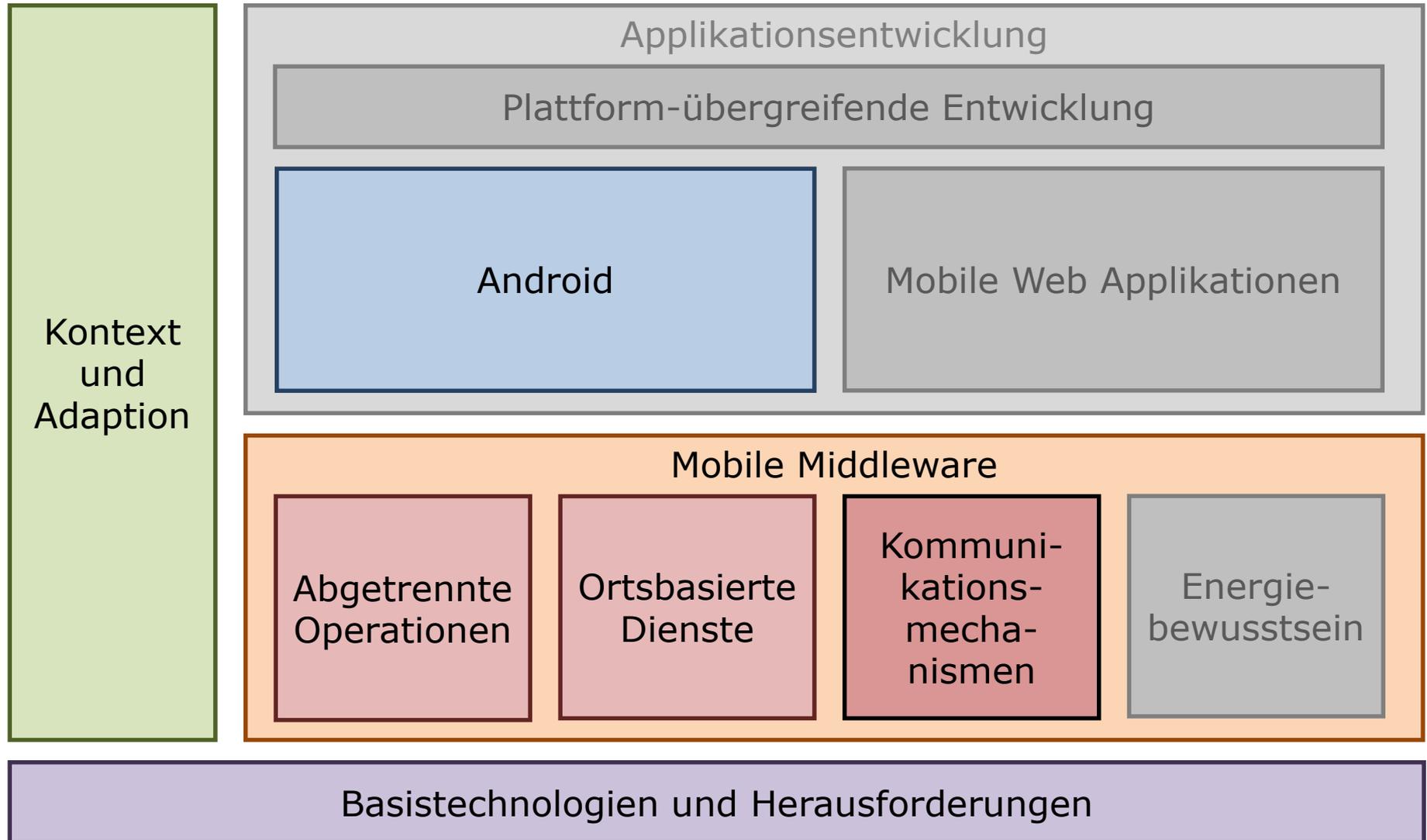


# Web- und App-Programmierung Kommunikations- mechanismen

mit Skriptmaterial von Dr.-Ing. T. Springer

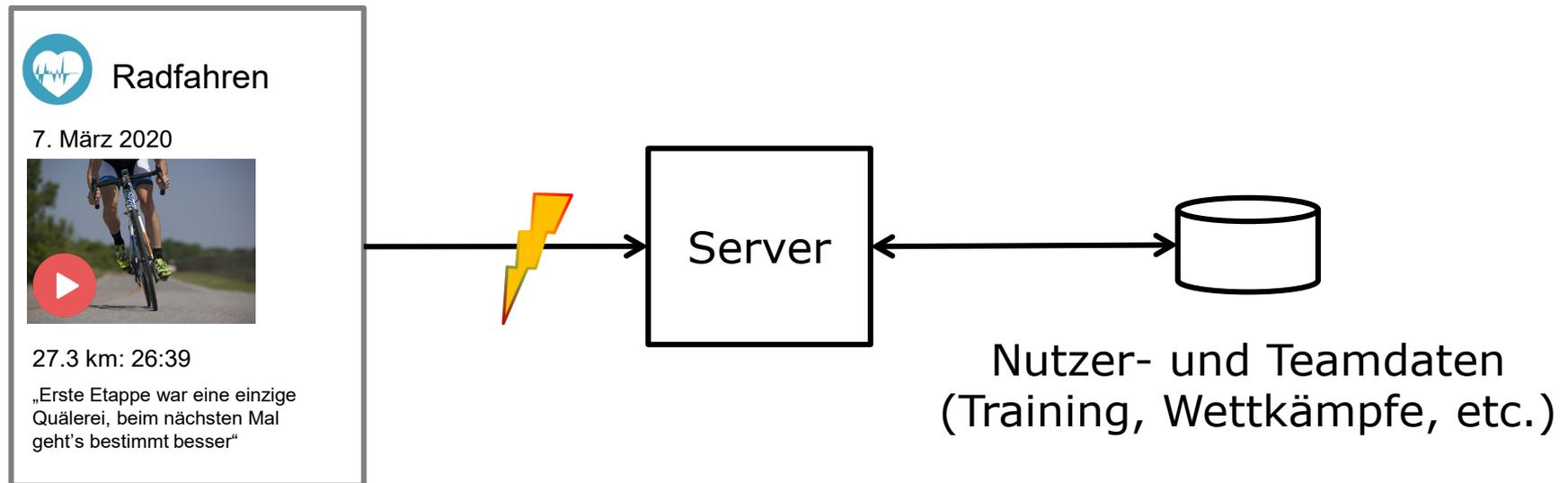
**Prof. Dr.-Ing. Tenshi Hara**  
tenshi.hara@ba-sachsen.de

# AUFBAU DER LEHRVERANSTALTUNG



## SOCIAL-FITNESS-APP – KONNEKTIVITÄTSHERAUSFORDERUNG

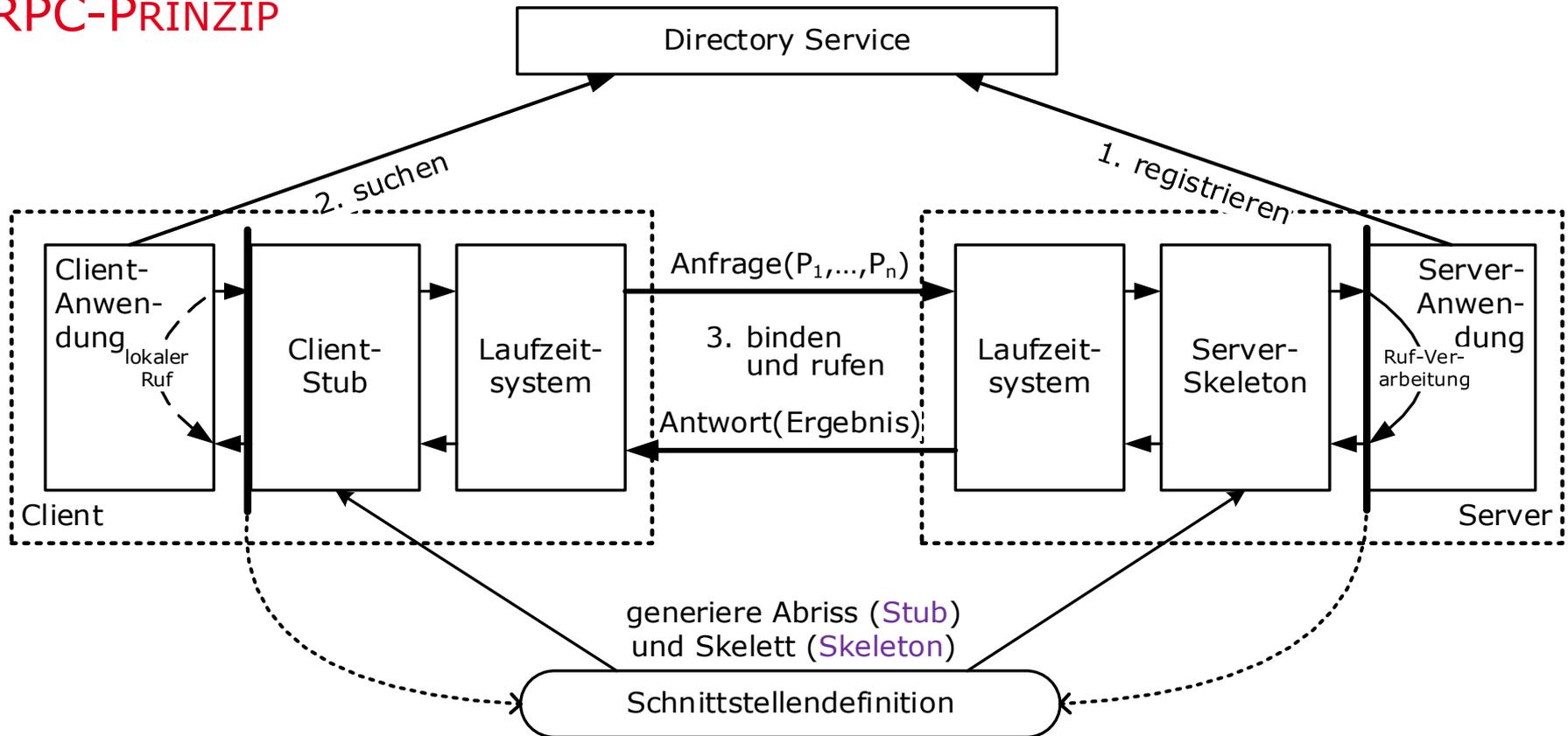
- Hochladen lokaler Trainingsdaten und Medien
  - Anfrage/Antwort-Mechanismus (Request/Response, bspw. HTTP)



- Probleme
  - heterogene Zugangnetzwerke
  - variierende Qualität und Stabilität

# Request/Response- Architektur

# RPC-PRINZIP



- erweitern des lokalen Prozeduraufrufs auf einen entfernten Zugriff
- Ziel: syntaktische und semantische Uniformität
  - Rufmechanismus (transparente Netzwerkkommunikation)
  - Sprachelemente und Fehlersemantik

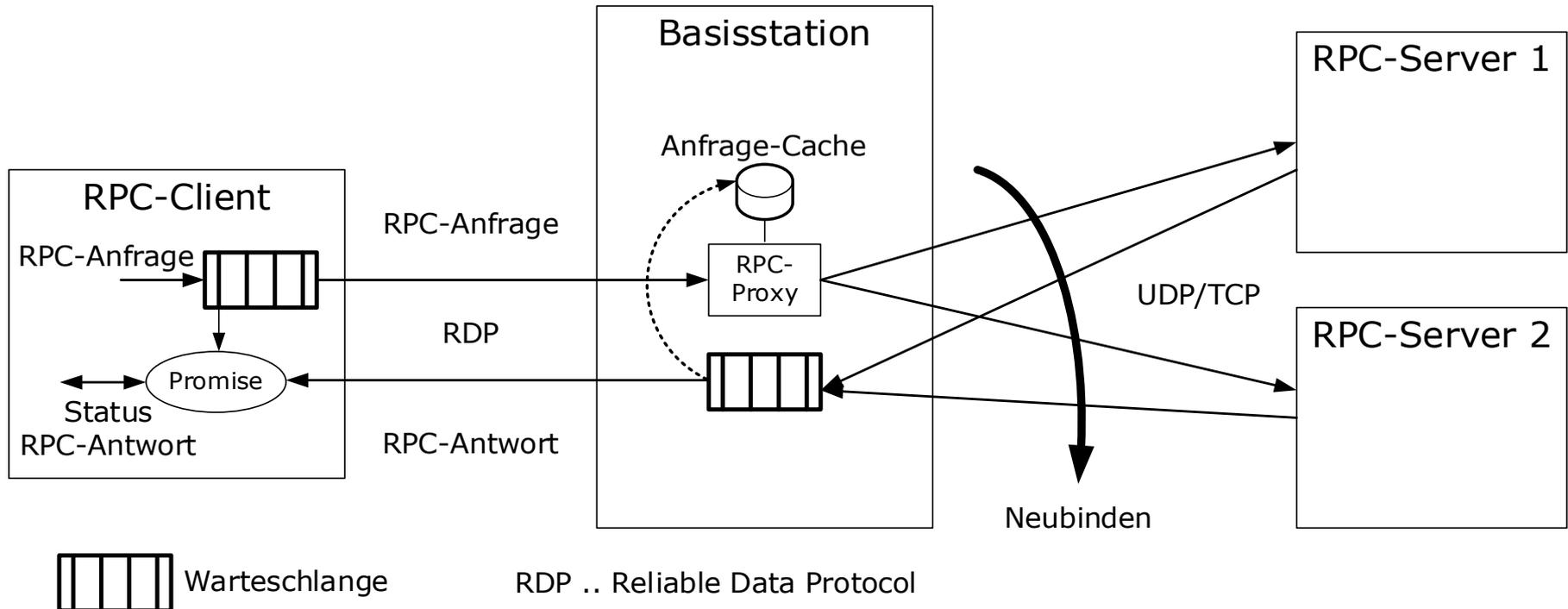
## RPC IN MOBILEN UMGEBUNGEN

- synchroner Kommunikationsmodus des RPC erfordert stehende und stabile Netzwerkverbindung; demgegenüber
  - häufige Trennungen in Drahtlosnetzwerken
  - hohe Verzögerungen durch Nachrichtenwiederholungen
  - hoher Energieverbrauch
    - Nutzer trennt Verbindung, wodurch Anwendung blockiert
- Rufe werden entsprechend logischem Programmablauf abgesetzt, aber
  - in Abtrennungsphasen können keine RPC abgesetzt werden
  - keine Bündelung multipler Rufe in eine Anfrage zur Ausnutzung kurzzeitiger Verbindungen (oder höherer Bandbreiten)
- Client bindet sich zu Beginn einer Konversation an den Server; demgegenüber wäre wegen Mobilität und Abtrennungen eine Neubindung an anderen Server notwendig, wird aber nicht unterstützt

## MOBILES RPC-KONZEPT (M-RPC)

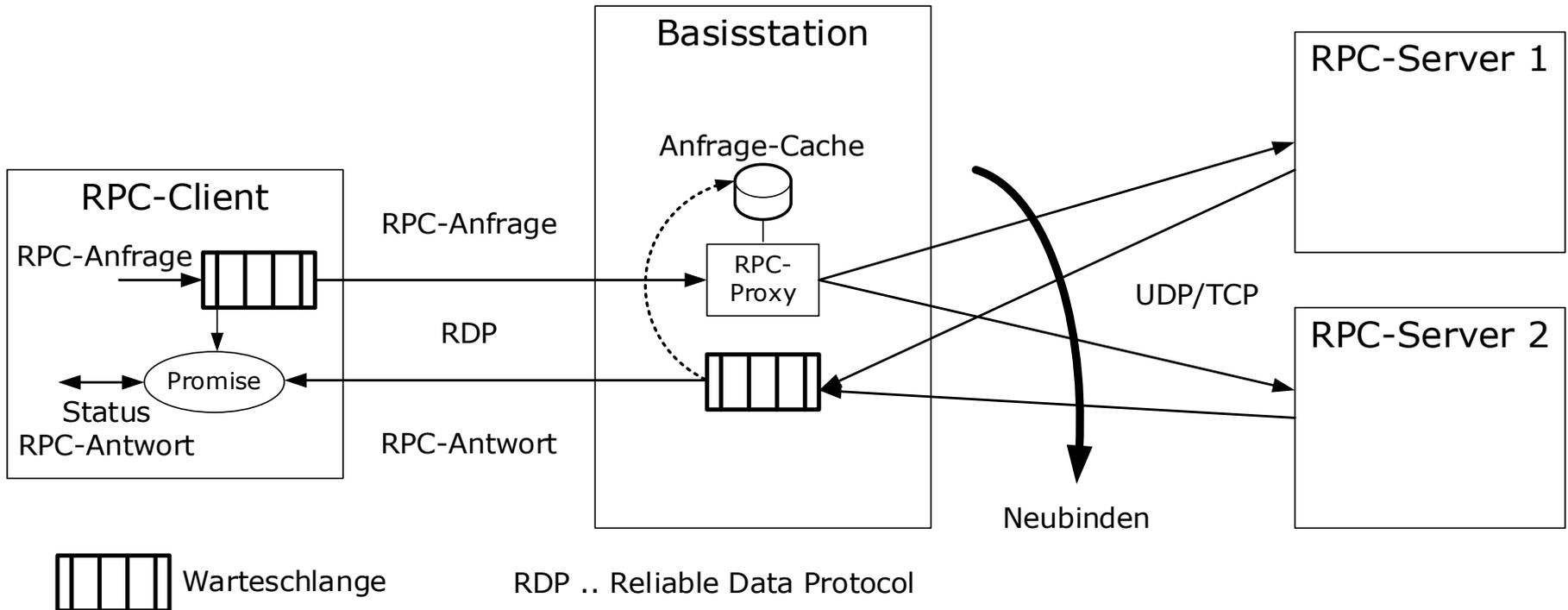
M-RPC ermöglicht

- zuverlässige Rufvermittlung über unzuverlässige Verbindung
- optimierte RPC-Kommunikation
- dynamisches (Neu)Binden



# M-RPC

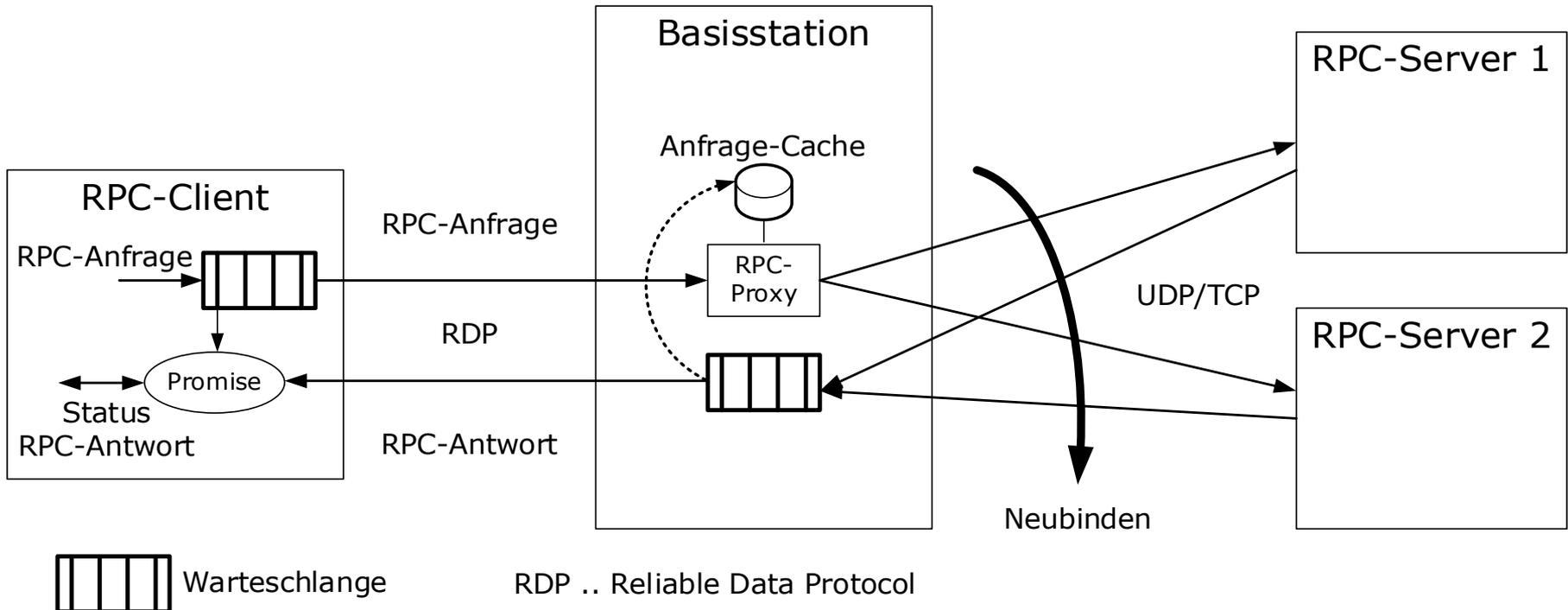
- Proxy auf Basisstation innerhalb der Netzwerkinfrastruktur
  - angepasstes Transportprotokoll zwischen mobilem Gerät und Basis
  - Anfragen in Cache gehalten bis Client Antwortempfang bestätigt
  - Neuübertragung von Anfragen durch Proxy
- Warteschlangen für Rufe und Ergebnisse auf Client und Proxy
- kumulierte Rufe → Bulking zur Durchsatzoptimierung



# M-RPC

dynamisches Neubinden wegen Indirektion über Proxy

- Client logisch an Server, aber physisch an Proxy gebunden
- neue physische Bindung nach Abtrennung oder Ortswechsel
- Server-Zustand muss berücksichtigt werden



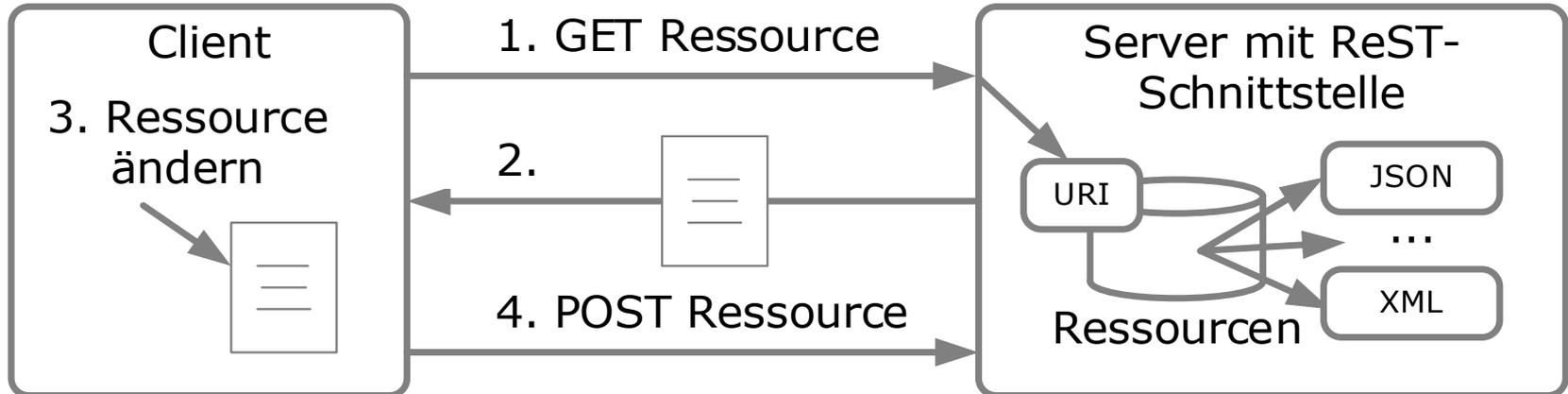
# REPRESENTATIONAL STATE TRANSFER (REST)

- Modell für verteilte Hypermediasysteme
  - erstmals von Roy Fielding (2000) spezifiziert
- im Web weit verbreitete Architektur
- viele Systeme bieten ReST-Schnittstellen
- im Vergleich zu Web Services sehr leichtgewichtig
  - Menge vordefinierte Operationen → CRUD
  - multiple Kodierungsformate (mit JSON auch kompakt)

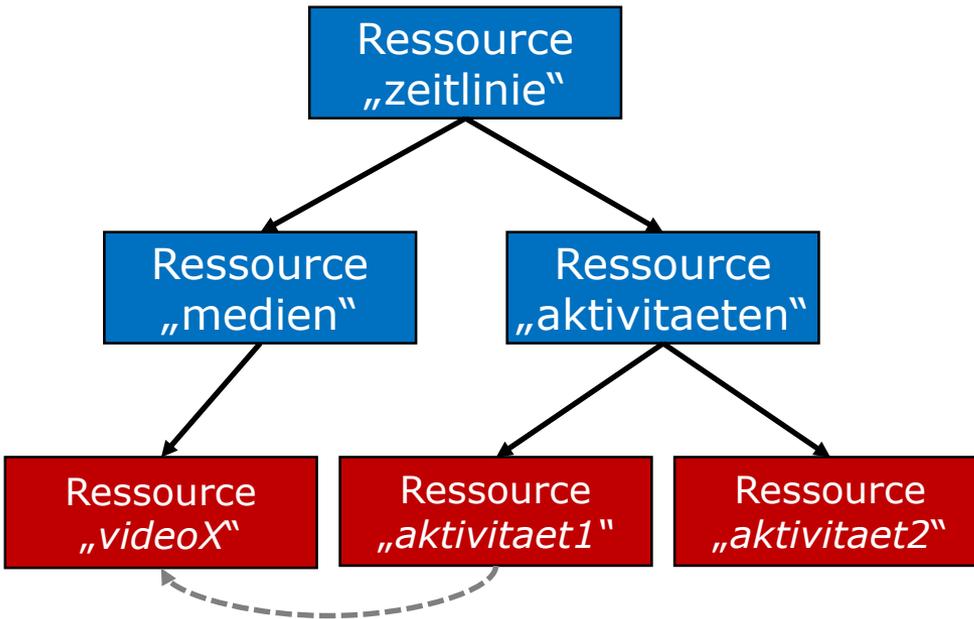


# REPRESENTATIONAL STATE TRANSFER (REST)

- basiert auf Client/Server-Architektur
- nutzt **zustandloses** Kommunikationsprotokoll  
→ mangels serverseitigem Kontext und nur clientseitiger Sitzung muss jede Anfrage alle notwendigen Informationen enthalten
- ReST-Anfragen an **Ressourcen** gebunden, nicht an Prozeduren wie bei RPC
  - Ressource: Web-Seite, Datensammlung, Bild, ...
  - jede Ressource muss über einen eindeutige **URI** erreichbar sein
  - HTTP-Methoden als Operationsmenge (**C**reate, **R**ead, **U**ppdate, **D**eleate)
  - kann *multiple* Repräsentationen haben (XML, JSON, ...)



# REST – URIS UND METHODEN



<https://example.com/zeitlinie>

<https://example.com/zeitlinie/aktivitaeten>

<https://example.com/zeitlinie/aktivitaeten/aktivitaet1>

Ressourcentyp	GET	POST	PUT	DELETE
<b>Sammlung</b> <a href="https://example.com/zeitlinie/aktivitaeten">https://example.com/zeitlinie/aktivitaeten</a>	<b>Liste</b> URIs und Details der Ressourcen auf	<b>Erzeuge</b> oder <b>ersetze</b> Sammlung in übergeordneter Ressource	<b>Erzeuge</b> neues Element	<b>Lösche</b> adressierte Sammlung
<b>Element</b> <a href="https://example.com/zeitlinie/aktivitaeten/aktivitaet1">https://example.com/zeitlinie/aktivitaeten/aktivitaet1</a>	<b>Liste</b> Details der adressierten Ressource in angebrachtem Format auf	<b>Erzeuge</b> oder <b>ersetze</b> adressierte Ressource	<b>Erzeuge</b> neues Element oder <b>ersetze</b> existierendes	<b>Lösche</b> adressierte Ressource

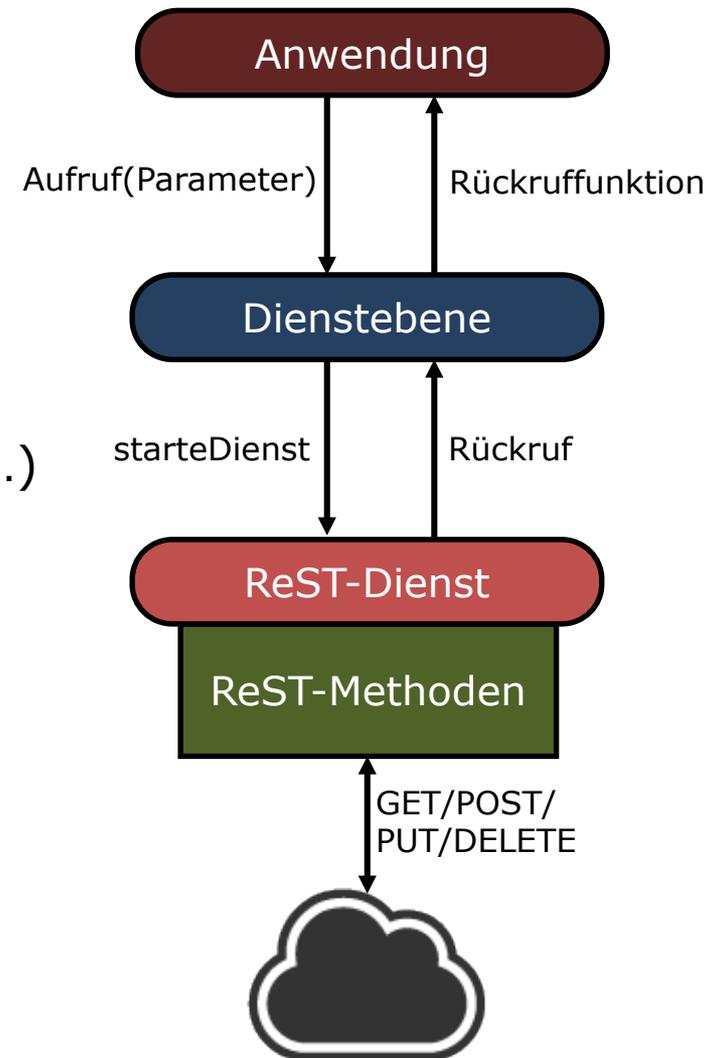
# JAVASCRIPT OBJECT NOTATION (JSON)

- Standardformat (spezifiziert als RFC 4627)
- unabhängig von der Programmiersprache
- menschenlesbare, textbasierte Datenkodierung
- JSON-Syntaxregeln
  - Daten liegen in Name:Wert-Paaren vor
  - Daten werden durch Kommata separiert
  - geschweifte Klammern halten Objekte
  - eckige Klammern halten Felder

```
"Aktivitaet":{
  "nutzer":"nutzerURI",
  "typ":"radfahrend",
  "entfernung":"120",
  "zeit":"05:21:12",
  "medium":"videoXURI"
}
"AktivitaetsListe":[
  {"nutzer1":"uri1","typ":"radfahrend"},
  {"nutzer2":"uri2","typ":"gehend"},
  {"nutzer3":"uri3","typ":"rennend"}
]
```

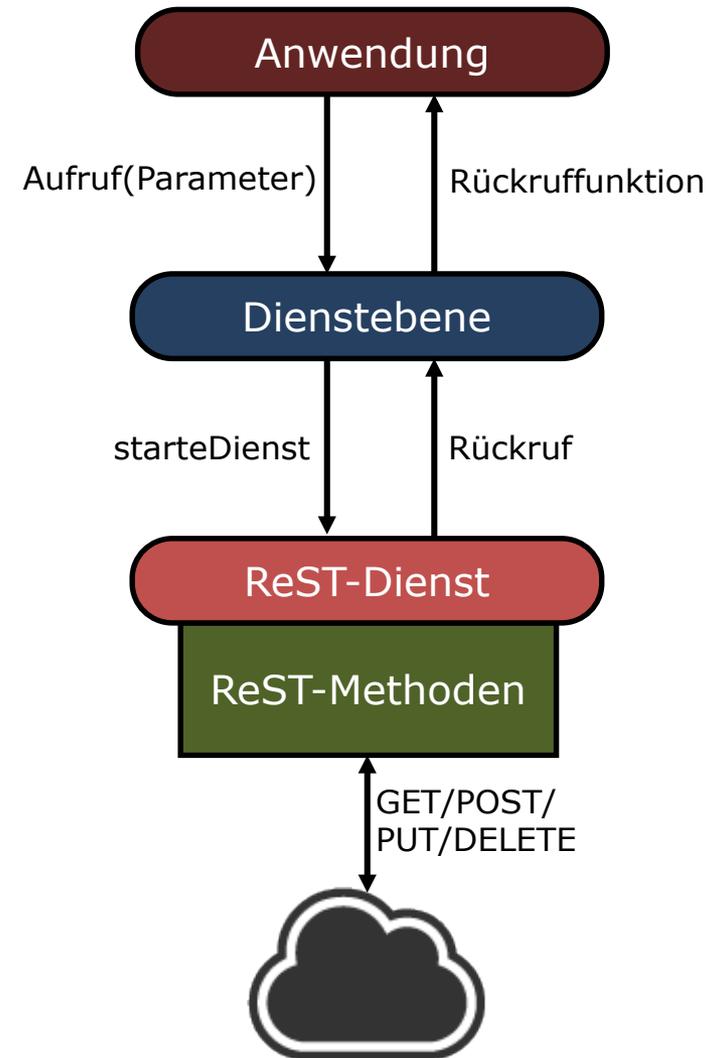
# REST – IMPLEMENTIERUNGSPRINZIP

- **Activity** repräsentiert Bildschirm in Android
- Dienstebene
  - applikationsspezifische API (holeZeitlinie(), holeAktivitaet(), ...)
  - Kartenanwendungsspezifische Dienstaufrufe auf ReST-Aufrufe
  - verwaltet Anfragen bspw. in einer Anfrageschlage
  - bietet Rückruffunktion zur Activity

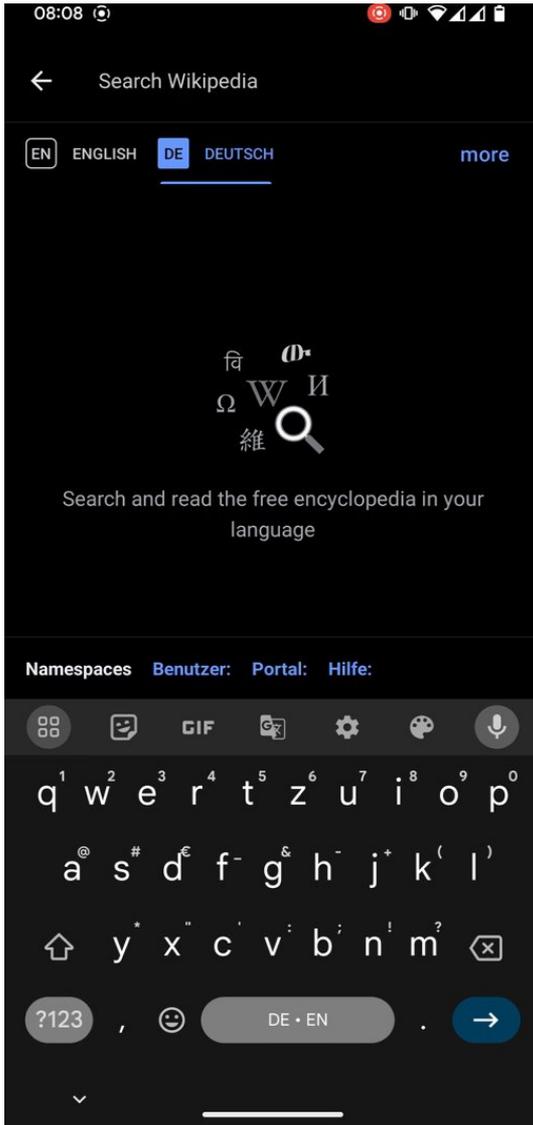


# REST – IMPLEMENTIERUNGSPRINZIP

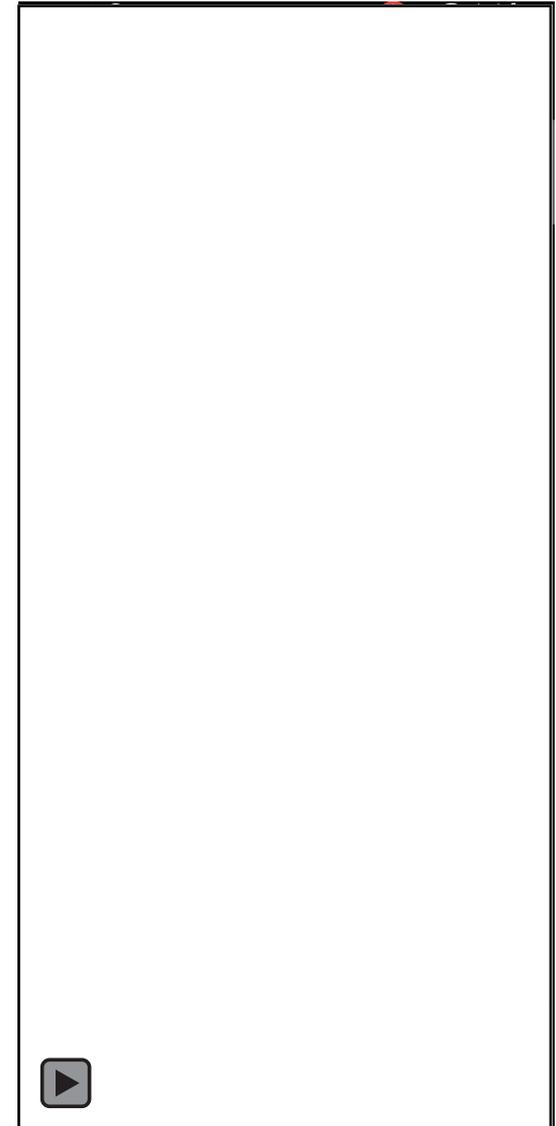
- ReST-Dienst (Dienst-API)
  - trennt Netzwerkanfragen vom GUI-Thread
  - kann fortlaufen während App inaktiv ist
  - Rückrufe zur Weiterleitung von Ergebnissen an die Dienstebene
- ReST-Methoden
  - generische Dienste für ReST-Kommunikation
  - implementiert GET/POST/...-Methoden
  - generiert Aufrufentitäten
  - verarbeitet Antworten



# BEISPIEL: WIKIPEDIA-CLIENT



wikiSearch(query, receiver)



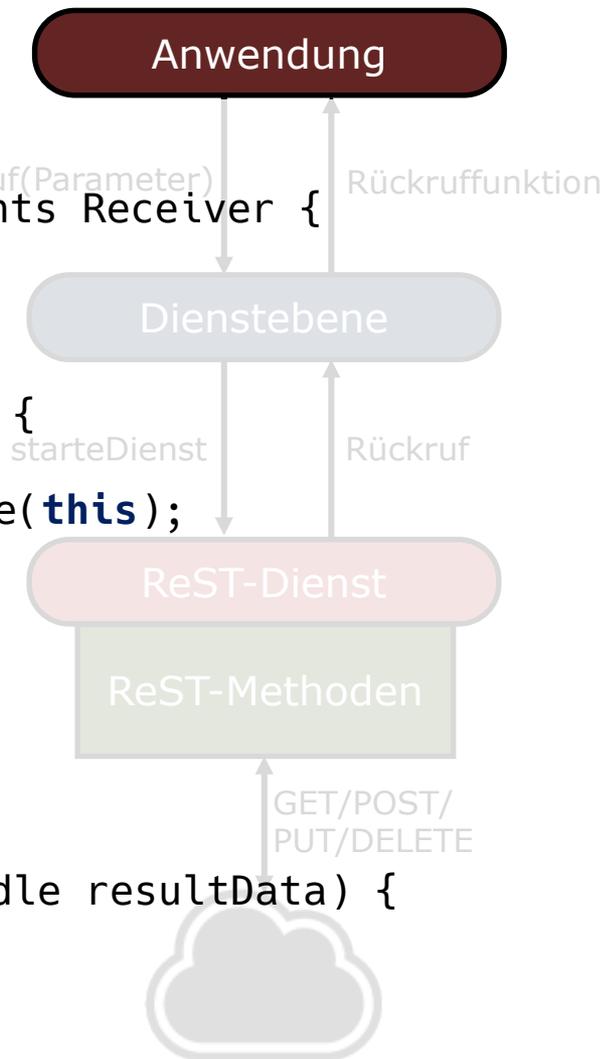
# ANDROID ACTIVITY

Beispiel: Wikipedia-Client zum Suchen und Anzeigen von Artikeln

```

public class MainActivity extends Activity implements Receiver {
    private WikiServiceHelper serviceHelper;
    public RESTReceiver mReceiver;
    ...
    public void onCreate(Bundle savedInstanceState) {
        ...
        serviceHelper = WikiServiceHelper.getInstance(this);
        mReceiver = new RESTReceiver(new Handler());
        mReceiver.setReceiver(this);
    }
    // Aufruf des ServiceHelper
    serviceHelper.wikiSearch(query, mReceiver);
    // Ergebnisempfang durch Rückruf
    public void onReceiveResult(int resultCode, Bundle resultData) {
        ...
    }
    ...
}

```



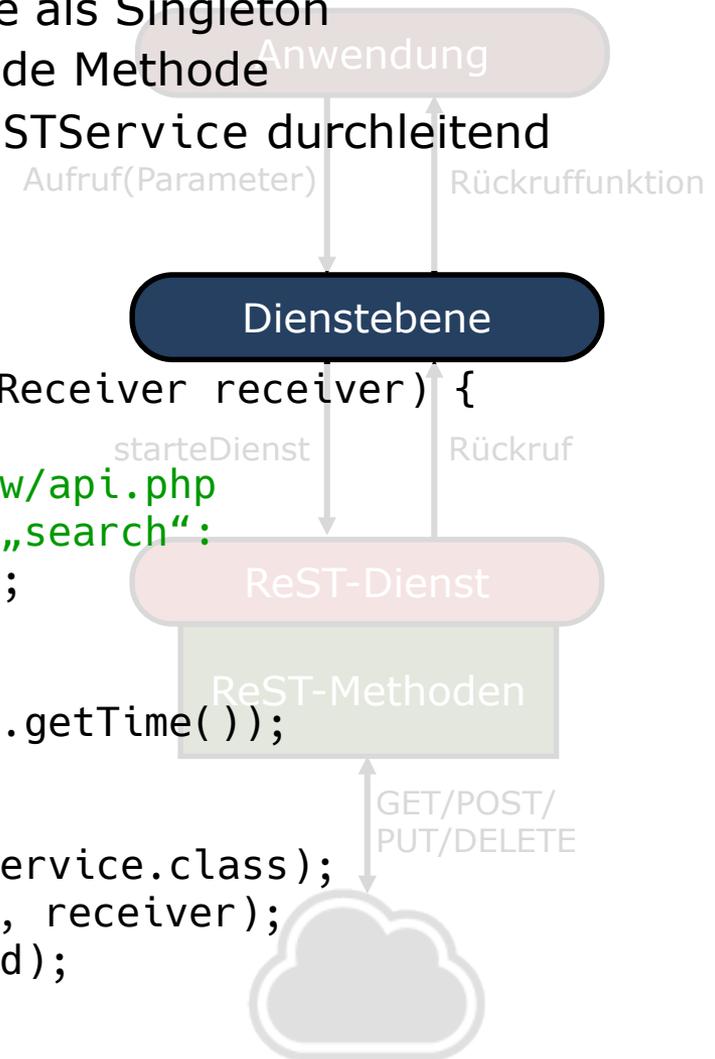
## SERVICE HELPER DER DIENSTEBENE

- einfache asynchrone API auf Anwendungsebene als Singleton
- erzeugt **Intents** und startet RESTService für jede Methode
- implementiert Empfänger (**Receiver**) → zum RESTService durchleitend

```

public class WikiServiceHelper
    implements Receiver {
    ...
    public void wikiSearch(String query, ResultReceiver receiver) {
        Bundle extras = new Bundle();
        // Zugriff auf https://de.wikipedia.org/w/api.php
        // erfordert die Parameter „action“ und „search“:
        extras.putString("action", "opensearch");
        extras.putString("search", query);
        // Anfrage-ID erzeugen
        String reqId = String.valueOf(new Date().getTime());
        // Receiver in HashMap hinterlegen
        callbacks.put(reqId, receiver);
        Intent intent = new Intent(context, RESTService.class);
        intent.putExtra(RESTService.RESULT_RECEIVER, receiver);
        intent.putExtra(RESTService.REQ_ID, reqId);
        context.startService(intent);
    }
    ...

```



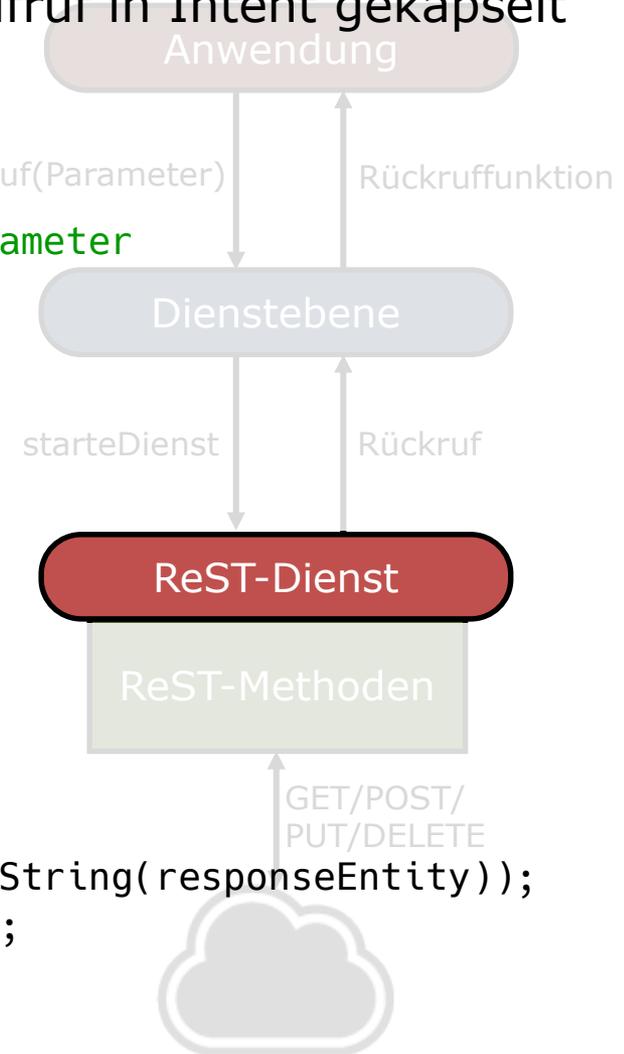
## REST-DIENST IN ANDROID

ReST-Dienst erweitert IntentService → ReST-Aufruf in Intent gekapselt

```

public class RESTService extends IntentService {
    ...
    protected void onHandleIntent(Intent intent) {
        // extrahiere URI der Ressource und weitere Parameter
        Uri action_uri = intent.getData();
        Bundle params = extras.getParcelable(PARAMS);
        // HTTP-Methode setzen
        int verb = extras.getInt(HTTP_VERB, GET);
        // Rückrufempfänger definieren und ID holen
        ResultReceiver receiver =
            intent.getParcelableExtra(RESULT_RECEIVER);
        String requestId = extras.getString(REQ_ID);
        ...
        // eigentlicher Aufruf der ReST-Methode
        ...
        Bundle resultData = new Bundle();
        resultData.putString(REST_RESULT, EntityUtils.toString(responseEntity));
        resultData.putString(ACTION, action.toString());
        ...
        receiver.send(statusCode, resultData);
        ...
    }
}

```

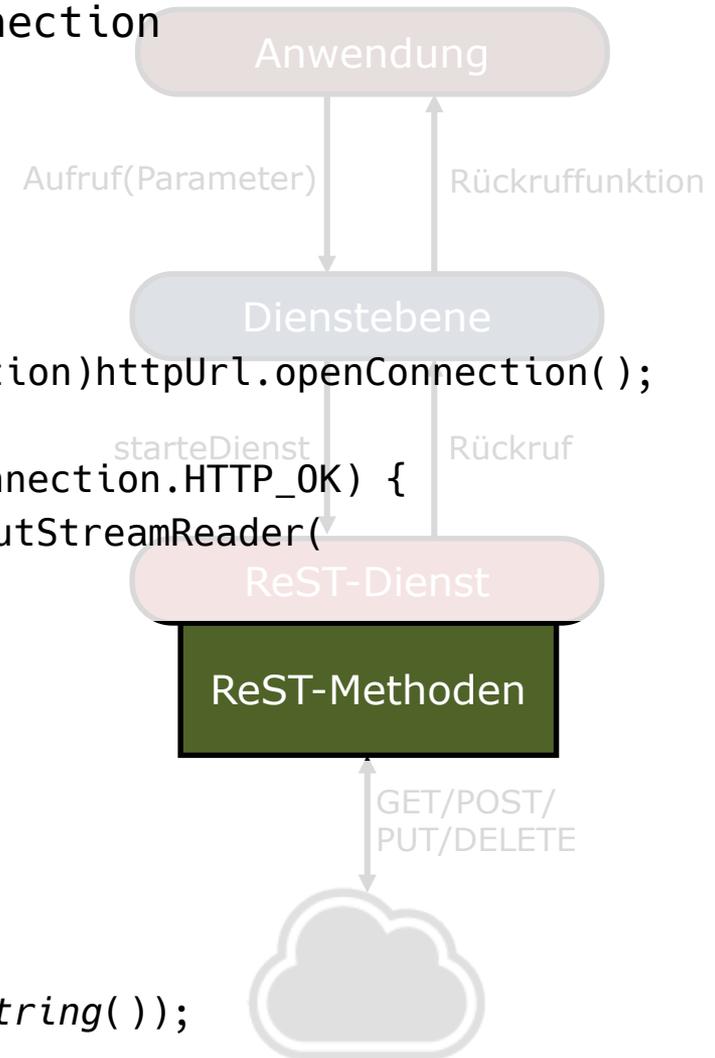


## REST-METHODEN IN ANDROID

ReST-Methoden basieren auf `java.net.URLConnection`

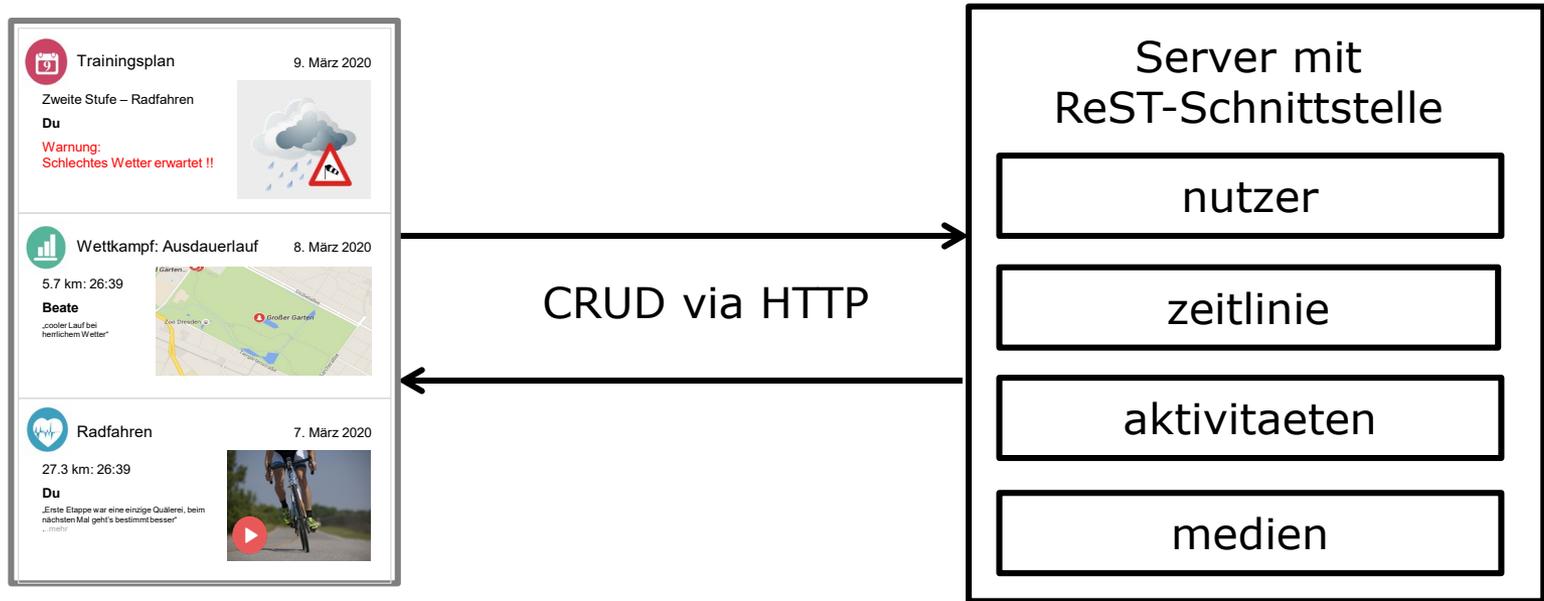
- bereiten HTTP-URL und HTTP-Anfrage-Inhalt vor
- führen die HTTP-Transaktion aus
- verarbeiten die HTTP-Antwort

```
URL httpUrL = new URL(urlString);
URLConnection httpConnection = (URLConnection)httpUrL.openConnection();
httpConnection.setRequestMethod("GET");
if (httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK) {
    BufferedReader in = new BufferedReader(new InputStreamReader(
        httpConnection.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();
    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();
    Bundle resultData = new Bundle();
    resultData.putString(REST_RESULT, response.toString());
    ...
}
```



# SOCIAL-FITNESS-APP – REST

- Abrufen von Zeitlinieneinträgen



- Problem mit ReST
  - heterogene Client-Anforderungen
  - Über-/Unterversorgung (**Over-Fetching** und **Under-Fetching**)

# Google Volley

# GOOGLE VOLLEY – HAUPTTEIGENSCHAFTEN

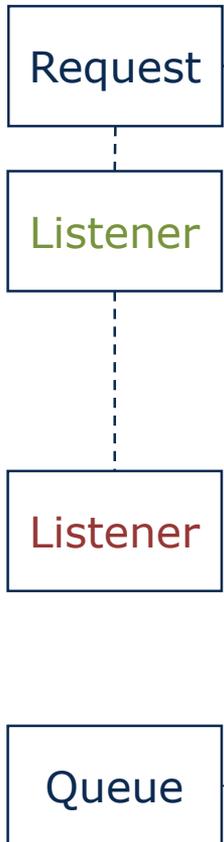
- begrenzte Bandbreite
  - Anfragepriorisierung
  - Anfrageabbruch
- Verbindungs- und Übertragungsfehler
  - Warteschlangen (**Queuing**)
  - Ergebnis-Caching
  - Anfragewiederholung

[springer2015]

# GOOGLE VOLLEY – GRUNDBAUSTEINE

- Anfragen (**Request**)
  - repräsentiert HTTP-Anfragen
  - Methoden: GET, POST, PUT, DELETE, ...
  - wohldefinierte Antworttypen (inkl. Standards wie String, JSON, ...)
- Anfragewarteschlange (**RequestQueue**)
  - einmaliges Element (**Singleton**) pro Anwendungsinstanz
  - verantwortlich für Terminierung und Initiierung von Anfragen
- Antwortempfänger (**ResponseListener**)
  - zum Empfang von Fehlern oder der eigentlichen Antwort

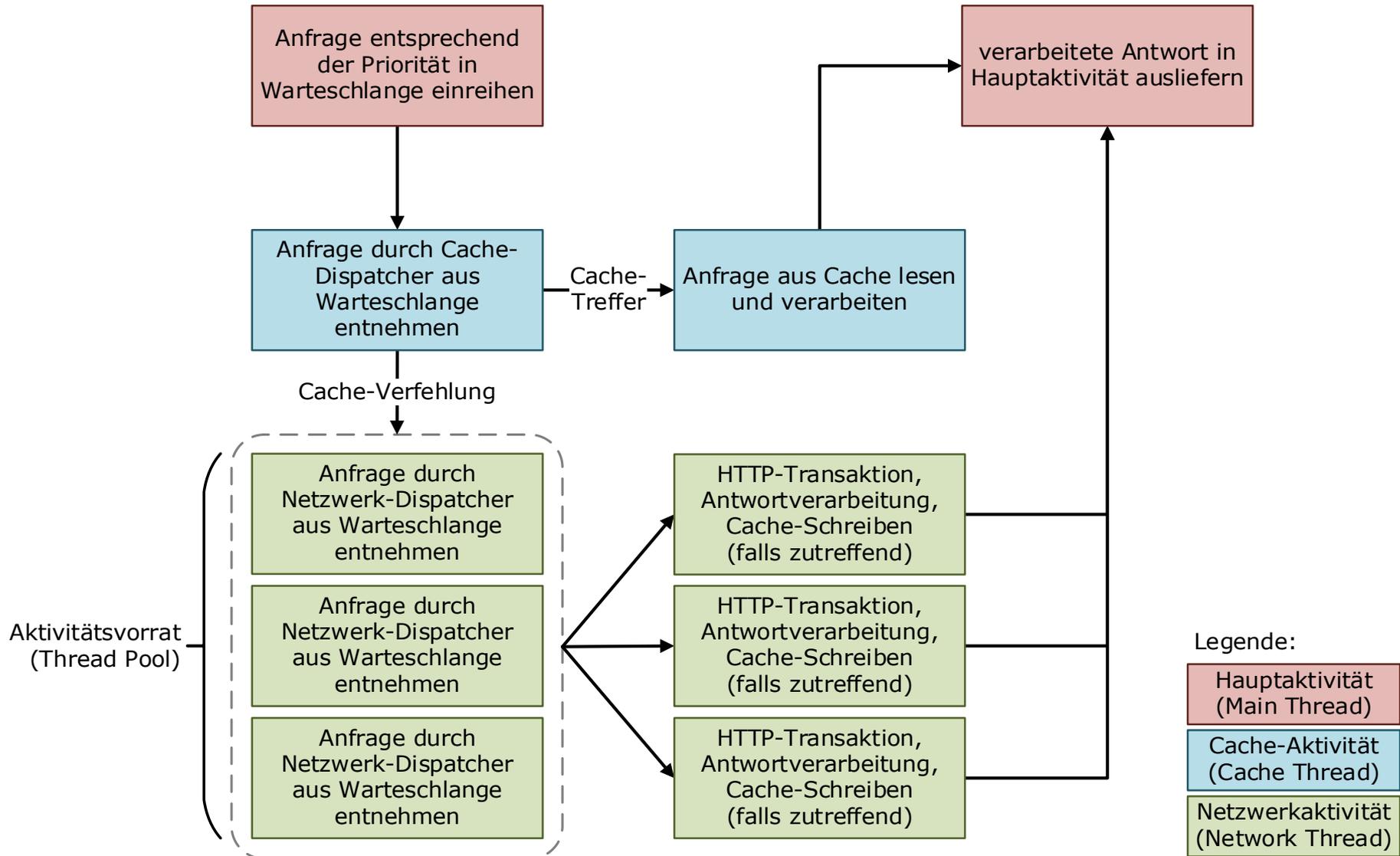
# GOOGLE VOLLEY – ANFRAGEINITIIERUNG



```

StringRequest req = new StringRequest(Request.Method.GET, "example.com",
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response) {
            // irgendwas tolles mit der Antwort machen...
        }
    },
    new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            // Fehler behandeln (Schuld jemand anderem geben)
        }
    }
);
// Anfrage der Warteschlange hinzufügen
Volley.newRequestQueue(this).add(req);
  
```

# GOOGLE VOLLEY – THREADING



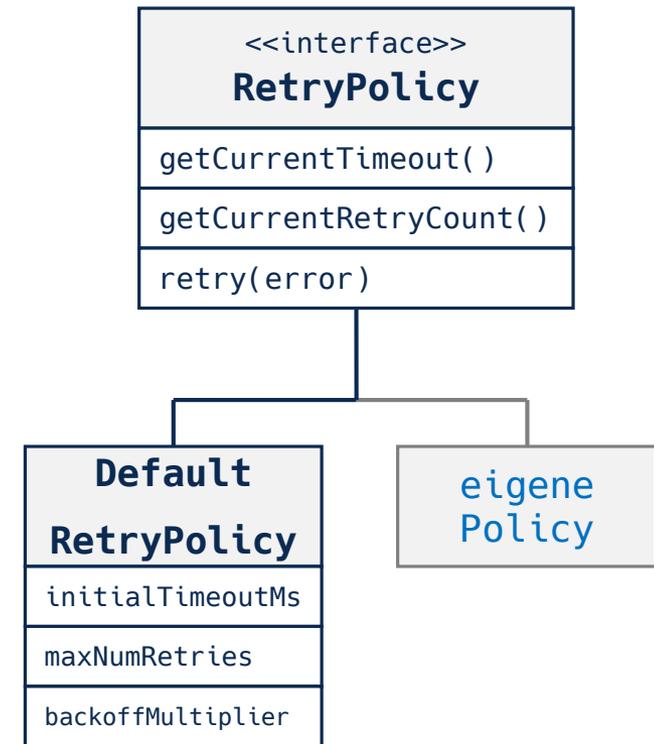
# GOOGLE VOLLEY – WIEDERHOLUNGSANFRAGEN

- Timeout
- Anzahl der Wiederholungen
- Backoff-Zeit

```
Req.setRetryPolicy(
    new DefaultRetryPolicy(
        initialTimeoutMs,
        maxNumRetries,
        backoffMultiplier
    )
);
```

Beispiel:

```
Req.setRetryPolicy(
    new DefaultRetryPolicy(1000, 3, 2.0f));
```

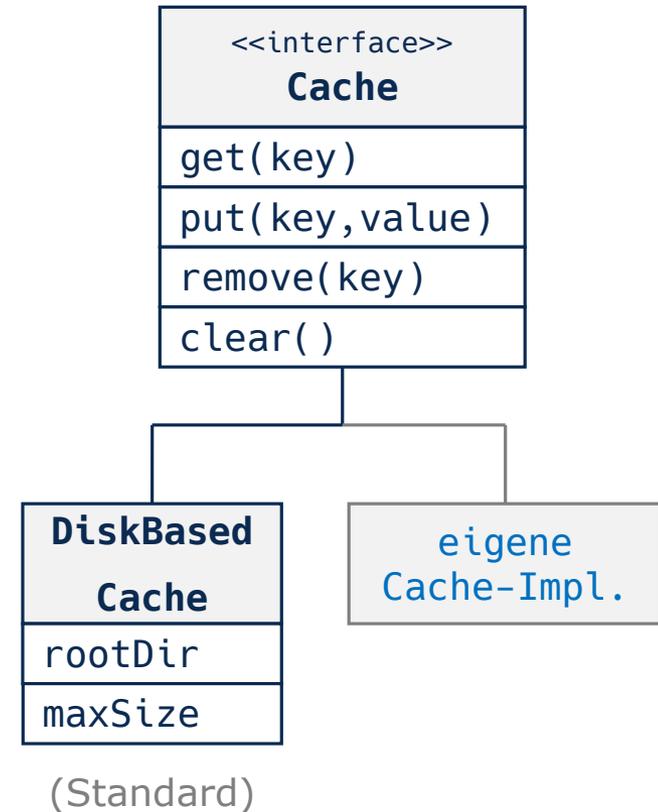


(Standard)

# GOOGLE VOLLEY – CACHING

- transparenter Antwort-Cache
- Cache-Schnittstelle

```
Cache cache = new DiskBasedCache(
    getCacheDir(),
    maxCacheSizeInBytes
);
RequestQueue queue = new RequestQueue(
    cache,
    network
);
```



# GOOGLE VOLLEY – PRIORISIERUNG

- nicht sofort verfügbar (kein **Out-of-the-Box**)
- vom Nutzer definierte Anfragen (abgeleitet vom `StringRequest`)

```
public abstract class PriorityRequest extends StringRequest {
    private Priority mPriority;
    public PriorityRequest(
        int method, String url, Response.Listener<String> listener,
        Response.ErrorListener errorListener, Priority priority
    ) {
        super(method, url, listener, errorListener);
        mPriority = priority;
    }
    @Override
    public Priority getPriority() {
        return mPriority;
    }
}
```

Priority
LOW
NORMAL
HIGH
IMMEDIATE

# GOOGLE VOLLEY – ANFRAGEABBRUCH

klare Abbruch-API

- individuelle Anfragen

```
req.cancel();
```

- Auswahl nach Tag

```
req.setTag("tag");
```

...

```
queue.cancelAll("tag");
```

- Auswahl nach Filter

```
queue.cancelAll(new LowPrioFilter());
```

```
public class LowPrioFilter implements RequestQueue.RequestFilter {
```

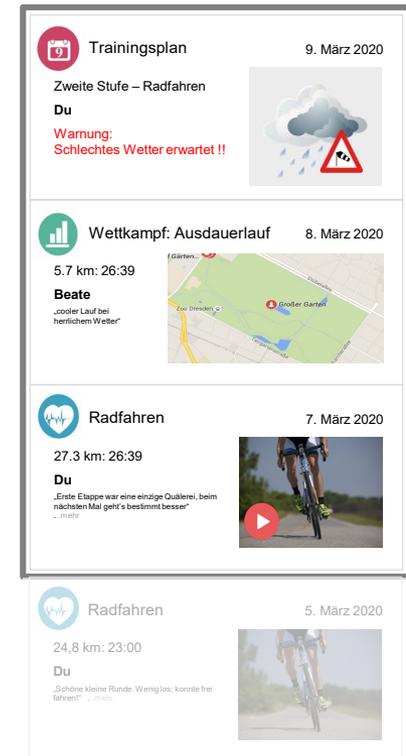
```
    @Override
```

```
    public boolean apply(Request<?> req) {
```

```
        return(req.getPriority() == Request.Priority.LOW);
```

```
    }
```

```
}
```



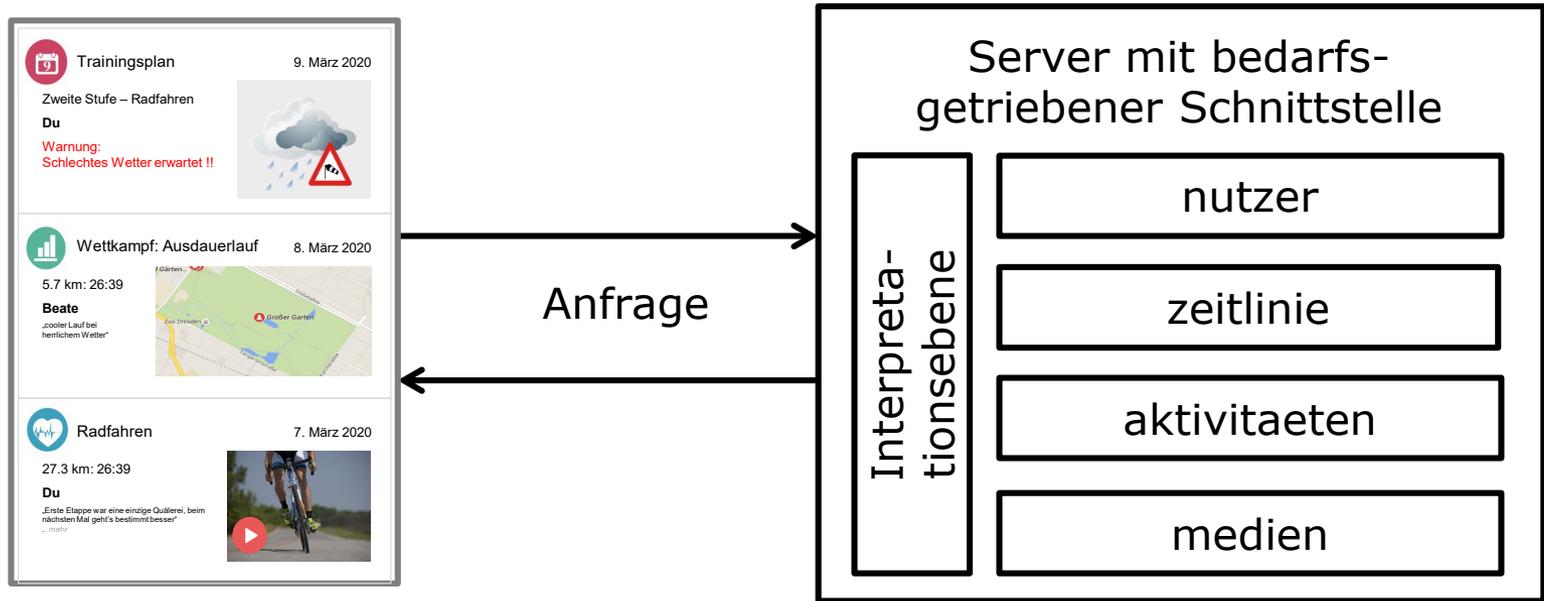
# bedarfsgetriebene Architekturen

# BEDARFSGETRIEBENE ARCHITEKTUREN [NOLAN, KOVAS; 2015]

## Drei Leitprinzipien

- **Bedarf**: Clients deklarieren ihre Anforderungen über eine Anfragesprache
- **Komposition**: Bedarfe für unterschiedliche Eigenschaften können über einen Rekursionsmechanismus mit in sich geschlossenen (Unter)Bedarfsmeldungen zusammengestellt werden
- Bedarf wird **rekursiv** durch den Dienst interpretiert (Interpretationsebene ist dem Backend ggü. agnostisch)

# SOCIAL-FITNESS-APP – BEDARFSGETRIEBENER ANSATZ



- Server spezifiziert verfügbare Daten durch ein Schema (**Introspektion**)
- Client erzeugt individuelle, dem Schema entsprechende, Anfragen
- Server validiert Anfragen gegen das Schema
- Auflöserfunktionen (**Resolver**) behandeln Anfragefelder und generieren Ergebnismenge (bspw. behandelt ein Resolver Nutzer und ihre Attribute, ein anderer behandelt die Aktivitäten)

## BEISPIEL: GRAPHQL [FACEBOOK; 2012]

- seit dem Jahr 2013 durch Facebook-Apps auf iOS und Android verwendet

```

type Nutzer {
  id : ID !
  // das ! ist notwendig
  name : String
  historie : [Zeitlinie]
}

type Zeitlinie {
  id : ID !
  name : String
  besitzer : Nutzer
  teilnehmer : [Nutzer]
  aktivitaeten : [Aktivitaet]
}

type Aktivitaet {
  id : ID !
  name : String
  typ : String
  zeitlinie : Zeitlinie
  medien : [Medium]
}

{
  Nutzer ( id : 123 ) {
    name
    zeitlinie ( id : 1 ) {
      name
      aktivitaeten {
        name
        typ
      }
    }
  }
}

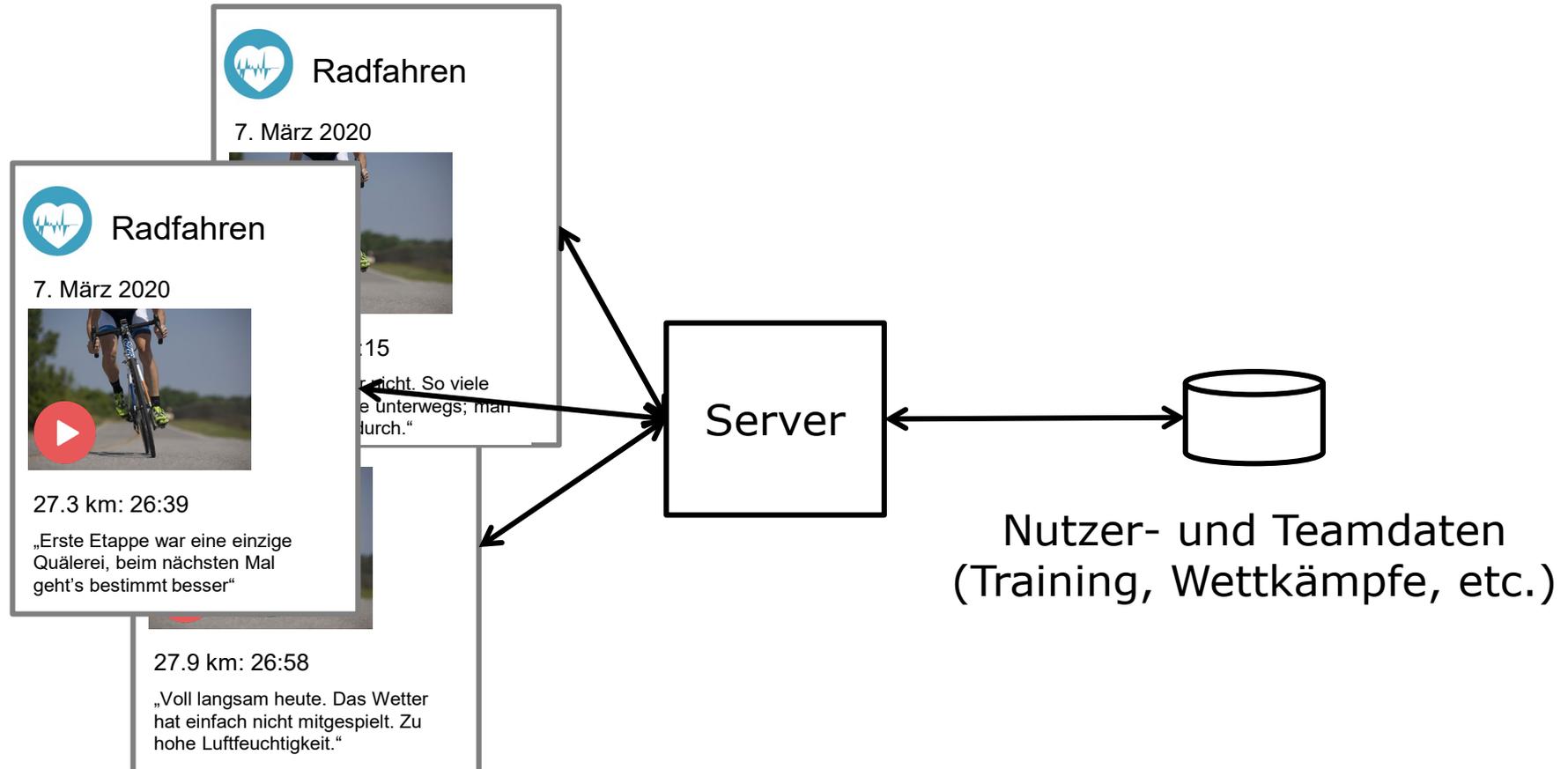
```

## BEDARFSGETRIEBENE ARCHITEKTUREN

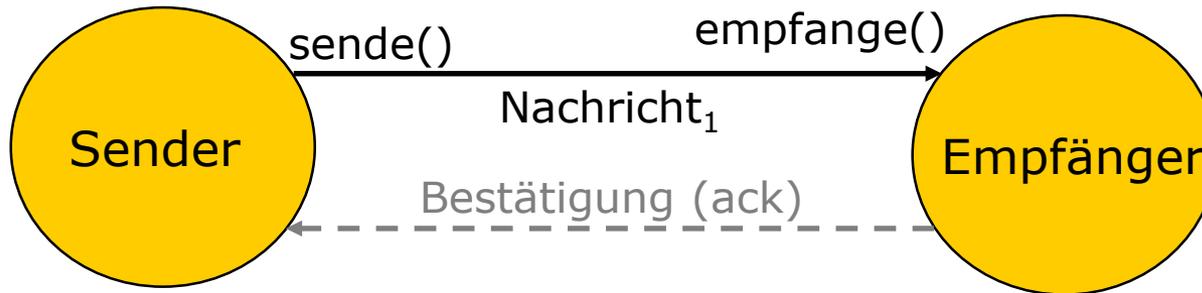
- Anfragen werden vom Client festgelegt
  - Server legt keine statische Schnittstelle fest
  - variable Ergebnismenge (bspw. Attribut- und Ressourcenteilmenge)
- Komposition von Anfragen
  - hierarchische Anfragen
  - ressourcenübergreifende Anfragen
- starke Schema- & Anfragetypisierung → Server validiert vor Bearbeitung
- speicher-, protokoll- und programmiersprachen-agnostisch
- stabile Anfrageschnittstelle
  - erweiterbar durch Schema
  - benötigt keine Versionierung

# SOCIAL-FITNESS-APP – KONNEKTIVITÄTSHERAUSFORDERUNG

- Distribution der Aktivitäten auf multiple Empfänger (Push vs. Pull)

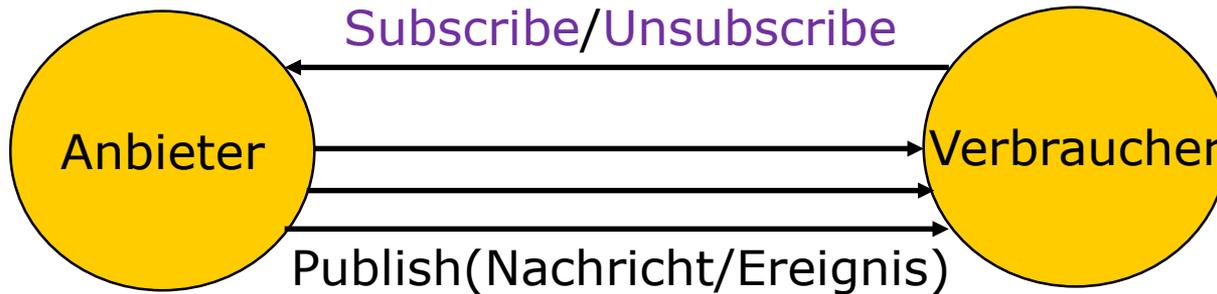


# NACHRICHTENDURCHREICHUNG



- grundlegendes Kommunikationsschema
- **asynchron** (wenn ohne Bestätigungen)
  - keine Zustellgarantie
- **synchron** (wenn mit Bestätigungen)
  - Client und Server müssen gleichzeitig online sein
  - Sendender ist kurzzeitig blockiert

## ABONNEMENT (PUBLISH/SUBSCRIBE)



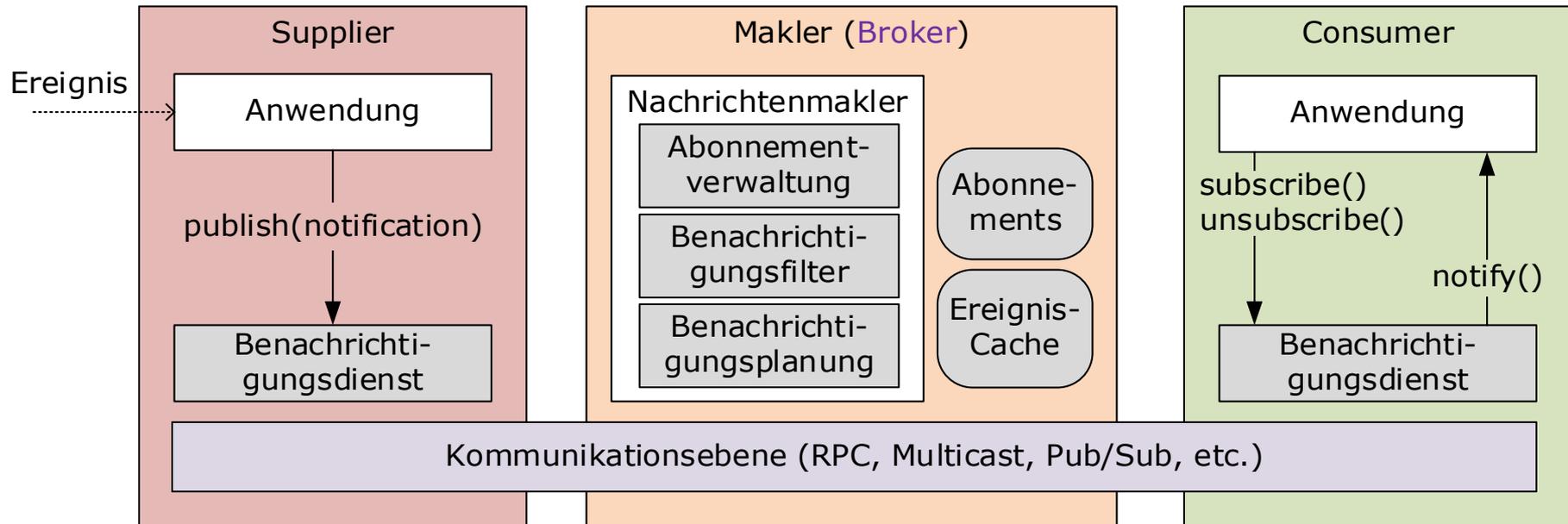
- zwei Rollen
  - Anbieter (**Supplier**) – erzeugt Nachrichten
  - Verbraucher (**Consumer**) – verarbeitet Nachrichten
- m:n-Kommunikation basierend auf Nachrichten
- inhärent asynchron
- flexibles Binden auf Basis von Abonnements (**Subscription**)
- direkte Kommunikation zwischen Anbieter und Verbraucher

# NACHRICHTENKANAL



- lose Kopplung – kein direkter Nachrichtenaustausch zwischen Anbietern und Verbrauchern
- Beziehung zwischen Anbietern und Verbrauchern basiert auf
  - Kanalauswahl – alle Nachrichten eines Kanals empfangen (**Channel Selection**)  
→ `subscribe("channel")`
  - Betreff – Filterung nach Schlüsselwort (**Topic Selection**) → `subscribe("topic")`
  - Hierarchie – Pfad in Baum bestimmt Unterbaum; Abonnement aller Knoten im Unterbaum (**Path Selection**) → `subscribe("de/sachsen/dresden/temperatur")`
  - Inhalt – semantische Filterung nach Inhalt (**Content Selection**)  
→ `subscribe(messageContains="temperature AND Dresden")`

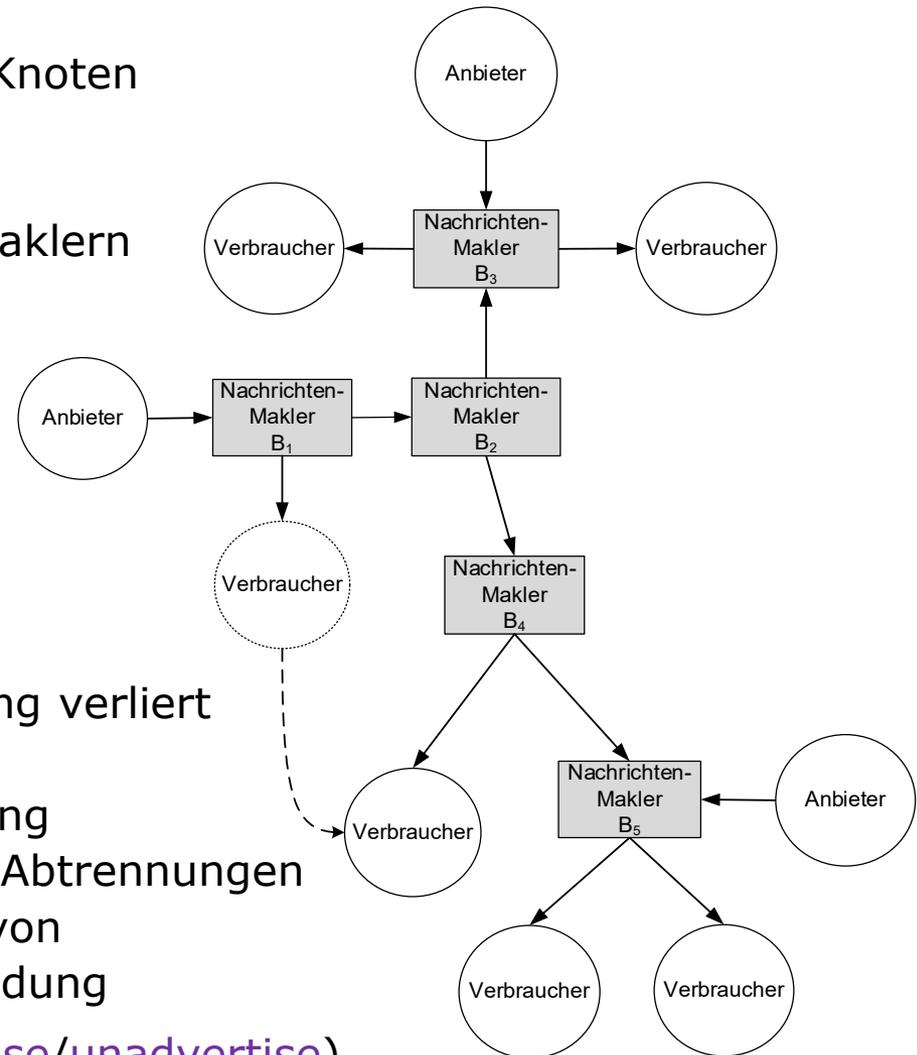
# MEHRSCHICHTIGE ARCHITEKTUR



- Benachrichtigung – Datum repräsentiert Ereignis
- Überlagerungsstruktur
  - ereignisbasierte Kommunikation zwischen Anbieter und Verbraucher
  - diverse zugrunde liegende Kommunikationsmechanismen

# MOBILITÄTSPROBLEME

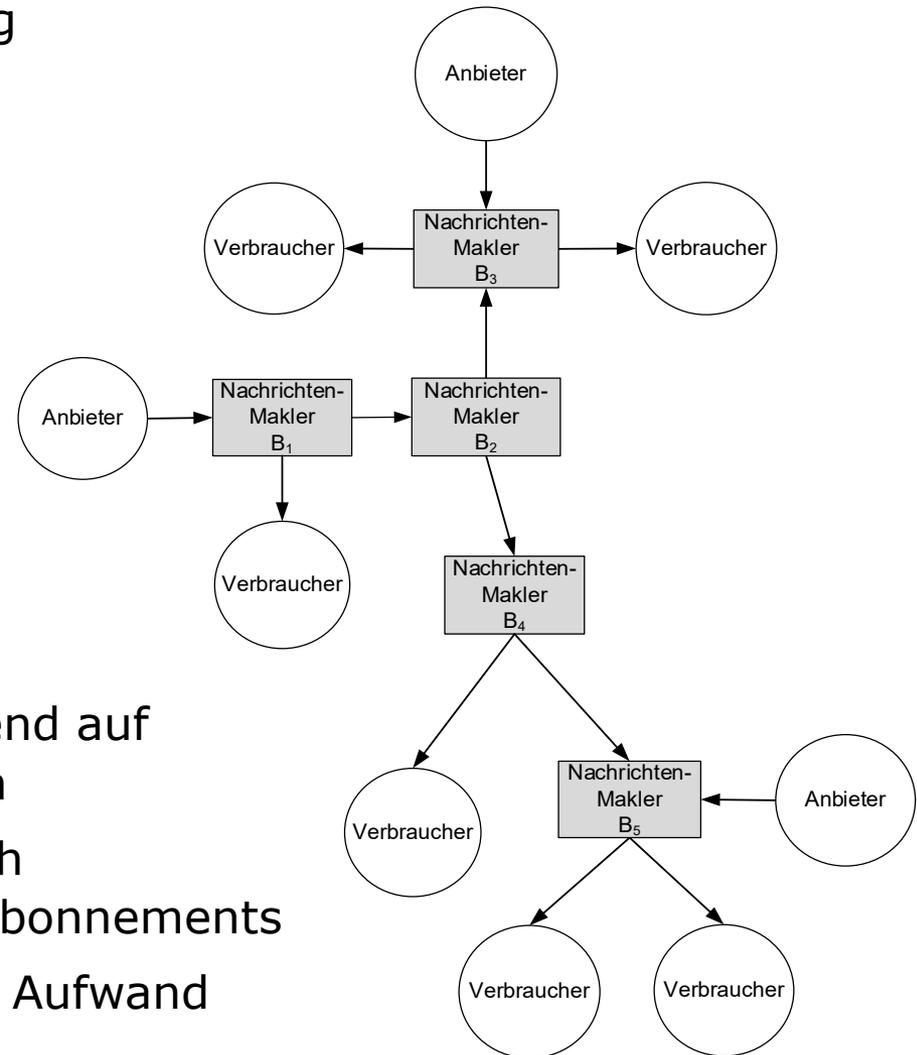
- physische Mobilität
  - Anbieter/Verbraucher auf mobilen Knoten
    - Ortsänderungen
    - Abtrennungen
  - transparente Übergabe zwischen Maklern
  - Anpassung der Benachrichtigungsrouten
- Verbraucher
  - explizites Re-Abonnieren am neuen Knoten
  - moveOut/moveIn-Operationen zur Abtrennung/Verbindung
  - nicht möglich, falls Client Verbindung verliert
  - fehlende Benachrichtigungen
  - erweiterte Infrastrukturunterstützung
    - **Heartbeat** zum Erkennen von Abtrennungen
    - automatische Neuzuweisung von Abonnements nach Neuverbindung
- Anbieter nutzen Ankündigungen (**advertise/unadvertise**)



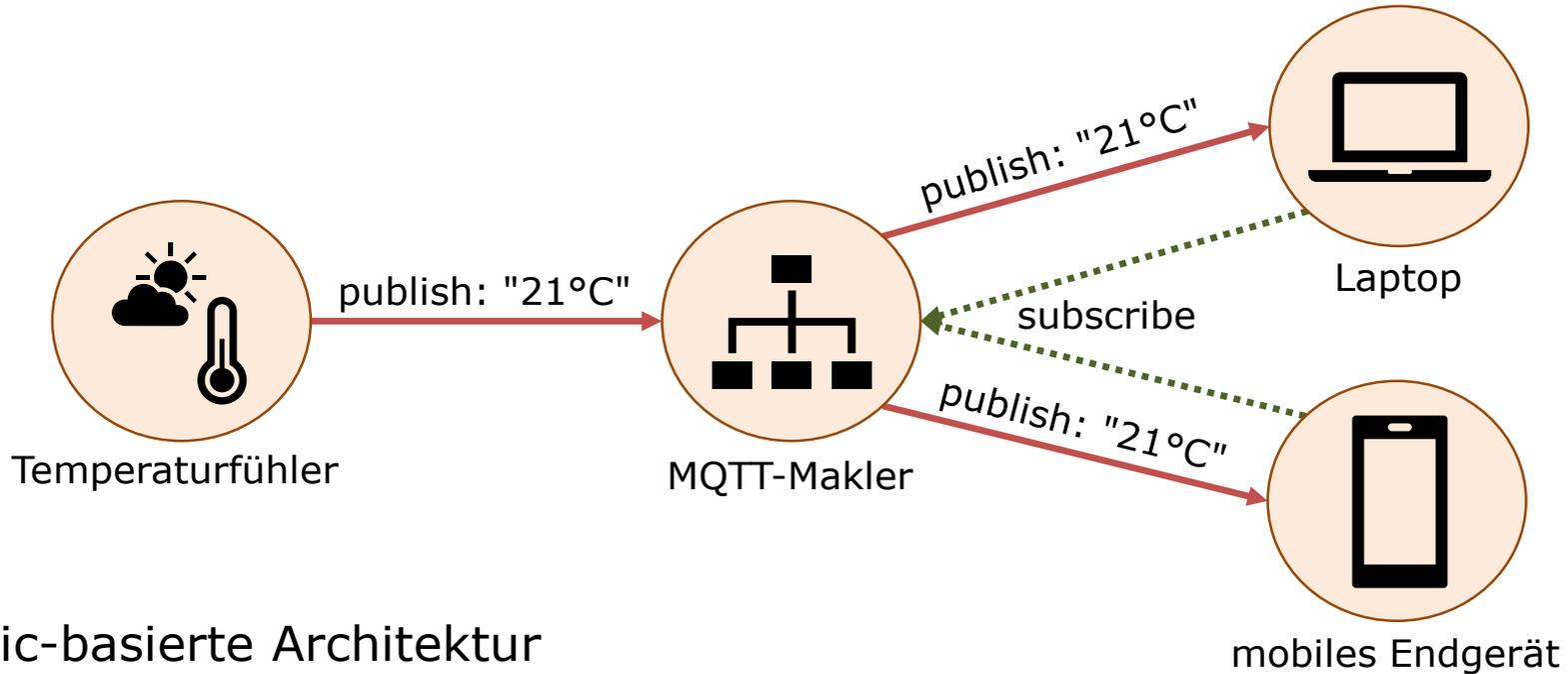
# ARCHITEKTUREN

## verteilte Benachrichtigungswegeplanung

- Fluten (**Flooding**)
  - Makler leitet Nachrichten an alle Nachbarn weiter
  - nur Makler mit direkt verbundenen Verbrauchern prüfen Filter
- filterbasiert Wegeplanung
  - Makler hält Wegetabellen mit (Filter; Ziel)-Paaren
  - permanente Aktualisierung basierend auf Subscribe/Unsubscribe-Ereignissen
  - Optimierung durch Vereinigung sich überlappender oder redundanter Abonnements
  - Ausgleich: Wegetabellengröße und Aufwand bei Aktualisierungen



# BEISPIEL: MESSAGE QUEUE TELEMETRY TRANSPORT (MQTT)



- Topic-basierte Architektur
- jede Nachricht hat ein Topic
- hierarchisch organisiert → Abonnements auf unterschiedlichen Ebenen

# MQTT – DIENSTQUALITÄT (QUALITY OF SERVICE)

drei Dienstqualitätsebenen

- höchstens einmal (**at-most-once**)

Ebene 0 garantiert eine Zustellung größter Mühe. Eine Nachricht wird durch den Empfänger nicht bestätigt und durch den Sender nicht gespeichert oder wiederholt. Die zugrunde liegende Garantie ist die von TCP (**fire and forget**).

- mindestens einmal (**at-least-once**)

Ebene 1 garantiert, dass Nachrichten mindestens einmal an den Empfänger zugestellt werden. Vervielfältigung nicht ausgeschlossen.

- genau einmal (**exactly-once**)

Ebene 2 garantiert, dass jede Nachricht mindestens und maximal einmal zugestellt wird. Sie ist die sicherste, aber auch langsamste Qualitätsebene. Die Garantie basiert auf zwei Verbindungen zwischen Anbieter und Verbraucher.

# MQTT – ABTRENNUNGSBEHANDLUNG

- gesicherte Nachrichten (**Retained Messages**):  
letzte Nachricht eines Topics wird vom Broker gespeichert und neuen Abonnenten automatisch zugestellt
- Verfügung (**Last Will and Testament / LWT**):  
Anbieter hinterlegt besondere Nachricht beim Broker, welche an alle Abonnenten ausgeliefert wird sobald er unerwartet abgetrennt wird
- beharrliche Sitzung (**Persistent Sessions**)
  - im Falle häufiger Abtrennungen
  - Broker hält persistent Sitzungsinformationen und alle Nachrichten eines Verbrauchers
  - neuverbundene Verbraucher müssen nicht reabonnieren, sondern erhalten sofort alle persistent zwischengespeicherten Nachrichten

## MQTT-BEISPIEL

- clientseitige Bibliothek: Eclipse Paho
- MQTT-Broker: Mosquitto
- Beispiel-Client für Android

### MQTT-Verbindungsaufbau

```
protected void onStart() {  
    super.onStart();  
    try {  
        MemoryPersistence persistence = new MemoryPersistence();  
        mqttClient = new MqttClient(  
            "tcp://198.51.100.222:1883",  
            "AndroidTest",  
            persistence  
        );  
        mqttClient.connect();  
        mqttClient.setCallback(this);  
    }  
    catch (MqttException e) { ... }  
}
```

# MQTT-BEISPIEL

## MQTT-Rückruf

```
public void connectionLost(Throwable cause) {
    Log.d("Main", "connection lost: " + cause);
}

public void messageArrived(String topic, MqttMessage message)
throws Exception {
    Log.d("Main", "message received for topic: " + topic
        + ",\r\nmessage: " + message);
}

public void deliveryComplete(IMqttDeliveryToken token) {
    Log.d("Main", "deliveryComplete for token: " +
token.getMessageId());
}
```

# MQTT-BEISPIEL

## MQTT-Abonnement

```
private boolean subscribe(String topic, int qos) {  
    try {  
        String topic = "topic/example";  
        int qos = 1;  
        mqttClient.subscribe(topic, qos);  
    }  
    catch (MqttException e) { ... }  
}
```

## MQTT-Veröffentlichung

```
MqttMessage message = new MqttMessage("Hello World".getBytes());  
mqttClient.publish("topic/example", message);
```

# MQTT-BEISPIEL

## MQTT-Verbindungsabbau

```
protected void onStop() {  
    super.onStop();  
    if (mqttClient != null) {  
        if (mqttClient.isConnected()) {  
            try {  
                mqttClient.disconnect();  
            }  
            catch (MqttException e) { ... }  
        }  
    }  
}
```

## VERGLEICH DER INTERAKTIONSSCHEMATA

### Request/Response (RPC)

- enge Kopplung (Neubinden schwer)
- 1:1-Kommunikation
- synchron (Probleme bei Abtrennung)
- vertrautes Interaktionsschema mit Anfragen und Antworten
- Client und Server werden synchronisiert
- zusätzlicher Aufwand für Zuverlässigkeit (at-most-once)
- Client/Server-Systeme (Datenverarbeitung mit Ergebnissen)

### Publish/Subscribe (Nachrichten)

- lose Kopplung (dynamisches Neubinden)
- m:n-Kommunikation
- asynchron (Abtrennungsbehandlung)
- simples Interaktionsschema
  - Basis für komplexere Interaktionen
  - zusätzlicher Aufwand für komplexe Interaktionen notwendig
- Synchronisierung muss explizit angestoßen werden
- keine Ergebnisse für Nachrichten
- zuverlässiger Nachrichtenaustausch auf Basis der Nachrichtenschlangen
- Lastverteilung, Parallelisierung, Stapelverarbeitung, Ereignisverteilung

# bidirektionale Kommunikation

## BIDIREKTIONALE KOMMUNIKATION – WEBSOCKETS

- persistente, bidirektionale, vollduplex TCP-Verbindung
- arbeitet auf bestehender oder neuer TCP-Verbindung
- initiiert durch WebSocket-Handshake via HTTP

```
GET /websocket HTTP/1.1
```

```
Host example.com
```

```
Origin: https://example.com
```

```
Connection: Upgrade
```

```
Upgrade: websocket
```

```
Sec-WebSocket-Key: mx3JhmBHL1EzLApfelhXDw==
```

```
Sec-WebSocket-Protocol: chat, superchat
```

```
Sec-WebSocket-Version: 13
```

- bestätigt vom Server, ebenfalls via HTTP

```
HTTP/1.1 101 Switching Protocols
```

```
Upgrade: websocket
```

```
Connection: Upgrade
```

```
Sec-WebSocket-Accept: HSacBirne0UkAGmm50zZy2HaaBc=
```

```
Sec-WebSocket-Protocol: chat
```

## BIDIREKTIONALE KOMMUNIKATION – WEBSOCKETS

- Erkennung prokurierter Kommunikation
  - Websocket selbst erkennt Proxys nicht
  - unterliegendes HTTP kann Proxy-Verbindung etablieren (**HTTP Tunnel**)
    - Client

```
CONNECT proxy.example.com:22 HTTP/1.1
Proxy-Authorization: Basic encoded-credentials
```
    - Server

```
HTTP/1.1 200 OK
```
    - anschließend sendet Client alle Anfragen an Proxy

```
SSH-2.0-OpenSSH_4.3
...
```
- Austausch von Klartext oder Text-Blöcken über etablierte Verbindung
- unverschlüsselt (`ws://...`) oder TLS-verschlüsselt (`wss://...`)  
→ Upgrade von `https://...` auf `ws://...` nicht möglich, umgekehrt schon!
- initiale HTTP-Verbindung kann nach WS-Etablierung geschlossen werden

# Zusammenfassung und Aufgaben

# ZUSAMMENFASSUNG

- Anfrage/Antwort-Interaktion (Request/Response)
  - mobiles RPC-Konzept
  - ReST-Dienste
  - Beispiel: Google Volley
- Bedarfsgesteuerte Architekturen
  - clientseitige Anfragen
  - Beispiel: GraphQL
- ereignisgetriebene Kommunikation
  - Abonnement-Mechanismus (Publish/Subscribe)
  - Ereigniskanal (Event Channel)
  - Beispiel: MQTT
- bidirektionale Kommunikation am Beispiel WebSockets

## AUFGABEN

- Diskutieren Sie mit Ihren Kommilitonen, wie Sie einen WWW-Server zu einem ReST-Dienst erweitern können.  
Welche Anpassungen sind an Firewalls notwendig, um ReST nutzen zu können? Welche Anpassungen an Proxys?
- Googlen Sie nach *XMPP* und diskutieren Sie mit Ihren Kommilitonen Ähnlichkeiten und Unterschiede zu MQTT!  
Was sind sinnvolle Einsatzszenarien für Message Queues?  
Wie würden Sie eine Chat-Anwendung umsetzen in MQTT? ...in XMPP?  
(Vergessen Sie nicht, dass Anwender offline sein können, aber trotzdem Nachrichten zugestellt bekommen wollen!)
- Diskutieren Sie mit Ihren Kommilitonen sinnvolle Szenarien für WebSocket-Kommunikation.  
Wann ist der parallele Einsatz von HTTP und WS sinnvoll?  
Wann lohnt es sich, die HTTP-Verbindung zu schließen? Wann nicht?

## REFERENZEN

Bakre, A.V. & Badrinath, B.R. M-RPC: A Remote Procedure Call Service for Mobile Clients; Mobile Computing and Networking, 1995, 97-110

Fiege, L., Mühl, G., Pietzuch, R.: Distributed Event-Based Systems. Springer, Berlin, 2006

Google I/O 2010 – Android REST client applications  
<http://www.youtube.com/watch?v=xHXn3Kg2IQE>

Google (2013): „Transmitting Network Data Using Volley“. Android Developers. Abgerufen am 15.06.2016 von  
<https://developer.android.com/training/volley>.

Google (2016): „Material Design“. Abgerufen am 15.06.2016 von  
<https://www.google.com/design/spec/components/lists.html>

Kirkpatrick, Ficus (2013): „Google I/O 2013 - Volley: Easy, Fast Networking for Android“. *YouTube*. Abgerufen am 15.06.2016 von  
<https://www.youtube.com/watch?v=yhv8l9F44qo>.

MQTT: <http://mqtt.org>