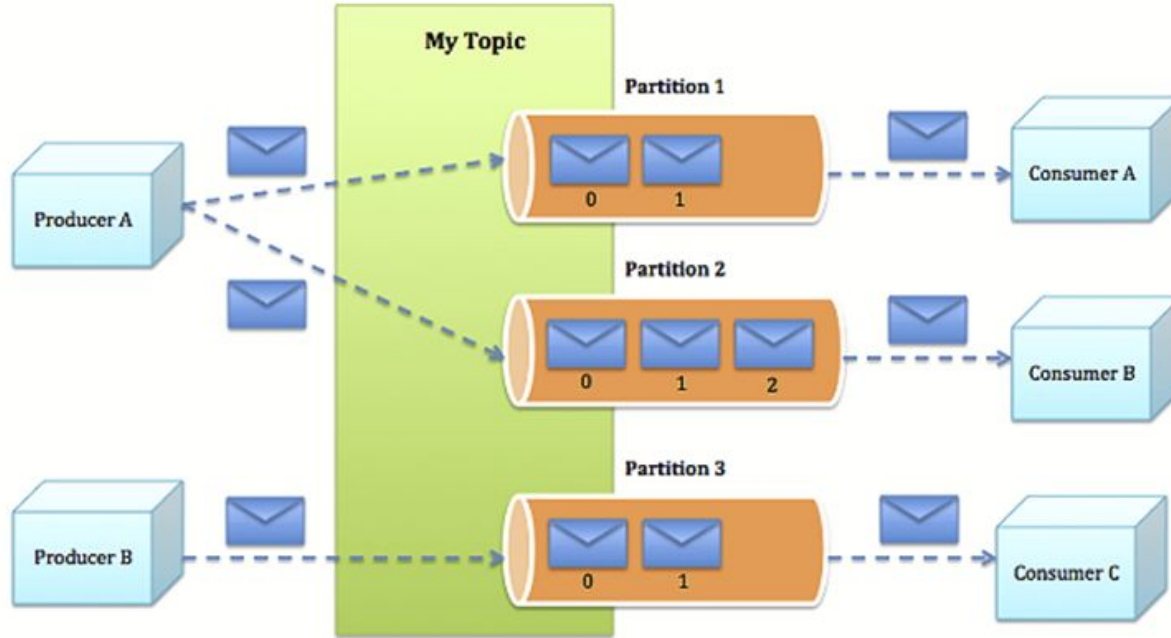
Alte Events werden gemäß definierter Kriterien gelöscht (z.B. Alter, max. Topic-Größe)



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

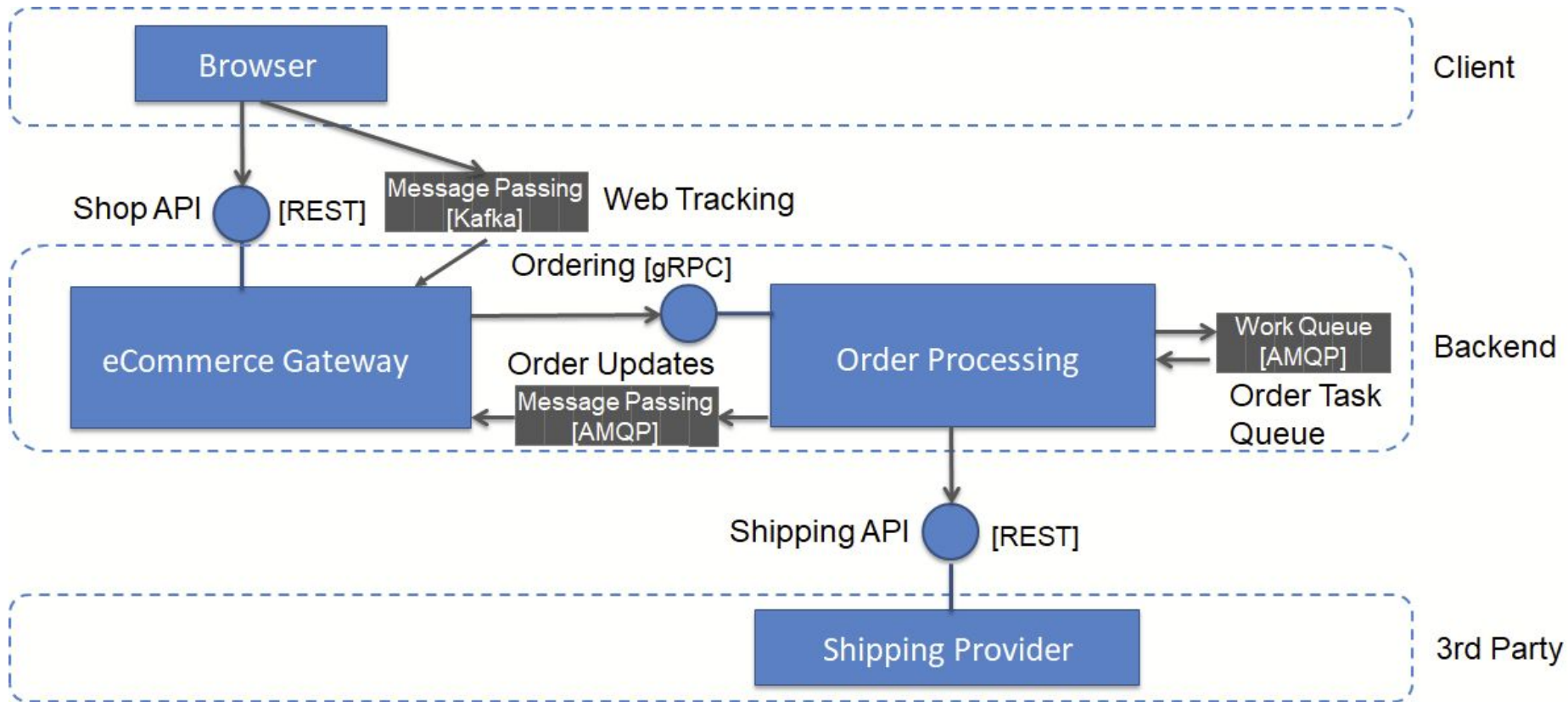
siehe:

- <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>



QA|WARE

# Beispielhafte Architektur





QA|WARE

# Literatur

# Literatur

## Bücher:

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

## Internet:

- Dissertation von Roy Fielding zu REST

[http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](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