Netzwerkprotokolle zur Datenkommunikation

Prof. René Pawlitzek und Robert Schöch, Version 1.2, 12.6.2023

0. Abstrakt

In diesem Wissensnugget werden drei bekannte Netzwerkprotokolle zur Datenübertragung bzw. die Cloud-Anbindung vorgestellt. Folgende Protokolle werden behandelt: HTTP, MQTT und CoAP. Bei allen drei Protokollen handelt es sich um Protokolle der Anwendungsschicht.

1. Einleitung

Ein Netzwerkprotokoll[1] ist ein Kommunikationsprotokoll für den Austausch von Daten zwischen Computern, die in einem Rechnernetz miteinander verbunden sind. Genauso wie ein menschliches Protokoll regelt ein Netzwerkprotokoll die gesamte Kommunikation zwischen zwei Rechnern, welche mit einem Netzwerk verbunden sind.

Netzwerkprotokolle definieren das Format, die Reihenfolge der zwischen den Netzeinheiten gesendeten und empfangenen Nachrichten und die bei der Übertragung und dem Empfang von Nachrichten durchgeführten Aktionen.

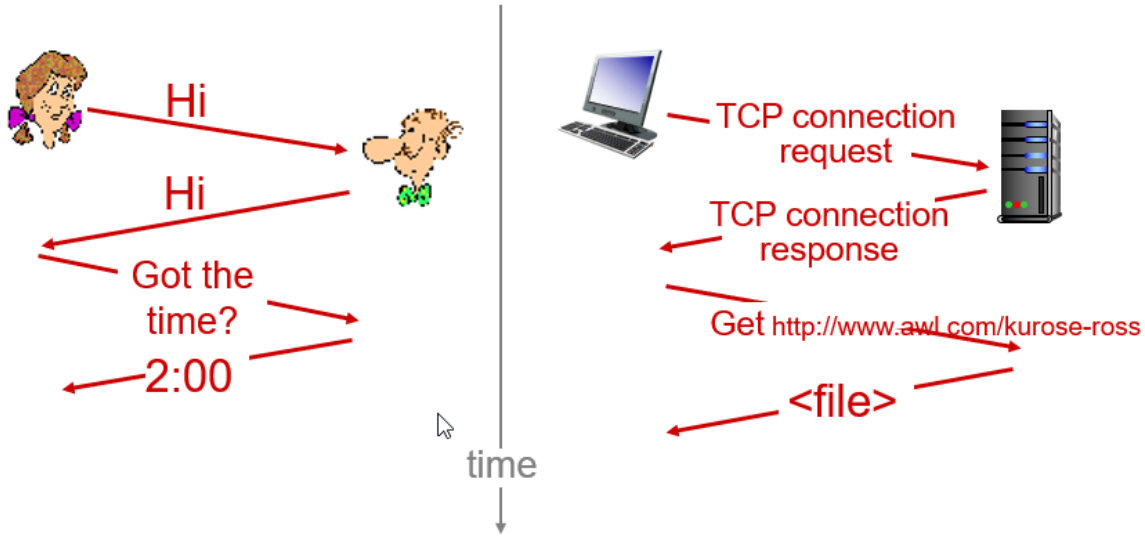

Abbildung 1: Menschliches Protokoll vs. Computernetzwerk-Protokoll

[1] https://de.wikipedia.org/wiki/Netzwerkprotokoll

Der Austausch von Nachrichten erfordert häufig ein Zusammenspiel verschiedener Protokolle, die unterschiedliche Aufgaben übernehmen. Um die damit verbundene Komplexität beherrschen zu können, werden die einzelnen Protokolle in Schichten organisiert. Im Rahmen einer solchen Architektur gehört jedes Protokoll einer bestimmten Schicht an und ist für die Erledigung spezieller Aufgaben zuständig. Protokolle höherer Schichten verwenden Dienste von Protokollen tieferer Schichten. Zusammen bilden die so strukturierten Protokolle einen Protokollstapel – in Anlehnung an das ISO-OSI-Referenzmodell[2]. Nachrichten einer bestimmten Schicht werden auch als Protokolldateneinheiten (protocol data units) bezeichnet.

Abbildung 2: Der Internet-Protokollstapel

Im Folgenden wollen wir uns mit den Netzwerkprotokollen der Anwendungsschicht beschäftigen. Ein Anwendungsschichtprotokoll definiert wie sich die Prozesse einer Anwendung, die auf verschiedenen Endsystemen (Computern in einem Netzwerk) laufen, Nachrichten zusenden. Wir betrachten:

- HTTP[3], das Hypertext Transfer Protocol, mit dem Webbrowser und Webserver Nachrichten austauschen.
- MQTT[4], Message Queuing Telemetry Transport, ein einfaches und schlankes Publisch/Subscribe Protokoll mit dem eingeschränkte Geräte Nachrichten in unzuverlässigen Netzwerken mit hoher Verzögerung und geringer Bandbreite austauschen können.
- CoAP[5], Constrained Application Protocol, ein Web-Transfer-Protokoll für das Internet of Things (IoT), das dem HTTP-Protokoll sehr ähnlich ist.

[2] https://de.wikipedia.org/wiki/OSI-Modell
[3] https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[4] https://de.wikipedia.org/wiki/MQTT
[5] https://de.wikipedia.org/wiki/Constrained_Application_Protocol

2. Das Web und HTTP

Das World Wide Web (kurz Web) ist, so wie E-Mail, eine Anwendung für das Internet[6]. Das WWW macht Ressourcen (Webseiten) leicht zugreifbar, versteckt die Tatsache, dass Ressourcen weltweit verteilt sind, und ist offen und skalierbar[7]. Es machte das Internet von einem Datennetzwerk unter vielen zum grössten Datennetz der Welt. Informationen können mit Einfachheit ins Netz gestellt werden, wo sie jederzeit rund um die Uhr verfügbar sind. Das Web setzt sich aus mehreren Bestandteilen zusammen, darunter das Protokoll (HTTP) für den Datenaustausch, ein Standard für die Formate der Webseiten (HTML), Webbrowser (synonym für Client-Programm) und Web-Server (synonym für Server-Programm).

Das Hypertext Transfer Protocol (HTTP) bildet das Herz des Webs und deren technische Spezifikationen sind in einem sogenannten Request for Comments (RFC)[8] definiert. HTTP bestimmt wie Webbrowser Webseiten von Web-Servern anfordern und wie Server die Webseiten zu den Clients übertragen. Für eine sichere Übertragung von Inhalten gibt es zu HTTP den ergänzenden und darauf aufbauenden Standard HTTPS, welcher Transportverschlüsselung beinhaltet.

Eine Webseite beinhaltet Objekte: Ein Objekt ist eine Datei – etwa eine HTML-Datei, ein JPEG-Bild, ein Java-Applet oder ein Videoclip, die unter einer einzelnen URL[9] (Uniform Resource Locator) zu erreichen ist. Enthält eine Webseite beispielsweise HTML-Text und fünf JPEG-Bilder, dann hat die Webseite sechs Objekte: die Basis HTML-Datei plus die fünf Abbildungen, die mithilfe der URLs referenziert werden. Jede URL besteht aus der Angabe des Protokolls, dem Hostnamen des Servers, auf dem sich die Objekte befinden, sowie dem Pfadnamen des Objekts. Beispielsweise setzt sich die URL

http://localhost:6789/data/picture.jpg

aus http:// als Kennzeichnung des Protokolls, localhost:6789 als Hostname mit Portnummer und /data/picture.jpg als Pfadangabe zusammen.

Beliebte Web-Server sind Apache und der Microsoft Internet Information Server, und bekannte Webbrowser sind Internet Explorer, Firefox und Chrome. Fordert ein Benutzer eine Webseite an (zum Beispiel durch Klicken auf einen Hyperlink), dann sendet der Webbrowser HTTP-Request-Nachrichten für die Objekte auf der Seite an den WebServer. Der Web-Server erhält die Anforderungen und antwortet mit HTTP-Response-Nachrichten, welche die Objekte enthalten. In der folgenden Abbildung ist die Client/Server-Architektur für die Webanwendung dargestellt:

[6] Das Internet ist ein Netzwerk von Netzwerken.
[7] Das Web ist ein Paradebeispiel für ein verteiltes System.
[8] 1 (RFC 7540) M. Belshe, R. Peon, M. Thomson, „Hypertext Transfer Protocol Version 2 - HTTP/2", RFC 7540, Mai 2015.
[9] https://de.wikipedia.org/wiki/Uniform_Resource_Locator

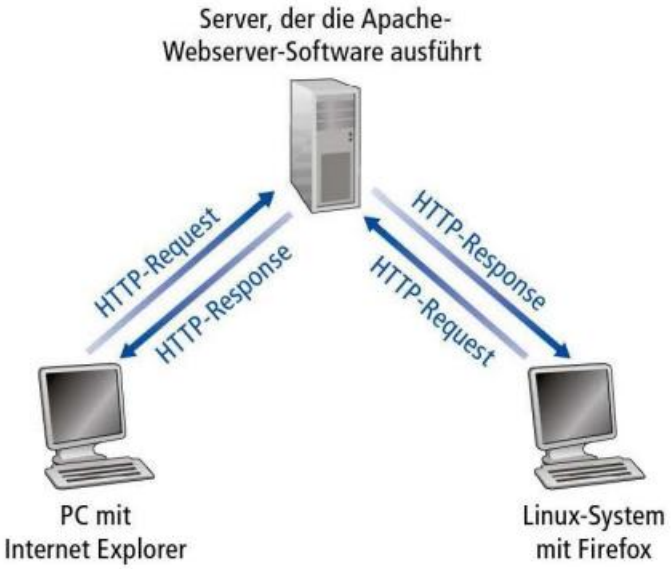

Abbildung 3: Request-Response-Verhalten von http

HTTP nutzt TCP als Transportprotokoll: Sobald die Verbindung hergestellt ist, tauschen der Browser und der Web-Server die Nachrichten über ihre Socket-Schnittstellen auf TCP-Ebene aus. Der Server sendet die angeforderten Dateien an den Client, ohne sich irgendwelche Informationen über den Zustand des Clients zu speichern. Das HTTP Protokoll wird daher als zustandslos bezeichnet, das heißt der Web-Server kann keinen Zusammenhang zwischen einzelnen Anfragen herstellen. Jede Anfrage ist somit eigenständig und unabhängig von vorangegangenen Anfragen.

2.1. HTTP-Nachrichtenformat

Es gibt zwei Arten von HTTP-Nachrichten, Request- und Response-Nachrichten. Beide werden im Folgenden besprochen:

2.1.1. HTTP-Request

Nachfolgend sehen Sie eine HTTP-Request-Nachricht:

```
GET /data/index.html HTTP/1.1
Host: localhost:6789
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux armv7l)
   AppleWebKit/537.36 (KHTML, like Gecko) Raspbian
   Chromium/72.0.3626.121 Safari/537.36
Accept: text/html,application/xhtml+xml,
   application/xml;q=0.9,image/webp,image/apng,*/*
Accept-Encoding: gzip, deflate, br
Accept-Language: de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7
```

Die erste Zeile einer HTTP-Request-Nachricht wird Request-Zeile (Anforderungszeile) genannt. Die anschliessenden Zeilen bezeichnet man als Header-Zeilen (Kopfzeilen). Die Request-Zeile hat drei Felder: das Methodenfeld, das URL-Feld und das Versions-Feld. Das Methodenfeld kann mehrere verschiedene Werte annehmen, darunter GET, POST, HEAD, PUT und DELETE. Die grosse Mehrheit

der HTTP-Request-Nachrichten verwendet die GET-Methode. GET wird benötigt, wenn der Browser ein Objekt anfordert, wobei das angeforderte Objekt im URL-Feld identifiziert wird. In diesem Beispiel fordert der Browser das Objekt /data/index.html an. Die Version ist selbsterklärend: In diesem Beispiel legt der Browser die Version HTTP/1.1 fest.

Die Header-Zeile Host: localhost:6789 legt den Host fest, auf dem sich das Objekt befindet. Mit der Header-Zeile Connection: Keep-Alive teilt der Browser dem Server mit, dass die Verbindung aufrecht gehalten werden soll. Die Header-Zeile User-Agent: ist die Browserkennung. Im Beispiel ist dies der Chromium. Das Feld Accept: signalisiert dem Server, welche Formate der Client akzeptiert. Die Header-Zeile Accept-Encoding: zeigt, welche komprimierten Formate der Client unterstützt. Schliesslich definiert die Header-Zeile Accept-Language, dass der Benutzer eine deutsche Version des Objekts wünscht, sofern ein solches Objekt auf dem Server existiert. Ist eine solche Version nicht verfügbar, dann soll der Server die Standardversion senden.

Abbildung 4 stellt das allgemeine Format einer Request-Nachricht dar und zeigt, neben der Request-Zeile und den Header-Zeilen, eine Leerzeile sowie einen Entity-Body (Formularfeld). Beinhaltet das Methodenfeld ein GET (siehe obiges Beispiel), dann ist das Formularfeld leer. Wird hingegen vom Benutzer eine Webseite angefordert, deren spezifischen Inhalt vom Eintrag im Formularfeld abhängt, dann wird die POST Methode angewendet.

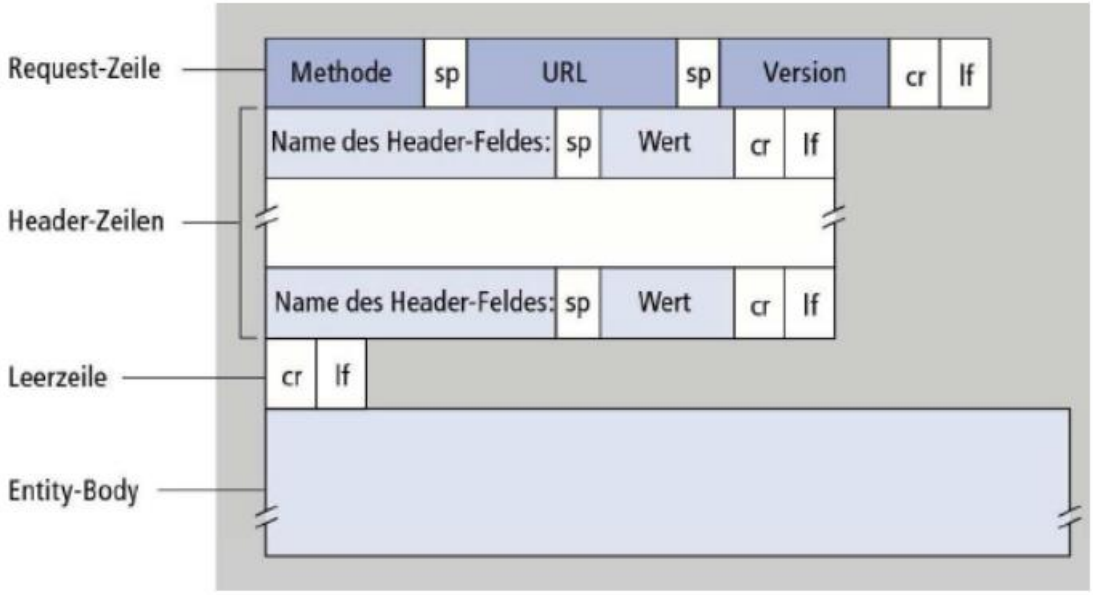

Abbildung 4: Allgemeines Format einer HTTP-Request-Nachricht

Die HEAD-Methode ist dem GET ähnlich: Erhält ein Server eine Anfrage mit HEAD, so antwortet dieser mit einer HTTP-Nachricht, lässt aber das angeforderte Objekt aus. Anwendungsentwickler verwenden die HEAD-Methode zum Debuggen. Die PUT Methode wird häufig in Verbindung mit Werkzeugen zum Veröffentlichen im Web benutzt. Sie erlaubt es einem Benutzer ein Objekt auf einen bestimmten Pfad auf einem Web-Server hochzuladen. Die PUT-Methode wird auch von anderen Anwendungen verwendet die Objekte auf Web-Server hochladen müssen. Die DELETE-Methode ermöglicht einem Benutzer oder einer Anwendung das Löschen eines Objektes auf einem Web-Server.

2.1.2. HTTP-Response

Nachfolgend sehen Sie eine typische HTTP-Response-Nachricht:

```
HTTP/1.1 200 OK
Content-Length: 54
Content-Type: text/html

<html><body><p>Hell World</p></body></html>
```

Die erste Zeile wird Statuszeile bezeichnet. In dieser Zeile gibt der Server die http-Version zurück sowie einen Response-Statuscode in der Form von drei Ziffern und eine kurze Textnachricht (Statusnachricht). Beispielsweise «200 OK» bedeutet «Anfrage war erfolgreich» oder «404 Not Found» wird als «angeforderte Ressource existiert nicht» interpretiert. Die Response-Nachricht kann ebenfalls Headerzeilen enthalten, von denen jede aus einem Wertepaar (Name des Headerfeldes: und Wert) besteht. Beispielsweise bedeutet «Content-Length:» die Länge der übertragenen Daten. Am Ende der Headerzeilen steht genau eine Leerzeile. Danach überträgt der Server die eigentlichen Informationen (Entity-Body), wie beispielsweise den HTML Code. Die folgende Abbildung stellt das allgemeine Format einer Response-Nachricht dar.

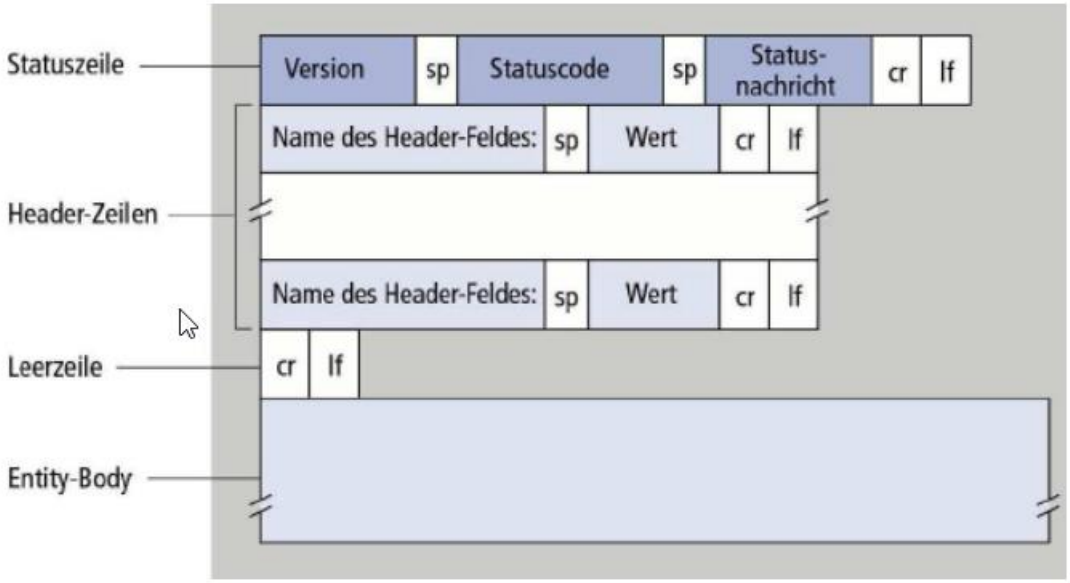

Abbildung 5: Allgemeines Format einer HTTP-Response-Nachricht

2.1.3. Darstellung von HTTP-Request und HTTP-Response

HTTP-Request und HTTP-Response können mit den im Browser eingebauten Entwicklertools visualisiert werden. Im Google Chrome Browser beispielsweise kann man sich mit der Funktionstaste F12 eine Ansicht (siehe Abbildung 6: Google Chrome Visulisierung von HTTP-Request und HTTP-Response) geben lassen, welche den Inhalt der Request- und Response-Nachrichten darstellt. Auch

mit Wireshark[10], einer freien Software zur Analyse und grafischen Aufbereitung von Datenprotokollen, kann man Request und Request visualisieren.

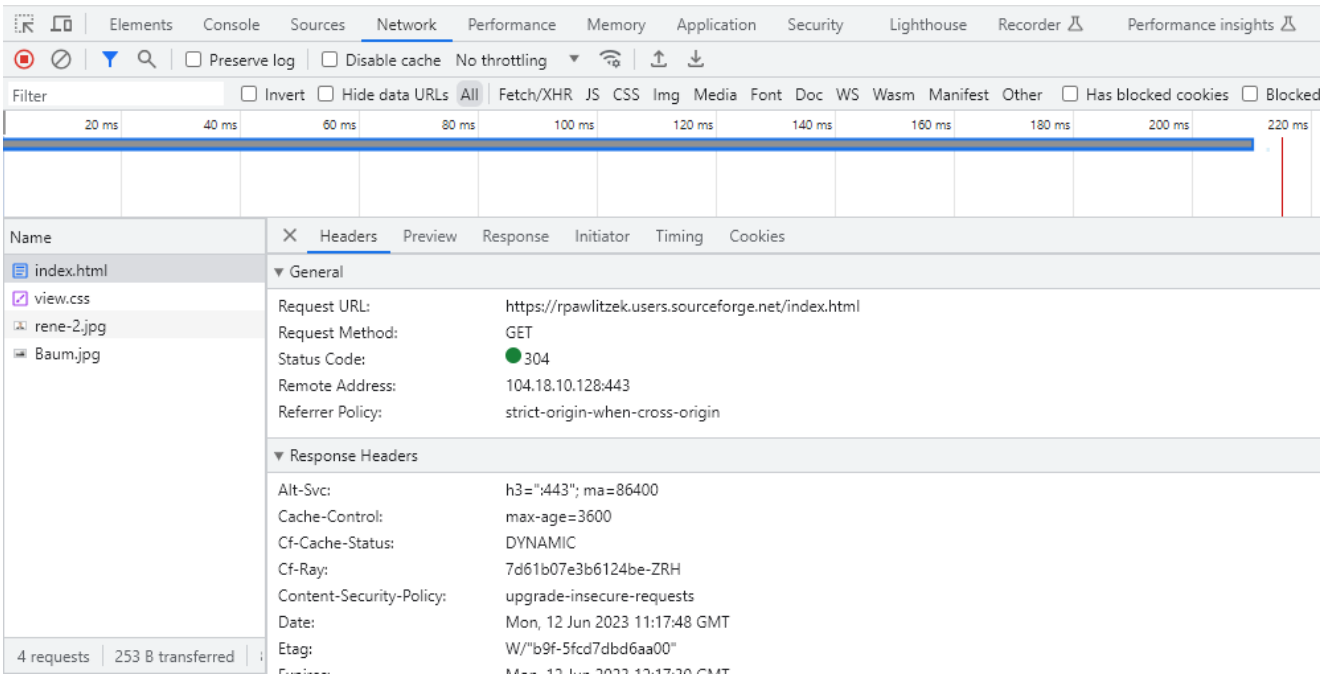

Abbildung 6: Google Chrome Visulisierung von HTTP-Request und HTTP-Response

2.3. REST

Schnittstellen zur Cloud oder zu einem Server sind meist RESTful. REST steht dabei für Representational State Transfer und ist ein Programmierparadigma, das von Roy Fielding im Jahr 2000 geprägt wurde. Eine Reihe von Prinzipen muss für vollständiges REST erfüllt sein, u.a. eine einheitliche Schnittstelle. Die Umsetzung von REST erfolgt fast ausschliesslich mit http bzw. https. Die Methoden von http werden dabei wie folgt eingesetzt:

Methode	Verwendungszweck
POST	Create: Erzeugung einer Ressource
GET	Read: Zugriff auf eine Ressource
UPDATE	Update: Aktualisierung einer Ressource
DELETE	Delete: Löschen einer Ressource

Nachfolgend wird erläutert wie eine REST-Schnittstelle mit CRUD (Create, Read, Update und Delete) implementiert werden kann: Ein RESTful WebService kann die Operationen Create, Read, Update und Delete (CRUD) mit Hilfe der http Methoden GET, POST, PUT und DELETE implementieren:

CREATE	POST http://reststore.com/orders	Bestellung aufgeben
READ	GET http://reststore.com/orders/233	Bestellung abfragen
UPDATE	PUT http://reststore.com/orders/233	Bestellung ändern
DELETE	DELETE http://reststore.com/orders/233	Bestellung stornieren

[10] https://de.wikipedia.org/wiki/Wireshark

2.4. Fallstudie: Cloudanbindung

Nachfolgend wird der Java-Code gezeigt, um Daten mit http und der GET Methode an ein Servlet in der Google Cloud zu übermitteln:

```
/* ------------------
     TempCloud.java
   ------------------ */


/*  Query current CPU temperature of the Raspberry Pi and send data to the cloud.  */


/*  Visualization: http://iot4ntb.appspot.com/sensorgraph.jsp?sensor=test  */


/*  IoT Workshop 2017, Prof. Rene Pawlitzek, NTB  */


import com.pi4j.system.SystemInfo;


import java.io.PrintWriter;
import java.net.Socket;
import java.util.Date;


public class TempCloud {


  private final static int delay = 60 * 1000;  // 1 min.

  private static int port = 80;
  private static String page = "/add";
  private static String host = "iot4ntb.appspot.com";


  private void uploadTemperature (String temp) throws Exception {

        // create a TCP socket
        Socket clientSocket = new Socket (host, port);

        // send the request for the page
        PrintWriter out = new PrintWriter (clientSocket.getOutputStream ());
        String parameters = "sensor=test&value=" + temp;
        String request = "GET " + page + "?" + parameters + " HTTP/1.1\r\nHost: " +
           host + "\r\n\r\n";  // don't forget host!!!
        System.out.println (request);
        out.print (request);
        out.flush ();
        out.close ();

        // close socket
        clientSocket.close ();

  } // uploadTemperature



  public void run () throws Exception {
        while (true) {

          // show timestamp
          System.out.println (new Date().toString ());

      // read CPU temperature
      float temp = SystemInfo.getCpuTemperature();
      String tempStr = "" + temp;

      // upload temperature to the cloud
```

```
      try {
        System.out.println ("Uploading temperature: " + tempStr);
        uploadTemperature (tempStr);
      } catch (Exception e) {
        System.out.println ("Uploading failed");
        e.printStackTrace ();
      } // try

      // sleep
      Thread.sleep (delay);

       } // while
  } // run


  public static void main(String[] args) throws Exception {
    // check if a command line arguments for the hostname and port are specified
    if (args.length >= 2) {
      try {
        host = args[0];
        port = Integer.parseInt (args[1]);
      } catch (Exception e) {
        System.out.println ("Invalid port: " + args[1]);
      } // try
    } // if
    System.out.println ("Host: " + host);
    System.out.println ("Port: " + port);
    System.out.println ();
       TempCloud tc = new TempCloud ();
       tc.run ();
  } // main


} // TempCloud


/* ----- End of File ----- */
```

Der Code übermittelt in regelmässigen Zeitabständen (delay) mit dem Aufruf der Methode uploadTemperature (tempStr); eine Zeichenkette, welche die CPU Temperatur des Raspberry Pi enthält in die Google Cloud. Die Daten werden von einem Servlet in der Google Cloud entgegengenommen und gespeichert. Die eigentliche Übertragung der Daten geschieht in der Methode uploadTemperature (String temp) und ist unverschlüsselt. Es wird eine Verbindung zum Zielhost geöffnet mit new Socket (host, port). Anschliessend wird der http-Request mit Hilfe eines PrintWriters konstruiert. Es handelt sich um einen GET-Request. Die Daten (parameters) werden mit dem Request übermittelt. Zum Schluss wird der PrintWriter und der Socket geschlossen.[11]

Nachfolgend wird der Java Code gezeigt, welcher die Daten in der Cloud entgegennimmt und in einer NonSQL-Datenbank (Google Datastore) abspeichert. Dieser Code läuft im Google App Engine (GAE). Es handelt sich um ein Java Servlet.

```
/* ------------------------------
     GoogleAppsSensorServlet.java
   ------------------------------ */


/*  Cloud Computing 2016-2017, Prof. Rene Pawlitzek, NTB  */
```

[11] Für die Übertragung der Daten könnten auch die Klassen URL und HttpURLConnection verwendet werden.

```
package ch.ntb.iot;


import java.io.IOException;
import java.util.Date;

import javax.servlet.http.*;

import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.datastore.KeyFactory;


@SuppressWarnings("serial")
public class GoogleAppsSensorServlet extends HttpServlet {


public void doGet(HttpServletRequest req, HttpServletResponse resp)throws IOException {

        String SensorName = req.getParameter("sensor");
         String value = req.getParameter("value");
         Date date = new Date();

         System.out.println("Sensor name="+SensorName+" value="+value+" date:"+date);

         Key SensorDataKey = KeyFactory.createKey("SensorData", SensorName);
         Entity sensorData = new Entity("Sensor", SensorDataKey);
         sensorData.setProperty("date", date);
         sensorData.setProperty("value", value);

         DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
         datastore.put(sensorData);

         resp.sendRedirect("/sensorgraph.jsp?sensor="+ SensorName);

} // doGet

} // GoogleAppsSensorServlet



/* ----- End of File ----- */
```

2.5. Zusammenfassung

HTTP ist ein einfaches textbasiertes Protokoll, welches für das World Wide Web (WWW) entworfen wurde. Es dient dem Datenaustausch zwischen Webbrowser (Client) und Web-Server. Ein Web-Client sendet Request-Nachrichten an einen WebServer, der auf diese Nachrichten mit Response-Nachrichten antwortet. Diese Response-Nachrichten beinhalten in der Regel den HTML Code für eine Webseite, die der Webbrowser visualisiert. HTTP eignet sich auch für das Internet of Things (IoT), trotz der relativ grossen Header in den Request- und Response Nachrichten.

3. MQTT

Das Message Queuing Telemetry Transport Protokoll (MQTT) ist ein offenes Netzwerkprotokoll, welches das Übertragen von Daten in Form von Nachrichten ermöglicht. Es implementiert das Publish/Subscribe-Muster und ist im Gegensatz zur Request/Response-Architektur ereignisgesteuert: Dabei wird die Punkt-zu-Punkt-Verbindung durch einen zentralen Server (Broker) ersetzt, zu dem sich Datenproduzenten (Producer) und -nutzer (Consumer) gleichermassen verbinden können. Das Senden (Publish) und Empfangen (Subscribe) von Nachrichten funktioniert über sogenannte Topics.

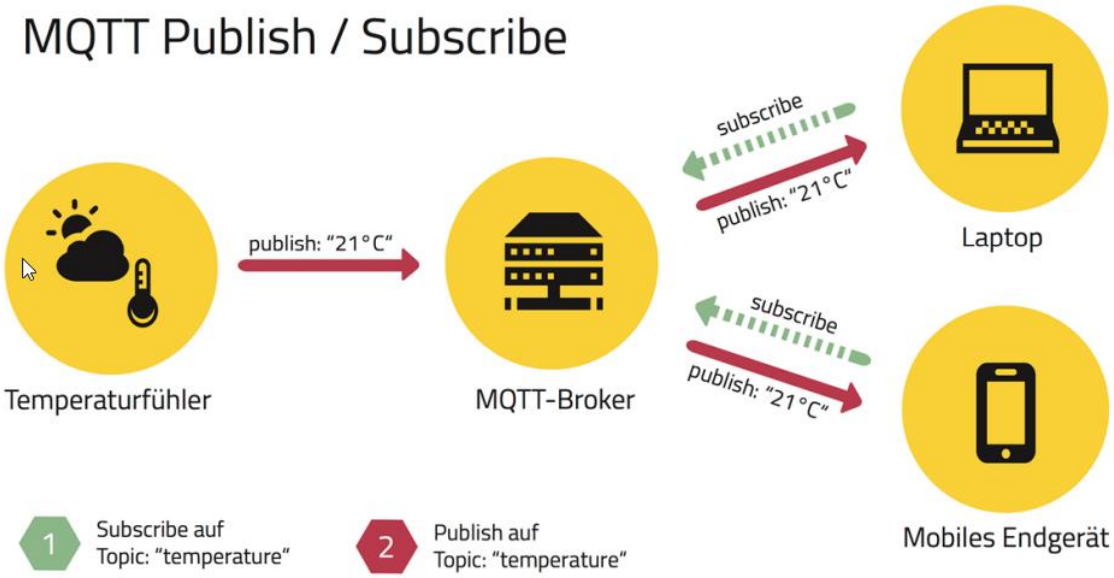

Abbildung 7: MQTT Publish/Subscribe, Quelle: https://www.hivemq.com/

Abbildung 7 zeigt beispielsweise ein Temperaturfühler, der seine Messungen veröffentlicht. Der MQTT-Broker überprüft nun, welche Interessenten ihrerseits einen Kanal zum Empfangen dieser Daten geöffnet haben und leitet die Nachrichten an diese weiter.

3.3. MQTT-Broker

Der MQTT-Broker hat die Rolle eines Vermittlers. Er empfängt alle publizierten Nachrichten und ist dafür zuständig, die abonnierten Clients zu kennen, die Nachrichten zu filtern und jede Nachricht an die interessierten Clients weiterzuleiten. Zudem ist er verantwortlich, die Clients zu authentifizieren und zu autorisieren.

Es gibt MQTT-Broker (z. B. Mosquitto™ oder HiveMQ), die auf einem eigenen Rechner installiert und über die lokale (private) IP-Adresse aufgerufen werden können. Neben der lokalen Installation gibt es auch öffentliche Broker, die über eine (öffentliche) Internetadresse angesprochen werden (z. B. test.mosquitto.org oder broker.hivemq.com).

3.4. Client

Ein Client hat die Möglichkeit Daten zu einem bestimmten Thema (engl. Topics) zu veröffentlichen (engl. publish, siehe Temperaturfühler in Abbildung 7) oder sich bestimmte Topics zu abonnieren (engl. subscribe, siehe mobiles Endgerät oder Laptop in Abbildung 1). Hat ein Client ein Topic abonniert, so werden vom MQTT-Broker empfangene Nachrichten an diesen Client weitergeleitet, sobald ein anderer Client eine Nachricht mit diesem Topic veröffentlicht. Die Kommunikation zwischen verschiedenen Clients findet also immer über einen MQTT-Broker statt. Ein Client kann gleichzeitig Nachrichten veröffentlichen und Nachrichten empfangen.

Für die Implementierung eines Clients wird beispielsweise die Eclipse Paho MQTT Bibliothek bereitgestellt. Ein Testclient von HiveMQ steht online unter http://www.hivemq.com/demos/websocket-client/ zur Verfügung.

3.5. Topics

Der Nachrichtenaustausch zwischen Client und Broker erfolgt über selbstgewählte Topics und ist folgendermassen aufgebaut:

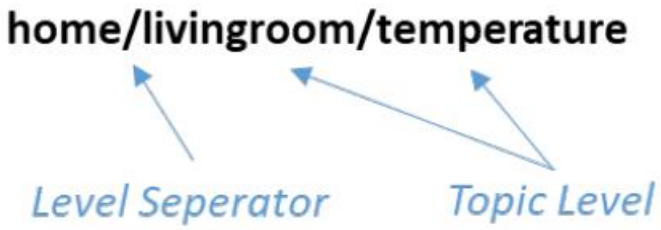

Abbildung 8: Zusammensetzung eines Topics

Ein Topic ist eine Zeichenkette, die eine Art Betreff der Nachricht bedeutet und ähnlich einer URL aufgebaut ist. Dieser setzt sich aus einer beliebigen Anzahl von Levels zusammen, die jeweils durch Separatoren (/) getrennt sind. Die Nachrichten die ein Client publiziert bestehen aus einem Topic und dem Payload, dem Inhalt der Nachricht, beispielsweise die Temperatur des Sensors.

3.6. Dienstgüte – Quality of Service (QoS)

Die Dienstgüte ist die Vereinbarung von Sender um Empfänger, die festlegt wie verbindlich die Nachrichten beim Empfänger ankommen. Es gibt drei verschiedene QoS-Level, mit denen die Nachrichten zugestellt werden sollen:

- Eine Nachricht wird höchstens einmal ausgeliefert. (QoS Level 0)
- Eine Nachricht wird mindestens einmal ausgeliefert. (QoS Level 1)
- Eine Nachricht wird genau einmal ausgeliefert. (QoS Level 2)

3.7. Fallstudie: Datenproduzent (Publisher) und Datenkonsument (Subscriber)

Nachfolgend wird ein in Java geschriebener Datenproduzent und Datenkonsument gezeigt. Der Producer verwendet die Methode client.publish() für das Publizieren der Nachrichten.

```
/* -----------------
     Producer.java
   ----------------- */


/*  IoT Workshop 2017, Prof. Rene Pawlitzek, NTB  */


/*  Note: This example requires the mosquitto MQTT broker and the Paho MQTT library. */
```

```
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;


public class Producer {


    // private static String broker   = "tcp://localhost:1883";
    private static String broker   = "tcp://inf01-server.ntb.ch:1883";
    private static String clientID = "NTB-Producer";
    private static String topic    = "home/livingroom/temp";
    private static int    QoS      = 2;    // quality of service


    private  MqttClient  client;


    public Producer () throws Exception {
        client = new MqttClient (broker, clientID, new MemoryPersistence());
        client.connect ();
        System.out.println ("Producer is connected");
    } // Producer


    public void run () throws Exception {
        int i = 0;
        while (true) {
            int temp = 20 + i;
            i = (i + 1) % 10;
            String data = "Temperature: " + temp + " [C]";
            client.publish (
                topic,                      //  topic
                data.getBytes("UTF-8"),     //  message
                QoS,                        //  QoS level
                true);                      //  retained message
            System.out.println (data);
            Thread.sleep (5000);
        } // while
    } // run


    public static void main (String[] args) {
        try {
            Producer producer = new Producer ();
            producer.run ();
        } catch (Exception e) {
            e.printStackTrace ();
        } // try
    } // main


} // Producer


/* ----- End of File ------ */
```

Der Subscriber nutzt den MqttCallback für den Empfang von neuen Nachrichten vom Broker. Die Methode messageArrived wird immer dann aufgerufen, wenn der Broker Daten an den Subscriber sendet.

```
/* -----------------
     Consumer.java
   ----------------- */


/*  IoT Workshop 2017, Prof. Rene Pawlitzek, NTB  */
```

```
/*  Note: This example requires the mosquitto MQTT broker and the Paho MQTT library. */


import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;


public class Consumer implements MqttCallback {

    // private static String broker   = "tcp://localhost:1883";
    private static String broker   = "tcp://inf01-server.ntb.ch:1883";
    private static String clientID = "NTB-Consumer";
    private static String topic    = "home/livingroom/temp";
    private static int    QoS      = 2;   // quality of service


    private MqttClient client;


    public Consumer () throws Exception {
        client = new MqttClient (broker, clientID, new MemoryPersistence());
        client.connect ();
        System.out.println ("Consumer is connected");
    } // Consumer


    public void run () throws Exception {
        client.setCallback (this);
        client.subscribe (topic, QoS);
    } // run


    /* ----- Implementation of MqttCallback ----- */


    public void connectionLost (Throwable arg) {
        System.out.println ("Connection lost");
    } // connectionLost


    public void deliveryComplete (IMqttDeliveryToken arg) {
        System.out.println ("Delivery complete");
    } // deliveryComplete


    public void messageArrived (String arg0, MqttMessage arg1) throws Exception {
        String msg = new String (arg1.getPayload ());
        System.out.println (msg);
    } // messageArrived


    public static void main (String[] args) {
        try {
            Consumer consumer = new Consumer ();
            consumer.run ();
        } catch (Exception e) {
            e.printStackTrace ();
        } // try
    } // main


} // Consumer


/* ----- End of File ----- */
```

Eine Cloudanbindung kann, neben HTTP/REST (siehe Kapitel 2) und CoAP (siehe Kapitel 4), auch mit MQTT realisiert werden, da Cloud Umgebungen in der Regel einen MQTT-Broker beinhalten (siehe Abbildung 9).

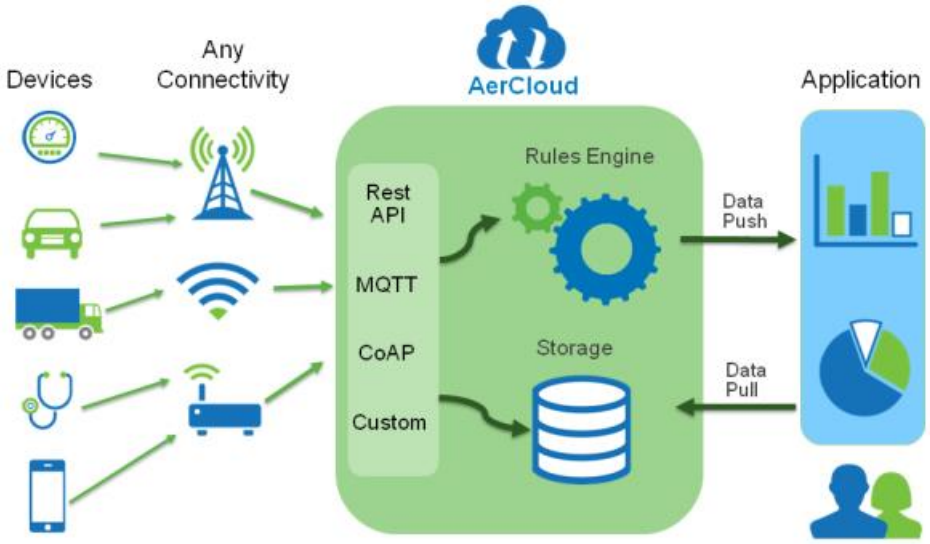

Abbildung 9: Source: http://www.aeris.com

3.8. Zusammenfassung

Eine Vielzahl von IoT-Anwendungen verwendet das Kommunikationsprotokoll MQTT. Dabei handelt es sich um ein Publish/Subscribe-Protokoll, das die zuverlässige Vernetzung von Anwendungen und Geräten ermöglicht.

Alle Anwendungen und Geräte werden als MQTT-Client behandelt und können über einen MQTT-Broker bidirektional miteinander kommunizieren. Clients können sowohl Nachrichten senden als auch empfangen. Das Senden wird als Publish und das Empfangen als Subscribe bezeichnet. Dabei werden Nachrichten mit Topics veröffentlicht. Jene Clients, die zuvor Topics beim MQTT-Broker abonniert (engl. subscribe) haben, empfangen die entsprechenden Nachrichten. Der MQTT-Broker verteilt so die Nachrichten an die interessierten Clients.

Eine MQTT-Anwendung lässt sich sowohl in Java als auch in JavaScript einfach mit der Paho[12] Bibliothek implementieren. Der MQTT-Broker kann öffentlich sein oder durch Installation auf einem eigenen Rechner laufen.

4. CoAP

Constrained Application Protocol (CoAP) ist ein Application Layer Protocol für eingeschränkte Geräte. Es ermöglicht Geräten die Kommunikation mit dem Internet. CoAP bietet sehr geringen Overhead und Einfachheit, was für IoT-Geräte wichtig ist.

[12] https://www.eclipse.org/paho/

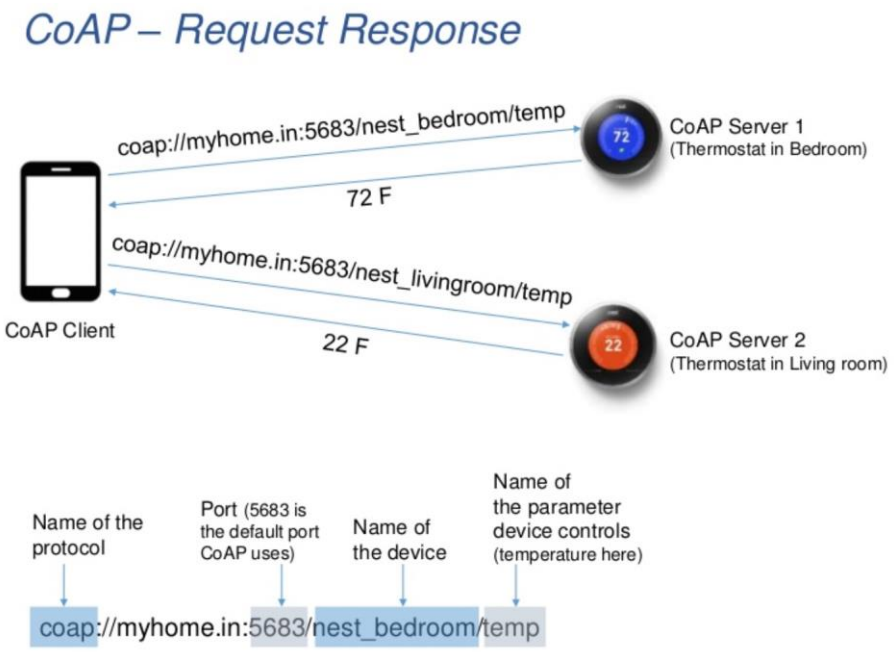

Abbildung 10: CoAP Request Response

Solche Geräte sind in der Regel tief eingebettet und haben wenig Speicher, wenig Rechenleistung und nur begrenzte Stromversorgung im Gegensatz zu herkömmlichen Internetgeräten. CoAP kann auf den meisten Geräten ausgeführt werden, die UDP unterstützen, da CoAP UDP als Transportprotokoll verwendet (siehe Abbildung 11: HTTP, MQTT und CoAP Protokollstack). CoAP ähnelt HTTP/REST, was es vertraut und leicht verständlich macht.

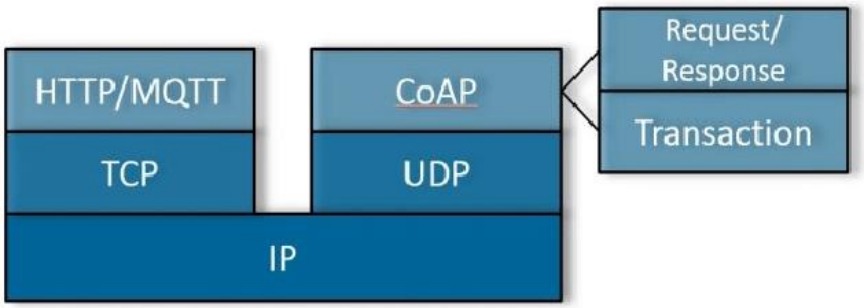

Abbildung 11: HTTP, MQTT und CoAP Protokollstack

Nachfolgend wird gezeigt, wie eine Kommunikation zwischen einem CoAP-Client und einem CoAP Server in Java realisiert wird. Es werden vier Operationen implementiert (Create, Read, Update und Delete)[13], um die Ressource secret zu manipulieren. Mit POST wird ein Geheimnis gespeichert (create), mit GET wird das Geheimnis abgefragt (read), mit PUT wird das Geheimnis verändert (update), und mit DELETE wird das Geheimnis gelöscht (delete).

```
/* --------------------------
     CoAPResourceSecret.java
   -------------------------- */


/*  Computerkommunikation 2018/2022, Prof. Rene Pawlitzek, NTB  */
```

[13] Hier spricht man von CRUD (Create, Read, Update und Delete).

```
package org.eclipse.californium.examples;


import static org.eclipse.californium.core.coap.CoAP.ResponseCode.*;

import java.io.ByteArrayInputStream;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import com.google.gson.Gson;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.eclipse.californium.core.CoapResource;
import org.eclipse.californium.core.CoapServer;
import org.eclipse.californium.core.coap.MediaTypeRegistry;
import org.eclipse.californium.core.coap.OptionSet;
import org.eclipse.californium.core.server.resources.CoapExchange;



public class CoapCRUDServer extends CoapResource {


        private  String                  topSecret = null;
        private  DocumentBuilder         builder;
        private  DocumentBuilderFactory  factory;


        public CoapCRUDServer (String name) {
                super (name);
                try {
                  factory = DocumentBuilderFactory.newInstance ();
                  builder = factory.newDocumentBuilder ();
                } catch (Exception e) {
                  e.printStackTrace ();
                } // try
                System.out.println ("Resource /" + name + " added");
        } // CoapResourceSecret


        // Create = POST
        @Override
        public void handlePOST (CoapExchange exchange) {
                System.out.println ("POST: " + exchange.getRequestText());
                exchange.accept();
                OptionSet options = exchange.getRequestOptions();
                if (options.isContentFormat (MediaTypeRegistry.TEXT_XML)) {
                        // <?xml version=\"1.0\"?><secret>4711</secret>
                        String xml = exchange.getRequestText ();
                        try {
                                // parse XML
                           StringBuilder xmlStringBuilder = new StringBuilder ();
                           xmlStringBuilder.append (xml);
                           ByteArrayInputStream input =
                              new ByteArrayInputStream (
                                xmlStringBuilder.toString().getBytes ("UTF-8"));
                           Document doc = builder.parse (input);
                           NodeList nodes = doc.getElementsByTagName ("secret");
                           Node node = nodes.item (0);
                           if (node.getNodeType () == Node.ELEMENT_NODE) {
                                   Element elem = (Element) node;
                             topSecret = elem.getTextContent ();
                             exchange.respond (CREATED, "XML parsed successfully");
                           } else {
                                   exchange.respond (BAD_REQUEST, "Unable to parse XML");
                           } // if
                        } catch (Exception e) {
                       e.printStackTrace();
                           exchange.respond (INTERNAL_SERVER_ERROR);
```

```
                        } // if
                } else if (options.isContentFormat(MediaTypeRegistry.APPLICATION_JSON)) {
                        //  "{ \"secret\": \"1234\" }"
                        // parse JSON
                        Gson gson = new Gson ();
                        String json = exchange.getRequestText ();
                        Secret secret = gson.fromJson (json, Secret.class);
                        topSecret = secret.secret;
                        exchange.respond (CREATED);
                } else {
                        topSecret = exchange.getRequestText();
                        exchange.respond (CREATED);
                } // if
        } // handlePOST


        // Read = GET
        @Override
        public void handleGET (CoapExchange exchange) {
                System.out.println ("GET: " + exchange.getRequestText());
                exchange.respond (topSecret);
        } // handleGET


        // Update = PUT
        @Override
        public void handlePUT (CoapExchange exchange) {
                System.out.println ("PUT: " + exchange.getRequestText());
                topSecret = exchange.getRequestText();
                exchange.respond (CHANGED);
        } // handlePUT


        // Delete = DELETE
        @Override
        public void handleDELETE (CoapExchange exchange) {
                System.out.println ("DELETE: " + exchange.getRequestText());
                topSecret = "";
                // delete();
                exchange.respond (DELETED);
        } // handleDELETE


        public static void main(String[] args) {
                System.out.println ("Creating CoAP server ...");
                CoapServer server = new CoapServer();
                System.out.println ("CoAP server created.");
                System.out.println ("Adding resources ...");
                server.add (new CoapCRUDServer("secret"));
                System.out.println ("Resources added.");
                System.out.println ("Starting CoAP server ...");
                server.start ();
                System.out.println ("CoAP server started.");
        } // main


} // CoapCRUDServer


/* ----- CoapCRUDServer ----- */
```

```
/* ----------------------
/* ----------------------
     CoapCRUDClient.java
   ---------------------- */


/*  Computerkommunikation 2018/2022, Prof. Rene Pawlitzek, NTB  */

/*
```

```
        Creating CoAP client ...
        SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
        SLF4J: Defaulting to no-operation (NOP) logger implementation
        SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
        CoAP client created
        POST:
        Code: 2.01
        Options: {}
        GET:
        Code: 2.05
        Options: {"Content-Format":"text/plain"}
        Response text: Das ist ein Geheimnis
        PUT:
        Code: 2.04
        Options: {}
        GET:
        Code: 2.05
        Options: {"Content-Format":"text/plain"}
        Response text: Das ist ein anderes Geheimnis
        DELETE:
        Code: 2.02
        Options: {}
        GET:
        Code: 2.05
        Options: {"Content-Format":"text/plain"}
        POST:
        Code: 2.01
        Options: {"Content-Format":"text/plain"}
        Response text: XML parsed successfully
        GET:
        Code: 2.05
        Options: {"Content-Format":"text/plain"}
        Response text: 4711
        POST:
        Code: 2.01
        Options: {}
        GET:
        Code: 2.05
        Options: {"Content-Format":"text/plain"}
        Response text: 1234

*/



package org.eclipse.californium.examples;

import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.coap.MediaTypeRegistry;


public class CoapCRUDClient {

        private static void printResponse (String method, CoapResponse response) {
                System.out.println (method + ":");
                if (response != null) {
                        System.out.println ("Code: " +  response.getCode());
                        System.out.println ("Options: " + response.getOptions());
                        String text = response.getResponseText();
                        if (text != null && !text.isEmpty())
                          System.out.println ("Response text: " + text);
                } else {
                        System.out.println ("Request failed");
                } // if
        } // printResponse


        public static void main (String[] args) {

                CoapClient client;
                CoapResponse response;

                System.out.println ("Creating CoAP client ...");
                client = new CoapClient("coap://localhost/secret");
```

```
                client.useCONs ();
                System.out.println ("CoAP client created");

                // Create = POST
                response = client.post("Das ist ein Geheimnis",
                  MediaTypeRegistry.TEXT_PLAIN);
                printResponse ("POST", response);

                // Read = GET
                response = client.get();
                printResponse ("GET", response);

                // Update = PUT
                response = client.put("Das ist ein anderes Geheimnis",
                  MediaTypeRegistry.TEXT_PLAIN);
                printResponse ("PUT", response);

                // Read = GET
                response = client.get();
                printResponse ("GET", response);

                // Delete = DELETE
                response = client.delete();
                printResponse ("DELETE", response);

                // Read = GET
                response = client.get();
                printResponse ("GET", response);

                // Create = POST
                response = client.post("<?xml version=\"1.0\"?><secret>4711</secret>",
                  MediaTypeRegistry.TEXT_XML);
                printResponse ("POST", response);

                // Read = GET
                response = client.get();
                printResponse ("GET", response);

                // Create = POST
                response = client.post("{ \"secret\": \"1234\" }",
                  MediaTypeRegistry.APPLICATION_JSON);
                printResponse ("POST", response);

                // Read = GET
                response = client.get();
                printResponse ("GET", response);

        } // main


} // CoapCRUDClient


/* ----- End of File ----- */
```

```
/* ---------------
     Secret.java
   --------------- */

/*  Computerkommunikation 2018/2022, Prof. Rene Pawlitzek, NTB  */

package org.eclipse.californium.examples;

public class Secret {

        public String secret;

} // Secret


/* ----- End of File ----- */
```

4.1. Zusammenfassung

CoAP ist ein schlankes Netzwerkprotokoll für Geräte mit eingeschränkten Möglichkeiten. Es ist mit HTTP/REST vergleichbar, nutzt jedoch UDP anstelle von TCP als Transportprotokoll und benötigt weniger Ressourcen. CoAP bietet standardgemäss Sicherheit, was bei HTTP und MQTT mit HTTPS bzw. MQTTS der Fall ist. Für die Implementierung kann die Eclipse Californium[14] Bibliothek genutzt werden.

5. Diskussion

Mit diesem Wissensnugget wurden drei bekannte Netzwerkprotokolle kurz vorgestellt. Alle drei Protokolle eignen sich für die Vernetzung von Geräten bzw. eine Cloud-Anbindung:

- Für HTTP sind viele Bibliotheken und Beispiele im Internet zu finden. Sicherheit wird mit HTTPS realisiert. Obwohl HTTP für den Austausch von Daten zwischen Browser und Web-Server konzipiert wurde, eignet sich HTTP auch für das Internet der Dinge (IoT).
- MQTT ist das IoT Netzwerkprotokoll schlechthin. Mit Hilfe eines Brokers wird das Publish / Subscribe Muster implementiert. Es gibt daher keine Punkt-zu-Punkt Kommunikation zwischen zwei Computern. Wie HTTPS, bietet MQTTS Sicherheit. Die Eclipse Paho Bibliothek ist in vielen Sprachen verfügbar, sodass schnell Lösungen realisiert werden können.
- CoAP ist ein Protokoll zur Vernetzung von einfachen Geräten mit beschränkten Möglichkeiten. Sicherheit ist bereits eingebaut. CoAP verhält sich (fast) wie HTTP/REST.

[14] https://www.eclipse.org/californium/