Softwaretechnik 2

Kommunikationsorientierte Middleware

Fachhochschule Dortmund
University of Applied Sciences and Arts

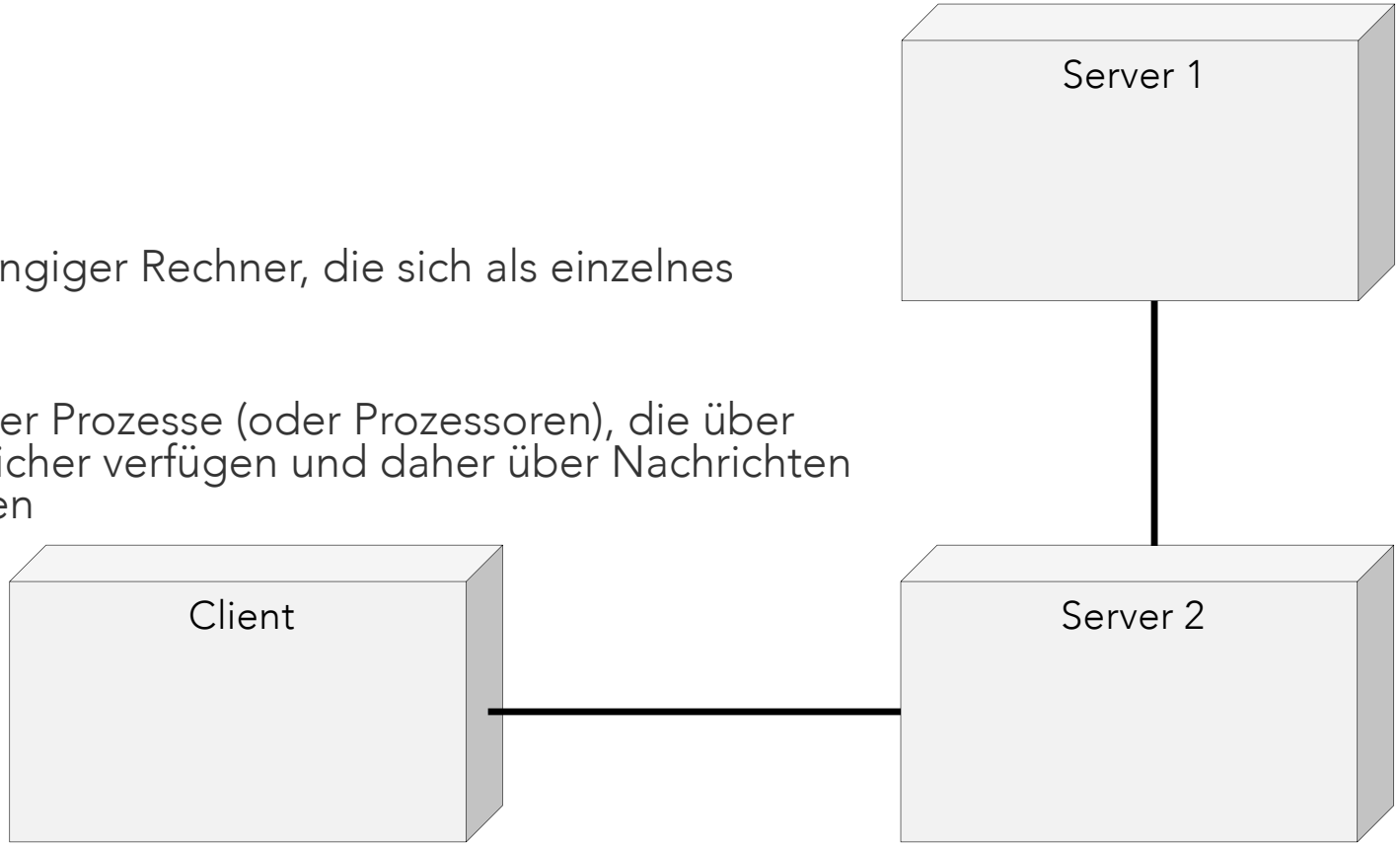
Struktur

Kommunikationsmodelle

Verteiltes System

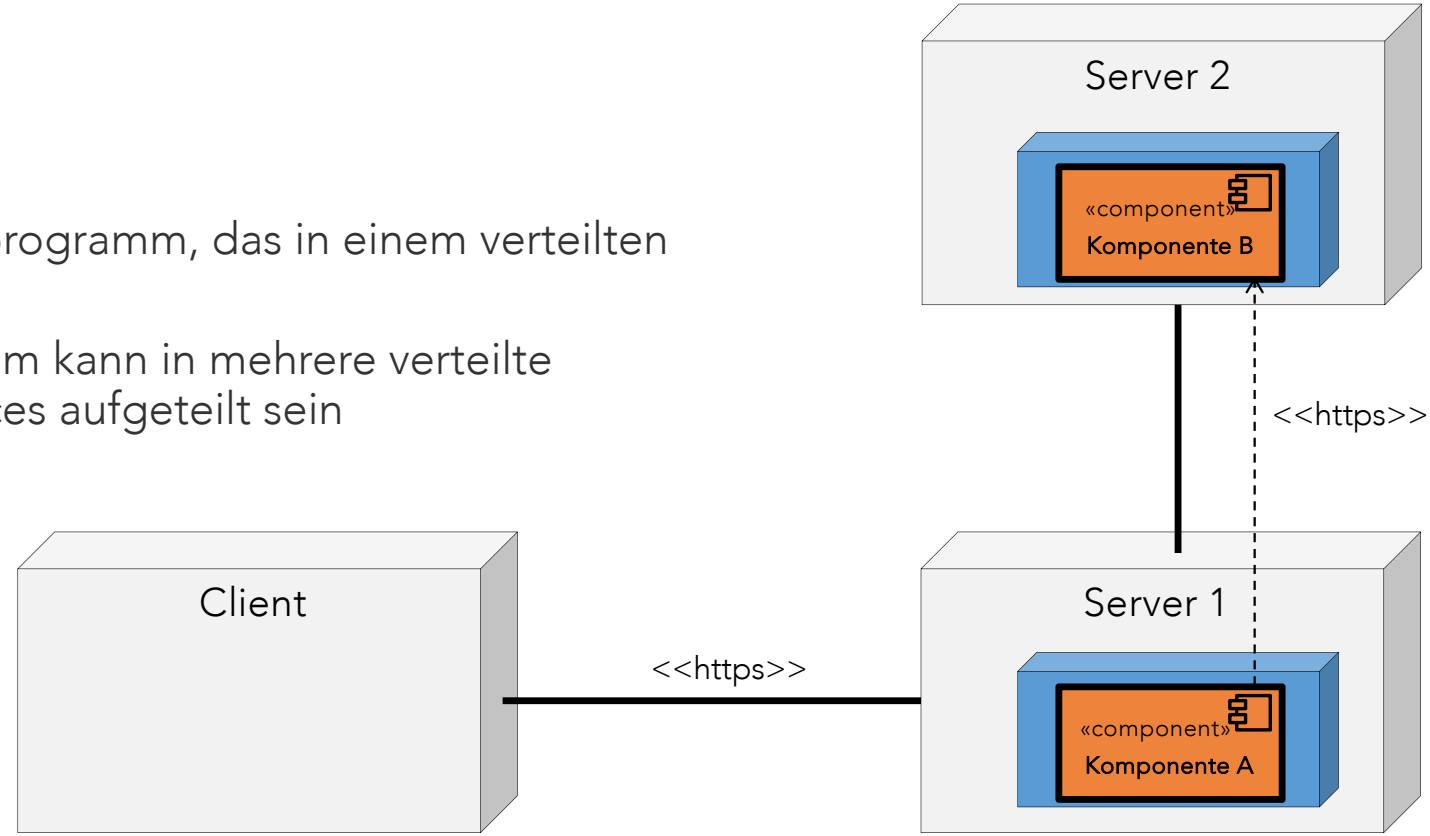
Verteiltes System

- nach Andrew Tanenbaum:
  Zusammenschluss unabhängiger Rechner, die sich als einzelnes System präsentieren
- nach Peter Löhr:
  eine Menge interagierender Prozesse (oder Prozessoren), die über keinen gemeinsamen Speicher verfügen und daher über Nachrichten miteinander kommunizieren



Verteilte Anwendung

- komplexes Anwendungsprogramm, das in einem verteilten System läuft
- das Anwendungsprogramm kann in mehrere verteilte Komponenten bzw. Services aufgeteilt sein



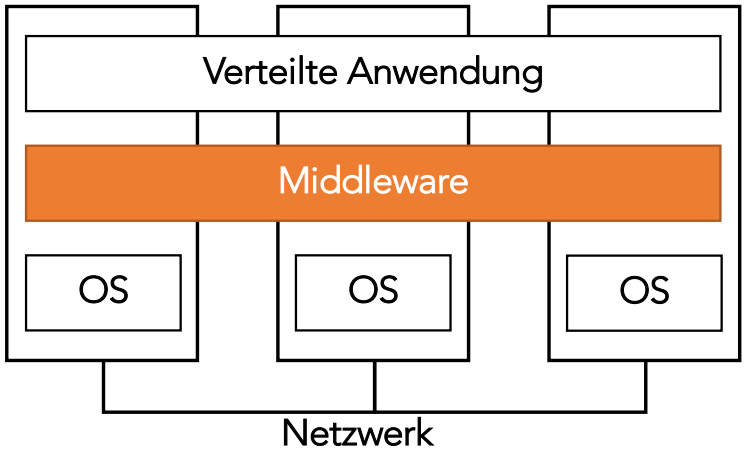
Kommunikationsmodelle

Kommunikationsorientierte Middleware

Middleware

- Allgemein:
  Middleware als Vermittlungsschicht zwischen Anwendung und Betriebssystem

- Bei verteilten Anwendungen:
  Bereitstellung einer Kommunikationsinfrastruktur mit dem Ziel alle Aspekte der Netzwerkprogrammierung zu abstrahieren



Kommunikationsmodelle

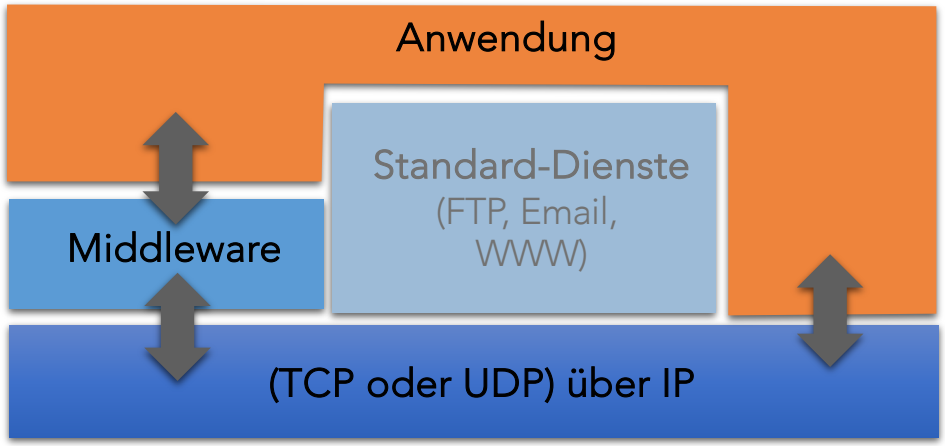
Kommunikationsorientierte Middleware

Middleware

- Sehr bequemer Weg zur Entwicklung von Anwendungen
- Datenrepräsentation, Objektlokalisierung etc. muss nicht von der Anwendung gemacht werden
- Oft viel Overhead

Direkte Netzprogrammierung

- Direkte Kontrolle aller Transportparameter
- Größere Flexibilität bei der Entwicklung neuer Protokolle
- Kann in vielen Fällen bessere Performance bringen
- Großes Problem: Datenrepräsentation



Kommunikationsmodelle

Synchron vs. Asynchron

- Unterschiedliche Kommunikationsmodelle für Middleware in verteilten Systemen:
  - Synchrone Kommunikation
  - Asynchrone Kommunikation

	prozedural	objektorientiert
synchron	Entfernte Prozeduraufrufe (RPC)	Entfernte Methodenaufrufe (RMI)
asynchron	Nachrichtenorientiertes Modell (Message Queuing)	

Struktur

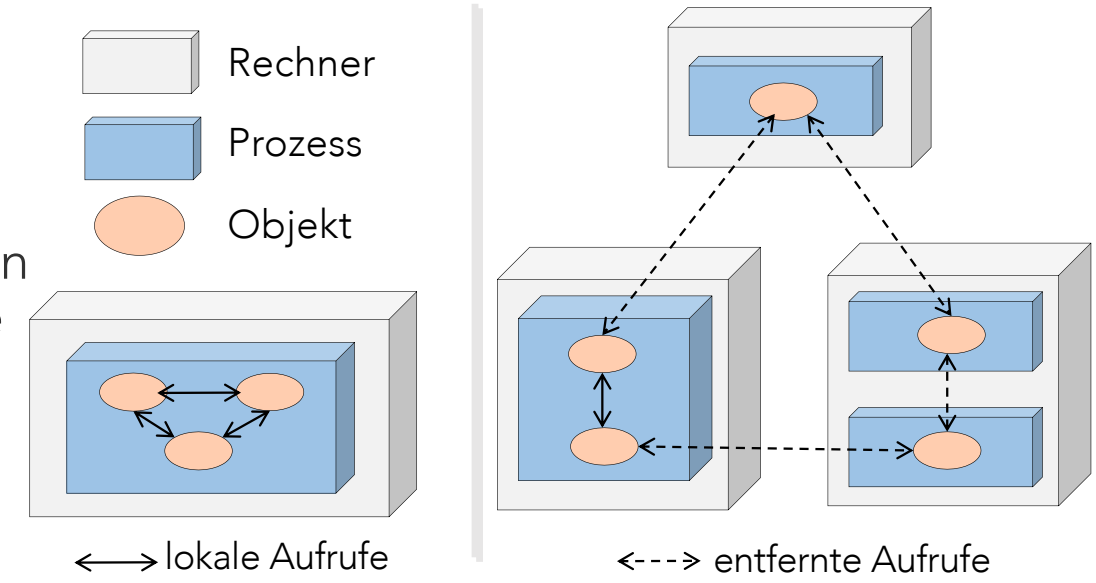
Synchrone Kommunikation

Lokales versus verteiltes Objektmodell

- Methodenaufrufe zwischen Objekten in entfernten/verschiedenen Prozessen

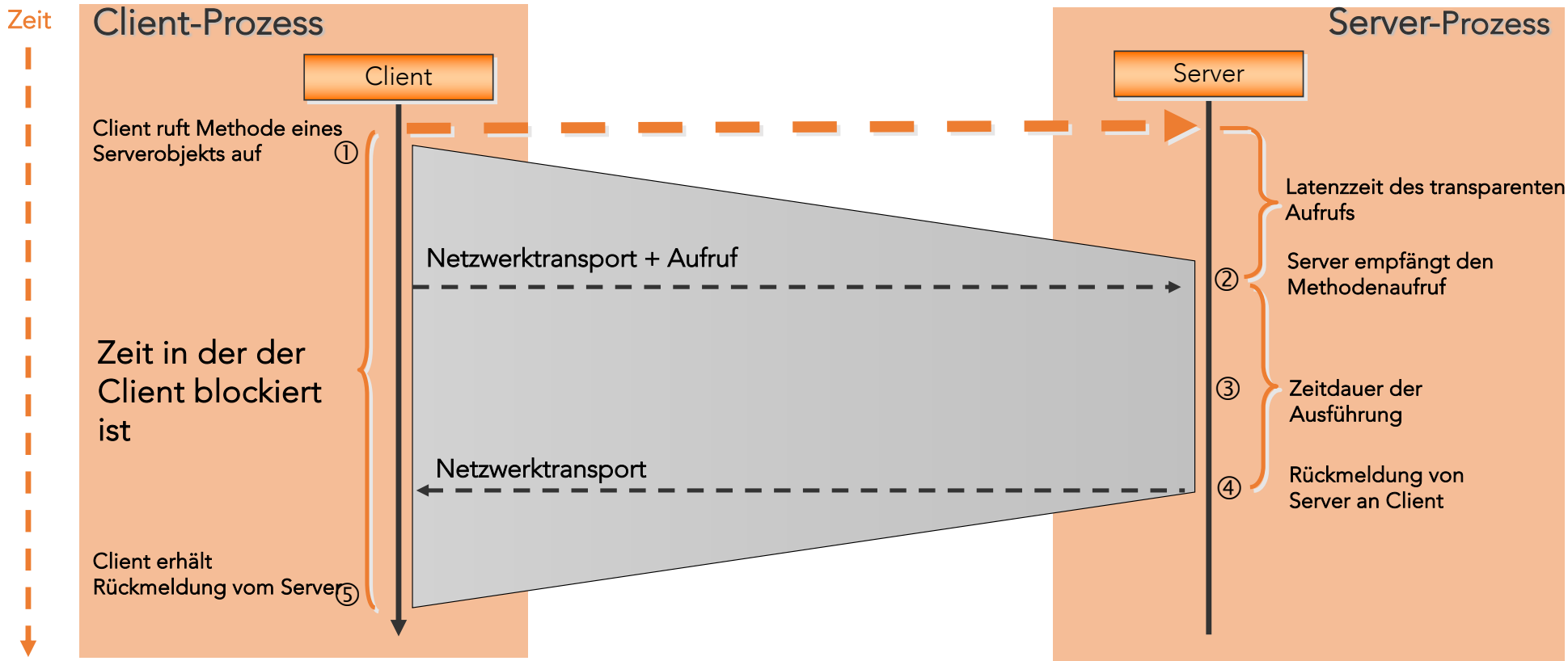
  → entfernte Methodenaufrufe
  (Remote Method Invocation, RMI)

- Kommunikationspartner (Prozesse) sind beim Senden oder beim Empfangen von Informationen solange blockiert, bis die Kommunikation abgeschlossen ist



Synchrone Kommunikation

Kommunikationsablauf



Synchrone Kommunikation

Lokales versus verteiltes Objektmodell

Lokales Objektmodell	Verteiltes Objektmodell
Aufruf an Objekten	Aufruf an Interfaces
Parameter und Ergebnisse als Referenzen	Parameter und Ergebnisse als Kopien
Alle Objekte fallen zusammen aus	Einzelne Objekte fallen aus
Keine Fehlersemantik	Komplizierte Fehlersemantik (Referenzintegrität, Netzfehler, Sicherheit etc.)

Struktur

Asynchrone Kommunikation

Nachrichtenorientiertes Modell

- Asynchrone Kommunikation ist durch das zeitlich versetzte Senden und Empfangen von Informationen gekennzeichnet

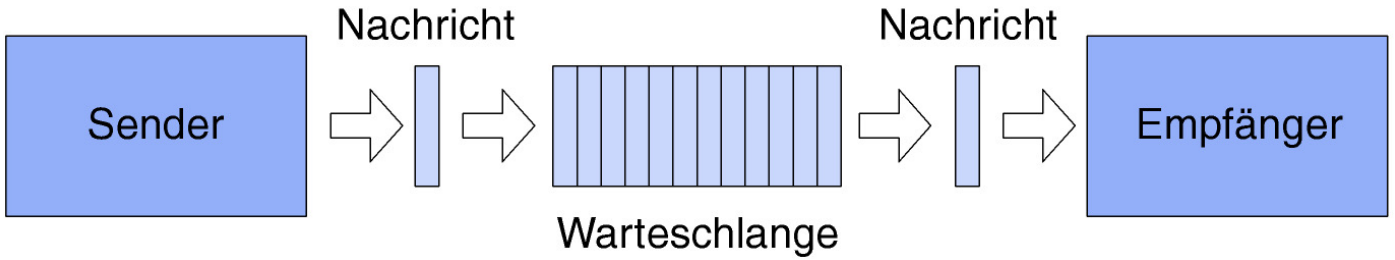
→ Keine Blockierung des Senders

- Kommunikation auf Basis des nachrichtenorientierten Modells ermöglicht den asynchronen Transport von Daten zwischen Prozessen
- Die Daten werden innerhalb von Nachrichten (Messages) übertragen
- Verschiedene Quality of Service Eigenschaften werden unterstützt

Asynchrone Kommunikation

Nachrichtenorientiertes Modell

- Grundlage für das nachrichtenorientierte Modell sind Warteschlangen:
  - Ein Sender stellt eine Nachricht in die Warteschlange ein
  - Der Empfänger nimmt die Nachricht aus der Warteschlange entgegen sobald er empfangsbereit ist
- Die Warteschlange arbeitet nach dem FIFO Prinzip
- Ziel: weitgehende Entkopplung von Sender und Empfänger

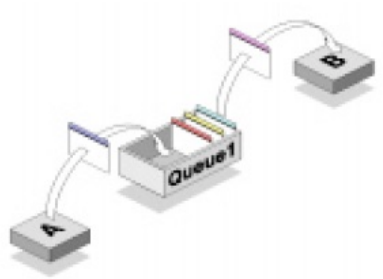


Asynchrone Kommunikation

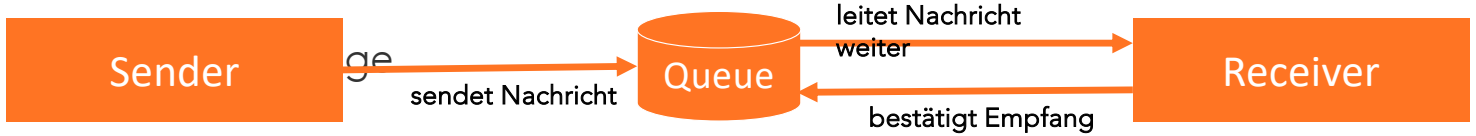
Nachrichtenorientiertes Modell – Point-to-Point

Point-to-Point

Die Nachrichtenübertragung findet zwischen zwei festgelegten Prozessen (Anwendungen) statt



- Die häufigste Umsetzung des Point-to-Point ist die einfache asynchrone Kommunikation:
  1. Sender erstellt Nachricht und sendet diese zur Queue
  2. Queue leitet Nachricht weiter an Receiver
  3. Receiver empfängt Nachricht und sendet eine Empfangsbestätigung an die Queue
  4. Receiver [illegible] ge

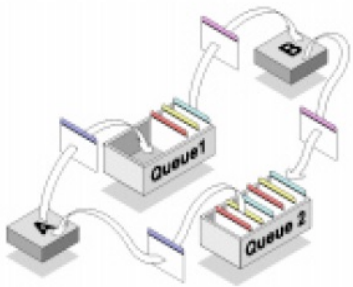


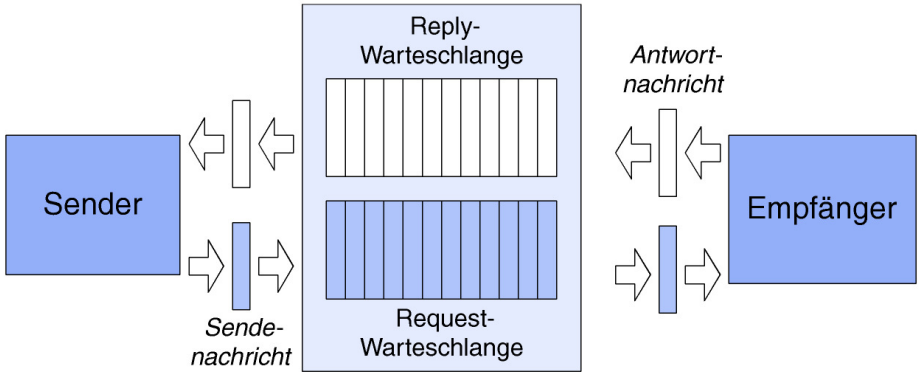
Asynchrone Kommunikation

Nachrichtenorientiertes Modell – Point-to-Point

Point-to-Point

- Das Request-Reply Modell ist eine spezielle Art des Point-to-Point Modells, das mit zwei Nachrichtenschlangen für Nachrichten und Antworten arbeitet
- Ermöglicht die **synchrone Kommunikation** (fachlich) über eine **asynchrone Middleware** (technisch)



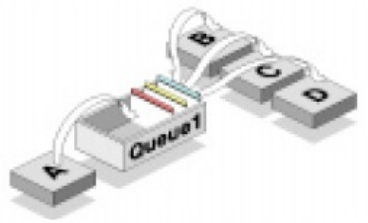


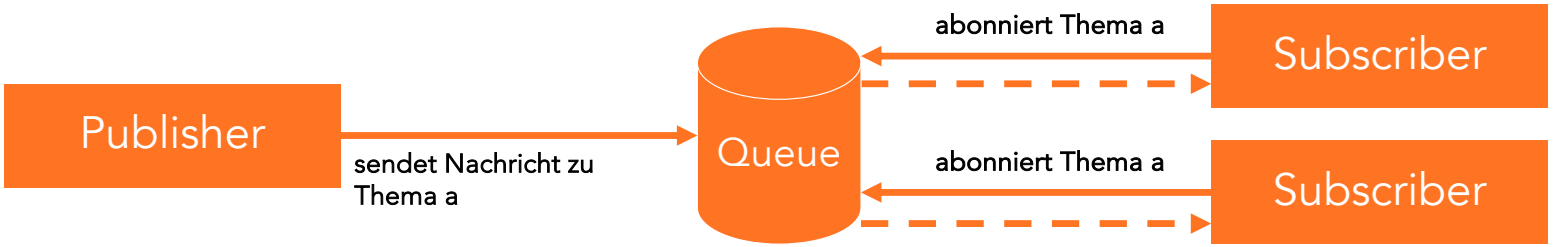
Asynchrone Kommunikation

Nachrichtenorientiertes Modell – Publish-Subscribe

Publish-Subscribe

- Eine Nachricht wird an alle erreichbaren Prozesse versendet:
  - Anwendungen erhalten Rollen als Publisher oder Subscriber
  - Subscriber abonnieren Nachrichten zu einem Thema
  - Publisher veröffentlichen Nachrichten zu einem Thema
  - Message-Broker übernimmt die Vermittlung der Nachrichten





Asynchrone Kommunikation

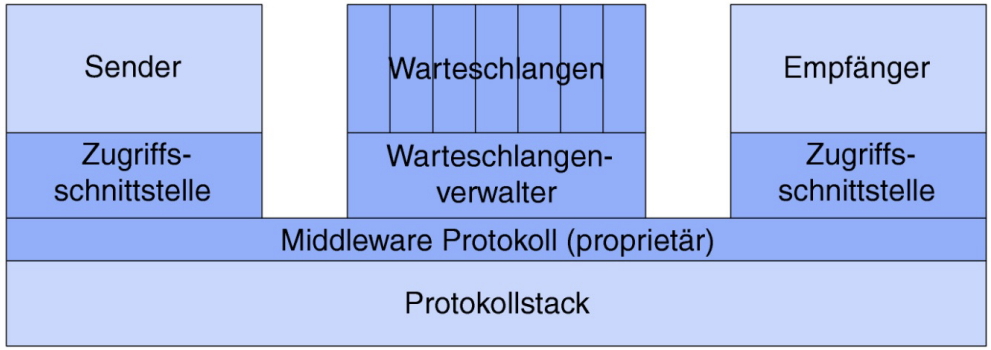
Vorteile

- Die kommunizierenden Prozesse können zu unterschiedlichen Zeiten laufen
- Die Prozesse kommunizieren ohne direkten Kontakt unter Verwendung von Queues
- Queues dienen als Puffer - der Empfängerprozess muss zum Versendezeitpunkt nicht zwangsweise laufen
- Einfaches, klares Kommunikationsmodell
- Laufzeitabhängigkeiten werden minimiert bzw. aufgehoben
- Netzwerkverbindungen werden reduziert
- Programme sind weniger anfällig gegenüber Netzwerkfehlern
- Für lose gekoppelte Systeme sehr gut geeignet

Asynchrone Kommunikation

Message-oriented Middleware

- Message-oriented Middleware (MoM) bezeichnet die Middleware-Technologie zur Umsetzung des nachrichtenorientierten Kommunikationsmodells
- MoM bietet neben der Abwicklung der asynchronen Kommunikation weitere Mechanismen und Dienste:
  - Unterstützung der verschiedenen Messaging Mode
  - Nachrichtenschlangenverwaltung (z.B. Zuordnung v ankommenden Nachrichten zu Nachrichtenschlang
  - Verbindungsmanagement
  - Zusicherungen QoS (z.B. Timeout, Priorität)



Asynchrone Kommunikation

JMS

- Java Message Service (JMS) ist eine offene und einheitliche API für MoM
- JMS ist herstellerunabhängig und spezifiziert ausschließlich Schnittstellen, welche von Drittanbietern implementiert werden müssen
- JSM ist Teil von Java EE und liegt aktuell in der Version 2.0 vor
  - JMS 3.0 ist als Teil von Jakarta EE geplant
- JMS Implementierungen (Auszug):
  - ActiveMQ
  - HornetQ
  - OpenJMS
  - Qpid

Asynchrone Kommunikation

JMS – Komponenten

- JMS Provider
  - Implementiert die JMS Schnittstellen, welche durch die Spezifikation vorgegeben warden
  - Kann zusätzlich auch noch weitere, nicht JMS-spezifische, Funktionalitäten bieten
- JMS Client
  - Software(-komponenten), welche Nachrichten erstellen oder konsumieren können
- Message
  - Beinhaltet die eigentlichen Informationen, die zwischen den Clients ausgetauscht werden
  - Verfügen über ein einfaches, aber sehr flexibles Format
    - Header: u.a. zur Identifikation und zum Routen der Messages
    - Properties (optional): zusätzliche Informationen die nicht durch den Header abgedeckt werden
    - Body (optional)

Asynchrone Kommunikation

JMS – Nachrichtenkanäle

JMS unterstützt zwei Arten von Nachrichtenkanälen

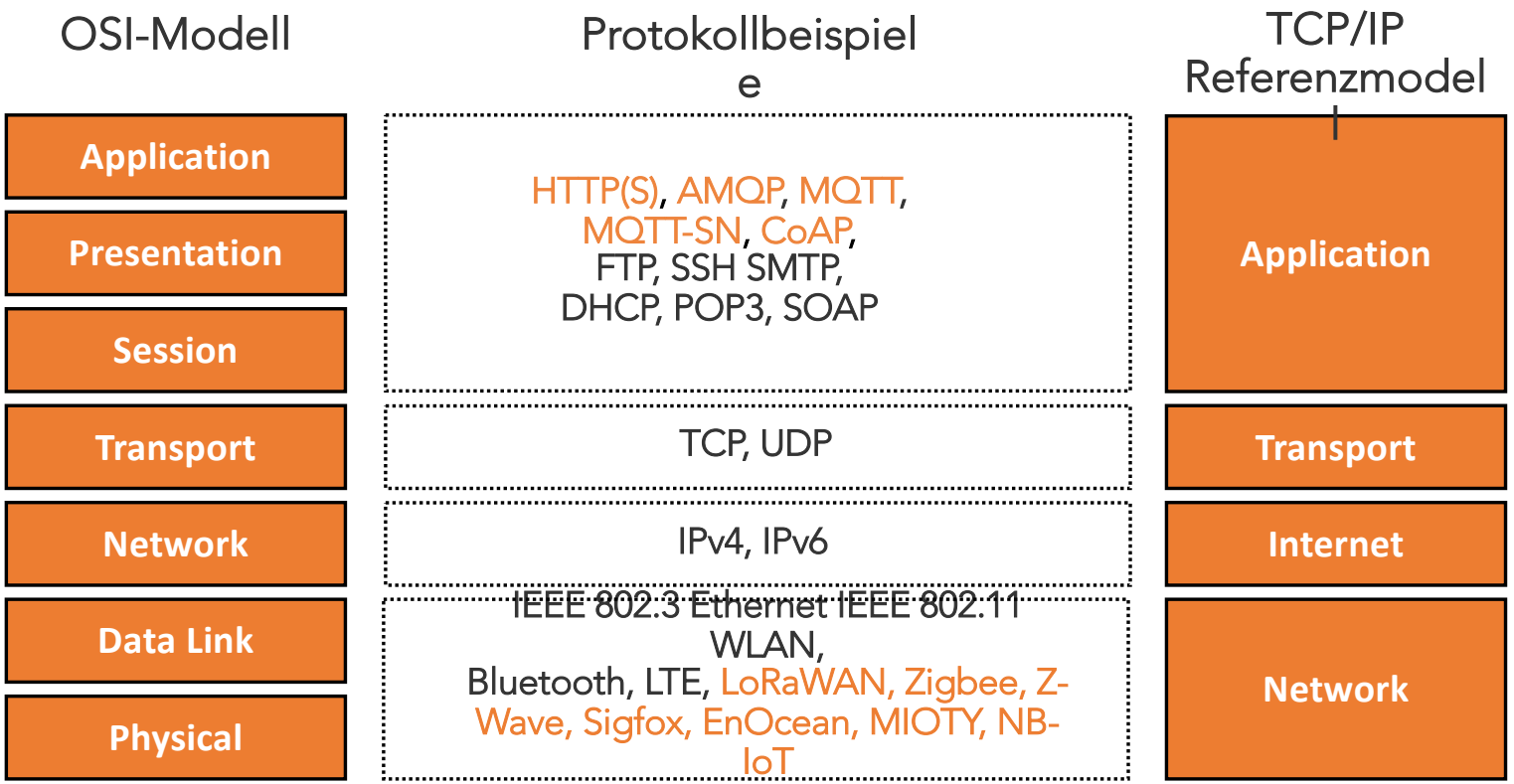
- Queues: Point-to-Point Kanäle für n:1 Kommunikation

- Topics: Publish-Subscribe Kanäle für n:m Kommunikation

Struktur

IoT-Protokolle

Referenzmodelle für Kommunikation



IoT-Protokolle

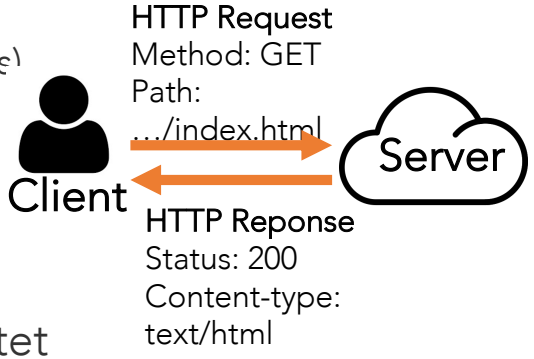
HTTP/HTTPS

- HTTP (Hypertext Transfer Protocol) ist ein synchrones Protokoll zur Datenübertragung auf Ebene der Anwendungsschicht
- 1989 ursprünglich von Tim Berners-Lee et al. für das World Wide Web geschaffen und von der IETF und dem W3C standardisiert
  - Zurzeit wird HTTP/3 auf Basis von Googles „HTTP-over-QUIC“ standardisiert
- HTTP setzt in der Regel auf das Transportprotokoll TCP auf und ist zustandslos, d.h. Informationen aus früheren Sitzungen (Sessions) gehen verloren
- HTTPS erweitert HTTP um eine Transportverschlüsselung mittels SSL/TLS
- HTTP nutzt Nachrichten (*Request* & *Response*) zur Kommunikation zwischen Client und Server
  - *Header* enthält Informationen bzgl. der verwendeten Kodierungen oder den Inhaltstyp
  - Der *Body* enthält die Nutzdaten, welche nicht nur auf Hypertext beschränkt → HTTP unterstützt beliebige Dateien

IoT-Protokolle

HTTP – Methoden & Statuscode

- HTTP hat verschiedene Anfragemethoden zur Kommunikation zwischen Client und Server definiert
- GET: Leseoperation zum Abruf von Ressourcen
  - am häufigsten verwendete Methode im Web
  - Laut HTTP-Spezifikation sicher und idempotent → keine Seiteneffekte
- PUT: Änderung einer existierenden oder Erzeugung einer noch nicht existierenden Ressource
  - Gegenstück zur GET-Methode
  - Ebenfalls idempotent (einmaliges oder mehrmaliges Aufrufen führt zum gleichen Ergebnis)
- POST: Anlegen einer neuen Ressource
  - Im Gegensatz zu PUT wird die Ressource unter einer vom Server gewählten URI angelegt
- DELETE: Löschen von Ressourcen
- Weitere Methoden: PATCH, HEAD, OPTIONS, TRACE und CONNECT
- Jede HTTP-Anfrage wird vom Server mit einem HTTP-Statuscode beantwortet
  - 1xx – Informationen / 2xx – Erfolgreiche Operation / 3xx – Umleitung / 4xx – Client-Fehler / 5xx – Server-Fehler



IoT-Protokolle

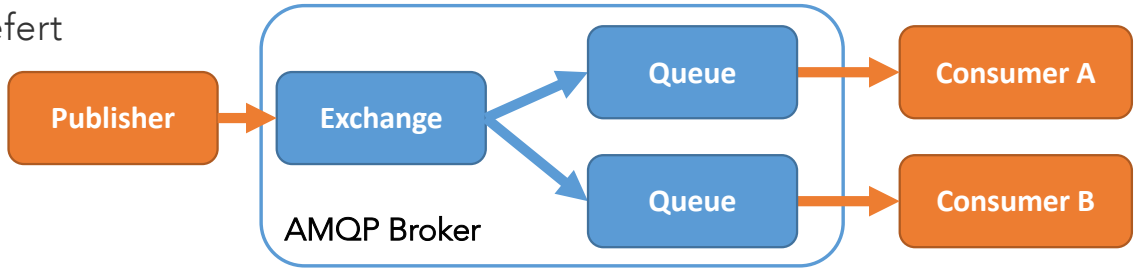
AMQP

- AMQP (**A**dvanced **M**essage **Q**ueuing **P**rotocol) stellt ein binäres Netzwerkprotokoll für nachrichtenorientierte Kommunikation auf Ebene der Anwendungsschicht dar
- 2003 von einem Konsortium aus Finanzinstitutionen und Softwarefirmen (Microsoft, Red Hat, JPMorgan Chase etc.) entwickelt und unter ISO/IEC 19464 standardisiert
  - Geschwindigkeit bei der Informationsübermittlung spielt in der Finanzbranche eine große Rolle
  - Aktuell ist AMQP in der Version 1.0 verfügbar
- JMS-Provider wie ActiveMQ oder Qpid basieren auf AMQP
- AMQP kommt oftmals zur Kommunikation zwischen Message-Brokern zum Einsatz
- Neben Publish-Subscribe können auch weitere Zustellungsformen genutzt werden

IoT-Protokolle

AMQP

- Nachrichten werden in sogenannten *Exchanges* veröffentlicht
- *Exchanges* verteilen mittels *Bindings* Nachrichtenkopien in *Queues*
  - *Direct Exchange*: Schickt Nachrichten mittels eines *Routing Key* an genau eine Queue mit passenden *Binding Key*
  - *Topic Exchange*: Ähnlich zu *Direct Exchange*, allerdings werden Platzhalter im *Routing Key* unterstützt, so dass mehrere Queues bedient werden können
  - *Fanout Exchange*: Nachricht wird an alle verfügbaren Queues verteilt
  - *Headers Exchange*: Properties im Header bestimmen die Verteilung der Nachrichten auf die Queues
- Die Nachrichten in der Queue werden anschließend
  - durch den AMQP-Broker an *Consumer* ausgeliefert
  - oder durch Consumer bei Bedarf abgerufen



IoT-Protokolle

MQTT

- Das MQTT (Message Queue Telemetry Transport) Protokoll wurde 1999 u.a. von IBM zur M2M-Kommunikation entwickelt, um das Vernetzen von IoT-Geräten mit geringen Ressourcen zu ermöglichen
- 2018 wurde Version 5.0 veröffentlicht, welche bei dem Standardisierungsgremium OASIS spezifiziert wurde
  - U.a. Verbesserungen hinsichtlich Skalierbarkeit, Fehlerbehandlung, Erweiterbarkeit sowie Authentifizierung und Autorisierung
  - Version 3.1.1 zur Zeit noch am verbreitetsten
- Realisiert das Publish-Subscribe Modell mit QoS Eigenschaften
  - unterschiedliche Servicequalitäten existieren, damit auch in instabilen Netzen die Übertragung gewährleistet werden kann
  - Metainformationen sollen serverseitig gespeichert werden, um die Notwendigkeit, diese nach Wiederaufnahme der Verbindung neu zu senden, zu unterbinden (session-awareness)
  - Datentypen unterschiedlichster Art sollen ohne Festlegung auf eine bestimmte Struktur mit dem Protokoll übertragen werden (datenagnostisch)
- Einsatzbeispiele für MQTT:
  - Heimautomatisierung (Smart Home)
  - Industrie 4.0
  - Vernetztes Fahren (Vehicle2X Kommunikation)
  - Sensornetzwerke (MQTT-SN)

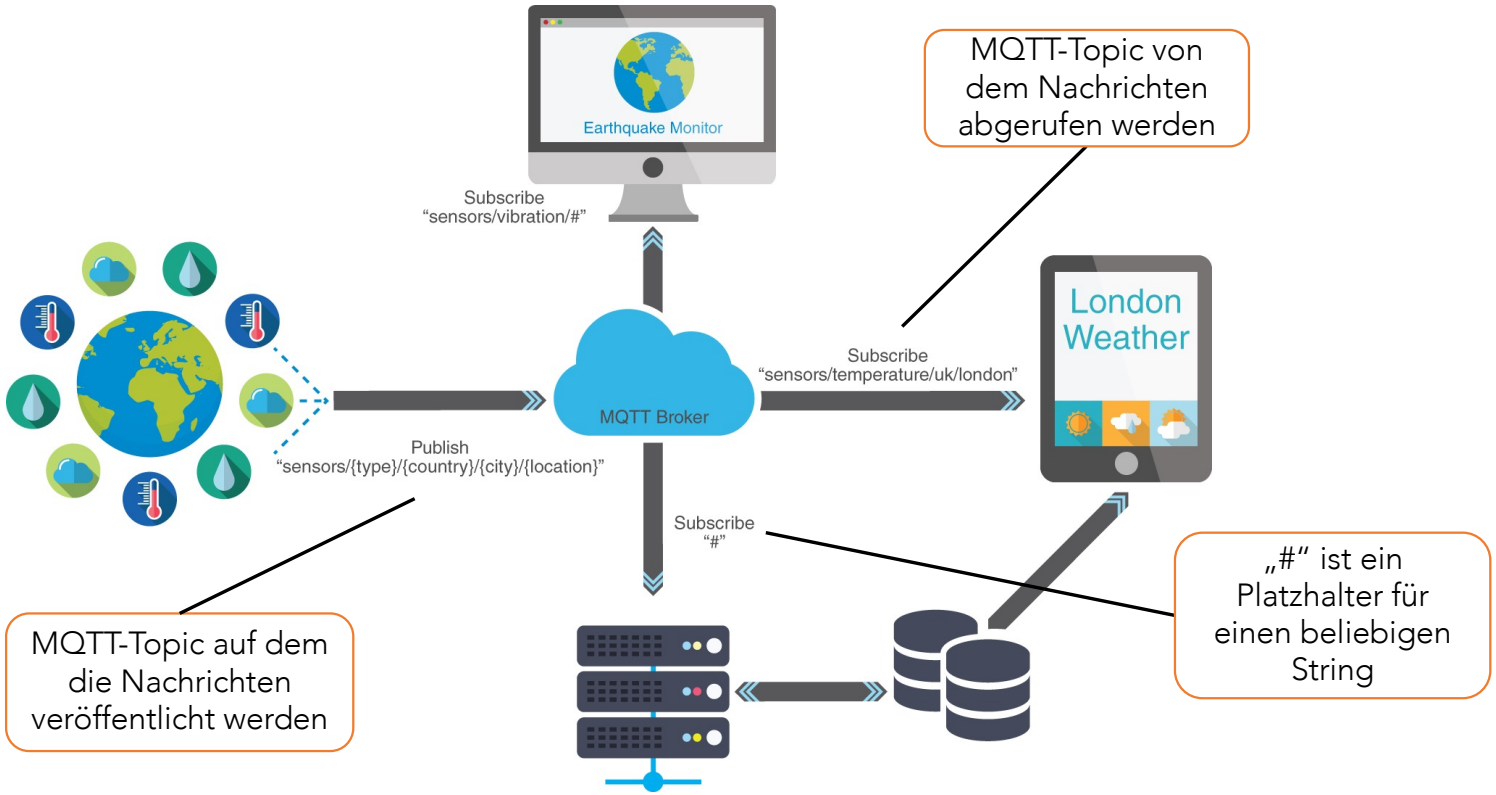
IoT-Protokolle

MQTT – Broker

- Ein MQTT Server („Broker“) entkoppelt Publisher und Subcriber
  - Ist für die Zustellung der Nachricht verantwortlich und verwaltet die Sessions
- Clients abonnieren Topics und empfangen Nachrichten über diese Topics
  - Clients verbinden sich mittels TCP mit dem Broker und bleiben verbunden
  - Bei MQTT-SN mittels UDP
  - Beispiel Client: Eclipse Paho (https://www.eclipse.org/paho)
- Der Broker hält die gesamte Datenlage seiner Kommunikationspartner
  - MQTT-Geräte kommunizieren mit dem MQTT Broker, um Daten bereitzustellen und Befehle entgegen zu nehmen
- Beispiele für MQTT Broker:
  - Eclipse Mosquitto (https://mosquitto.org)
  - HiveMQ (https://www.hivemq.com)
  - ActiveMQ (https://activemq.apache.org)

IoT-Protokolle

MQTT – Beispiel



IoT-Protokolle

MQTT – Beispiel 1/2

- Veröffentlichen von Nachrichten (Publish) mit Eclipse Paho

MQTT-Testbroker von Eclipse
Alternative:
https://test.mosquitto.org

```
public static void main(String[] args) throws MqttException {

      String messageString = "Hello World from Java!";

      MqttClient client = new MqttClient("tcp://mqtt.eclipse.org:1883",
                                         MqttClient.generateClientId());
      client.connect();

      MqttMessage message = new MqttMessage();
      message.setPayload(messageString.getBytes());
      client.publish("fhdo/fb4/swt2", message);

      client.disconnect();
}
```

Der Payload repräsentiert den Inhalt einer MQTT-Nachricht

MQTT-Topic auf dem die Nachrichten veröffentlicht werden

IoT-Protokolle

MQTT – Beispiel 2/2

- Abonnieren von Nachrichten (Subscribe) mit Eclipse Paho

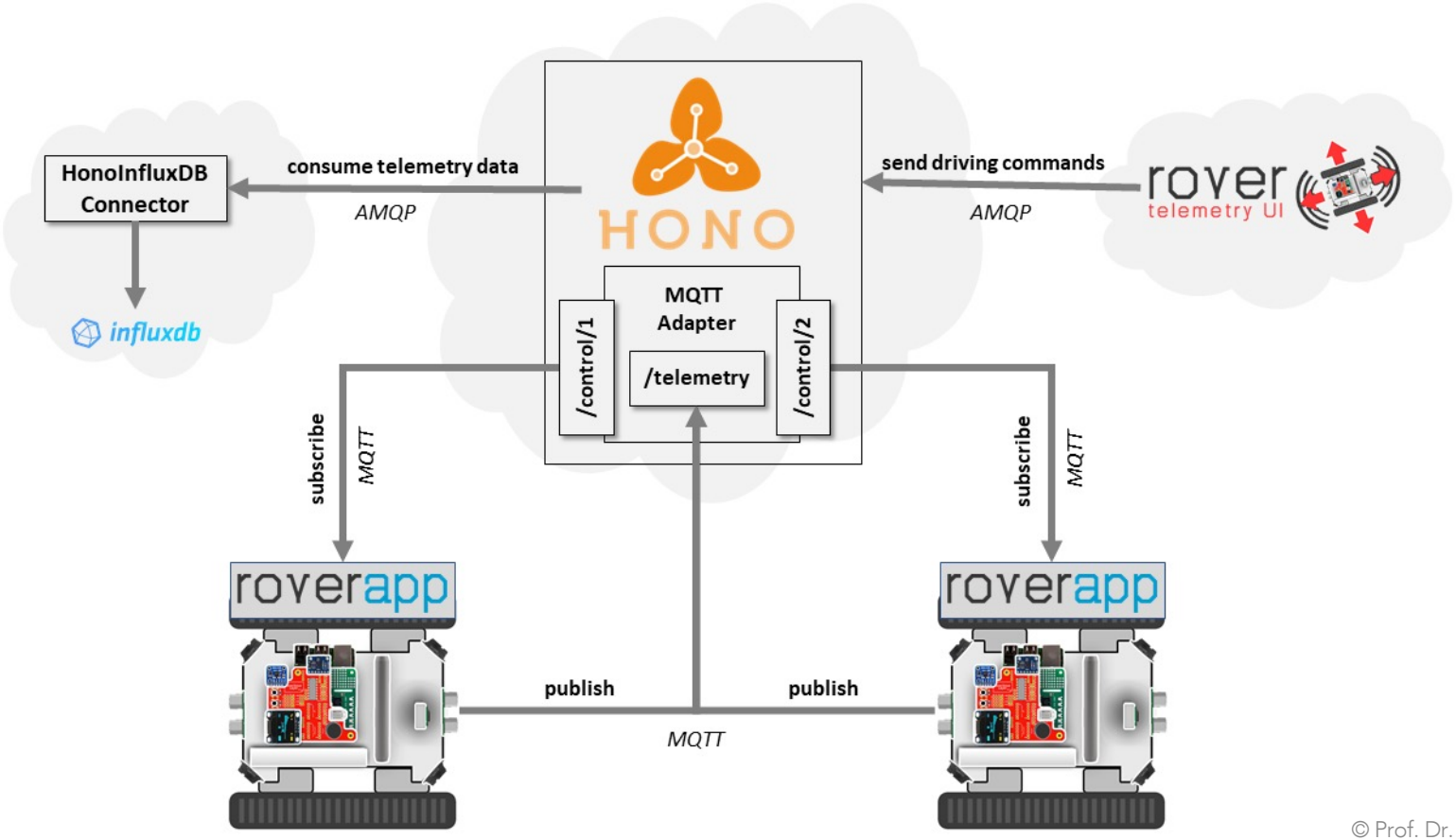
```
public static void main(String[] args) throws MqttException {

      MqttClient client = new MqttClient("tcp://mqtt.eclipse.org:1883",
                              MqttClient.generateClientId());
      client.setCallback(new SimpleMqttCallBack());
      client.connect();
      client.subscribe("fhdo/fb4/swt2");
}

public class SimpleMqttCallBack implements MqttCallback {

      public void messageArrived(String s, MqttMessage mqttMessage) throws Exception {
            System.out.println("Message received:\t"+ new
                                    String(mqttMessage.getPayload()));
      }
}
```

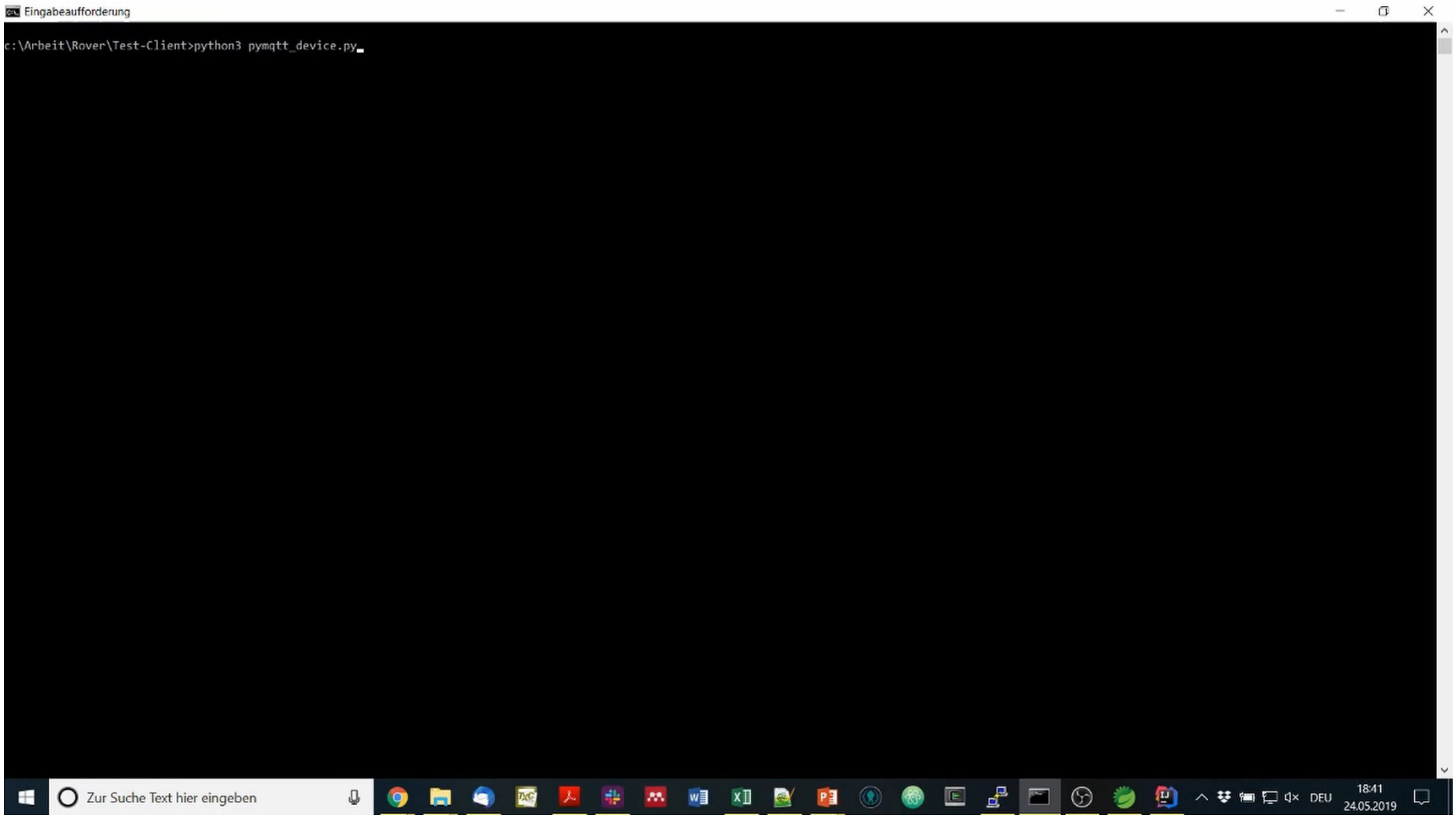
Für jede empfangene Nachricht wird die Callback-Methode ausgelöst

MQTT-Topic von dem Nachrichten abgerufen werden

IoT-Protokolle

MQTT – Beispiel Connected Vehicles




Eingabeaufforderung
c:\Arbeit\Rover\Test-Client>python3 pymqtt_device.py
Zur Suche Text hier eingeben
DEU
18:41
24.05.2019

IoT-Protokolle

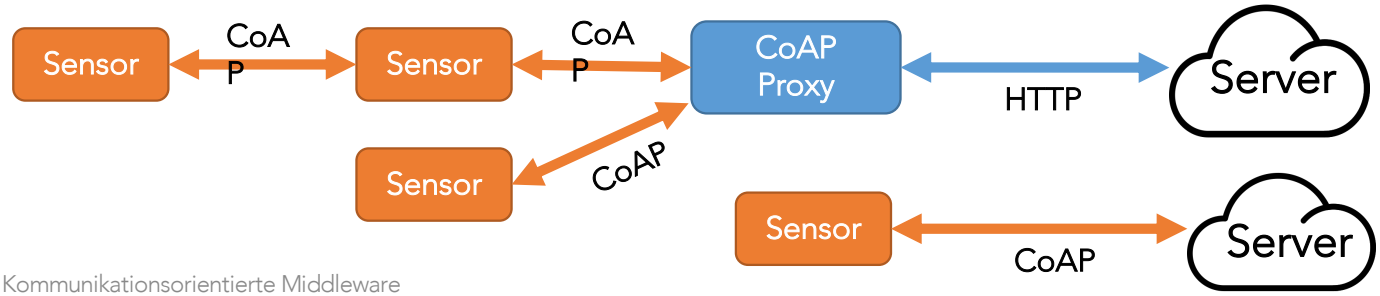
CoAP

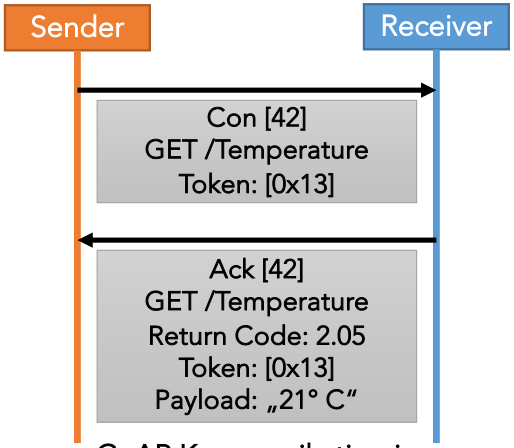
- CoAP (Constrained Application Protocol) ist ein Protokoll auf Anwendungsebene, welches speziell für stark ressourcenbeschränkte Geräte (Nodes) im M2M Kontext hin entwickelt wurde
  - Typische CoAP-Nodes sind eingebettete Systeme in Form von 8-bit Mikrocontroller mit einer geringen Menge an ROM und RAM
  - Insbesondere für verlustbehaftete und energiearme Sensornetzwerke geeignet (z.B. Low-Power Wireless Personal Area Networks (6LoWPANs) mit < 10 kbit/s Datentransferaten)
  - Anwendungsbeispiele: Automatisierungssysteme, Industrie 4.0, Smart Grids, Raumfahrt
- CoAP wird zur Zeit im Rahmen des IETFs standardisiert (RFC 7252)
- Bei CoAP handelt es sich im Prinzip um eine leichtgewichtige Alternative zu HTTP
  - Einfache Integration in bestehende Webanwendungen
  - Der Speicherbedarf von CoAP Paketen (mind. 4 Byte für den Header) ist deutlich kleiner als bei HTTP
  - Parsing von Nachrichten ist auch mit wenig Arbeitsspeicher möglich
- CoAP verwendet standardmäßig UDP als unterliegendes Protokoll
  - Paketbasierte Kommunikation (UDP) ist performanter als verbindungsorientierte Kommunikation (TCP)
  - UDP unterstützt weiterhin Multicasting
- DTLS (Datagram Transport Layer Security) ermöglicht eine abgesicherte Kommunikation
  - Aufgrund der Nutzung von UDP ist CoAP jedoch anfällig für DDoS-Attacken, z.B. via UDP-Flooding

IoT-Protokolle

CoAP

- CoAP Nachrichten setzten sich aus einem Header und einem Body zusammen
  - Header enthält u.a. Informationen zu Version, Nachrichtentyp und ID
- CoAP ist ein Client-Server Protokoll
  - Jeder Node kann sowohl als Client als auch Server fungieren
  - Über Proxys kann Interoperabilität mit HTTP gewährleistet werden
- Ähnlich wie HTTP definiert CoAP zwei Nachrichtypen: *Request* & *Response*
  - HTTP-Requests GET, PUT, POST und DELETE werden unterstützt
  - Mittels OBSERVE können Zustandsänderungen von Ressourcen kontinuierlich und ereignisbasiert beobachtet werden

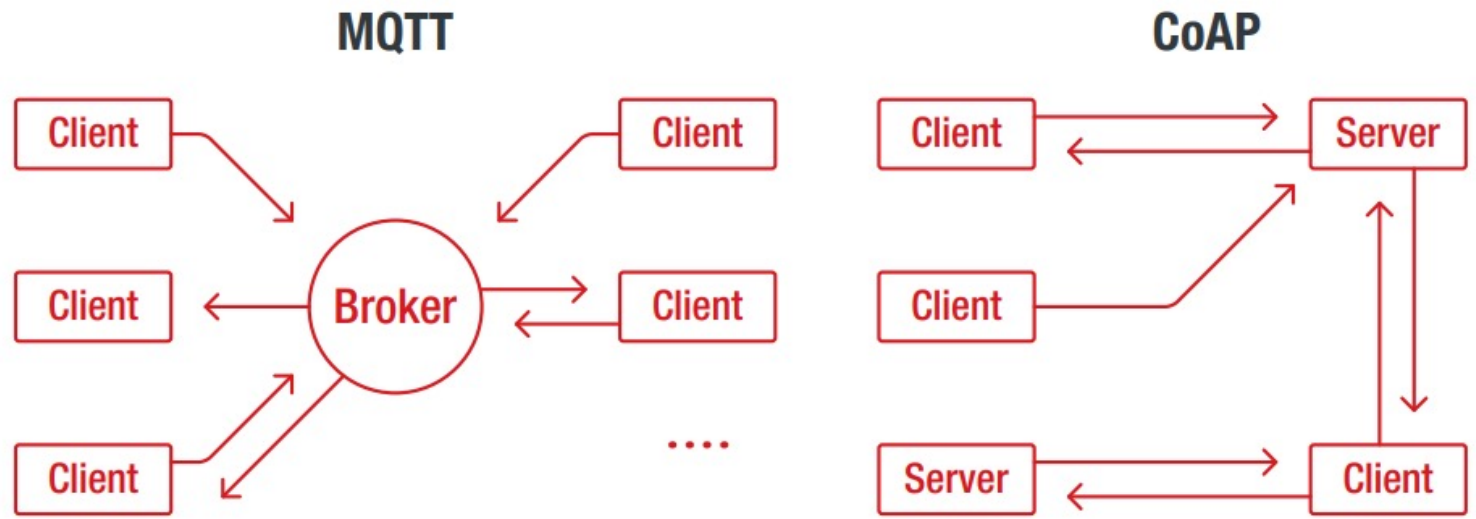




CoAP Kommunikation in der Piggybacking Variante

IoT-Protokolle

MQTT vs. CoAP



Publish-Subscribe

Client-Server

Maggi, Federico, Rainer Vosseler, and Davide Quarta. "The fragility of industrial IoT's data backbone." *Trend Micro Inc., https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/mqtt-and-coap-security-and-privacy-issues-in-iot-and-iiot-communication-protocols* (2018).

Fachhochschule Dortmund
University of Applied Sciences and Arts

IoT-Protokolle

Netzwerkprotokolle aus dem IoT-Kontext

we focus on students

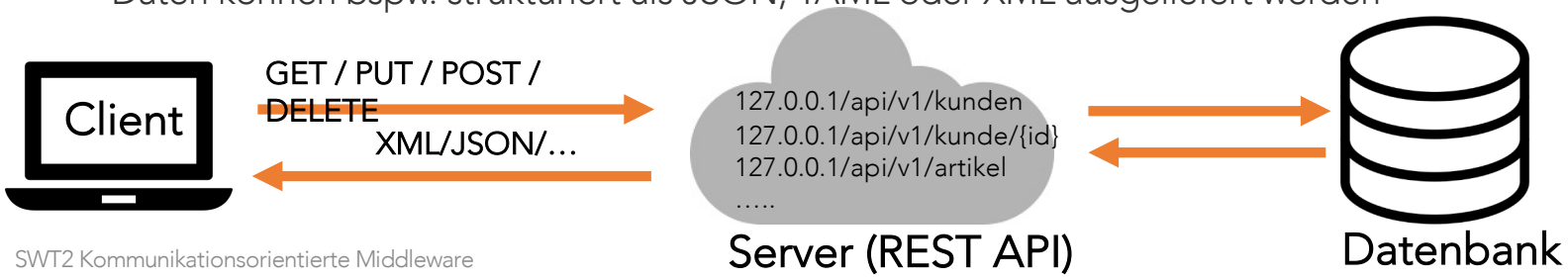
- Low Power Wide Area Networks (LPWAN) repräsentieren Netzwerke, die durch eine große Reichweite und einem niedrigen Energieverbrauch der Endgeräte gekennzeichnet sind
  - Vor allem für die Bereiche Logistik, Smart City oder Smart Agriculture bieten sich LPWANs an
- LoRaWAN (**Lo**ng **Ra**nged **W**ide **A**rea **N**etwork) ist ein drahtloses Netzwerkprotokoll für LPWANs
  - LoRaWAN ist energieeffizient und ermöglicht die Datenübertragung über eine große Entfernung
  - Reichweite erstreckt sich von 2-5 km in Stadtgebieten bis zu 40 km in ländlichen Gebieten mit Datenübertragungsraten von 292 Bit/s bis 50 kbit/s
- Sigfox ist ebenso wie LoRaWAN ein verbreitetes Protokoll für LPWAN
  - Sende- und Empfangsinfrastruktur in ca. 60 Ländern vorhanden, in Deutschland eine Abdeckung von etwa 85%
  - Operiert ebenso wie LoRaWAN im lizenzfreien 868-MHz-Band, allerdings werden Daten zentral über Sigfox abgerufen → keine Datenhoheit
- Weitere offene Protokolle für LPWANs: MIOTY, NB-IoT oder LTE-M
- EnOcean ist ein besonders energieeffizientes Funkprotokoll, welches vor allem für die Gebäudeautomation mittels batterieloser Funksensorik (*Energy Harvesting*) eingesetzt wird
- Thread, Z-Wave und ZigBee sind weitere verbreitete Protokolle aus dem Bereich Gebäudeautomation / Smart Home

Struktur

Kommunikation in verteilten Systemen

REST

- Representational State Transfer (REST) ist ein Architekturstil für verteilte Systeme
- Von Roy Fielding auf Basis von HTTP 1.1 entwickelt und im Jahr 2000 veröffentlicht [1]
- Aufgrund der großen Flexibilität stark in der Industrie für die Realisierung von Webservices verbreitet
  - REST in Kombination mit JSON [2] ist der heute am weitesten verbreitete Ansatz für Programmierschnittstellen
  - CoAP: REST des kleinen Geräts
- World Wide Web stellt bereits einen Großteil der benötigten Infrastruktur bereit
  - REST basiert auf einer Client-Server-Architektur
  - Nutzt die HTTP-Methoden GET, POST, PUT und DELETE für CRUD-Operationen
  - Daten können bspw. strukturiert als JSON, YAML oder XML ausgeliefert werden



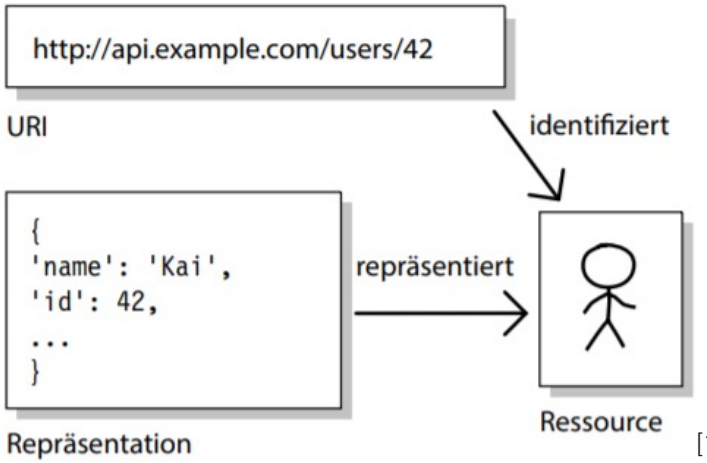
[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[2] JSON-API-Spezifikation: http://jsonapi.org

Kommunikation in verteilten Systemen

REST – Architekturprinzipien

REST definiert verschiedene Architekturprinzipien [1]

- Eindeutige Identifikation von Ressourcen
  - Jede Ressource einer verteilten Anwendung besitzt eine einheitliche Identifikation (URI)
- Verwendung von Hypermedia
  - Der Client einer REST-Schnittstelle navigiert ausschließlich über URIs, welche vom Server bereitgestellt werden
- Verwendung von HTTP-Standardmethoden
  - GET, POST, PUT, DELETE
- Unterschiedliche Repräsentationen von Ressourcen
  - Ressourcen können in unterschiedlichen Repräsentationen (JSON, XML, etc.) ausgeliefert werden, während die Veränderung einer Ressource nur über eine Repräsentation erfolgen soll
- Zustandslosigkeit
  - Kommunikation zwischen Client und Server findet zustandslos statt, d.h. eine REST-Nachricht enthält alle benötigten Informationen (z.B. API-Schlüssel, Benutzer-ID, etc.)



[1]

[1] Tilkov, Stefan, et al. *REST und HTTP: Entwicklung und Integration nach dem Architekturstildes Web*. dpunkt. verlag, 2015.

Kommunikation in verteilten Systemen

REST – JAX-RS & Spring MVC

- Bei JAX-RS (Jakarta RESTful Web Services) handelt es sich um eine Java Schnittstellenspezifikation für die Erstellung von RESTful Webservices
  - Aktuell ist die JAX-RS Spezifikation in der Version 2.1 verfügbar (JSR-370)
  - JAX-RS ist einer der zentralen Komponenten von Java EE / Jakarta
- JAX-RS stellt verschiedene Annotationen breit,
  - *@Path* gibt den Pfad zu einer Ressource an (URI), während *@Get*, *@Post* etc. die HTTP-Methoden repräsentieren
  - Über *@Produces* und *@Consumes* können die zur Kommunikation verwendeten/benötigten Datenformate spezifiziert werden
  - *@*Param* Annotationen erlauben das Abgreifen von Parametern aus Requests
- Die Referenzimplementierung für JAX-RS lautet Jersey
- Spring MVC beinhaltet ebenfalls eine vollständige Implementierung eines REST-Stacks, allerdings nicht auf Basis von JAX-RS
  - Neben der Bereitstellung von REST-Schnittstellen integriert Spring MVC auch entsprechende Persistierungsmechanismen (z.B. Hibernate)

Kommunikation in verteilten Systemen

REST API mit Spring MVC – Beispiel 1/2

Annotation legt fest, welche HTTP-Methode verwendet werden soll

```
@GetMapping("/api/journey/{id}")
public Journey getJourneyById(@PathVariable Long id) {
    return journeyRepository.findById(id).orElseThrow(JourneyNotFoundException::new);
}

@GetMapping("/api/journeys")
public List<Journey> getAllJourneys() {
    List<Journey> journeys = new ArrayList<Journey>();
    journeyRepository.findAll().forEach(journeys::add);
    return journeys;
}

@PostMapping("/api/journey")
@ResponseStatus(HttpStatus.CREATED)
public Journey createJourney(@RequestBody Journey journey) {
    return journeyRepository.save(journey);
}
```

Eindeutige URI zur Identifikation von Ressourcen

Kommunikation in verteilten Systemen

REST API mit Spring MVC – Beispiel 2/2

Wildcards für den Abruf von spezifischen Ressourcen

```
@PutMapping("/api/journey/{id}")
public Journey updateJourney(@RequestBody Journey journey, @PathVariable Long id) {
    Journey updateJourney = journeyRepository.findById(id)
            .orElseThrow(JourneyNotFoundException::new);
    updateJourney.setOrigin(journey.getOrigin());
    updateJourney.setDestination(journey.getDestination());
    return journeyRepository.save(updateJourney);
}

@DeleteMapping("/api/journey/{id}")
public void deleteJourney(@PathVariable Long id) {
    journeyRepository.deleteById(id);
}
```

Spezifischer Code für die REST-Schnittstelle zur Verarbeitung der Anfrage

Kommunikation in verteilten Systemen

REST API GET Request + Response – Beispiel

Mittels HTTP-basierter Anfragen können Ressourcen abgerufen, erstellt oder manipuliert werden..

```
GET http://localhost:8080/api/journey/1
HTTP/1.1 Accept: application/vnd.api+json



{
   "journey":{
      "id":"1",
      "name":"Ägypten",
      "activity":[
         {
            "category":"Tauchen",
            "operator":"Deep Blue Dive"
         }
      ]
   }
}
```

Im HTTP-Header legt der Client das Format der Antwort fest

Antwort vom Server im JSON Format

Kommunikation in verteilten Systemen

REST API POST Request – Beispiel

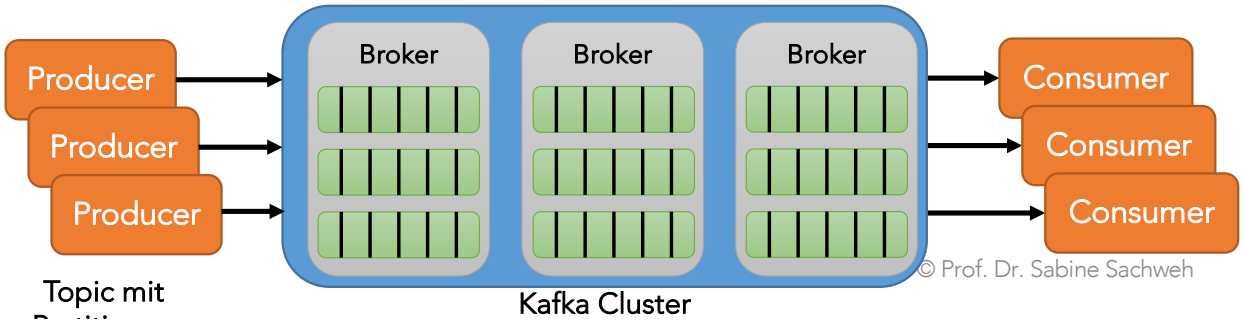

```
POST http://localhost:8080/api/journey/ HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
   "journey":{
      "name":"Spanien",
      "activity":[
         {
            "category":"Kultur",
            "sight":"Sagrada Família"
         },
         {
            "category":"Kultur",
            "sight":"Montserrat"
         }
      ]
   }
}
```

ID wird vom Server vergeben und muss daher nicht angegeben werden

Kommunikation in verteilten Systemen

Streaming-Plattform – Apache Kafka

- Big Data und das IoT stellen an verteilte Systeme zahlreiche neue Herausforderungen hinsichtlich Skalierbarkeit und Verfügbarkeit
  - Systemschnittstellen müssen so konzipiert sein, dass auch eine große Anzahl an Clients mit großen Datenmengen flexibel und zeitnah bedient werden können
  - Klassische Middleware-Lösungen stoßen hier an ihre Grenzen
- Streaming-Plattformen vereinen unterschiedliche Middleware Komponenten (Messaging, Datenbank, Datenverarbeitung) in einer Infrastruktur und ermöglichen so die effiziente Verarbeitung von großen Datenströmen bei gleichzeitig hoher Verfügbarkeit
- Apache Kafka ist eine Open Source Streaming-Plattform, welche auf Basis einer verteilten Architektur ein echtzeitfähiges Messaging-System mit einer integrierten Datenhaltung und –verarbeitung kombiniert
  - Kafka agiert als Messaging-System zwischen Sender und Empfänger, ist jedoch im Vergleich zu reinen Nachrichten-Queues fehlertolerant und extrem skalierbar
  - Ein Kafka Cluster setzt sich aus (verteilten) *Brokern* zusammen, die Key-Value Nachrichten mit einem Zeitstempel in *Topics* speichern
  - Ein *Topic* ist wiederum in *Partitionen* aufgeteilt, die innerhalb des Kafka Clusters verteilt und repliziert werden können → hohe Verfügbarkeit und Skalierbarkeit
- Kafka eignet sich insbesondere für den Bereich des maschinellen Lernens
  - Z.B. durch das Trainieren von Modellen in Echtzeit
- Weitere Anwendungsgebiete:
  - Web-Analytics, Stream Processing
  - Microservices



Kommunikation in verteilten Systemen

Weitere Aspekte zur Schnittstellendefinition

- Open-Source Frameworks wie *Swagger* unterstützen Entwickler bei der Schnitttstellendefinition und -dokumentation
  - Durch Codegenerierung und einer automatisierten Dokumentation wird ein Großteil des Entwicklungs- und Dokumentationsaufwandes abgenommen
- Eine übersichtliche und detaillierte Dokumentation der angebotenen Schnittstellen ist enorm wichtig für die Nutzung und Wartung der Anwendung
  - *OpenAPI* ist eine Spezifikation zur Beschreibung von REST-konformen Programmierschnittstellen (API) auf Basis eines offenen und herstellerneutralen Beschreibungsformats
- Auch das ausgiebige Testen der API mit unterschiedlichen Daten und Formaten ist
  - Der leichtgewichtige REST-Client *Postman* ermöglicht auf effiziente Weise das Erstellen von HTTP-Requests und somit das Testen der REST-API

Weitere Fragen