Standort-basierte Daten via MQTT

Seminararbeit “Webservice Security”

Marco Wettstein

2015-05-14

Inhaltsverzeichnis

1 Einleitung

2 Ausgangslage

2.1 Timetraces

Im Rahmen einer Seminararbeit wurde für die Controlling- und Zeiterfassungs-Applikation “controllr” (siehe Abbildung 1) eine neue Client-Anwendung gebaut, welche durch die Integration verschiedener Dienste wie Github, Redmine und Google Calendar eine Art Protokoll der geleisteten Arbeit erstellt. Aus den Einträgen dieses Protokoll können in der Anwendung direkt Zeiteinträge in “controllr” erstellt werden. Abbildungen 2 zeigt das Arbeitsprotokoll von *TimeTraces*. Durch Anwählen eines Eintrages wird eine vor-ausgefüllte Eingabemaske angezeigt, welche den Zeiteintrag über eine REST-Schnittstelle an “controllr” sendet (Abbildung 3).

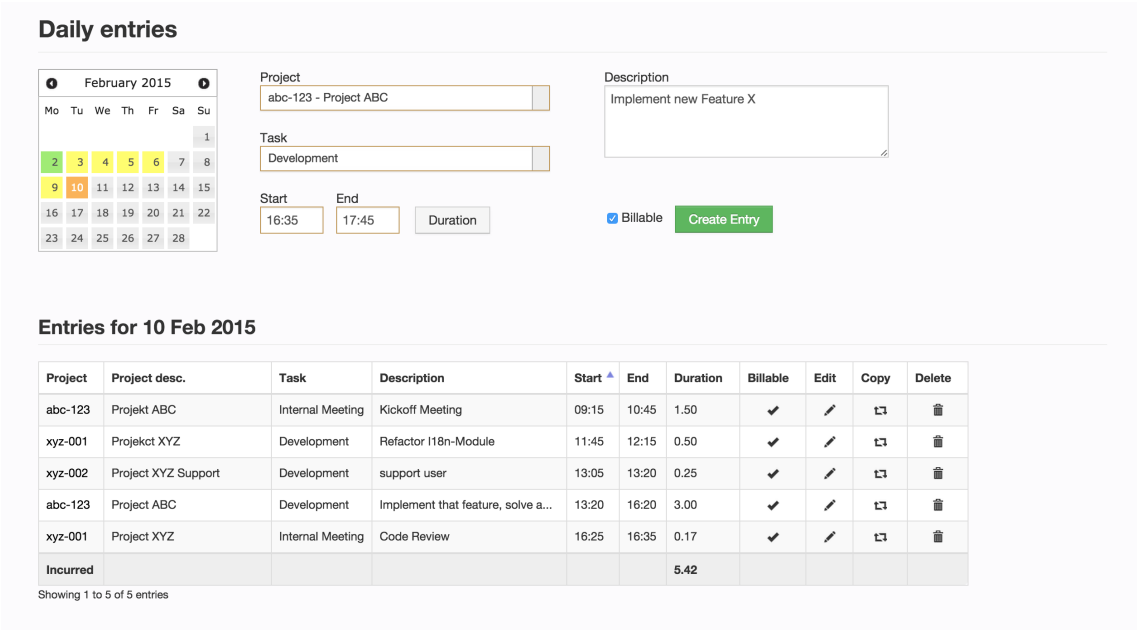

Abbildung 1: Screenshot von “Controllr” (Quelle (Wettstein, 6))

TimeTraces wurde als “Meteor”-Anwendung gebaut (siehe dazu Abschnitt **??**) und ist eine Client-Server-Anwendung, welche externe Dienste integriert. Die Anwendung speichert dabei ausser den Benutzer-Logins und den Einstellungen der Benutzer keine weiteren Daten (Siehe Abschnitt 2.1.2). Sämtliche Daten werden dabei vom Serverteil der Anwendung aggregiert und an den Client gesendet. Die Daten werden vom Server dabei über REST-Schnittstellen in einem Polling-Verfahren abgerufen. Das Polling wird gestartet, sobald der clientseitige Teil der Anwendung die Daten über eine DDP-Subscription abonniert und beendet, sobald der Client die Subscription beendet.

Abbildung 4 zeigt den Ablauf einer Subscription eines Clients.

2.1.1 Standortverlauf als weitere Event-Quelle

TimeTraces nutzt bisher *Github*, *Google Calendar* und *Redmine* als Event-Quellen. Als weitere Event-Quelle soll nun der Standort-Verlauf des Benutzers genutzt werden. Diese Daten sollen dem Benutzer helfen, die Zeiteinträge genauer zu erfassen. Der Benutzer sieht somit nicht nur, was wann gearbeitet wurde, sondern auch wo. Es löst zudem das Problem, dass es oft schwierig ist, sich an den Startzeitpunkt einer Arbeit zu erfassen: es kann z.b. festgestellt werden, wann das Büro betreten wurde.

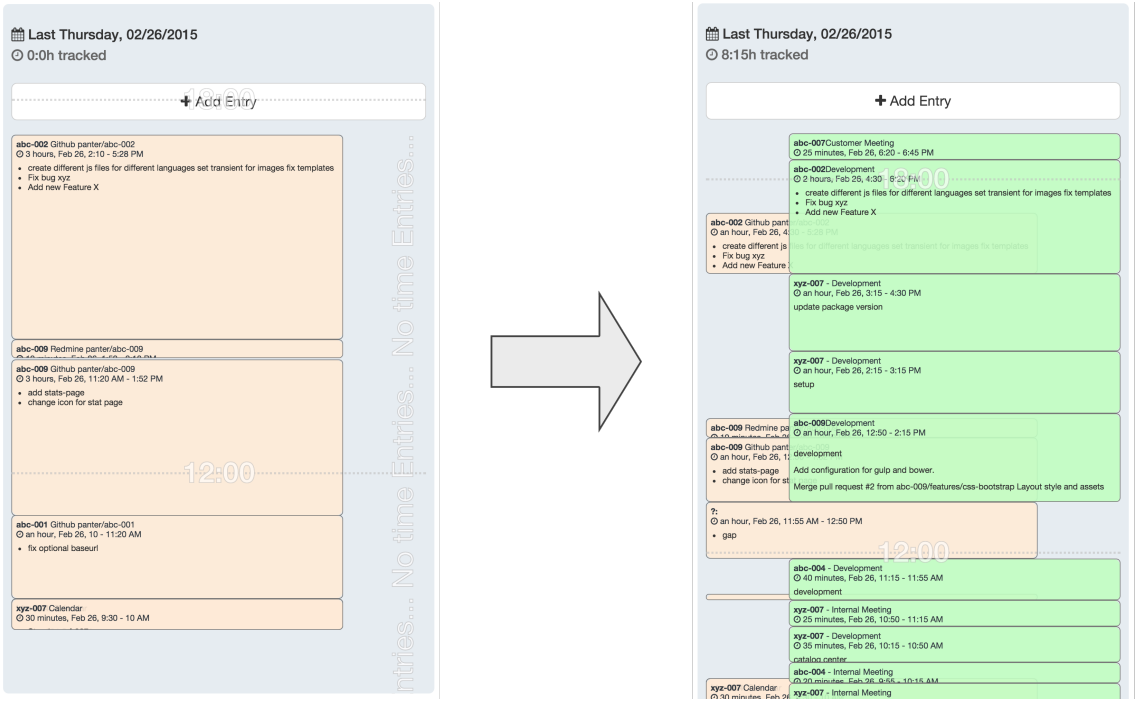

Abbildung 2: Darstellung der Event-Liste eines Tages in *TimeTraces*. Quelle (Wettstein, 22)

2.1.2 No-Data-Konzept

Im Zeitalter von Cloud-Computing ist es schwierig geworden, genau zu bestimmen, wo die Daten eines Benutzers überall sind und wer die Kontroller über diese Daten hat.

Beim Design von *TimeTraces* wurde dies berücksichtig, indem nur die nötigsten Daten, wie Einstellungen von Benutzer gespeichert werden. Die verwendeten Daten der Event-Quellen, wie Kalender udn Github werden dabei nicht dupliziert und gespeichert, sondern lediglich an die Client-Anwendung weitergeleitet.

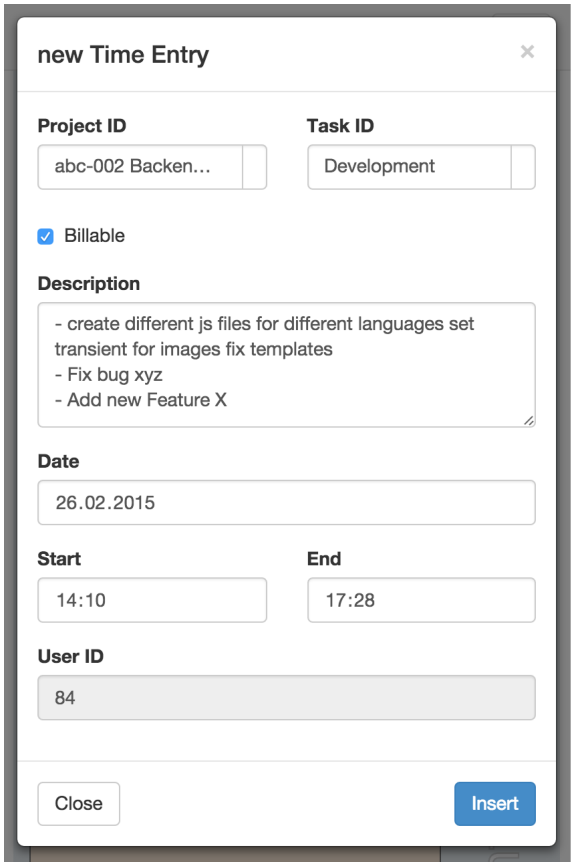

Abbildung 3: Eingabemaske für einen Zeiteintrag in *TimeTraces*. Alle Felder werden vorausgefüllt. Quelle (Wettstein, 23)

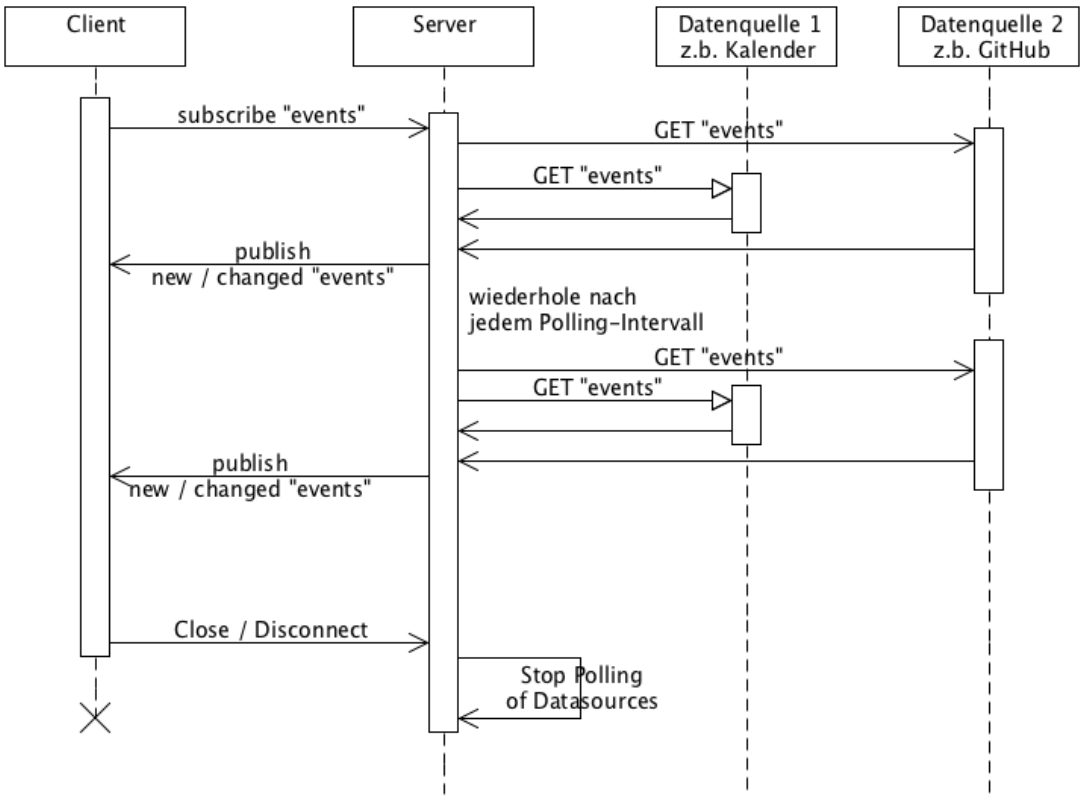

Abbildung 4: Ablauf einer Subscription von *TimeTraces* zwischen Client - Server und externen Quellen. Quelle (Wettstein, 19)

2.2 Standort-Dienste und MQTT

Die Fähigkeiten zur Standort-Bestimmung (GPS, o.ä.) auf Smartphones ermöglicht viele Anwendungen, wie Navigation, Standort-bezogene Informationen oder Protokollierung für Sportler, Wanderer, o.ä. Es existieren viele Dienste zum diesem Thema, unter anderem von Google, wie *Google+* oder den eingestellten Dienst *Google Latitude*. Im mobilen Betriebsystem *Android* ist die Standort-Bestimmung zudem seit den ersten Versionen ein wichtiges Feature.

Neben solchen Cloud-Diensten existieren auch Systeme, welche auf eigener Infrastruktur - und damit unabhängig von Drittanbieter betrieben werden können.

Ein solches System ist die Anwendung *OwnTracks*, welche als Ersatz für den eingestellten Dienst *Google Latitude* entwickelt wurde. Sie kann auf IOS- und Android-Geräten installiert werden und kommuniziert über einen *Broker* über das Protokol *MQTT*, welches als *Protokol des Internets der Dinge* standardisiert werden.

3 Aufgabenstellung

Auf einem mobilen Endgerät des Benutzers soll eine Anwendung laufen, welcher seinen Standort periodisch an eine zentrale Stelle übermittelt.

Die Anwendung *TimeTraces* wird um eine Funktion erweitert, welche diese Daten abfragt und auswertet. Der Benutzer soll dabei sehen, zu welchem Zeitpunkt er sich an welchem Standort aufgehalten hat. Die Daten werden also nicht in Echtzeit benötigt (wie es beispielsweise häufig bei Standort-basierter Werbung der Fall ist), sondern mit Verzögerung (einige Tage bis wenige Wochen).

Als Übertragungsprotokoll soll dabei *MQTT* verwendet werden

3.1 Zielsetzung

Die Ziele dieser Arbeit sind

- Vertiefung in das Thema MQTT
- Betrachtung sicherheitsrelevanter Aspekte von MQTT,
- Betrachtung generell sensitiver User-Daten, wie Standortverlauf
- Setup einer geeigneten MQTT-Broker-Lösung mit geeigneten Sicherheitseinstellungen
- Verbinden von *OwnTracks* oder einer ähnlichen Anwendung und *TimeTraces* via MQTT und dem gewählten Broker
-

4 Recherche

4.1 Standord-Dienste

4.1.1 Google Latitude

Google Latitude war ein Dienst, welche es einem Benutzer ermöglichte, seinen Standort an seine Freunde zu senden und zu sehen, wo sich seine Freunde aufhalten.

Der Dienst wurde eingestellt.[1]

4.1.2 Google+

Im Sozialen Netzwerk *Google+* ist es möglich, seinen Standort an Freunde zu teilen. Der Dienst nutzt dabei das mobile Endgerät des Benutzers für die Strandort-Protokollierung, sofern man die entsprechende App installiert hat.

Über eine *API* lassen sich Daten des Sozialen Netzwerkes programmatisch abfragen. Über

```
GET https://www.googleapis.com/plus/v1/people/userId
```

lassen sich Profil-Daten eines Benutzers abfragen. Wählt man `me` als `userId`, so kann das eigene Profil abgefragt werden. Die Dokumentation der Schnittstelle[2] erwähnt das Feld `currentLocation` als den aktuellen Standort der Person. Eine Test der Schnittstelle hat aber ergeben, dass das Feld nicht existiert. Auch eine gezielte Abfrage wie in Abbildung 5 gezeigt, ergab, dass das Feld nicht existiert.

[1] Siehe (“Latitude Wurde Eingestellt”)
[2] Siehe (“Google+ Platform - People (Personen)”)

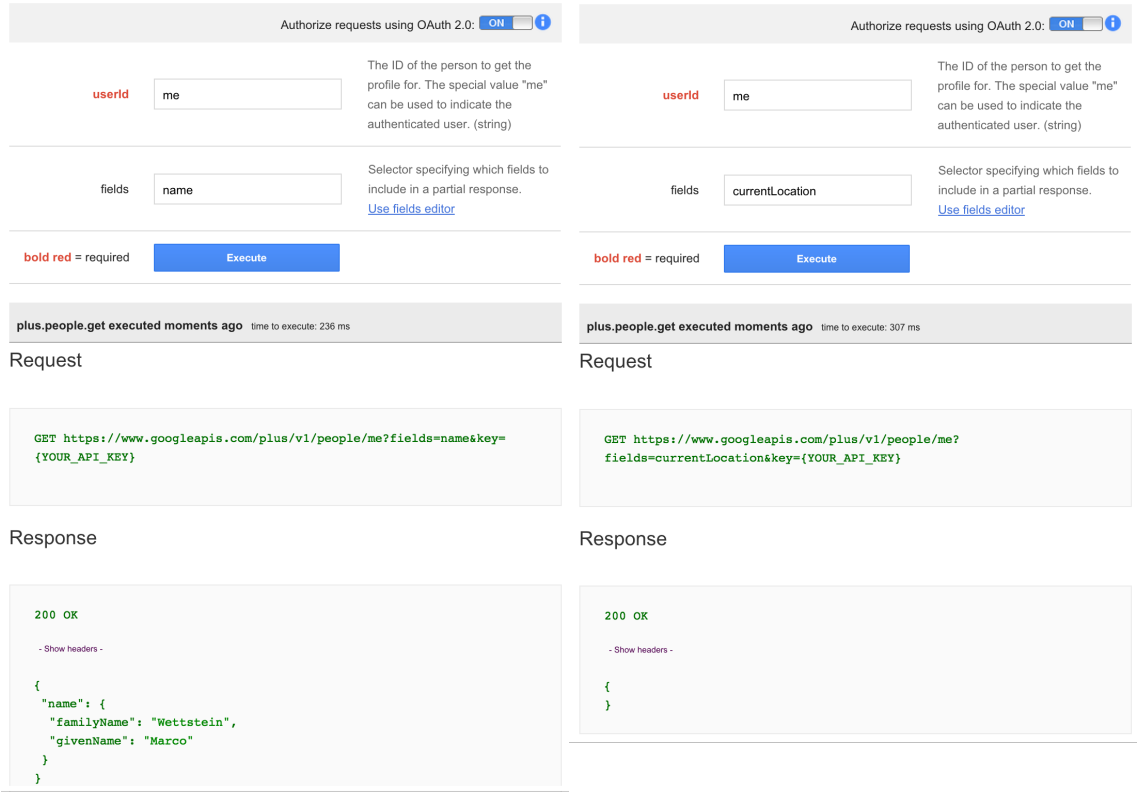

Abbildung 5: Abfrage der Google+-Schnittselle. Wird *name* abgefragt, erscheint eine Antwort, bei *currentLocation* nicht.

Es handelt es sich hier offenbar um einen Fehler. Im *issue-tracker* der *google-plus-platform* ist ein entsprechender Eintrag[3] verfasst word mit dem Datum 24.07.2013. Bis zum Zeitpunkt des Erfassens dieser Arbeit ist keine Lösung zum Problem eingegangen.

4.1.3 OwnTracks

Owntracks wurde als Ersatz für den eingestellten Google Standort-Dienst “Latitude” entwickelt und ursprünglich in Anlehnung an das Vorbild und dem Verwendeten Protokoll als *MQTTitude* bezeichnet.[4]

Die Anwendung zeichnet den Standort des Benutzers im Hintergrund auf und sendet die Daten an einen zu definierenden MQTT-Broker unter einem wählbaren *Topic* (Siehe Abschnitt 4.2.5). Dabei können verschiedene Einstellungen wie die Häufigkeit der Standortprotokollierung

OwnTracks ist als quelloffene Anwendung für IOS und Android erhältlich und ist unter der *Eclipse Public Licence* veröffentlicht.[5]

Die Verwendung von *OwnTracks* ist im Rahmen dieser Arbeit als Ausgangslage vordefiniert und gibt das Protokoll *MQTT* vor, es sollen aber auch Alternativen betrachtet werden[6]

TODO: reinnehmen?

https://play.google.com/store/apps/details?id=com.glympse.android.glympse http://onetouchlocation.creativeworkline.com/

4.2 MQTT

MQ Telemetry Transport oder kurz *MQTT* ist ein Protokoll für die Maschine-zu-Maschine-Kommunikation von Telemetrie-Daten. MQTT wurde insbesondere für Leistungsschwache Endgeräte entwickelt, sowie für Netzwerke mit hoher Verzögerung oder geringer Leistung. So wurde MQTT auch für die Kommunikation über Satelliten genutzt.[7]

MQTT wurde 1999 von Dr Andy Stanford-Clark (IBM) und Arlen Nipper (Arcom, Eurotech) entwickelt.[8]

4.2.1 Netzwerklayer und Sicherheit

MQTT ist im TCP/IP-Referenzmodell in der Anwendungs-Schicht angesiedelt und nutzt TCP als Übetrtragungsprotokoll. Die Übetragung kann mittels SSL/TSL verschlüsselt werden, allerdings erhöht das den Leistungsbedarf der Übertragung signifikant. MQTT sieht innerhalb des Protokolles keine Verschlüsselung vor, es ist jedoch möglich, sich mittels Benutzername und Passwort zu authentifizieren.[9]

Mit *MQTT-SN* steht eine Variante für nicht-TCP/IP-Netzwerke, wie *ZigBee* zur Verfügung.[10]

4.2.2 Anwendungen

Durch diese Optimierungen ist MQTT für Sensoren, wie Temperatur-, Feuchtigkeits oder Durckmesser, Lichtschalter, Bewegungsmelder und Aktoren, wie Lampen, Motoren, Relais oder ähnliches geeignet. MQTT wurde 2013 als Protokoll des *Internets der Dinge* standardisiert und bietet somit beispielsweise eine standardisierte Übertragungsmöglichkeit für die Hausautomation.[11]

Durch die geringe Leistungsaufnahme ist MQTT ebenfalls geeignet für mobile Endgeräte, wie Smartphones, wo lange Akkulaufzeit und geringe Datenübertragung wünschenswert sind.

[3] Siehe (“Field ‘CurrentLocation’ Is Not Provided in API Result (or Empty), While the Field Is Cleary Filled on the Page.”)
[4] Siehe Quelle (“OwnTracks”)
[5] Lizenz und Quelle unter (“OwnTracks Lizenz”)
[6] Owntracks nutzt MQTT als Übertragungsprotokoll. Es ist aber auch denkbar, dass anderen Anwendungen ein anderes Protokoll verwenden.
[7] Siehe Einleitung unter der Offiziellen Seite von MQTT (**???**).
[8] Siehe (“MQTT - Frequently Asked Questions”)
[9] Siehe (“MQTT - Frequently Asked Questions”)
[10] Siehe (“MQTT for Sensor Networks – MQTT-SN”).
[11] Quelle (“Wikipedia - MQ Telemetry Transport”)

4.2.3 Topics und Publish-Subscribe

MQTT folgt dem Konzept einer *Message oriented Middleware* und ermöglicht das *Beobachter-Entwursfmuster*, welches auch *publish-subscribe* genannt wird.[12] Dabei *abonnniert* (“subscribe”) ein Client ein bestimmtes *Topic*. Ein Client kann auf ein *Topic* eine Nachricht *veröffentlichen* (“publish”), welche dann alle Clients erhalten, die dieses *Topic* abbonniert haben. Ein *Broker* dient dabei als Vermittler zwischen den Clients und leitet die Nachrichten an die für sie bestimmten Clients weiter. (Siehe Abschnitt 4.2.5)

4.2.4 Qualitiy of Serivce (QoS)

MQTT sieht 3 Stufen für die Übertragungs-Qualität einer Nachricht vor:

QoS 0 - At most once delivery: Die Nachricht wird **höchstens einmal** zugestellt. Nachrichten mit QoS 0 können verloren gehen, wenn ein Client die Verbindung unterbricht oder ein Broker offline ist. Der Vorteil an QoS 0 liegt primär in der Performance, da Nachrichten nicht zwischengespeichert und nicht protokolliert werden muss, welcher Benutzer welche Nachricht erhalten hat.

QoS 1 - At least once delivery: Clients und Broker versuchen, die Nachrichten **mindestens einmal** zuzustellen. Es ist möglich, dass Nachrichten mehrfach zugestellt werden.[13]

QoS 2 - Exactly once delivery: Diese Stufe garantiert, dass eine Nachricht genau einmal zugestellt wird. Die Stufe stellt somit wie QoS 1 den Empfang einer Nachricht sicher und vermeidet dabei Duplikate. QoS 2 stellt somit die höchste Qualitätsstufe der Übertragung dar und erfordert damit auch mehr Komplexität und Rechenleistung in Client und Broker.

4.2.5 Broker

Broker verbinden die verschiedenen Clients und dienen als Vermittler der Nachrichten. Sie nehmen Nachrichten von Clients entgegen und senden sie an andere Clients, welche das *Topic* der Nachricht abbonniert haben. Broker berücksichtigen dabei die QoS-Stufe der Nachricht und müssen bei entsprechender QoS-Stufe Nachrichten auch Zwischenspeichern.

4.3 Verfügbare MQTT-Broker

4.3.1 Mosquitto

Mosquitto ist ein quelloffener MQTT-Broker und wurde unter der BSD-Lizenz veröffentlicht. Für verschiedene Plattformen und Betriebsysteme stehen vorkompilierte Pakete als Download oder in Paketmanagern zur Verfügung.[14]

Der Broker kann auf einem eigenen Server installiert werden, setzt somit aber bestehende Infrastruktur voraus.

Mosquitto speichert Daten im Arbeitspeicher und persistiert die Daten periodisch auf den Datenträger. [ˆfn_mosquitto_ autosave_interval]

Weiterhin kann Mosquitto sich mit weiteren Brokern via *Bridge* verbinden.[15]

[12] Siehe Quelle [ˆfn_observer_pattern].
[13] Die in der Quelle (“What Is MQTT and How Does It Work with WebSphere MQ?”) angebene Seite zeigt eine Übersicht über MQTT und die Verschiedenen QoS-Level.
[14] Siehe (“Mosquitto Homepage”)
[15] Siehe (“Mosquitto Bridges Options”)

4.3.1.1 Verschlüsselung und Authentifizierung

Mosquitto erlaubt Zertifikat-basierte Veschlüsselung mittels TLS/SSL. Der Server weisst dabei an den Client ein Zertifikat aus, welches der Client verifiziert. Umgekehrt besteht die Option, dass der Client sich gegenüber dem Server ebenfalls mit einem Zertifikat authentifizieren muss. Dies kann dazu genutzt werden, einen User zu identifzieren. Ohne Client-Zertifikat ist auch eine Authentifizierung mittels Benutzername und Passwort möglich. Statt eines Zertifikates kann auch ein *pre-shared-key*-Verfahren genutzt werden, wobei vorgängig ein Schlüssel ausgetauscht wird.[16]

Nutzt mann die Client-Authentifizierung, so ist es möglich, die Zugriffsrechte eines Clients auf einzelne Topics einzuschränken. Dabei können für ein Topic reine Lese- und Schreibrechte oder Beides vergeben werden.[17] Es ist darüber hinaus auch möglich, nicht-authentifizierte Benutzer auszuschliessen.

Mosquitto unterstützt weiterhin die Kommunikation über Websockets, d.h. es kann via Websockets direkt mit dem MQTT-Broker kommuniziert werden.

4.3.2 HiveMQ

HiveMQ ist ein laut Hersteller für Unternehmen optimierter MQTT-Broker und zeichne sich durch hohe Performance, gute Skalierbarkeit (durch Clustering), hohe Sicherheit, sowie 100%iger MQTT 3.1.1 Unterstützung überzeugen.

Wie Mosquitto untertützt HiveMQ Kommunikation über Websockets, (X509) Zertifikat-Authentifizierung, sowie Bridging.

HiveMQ ist kostenpflichtig, es stehen monatliche Lizenzen oder einmalige Lizenzen zur Verfügung. Die Kosten für die einmalige Lizenz bewegen sich zwischen 325 € und 6250 €. Eine kostenlose Testversion steht ebenfalls zur Verfügung.

4.3.3 Moquette

Moquette ist ein Quelloffener MQTT-Broker, welcher in Java implementiert ist. Er steht unter der Apache License 2.0.

Der Broker wird aktiv weiterentwickelt, unterstützt aber weniger Funktionen von MQTT als Mosquitto oder HiveMQ. So ist es u.a. noch nicht möglich, Zugriffsrechte für einzelne User zu setzen. QoS 0,1 und 2 werden unterstützt.[18]

4.3.4 Mosca

Dieser Broker ist als *node.js*-Applikation und über den *Node Package Manager* (*npm*) erhältlich. Er kann direkt als Broker genutzt werden oder in einer anderen *node.js*-Applikation als Modul genutzt werden.

Mosca ist MQTT 3.1 kompatibel, unterstützt aber nur QoS 0 und 1. Nachrichten können auf einer MongoDB oder Redis-Datenbank persistiert werden.

Der Quellcode von Mosca ist unter Github veröffentlicht, es ist allerdings keine Lizenz angegeben.[19]

4.3.5 GnatMQ

GnatMQ ist ein auf dem .NET-Framework von Microsoft basierender MQTT-Broker. Er unterstützt alle 3 QoS-Stufen, Authentifizierung via Benutzername/Passwort, sowie Zugriffskontrolle, jedoch noch keine SSL/TLS-Verbindunng und kein *Bridge*-Modus.

[16] Siehe (“Mosquitto Authenfication”)
[17] Siehe Option * acl_file* in Quelle (“Mosquitto General Options”)
[18] Siehe (“Github Moquette”)
[19] Siehe (“Github Mosca”)

4.3.6 Gehostete MQTT-Broker

Statt einer selbst verwalteten Lösung, kommen auch Cloud-Dienste in Frage (*software as a service*).

4.3.6.1 CloudMQTT *CloudMQTT* ist ein gehosteter MQTT-Broker. Es gibt ein kostenloses Abonnement und zwei kostenpflichtige ($19, resp. $99 pro Monat).

Der Dienst unterstützt u.a. Authentifizierung, Zugriffskontrolle (*ACL*), *Bridge*-Modus, sowie Kommunikation über Websockets.

20

4.3.6.2 Self-Hosted gegen Cloud-Lösung Bei Standort-Daten handelt es sich um sensible Daten. Diese besitzen auch einen gewissen *Business Value*, was bei Standort-basierter Werbung zu sehen ist. Bei Cloud-Lösungen wird die Kontrolle über diese Daten an einen Fremdanbieter abgegeben. Im Rahmen dieser Arbeit ist dies zweitrangig; in einer produktiven Anwendung ist dies aber nicht zu vernachlässigen.

Cloud-Lösungen bieten aber den Vorteil des einfacheren Setup. Skalierung und Betrieb kann ebenfalls an den Anbieter abgegeben werden.

4.3.7 Wahl

Alle untersuchten Broker kämen prinzipiell in Frage (bei den Kostenpflichtigen in einer Test-Version), die Wahl fiel jedoch auf Mosquitto, da dieser sehr gut dokumentiert ist und sehr viele MQTT-Features unterstützt. Somit kann sich mit vielen Aspekten von MQTT auseinandergesetzt werden.

Für die Installation steht u.a. eine (virtuelle) Ubuntu Instanz (12.04 LTS) in einem Rechenzentrum zur Verfügung. Mosquitto kann direkt über Ubuntu/Debian's Paketmanager *apt* installiert werden.

[20] ("CloudMQTT")

5 Konzept & Design

5.1 Sicherheitsaspekte

5.1.1 Location Privacy

Standortdaten sind Sicherheitsrelevant. Es handelt sich um sensible Daten, die insbesondere vor Missbrauch geschützt werden müssen.

Der Schutz der *Location Privacy*, d.h. der Schutz der privaten Standortdaten ist spätestens seit dem Aufkommen GPS-fähiger Smartphones ein wichtiges Thema.

Zum Schutz der Standortdaten im Kontext der sogenannten *Vorratsdatenspeicherung* in Deutschland schrieb der *Chaos Computer Club* 2009:

> "Durch die ungebremste Aufzeichnung der digitalen Spuren wird das Mobiltelefon mehr und mehr zu einer Ortungswanze, sofern dem speicherwütigen Staat nicht Einhalt geboten wird. Sollte die Vorratsdatenspeicherung vor Gericht Bestand haben, bedeutet das praktisch ein Ende der Freiheit, unbeobachtet und ungestört zu leben"

(Frank Rieger, "Chaos Computer Club Veröffentlicht Stellungnahme Zur Vorratsdatenspeicherung")

6 Jahre säpter bleibt der Schutz der Standortdaten ein aktuelles Thema, die Universität Zürich hat dazu (Stand Mai 2015) eine Masterarbeit ausgeschrieben.[21]

5.1.2 Verschlüsselung

Das Verschlüsseln der Übertragung zwischen Endgeräten dient dem Schutz der Daten vor dem Einblick oder Manipulation durch Dritte.

Eine Verschlüsselung kann *Ende-zu-Ende* über die *gesamte Übertragungsstecke* erfolgen, d.h. beispielsweise von der Aufzeichnung der Daten auf dem Mobil-Gerät, bis zur Anzeige, resp. Auswertung der Daten auf der endgültigen Anwendung.

Die Übertragung kann auch nur Streckenweise veschlüsselt werden. *SSL/TLS* verschlüsselt beispielsweise die TCP/IP-Verbindung zwischen dem Mobil-Gerät und dem MQTT-Broker, der Broker selbst "sieht" die Daten allerdings im Klartext. MQTT sieht selbst keine Veschlüsselung innerhalb des Protokolles vor, es ist jedoch mit einem eigenen Verfahren möglich die Anwendungsdaten zu verschlüsseln. Ohne eine solche Verschlüsselung ist ein MQTT-Broker aber eine sicherheitsrelevante Stelle. Sensible Daten müssen also über einen vertraulichen Broker gesendet werden. Auch Übergänge von MQTT in ein anderes Protokoll (z.b. Websockets) sind Sicherheitsrelevant. Die entsprechenden Protokolle müssen eine gleichwertige Verschüsselung benutzen (z.b. *HTTPS*).

5.1.3 Authentifizierung

Neben der *Vertraulichkeit* und der *Integrität* der Daten ist ferner die *Authentizität*, d.h. die Echtheit der Daten, sowie die *Verbindlichkeit* und *Zurechenbarkeit* von Bedeutung.

Jemand, der den Dienst nutzt, muss sich also ausweisen können. Dies kann über Benutzername und Passwort, über einen Schlüssel oder ein Zertifikat erfolgen.

Für die zu erstellende Anwendung soll ein Benutzername-Passwort-Verfahren zur Anwendung kommen.

[21] Siehe ("Masterarbeit - Location Privacy - Probleme Und Lösungen")

5.2 Architektur

Die Standort-Daten des Benutzers werden nicht in Echtzeit, sondern mit Verzögerung benötigt, um nachträglich den Standort-Verlauf des Benutzers zu rekonstruieren. Es ist also nötig, die Daten zwischenzuspeichern.

Da die Anwendung *TimeTraces* ein Server-Teil hat, könnte dieser für die Standortdatenspeicherung in Frage kommen.

Dagegen spricht aber, dass *TimeTraces* aus konzeptuellen Überlegungen keine Event-Daten speichert (siehe Abschnitt 2.1.2). Möchte man dieses Konzept beibehalten, so muss das Speichern der Daten auf eine andere Anwendung verlagert werden.

Die Broker-Software *Mosquitto* selbst kommt dafür ebenfalls nicht in Frage, da MQTT mit QoS 1 und 2 zwar Nachrichten persistiert, für eine ein- oder mehrwöchige Zwischenspeicherung ist ein Broker aber nicht ausgelegt.

5.3 Standort-Service als Brücke

Um dieses Problem zu lösen, kann ein Service erstellt werden, der zwischen *TimeTraces* und dem MQTT-Broker *Mosquitto* vermittelt. Da *TimeTraces* auf *Meteor* basiert, bietet es sich an, diesen Brücken-Dienst ebenfalls als Meteor-Applikation zu erstellen und zu *TimeTraces* via *DDP* zu kommunizieren.[22] Prinzipiell können die Standort-Daten auch über eine REST-Schnittstelle an *TimeTraces* übermittelt werden, da sie nur in eine Richtung gesendet werden. Da es sich aber bei *MQTT* und *DDP* Message-Oriented-Middleware handelt mit dem Publish-Subscribe-Muster, bietet es sich an, *DDP* für die Übertragung zu verwenden.

5.4 Konzeption eines Standort-Services

5.4.1 Rollen

Der Standortservice benötigt zwei Rollen: der *Benutzer*, welche den Dienst nutzt und den *Administrator*, welcher den Dienst betreut und betreibt.

Weiterhin findet eine *Mashine-to-mashine*-Kommunikation mit dem vorgelagerten MQTT-Broker und der nachgelagerten Anwendung *TimeTraces* statt.

5.4.2 User-Stories

(TODO): Story-Cards

Sc-01	Als Benutzer möchte ich mich für den Service registrieren können
Story	Neue Benutzer sollen sich registrieren können, damit sie sich später anmelden können
Akzeptanzkriterien:	☐ User kann Email angeben ☐ User kann Passwort angeben ☐ User TODO
Priorität	MUST

[22] Eine Evaluation möglicher Technologien für einen solchen Service ist nicht Teil dieser Arbeit. Die Wahl der Technologie basiert hier auf Vorwissen und Erfahrung des Autors.

Sc-02	Als Benutzer möchte ich mich an den Service anmelden können
Story	Registrierte Benutzer können sich anmelden
Priorität	MUST

Sc-03	Als Benutzer möchte ich meine Standort-Protokoll-Anwendung (z.b. *Owntracks*) mit dem Dienst verbinden können
Story	Der Benutzer kann seine Protokoll-Anwendung so konfigurieren, dass sie den Standort an den Dienst sendet
Priorität	MUST

Sc-04	Als Benutzer möchte ich meine Standortdaten über eine Schnittstelle abfragen können
Story	Der Service soll Schnittstellen zur Verfügung stellen, mit welcher ein Anwendungen das Standort-Protokoll des Benutzers abfragen können
Priorität	MUST

Sc-05	Als Benutzer möchte ich Standort-Einträge löschen können.
Story	Bestehende Einträge des Standort-Protokolles sollen sich permanentl öschen lassen können
Priorität	MAY

5.4.3 System

5.4.4 Sicherheitsaspekte

Der Dienst kommuniziert mit zwei weiteren Diensten und benötigt daher auch zwei Übertragungskanäle, welche potentielle Sicherheitsrisiken bergen.

Beide Kanäle müssen über eine aktuelle Technologie verschlüsselt werden.

5.4.4.1 Datenschutz Der Dienst speichert die schützenswerten Standort-Daten und muss entsprechend vor Missbrauch geschützt werden.

(TODO NFR):

- Ein angemeldeten Benutzer kann nur seine Standort-Daten auslesen und bearbeiten
- Ein nicht-angemeldeter Benutzer kann keine Daten sehen
- Die Zugriffsdaten für einen Benutzer müssen hinreichend geschützt sein.

6 Umsetzung

TODO

6.1 Screenshots

TODO

7 Diskussion

TODO

8 Ausblick

8.1 Indoor-Standorte mittels Beacons

8.2 Generischer Event-Service

A Annotierter Quelltext

Nachfolgend der Quellcode des Location-Services im *Literate CoffeeScript*-Stil.[23]

(app.coffee.md)

A.1 Simple location service with Meteor

This app connects to a mqtt-broker that will receive location-data from “OwnTracks” and will expose it in

- a table (using aldeed:tabular)
- a google map
- a DDP-API
- a REST-API (using nimble:restivus)

A.1.1 Collections

First we need a collection to store the locations of the users.

```
Collections =
    Locations: new Meteor.Collection "Locations"
```

We define now a shared helper functions to get the topic string for a user or vis-verca. The topics have the pattern location/:userId

```
Topic =
    getTopicForUser: (user) -> "location/#{user._id}"
    getUserIdForTopic: (topic) -> topic.split("/")[1]
    getUserForTopic: (topic) -> Meteor.users.findOne
        @getUserIdForTopic topic
```

A.1.2 DDP-API

We publish the data, so that a ddp client can subscribe to the data with subscribe “myLocations”, from: Date, to: Date

Note: we do not need publishs for the client-view, because aldeed:tabular handles that for us.

```
if Meteor.isServer then Meteor.publish "myLocations", (params)->
    selector = userId: @userId
    if params?.from?
        selector.tst ?= {}
        selector.tst["$gte"] = params.from
    if params?.to?
        selector.tst ?= {}
        selector.tst["$lte"] = params.to
    Collections.Locations.find selector
```

[23] Siehe (“Literate CoffeeScript”)

A.1.3 REST-API

We provide additionally a REST-API using nimble:restivus. Here are the settings:

```
if Meteor.isServer
    Restivus.configure
        useAuth: yes
        prettyJson: yes
    Restivus.addCollection Collections.Locations,
        routeOptions:
            authRequired: yes
```

A.1.4 MQTT-Bridge

We now connect the mqtt-broker with our collection.

First, start a mqtt connection to the broker as soon as the server starts:

```
if Meteor.isServer then Meteor.startup ->
    mqttClient = mqtt.connect "mqtts://lab.macrozone.ch:8883",
        clientId: "location.macrozone.ch-#{ process.env.NODE_ENV }"
        rejectUnauthorized: no
    mqttClient.on "error", Meteor.bindEnvironment (error) -> console.
       log error
```

We initialize a geocoder (google by default), that will resolve our position to adresses.

```
    geoCoder = new GeoCoder
```

A.1.4.1 Handle messages

We listen now to messages from the broker.

```
    mqttClient.on "message", Meteor.bindEnvironment (topic, data) ->
```

Owntracks sends data as JSON-Strings, so it needs to be decoded. The timestamp ‘tst’ gets converted to a Javascript-Date. We use the old timestamp as the id for the message to prevent duplicates:

```
        message_id = data.tst
        data = JSON.parse data.toString "utf-8"
        data.tst = new Date parseInt(data.tst,10)*1000
```

Find the User for the topic “location/:userId”, decode the position (lat, lon) to an adress with the geocoder and insert it into “Locations”

```
        user = Topic.getUserForTopic topic
        if user? and data._type is "location"
            data.userId = user._id
            data.geo = _.first geoCoder.reverse data.lat, data.lon
            Collections.Locations.upsert message_id, data
```

A.1.4.2 Handle mqtt-subscriptions for users

We subscribe for every user and stop subscriptions for removed users. We start observing users after (re-) connect to the broker. In case of re-connect, stop the old observation

```
    userObserveHandle = null
    mqttClient.on "connect", Meteor.bindEnvironment ->
        console.log "connected to mqtt-broker"
        userObserveHandle?.stop()
```

```
        userObserveHandle = Meteor.users.find().observe
            added: (user) ->
                startSubscriptionForUser user
            removed: (user) ->
                stopSubscriptionForUser user
```

Helpers to start and stop subscriptions on the mqttClient

```
    startSubscriptionForUser = (user) ->
        startSubscription Topic.getTopicForUser user
    stopSubscriptionForUser = (user) ->
        stopSubscription Topic.getTopicForUser user
    startSubscription = (topic) ->
        console.log "subscribed to #{topic}"
        mqttClient.subscribe topic
    stopSubscription = (topic) ->
        console.log "unsubscribed to #{topic}"
        mqttClient.unsubscribe topic
```

A.1.5 Client Views

A.1.5.1 Tabular The package aldeed:tabular will provide us with a data-table-like html-component. We use it to show the location history to the user. It is included in app.jade as `+tabular`. Subscriptions are handled automatically, depending on the filters on the table.

```
TabularTables =
    Locations: new Tabular.Table
        name: "Locations"
        collection: Collections.Locations
        order: [[0, "desc"]]
        columns: [
            {
                data: "tst"
                title: "Time"
                render: (val) ->
                    moment(val).calendar()
            },
            {
                data: "geo"
                title: "Address"
                render: (geo) ->
                    "#{geo.city}, #{geo.streetName} #{geo.streetNumber
                        }" if geo?
            }
            {data: "lat", title: "Latitude", width: "80px"}
            {data: "lon", title: "Longitude", width: "80px"}
            {data: "batt", title: "Battery", width: "40px"}
        ]
```

Restrict the data to the current user.

```
        selector: (userId) -> {userId}
```

We define additionally some template-helpers on the client view

```
if Meteor.isClient
    Template.registerHelper "dateFormat", (date) ->
        (moment date).format("YYYY-MMM-DD HH:mm")
    Template.registerHelper "topicForCurrentUser", ->
```

```
5           Topic.getTopicForUser Meteor.user()
6       Template.registerHelper('TabularTables', TabularTables)
```

A.1.5.2 Routes We define a route to show the user the current location history and where unauthorized users can login or register. This will map to the template defined in app.jade

```
1 Router.route "/",
2     template: "locations"
3     layoutTemplate: "layout"
```

A.1.5.3 Google Maps We additionally show the current published locations in a google-map. We use the package dburles:google-maps for that.

```
1  if Meteor.isClient
2      Template.locations.helpers
3          googleMapsOptions: ->
4              if GoogleMaps.loaded()
5                  zoom: 8
6                  center: new google.maps.LatLng(-37.8136, 144.9631)
7
8      Template.locations.onCreated ->
9          GoogleMaps.load()
10         GoogleMaps.ready "locationsMap", (map) =>
11             markers = {}
12             bounds = new google.maps.LatLngBounds
13             shouldSetBounds = no
14             getPosition = (location) -> new google.maps.LatLng
                   location.lat, location.lon
```

We now observe now the current data in the Locations-Collection and add markers to google maps for them

```
1              observeHandle = Collections.Locations.find().observe
2                  added: (location) ->
3                      position = getPosition location
4                      markers[location._id] = new google.maps.Marker
5                          position: position
6                          map: map.instance
7                          title: "#{location.tst} - #{location.geo?.city
                               }, #{location.geo?.streetName}, #{location
                               .geo?.streetNumber}"
8                      bounds.extend position
9                      map.instance.fitBounds bounds
10                 changed: (location, oldLocation) ->
11                     position = getPosition location
12                     markers[location._id].setPosition position
13                     bounds.extend position
14                 removed: (location) ->
15                     markers[location._id].setMap null
16                     delete markers[location._id]
17
18             Template.locations.onDestroyed -> observeHandle.stop()
```

A.2 File app.jade

```
1
2  template(name="locations")
3      .row.locations
4          .map.col-sm-5
5              +googleMap(name="locationsMap" options=googleMapsOptions)
6          .locations.col-sm-7
7              +tabular(table=TabularTables.Locations class="table table-
                   striped table-bordered table-condensed")
8
9  template(name="layout")
10     .container-fluid
11         +loginButtons
12         if currentUser
13             +userInfo
14             +yield
15         else
16             .container
17                 .alert.alert-warning
18                     i.glyphicon.glyphicon-log-in
19                     |  Please login or register.
20
21 template(name="userInfo")
22     p Owntracks-Topic: #{topicForCurrentUser}
```

A.3 File app.styl

```
1
2  .map
3      height: 400px
```

Quellenangaben

“Chaos Computer Club Veröffentlicht Stellungnahme Zur Vorratsdatenspeicherung.” http://www.ccc.de/updates/2009/vds-gutachten.

“CloudMQTT.” http://www.cloudmqtt.com/.

“Field ‘CurrentLocation’ Is Not Provided in API Result (or Empty), While the Field Is Cleary Filled on the Page.” https://code.google.com/p/google-plus-platform/issues/detail?id=620.

“Github Moquette.” https://github.com/andsel/moquette.

“Github Mosca.” http://mcollina.github.io/mosca/.

“Google+ Platform - People (Personen).” https://developers.google.com/+/api/latest/people.

“Latitude Wurde Eingestellt.” https://support.google.com/gmm/answer/3001634?hl=de.

“Literate CoffeeScript.” http://coffeescript.org/#literate.

“Masterarbeit - Location Privacy - Probleme Und Lösungen.” http://www.ifi.uzh.ch/isr/teaching/masterarbeiten/locationprivacy.html.

“Mosquitto Authenfication.” http://mosquitto.org/man/mosquitto-conf-5.html#idp49111296.

“Mosquitto Bridges Options.” http://mosquitto.org/man/mosquitto-conf-5.html#idp49322480.

“Mosquitto General Options.” http://mosquitto.org/man/mosquitto-conf-5.html#idp49117824.

“Mosquitto Homepage.” http://mosquitto.org/.

“MQTT - Frequently Asked Questions.” http://mqtt.org/faq.

“MQTT for Sensor Networks – MQTT-SN.” http://mqtt.org/tag/mqtt-s.

“OwnTracks.” https://github.com/owntracks/owntracks/wiki.

“OwnTracks Lizenz.” https://github.com/owntracks/android/blob/master/LICENSE.

Wettstein, Marco. “TimeTraces - Seminararbeit ‘Entwickeln von Anwendungen Für Hand Held’.”

“What Is MQTT and How Does It Work with WebSphere MQ?” https://www.ibm.com/developerworks/mydeveloperworks/blogs/aimsupport/entry/what_is_mqtt_and_how_does_it_work_with_websphere_mq?lang=en.

“Wikipedia - MQ Telemetry Transport.” http://de.wikipedia.org/wiki/MQ_Telemetry_Transport.

. http://mqtt.org/.