Fachhochschule Südwestfalen
University of Applied Sciences

Deutsches Zentrum für Luft- und Raumfahrt
DLR

Aufbau einer Kommunikations-Infrastruktur auf Basis von MQTT

Bachelorarbeit im Studiengang Angewandte Informatik
der Fachhochschule Südwestfalen

Lukas-Marius Biskoping

21.10.2013

Erstgutachter: Prof. Dipl.-Ing. Dipl.-Ing. Ulrich Lehmann
Zweitgutachter: M. Sc. Doreen Seider

Sommersemester 2013

Inhalt

1 Das Projekt Meetify 7

1.1 Was ist Meetify 7

1.2 Motivation für diese Arbeit 7

1.3 Funktionsbeschreibung von Meetify 8

2 Technologien und Frameworks in Meetify 10

2.1 Zuvor eingesetzt 10
2.1.1 Django 10
2.1.2 REST 10

2.2 Neu eingesetzt 11
2.2.1 MQTT 11
2.2.2 SQLlite3 und SQLAlchemy 14

3 Eingesetzte Programmiersprachen 15

4 Relevante alte Kernelemente von Meetify 16

4.1 Meetify-Android-App 16
4.1.1 Account-, Meeting- und Requestclient 16
4.1.2 Participant und Moderator 16
4.1.3 REST-Konstanten 17

4.2 Meetify-REST-Backend 18
4.2.1 Account-, Meeting- und Requestverwaltung 18
4.2.2 HTTP-Client 18
4.2.3 Datenhaltung und persistente Speicherung 18

4.3 Überblick Systemaufbau 19

5 Neue und veränderte Elemente von Meetify 20

5.1 Meetify-Android-App 20
5.1.1 Account-, Meeting- und Requestclient 20
5.1.2 MQTT-Client 21
5.1.3 Participant und Moderator 21
5.1.4 MQTT-Konstanten 22

5.2 Meetify-MQTT-Backend 22
5.2.1 Account-, Meeting- und Requestverwaltung 22
5.2.2 Datenhaltung und persistente Speicherung 23
5.2.3 MQTT-Client 24
5.2.4 MQTT-Broker 25

5.3 Überblick Systemaufbau 25

6 Kommunikationsablauf REST 26

6.1 Kommunikation zwischen Meetify-Client und Backend 26

6.2 Beschreibung der Datenpakete 27

6.3 Aktualisierung der Requestliste 28

7 Kommunikationsablauf MQTT 30

7.1 Kommunikation zwischen Meetify-Client und Backend 30

7.2 Beschreibung der Datenpakete 31

7.3 Aktualisierung der Requestliste 33

8 Netzlastanalyse anhand eines simulierten Meetings 36

8.1 Beschreibung und Ablauf des simulierten Meetings 36

8.1.1 Aufbau und Vorgaben für das Meeting 36

8.1.2 Zeitlicher Ablauf des Meetings 36

8.2 Netzlastanalyse anhand der verschickten Pakete bei REST 38

8.3 Netzlastanalyse anhand der verschickten Pakete bei MQTT 40

8.4 Vergleich der Netzlasten 42

8.5 Netzlastabhängigkeit bei MQTT und REST 42

9 Verbesserungsmöglichkeiten 44

9.1 Verschlüsselung der Nachrichten 44

9.2 Accountsystem mit Zugriffsrechten 44

9.3 Verringerung der Netzlast durch Optimierung des Kommunikationsablaufes 44

10 Zusammenfassung 45

11 Literaturverzeichnis 46

12 Abbildungsverzeichnis 47

13 Danksagung 48

14 Eidesstattliche Erklärung 49

15 Anhang 50

15.1 Anhang A: Durchgeführte Testszenarien 50
15.1.1 Testumgebung: 50
15.1.2 Durchgeführte Tests: 50

15.2 Anhang B: Projektdateien 53

1 Das Projekt Meetify

In dem Projekt Meetify ist vor Beginn dieser Arbeit eine Kommunikationsinfrastruktur auf Basis von REST implementiert.

Diese wurde im Laufe dieser Arbeit durch eine Infrastruktur auf Basis von MQTT ersetzt.

1.1 Was ist Meetify

Meetify ist eine mobile Applikation für Android und iOS, welche in einer Gruppendiskussion einen fairen Ablauf gewährleisten soll. Wird eine Gruppendiskussion mithilfe von Meetify moderiert, können alle Teilnehmer über Meetify einen Sprachwunsch äußern. Der Moderator der Diskussion erhält eine zeitlich sortierte Liste der Sprachwünsche und kann, unterstützt durch Meetify, die Teilnehmer aufrufen.

Meetify wird entwickelt vom Deutschen Zentrum für Luft- und Raumfahrt.

1.2 Motivation für diese Arbeit

Die Kommunikation der App basiert auf dem Client-Server-Prinzip. Alle Meetify-Clients (Mobilgeräte vom Moderator und den Teilnehmern) melden sich auf einem zentralen Server an und rufen die gewünschten Daten ab.
Der zentrale Server wird in diesem Dokument Meetify-Backend genannt.

Da die im Meetify-Backend abgelegten Daten Änderungen unterworfen sind, müssen die Meetify-Clients diese regelmäßig neu abrufen. Für diese Kommunikation wurde sowohl in Meetify-Client als auch im Meetify-Backend eine REST-Schnittstelle implementiert.

Änderungen können zum Beispiel das An-, Abmelden von Meetify-Clients, Erstellen, Ändern oder Löschen eines Sprachwunsches, Meetings oder Nutzerprofils sein.

Für das regelmäßige Neuanfordern der Daten wurde das Polling-Prinzip umgesetzt. Dieses Prinzip sieht vor, dass jeder Meetify-Client in regelmäßigen Abständen die gewünschten Daten vom Meetify-Backend abruft.

Da Änderungen der betroffenen Daten (in einer laufenden Diskussion lediglich alle paar Minuten) im Verhältnis zur Abruffrequenz/Pollingfrequenz (Abruf der Daten in jeder Sekunde) selten auftreten, ist Meetify aufgrund der hohen verursachten Netzlast, durch das Polling, nicht für einen produktiven Einsatz geeignet.

Das Ziel dieser Arbeit ist es daher die Kommunikation der Androidversion von Meetify vom Polling-basierenden auf einen Event-basierenden Ansatz umzustellen. Dabei wird eine neue Kommunikationsinfrastruktur auf Basis von MQTT aufgebaut.

1.3 Funktionsbeschreibung von Meetify

Ein Nutzer kann sich über die Meetify-Oberfläche ein Nutzerprofil anlegen, welches dann von Meetify genutzt wird, um sich am Meetify-Backend einzuloggen.

Dabei kann der Nutzer eine von zwei Rollen wählen: Moderator oder Teilnehmer.

Loggt sich ein Nutzer als Moderator in Meetify ein, kann er ein neues Meeting erstellen oder vom ihm bereits erstellten Meetings wieder beitreten, diese bearbeiten oder löschen.
Innerhalb eines Meetings erhält der Moderator dann alle auftretenden oder bereits aufgetretenen Sprachwünsche aller Teilnehmer des Meetings.
Die Meldungen sind in der Reihenfolge des zeitlichen Auftretens sortiert.
Der Moderator kann nun einzelne Meldungen aktivieren und somit „das Wort" der Diskussion an den betreffenden Teilnehmer des Meetings weitergeben. Weiterhin kann der Moderator Meldungen auch löschen. Dies gilt ebenfalls, wenn die Meldung bereits aktiviert wurde. Beim Ausloggen bleiben alle Meetings welche nicht durch den Nutzer gelöscht wurden und alle dazu gehörenden Meldungen erhalten.

Zum Einloggen als Teilnehmer in Meetify wird ein Kürzel eines bereits bestehenden Meetings benötigt.
Während des Einloggens tritt der Nutzer direkt dem Meeting bei, welches er durch das Kürzel angegeben hat.
Dieses Kürzel wird von Meetify beim Erstellen eines Meetings erzeugt und muss somit vom Moderator nach der Erstellung des Meetings verbreitet werden.
Nach einer kurzen Informationsseite des Meetings hat der Nutzer die Möglichkeit einen Sprachwunsch zu äußern.
Daraufhin erhält der Nutzer durch die Meetify-Oberfläche eine Information, wie viele andere Sprachwünsche noch vor dem eigenen liegen.

Weiterhin erhält der Nutzer eine Benachrichtigung, sobald sein Sprachwunsch aufgerufen oder gelöscht wird.
Mit dem Verlassen des Meetings wird ein eventuell noch bestehender Sprachwunsch des Nutzers gelöscht und der Nutzer direkt von Meetify abgemeldet.

Neben der eigentlichen App von Meetify, über die jegliche Benutzerinteraktion abläuft, existiert noch das Meetify-Backend. Dieses Meetify-Backend arbeitet auf einem zentralen Server und verwaltet jegliche User, Meetings und Sprachwünsche, welche durch Benutzerinteraktionen erstellt, abgerufen, bearbeitet oder gelöscht werden.

In diesem Dokument werden Sprachwünsche unter dem Begriff des Request gehandhabt.

2 Technologien und Frameworks in Meetify

Die hier aufgeführten Technologien und Frameworks sind lediglich die für diese Arbeit relevanten Bestandteile von Meetify und somit keine vollständigen Auflistungen aller in Meetify verwendeten Technologien und Frameworks.

2.1 Zuvor eingesetzt

2.1.1 Django

„Django ist ein in Python geschriebenes Framework, das die schnelle Entwicklung von Web-Applikationen ermöglicht. Dabei wird Wert auf sauberen Code und die Wiederverwendbarkeit einzelner Komponenten gelegt."
(Deutscher Django Verein e.V.)

Das Django-Framework wurde für das Meetify-Backend als Komplettlösung eingesetzt. In dem Framework wurden die Datenbank, Zugriffs-URLs, Nutzer-, Meeting- und Requestdaten definiert, verwaltet und gespeichert.

2.1.2 REST

Representational State Transfer, in Deutsch *Repräsentative Statusübertragung*, beschreibt ein Programmierparadigma für Webanwendungen zum Übertragen und Abfragen von Daten von einem Server.

Zur Kommunikation setzt das REST-Prinzip direkt auf dem **H**yper**t**ext **T**ransfer **P**rotokoll (HTTP) auf.
Der Kommunikationsablauf zwischen Meetify-Clients und dem Backend wurde auf dem REST-Prinzip aufgebaut.

Das REST-Prinzip sieht vor, dass ein Client als aktives Element der Kommunikation den Status eines Objektes vom passiven Element, dem Server, anfordert.
Dabei gibt REST eine Struktur für die genutzten URLs vor, sodass zu einem bestimmten angefragten Objekt immer eine bestimmte URL existiert.
Beispiel einer solchen URL:
http://meetify-backend.de/meetings/Meeting1

Über die HTTP-Methoden POST, GET, PUT, DELETE kann dann über die durch den Server zu erledigende Aufgabe unterschieden werden.
Die HTTP-Methoden stellen dabei in der Regel die übliche CRUD-Funktionalität (**C**reate, **R**ead, **U**pdate, **D**elete) dar.

Dabei muss jede Anfrage alle Informationen enthalten, die der Server zum Interpretieren der Anfrage benötigt.

(Sinngemäß übernommen aus der Quelle: (Bayer, 2002))

2.2 Neu eingesetzt

2.2.1 MQTT

2.2.1.1 MQTT-Kurzbeschreibung

Message **Q**ueuing **T**elemetry **T**ransport (MQTT) ist ein Maschine-zu-Maschine (M2M)/"Internet of Things" Verbindungs- und Kommunikationsprotokoll.

Es wurde als extrem schlankes Event-basierendes Protokoll entwickelt. Wobei das Hauptaugenmerk dabei auf Anwendungen lag, bei denen eine geringe Quellcodegröße benötigt wird oder die vorhandene Netzwerkbandbreite sehr begrenzt, teuer oder instabil ist.

Daher eignet sich MQTT, durch seine geringe Größe in Quellcode und Netzwerkpaketen, sehr gut für mobile Applikationen.

(Sinngemäß übersetzt aus der Quelle: (mqtt.org))

In Meetify wird die MQTT-Protokoll Version 3.1 verwendet.

2.2.1.2 MQTT-Funktionsweise

Bei der Verwendung des MQTT-Protokolls werden Nachrichten immer zwischen zwei Parteien verschickt. Die Parteien sind der MQTT-Client und der MQTT-Broker.

Zu Anfang verbindet sich der MQTT-Client mit einer eindeutigen ID mit dem MQTT-Broker. Dabei kann der MQTT-Broker so eingestellt werden, dass entweder alle sich verbindenden MQTT-Clients oder nur die MQTT-Clients mit einem gültigen Username/Passwort Login zugelassen werden.

Einmal verbunden kann ein MQTT-Client Nachrichten an den MQTT-Broker übermitteln. Diese Nachrichten werden dabei an sogenannte Topics gesendet.
Ein Topic kann zur einfacheren Vorstellung als Thema betrachtet werden, zu dem ein MQTT-Client eine Textnachricht „äußern" kann.

Ein Topic kann auch Untertopics besitzen. Topics sind vom Aufbau her ähnlich zu Systempfaden.
Beispiele:
- „Messwerte/2013/Januar/14"
- „Messwerte/2013/Dezember"

Dabei bilden „Januar" und „Dezember" jeweils ein eigenes Untertopic von „2013", welches wiederum ein Untertopic von „Messwerte" darstellt.

Eine Nachricht kann immer nur an ein bestimmtes Topic gesendet werden, dabei ist es auch möglich die Nachricht an jede Zwischenebene eines Topic-Pfades zu senden.
An das obige Beispiel angelehnt könnte daher eine Nachricht auch an „Messwerte/2013" gesendet werden.

Ein Topic wird erstellt, sobald die erste Nachricht das Topic erreicht, oder sobald die erste Subscription auf diesem Topic stattfindet.

Eine Subscription kann als Abonnement betrachtet werden und wird von den MQTT-Clients vorgenommen.
Dabei kann ein MQTT-Client gleichzeitig bei vielen Topics subscribed sein.

Sobald der MQTT-Broker eine Nachricht auf einem Topic erhält sendet er diese Nachricht weiter an alle zu dem betreffenden Topic subscribten MQTT-Clients.
Nachdem dies geschehen ist, löscht der MQTT-Broker die Nachricht wieder.

Durch Username/Passwort Logins kann auch bestimmt werden mit welchen Topics ein MQTT-Client in Berührung kommen darf. Dies betrifft sowohl das Subscriben als auch das Senden von Nachrichten.

Weiterhin ist es möglich eine Nachricht als „retained" (zurückbehalten) zu klassifizieren. Wird eine Nachricht, die als „retained" klassifiziert wurde, an ein Topic des MQTT-Brokers gesendet, wird sie wie eine normale Nachricht an alle subscribten MQTT-Clients verbreitet, danach aber nicht gelöscht.

Meldet ein neuer MQTT-Client eine Subscription auf einen Topic mit einer als „retained“ gekennzeichneten Nachricht an, so wird ihm die gekennzeichnete Nachricht direkt zugesandt. Unabhängig davon, ob zwischen der gekennzeichneten Nachricht und dem Subscriben des neuen MQTT-Clients noch andere nicht gekennzeichnete Nachrichten an denselben Topic gesendet wurden.
Dieser Mechanismus wird „The last known good“ genannt und sorgt dafür, dass ein neuer MQTT-Client sofort verwertbare Informationen erhält, anstatt auf die nächste Nachricht warten zu müssen. Dies ist zum Beispiel nützlich bei einer Messdatenverteilung, wodurch ein neuer MQTT-Client sofort auf einen aktuellen Stand gebracht werden kann, ohne auf neue Messdaten warten zu müssen.

Es kann in einem Topic immer nur eine Nachricht als „retained“ gekennzeichnet werden. Wird eine neue derart gekennzeichnete Nachricht an dasselbe Topic gesendet, so wird die alte Nachricht gelöscht.

MQTT verwendet beim Übermitteln der Nachrichten einen QualityofService (QoS). Dieser bestimmt mit welcher Zuverlässigkeit die Nachrichten beim Empfänger eingehen.

Die möglichen QoS-Verfahren sind:

– QoS-0
  Eine FireAndForget Methode, bei der die Nachricht abgesendet und danach vom Sender gelöscht wird. Dies ist die einfachste und sparsamste Methode, da lediglich ein Paket verschickt wird. Es gibt dabei aber keinerlei Sicherheit, dass die Nachricht auch beim Empfänger angekommen ist. Diese Methode kann von der neuen MQTT-Implementierung in Meetify genutzt werden.
  Netzwerklast: 1 Paket

– QoS-1
  Diese Methode stellt durch eine einfache Empfangsbestätigung sicher, dass die betreffende Nachricht beim Empfänger angekommen ist. Dadurch kann es aber vorkommen, dass eine Nachricht doppelt beim Empfänger eintrifft.
  Dieser Fall tritt ein, wenn die erste Nachricht so stark verzögert beim Empfänger eintrifft, dass die Wartezeit des Senders auf die Bestätigung abläuft und dieser die Nachricht ein zweites Mal abschickt.
  Diese Methode kann nicht in der aktuellen MQTT-Implementierung von Meetify genutzt werden, da keine explizite Duplikatsbehandlung implementiert wurde.
  Minimale Netzwerklast: 2 Pakete

- QoS-2
  Diese Methode stellt sicher, dass eine Nachricht genau einmal beim Empfänger angekommen ist. Dazu wird ein aufwendiges Handshakeverfahren genutzt. Diese Methode kann von der neuen MQTT-Implementierung in Meetify genutzt werden.
  Minimale Netzwerklast: 4 Pakete

  Weitere und detailliertere Informationen sind auf der Protokoll-Spezifikationsseite zu MQTT v3.1 von IBM (International Business Machines Corporation (IBM)) zu finden.

2.2.2 SQLlite3 und SQLAlchemy

SQLlite ist eine Bibliothek, welche eine selbständige, serverlose, konfigurationsfreie SQL-Datenbank-Engine bereitstellt.
SQLlite wurde als Datenbank-Layer gewählt, da es durch seine Konfigurationsfreiheit sehr effizient zu nutzen ist. Weiterhin bietet das serverlose Arbeiten von SQLlite die Möglichkeit einen großen Overhead an Systemressourcen im Meetify-Backend zu sparen und vereinfacht gleichzeitig den Umgang und die Implementierung des Meetify-Backends.
Außerdem speichert SQLlite die gesamte Datenbank in einer einzigen Datei im normalen Dateisystem und erleichtert auch hier die Verwaltung und erhöht die Systemunabhängigkeit des Meetify-Backends.

SQLAlchemy ist ein in Python geschriebenes Toolkit für eine objektrelationale Verarbeitung von SQL-Anfragen.

Diese beiden Technologien werden innerhalb von Meetify genutzt, um eine einfache, persistente Datenhaltung und gleichzeitig den vereinfachten Umgang mit Objekten im Meetify-Backend zu erhalten.

3 Eingesetzte Programmiersprachen

- Python in Version 2.7.5
  Python wurde zum Implementieren des Meetify-Backend gewählt, da diese Programmiersprache innerhalb des DLR weitreichend bekannt ist und somit die Wartbarkeit deutlich erleichtert.

- Java in Version 1.6
  Der Androidclient der Meetify-App wurde in Java 1.6 unter Verwendung der Android-API-17 (4.2.2) implementiert.

4 Relevante alte Kernelemente von Meetify

4.1 Meetify-Android-App

4.1.1 Account-, Meeting- und Requestclient

Zum Abrufen der verschiedenen Informationen vom Meetify-Backend, besitzt Meetify neben dem eigentlichen HTTP-Client drei weitere interne Datenschnittstellen: Den Accountclient, den Meetingclient und den Requestclient.

Jeder dieser internen Clients bietet Methoden zum Abrufen und Verwalten aller relevanten Daten eines bestimmten Accounts, Meetings oder Requests und greift dabei, zur Kommunikation mit dem Backend, auf den zur Verfügung stehenden HTTP-Client zurück.

Der grundlegende Ablauf der einzelnen Methoden ist dabei weitestgehend gleich.
Die aufgerufene Methode generiert und startet einen neuen, asynchron zur Oberfläche ausgeführten Task. Dieser Task ist dann zuständig für das Senden der Anfrage und Empfangen einer Antwort mittels des HTTP-Clients.
Dabei definiert die aufgerufene Methode mittels der an den Task übergebenen Parameter, an welche URL und mit welcher HTTP-Methode (POST, GET, PUT, DELETE) die zu übertragenden Daten übermittelt werden sollen.
Weiterhin wird dem Task vorgegeben, wie bei einer erfolgreichen Antwort reagiert werden soll. In einem Fehlerfall ist der Task bereits vorkonfiguriert, um entsprechende Meldungen an die Benutzeroberfläche zu senden.

4.1.2 Participant und Moderator

Participant (in Deutsch Teilnehmer oder Teilnehmer) und Moderator definieren die beiden unterschiedlichen Rollen in denen sich ein Meetify-Nutzer befinden kann.

Die Rolle Participant stellt dabei einen einfachen Teilnehmer eines Meetings dar und gibt dem Nutzer die Möglichkeit einen Sprachwunsch zu äußern. In Meetify kann es eine unbegrenzte Anzahl an Teilnehmern pro Meeting geben.

Die Rolle Moderator stellt den Ersteller eines Meetings dar. Diesem obliegt die Aufgabe auftretende Sprachwünsche der Teilnehmer zu verwalten.

Beide Rollen rufen an gegebener Stelle in jeder Sekunde die Methoden des Requestclient auf, um die vorhandenen Sprachwünsche eines Meetings zu aktualisieren und sind damit Ursprung des Polling und der hohen Netzlast.

4.1.3 REST-Konstanten

Diese Konstanten werden vom Accountclient, Meetingclient, Requestclient und HTTP-Client genutzt und beinhalten folgende Elemente:

- Angabe der Wartezeit zwischen zwei Request-Aktualisierungen
- Einen Timeout für die Verbindung zum Meetify-Backend
- Eine Vielzahl an URL-Vorgaben für die einzelnen Meetify-Objekte (User, Meetings, Requests)
- Eine Vielzahl Definitionen für die eindeutige Zuordnung von übermittelten Informationen
- Eine CSRF-Token Definition für den HTTP-Client

4.2 Meetify-REST-Backend

4.2.1 Account-, Meeting- und Requestverwaltung

Die empfangenen Informationen sind durch das Django-Framework an bestimmte Objekte (User, Meeting, etc.) gebunden. Die Anfragen werden entsprechend der URL, an welche sie geschickt wurden, ausgewertet und damit verbundene Aktionen ausgeführt.
Zum Beispiel das Erstellen eines neuen Meetings.
Daraufhin wird eine Antwort formuliert, welche wiederum durch das Django-Framework zurückgeschickt wird.

4.2.2 HTTP-Client

Das Django-Framework verwaltet eigenständig einen HTTP-Client, welcher zum Empfangen und Senden von Nachrichten durch das Django-Framework verwendet wird.

4.2.3 Datenhaltung und persistente Speicherung

Das Django-Framework übernimmt auch hier die gesamte Verwaltung einer persistenten, relationalen Datenbank und stellt zu den vorgegebenen Klassen (User, Meeting, Request, etc.) entsprechende Objekte bereit. Über diese Objekte können dann die nötigen Änderungen vorgenommen werden oder neue Objekte erstellt werden.

4.3 Überblick Systemaufbau

Wie in Abbildung 1 dargestellt, übernimmt im Meetify-Backend das Django-Framework sämtliche benötigten Funktionalitäten.
Im Meetify-Client befindet sich neben der Verwaltung ein HTTP-Client, welcher für die Kommunikation mit dem Backend zuständig ist. Die Verwaltung besteht dabei unter anderem aus dem unter Punkt 4.1.1 beschriebenen Accountclient, Meetingclient und Requestclient.

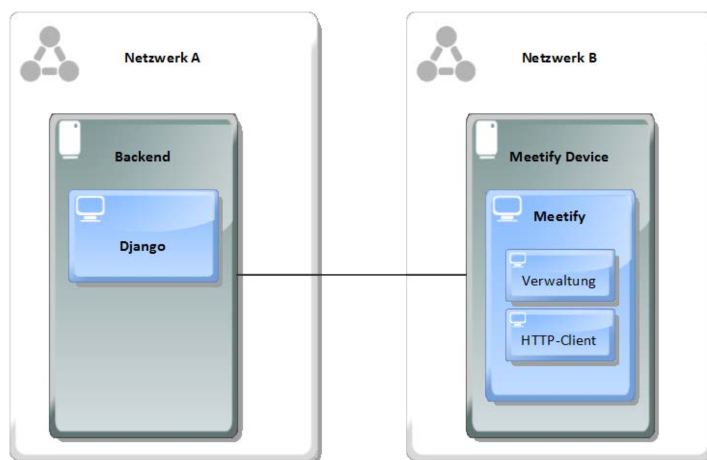

Abbildung 1: Systemaufbau REST

5 Neue und veränderte Elemente von Meetify

5.1 Meetify-Android-App

5.1.1 Account-, Meeting- und Requestclient

Die alten Clients wurden von ihrem Aufbau und der Funktionalität weitgehend übernommen.

Statt einer URL wird nun ein Topic-Pfad vorgeben, welcher an den Aufbau der ursprünglichen URLs angelehnt ist.

Die für das Aktualisieren der Requestliste zuständigen Methoden wurden umgeschrieben, um dem Event-basierten Ansatz zu entsprechen. So wird bei einmaligem Aufruf einer solchen Methode eine entsprechende Event-Topic-Subscription durchgeführt, welche bei jeder, für dieses Event relevanten, Änderung am Datenbestand reagiert. Der detaillierte Ablauf aktualisieren der Requestliste ist unter Punkt 6.3 beschrieben.

Weitere Änderungen mussten zum Extrahieren der Daten aus den Nachrichten vorgenommen werden.
Das Übermitteln von Daten konnte bei der REST-Implementierung über Key-Value Paare im HTTP-Kontext realisiert werden.
Da ein zusätzlicher Kontext bei MQTT-Nachrichten völlig entfällt, musste hier auf eine feste Formatierung des übermittelten Payload, also dem eigentlichen Nachrichtentext, zurückgegriffen werden.

So werden innerhalb einer Nachricht die Zuordnung (Key) und der Wert (Value) der zu übermittelnden Information durch ein vordefiniertes Trennzeichen und Key-Value Paare durch ein weiteres vordefiniertes Trennzeichen voneinander getrennt.
Diese Trennzeichen sind in den MQTT-Konstanten frei definierbar, müssen jedoch aus Zeichen oder einer Zeichenkette bestehen, welche nicht als Key oder Value in einer Nachricht auftreten können.

Ebenfalls angepasst werden musste der asynchron ausgeführte Task, welcher durch die einzelnen Methoden generiert und gestartet wird.
Dieser musste nun die zu versendenden Daten gemäß der oben beschriebenen Vorgaben aufbereiten und statt eines HTTP-Clients mit einem MQTT-Client arbeiten.

Der Ablauf beim Aufruf dieser Clients ist jedoch gleich geblieben.
Wird eine Methode dieser Clients aufgerufen wird ein neuer asynchroner Task generiert.
Ihm wird dann ein Zieltopic und die Anfrageart vorgeben. Er wird daraufhin mit den zu übermittelnden Daten gestartet. Die aufgerufene Methode setzt ebenfalls fest, wie der Task auf eine Antwort reagieren soll.
Im Fehlerfall gibt der Task entsprechendes an die Oberfläche weiter.

5.1.2 MQTT-Client

Als MQTT-Client wurde der Eclipse-Paho-MQTT-Client in Version 3 in eine Wrapperklasse implementiert.
Diese Wrapperklasse übernimmt die Konfiguration, das Verbindungsmanagement und steuert das Senden und Weiterleiten von empfangenen Nachrichten an die Verwaltung.

Für das Senden von Nachrichten kann ein QualityofService (QoS) angegeben werden.
Für diese Anwendung wurde das Qos-0 Verfahren gewählt, welches den geringsten Netzwerktraffic verursacht. Weitere Informationen zum QualityofService sind unter Punkt 2.2.1.2 aufgeführt

Das von Meetify verwendete QoS-Verfahren und diverse weitere Optionen können in den MQTT-Konstanten angepasst werden.

5.1.3 Participant und Moderator

Wie bereits unter Punkt 4.1.2 beschrieben, sind Participant (in Deutsch Teilnehmer oder Teilnehmer) und Moderator die beiden unterschiedlichen Rollen in denen sich ein Meetify-Nutzer befinden kann.

Die Rolle Participant stellt dabei einen einfachen Teilnehmer eines Meetings und die Rolle Moderator den Ersteller eines Meetings dar.

Die für das Aktualisieren der Requestliste zuständigen Methoden dieser beiden Rollen wurden umgeschrieben, um dem neuen Event-basierenden Ansatz zu entsprechen. Es wird also lediglich einmal die Event-Subscription aktiviert oder deaktiviert anstatt immer wieder eine Anfrage zu senden.

5.1.4 MQTT-Konstanten

Die Konstanten der REST-Implementierung wurden weitestgehend übernommen.

- Timeout für die Verbindungen
- Keep-Alive-Intervall
- Topic-Konventionen (angelehnt an die URL-Konvention aus der alten REST-Implementierung
- Eine Vielzahl Definitionen für die eindeutige Zuordnung von übermittelten Informationen (übernommen aus der alten REST-Implementierung, mit Ergänzungen)
- Diverse Parameter die für den MQTT-Client relevant sind (z.B.: Logindaten, QoS, Nachrichten-Konventionen)

5.2 Meetify-MQTT-Backend

Das Meetify-Backend wurde vollständig neu implementiert, da das Django-Framework ein reines HTTP-Framework darstellt und somit nicht weiterverwendet werden konnte.

5.2.1 Account-, Meeting- und Requestverwaltung

Angelehnt an das vorhergehende Verfahren wird hier auch nach dem verwendeten Topic und der verwendeten Methode einer Anfrage unterschieden, wie die erhaltenen Daten ausgewertet werden.
Die nötigen Aktionen werden daraufhin ausgeführt, z.B. ein neues Meeting erstellt, eine entsprechende Antwort generiert und zurückgeschickt.

Zusätzlich zum normalen Ablauf wurden Methoden eingeführt um bei Bedarf eine zusätzliche Aktualisierung der Requestliste des betreffenden Meetings vorzunehmen, wodurch der Eventmechanismus aktiviert und betreffende Meetify-Clients aktualisiert werden.

5.2.2 Datenhaltung und persistente Speicherung

Zur persistenten Speicherung wird eine SQLlite3 Datenbank verwendet. Mit Hilfe von SQLAlchemy werden aus vordefinierten Klassen sowohl das Datenbankmodell, als auch entsprechende Objekte erzeugt und verwaltet. Auf diese Objekte wird beim Bearbeiten der Anfragen zurückgegriffen.

Die hierfür verwendete Klassen- und Datenbankstruktur setzt sich wie in der nachfolgenden Abbildung 2 dargestellt zusammen:

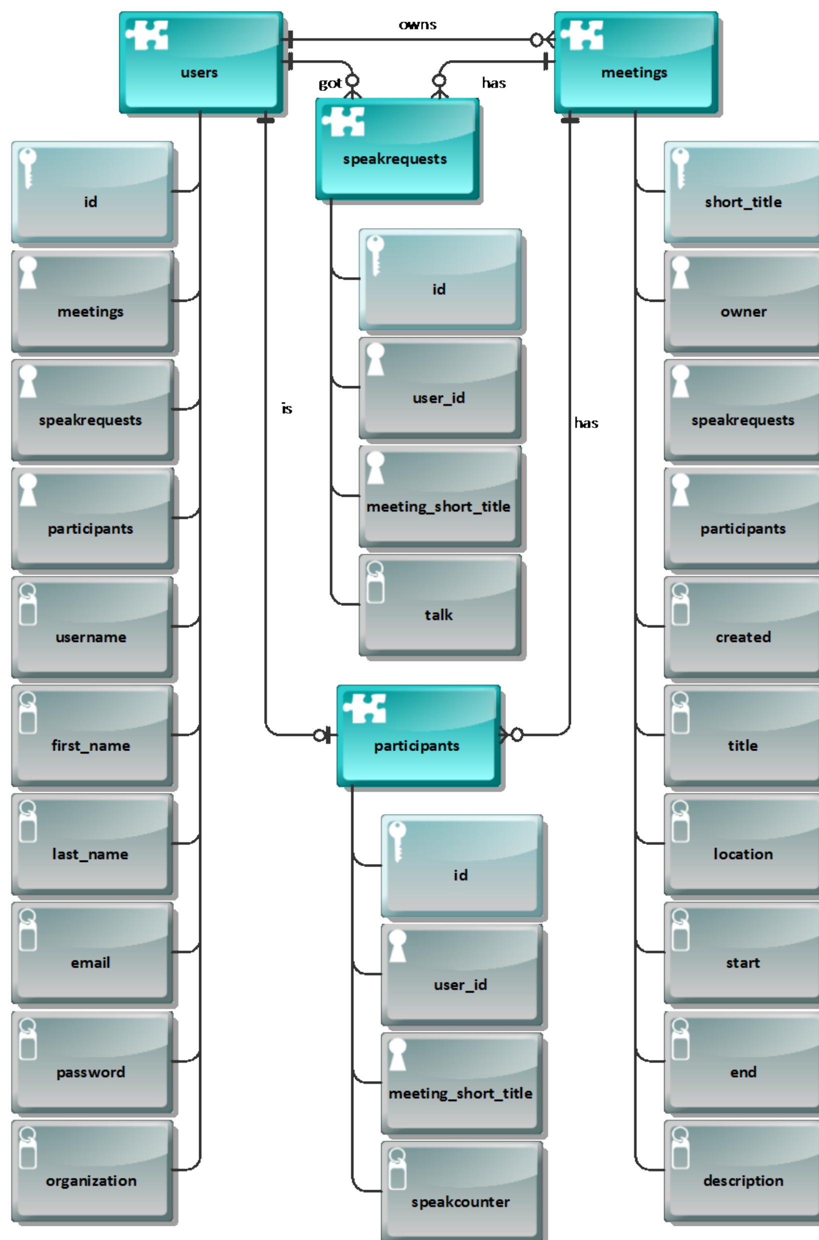

Abbildung 2: Datenmodell des MQTT-Backend

Die vorhandenen Klassen/Entitäten sind:

- users
  Ein Objekt dieser Klasse/Entität entspricht einem Meetify-Nutzer

- meetings
  Ein Objekt dieser Klasse/Entität stellt ein erstelltes Meeting dar

- participants
  Ein Objekt dieser Klasse/Entität stellt einen Teilnehmer eines Meetings dar

- speakrequests
  Ein Objekt dieser Klasse/Entität stellt ein Sprachwunsch innerhalb eines Meetings dar.

Das Datenbankmodell stellt folgende Beziehungen dar:
- Ein User kann der Besitzer von keinem, einem oder vielen Meetings sein.
- Ein User kann Teilnehmer von keinem oder einem Meeting sein
- Ein User kann keinen, einen oder viele Sprachwünsche haben
- Ein Meeting hat genau einen Besitzer
- Ein Meeting kann keinen, einen oder viele Teilnehmer haben
- Ein Meeting kann keinen, einen oder viele Sprachwünsche haben
- Ein Teilnehmer ist genau ein User und in genau einem Meeting
- Ein Sprachwunsch gehört genau einem User und zu genau einem Meeting

5.2.3 MQTT-Client

Als MQTT-Client wird eine Python-Implementierung des Mosquitto-MQTT-Client in Version 1.1.2 verwendet (Quelle: (Light)).
Diese wird durch entsprechende Funktionen so konfiguriert, dass eingehende Nachrichten abhängig von Topic und Anfragetyp an die Verwaltung weitergeleitet werden.

Auch dieser Client arbeitet mit den in dem Meetify-Client definierten Konstanten und einem QoS-0.

Zusätzlich steht der MQTT-Client unter der Kontrolle des Meetify-Backend, welches bei auftretenden Problemen reagiert.
Verbunden wird dieser MQTT-Client zu dem, auf dem gleichen Host laufenden, MQTT-Broker, welcher ebenfalls unter der Kontrolle des Meetify-Backend steht.

5.2.4 MQTT-Broker

Verwendet wird der Mosquitto-MQTT-Broker in Version 1.1.2 (Quelle: (Light). Dieser steht unter Kontrolle des Meetify-Backends.

Der MQTT-Broker ist vorkonfiguriert, so dass nur zulässige MQTT-Clients von Meetify sich verbinden können.
Der MQTT-Broker arbeitet vollständig eigenständig und wird nur im Fehlerfall vom Meetify-Backend automatisch neu gestartet.

5.3 Überblick Systemaufbau

Anstelle des Django-Frameworks befinden sich, wie in Abbildung 3 dargestellt, im Meetify-Backend nun mehrere einzelne Systeme:
- Eine durch SQLlite und SQLAlchemy verwaltete Datenbank
- Eine Verwaltung für die Abarbeitung der Anfragen
- Ein MQTT-Client zum Empfangen und Senden der Nachrichten
- Ein MQTT-Broker bei dem sich alle MQTT-Clients einloggen und welcher für die Verteilung der Nachrichten zuständig ist.

Im Meetify-Client hingegen wurde lediglich der HTTP-Client durch einen MQTT-Client ersetzt (siehe Abb. 3).

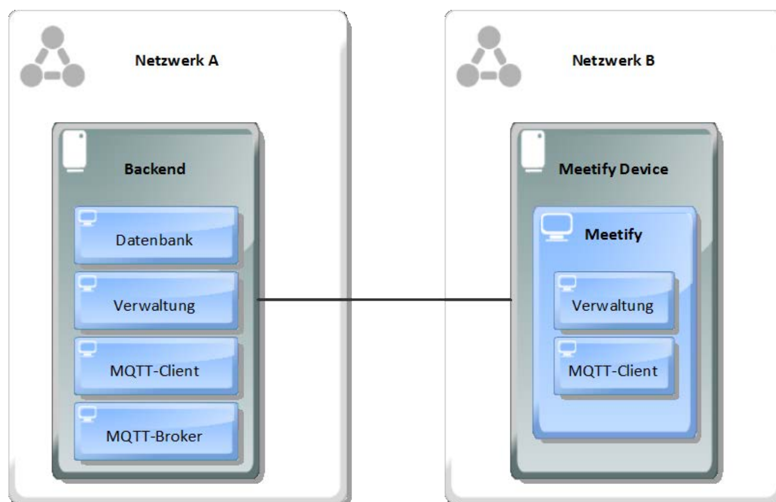

Abbildung 3: Systemaufbau MQTT

6 Kommunikationsablauf REST

6.1 Kommunikation zwischen Meetify-Client und Backend

Die Kommunikation zwischen einem Meetify-Client und dem Backend fand immer auf die gleiche Weise statt:

Ausgelöst durch eine Benutzerinteraktion mit der Meetify-Oberfläche, sendet der Meetify-Client eine entsprechende Anfrage an das Meetify-Backend.
Das Meetify-Backend arbeitet die für die Anfrage vorgesehenen Schritte ab und sendet eine Antwort zurück an den Meetify-Client (siehe Abb. 4).

Abbildung 4: Generelle Kommunikation

Dabei stellt das Meetify-Backend den passiven Teil und der Meetify-Client den aktiven Teil der Kommunikation dar, da das Meetify-Backend von sich aus nie aktiv wird.
Diese Vorgehensweise der Kommunikation entspricht der REST-Implementierungsstrategie.

Einzige Ausnahme bildet hier das Aktualisieren der Requestliste, wie unter Punkt 6.3 beschrieben. Dabei wird der Anfrage-Antwort-Prozess nicht bei einer Benutzerinteraktion, sondern jede Sekunde durchgeführt.

6.2 Beschreibung der Datenpakete

Die Datenpakete sind einfache HTTP-Request- und HTTP-Response-Pakete, in welchen die zu übermittelnden Werte als Key-Value-Paare im HTTP-Parameter-Kontext gespeichert sind.

In den HTTP-Request Paketen wird ebenfalls eine HTTP-Methode angeben, welche beeinflusst, wie das Backend mit den Daten umgeht.
Die Methoden sind POST, GET, PUT und DELETE.

Sofern benötigt, wurden die Methoden wie folgt verwendet:

- POST
  Zum Erstellen eines neuen Objektes

- GET
  Zum Ermitteln eines bestehenden Objektes

- PUT
  Zum Aktualisieren eines bestimmten Objektes

- DELETE
  Zum Löschen eines bestimmten Objektes

Dies repräsentiert die übliche CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) Funktionalität, wie sie von dem REST-Paradigma vorgeschlagen wird.
Die in den HTTP-Request verwendeten URLs adressieren immer genau das für den HTTP-Request relevante Objekt. Dies entspricht ebenfalls dem REST-Paradigma.

6.3 Aktualisierung der Requestliste

Um die Requestliste eines Meetify-Client zu aktualisieren, muss dieser in regelmäßigen Abständen eine erneute Anfrage an das Backend senden.
Dabei muss die Abrufrate der Requestliste sehr hoch sein, da eine Änderung sofort bei allen Meetify-Clients angezeigt werden soll. Daraus ergibt sich ein in Abbildung 5 dargestellter Ablauf.

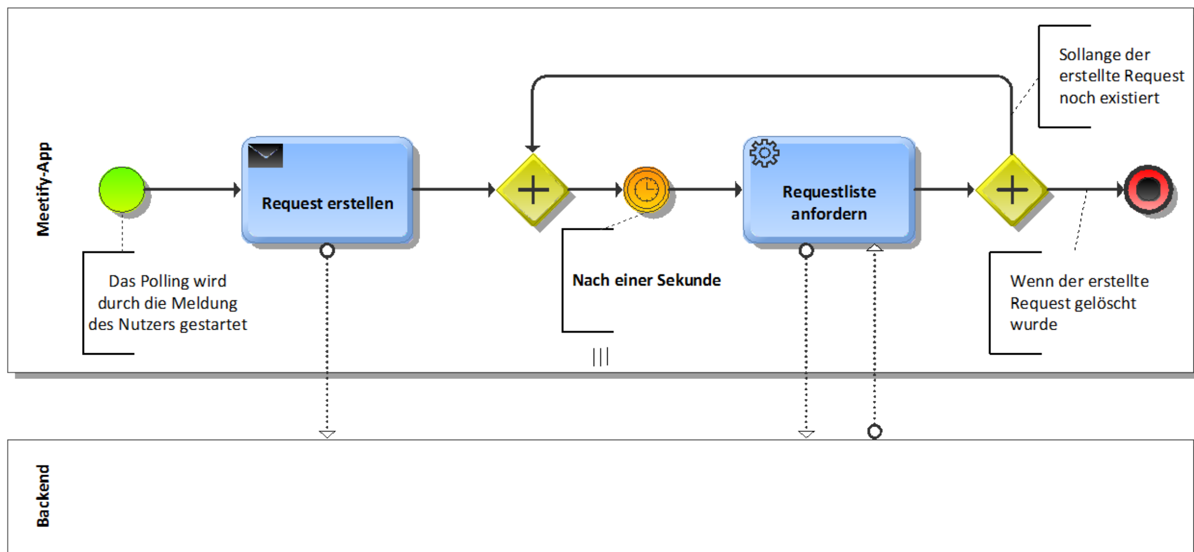

Abbildung 5: Pollingablauf von REST für einen Teilnehmer

Jeder Teilnehmer-Meetify-Client der einen Sprachwunsch äußert sendet einen entsprechenden Request an das Meetify-Backend. Nach dem eine Sekunde vergangen ist, muss der Meetify-Client die aktuelle Requestliste vom Meetify-Backend anfordern.
Dies wiederholt sich solange wie der erstellte Request, also der Sprachwunsch, existiert. Das gilt auch, wenn dem Sprachwunsch durch den Moderator stattgegeben wurde.
Erst wenn der der Request gelöscht wird, also der Meetify-Nutzer „das Wort" wieder an den Moderator abgibt, wird der Ablauf beendet.

Da somit jeder sich meldende oder aufgerufene Meetify-Client in jeder Sekunde den aktuellen Status abfragen muss, entsteht eine sehr hohe Netzwerklast.

Zusätzlich muss der Meetify-Client eines Moderators, nach dem Betreten eines Meetings, ebenfalls in jeder Sekunde die aktuelle Requestliste anfordern um neue oder gelöschte Sprachwünsche anzeigen zu können. Dies wird erst mit dem Verlassen des Meetings beendet.

Die grundlegende Natur von Meetify als Gruppendiskussions-Tool schränkt damit die effektive Nutzbarkeit durch die sehr hohe Netzlast stark ein. Diese hohe Netzlast tritt insofern auf, da bei einer Gruppendiskussion in der Regel alle, oder zumindest viele, Personen der Diskussion einen Ort und damit meist auch einen Netzwerkzugang nutzen.

7 Kommunikationsablauf MQTT

7.1 Kommunikation zwischen Meetify-Client und Backend

Der generelle Ablauf der Kommunikation hat sich im Vergleich zu vorherigen REST-Implementierung nicht verändert.

Wie schon beim REST-Meetify-Backend zuvor Antwortet das Meetify-MQTT-Backend nur wenn es eine Anfrage erhält (siehe Abb. 6).

Abbildung 6: Generelle Kommunikation

Auch hier stellt das Meetify-Backend in erster Linie wie zuvor den passiven und der Meetify-Client den aktiven Teil der Kommunikation dar.
Ausnahme bildet hier die Aktualisierung der Requestliste. Genauer beschrieben unter Punkt 7.3.

Tatsächlich findet etwas mehr Kommunikation statt. Der MQTT-Client des Meetify-Backends empfängt die Nachrichten im Gegensatz zur REST-Implementation nicht direkt.
Jede Nachricht, die an das Meetify-Backend gesendet wird, wird in erster Linie an den MQTT-Broker gesendet, welcher die Nachrichten dann entsprechend weiterleitet.
Da diese Kommunikation aber innerhalb des Meetify-Backends stattfindet, ist sie für diese extern betrachtete Kommunikation bedeutungslos.

Die zuvor benötigten URLs wurden als Topic-Pfade übernommen und zur eindeutigen Zuordbarkeit am Ende des Topic-Pfades durch die für jeden MQTT-Client im Meetify-Client eindeutige ID ergänzt.

Schickt ein MQTT-Client eine Nachricht an ein Topic zu dem er selbst subscribed ist, so erhält er seine eigene Nachricht. Da dies unerwünschter Netzwerkverkehr ist, sendet der Meetify-Client auf dem normalen Topic seine Anfrage, erwartet eine Antwort aber auf einem spezifischen Antwort-Topic, welcher ein Untertopic des Anfrage-Topics darstellt.
Vereinfachtes Beispiel:
Der MQTT-Client sendet seine Anfrage auf „meeting/abcd“ und erwartet die Antwort auf „meeting/abcd/res“.

7.2 Beschreibung der Datenpakete

Die Daten, welche mit einer Nachricht übermittelt werden, im MQTT-Kontext Payload genannt, liegen als einfacher Text vor.
Da ein einfaches Setzen und Abrufen von Key-Value-Parametern wie in den vorherigen HTTP-Nachrichten nicht funktioniert, wurde hierfür eine Konvention geschaffen, um in dem Payload dennoch alle Informationen unterzubringen und eindeutig zuordnen zu können:

- Zum Separieren wichtiger Elemente im Payload wird folgende festgelegte Zeichenkette genutzt: “; “ (ohne Anführungszeichen)
  In Worten: Semikolon Leerzeichen
  Folgend SEPERATOR genannt.

- Zum Separieren von Key und Value in Key-Value Paaren wird folgende festgelegte Zeichenkette genutzt:
  “ ~ “ (ohne Anführungszeichen)
  In Worten: Leerzeichen Tilde Leerzeichen
  Folgend GAP genannt.

- Eine von einem Meetify-Client gesendete Nachricht fängt immer mit der verwendeten Requestmethode an.
  Gefolgt von einem SEPERATOR.
  Vorhandene Requestmethoden sind: POST, GET, PUT, DELETE

- Eine vom Meetify-Backend gesendeten Nachricht beginnt immer mit dem definiertem Serverpattern:
  “server;“ (ohne Anführungsstriche)

- Nach der Requestmethode oder Serverpattern folgen alle zu übermittelnden Key-Value Paare. Ein Key-Value Paar wird mit einem SEPERATOR abgeschlossen.

- Tritt ein Key mehrfach auf, so wird dem Key eine laufende Nummer hinzugefügt, beginnend bei 0.

Beispiel für eine Anfrage von einem Meetify-Client:
Topic: „`user/2/1377041312464`“
Inhalt: „`DELETE; id ~ 2;`“

Antwort des Meetify-Backend auf die Anfrage:
Topic: „`user/2/1377041312464/res`“
Inhalt: „`server; success`”

In diesem Beispiel fordert der Meetify-Client die Löschung des Userobjektes mit der id = 2 an.
Das Meetify-Backend bestätigt die erfolgreiche Löschung.

Alle festgelegten Zeichenketten dieser Konvention können in den MQTT-Konstanten editiert werden, müssen aber zwischen MQTT-Client und MQTT-Broker abgeglichen werden.

Wie oben bereits angesprochen, benötigt eine Anfrage eine Requestmethode. Diese wurden aus der vorhergehenden REST-Implementierung übernommen.
Die Methoden sind POST, GET, PUT und DELETE

Sofern benötigt, wurden die Methoden wie folgt verwendet:

- POST
  Zum Erstellen eines neuen Objektes

- GET
  Zum Ermitteln eines bestehenden Objektes

- PUT
  Zum Aktualisieren eines bestimmten Objektes

- DELETE
  Zum Löschen eines bestimmten Objektes

Dies repräsentiert die übliche CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) Funktionalität.

Diese Definition gilt nur für die von Meetify generierten Nachrichten.
Zusätzlich werden noch weitere Nachrichten von dem MQTT-Clients an den MQTT-Broker zum Anmelden und Subscriben gesendet.
Diese Nachrichten sind durch das MQTT Protokoll vordefiniert und können in der MQTT-Protokoll-Spezifikation (International Business Machines Corporation (IBM)) eingesehen werden.

7.3 Aktualisierung der Requestliste

Die Voraussetzungen für die Aktualisierung der Requestliste sind die gleichen wie schon bei der REST-Implementierung.
Der Moderator-Meetify-Client und jeder Teilnehmer-Meetify-Client mit einem Sprachwunsch in einen Meeting muss zu jedem Zeitpunkt die aktuelle Liste der Sprachwünsche besitzen.

Anstatt wie zuvor jede Sekunde von jedem betroffenen Meetify-Client die aktuelle Requestliste anfordern zu lassen, wird nur der eigene Request abgesetzt und lediglich eine weitere Topic-Subscription vorgenommen (siehe App. 7).

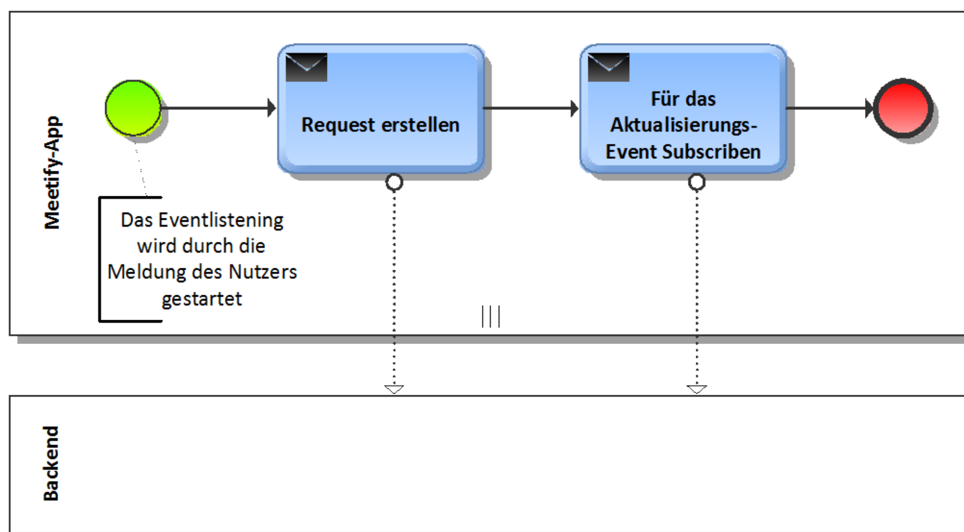

Abbildung 7: Event-Subscription

Der betroffene Moderator nimmt diese Topic-Subscription bereits beim Beitritt zum Meeting vor.

Diese zusätzliche Topic-Subscription gilt einem besonderen Topic des betreffenden Meetings. Dieses Topic wird Eventtopic genannt. Jedes Meeting besitzt ein solches Eventtopic als Untertopic, für die zurzeit aktuelle Requestliste.
An dieses Eventtopic sendet ausschließlich das Meetify-Backend Nachrichten.

Falls ein Request (Sprachwunsch eines Teilnehmers) eines Meetings erstellt, bearbeitet oder gelöscht wurde, sendet das Backend an das entsprechende Eventtopic des betroffenen Meetings die aktuelle und vollständige Requestliste (siehe Abb. 8).

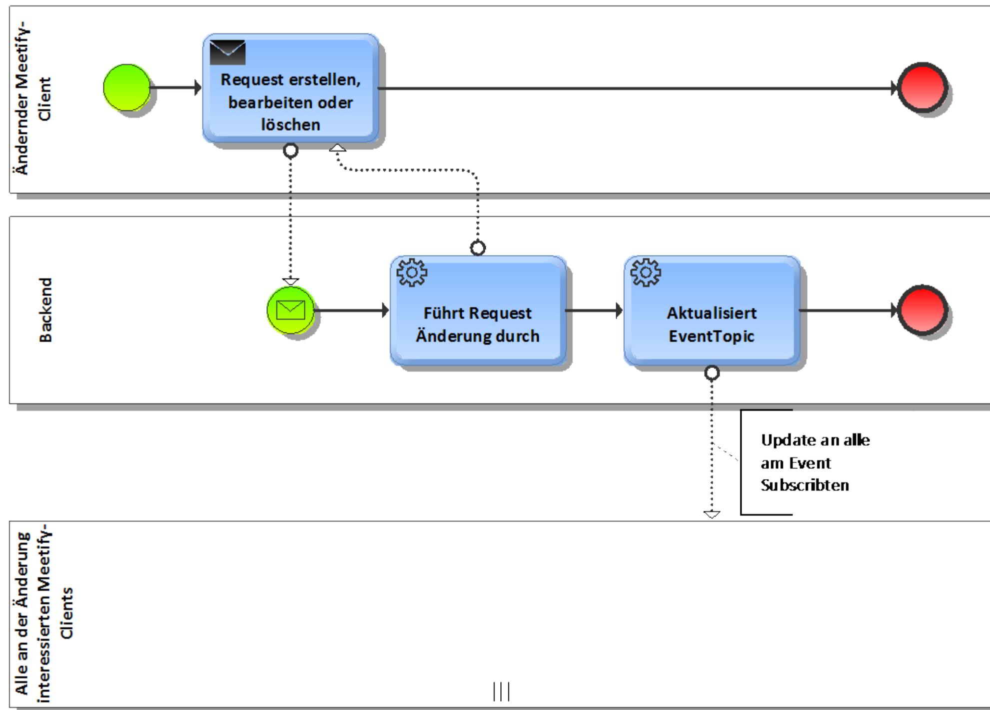

Abbildung 8: Auslösen des Events

So erhält jeder zu dem Eventtopic des Meetings subscribte Meetify-Client stets die aktuelle und vollständige Requestliste.
Die geschieht also immer nur genau dann, wenn es eine tatsächlich Änderung an der Liste gab und ohne das zusätzliche Anfragen nötig sind.

Hat ein Meetify-Client die aktualisierte Liste erhalten, kann dieser dann entsprechend auf die neue Requestliste reagieren, indem er entweder die neuen Daten aufbereitet und an die Oberfläche weiterleitet oder Subscription am Eventtopic wiederruft (siehe Abb. 9).

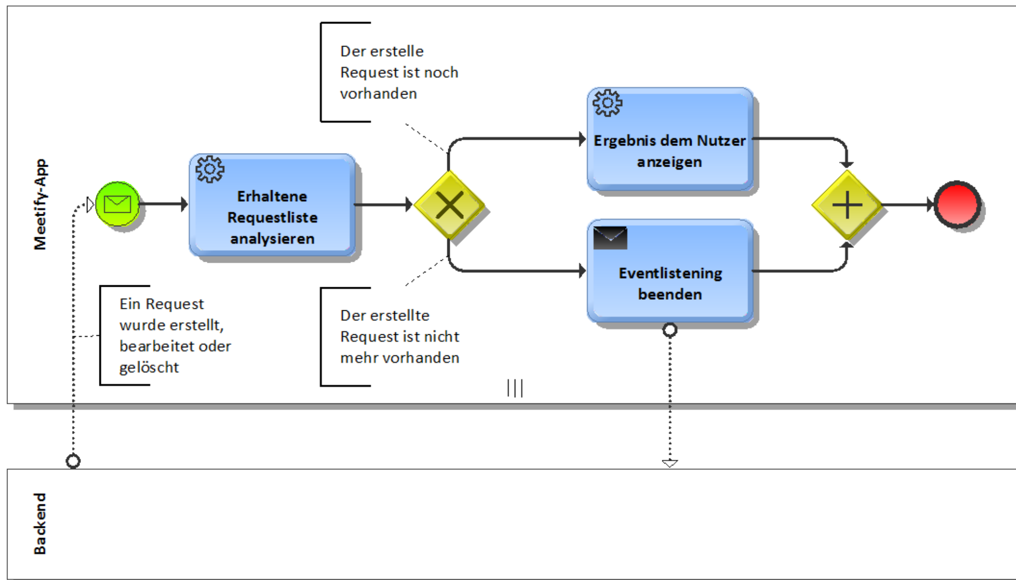

Abbildung 9: Abarbeiten des Events durch die Meetify-Clients

8 Netzlastanalyse anhand eines simulierten Meetings

8.1 Beschreibung und Ablauf des simulierten Meetings

8.1.1 Aufbau und Vorgaben für das Meeting

Dieses Meeting dient dem Ermitteln der Netzlast der alten REST-Implementierung, im Vergleich zur neuen MQTT-Implementierung.

Dazu wird ein Meeting mit 4 Meetify-Clients aufgebaut.
Der erste Meetify-Client (User1) übernimmt die Rolle des Moderators. Die restlichen drei Meetify-Clients treten dem Meeting als Teilnehmer bei (User2-4).

Das simulierte Meeting dauert genau 30 Minuten und wurde einmal für die REST-Implementierung und einmal für die MQTT-Implementierung durchgeführt.

Die Analyse der Netzlast wird aufgrund der während des Meetings übermittelten Pakete vorgenommen. Dazu wurde von jedem Meetify-Client die Anzahl von gesendeten und empfangenen Paketen gezählt, wodurch jeglicher für diese Analyse relevante Netzwerkverkehr abgedeckt wird.
Im Falle der MQTT-Implementierung betrifft das auch die Verwaltungspakete, welche nur zwischen MQTT-Broker und MQTT-Client übermittelt werden und nicht direkt das Meetify-Meeting betreffen.
Ein beispiel für diese Verwaltungspakete sind unter anderem Login-Pakete zum Verbinden des MQTT-Clients mit dem MQTT-Broker.

8.1.2 Zeitlicher Ablauf des Meetings

- Minute 0:
  Start der Simulation.
  Der Moderator (User1) loggt sich in Meetify ein und erstellt ein neues Meeting.

- Minute 1:
  Die drei restlichen Meetify-Clients (User2-4) melden sich als Teilnehmer am neu erstellten Meeting an.

- Minute 3:
  User2 sendet einen Sprachwunsch.

- Minute 5:
  User3 sendet einen Sprachwunsch.

- Minute 7:
  Der Moderator übergibt das Wort an User2.

- Minute 9:
  User4 sendet einen Sprachwunsch.

- Minute 11:
  User3 zieht seinen Sprachwunsch zurück.

- Minute 12:
  User2 gibt das Wort zurück an den Moderator.

- Minute 13:
  Der Moderator übergibt das Wort an User4.

- Minute 14:
  User2 sendet einen Sprachwunsch.

- Minute 16:
  User3 sendet einen Sprachwunsch.

- Minute 22:
  User4 gibt das Wort zurück an den Moderator.
  Der Sprachwunsch von User2 wird vom Moderator gelöscht.
  Der Moderator übergibt das Wort an User3.

- Minute 23:
  User2 sendet einen Sprachwunsch.

- Minute 25:
  User4 sendet einen Sprachwunsch.

- Minute 28:
  Der Moderator löscht den aktiven Sprachwunsch von User3
  und übernimmt das Wort.

- Minute 29:
  Alle Teilnehmer verlassen das Meeting.

- Minute 30:
  Das Meeting wird vom Moderator gelöscht und er meldet sich
  ab.

  Ende der Simulation.

8.2 Netzlastanalyse anhand der verschickten Pakete bei REST

Übersicht verschickter Pakete:

Meetify-Client	Verschickte Pakete	
	Meetify-Verwaltung	Polling
User1 (Moderator)	28	3415
User2 (Teilnehmer)	24	2763
User3 (Teilnehmer)	20	2148
User4 (Teilnehmer)	22	2052
Gesamt	94	10378

Abbildung 10: Übersicht über alle während des REST-Meetings versendeten und empfangenen Pakete

In der ersten Spalte der Tabelle 1 sind die verwendeten Meetify-Clients (User1-4) aufgelistet. In Klammern hinter dem Namen des Meetify-Clients steht die Rolle die der Client während des Meetings eingenommen hat.
Die zweite Spalte ist die Summe der von dem betreffenden Meetify-Client gesendeten und empfangenen Netzwerkpakete, die während des Meetings anfielen. Nicht mitgezählt wurden diejenigen Pakete, welche zum Aktualisieren der Requestliste gesendet und empfangen wurden.
Die dritte Spalte ist die Summe der Pakete, welche vom betreffenden Meetify-Client zum Aktualisieren der Requestliste gesendet oder empfangen wurden.

Für das gesamte Meeting wurden 10.472 Nachrichtenpakete versendet.

In der zweiten Spalte "Meetify-Verwaltung" der Tabelle ist zu sehen, dass die Anzahl der Pakete, die für tatsächliche Ereignisse (ein Nutzer tritt dem Meeting bei, ein Sprachwunsch wird geäußert, etc.) verwendet wurden, nicht sehr hoch. Keiner der verwendeten Meetify-Clients hat für den unter Punkt 8.1.2 beschriebenen Ablauf mehr als 30 Pakete benötigt.
Um jedoch bei einem aktiven Sprachwunsch die Requestliste aktuell zu halten hat jeder Meetify-Client während des Meetings, wie in der dritten Spalte "Polling" der Tabelle zu sehen, mehr als 2000 Pakete benötigt. Der Moderator-Client benötigte sogar weit über 3000 Pakete.

Dies bedeutet, dass lediglich ~0,9% der versendeten Nachrichten tatsächlich einem Ereignis des Meeting zuzuschreiben sind und ~99,1% keinen effektiven Nutzen hatten.

Wie bereits angedeutet muss die Aktualisierungsfrequenz der Requestliste so hoch sein, damit eine Änderung direkt an alle Teilnehmer des Meetings weitergeleitet wird. Jedoch ist aus der obigen Tabelle erkennbar, dass diese Strategie einen effizienten Einsatz von Meetify verhindert, da bei bereits vier beteiligten Personen ein massives Nachrichtenvolumen entsteht.

Die folgende Abbildung stellt noch einmal die Anzahl der Pakete dar, welche während des Meetings durch die Meetify-Clients versendet oder empfangen wurden.
Die Pakete wurden nach der jeweiligen Minute sortiert, in welcher sie versendet oder empfangen wurden. Dadurch ist eine direkte Zuordnung der Anzahl der versendeten und empfangenen Pakete zum unter Punkt 8.1.2 beschriebenen Meeting-Ablaufs möglich.

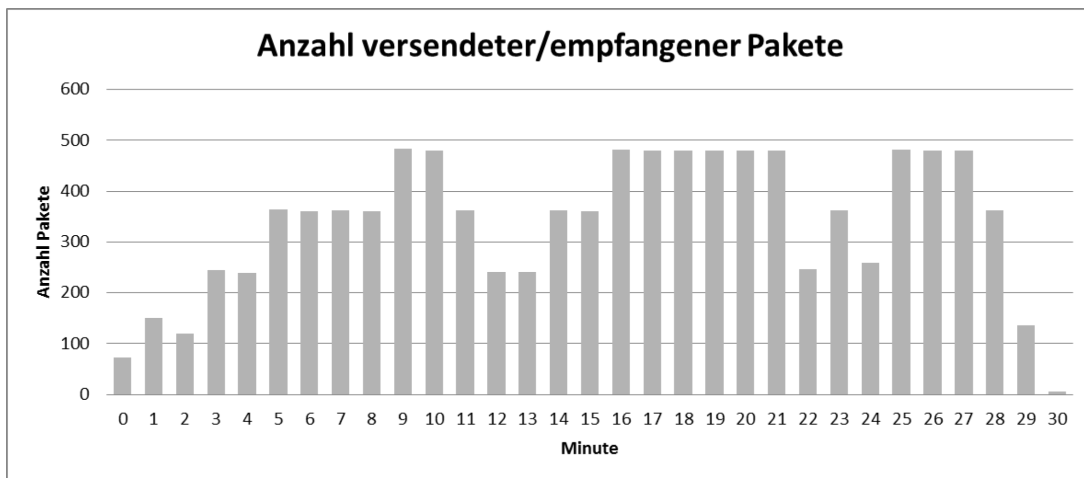

Abbildung 11: Versendete/Empfangene Pakete im REST-Meeting pro Minute

Vergleicht man die Anzahl der versendeten/empfangenen Pakete mit der Ablaufbeschreibung des Meetings so ist erkennbar, dass die Anzahl der in der betreffenden Minute versendeten Pakete direkt davon abhängig ist, wie viele der Zuhörer einen aktiven Sprachwunsch besitzen.

So wurden zum Beispiel von Minute 2 bis 3, in der noch kein Zuhörer einen Sprachwunsch geäußert hatte, etwas mehr als 100 Pakete versendet und empfangen, welche hauptsächlich vom Moderator stammen.
Von Minute 17 bis 18 wurden fast 500 Pakete versendet oder empfangen. Zu dieser Zeit hatten alle drei Teilnehmer einen aktiven Sprachwunsch.

8.3 Netzlastanalyse anhand der verschickten Pakete bei MQTT

Übersicht verschickter Pakete

Meetify-Client	Verschickte Pakete	
	Meetify-Payload	MQTT-Verwaltung
User1 (Moderator)	54	16
User2 (Teilnehmer)	48	22
User3 (Teilnehmer)	44	21
User4 (Teilnehmer)	38	22
Gesamt	184	81

Abbildung 12: Übersicht über alle während des MQTT-Meetings versendeten und empfangenen Pakete

In der ersten Spalte der Tabelle 1 sind die verwendeten Meetify-Clients (User1-4) aufgelistet. In Klammern hinter dem Namen des Meetify-Clients steht die Rolle, die der Client während des Meetings einnahm.
Die zweite Spalte ist die Summe der von dem betreffenden Meetify-Client gesendeten und empfangenen Netzwerkpakete, die während des Meetings anfielen. Im Gegensatz zur Übersicht des REST-Meetings sind hier auch die Pakete zum Aktualisieren der Requestliste enthalten.
Die dritte Spalte stellt die Summe der Pakete dar, welche vom betreffenden Meetify-Client zusätzlich zur Meetify-Funktionalität versendet oder empfangen wurden. Diese Pakete dienen der Verwaltung der MQTT-Kommunikation. Dies betrifft zum Beispiel das Einloggen in den MQTT-Broker oder das Subscriben zu einem Topic.

Für das gesamte Meeting wurden 265 Nachrichtenpakete versendet.

Die zusätzlichen Pakete, welche nur für die MQTT-Verwaltung verschickt oder empfangen wurden, machen einen Anteil von ~31% der insgesamt versendeten Nachrichten aus.

Die folgende Abbildung stellt noch einmal die Anzahl der Pakete dar, welche während des Meetings durch die Meetify-Clients versendet oder empfangen wurden.
Die Pakete wurden nach der jeweiligen Minute sortiert, in welcher sie versendet oder empfangen wurden. Dadurch ist eine direkte Zuordnung der Anzahl der versendeten und empfangenen Pakete zum unter Punkt 8.1.2 beschriebenen Meeting-Ablaufs möglich.

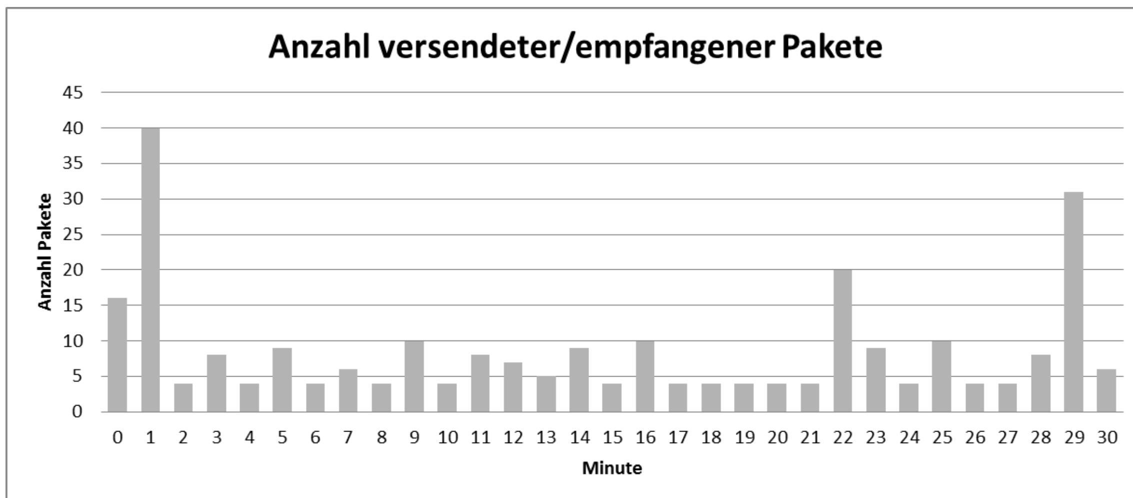

Abbildung 13: Versendete/Empfangene Pakete im MQTT-Meeting pro Minute

Vergleicht man die Anzahl der versendeten/empfangenen Pakete mit der Ablaufbeschreibung des Meetings so ist erkennbar, dass die Anzahl der in der betreffenden Minute versendeten Pakete, direkt von der Anzahl der Nutzerinteraktionen abhängen.

So wurden zum Beispiel von Minute 1 bis 2, wo sich die Zuhörer einloggten und dem Meeting beitraten, 40 Pakete versendet und empfangen.
Von Minute 17 bis 18 wurden lediglich 4 Pakete versendet und empfangen. Zu diesem Zeitpunkt hat laut Ablaufplan des Meetings keine Interaktion stattgefunden.
Dass dennoch 4 Pakete versendet wurden, liegt an einer Eigenschaft des MQTT-Clients. Dieser sendet alle 60 Sekunden (Keep-Alive-Intervall) ein „Ping"-Paket an den MQTT-Broker, um die Verbindung zum MQTT-Broker aktiv zu halten.

8.4 Vergleich der Netzlasten

Der einfache Vergleich der insgesamt übermittelten Pakete zeigt deutlich den Vorteil der neuen MQTT-Implementierung.
So mussten bei der MQTT-Implementierung lediglich 265 Pakete im Gegensatz zu 10.472 Paketen bei der REST-Implementierung versendet werden.

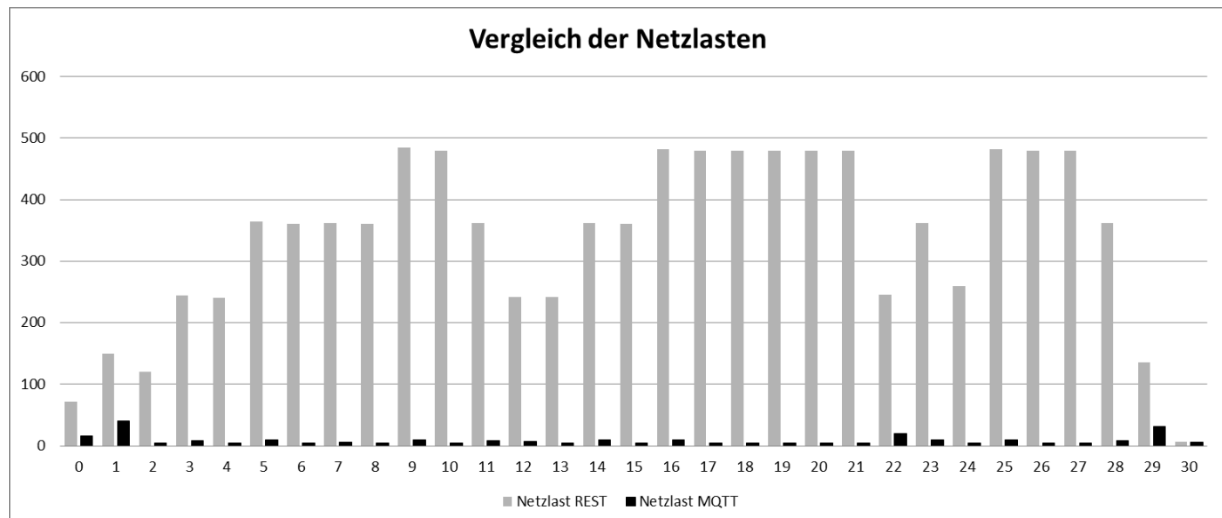

Abbildung 14: Vergleich der Netzlasten

Dies entspricht einer Verringerung der Netzlast, bezogen auf das durchgeführte Meeting, von ca. 92,46% durch die MQTT-Implementierung.

8.5 Netzlastabhängigkeit bei MQTT und REST

Sowohl bei der REST-Implementierung als auch bei der MQTT-Implementierung hängt die tatsächliche Anzahl an versendeten Paketen direkt von bestimmten Faktoren ab.

Im Falle der REST-Implementierung ist dies die Anzahl aktiver Sprachwünsche.
Sendet während eines Meetings kein Teilnehmer einen Sprachwunsch, so werden lediglich Pakete vom Moderator versendet. Dies sind in der Regel 120 Pakete pro Minute.
Für jeden Teilnehmer mit aktivem Sprachwunsch können ebenfalls 120 Pakete pro Minute hinzugerechnet werden.
Die eigentliche Verwaltung von Meetify fällt selbst wenn kein Sprachwunsch geäußert wurde nicht ins Gewicht.

Für die MQTT-Implementierung hängt die Netzlast direkt von den Aktionen der Nutzer ab.
Meldet sich ein Nutzer zu einem Meeting an oder von einem ab, wird in der Regel immer eine feste Anzahl von Paketen verschickt. Je nach Aktion sind das, inklusive Verwaltungspakete von MQTT, 6-13 Pakete. Dies ist unabhängig davon, ob bereits andere Nutzer angemeldet sind.

Für die Netzlast interessanter ist der Umgang mit Sprachwünschen. Also das Äußern, Aktivieren und Löschen von Sprachwünschen.
Bei diesen Aktionen hängt die Anzahl der versendeten Pakete direkt von der Anzahl der Meetify-Clients ab, welche zu dem Zeitpunkt der Aktion einen aktiven Sprachwunsch besitzen.
Dabei werden mindestens 4-5 Pakete versendet, wenn neben dem die Aktion ausführenden Teilnehmer kein weiterer Teilnehmer einen Sprachwunsch geäußert hat.
Für jeden weiteren Teilnehmer mit einem Sprachwunsch wird ein weiteres Paket verschickt.

Weiterhin in geringem Maße ausschlaggebend für die anfallende Netzlast sind die bereits angedeuteten Keep-Alive-Pakete (Pings). Von jedem am Meeting beteiligten MQTT-Client wird nach 60 Sekunden ein Keep-Alive-Paket versendet, um die Verbindung zum MQTT-Broker aufrecht zu erhalten. Dies gilt aber nur, wenn von dem betreffenden MQTT-Client innerhalb der 60 Sekunden kein anderes Paket an den MQTT-Broker gesendet wurde.

9 Verbesserungsmöglichkeiten

Für die MQTT-Implementierung gibt es Verbesserungen, welche noch nachträglich implementiert werden könnten.
Drei Verbesserungen wurden unter den folgenden Punkten kurz erläutert.

9.1 Verschlüsselung der Nachrichten

Die mittels MQTT übermittelten Nachrichten werden als klar lesbarer Text übermittelt. Es ist daher möglich, von außerhalb des Meetify-Clients oder des Meetify-Backends, den Inhalt der Nachrichten zu lesen oder zu verändern.
Lediglich das Passwort der Nutzer wird unkenntlich gemacht.
Der Mosquitto-Broker bietet die Möglichkeit ein SSL/TLS-Modul zu nutzen, wodurch die Nachrichtenübertragung sicherer gestaltet werden könnte.

9.2 Accountsystem mit Zugriffsrechten

Das Django-Framework, welches im alten Meetify-Backend für die Kommunikation und Datenhaltung zuständig war, bot die Möglichkeit eines Autorisierungssystems. Dadurch war es nicht möglich, dass zum Beispiel die Daten eines Meetings von jemand anderem als dem Ersteller und damit dem Besitzer des Meetings geändert wurden.
Die Meetify-Oberfläche verhindert derartiges Verhalten zwar, jedoch wäre es möglich, über eine manuell veränderte Nachricht derartige Eingriffe vorzunehmen.
Daher könnte die Einführung eines einfachen Autorisierungssystems ein weiterer Schritt für Meetify sein.

9.3 Verringerung der Netzlast durch Optimierung des Kommunikationsablaufes

Die neue MQTT-Implementierung basiert immer noch auf dem Kommunikationsprinzip der alten REST-Implementierung.
Dies bedeutet, dass auf jede Anfrage eines Meetify-Clients eine Antwort gesendet wird.
In bestimmten Fällen ist eine explizite Antwort vom Meetify-Backend jedoch überflüssig. Zum Beispiel beim Löschen des Nutzeraccounts reagiert der Meetify-Client gleich, unabhängig davon ob das Löschen ein Erfolg war oder nicht.
So könnten durch eine genaue Analyse der stattfindenden Kommunikationen und der entsprechenden Reaktionen des Meetify-Clients, die anfallenden Nachrichtenpakete noch weiter reduziert werden.

10 Zusammenfassung

Meetify ist eine Applikation für mobile Geräte. Sie hilft bei Gruppendiskussionen die Äußerung eines Sprachwunsches zu vereinfachen und diesen in der Reihenfolge ihres zeitlichen Auftretens stattzugeben. Dadurch ist innerhalb einer mit Meetify unterstützten Gruppendiskussion eine faire und übersichtliche Behandlung aller Sprachwünsche gewährleistet.
Dazu muss jeder Teilnehmer über ein mit Meetify ausgestattetes Gerät (Smartphone, Tablet oder vergleichbare Geräte) verfügen.

Für den nötigen Datenaustausch der Meetify-Geräte wurde das Client-Server-Prinzip umgesetzt.
So existiert in einem für die Meetify-Clients zugreifbaren Netzwerk, ein Server für das Meetify-Backend. Alle Meetify-Clients müssen sich zum Datenaustausch mit diesem Meetify-Backend verbinden.

Für die Verbindung zum Server und damit dem Meetify-Backend, wurde ursprünglich auf das HTTP-Protokoll, mittels einer Implementierung nach dem REST-Prinzip, zurückgegriffen.
Diese REST-Implementierung löst jedoch bereits bei wenigen Meetify-Clients eine sehr hohe Netzwerklast aus, wodurch Meetify nicht effektiv genutzt werden kann.

Im Rahmen dieser Arbeit wurde die Kommunikation der Meetify-Clients von dem auf REST-basierenden Ansatz auf einen MQTT und damit Event-basierenden Ansatz umgestellt.

Dazu wurde der in den Meetify-Clients eingesetzte HTTP-Client entfernt und durch einen MQTT-Client, samt Verwaltung, ersetzt.
Das alte Meetify-Backend basierte auf dem Django-Framework, welches ein reines HTTP-Framework darstellt.
Daher wurde ein neues Meetify-Backend entwickelt, welches MQTT-basierend ist.

Bei der neuen Implementierung vom Meetify-Client und dem Meetify-Backend, sind die Kommunikationsabläufe weitestgehend an die alte Implementierung angelehnt und wurde lediglich bei den Elementen, die eine hohe Netzwerklast erzeugen, auf den MQTT-Event-basierenden Ansatz umgestellt.

Durch die Umstellung von dem REST- und Polling-basierenden Ansatz auf den MQTT- und Event-basierenden Ansatz, konnte die entstehende Netzwerklast bei einer 30 Minuten dauernden simulierten Gruppendiskussion von 10.472 auf 265 versendete und empfangene Datenpakete verringert werden. Dies entspricht einer verringerten Netzwerklast von ca. 92,46%.

Durch die neue MQTT-Implementierung, kann Meetify nun effektiv zum Unterstützen von Gruppendiskussionen verwendet werden.

11 Literaturverzeichnis

Bayer, Thomas. 2002. oio.de. [Online] 27. 11 2002. [Zitat vom: 12. 10 2013.] http://www.oio.de/public/xml/rest-webservices.pdf.
Deutscher Django Verein e.V. http://django-de.org. [Online] [Zitat vom: 12. 10 2013.] http://django-de.org/ueber-django/.
Foundation, The Eclipse. wiki.eclipse.org. [Online] [Zitat vom: 13. 10 2013.] http://wiki.eclipse.org/Paho.
International Business Machines Corporation (IBM) . ibm.com. [Online] [Zitat vom: 13. 10 2013.] http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html.
Light, Roger. mosquitto.org. [Online] [Zitat vom: 13. 10 2013.] http://mosquitto.org/.
—. mosquitto.org. [Online] [Zitat vom: 13. 10 2013.] http://mosquitto.org/documentation/python/.
m2m.eclipse.org. eclipse.org. [Online] [Zitat vom: 13. 10 2013.] http://www.eclipse.org/paho/.
mqtt.org. mqtt.org. [Online] [Zitat vom: 12. 10 2013.] http://mqtt.org/.

12 Abbildungsverzeichnis

Abbildung 1: Systemaufbau REST....19
Abbildung 2: Datenmodell des MQTT-Backend....23
Abbildung 3: Systemaufbau MQTT....25
Abbildung 4: Generelle Kommunikation....26
Abbildung 5: Pollingablauf von REST für einen Teilnehmer....28
Abbildung 6: Generelle Kommunikation....30
Abbildung 7: Event-Subscription....33
Abbildung 8: Auslösen des Events....34
Abbildung 9: Abarbeiten des Events durch die Meetify-Clients....35
Abbildung 10: Übersicht über alle während des REST-Meetings versendeten und empfangenen Pakete....38
Abbildung 11: Versendete/Empfangene Pakete im REST-Meeting pro Minute....39
Abbildung 12: Übersicht über alle während des MQTT-Meetings versendeten und empfangenen Pakete....40
Abbildung 13: Versendete/Empfangene Pakete im MQTT-Meeting pro Minute....41
Abbildung 14: Vergleich der Netzlasten....42

13 Danksagung

Im Rahmen dieser Arbeit möchte ich mich bei einigen Personen für ihre Unterstützung während meiner Bearbeitungszeit der Bachelorarbeit bedanken.

– Bei Prof. Dipl.-Ing. Dipl.-Ing. Ulrich Lehmann für die kompetente Beratung und Unterstützung meiner Bachelorarbeit, sowie seiner schier grenzenlosen Motivation und guten Laune, welche ohne Zweifel abgefärbt haben.

– Bei M. Sc. Doreen Seider für ihre freundliche Aufnahme in das Institut für Simulations- und Softwaretechnik im DLR und ihre stets gute Betreuung.

– Bei allen Mitarbeitern der Abteilung Verteilte Systeme und Komponentensoftware für die freundliche Arbeitsumgebung und Hilfsbereitschaft. Besonders bei Jan Sippli und Volker Poddey, für die gute Betreuung bei jeglichen Hard- und Software-Problemen.

– Bei meiner Familie und meinen Freunden für die immerwährende Unterstützung, besonders in solch einer von Stress geplagten Zeit.

14 Eidesstattliche Erklärung

Ich versichere, die Bachelorarbeit „Aufbau einer Kommunikations-Infrastruktur auf Basis von MQTT“ selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Bad Fredeburg, den 21.10.2013

____

Lukas-Marius Biskoping

15 Anhang

15.1 Anhang A: Durchgeführte Testszenarien

15.1.1 Testumgebung:

Alle Tests wurden auf zwei verschiedenen Windows 7 64-bit Systemen aus der Entwicklungsumgebung Eclipse Juno heraus gestartet.
Verwendete Android-Geräte (falls im jeweiligen Test nicht anders beschrieben):

- 2-3x Android Api-17 Emulatoren
- 1x Lenovo ThinkPad

Die Emulatoren liefen zusammen mit dem Meetify-Backend auf dem gleichen Host.

Das Lenovo ThinkPad wurde über eine einfache WLAN-Infrastruktur mit dem Host des Meetify-Backends verbunden.

15.1.2 Durchgeführte Tests:

Die hier aufgeführte Reihenfolge der Tests ist zufällig.

Einfacher funktionaler Test über alle Funktionalitäten.

Grober Ablauf:

- Anmeldung mehrerer Quick-Login Nutzer
- Erstellen eines Meetings
- Beitritt der restlichen Nutzer
- Diverse Sprachwünsche, Rücknahmen, Neuverbindungen, etc. (Simulierter Meeting Ablauf)

(Wurde mehrfach mit Variationen wiederholt)

Aufgetretene Probleme (Reihenfolge zufällig):

- Quicklogin ohne Angabe eines Namens ist möglich.
  Lösung: Abfrage auf mindestens ein Zeichen im Namensfeld eingeführt.
- Beim Login bzw. beim Registrieren führten überschüssige Leerzeichen in Name, Email, etc. zu unerwartetem Verhalten.
  Lösung: Überschüssige voranstehende oder folgende Leerzeichen werden abgeschnitten.

- Beim Simulieren von Serverantworten wurden Parameter vergessen oder falsch geschrieben. Dies führte zu einem Fehler in der Client-App.
  Eine Lösung wurde bislang nicht implementiert, da dieser fehlerfall nur durch "Hacking" zu erreichen ist.
  Mögliche Lösung: Prüfung auf Existenz eines geforderten Parameters mit anschließender Fehlerbehandlung, entweder über einen Standartwert oder eine Fehlerausgabe.
- Sporadische Disconnects der Emulator MQTT-Clients.
  Dieses Problem trat bislang nie auf dem ThinkPad auf.
  Lösung: Der MQTT-Client verbindet sich nun automatisch neu, sobald er verwendet wird oder er zu einem Eventtopic subscribed ist.
- Reconnects verursachen Doppelverbindungsversuche, wenn während eines Reconnects der MQTT-Client verwendet wird. Der Nutzer erhält dadurch irreführende Fehlermeldungen.
  Lösung: Derartige Dopplungen werden nun abgefangen.
- Extreme Latenz beim Empfangen einer Antwort vom Meetify-Backend führte sporadisch zu doppelten Antworten, wodurch die aktuell laufende Abfrage des Meetify-Clients mit veralteten Daten beantwortet wurde.
  Lösung: Für jeden Abfragetyp existiert nur noch ein einziger Nachrichtenspeicher, in welchem nur die letzte Antwort gespeichert und nach dem Auslesen gelöscht wird.
- Verlust der Event-Subscribtion nach einem Reconnect.
  Lösung: Nach einem erfolgreichen Reconnect registriert sich der Meetify-Client automatisch für alle noch existierenden Events.
- Userpasswörter sind in Klartext lesbar in den Nachrichten enthalten.
  Lösung: Das Userpasswort wird nun mit einem Salt vermischt, via MD5 gehashed und in einen Hex-String umgewandelt. Dadurch werden die Passwörter als 32 Zeichen lange Hex-Strings innerhalb der Nachricht dargestellt. Dies ist keine Verschlüsselung, da das gecryptete Password ohne Veränderung vom Meetify-Backend gespeichert beziehungsweise verglichen wird. Es sorgt lediglich dafür, dass die Userpasswörter nicht durch einfaches Mitlesen der Nachrichten lesbar sind.
- Ein leer gelassener End-Termin eines Meetings führt zu fehlerhaftem Verhalten der App.
  Lösung: Ist kein End-Termin durch den Nutzer vorgegeben, wird automatisch ein End-Termin eine Stunde nach dem Start-Termin festgelegt.

- Sporadisch haben nicht alle emulierten Clients jedes auftretende Event "mitbekommen". Die Ursache bleibt ungeklärt, da das Verhalten nur in den ersten Tests auftauchte und seit der Behebung anderer Probleme nicht mehr aufgetreten ist.
  Nachträgliche Vermutung: Ein unerkannter Disconnect des Clients und damit Verlust der Event-Subscribtion.
  Lösung: Lösung anderer gelisteter Probleme.
- Das Aktualisieren der Requestliste führte zum unerwarteten Verbindungsverlust des MQTT-Clients im Meetify-Backend. Ursache ist der verwendete TCP-Socket. Dieser wird von dem Mosquitto-MQTT-Client selbst verwaltet und eine genaue Ursachenforschung war daher nur begrenzt möglich.
  Vermutung: Der TCP-Socket hat nicht genug Zeit die Nachrichten zu verarbeiten (Vermutung stammt aus den angezeigten Fehlermeldungen.).
  Lösung: Die Erhöhung der maximalen IDLE-Zeit beim Aufruf der loop-Funktion des MQTT-Clients .

Provozierter Absturz des MQTT-Brokers im Meetify-Backend

Grober Ablauf:
- Das Meetify-Backend wird initialisiert
- Mehrere Meetify-Clients verbinden sich
- Der Prozess des Mosquitto-MQTT-Brokers wird gestoppt
- Das Meetify-Backend startet den Prozess automatisch neu

Aufgetretene Probleme:
- Die Zugriffsschicht zur Datenbank gewährte keinen Zugriff mehr auf die bislang erstellten Objekte.
  Ursache: Erstellte Objekte können nur vom erstellenden Thread/Prozess aus genutzt werden.
  Lösung: Mit dem Neustarten des MQTT-Brokers wird die Datenbankschicht geschlossen und ebenfalls neu verbunden.
- Das automatische Neuverbinden des MQTT-Clients sorgte für doppelte Verbindungen.
  Lösung: Nach dem Neustart des Brokers wird der MQTT-Client nicht mehr durch die eingebaute Reconnect-Lösung, sondern durch das Meetify-Backend selbst neu verbunden.

Provoziertes "Aufhängen" des Backends

Grober Ablauf:
- Das Meetify-Backend wird initialisiert
- Der MQTT-Client des Meetify- Backends wird zum Disconnect gezwungen
- Das Meetify-Backend startet die notwendigen Dienste neu

Aufgetretene Probleme:
- Das Neustarten der blockierten Dienste erwies sich als sehr komplex (siehe: Provozierter Absturz des MQTT-Brokers im Meetify-Backend).
  Der MQTT-Client selbst sorgte dabei ebenfalls für Verbindungsprobleme.
  Lösung: Wenn beim MQTT-Broker oder dem MQTT-Client ein Problem mit der Verbindung festgestellt wird, werden die Datenbank, der MQTT-Client und der MQTT-Broker neugestartet.
  Das Meetify-Backend führt also einen automatischen Neustart durch.

Login bei einem nicht existierenden Host

Grober Ablauf:
- Das Meetify-Backend ist nicht aktiv
- Der Meetify-Client startet einen Verbindungsversuch zum Meetify-Backend

Aufgetretene Probleme:
- Der Client zeigt erwartungsgemäß eine Fehlermeldung an.
  Jedoch nicht mit der erwarteten Textinformation des Fehlers.
  Lösung: Abfangen der relevanten Fehlermeldung und manuelles Einfügen des erwarteten Fehlertextes.

15.2 Anhang B: Projektdateien

Siehe beigefügte CD.