Heinrich C. Mayr, Martin Pinzger (Hrsg.): INFORMATIK 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 1851

xHealth: Eine MQTT und REST basierte Architektur zum Zugriff auf Sensordaten smarter Objekte

Manuel Guttenberger[1], Klemens Waldhör[2]

Abstract: Smarte Objekte wie Smartwatches oder Wearables verfügen über Gyro- und Beschleunigungssensoren zur Erfassung der Umwelt. Zur Erkennung von Mustern in Bewegungsdaten mittels Data Mining sind für viele Anwendungsfälle die Sensordaten zunächst an einen zentralen Service zu übermitteln, um komplexe Berechnungen wie Klassifikation durchzuführen. Im Rahmen dieser Arbeit wird ein Konzept, Anwendungsfälle sowie ein Prototyp für das System xHealth vorgestellt. xHealth empfängt Sensordaten über das MQTT-Protokoll sowie einer RESTful API basierend auf HTTP, speichert diese und stellt diese für weitere Analysen aus dem Bereich Big Data zur Verfügung.

Keywords: Healthcare, Digital, Transformation, Internet of Things, Web of Things, M2M, Sensordaten, Cloud, Architektur, REST, HTTP, MQTT, API, ADL/EDL Erkennung

1 Einleitung

Gartner prognostiziert für das Jahr 2020 mehr als 25 Mrd. Geräte im „Internet of Things“ (IoT) [Pe16]. Insbesondere Wearables und Smartwatches werden durch diese Entwicklung an Bedeutung gewinnen. Diese smarten Objekte schaffen die technische Basis für den Service von xHealth. Zur Analyse von Sensordaten smarter Objekte über einen zentralen Service mittels Data Mining werden geeignete Protokolle benötigt. Die Vielfalt an verfügbaren Protokollen im IoT ist ebenso groß wie die Bandbreite unterschiedlicher Geräte, welche das IoT mangels Interoperabilität vom „Internet of Things“ zu einem „Intranet of Things“ mutieren lässt. Es fehlt ein universelles Protokoll, wie es etwa im Web mit dem HTTP-Protokoll existiert. Hier setzt die Idee des Web of Things (WoT) an, in welchem etablierte Webtechnologien wie HTTP und RESTful APIs für den Datenaustausch zwischen smarten Objekten verwendet werden [GT15]. Ein populäres Protokoll im IoT stellt MQTT (Message Queue Telemetry Transport) dar [W316]. MQTT ist ein effizientes und schlankes Protokoll, welches zur Kommunikation zwischen smarten Objekten entwickelt wurde. Je nach Sensor werden mehrere hundert Messungen pro Sekunde generiert, meist in mehreren Achsen (x,y,z) sowie einem dazu passenden Zeitstempel, ggf. weitere Zusatzinformationen. Werden nur Gyrometer- und Beschleunigungsdaten mit jeweils drei Achsen verwendet, wird pro Sekunde ein Datendurchsatz von mindestens 5000 – 6000 Byte (600 – 1000 Sensordaten) erzeugt. Wichtig

[1] FOM Hochschule für Oekonomie und Management, Hochschulstudienzentrum Nürnberg
City Park Center, Zeltnerstr. 19, 90443 Nürnberg, manuel.guttenberger@mgu.io
[2] FOM Hochschule für Oekonomie und Management, Hochschulstudienzentrum Nürnberg
City Park Center, Zeltnerstr. 19, 90443 Nürnberg, klemens.waldhoer@fom.de

ist zu berücksichtigen, dass diese Sensordaten asynchron anfallen, daher bei den meisten Geräten mittels Callback-Funktionen des jeweiligen Sensors realisiert sind und übertragen werden sollen. Für Geräte wie Wearables und Smartwatches stellt dies eine hohe Hardwareauslastung dar, insbesondere da diese Geräte meist noch weitere Aktionen parallel durchführen, über eigenes Energiemanagement zur Akkulaufzeitoptimierung verfügen, das die Geräte wann immer möglich in einen Schlafzustand versetzt, und gleichzeitig nicht über die Hardwareausstattung eines PC oder auch eines Mobiltelefons verfügen. Eine effiziente und schlanke Übertragung der Daten stellt damit einen entscheidenden Faktor bei der Implementierung dar.

2 Forschungsfragen und Methodik

Eine erste und naheliegende Lösung wäre, Sensordaten mittels HTTP (neben Alternativen wie Bluetooth in der Kommunikation zwischen Smartwatches und Mobiltelefonen) an einen entsprechenden Webserver zu übertragen. HTTP Requests erzeugen aber einen entsprechenden Overhead an übermittelter Information, was zu einer entsprechenden Belastung der Verbindung und beteiligten Anwendungen führt.

Die zentrale Forschungsarbeit dieser Arbeit ist damit:

- Können mittels MQTT Sensordaten effizienter übermittelt werden als mit HTTP?
- Oder anders formuliert: Ist MQTT das Protokoll der Wahl bei der Übertragung von Sensordaten?

Im Rahmen dieser Arbeit wird ein Konzept für das System xHealth vorgestellt und mittels Prototyping zwei Szenarien exemplarisch implementiert. Im ersten Szenario wird eine xHealth basierte Wetterstation implementiert. Im zweiten Anwendungsfall wird die ADL/EDL-Erkennung auf Smartwatches auf ihre Tauglichkeit in Bezug auf MQTT untersucht. Zur Beantwortung der Forschungsfrage wird der Datendurchsatz von HTTP und MQTT mit einem Arduino Uno Mikrocontroller untersucht.

3 Das MQTT Protokoll

3.1 Entwicklung von MQTT

MQTT wurde im Jahr 1999 von Stanford-Clark (IBM) und Nipper (Eurotech) zur Überwachung von Pipelines in der Wüste entwickelt. Zur Datenübertragung wurden teure Satellitenverbindungen verwendet, weshalb die Effizienz des Protokolls das vordergründige Ziel bei der Entwicklung darstellte [HI16].

Beflügelt wurde das Interesse an MQTT durch das 2011 gestartete Projekt Eclipse Paho, welches u.a. Implementierungen für C, Java, JavaScript, .Net und Android Services bereitstellt [EC16b]. Seit 2014 ist MQTT in der Version 3.1.1 von der Organisation OASIS standardisiert und befindet sich aktuell in der Entwicklung zur Norm ISO/IEC 20922 [BG14], [IS16].

Neben der M2M-Kommunikation und dem IoT wird MQTT auch für Messenger Dienste wie den Facebook Messenger eingesetzt. Durch den Einsatz von MQTT und einer persistenten Verbindung zwischen mobilen Endgeräten und Server konnte die Latenz beim Nachrichtenversand um ein Vielfaches gesenkt werden [Zh16].

3.2 Nachrichtenversand mittels MQTT

MQTT basiert auf einer Client-Server-Architektur und implementiert das Publish/Subscribe Pattern Pattern [BG14]. Im TCP/IP-Referenzmodell ist MQTT in der Anwendungsschicht angesiedelt. Zum Versand von Nachrichten muss ein Client mittels eindeutiger ID eine Verbindung zum Broker aufbauen. Ein Broker ist ein Server und kümmert sich um den Nachrichtenversand. Nach dem Verbindungsaufbau kann ein Client (sog. Publisher) eine Nachricht auf einem Topic veröffentlichen und Topics für den Empfang von Nachrichten abonnieren (sog. Subscriber).

Im Beispiel von Abb. 1 wird vom Client „DataMining“ das Topic „gyro“ abonniert. Der Client „Notfallzentrale“ ist Subscriber des Topics „alarm“. Sobald der Client „Smartwatch“ die Daten eines Gyrosensors (x,y,z) auf dem Topic „gyro“ veröffentlicht, werden diese vom Broker an „DataMining“ gesendet. „DataMining“ analysiert die Sensordaten. Wird bspw. ein Ereignis wie Sturz erkannt, wird eine Nachricht auf dem Topic „alarm“ veröffentlicht und vom Broker an den Client „Notfallzentrale“ weitergeleitet. Zur Sicherstellung des Nachrichtenversands stellt MQTT folgende QoS-Level bereit [BG14]:

- QoS-Level 0: Versand von max. einer Nachricht („Fire and Forget“)
- QoS-Level 1: Versand von min. einer Nachricht, Duplikate möglich
- QoS-Level 2: Versand von genau einer Nachricht, keine Duplikate

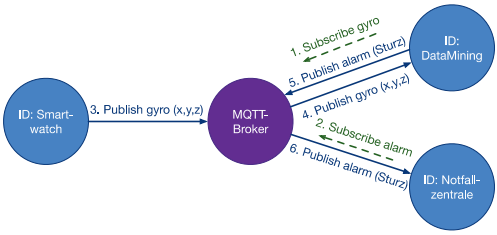

Abb. 1: Grundlegende Funktionsweise des Nachrichtenversands mit MQTT

3.3 Topics und Topic-Filter

MQTT verwendet Topics zum Filtern von Nachrichten. Ein Topic ist eine UTF8-Zeichenkette. Topics müssen mindestens ein Zeichen lang sein und können Leerzeichen enthalten. Die Topic-Namen sind frei wählbar und können zum Abbilden komplexer Strukturen hierarchisch aufgebaut werden (sog. Topic Levels). Ein Beispiel für ein Topic ist: `accelerometer/smartwatch`.

Zum Filtern von Topics stellt MQTT zwei Wildcard-Zeichen bereit. Wie in Tab. 1 ersichtlich können mittels Pluszeichen (+) eine Ebene und mit dem Rautezeichen (#) mehrere Ebenen eines Topics gefiltert werden [BG14]. Zusätzlich wird noch das $ Zeichen für Broker interne Verwendung verwendet.

Filter	Exemplarische MQTT-Topics
`+/smartwatch`	accelerometer/smartwatch oder gyrosensor/smartwatch
`gyrosensor/#`	gyrosensor/smartwatch/001 oder gyrosensor/smartwatch/002

Tab. 1: Einsatz von Wildcard-Zeichen zum Filter von MQTT-Topics

3.4 Aufbau eines MQTT-Pakets

MQTT zeichnet sich durch einen geringen Protokoll-Overhead von nur 2 Byte[3] im fixen Header aus. Das erste Byte im fixen Header unterteilt sich wie in Abb. 2 ersichtlich in zwei Nibbles. Das erste Nibble dient als Indikator eines von insgesamt 14 Pakettypen. Über das zweite Nibble können zugehörige Flags wie das QoS-Level bei einem Publish-Paket übermittelt werden. Im zweiten Byte wird die verbleibende Paketlänge übermittelt. Optional können gemäß Spezifikation drei weitere Bytes für die Längenangabe verwendet werden. Abhängig vom Pakettyp sind der variable Header sowie die Payload zum Teil optional. So wird bspw. bei QoS-Level 1 und 2 der variable Header zur Identifizierung der Pakete verwendet. Bei einem Publish-Paket wird die Nachricht über die Payload übermittelt [BG14].

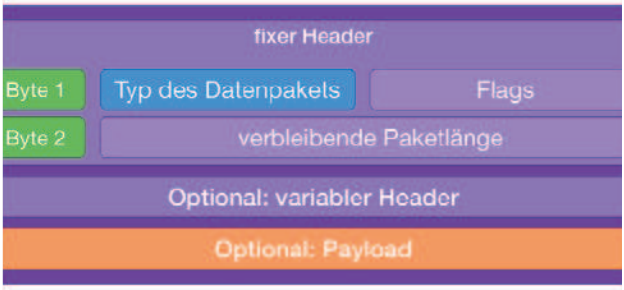

Abb. 2: Aufbau eines MQTT-Datenpakets

[3] Mit 2 Byte können max. 127 Byte Payload, bei 5 Byte max. 256 MB übermittelt werden [BG14].

Kritisiert wird in diesem Zusammenhang die fehlende Möglichkeit, Metadaten zur übertragenen Payload zu übermitteln. Im Vergleich zu MQTT stellt bspw. HTTP einen Content-Type-Header für die Spezifizierung des Entity-Bodies bereit [Fi14]. Mitentwickler Stanford-Clark begründet diese Entscheidung mit „MQTT is intended as a transport, not a protocol. Its job is to get the data from one machine to another, not to define what that data is“ [Cr16]. MQTT ist damit als die Basis für ein eigenes Protokoll zu sehen, in welchem das Datenaustauschformat sowie die Semantik der Daten spezifiziert werden muss.

3.5 Mögliche Datenaustauschformate für MQTT

Das Protokoll MQTT gibt kein Datenaustauschformat vor. Dies wirft die Frage auf, welche Austauschformate sich für MQTT anbieten. Stanford-Clark nennt JSON und „Protocol buffers“ von Google (protobuf) als interessante Austauschformate für MQTT [Cr16]. Mit JSON werden Daten textuell übertragen und sind somit menschenlesbar. Mit einem binären Datenformat wie protobuf können Daten bspw. durch die Verwendung von Datentypen platzsparender übermittelt werden [PR16].

4 Architekturentwurf xHealth

4.1 Analyse

In einem möglichen Anwendungsfall empfängt xHealth die Sensordaten eines smarten Objekts über einen MQTT Broker zur weiteren Analyse mittels Data Mining, um bspw. Ereignisse wie einen Sturz zu erkennen. Bei Sturzerkennung wird ein externes System wie eine Notfallzentrale informiert. Zur Analyse sind Daten eines Gyro- und Beschleunigungssensor 245-mal pro Sekunde4 an xHealth übermittelt.

Mittels RESTful Services basiert auf HTTP sollen nach der Vision des WoT Sensordaten übermittelt bzw. abgerufen werden können. Hierbei ist insbesondere die Interoperabilität zum Protokoll MQTT von Bedeutung. In einem Anwendungsfall werden Temperatur- und Luftdrucksensordaten mittels MQTT an xHealth übermittelt und von einer Smartwatch über die RESTful API abgerufen. Beide Anwendungsfälle werden in Kapitel 5 exemplarisch Implementiert.

4.2 Softwarearchitektur

Ziel der Softwarearchitektur ist eine möglichst unabhängige Entwicklung der Komponenten [Ru12]. Nachfolgend soll xHealth als Whitebox betrachtet und in Komponenten gegliedert werden. Für die Definition der Komponenten wird ein Use-Case-orientierter

[4] gängige Frequenz des im Prototyping verwendeten Gyrosensors „L3GD20H" [ST16]

Ansatz gewählt. Dieser macht davon Gebrauch, dass jeder Use-Case aus der Softwareanalyse eine entsprechende Applikationslogik benötigt [Ru12]. Hierzu wird das Layer-Pattern eingesetzt, welches die zukünftige Wartung vereinfacht und eine inkrementelle Entwicklung ermöglicht [Su12].

Auf dem obersten Layer befinden sich die der MQTT-Broker und RESTful API sowie die Benutzeroberfläche. Auf der Applikationsschicht sind sechs Module angesiedelt. Die Komponente **„Data Mining“** erhält Daten der Schnittstellen zur Erkennung von Mustern in Bewegungsdaten. **„Workflow Management“** dient zur Verwaltung von Workflows wie bspw. dem Nachrichtenversand bei Mustererkennung. Mittels **„Reporting“** können bspw. erkannte Muster mittels User Interface (GUI) visualisiert werden. Ebenfalls über das GUI bedienbar sind die administrativen Komponenten. Die Verwaltung smarter Objekte erfolgt über die Komponente **„Device Management“**. Das „**Access Management**“ reglementiert die Zugriffsrechte auf die APIs. Rollen und Benutzerkonten werden im **„User Management“** verwaltet. Die Datenhaltung erfolgt im untersten Layer. Die Speicherung von Entitäten wie Sensordaten soll in einer **Datenbank** erfolgen. Mittels Dateisystem können Daten wie Konfigurationen gespeichert werden.

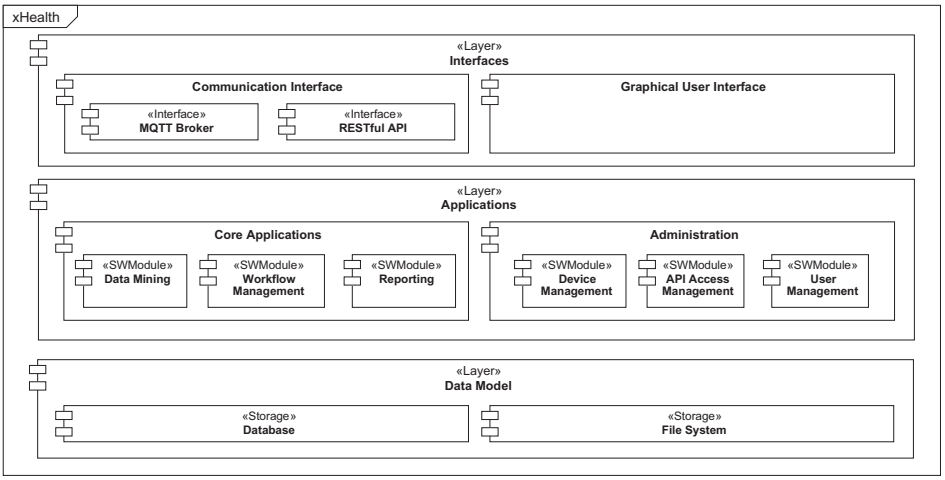

Abb. 3: Komponenten von xHealth sowie die Konnektivität zu weiteren Systemen

Aus technologischer Sicht eignet sich für die Realisierung bspw. Open Source Software basierend auf einer MEAN-Architektur. Der MEAN-Stack besteht aus der NoSQL-Datenbank MongoDB, Express als Webserver-Framework, AngularJS Frontend-Framework und Node.js als Serverplattform [Ha14]. Vorteile einer MEAN-Architektur liegen vor allem in der Skalierbarkeit und dem nichtblockierenden Zugriff auf I/O-Ressourcen durch asynchrone JavaScript-Programmierung [Sn16]. Weiter hat sich MongoDB als Speicher für Sensordaten und Big Data Anwendungen bewährt. So setzt bspw. Bosch für die IoT Cloud auf eine MongoDB basierte Architektur [BO14].

MQTT-API

Zum Nachrichtenversand mittels MQTT wird ein Broker benötigt. Mosquitto ist ein Open Source Broker basierend auf der Programmiersprache C, welcher im Rahmen des Projekts „Eclipse IoT“ entwickelt wird. Der Broker zeichnet sich durch einen besonders schonenden Umgang mit Ressourcen aus. So beanspruchen bspw. 1000 verbundene Clients lediglich 3 MB RAM [EC16a].

Sensordaten werden von smarten Objekten über folgendes Topic-Pattern veröffentlicht: <api-version>/<device-id>/<sensortyp>

In der ersten Ebene wird eine API-Version übermittelt, um nicht abwärtskompatible Änderungen an der API vornehmen zu können. In der zweiten Ebene wird die Device-ID übermittelt. Das dritte Level dient der Zuordnung unterschiedlicher Sensortypen. Weiter existiert ein Rückkanal, um Daten von xHealth an das jeweilige smarte Objekt zu versenden. Hierzu wird die Ebene der Device-ID verwendet.

Topic	Beschreibung
v1/123/gyr	Gyrosensor-Daten von Device 123
v1/123/acc	Beschleunigungssensor-Daten von Device 123
v1/123/tem	Temperatur-Daten von Device 123
v1/123/pre	Luftdruck-Daten von Device 123
v1/123	Rückkanal für Device 123

Tab. 2: Topic-Struktur von xHealth am Beispiel von Device 123

Zur Spezifizierung der Semantik der zu übermittelten Daten soll das binäre Format protobuf verwendet werden. Tab. 3 zeigt einen Versuch, in welchem ein Datensatz eines Gyrosensors bestehend aus x-, y- und z-Werten in der Einheit rad/s mit verschiedenen Datenformaten übermittelt und die Datenmenge gemessen wurde. Mit protobuf können Daten platzsparender als mit JSON oder CSV übermittelt werden.

Datenformat	Typ	Payload	Datenmenge
CSV	textuell	-0.02,-0.09,-0.01	17 Byte
JSON	textuell	{"x":-0.02,"y":0.09,"z":-0.01}	30 Byte
protobuf	binär	0d0ad7a3bc15ec51b8bd1d0ad723bc	15 Byte

Tab. 3: Vergleich der Datenmenge am Beispiel eines Gyrosensor-Datensatzes

Die Validierung der Daten erfolgt durch die Definition von Schemata, welches von jedem Kommunikationspartner zu implementieren ist. Zugriffsklassen für eine spezifische Programmiersprache können mittels Compiler erzeugt werden.

Zugriffsklassen für die Verwendung eines Schemas mit unterschiedlichen Programmiersprachen wie C++ oder Java können mittels Compiler erzeugt werden [PR16]. Mit JavaScript kann mittels dem Modul protobuf.js ein protobuf Schema ohne Einsatz eines Compilers verwendet werden [Wi16].

5 Anwendungsfälle für xHealth

5.1 Prototypische xHealth basierte Wetterstation

Die prototypische Implementierung von xHealth erfolgt auf einer Amazon EC2-Instanz vom Typ „t2.micro“ mit Ubuntu 14.04 LTS. Zur Realisierung der MQTT- sowie RESTful API wird das IoT-Framework Node-RED basierend auf Node.js und der MQTT Broker Mosquitto eingesetzt.

Abb. 4 zeigt eine App auf einer Smartwatch, welche Daten eines Temperatur- und Luftdrucksensors mittels HTTP-Methode GET über die RESTful API von xHealth abruft und auf dem Display darstellt. Diese Daten wurden zuvor von einem Arduino Uno ausgelesen und mit dem MQTT-Client „PubSubClient“ an xHealth übermittelt [On16]. Der Nachrichtenversand mittels MQTT erfolgt über die Topics in Tab. 4. Für die Entwicklung der App wird das Garmin Connect IQ SDK sowie die Eclipse IDE eingesetzt [GA16]. Die Programmierung erfolgt in „Monkey C“. Diese Sprache wurde für Wearables entwickelt und weist Ähnlichkeiten zu Java, Ruby und Python auf [GA16]. Als Basis der eigenen Entwicklung dient die mit dem SDK gelieferte App „JsonRequest“.

Topic	Beschreibung
v1/001/tem	Temperatur-Daten von Gerät 001 (Arduino)
v1/001/pre	Luftdruck-Daten von Gerät 001 (Arduino)

Tab. 4: MQTT-Topics des Arduino Uno

Die Übermittlung der Daten erfolgt mit einem protobuf-Schema.

```
message Weather{
    required float temperature = 1;
    required float pressure = 2;}
```

Nachfolgendes Beispiel zeigt einen JSON-Response mittels dem Tool „curl“.

```
curl -X GET http://api.xHealth.io/v1/001/wetter
{"temp":21.60,"pressure":967.309998}
```

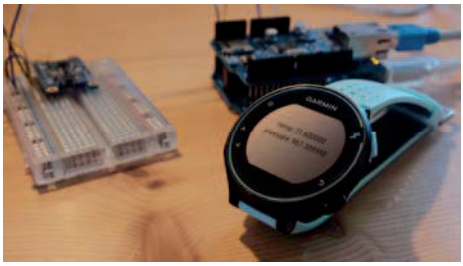

Abb. 4: Abruf von Meteodaten über xHealth auf einer Garmin Forerunner 235 Smartwatch

5.2 Smartwatch basierte ADL/EDL Erkennung

Die Erkennung von Aktivitäten des täglichen Lebens (ADLs) wie Zähneputzen, Rasieren, Trinken, Essen, Kämmen, Schreiben etc. sowie Ereignisse des täglichen Lebens (EDLs) wie Aufstehen, Zubettgehen, aber auch Stürze können helfen, frühzeitig Veränderungen im Verhalten von Personen (z.B. in der Agilität etwa bei altersbedingter Demenz) bzw. kritische, gesundheitsgefährdende Situationen wie Dehydrierung, Sturz, Desorientierung zu erkennen und ggf. notwendige Gegenmaßnahmen zu ergreifen [LW15a], [LW15b], [WA15]. Smartwatches sind ideale Werkzeuge, um die dafür notwendigen Daten erfassen und diese Aktivitäten erkennen und analysieren zu können [BA15], ohne dass deren Träger durch den Einsatz solcher Technologien stigmatisiert werden, wie es bei den derzeit am Markt erhältlichen Produkten der Fall ist [LW15b]. [Ba15] und [LW15] verwenden verschiedene Methoden des Data Mining um aus Sensordaten ADLs in Realzeit erkennen lassen können. Dazu werden verschiedene Modelle entwickelt und trainiert (logistische Regression, neuronale Netze, NMM und Entscheidungsbäume). Mit diesen Modellen werden zur Laufzeit die Sensordaten überwacht und die entsprechenden ADLs/EDLs ermittelt. Ziele von [LW16] ist es eine Notfall-App zu entwickeln, die den Träger im Alltag und in Notfallsituationen unterstützt. [15c] beschreibt einen Architekturvorschlag, wie Anwendungen auf Smartwatches modular gestaltet werden können, sodass sie den Anforderungen. Die App läuft auf einer Samsung Gear S und ist in Tizen mit JavaScript realisiert.

Die Erkennung der ADLs/EDLs erfolgt in mehreren Schritten (Abb. 5):

1) Die Sensordaten werden in 1-Sekunden oder 5-Sekunden-Abständen zu 10-Sekunden-Blöcken zusammengefasst,

2) aus diesen 10-Sekunden-Blöcken werden statistische Parameter wie Mittelwert, Standardabweichung, Interquartilsabstand, Nulldurchgänge etc. berechnet und

3) die Parameter werden auf das entsprechende trainierte Modell angewandt und die passende ADL/EDL ermittelt.

Bei der Erkennung von ADLs unmittelbar auf der Smartwatch ist insbesondere die Ermittlung der statistischen Parameter sehr rechenaufwändig. Es müssen für das beschriebene 10-Sekunden Auswerteintervall etwa 4000 Sensorwerte kontinuierlich alle 1 oder 5 Sekunden je nach Implementierung verrechnet werden. Wie verschiedene Experimente, u.a. auf einer Samsung Gear Live Android Wear Implementierung in Java zeigt, überfordert dies die Hardware der Smartwatch trotz verschiedener Optimierung der statistischen Routinen. Ähnliches gilt für die Samsung Gear S mit Tizen und JavaScript als Implementierungssprache. Nach etwa 30 – 40 Sekunden werden die Berechnungen nur mehr stark zeitverzögert durchgeführt, es beginnen Intervalle zu fehlen. Um trotzdem eine Realzeiterkennung zu ermöglichen, wurde in einem ersten Versuch die Auswertung auf einen Server verlagert, auf dem insbesondere das Data Mining Tool RapidMiner zur Anwendung kommt. Wie in Abb. 6b) dargestellt, wurde dazu ein entsprechender HTTP/REST basierter Service mit JSON als Austauschformat aufgesetzt. Erste Versuche

zeigen, dass damit Realzeit-Analysen möglich sind, aber durch die Verwendung von HTTP einen entsprechenden großen Overhead erzeugt wird. Dieser fällt bei einer Verwendung im häuslichen WLAN mit entsprechenden Übertragungsgeschwindigkeit kaum ins Gewicht, bei schlechten Verbindungsqualitäten kann es aber zu merklichen Verzögerungen kommen. Gerade außer Haus hier ist aber eine rasche Reaktion notwendig.

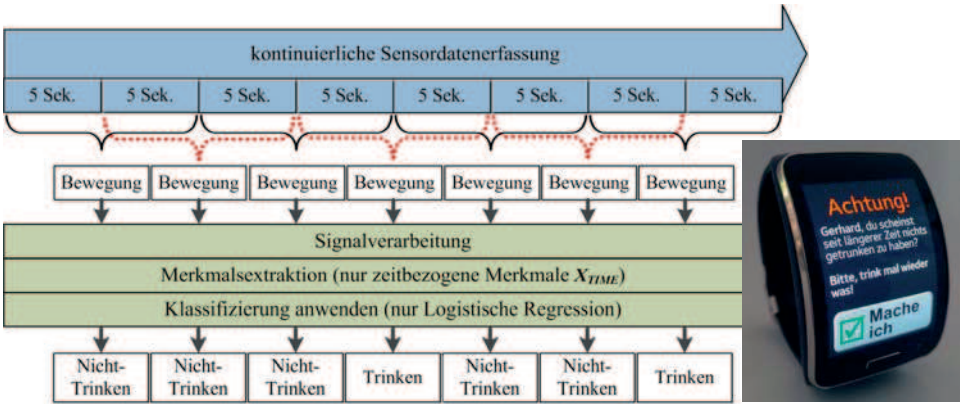

Abb. 5: Datenerfassung und Realzeiterkennung von ADLs/EDLs [WB15], [WL16]

MQTT kann hier Verbesserungen bringen: a) durch den geringeren Protokolloverhead, b) durch eine entsprechende Komprimierung der übertragenen Daten, etwa wie beschrieben mittels protobuf und c) durch die Möglichkeit einen entsprechenden QoS Level zu wählen. So bietet sich hier QoS-Level 0 an, da bei der großen Anzahl übertragener Daten einige fehlende Sensordaten die ADL/EDL-Erkennungsraten kaum beeinflussen. Analog wäre auch der QoS-Level 2 denkbar, um die multiple Übertragung von Daten zu verhindern.

Ein mögliches Einsatzszenario der Architektur wird in Abb. 6 beschrieben. Am zentralen xHealth Broker können sich mehrere Smartwatches und Anwendungen anmelden und unterschiedliche Topics subskribieren bzw. publizieren (sensor, adl, Tab. 5). [Wn]-Smartwatches werden von betreuungsbedürftigen Personen, deren ADLs erfasst werden sollen, getragen. Sie publizieren Sensordaten, die vom ADL BDA Server subskribiert werden. Dieser führt die ADL/EDL Erkennung durch und publiziert seinerseits die erkannten ADLs. Die W1…Wn Smartwatches werden über ihre ADL Subskriptionen informiert, etwa um im Falle eines Sturzes noch entscheiden zu können, ob eine Benachrichtigung der Notrufzentrale erfolgen soll. Parallel dazu kann der Träger der Smartwatch C1, z.B. eine Betreuungsperson, über den Sturz über die ADL Subskription informiert werden, ebenso wie die Notfallzentrale.

Im Rahmen eines ersten Proof of Concepts (Abb. 7) konnte gezeigt werden, dass mit diesem Ansatz problemlos die erzeugten Mengen an Sensordaten übertragen, am Server verarbeitet und erkannte ADLs an die Smartwatches übertragen werden können.

Filter	ADL/EDL MQTT-Topics
`v1/001/adl`	Abonniere Token für ADL Erkennung für Gerät 001
`v1/+/adl`	Abonniere Token für ADL Erkennung für eine weiteren PC-basierten ADL-Klienten, etwa eine Notrufzentrale. Hier werden alle ADLs von allen angemeldeten Geräten ausgewertet
`v1/+/sensor`	Abonniere Token für Sensordaten der Smartwatches

Tab. 5: MQTT-Topics zur Erkennung vom ADLs/EDLs

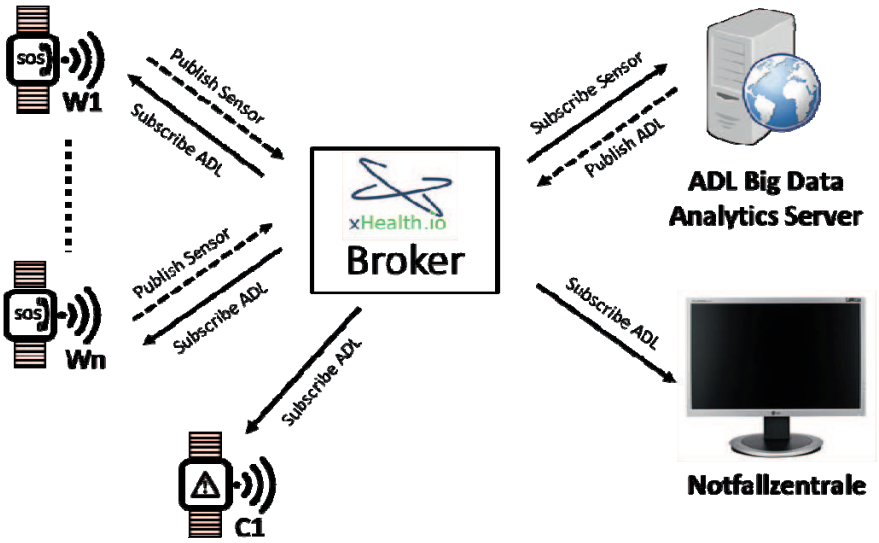

Abb. 6: Einsatzszenario xHealth zur ADL/EDL Erkennung

Abb. 7: Testumgebung ADL Server / Smartwatch

6 Untersuchung der Effizienz

Zur Ermittlung des Datendurchsatzes werden zwischen dem Arduino Uno und xHealth jeweils 10.000 Pakete in der Reihenfolge 10, 50 und 100 Byte Payload übermittelt. Der Nachrichtenversand mittels MQTT erfolgt unter Verwendung von QoS-Level 0 und dem MQTT-Client „PubSubClient“[On16]. Die Übermittlung der Daten an die RESTful API erfolgt mittels HTTP-Methode „POST“.

Abb. 8 zeigt mittels Wireshark IO-Graph die zuvor mit tshark serverseitig aufgezeichneten Netzwerkdaten der Protokolle MQTT und HTTP.

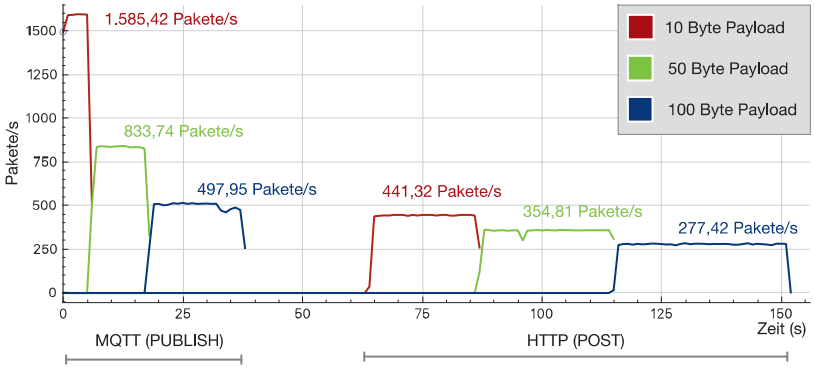

Abb. 8: Analyse der von xHealth empfangenen Pakete mit Wireshark

Der Arduino Uno arbeitet mit einer Taktfrequenz von 16 MHz basiert auf einer 8 Bit Architektur und verfügt über 2 KB RAM, was die Implementierung von TCP/IP sowie die maximale Übertragungseinheit beim Datenaustausch limitieren. Im Vergleich hierzu arbeiten aktuelle Smartwatches mit bis zu vier Prozessorkernen basiert auf einer 32 Bit Architektur mit einer Taktfrequenz von mehr als einem GHz.

7 Fazit

Basiert auf der prototypischen Implementierung von xHealth sowie der Untersuchung der Effizienz von MQTT wird folgendes Fazit gezogen:

- MQTT und xHealth bieten eine interessante Möglichkeit die Sensordaten auf einem Server auswerten zu lassen und den Tradeoff zwischen beschränkter Smartwatch Hardware und Erkennungsgeschwindigkeit zu optimieren.

- Das MQTT Publish/Subscriber Modell von xHealth bieten für die ADL/EDL Erkennung und Einbindung in ein Gesamtsystem zur Notfallunterstützung eine solide Grundlage für eine einfach erweiterbare Kommunikationsarchitektur.

- Die Interoperabilität der übermittelten Daten zwischen MQTT/protobuf und HTTP/JSON konnte am Beispiel des Anwendungsfalls „Wetterstation“ aufgezeigt werden.
- Mit MQTT kann zwischen einem Arduino Uno und xHealth basiert auf einer Payload von 10 Byte mehr als der 3-fache Datendurchsatz erreicht werden.
- Beim Datenaustausch über MQTT kann durch den Einsatz von protobuf die Datenmenge der zu übermittelnden Payload im Vergleich zu JSON am Beispiel der Daten eines Gyrosensors um etwa 50 % reduziert werden.

Literaturverzeichnis

[Ba15] R. Baldauf, “Mobile sensorbasierte Erkennung von Trinkbewegungen,” Thesis, FOM, Nürnberg, 2015.

[BO14] Bosch Si Whitepaper: IoT and Big Data, www.bosch-si.com/lp/iot-big-data.html, Stand: 15.05.16.

[BG14] Banks, A.; Gupta, R.: MQTT Version 3.1.1. OASIS, 2014.

[Cr16] Craggs, I.: Why doesn’t MQTT have a payload format, www.modelbasedtesting.co.uk/?p=243, Stand: 02.04.2016.

[EC16a] Eclipse Mosquitto, projects.eclipse.org/projects/technology.mosquitto, Stand: 15.05.2016.

[EC16b] Eclipse Paho - Open Source messaging for M2M, www.eclipse.org/paho, Stand: 20.04.2016.

[Fi14] Fielding, R. et.al.: Hypertext Transfer Protocol – HTTP/1.1 (RFC: 7230). Internet Engineering Task Force, 2014.

[GA16] Garmin Developer – connect IQ SDK, developer.garmin.com/connect-iq/overview/, Stand: 15.05.2016. [GT15] Guinard, D.; Trifa, V.: Building the Web of Things - MEAP. Manning Publications, New York, S. 23, 2015.

[Ha14] Haviv, A.: MEAN Web Development. Packt Publishing Ltd, Birmingham, S. 7, 2014.

[HI16] HiveMQ – MQTT 101, www.hivemq.com/blog/how-to-get-started-with-mqtt, Stand: 02.05.2016.

[IS16] ISO/IEC 20922 - Message Queuing Telemetry Transport (MQTT) v3.1.1, www.iso.org/iso/catalogue_detail.htm?csnumber=69466, Stand: 21.04.2016.

[KK16] Klose, B.; Klose, H.: Meteorologie: eine interdisziplinäre Einführung in die Physik der Atmosphäre. Springer-Verlag, Berlin, S. 3, 2016.

[LW15a] Lutze, R., Waldhör, K., Baldauf, R.: Dehydration Prevention and Effective Support of Elderly by the Use of Smartwatches, in Proceedings of the IEEE Healthcom 2015, 2015.

[LW15b] Lutze R., Waldhör K.: “SmartWatches als Hausnotrufsysteme der nächsten Generation,” in 8. AAL Kongress - Zukunft Lebensräume 2015, Berlin: VDE Verlag, 2015.

[LW15c] Lutze R., Waldhör K.: A Smartwatch Software Architecture for Health Hazard Handling for Elderly People, in ICHI 2015, 2015, pp. 356–361, 2015.

[LW16] Lutze R., Waldhör K.: Integration of Stationary and Wearable Support Services for an Actively Assisted Life of Elderly People: Capabilities, Achievements, Limitations, Prospects - A Case Study, In: Wahlster W (Hrsg.) Zukunft Lebensräume 2016, Springer, Frankfurt, 2016.

[MO16] Mosca – MQTT broker as a module, www.mosca.io, Stand: 07.05.2016.

[On16] O’Leary, N.: PubSubClient - Arduino Client for MQTT, pubsubclient.knolleary.net, Stand: 26.04.2016.

[Pe16] Pettey, C.: Smarter With Gartner: The Internet of Things and the Enterprise, www.gartner.com/smarterwithgartner/the-internet-of-things-and-the-enterprise, Stand: 08.05.2016.

[PM16] PM2 - Advanced Node.js process manager, www.pm2.keymetrics.io, Stand: 08.05.2016.

[PR16] Protocol Buffers, developers.google.com/protocol-buffers, Stand: 15.05.2016.

[Sn16] InfoWorld - Why Node.js beats Java and .Net for Web, mobile, and IoT apps, www.infoworld.com/article/2975233/javascript/why-node-js-beats-java-net-for-web-mobile-iot-apps.html, Stand: 15.05.2016.

[ST16] STMicroelectronics - L3GD20H Datasheet, www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00060659.pdf, Stand 04.04.2016.

[Su12] Summerville, I.: Software Engineering. Pearson Deutschland GmbH, München, S. 194, 2012.

[Ru12] Rupp, C. et.al.: UML2 glasklar. Carl Hanser Verlag, München, S. 93, 2012.

[W316] W3C - Bindings To Common Protocols, www.w3.org/WoT/IG/wiki/Bindings_To_Common_Protocols, Stand: 10.05.2016.

[WB15] Waldhör, K., Baldauf, R.: Recognizing Drinking ADLs in Real Time using Smartwatches and Data Mining. In S. Fischer, I. Mierswa, & G. Schäfer (Eds.), Proceedings of the RapidMiner Wisdom Europe (2015) (pp. 1–18). Aachen: Shaker., 2015

[Wi16] Wirtz, D. et.al.: GitHub - protobufjs, github.com/dcodeIO/protobuf.js, Stand: 15.05.2016.

[WL16] Waldhör, K., Lutze, R., Baldauf, R.: ADL Erkennung mit Smartwatches: Data Mining zur Realzeiterkennung von Aktivitäten des täglichen Lebens (ADL). In K. Waldhör & R. Lutze (Eds.), wearables /at /work. Konferenz: wearables \at \work 2016. 17.02.2016, München, Nürnberg: FOM, 2016.

[Zh16] Zhang, L.: Building Facebook Messenger, www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920, Stand: 22.03.2016.